Accelerating Multilinear Maps and Structured Sparse Tensor Kernels

Vivek Bharadwaj

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-146
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-146.html

August 4, 2025

Accelerating Multilinear Maps and Structured Sparse Tensor Kernels

by

Vivek Bharadwaj

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor James Demmel, Co-chair
Adjunct Associate Professor Aydın Buluç, Co-chair
Professor Katherine Yelick
Assistant Professor Michael Lindsey

Summer 2025

Accelerating Multilinear Maps and Structured Sparse Tensor Kernels

Vivek Bharadwaj

Abstract

Accelerating Multilinear Maps and Structured Sparse Tensor Kernels

by

Vivek Bharadwaj

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor James Demmel, Co-chair

Adjunct Associate Professor Aydın Buluç, Co-chair

Linear maps dominate machine learning and scientific computing workloads today. What about multilinear maps? Just as a linear map with one argument can be represented by a 2D matrix, a D-dimensional multilinear map is represented by a $(D + 1)$-dimensional tensor. We apply the map by flattening the tensor into a matrix and multiplying it by the Kronecker product of the inputs. When a batch of inputs is provided, this primitive is known as the Matricized-Tensor-Times-Khatri-Rao Product (MTTKRP). Efficient multilinear maps are essential in computational chemistry, multi-way data analysis, and signal processing. Unfortunately, they receive comparatively less interest from theorists and high-performance kernel designers.

We optimize the multilinear map in two applications, making contributions that span theory and practical implementation. We first examine equivariant graph neural networks, which use a structured sparse tensor to interact node features with edge features. In response, we design a GPU kernel generator that matches or exceeds the best closed-source implementations for the problem. Our package, OpenEquivariance, provides 5-6x end-to-end speedup for training quantum chemical foundation models. Our focus then shifts to Candecomp / PARAFAC decomposition, a higher-dimensional analogue of the matrix singular value decomposition. Here, we use randomized linear algebra to accelerate the MTTKRP in tall, overdetermined linear least-squares problems, scaling our work to thousands of CPU cores. The remaining chapters detour by adapting this randomized algorithm to sketch tensor trains, structures that originated in quantum mechanical computations. We also design communication-avoiding algorithms for a pair of kernels used in matrix completion and graph attention networks. Our work demonstrates that sustained attention to the multilinear map yields fruit across the computational stack.

Contents

List of Figures

List of Tables

Symbols

General

$\boldsymbol{A}^\top$	Transpose
$\det \boldsymbol{A}$	Determinant
$\boldsymbol{A}^{-1}$	Inverse
$\boldsymbol{A}^+$	Moore-Penrose pseudoinverse
Tr	Trace
$\mathcal{A}, \mathcal{B}, \mathcal{C}...$	Tensors with 3+ dimensions

Matrix Products

$\cdot$	Matrix-matrix multiplication
$\circledast$	Hadamard / elementwise product
$\otimes$	Kronecker product
$\odot$	(Columnwise) Khatri-Rao product

Inner Products and Norms

$\langle \cdot, ..., \cdot \rangle$	Generalized inner product
$\|\cdot\|_F$	Frobenius norm

Algorithmic Complexity

$O(...)$	Asymptotic upper bound
$\Omega(...)$	Asymptotic lower bound
$\Theta(...)$	Asymptotic tight bound
$\tilde{O}(...)$	Asymptotic upper bound omitting polylogarithmic factors

Contributions, Funding, and Disclaimers

This dissertation contains material adapted from the following conference papers, which were published by the author (V. Bharadwaj) in collaboration with others:

1. An Efficient Sparse Kernel Generator for O(3)-Equivariant Neural Networks [Bha+25].
2. Fast Exact Leverage Score Sampling from Khatri-Rao Products with Applications to Tensor Decomposition [Bha+23].
3. Distributed-Memory Randomized Algorithms for Sparse Tensor CP Decomposition [Bha+24a].
4. Efficient Leverage Score Sampling for Tensor Train Decomposition [Bha+24b].
5. Distributed-Memory Sparse Kernels for Machine Learning [BBD22].

V. Bharadwaj was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Department of Energy Computational Science Graduate Fellowship under Award Number DE-SC0022158. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility, under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP0024170. Funding sources for the remaining authors are printed in the original conference papers.

No generative artificial intelligence was used to prepare the text of this dissertation. The Github Copilot assistant was occasionally used to accelerate code development, but the vast majority of all software was written manually. This document was typeset with LaTeX (pdfTeX 3.141592653-2.6-1.40.24).

Acknowledgments

I am incredibly lucky that my PhD was both happy and productive. Chief credit belongs to my advisers Aydın Buluç and James Demmel, who spared neither attention nor expense year after year. Jim and Aydın gave me incredible latitude in the ideas I chose to pursue, supplemented by strong research suggestions as those ideas developed. I am still surprised that I was accepted to Berkeley given the deep bench of talent in high-performance computing, and I am very grateful.

Many contributors shaped the research contained in this dissertation. Russell Castro, Austin Glover, Laura Grigori, Riley Murray, Guillaume Rabusseau and Beheshteh Rakhshan were insightful collaborators, and I will fondly remember the time we spent working together. Special thanks goes out to my co-author and mentor Osman Asif Malik, who poured much time and attention into our shared projects.

I shared my graduate school experience with brilliant colleagues across the PASSION and BeBOP groups. Benjamin Brock, Giulia Guidi, and Alok Tripathy helped me from admissions, to prelims, to qualifying exams - thank you. It was also a pleasure to work with Roger Hsiao, Tianyu Liang, Yen-Hsiang Chang, and Gabriel Raulet, whom I wish very productive research careers. It was my fortune to interact with many talented members of the Department of Energy Computational Science Graduate Fellowship Program—Caleb Ju, in particular. I also shared many experiences with the Berkeley SLICE lab and Programming Systems group. Despite my tenuous affiliation with the latter, Sarah Chasins generously lent me her faculty perspective over many group social hours.

Michael Lindsey and Katherine Yelick were invaluable members of my dissertation committee; their attention, even in the face of so many other faculty commitments, was much appreciated. Federico Busato was a wonderful supervisor during my summer at NVIDIA. I am sure that CUDA and cuSPARSE are orders of magnitude better today thanks to his efforts.

I'm only in graduate school today because of the wonderful undergraduate education I received at Caltech. Chris Umans, Rose Yu, and Mikhail Shapiro were incredible faculty mentors, and I hope that I have the chance to pay forward every assistance they rendered me. I still remember Chris's CS21 class and Adam Sheffer's MA6a, both of which I took as a freshman. No lectures besides theirs have inspired such deep awe for discrete mathematics and theoretical computer science.

Hanwen Zhang, Anant Kale, Allison Wang, Emily Pan, and Daniel Mark taught me that an academic life was worth pursuing, even for a little while. Aditi Lahiri backed every decision I made through the latter half of graduate school—for this and so much of her support, I am incredibly grateful.

The last, and most important, acknowledgements go to my family: Shankar, Shyamala, and Vijay Bharadwaj, for a lifetime's worth of support.

Berkeley, California
August 2025

Chapter 1

Introduction: Multilinear Maps and Tensors

Linear maps form the backbone of today's machine learning and scientific computing models. Consider a linear function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$; any such map can be written in the form $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{A}\boldsymbol{x}$ for some matrix $\boldsymbol{A} \in \mathbb{R}^{m \times n}$. When a batch of inputs $\boldsymbol{X}$ is provided, $\boldsymbol{f}(\boldsymbol{X}) = \boldsymbol{A}\boldsymbol{X}$ relies on general matrix-matrix multiplication (GEMM), an extraordinarily-optimized numerical operation (called a *kernel*) on modern processors. Matrix multiplication has been studied at every level of the "computer science stack": theorists continuously drive down its complexity [Alm+25] or try to approximate its output [DKM06], algorithm designers study approaches to reduce its memory / communication traffic [Dem13], and silicon architects design exotic GEMM accelerators [Che+20b]. When either $\boldsymbol{A}$ or $\boldsymbol{X}$ is sparse, the range of both applications and computational optimizations expands substantially [Dav19].

Our work goes a step further by examining *multilinear maps*, multivariate functions that are linear when restricted to each coordinate [Lan87]. Consider a multilinear map $\boldsymbol{f} : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^k$; like the prior example, we can write $\boldsymbol{f}$ as

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{A}\,(\boldsymbol{x} \otimes \boldsymbol{y})$$

for some $\boldsymbol{A} \in \mathbb{R}^{k \times nm}$, where $\otimes$ denotes the Kronecker product (see Table 1.1). $\boldsymbol{A}$ encodes the map structure and is frequently interpreted as a higher-dimensional array $\mathcal{A} \in \mathbb{R}^{k \times n \times m}$, or *tensor*. In an analogous way, we can define multilinear functions that accept arbitrary argument counts and batched inputs $\boldsymbol{X}, \boldsymbol{Y}$. Computer scientists have long studied multilinear maps and ways to implement them efficiently [KB09], but the research landscape has evolved rapidly. *This dissertation exploits new application-specific tensor structures, advances in theoretical machinery, and novel hardware programming techniques to accelerate tensor kernels well beyond the state of the art.*

Our contributions range from new randomized algorithm development with best-in-class

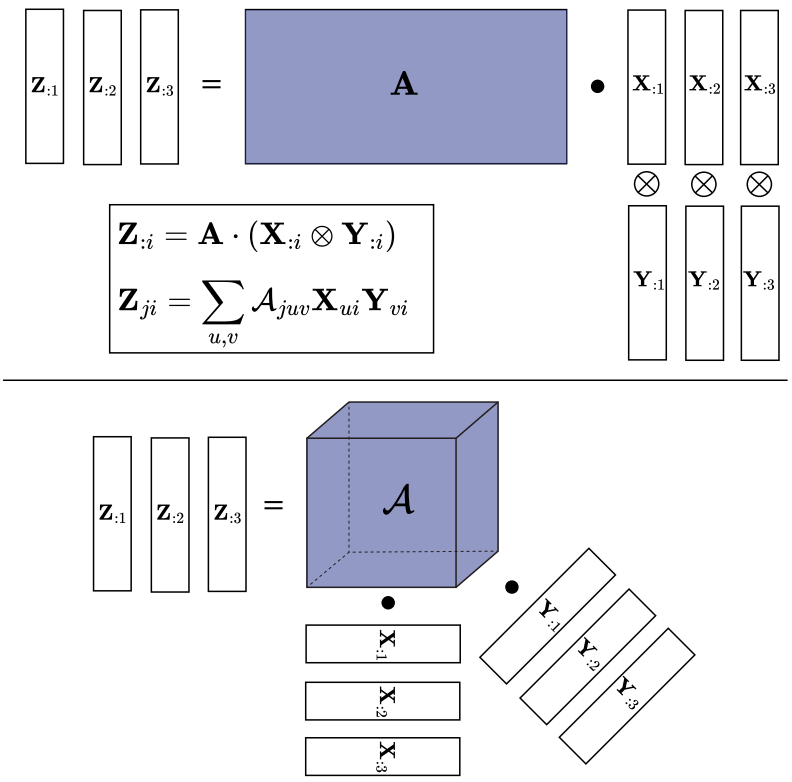


Figure 1.1: A batched multilinear map evaluation, illustrated as matrix multiplication against a column-wise Kronecker product (top) and as a tensor contraction (bottom). Equations for both formulations are boxed.

asymptotic guarantees to fine-grained accelerator kernel engineering. Some of our approaches take advantage of specific problem structure (e.g., patterns within the tensor), while others are more general.

1.1 Notation and the MTTKRP Kernel

This section defines some notation that recurs throughout the text. We use boldface lowercase characters for vectors ($\boldsymbol{u}, \boldsymbol{v}, ...$) and boldface uppercase characters for matrices ($\boldsymbol{A}, \boldsymbol{B}, ...$). We also use boldface characters to denote vector and matrix-valued functions, and all vector spaces in this work are over the field $\mathbb{R}$. A *tensor* is a multidimensional array that generalizes vectors and matrices to 3+ dimensions. We use script characters (e.g. $\mathcal{A}$, $\mathcal{B}$) to denote such objects. Matlab notation (e.g. $\boldsymbol{A}[i, :]$ or $\boldsymbol{A}_{i:}$) denotes a slice of a tensor across particular indices [Dem97]; the latter two examples refer to the i-th row $\boldsymbol{A}$. When slicing a matrix

Operation	Dim($\boldsymbol{X}$)	Dim($\boldsymbol{Y}$)	Dim($\boldsymbol{Z}$)	Definition
$\boldsymbol{Z} = \boldsymbol{X} \cdot \boldsymbol{Y}$	(m, k)	(k, n)	(m, n)	$\boldsymbol{Z}[i,j] = \sum_{a=1}^{k} \boldsymbol{X}[i,a]\,\boldsymbol{Y}[a,j]$
$\boldsymbol{Z} = \boldsymbol{X} \circledast \boldsymbol{Y}$	(m, n)	(m, n)	(m, n)	$\boldsymbol{Z}[i,j] = \boldsymbol{X}[i,j]\,\boldsymbol{Y}[i,j]$
$\boldsymbol{Z} = \boldsymbol{X} \otimes \boldsymbol{Y}$	(m_1, n_1)	(m_2, n_2)	$(m_2 m_1, n_2 n_1)$	$\boldsymbol{Z}[(i_2,i_1),(j_2,j_1)] = \boldsymbol{X}[i_1,j_1]\,\boldsymbol{Y}[i_2,j_2]$
$\boldsymbol{Z} = \boldsymbol{X} \odot \boldsymbol{Y}$	(m_1, n)	(m_2, n)	$(m_2 m_1, n)$	$\boldsymbol{Z}[(i_2,i_1),j] = \boldsymbol{X}[i_1,j]\,\boldsymbol{Y}[i_2,j]$

Table 1.1: Matrix product definitions.

$\boldsymbol{A} \in \mathbb{R}^{n \times m}$, we use the convention that $\boldsymbol{A}[i,:]$ is a $1 \times m$ row vector and $\boldsymbol{A}[:,i]$ is an $n \times 1$ column vector. With the notation fixed, a function $\boldsymbol{f} : \mathbb{R}^{i_1} \times ... \times \mathbb{R}^{i_N} \to \mathbb{R}^{i_{N+1}}$ is multilinear iff

$$\boldsymbol{f}(..., \alpha \boldsymbol{x}_k, ...) = \alpha \boldsymbol{f}(..., \boldsymbol{x}_k, ...)$$
$$\boldsymbol{f}(..., \boldsymbol{x}_k + \boldsymbol{\delta}, ...) = \boldsymbol{f}(..., \boldsymbol{x}_k, ...) + \boldsymbol{f}(..., \boldsymbol{\delta}, ...),$$

for any $\alpha \in \mathbb{R}$, index $1 \leq k \leq N$, and $\boldsymbol{\delta} \in \mathbb{R}^{i_k}$.

Table 1.1 lists four linear algebraic primitives that we use heavily: standard matrix multiplication, the Hadamard (or entrywise) product, the Kronecker Product, and the Khatri-Rao Product. We assume readers are familiar with standard multiplication ($\cdot$) and the elementwise product ($\circledast$) of two matrices, which are covered thoroughly in linear algebra textbooks [Dem97; GV13]. Some texts use $\odot$ to denote the Hadamard product, but we reserve the latter symbol for the Khatri-Rao product. All four operators are multilinear in their arguments.

The Kronecker product ($\otimes$) of $\boldsymbol{X} \in \mathbb{R}^{m_1 \times n_1}$ and $\boldsymbol{Y} \in \mathbb{R}^{m_2 \times n_2}$ produces $\boldsymbol{Z} \in \mathbb{R}^{m_1 m_2 \times n_1 n_2}$, where each entry of $\boldsymbol{Z}$ is a product of a uniquely-indexed pair of elements from $\boldsymbol{X}$ and $\boldsymbol{Y}$. The Kronecker product of column vectors $\boldsymbol{x} \in \mathbb{R}^{m_1}, \boldsymbol{y} \in \mathbb{R}^{m_2}$ produces $\boldsymbol{z} \in \mathbb{R}^{m_1 m_2}$, which can be reshaped to form the outer product matrix $\boldsymbol{x}\boldsymbol{y}^\top \in \mathbb{R}^{m_1 \times m_2}$.

A critical operation in this work is the (column-wise) Khatri-Rao product ($\odot$), a column-wise Kronecker product of its matrix arguments. For matrix $\boldsymbol{Z} = \boldsymbol{X} \odot \boldsymbol{Y}$ with n columns, we have

$$\boldsymbol{Z}[:,j] = \boldsymbol{X}[:,j] \otimes \boldsymbol{Y}[:,j], \quad 1 \leq j \leq n.$$

Now given a batch of inputs $\boldsymbol{x}_1, ..., \boldsymbol{x}_B$ and $\boldsymbol{y}_1, ..., \boldsymbol{y}_B$, we have a straightforward formula for the batch of outputs $\boldsymbol{z}_1, ..., \boldsymbol{z}_B$ from $\boldsymbol{f}$: putting $\boldsymbol{X} = [\boldsymbol{x}_1 \ ... \ \boldsymbol{x}_B]$ (and analogously for $\boldsymbol{Y}$ and $\boldsymbol{Z}$, we write

$$\boldsymbol{Z} = \boldsymbol{f}(\boldsymbol{X}, \boldsymbol{Y}) = \boldsymbol{A} \cdot (\boldsymbol{X} \odot \boldsymbol{Y}).$$

The operation above is known as the *Matricized-Tensor-Times-Khatri-Rao product* (MTTKRP). Just as matrix multiplication allows us to efficiently evaluate a large batch of linear maps, the MTTKRP allows us to efficiently evaluate a batch of multilinear maps. We can

easily extend this operation to arbitrary argument count:

$$\boldsymbol{Z} = \boldsymbol{f}(\boldsymbol{X}_1, \boldsymbol{X}_2, ..., \boldsymbol{X}_N) = \boldsymbol{A} \cdot (\boldsymbol{X}_1 \odot ... \odot \boldsymbol{X}_N)\,,$$

where $\boldsymbol{A}$ can be reshaped into an $(N+1)$-dimensional tensor $\mathcal{A}$. Figure 1.1 illustrates the kernel for $N = 2$ inputs.

1.2 Why Accelerate Batched Multilinear Maps?

Multilinear maps and the MTTKRP kernel may seem esoteric at first glance, and they receive less attention than matrix multiplication from the research and industry communities. Nonetheless, efficient kernels for multilinear maps command our interest for four key reasons:

1. **They are critical bottlenecks** in a variety of applications that span computational chemistry [Bat+22b], multidimensional data analysis [BK25] , and signal processing [YPP10].

2. **They are challenging to accelerate**, involving unwieldy matrix dimensions and intractably-large intermediate quantities. The tensor encoding the map structure has a worst-case parameter count that grows exponentially in the number of map arguments, quickly overflowing computer memory. Materializing the full Kronecker / Khatri-Rao products likewise requires exponential storage and high memory traffic; we avoid it whenever possible. Most of our work will focus on *sparse tensors* with additional structure. While such tensors are tractable to store and compute with, their sparsity results in drastically lower computational efficiency.

3. **They offer novel opportunities** to exploit structure that does not exist in the matrix case. For example, we show in Chapter 3, algorithms that do not confer special advantages for unstructured matrix multiplication become highly advantageous to approximate the output of the MTTKRP.

4. **They offer a springboard** to other areas of linear algebra and scientific computing research. In Chapter 5, we apply theoretical machinery developed for the MTTKRP to accelerate tensor structures that originated in quantum physics. In Chapter 6, we explore graph embedding techniques using sparse matrix kernels similar to Chapter 2.

1.3 Prior Work and New Opportunities

The Khatri-Rao product first appeared in 1968 in the context of functional analysis [KR68]. Its definition in the original text and subsequent works [GV13] is more general than what we use: there, the Kronecker product is performed across an arbitrary tiling of the input matrices,

not column by column. We restrict ourselves to the column-wise Khatri-Rao product, which has attractive spectral properties and finds applications across computational science and engineering [LT08].

The MTTKRP with a column-wise Khatri-Rao product finds its heaviest use in *tensor decomposition*, a generalization of the matrix singular value decomposition to higher-dimensional arrays. A survey by Kolda and Bader [KB09] along with a recent textbook from Ballard and Kolda [BK25] provide key details, and we explore this application in Chapters 3 and 4. Unsurprisingly, members of the tensor community have driven major progress in optimizing multilinear map computations and the MTTKRP. We highlight select contributions below, many of which build on analogous work for matrix multiplication.

1.3.1 Reducing Memory Traffic

Researchers in high performance computing have long known that many applications are limited by the rate at which they fetch data from a large pool of slow memory to a limited pool of fast memory [Don+03]. Modern processors can compute much more rapidly than data can be retrieved, and the gap between peak performance and memory bandwidth continues to widen [Dee+25]. Dense matrix multiplication, however, avoids the memory bottleneck through careful data reuse and tiling (see [Dem97], Chapter 2.6). Given a fast memory of capacity M, tiling enables the $O(n^3)$ algorithm for matrix multiplication to move only $O(n^3/\sqrt{M})$ data words from slow to fast memory, resulting in processor utilization close to the machine peak [Dem97]. The same tiling techniques extend to dense tensor MTTKRP and enable high performance [BKR18].

A fascinating line of theoretical work explores whether tiling strategies for matrix multiplication are optimal. In terms of asymptotic communication, Hong and Kung [HK81] answer this question affirmatively: for $O(n^3)$-matrix multiplication, there exists no scheduling of memory transactions and computation that moves *fewer* than $O(n^3/\sqrt{M})$ data words. Irony et al. [ITT04] reprove the same result using a geometric argument based on the Loomis-Whitney inequality, and Ballard et al. [BKR18] extend the result to show optimality of tiling strategies for the MTTKRP.

Data reuse and tiling are major components of the high performance MTTKRP kernels that we develop in Chapter 2 for chemical foundation models. We use heuristics loosely inspired by the lower bound proofs by Irony et al. [ITT04] to minimize traffic from GPU DRAM to the faster SRAM, and from the SRAM to thread registers.

1.3.2 Optimizations for Sparsity

Explicit dense tensors with large side lengths and high dimension rarely appear in practice. Consider, for example, a multilinear map that takes five vector arguments of dimension 100

each. The tensor $\mathcal{A}$ encoding the map would require 10^{12} data words if all entries were nonzero, a significant computational burden without the resources of a large compute cluster. Thankfully, many real-world tensors are sparse with only a small fraction of nonzero entries. For example, the Uber pickup tensor [Smi+17] has dimensions $183 \times 24 \times 1140 \times 1717$ and requires $\approx 10^{10}$ data words to store as a dense array, but it only contains 3.3×10^6 nonzero entries.

As with sparse matrices, sparse tensors are generally faster to compute with, but exhibit poorer data reuse and processor utilization. A major concern is the storage of the tensor itself. Sparse tensors may still occupy several gigabytes of disk space [Smi+17], while the order of nonzero storage controls the data access pattern to the dense input matrices. Multiple lines of work [SK15; Nis+19; Hel+21] achieve high performance using compressed, specialized storage formats for the sparse tensor.

Every MTTKRP computation we optimize in this work involves a sparse tensor. Excepting Chapter 6, we do not use storage formats more complicated than simple lists of nonzero coordinates. More complex, compressed storage schemes either introduce high overhead through memory indirections (Chapter 2) or fail to compose with our proposed algorithmic improvements (Chapters 3, 4, and 5). As an example, we explore randomized algorithms requiring random access to the sparse tensor, while formats such as Compressed Sparse Fiber [Smi+15; Nis+19] optimize for sequential iteration through nonzero elements.

1.3.3 Parallelization Strategies

The inputs to the multilinear maps we consider here are massive, so we rely on parallel computing to reduce runtime. MTTKRP kernels have been optimized for shared memory multiprocessors [Hay+18] and on clusters of nodes communicating through an interconnect [Smi+15; BKR18; BHR18]. In the latter three works, the bottleneck to distributed MTTKRP is data movement from processor to processor. Techniques such as data replication [ACS90; Joh93; Aga+95; SD11; Kwa+19] carry over from works that optimize distributed matrix multiplication, enabling algorithm designers to trade communication for higher per-processor memory usage. As with matrix multiplication, the same techniques that prove lower bounds for memory traffic also establish lower bounds for inter-processor communication in the distributed setting [BKR18].

Both lower bounds and optimal algorithms change, however, when we introduce communication imbalance with randomized methods (discussed further in 1.3.5). Randomized algorithms allow us to efficiently communicate a compressed form of the input arguments to a multilinear map, avoiding more expensive communication of the output. As well, Chapter 2 develops an execution strategy for a specific MTTKRP calculation that exploits the hierarchy of parallelism and caches available on GPUs.

1.3.4 Automatic Tensor Compilers / Execution Engines

Optimizing parallelism, sparsity, and communication quickly results in a vast design space for MTTKRP kernels. This creates a challenging environment for users, who must trade off programming efficiency for more complex, high-performance algorithms. To bridge these competing interests, projects including the Tensor Algebra COmpiler (TACO) [Kjo+17] can automatically generate a variety of performant tensor kernels for shared-memory CPUs and GPUs. Likewise, the Cyclops Tensor Framework [Sol+14], DISTAL [YAK22a], and SpDISTAL [YAK22b] are all capable of executing the MTTKRP on a cluster of interconnected processors given a lightweight user specification. More general domain specific languages - including Triton [TKC19] and TileLang [Wan+25] - enable users to quickly write efficient linear algebra code for GPUs, heavily targeting matrix multiplication.

While promising, these tools have limitations that prevent us from realizing their full potential. Chapters 3 and 4 require random accesses into a sparse tensor, while Chapter 6 communicates sparse matrix chunks repeatedly between processors. TACO, on the other hand, stores tensors in a specialized, opaque format, amortizing away the construction cost over repeated kernel calls. The format is antagonistic to both distributed-memory communication and random access slicing. Similarly, Triton enhances programmer productivity by automatically managing a small pool of fast GPU memory, but introduces overheads that discourage the warp-asynchronous programming model we adopt in Chapter 2. Sparsity produces yet more challenges for distributed tensor engines. For example, the sparsity-induced communication imbalances in Chapters 4 and Chapter 6 require a careful mathematical analysis to exploit, while communication of sparse tensors proves more difficult than their dense counterparts.

Kernel / operator fusion provides another opportunity for compilers, especially those that target machine learning models [Li+21]. Programs that combine back-to-back operations in a data pipeline can elide intermediate memory traffic and reduce runtime significantly. FlashAttention [Dao+22] serves as a recent high-profile example, fusing matrix multiplication with a subsequent softmax operation for transformer layers. A variety of compilers - XLA [Sab20], Halide [Rag+13], and TVM [Che+18] among them - apply some level of operator fusion to the computational graphs of deep learning models.

We argue that the kernel fusion strategies introduced in this dissertation would prove difficult for high-level machine learning compilers to discover automatically. In Chapter 2, we fuse an MTTKRP operation with graph convolution by persisting an SRAM buffer that is only written out when transitioning between rows of the graph adjacency matrix. This optimization relies on row-major ordering of the graph adjacency matrix, an external piece of problem-specific information that programmers must exploit manually. For the distributed-memory case, Chapter 6 employs kernel fusion within a distributed matrix multiplication algorithm, which interleaves computation with an intricate schedule of processor-to-processor data shifts.

1.3.5 Randomized Linear Algebra

From 2008 to 2010, Rokhlin and Tygert [RT08] followed by Avron et al. [AMT10] proved that randomized algorithms could accelerate matrix QR decomposition with minimal accuracy loss (up to numerical roundoff). The key ingredient is *linear sketching*, a lossy compression technique that preserves the column-space geometry of a target matrix. The linear algebra community has since poured effort into developing a variety of sketching matrices, each exhibiting a trade-off between runtime and accuracy [Woo+14]. Sketching finds broad use in approximate matrix factorization [HMT11; Che+25] and many related applications [Mur+23].

Researchers have only recently turned to sketching matrices with Khatri-Rao or Kronecker structure [PP13], [Dia+18], [Ahl+20], [FFG22]. Sketching these matrix products is difficult, as even explicit materialization would an consume inordinate amount of runtime and memory (typically defeating any benefits a sketch would offer). In Chapter 3, we encounter a linear least squares problem $\arg\min_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_F$ where the design matrix $\boldsymbol{A}$ is a Khatri-Rao product and the observation matrix $\boldsymbol{B}$ is sparse. Solving the problem involves an MTTKRP calculation, which we accelerate with a custom sampling algorithm that preserves the structure of both $\boldsymbol{A}$ and $\boldsymbol{B}$. Chapter 4 examines efficient implementation of this algorithm on a grid of communicating processors, while Chapter 5 adapts the algorithm to sketch *tensor trains*, a different class of tensor network.

1.4 Outline of this Dissertation

Having previewed our major contributions, we spend the first three chapters of this dissertation optimizing the MTTKRP using randomization, parallelization, and communication reduction. Two additional chapters detour by applying our methods to different tensor structures and exploring related applications. The material is organized as follows:

- In **Chapter 2**, we optimize the *Clebsch-Gordon (CG) tensor product*, a key kernel in rotation-equivariant graph neural networks, using a variety of high-performance GPU optimizations. The computation takes the form of an MTTKRP with a predefined, highly-structured tensor. Across multiple GPU models, our optimizations yield an order of magnitude speedup over the best open-source codes implementing the kernel, as well as 5-6x end-to-end runtime reduction for large chemistry foundation models.
- In **Chapter 3**, we devise a new sampling-based sketch to accelerate computation of $\arg\min_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_F$, where $\boldsymbol{A}$ is Khatri-Rao product of several matrices. The sketch accelerates an MTTKRP calculation required to solve each linear least-squares problem. We apply our algorithm to iteratively compute the Candecomp / PARAFAC (CP) decomposition of large sparse tensors, achieving best-in-class asympototic complexity for each linear least-squares solve required. In practice, our algorithm is 1.5-2.5x faster

than a state-of-the-art baseline sampler to achieve a fixed accuracy threshold for tensor decomposition. We also require (conservatively) 54x fewer samples on certain tensors compared to the baseline to achieve the same accuracy watermark.

- In **Chapter 4**, we create distributed-memory parallel formulations of the sketching algorithm in Chapter 3 and a simpler existing algorithm for the same problem. We optimize these parallel formulations to avoid processor to processor communication and demonstrate their effectiveness on thousands of CPU cores. To decompose the Reddit tensor [Smi+17] with billions of nonzeros on 512 CPU cores, we show an 11x runtime reduction over SPLATT to achieve more than 99% of the latter's accuracy.

- In **Chapter 5**, we use tools from Chapter 3 to efficiently construct subspace embeddings for data-sparse chains of 3D tensors. These chains—components of the *tensor train decomposition*—arise in contexts spanning numerical methods, quantum chemistry, and machine learning. We consider the case where a specific flattening of each tensor is an orthonormal matrix, which leads to an efficient sampling data structure. In addition to our complexity guarantees, we show that our sampler makes fast progress on sparse tensor train decomposition. Applying randomization to the linear least squares problems in sparse tensor train decomposition, we exhibit up to 26x speedup over a simple baseline code.

- In **Chapter 6**, we revisit graph neural networks and alternating tensor factorization applications from the prior chapters while studying the sampled-dense dense matrix multiplication (SDDMM) operation. Optimizing the kernel alongside its counterpart, sparse-dense matrix multiplication, requires the familiar techniques of theoretical communication analysis and kernel fusion. With these optimizations, our algorithms achieve 3-5x speedup over comparable methods in the PETSc package [Bal+21]. We benchmark strong scaling on real-world sparse matrices, demonstrating consistent runtime reductions up to 17,000 CPU cores.

- In **Chapter 7**, we summarize recent progress and remaining technical challenges unexplored by our work.

1.5 Additional Details

As a warm-up, we provide a straightforward proof of the elementary result that tensor-times-vector kernels can compute any multilinear map. We restrict ourselves to two inputs, with the generalization as a straightforward exercise.

Proposition 1.5.1 (Multilinear Maps and Tensor-Times-Vector Kernels)**.** *Any function* $\boldsymbol{f} : \mathbb{R}^m \times \mathbb{R}^n \to \mathbb{R}^k$ *of the form*

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{A} \cdot (\boldsymbol{x} \otimes \boldsymbol{y}).$$

is multilinear in its arguments. Conversely, any multilinar function of two arguments can be represented in the form above for an appropriate choice of $\boldsymbol{A} \in \mathbb{R}^{k \times mn}$.

Proof. The first statement is straightforward: for any scalar α,

$$\begin{aligned} \boldsymbol{f}(\alpha \boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{A}(\alpha \boldsymbol{x} \otimes \boldsymbol{y}) \\ &= \alpha \boldsymbol{A}(\boldsymbol{x} \otimes \boldsymbol{y}) \quad \text{(Multilinearity of } \otimes \text{ and } \cdot\text{)} \\ &= \alpha \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}), \end{aligned} \tag{1.1}$$

with an analogous proof when the scalar multiplies $\boldsymbol{y}$. Likewise for $\boldsymbol{u} \in \mathbb{R}^m$,

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x} + \boldsymbol{u}, \boldsymbol{y}) &= \boldsymbol{A}\left[(\boldsymbol{x} + \boldsymbol{u}) \otimes \boldsymbol{y}\right] \\ &= \boldsymbol{A}\left[\boldsymbol{x} \otimes \boldsymbol{y}\right] + \boldsymbol{A}\left[\boldsymbol{u} \otimes \boldsymbol{y}\right] \\ &= \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{f}(\boldsymbol{u}, \boldsymbol{y}). \end{aligned} \tag{1.2}$$

To prove the converse, let $\boldsymbol{e}_1, ..., \boldsymbol{e}_{\max(m,n)}$ be standard basis vectors and define

$$\boldsymbol{A}\left[:, (i,j)\right] = \boldsymbol{f}(\boldsymbol{e}_i, \boldsymbol{e}_j),\ 1 \le i \le m, 1 \le j \le n.$$

Then given any $\boldsymbol{x} = \sum_i \alpha_i \boldsymbol{e}_i$ and $\boldsymbol{y} = \sum_j \beta_j \boldsymbol{e}_j$, we have

$$\begin{aligned} \boldsymbol{f}(\boldsymbol{x}, \boldsymbol{y}) &= \boldsymbol{f}\left(\sum_i \alpha_i \boldsymbol{e}_i, \sum_j \beta_j \boldsymbol{e}_j\right) \\ &= \sum_i \sum_j \alpha_i \beta_j \boldsymbol{f}\left(\boldsymbol{e}_i, \boldsymbol{e}_j\right) && \text{(Multilinearity of } \boldsymbol{f}\text{)} \\ &= \sum_i \sum_j \alpha_i \beta_j \boldsymbol{A}\left[:, (i,j)\right] \\ &= \boldsymbol{A} \cdot (\boldsymbol{x} \otimes \boldsymbol{y}) && \text{(Definition of TTV Kernel).} \end{aligned} \tag{1.3}$$

Similarly, it is straightforward to show that the MTTKRP can represent any function that evaluates a batch of multilinear maps, since the results above hold independently for each column of the Khatri-Rao product. □

Chapter 2

Optimizing Tensor Products for O(3)-Equivariance

Our study of tensor kernels begins with the *Clebsch-Gordon (CG) tensor product*, an operation that originated in quantum mechanics and has been co-opted by the deep learning community [Tho+18; KLT18]. Neural network designers use the CG tensor product to build models constrained by physical symmetries, which have several practical benefits. This kernel implements a structured multilinear map, and we will exploit that structure to accelerate CG contraction. Our implementations enjoy over 10x speedup over e3nn, an open-source library, and on-par performance with cuEquivariance v0.4.0, a closed-source package from NVIDIA. We provide 5-6x end-to-end runtime reduction for the chemistry foundation models Nequip [Bat+22b] and MACE [Bat+24], both of which have since adopted our package as a

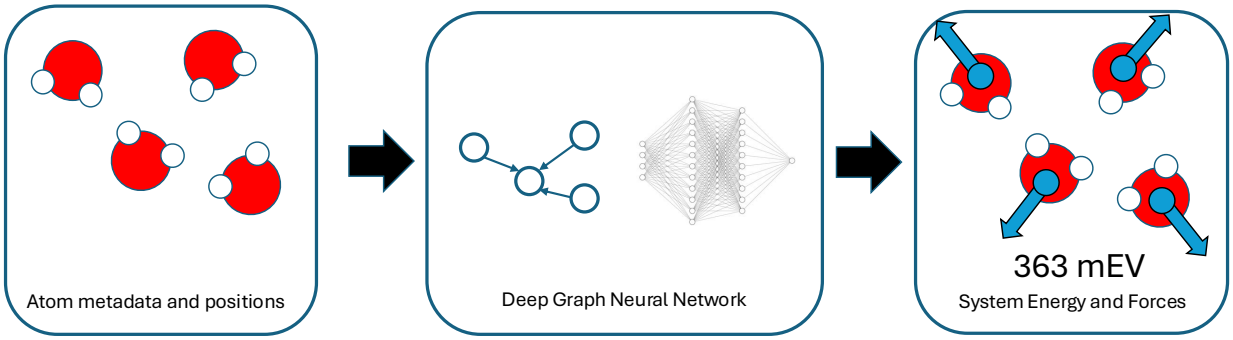


Figure 2.1: Pipeline for molecular property prediction. A configuration of atoms, typically encoded as a graph with associated node features, is fed to a graph neural network that estimates the potential energy and atomic forces.

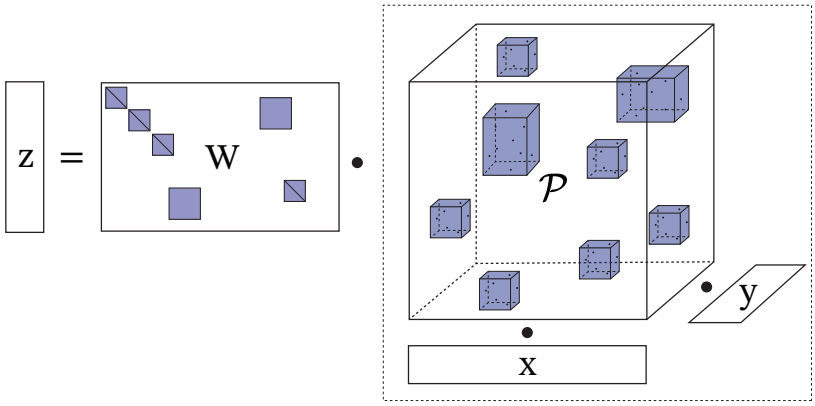


Figure 2.2: The CG tensor product, which contracts a block-sparse tensor $\mathcal{P}$ with two dense vectors. It is usually followed by multiplication with a structured weight matrix $\boldsymbol{W}$, and by convention,"CG tensor product" refers to the sequence of both operations. Each blue block is itself sparse (see Figure 2.4), and several blocks may share identical structure.

backend kernel provider.

Although we optimize a narrowly-defined tensor kernel here, the methods we deploy - maximizing instruction-level parallelism, minimizing global memory traffic, reducing synchronization overhead - are classic and general. Our kernel generator serves as a case study in exploiting specific tensor structure before we examine higher-level algorithmic challenges in Chapter 3.

2.1 Introduction

Equivariant deep neural network models have become immensely popular in computational chemistry over the past seven years [Tho+18; Wei+18; KLT18]. Consider a function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$. Informally, $\boldsymbol{f}$ is *invariant* if a class of transformations applied to its argument results in no change to the function output. A function is *equivariant* if a transformation applied to any input argument of $\boldsymbol{f}$ can be replaced by a compatible transformation on the output of $\boldsymbol{f}$. For example: a function predicting molecular energy based on atomic positions should not change its result if the atom coordinates are rotated, translated, or reflected (invariance). Likewise, a function predicting 3D forces on point masses should rotate its predictions if the input coordinate system rotates (equivariance). The latter property is termed *rotation equivariance*, and it is the focus of our work. Rotation equivariant neural architectures appear in the AlphaFold [Jum+21] version 2 model for protein structure prediction, the DiffDock [Cor+23] generative model for molecular docking, and the Gordon Bell finalist Allegro [Mus+23] for supercomputer-scale molecular dynamics simulation, among a host of other examples [SUG21;

AHK19; Bat+22b; Bat+24; Kok+24]. Figure 2.1 illustrates the input and objective for typical graph neural network chemistry models.

A core kernel in many (though not all) rotation equivariant neural networks is the Clebsch-Gordon (CG) tensor product, which combines two feature vectors in an equivariant model to produce a new vector [Tho+18]. This multilinear operation, illustrated in Figure 2.2, contracts a highly-structured block sparse tensor with a pair of dense input vectors, typically followed by multiplication with a structured weight matrix. It is frequently used to combine node and edge embeddings in equivariant graph neural networks, which are used for molecular energy prediction in computational chemistry [Bat+22b; Bat+22a; Mus+23] (see Figure 2.7). With its low arithmetic intensity and irregular computation pattern, the CG tensor product is difficult to implement efficiently in frameworks like PyTorch or JAX. Because the CG tensor product and its derivatives must be evaluated millions of times on large atomic datasets, they remain significant bottlenecks to scaling equivariant neural networks.

We introduce an open source kernel generator for the Clebsch-Gordon tensor product on both NVIDIA and AMD GPUs. Compared to the popular e3nn package [Gei+22] that is widely used in equivariant deep learning models, we offer up to one order of magnitude improvement for both forward and backward passes. Our kernels also exhibit up to 1.0-1.3x speedup over NVIDIA's closed-source cuEquivariance v0.4.0 package [Gei+24] on configurations used in graph neural networks, although our second derivative kernel is slower by 30% on certain inputs. Our key innovations include:

Exploiting ILP and Sparsity: Each nonzero block of $\mathcal{P}$ in Figure 2.2 is a structured sparse tensor (see Figure 2.4 for illustrations of the nonzero pattern). Popular existing codes [Gei+22] fill these blocks with explicit zeros and use optimized dense linear algebra primitives to execute the tensor product, performing unnecessary work in the process. By contrast, we use Just-in-Time (JIT) compilation to generate kernels that only perform work for nonzero tensor entries, achieving significantly higher throughput as block sparsity increases. While previous works [KT24; Kok24] use similar fine-grained approaches to optimize kernels for each block in isolation, we achieve high throughput by JIT-compiling a single kernel for the entire sparse tensor $\mathcal{P}$. By compiling kernels optimized for an entire sequence of nonzero blocks, we exhibit a degree of instruction level parallelism and data reuse that prior approaches do not provide.

Static Analysis and Warp Parallelism: The structure of the sparse tensor in Figure 2.2 is known completely at model compile-time and contains repeated blocks with identical nonzero patterns. We perform a static analysis on the block structure immediately after the equivariant model architecture is defined to generate a computation schedule that minimizes global memory traffic. We break the calculation into a series of subkernels (see Figure 2.5), each implemented by aggressively caching $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ in the GPU register file.

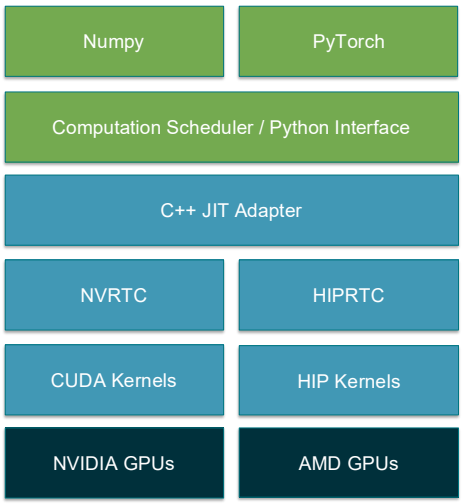


Figure 2.3: Software stack of our sparse kernel generator. Users request optimized kernels via a high-level Python specification, which is processed through a template engine. The generated kernel source code is fed to a C++ adapter that interfaces with vendor-specific JIT compilers. Users dispatch compiled kernels by interacting with Numpy or PyTorch.

We adopt a kernel design where each GPU warp operates on distinct pieces of coalesced data and requires no block-level synchronization (e.g. `__syncthreads()`). To accomplish this, each warp manages a unique portion of the shared memory pool and uses warp-level matrix multiplication primitives to multiply operands against nonzero blocks of the structured weight matrix.

Fused Graph Convolution: We demonstrate benefits far beyond reduced kernel launch overhead by fusing the CG tensor product with a graph convolution kernel (a very common pattern [Tho+18; Bat+22b; Bat+22a]). We embed the CG tensor product and its backward pass into two algorithms for sparse-dense matrix multiplication: a simple, flexible implementation using atomic operations and a faster deterministic version using a fixup buffer. As a consequence, our work is the first to reap significant model memory savings, a reduction in global memory writes, and data reuse at the L2 cache level.

Section 2.2 provides a brief introduction to equivariant neural networks, with Section 2.2.2 providing a precise definition and motivation for the CG tensor product. Section 2.3 details our strategy to generate efficient CG kernels and the design decisions that yield high performance. We validate those decisions in Section 2.4 on a range of benchmarks from chemical foundation models and other equivariant architectures.

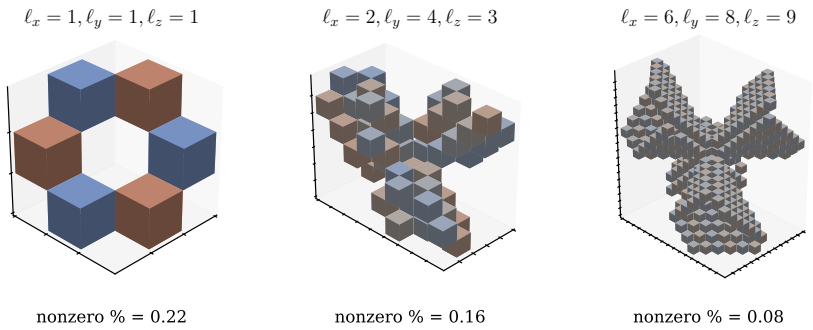


Figure 2.4: Three examples of coefficient tensors depicted by blue cubes in Figure 2.2. The fraction of zero entries increases with the tensor order, and the full CG tensor contains several copies of the blocks pictured here.

2.2 Preliminaries and Problem Description

Our notation and description of equivariance follow Thomas et al. [Tho+18] and Lim and Nelson [LN23]. Let G be an abstract group of transformations, and let $\boldsymbol{D}_{\text{in}} : G \to \mathbb{R}^{n\times n}, \boldsymbol{D}_{\text{out}} : G \to \mathbb{R}^{m\times m}$ be a pair of *representations*, group homomorphisms satisfying

$$\boldsymbol{D}_{\text{in}}(g_1 \cdot g_2) = \boldsymbol{D}_{\text{in}}(g_1) \cdot \boldsymbol{D}_{\text{in}}(g_2) \quad \forall g_1, g_2 \in G,$$

and likewise for $\boldsymbol{D}_{\text{out}}$. A function $\boldsymbol{f} : \mathbb{R}^n \to \mathbb{R}^m$ is equivariant with respect to $\boldsymbol{D}_{\text{in}}$ and $\boldsymbol{D}_{\text{out}}$ iff

$$\boldsymbol{f}(\boldsymbol{D}_{\text{in}}(g) \cdot \boldsymbol{v}) = \boldsymbol{D}_{\text{out}}(g) \cdot \boldsymbol{f}(\boldsymbol{v}) \quad \forall \boldsymbol{v} \in \mathbb{R}^n, g \in G.$$

A function is invariant if the equivariance property holds with $\boldsymbol{D}_{\text{out}}(g) = \boldsymbol{I}^{m\times m}$ for all $g \in G$.

In our case, $\boldsymbol{f}$ is a neural network composed of a sequence of layers, expressed as the function composition

$$\boldsymbol{f}(\boldsymbol{v}) = \boldsymbol{\phi}_N \circ ... \circ \boldsymbol{\phi}_1(\boldsymbol{v}).$$

Here, $\boldsymbol{D}_{\text{in}}$ and $\boldsymbol{D}_{\text{out}}$ are derived from the dataset, and the task is to fit $\boldsymbol{f}$ to a set of data points while maintaining equivariance to the chosen representations. Network designers accomplish this by imposing equivariance on each layer and exploiting a composition property [Tho+18]: if $\boldsymbol{\phi}_i$ is equivariant to input / output representations $(\boldsymbol{D}_i, \boldsymbol{D}_{i+1})$ and $\boldsymbol{\phi}_{i+1}$ is equivariant to $(\boldsymbol{D}_{i+1}, \boldsymbol{D}_{i+2})$, then $\boldsymbol{\phi}_{i+1} \circ \boldsymbol{\phi}_i$ is equivariant to $(\boldsymbol{D}_i, \boldsymbol{D}_{i+2})$. These intermediate representations are selected by the network designer to maximize predictive capability.

2.2.1 Representations of O(3)

In this paper, we let $G = O(3)$, the group of three-dimensional rotations including reflection. A key property of real representations of $O(3)$ is our ability to block-diagonalize them into a

canonical form [LCK24]. Formally, for any representation $\boldsymbol{D} : O(3) \to \mathbb{R}^{n\times n}$ and all $g \in G$, there exists a similarity matrix $\boldsymbol{P}$ and indices $i_1, ..., i_D$ satisfying

$$\boldsymbol{D}(g) = \boldsymbol{P}^{-1} \begin{bmatrix} \boldsymbol{D}^{(i_1)}(g) & & 0 \\ & \ddots & \\ 0 & & \boldsymbol{D}^{(i_D)}(g) \end{bmatrix} \boldsymbol{P}$$

where $\boldsymbol{D}^{(0)}(g), \boldsymbol{D}^{(1)}(g), ...$ are a family of elementary, *irreducible* representations known as the Wigner D-matrices. For all $i \geq 0$, we have $\boldsymbol{D}^{(i)}(g) \in \mathbb{R}^{(2i+1)\times(2i+1)}$. In the models we consider, all representations will be exactly block diagonal (i.e. $\boldsymbol{P}$ is the identity matrix), described by strings of the form

$$\boldsymbol{D}(g) \cong \texttt{"3x1e + 1x2o"}.$$

This notation indicates that D has three copies of $D^{(1)}$ along the diagonal followed by one copy of $D^{(2)}$. We refer to the term `"3x1e"` as an irrep (irreducible representation) with $\ell = 1$ and multiplicity 3. The suffix letters, "e" or "o", denote a parity used to enforce reflection equivariance, which is not relevant for us (we refer the reader to Thomas et al. [Tho+18] for a more complete explanation).

2.2.2 Core Computational Challenge

Let $\boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m$ be two vectors from some intermediate layer $\boldsymbol{\phi}$ of an equivariant deep neural network. For example, $\boldsymbol{x}$ could be the embedding associated with a node of a graph and $\boldsymbol{y}$ a feature vector for an edge (see Figure 2.7, bottom). We can view both vectors as functions $\boldsymbol{x}(\boldsymbol{v}), \boldsymbol{y}(\boldsymbol{v})$ of the network input $\boldsymbol{v}$, which are equivariant to $(\boldsymbol{D}_{\text{in}}, \boldsymbol{D}_x)$ and $(\boldsymbol{D}_{\text{in}}, \boldsymbol{D}_y)$ respectively. An equivariant graph convolution layer $\boldsymbol{\phi}$ interacts $\boldsymbol{x}$ and $\boldsymbol{y}$ to produce a new vector $\boldsymbol{z}$. To ensure layer-equivariance of ϕ, $\boldsymbol{z}(\boldsymbol{v})$ must be equivariant to $(\boldsymbol{D}_{\text{in}}, \boldsymbol{D}_z)$, where $\boldsymbol{D}_z$ is a new representation selected by the network designer.

The Kronecker product provides an expressive, general method to interact $\boldsymbol{x}$ and $\boldsymbol{y}$: if $\boldsymbol{x}(\boldsymbol{v})$ and $\boldsymbol{y}(\boldsymbol{v})$ are equivariant to the representations listed above, then $\boldsymbol{z}(\boldsymbol{v}) = \boldsymbol{x}(\boldsymbol{v}) \otimes \boldsymbol{y}(\boldsymbol{v})$ is equivariant to $(\boldsymbol{D}_{\text{in}}, \boldsymbol{D}_x \otimes \boldsymbol{D}_y)$. Unfortunately, $\boldsymbol{x} \otimes \boldsymbol{y} \in \mathbb{R}^{nm}$ may have intractable length, and we cannot drop arbitrary elements of the Kronecker product without compromising the equivariance property.

Let $\boldsymbol{P} \in \mathbb{R}^{nm\times nm}$ be the similarity transform diagonalizing $\boldsymbol{D}_x \otimes \boldsymbol{D}_y$. To reduce the dimension of the Kronecker product, we first form $\boldsymbol{P}(\boldsymbol{x}(\boldsymbol{v}) \otimes \boldsymbol{y}(\boldsymbol{v}))$, an equivariant function with block-diagonal output representation $\boldsymbol{P}(\boldsymbol{D}_x \otimes \boldsymbol{D}_y)\boldsymbol{P}^{-1}$. We can now safely remove segments of $\boldsymbol{P}(\boldsymbol{x}\otimes\boldsymbol{y})$ corresponding to unneeded higher-order Wigner blocks and recombine its components through a trainable, structured weight matrix. The result, $\boldsymbol{z}(\boldsymbol{v}) = \boldsymbol{W}\boldsymbol{P}(\boldsymbol{x}(\boldsymbol{v}) \otimes \boldsymbol{y}(\boldsymbol{v}))$, has a new output representation $\boldsymbol{D}_z$ and can be much shorter than $\boldsymbol{x} \otimes \boldsymbol{y}$.

When both $\boldsymbol{D}_x$ and $\boldsymbol{D}_y$ are representations in block-diagonal canonical form, the transform $\boldsymbol{P}$ is a highly structured block-sparse matrix containing nonzero *Clebsch-Gordon coefficients*. After potentially reducing $\boldsymbol{P}$ to k rows (removing segments corresponding to unneeded Wigner D-blocks), we can reshape it into a block-sparse tensor $\mathcal{P} \in \mathbb{R}^{m \times n \times k}$ contracted on two sides with $\boldsymbol{x}$ and $\boldsymbol{y}$. We call this operation (along with multiplication by a structured weight matrix $\boldsymbol{W} \in \mathbb{R}^{k \times k'}$) the **CG tensor product**, illustrated in Figure 2.2. It can be expressed by a matrix equation, a summation expression, multilinear tensor contraction (popular in the numerical linear algebra community), or Einstein notation:

$$\begin{aligned} \boldsymbol{z} &= \boxed{\mathrm{TP}(\mathcal{P}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})} \\ &:= \boldsymbol{W} \cdot \boldsymbol{P} \cdot (\boldsymbol{x} \otimes \boldsymbol{y}) \\ &:= \boldsymbol{W} \sum_{i=1,j=1}^{m,n} \boldsymbol{x}\,[i]\,\boldsymbol{y}\,[j]\,\mathcal{P}\,[ij:] \\ &:= \mathcal{P} \times_1 \boldsymbol{x} \times_2 \boldsymbol{y} \times_3 \boldsymbol{W} \\ &:= \mathrm{einsum}(\text{“}ijk, i, j, kk' \to k'\text{”}, \mathcal{P}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W}). \end{aligned} \tag{2.1}$$

Our goal is to accelerate computation of $\mathrm{TP}(\mathcal{P}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})$ for a variety of CG coefficient tensors $\mathcal{P}$. Given $\partial E/\partial \boldsymbol{z}$ for some scalar quantity E, we will also provide an efficient kernel to compute $\partial E/\partial \boldsymbol{x}$, $\partial E/\partial \boldsymbol{y}$, and $\partial E/\partial \boldsymbol{W}$ in a single pass. These gradients are required during both training and inference for some interatomic potential models.

2.2.3 Structure in the Sparse Tensor and Weights

Suppose $\boldsymbol{D}_x$, $\boldsymbol{D}_y$, and $\boldsymbol{D}_z$ each consist of a single Wigner block. In this case, the tensor $\mathcal{P}$ in Figure 2.2 has a single nonzero block of dimensions $(2\ell_x + 1) \times (2\ell_y + 1) \times (2\ell_z + 1)$. Figure 2.4 illustrates three blocks with varying parameters, which are small, (current models typically use $\ell_x, \ell_y, \ell_z \leq 4$), highly structured, and sparse.

Every nonzero block of the general tensor $\mathcal{P}$ in Figure 2.2 takes the form $\mathcal{P}^{(\ell_x, \ell_y, \ell_z)}$. Therefore, we could implement the CG tensor product by repeatedly calling the kernel in Figure 2.5A: a small tensor contraction followed by multiplication by a tile from the weight matrix $\boldsymbol{W}$. In practice, this is an inefficient strategy because $\mathcal{P}$ may contain hundreds of blocks with identical nonzero structures and values.

Instead, the CG tensor product splits into a sequence of subkernels [KT24] that match the structure in $\mathcal{P}$ and $\boldsymbol{W}$. We target two common patterns. First, the CG tensor product may interact b unique segments of $\boldsymbol{x}$ with a common segment of $\boldsymbol{y}$ using the same block from $\mathcal{P}$, followed by multiplication by a submatrix of weights from $\boldsymbol{W}$ rearranged along a diagonal. Figure 2.5B illustrates Kernel B as a contraction of a sparse tensor with a matrix $\boldsymbol{X}$ (containing the b rearranged segments from $\boldsymbol{x}$) and the common vector $\boldsymbol{y}$. It appears in the Nequip [Bat+22b] and MACE [Bat+22a] models. Kernel C is identical to kernel B,

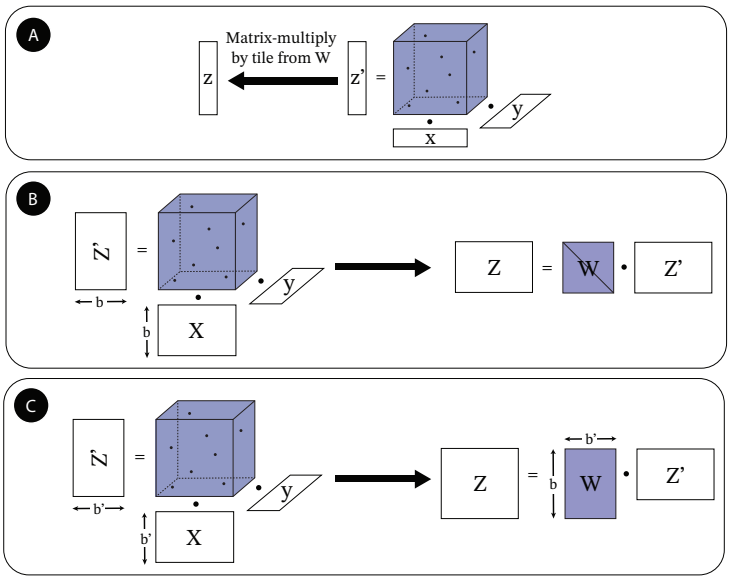


Figure 2.5: Fundamental subkernels that compose to implement the CG tensor product in Figure 2.2. In (A), $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ refer to segments of the longer vectors in Figure 2.2, and $\boldsymbol{W}$ contains entries from the larger weight matrix rearranged appropriately. In (B) and (C), $\boldsymbol{X}$ and $\boldsymbol{Z}$ refer to segments from $\boldsymbol{x}$ and $\boldsymbol{z}$ that have been reshaped into matrices to exploit repeating sparse tensor structure. b and b' are multiples of 32 in many models.

but arranges the weights in a dense matrix $\boldsymbol{W} \in \mathbb{R}^{b' \times b}$. Here, $\boldsymbol{Z}$ and $\boldsymbol{X}$ may have distinct row counts. The latter operation appears in DiffDock [Cor+23] and 3D shape classifiers by Thomas et al. [Tho+18]. Both operations can be extended to interact multiple segments from $\boldsymbol{y}$, but we could not find this pattern in existing equivariant models. While kernels besides B and C are possible, they rarely appear in practice.

2.2.4 A Full Problem Description

Armed with the prior exposition, we now give an example of a specific CG tensor product using the notation of the e3nn software package [Gei+22]. A specification of the tensor product consists of a sequence of subkernels and the representations that partition $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ into segments for those kernels to operate on:

$$\begin{aligned} \boldsymbol{D}_x &\cong \texttt{32x2e + 32x1e} \\ \boldsymbol{D}_y &\cong \texttt{1x3e + 1x1e} \\ \boldsymbol{D}_z &\cong \texttt{32x5e + 16x2e + 32x3e} \\ &[(1,1,1,\text{“}B\text{”}),(1,2,2,\text{“}C\text{”}),(1,2,3,\text{“}C\text{”})]\,. \end{aligned} \tag{2.2}$$

Here, $\boldsymbol{x}$ and $\boldsymbol{y}$ are partitioned into two segments, while $\boldsymbol{z}$ is partitioned into three segments. The list of tuples on the last line specifies the subkernels to execute and the segments they operate on. The first list entry specifies kernel B with the respective first segments of $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ ($b = 32$). Likewise, the second instruction executes kernel C with the first segment of $\boldsymbol{x}$ and the second segment of $\boldsymbol{y}$ ($b' = 32, b = 16$) to produce the second segment of $\boldsymbol{z}$.

2.2.5 Context and Related Work

Variants of rotation equivariant convolutional neural networks were first proposed by Weiler et al. [Wei+18]; Kondor et al. [KLT18]; and Thomas et al. [Tho+18]. Nequip [Bat+22b], Cormorant [AHK19], and Allegro [Mus+23] deploy equivariant graph neural networks to achieve state-of-the-art performance for molecular energy prediction. Other works have enhanced these message-passing architectures by adding higher-order equivariant features (e.g. MACE [Bat+22a; Bat+24] or ChargE3Net [Kok+24]). Equivariance has been integrated into transformer architectures [Fuc+20; LS23] with similar success.

Equivariant Deep Learning Software The e3nn package [GS22; Gei+22] allows users to construct CG interaction tensors, compute CG tensor products, explicitly form Wigner D-matrices, and evaluate spherical harmonic basis functions. PyTorch and JAX versions of the package are available. The e3x package [UM24] offers similar functionality. Allegro [Mus+23] modifies the message passing equivariant architecture in Nequip [Bat+22b] to drastically reduce the number of CG tensor products and lower inter-GPU communication.

The cuEquivariance package [Gei+24], released by NVIDIA concurrent to this project's development, offers the fastest implementation of the CG tensor product outside of our work. However, their kernels are closed-source and do not appear to exploit sparsity *within* each nonzero block of $\mathcal{P}$. Our kernel performance matches or exceeds cuEquivariance, often by substantial margins. Other relevant codes include GELib [KT24], Sphericart [Big+23], and Equitriton [LGM24b]. The former efficiently computes CG tensor products, while the latter two accelerate spherical harmonic polynomial evaluation.

Alternatives to CG Contraction The intense cost of the CG tensor product has fueled the search for cheap yet accurate algorithms. For example, Schütt et al. [SUG21] propose an equivariant message passing neural network that operates in Cartesian space, eliminating the need for CG tensor products. For SO(3)-equivariant graph convolution, Passaro and Zitnick [PZ23] align the node embeddings with spherical harmonic edge features before interacting the two. This innovation sparsifies the interaction tensor and asymptotically decreases the cost of each tensor product. We provide a full explication of their algorithm in Section 2.6. Luo et al. [LCK24] also produce asymptotic speedups by connecting the tensor product with a spherical harmonic product accelerated through the fast Fourier transform, an operation they call the *Gaunt tensor product*. Xie et al. [Xie+24] counter that the Gaunt tensor product

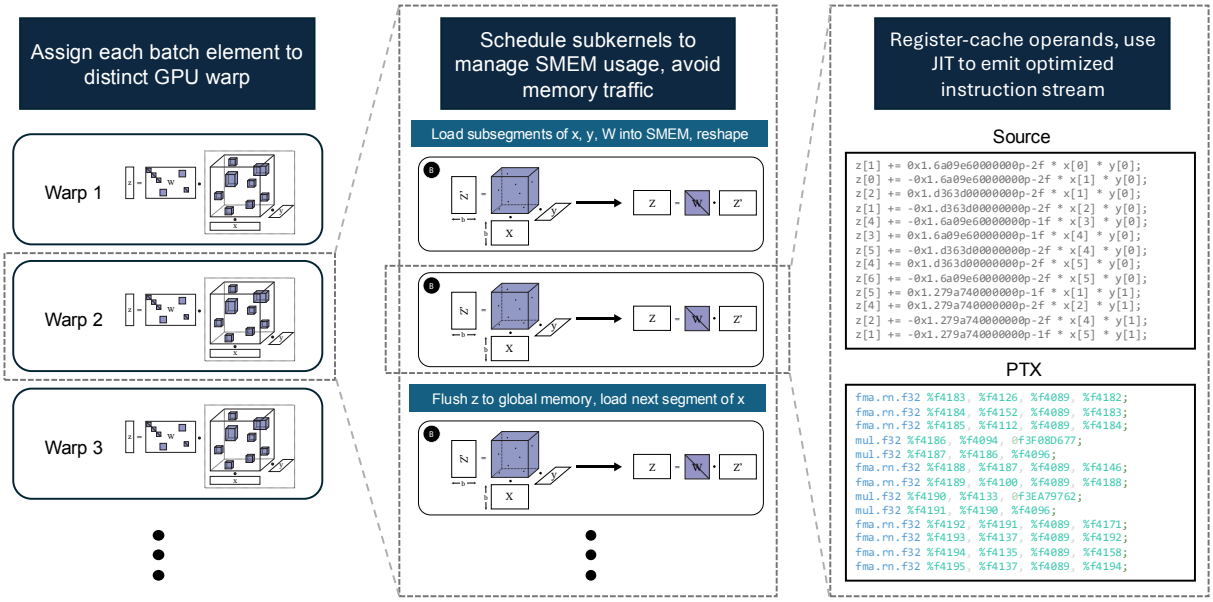


Figure 2.6: Design overview of our efficient CG kernel generator. Each element of a batch is dispatched to an independently-operating warp, which loads and evicts segments of its operands as it computes. These computations materialize as long sequences of arithmetic from unrolled loops, maximizing instruction-level parallelism.

does not produce results directly comparable to the CG tensor product, and that the former may sacrifice model expressivity.

GPU Architecture GPUs execute kernels by launching a large number of parallel threads running the same program, each accessing a small set of local registers. In a typical program, threads load data from global memory, perform computation with data in their registers, and store back results. Kernels are most efficient when groups of 32-64 threads (called "warps") execute the same instruction and perform memory transactions on contiguous, aligned segments of data. Warps are grouped into cooperative thread arrays (CTAs) that communicate through a fast, limited pool of shared memory. Within each CTA, individual warps may execute asynchronously relative to one another. Warps can synchronize at the CTA level, but the synchronization incurs overhead.

2.3 Engineering Efficient CG Kernels

Our task is to generate an efficient CG tensor product kernel given a problem specification outlined in Section 2.2.4. Algorithm 1 describes the logic of a kernel that operates on a large batch of inputs, each with a distinct set of weights (see Figure 2.7A). We assign each

$(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})$ input triple to a single GPU warp, a choice which has two consequences. First, it enables each warp to execute contiguous global memory reads / writes for $\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W}$ and $\boldsymbol{z}$. Second, it allows warps to execute in a completely asynchronous fashion without any CTA-level synchronization, boosting throughput significantly. The weights $\boldsymbol{W}$ are stored in a compressed form without the zero entries illustrated in Figure 2.2.

After the division of labor, each warp follows a standard GPU kernel workflow. The three inputs are staged in shared memory, the kernels in Equation (2.2) are executed sequentially, and each output $\boldsymbol{z}_b$ is stored back. Each warp operates on a unique partition of the CTA shared memory which may not be large enough to contain the the inputs and outputs. In the latter case, chunks of $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{W}$, and $\boldsymbol{z}$ are staged, and the computation executes in phases according to a schedule described in Section 2.3.1.

Algorithm 1 High-Level CGTP Algorithm

Require: Batch $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_B$, $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_B$, $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_B$

for $b = 1, \ldots, B$ **do**
  for $\text{seg}_i \in$ schedule **do**
    Load $\boldsymbol{x}_{\text{smem}} \leftarrow \boldsymbol{x}_b[\text{seg}_{i\ \text{(x start)}} : \text{seg}_{i\ \text{(x end)}}]$
    Load $\boldsymbol{y}_{\text{smem}}, \boldsymbol{W}_{\text{smem}}$ similarly
    Set $\boldsymbol{z}_{\text{smem}} \leftarrow 0$
    for all $\text{kern}_j \in \text{seg}_i$ **do**
      Set $\boldsymbol{X}_{\text{kern}}$ as a reshaped range of $\boldsymbol{x}_{\text{smem}}$
      Set $\boldsymbol{y}_{\text{kern}}, \boldsymbol{W}_{\text{kern}}$ similarly
      Compute $\boldsymbol{Z}_{\text{kern}} \leftarrow \text{kern}_j(\boldsymbol{X}_{\text{kern}}, \boldsymbol{y}_{\text{kern}}, \boldsymbol{W}_{\text{kern}})$
      Flatten and store $\boldsymbol{Z}_{\text{kern}}$ to a subrange of $\boldsymbol{z}_{\text{smem}}$
    Store $\boldsymbol{z}_b[\text{seg}_{i\ \text{(z start)}} : \text{seg}_{i\ \text{(z end)}}] \mathrel{+}= \boldsymbol{z}_{\text{smem}}$

We launch a number of CTAs that is a constant multiple of the GPU streaming multiprocessor count and assign 4-8 warps per CTA. The batch size for the CG tensor product can reach millions [Mus+23] for large geometric configurations, ensuring that all warps are busy. The computation required for each batch element can exceed 100K FLOPs for typical models [Mus+23; Bat+22b], ensuring that the threads within each warp are saturated with work.

2.3.1 Computation Scheduling

A key obstacle to efficient kernel implementation is the long length of the $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ feature vectors that must be cached in shared memory. The sum of their vector lengths for large MACE [Bat+24] and Nequip [Bat+22b] configurations can exceed 10,000 data words. Given that the warps in each CTA partition the shared memory, staging all three vectors at once (along with the weights in $\boldsymbol{W}$) is infeasible.

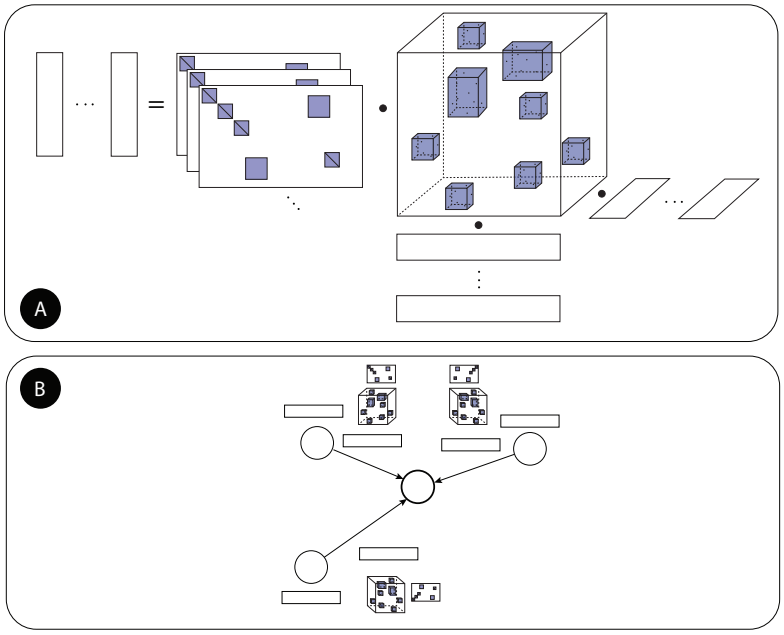


Figure 2.7: Applications of the CG tensor product. The simplest and most general use case (A) calls the kernel repeatedly with distinct $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{W}$ inputs. Interatomic potential models embed the operation in a graph convolution (B), where the tensor product combines node features with edge features.

To manage our limited shared memory, we execute the computation in phases that are scheduled at model compile-time. We break the list of block tensor contractions from Equation (2.2) into phases so that the sum of chunks from $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{W}$ and $\boldsymbol{z}$ in each phase fits into a warp's shared memory allotment. We then schedule loads and stores, hard-coding the relevant instructions into each kernel using our JIT capability. When more than a single computation phase is required, our goal is to generate a schedule that minimizes global memory reads and writes. We use a few simple heuristics:

1. If $\boldsymbol{x}$ and $\boldsymbol{y}$ can fit into a warp's shared memory partition (but not $\boldsymbol{z}$ and $\boldsymbol{W}$), then segments of $\boldsymbol{z}$ and $\boldsymbol{W}$ are streamed in through multiple phases of computation. In each phase, the kernels that touch each segment of $\boldsymbol{z}$ are executed.

2. Otherwise, we use a greedy algorithm. In each phase, the shared memory pool is filled with as many segments of $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{W}$ and $\boldsymbol{z}$ that can fit. Segments are flushed and reloaded as needed.

Case 1 covers most problem configurations in equivariant graph neural networks and minimizes

global memory writes, while Case 2 enables reasonable performance even with constrained shared memory resources. Large CG tensors (e.g. Nequip-benzene in Figure 2.8) may require 20-40 phases of computation per tensor product, and our scheduling substantially reduces global memory transactions.

2.3.2 JIT Subkernel Implementations

In preparation to execute a subkernel, suppose we have loaded $\boldsymbol{x}$, $\boldsymbol{y}$ and $\boldsymbol{W}$ into shared memory and reshaped subranges of all three to form $\boldsymbol{X}_{\text{kern}}, \boldsymbol{y}_{\text{kern}}$, and $\boldsymbol{W}_{\text{kern}}$. For the rest of Section 2.3.2 and all of Section 2.3.3, we omit the "kern" subscripts. Algorithm 2 gives the pseudocode to execute either kernel B or C from Figure 2.5 using these staged operands.

Algorithm 2 Subkernel B & C Warp-Level Algorithm

Require: $\boldsymbol{X} \in \mathbb{R}^{b' \times (2\ell_x+1)}$, $\boldsymbol{y} \in \mathbb{R}^{(2\ell_y+1)}$, $\boldsymbol{W} \in \mathbb{R}^{b \times b'}$
Require: Sparse tensor $\mathcal{P}^{(\ell_x,\ell_y,\ell_z)}$ for subkernel
for $t = 1, \ldots, b'$ **do**
  Load $\boldsymbol{x}_{\text{reg}} \leftarrow \boldsymbol{X}[t,:]$, $\boldsymbol{y}_{\text{reg}} \leftarrow \boldsymbol{y}$
  Initialize $\boldsymbol{z}_{\text{reg}} \in \mathbb{R}^{2\ell_z+1}$ to 0
  for $(i,j,k,v) \in \text{nz}(\mathcal{P})$ **do**
    $\boldsymbol{z}_{\text{reg}}[k] \mathrel{+}= v \cdot \boldsymbol{x}_{\text{reg}}[i] \cdot \boldsymbol{y}_{\text{reg}}[j]$
  if $\boldsymbol{W}$ is diagonal **then**
    $\boldsymbol{Z}[t,:] \mathrel{+}= \boldsymbol{W}[t,t] \cdot \boldsymbol{z}_{\text{reg}}$
  else
    Store $\boldsymbol{Z}'[t,:] \leftarrow \boldsymbol{z}_{\text{reg}}$
    Compute $\boldsymbol{Z} \mathrel{+}= \boldsymbol{W} \cdot \boldsymbol{Z}'$

Each thread stages a unique row row of $\boldsymbol{X}$ and $\boldsymbol{Z}$, as well as the entirety of $\boldsymbol{y}$, into its local registers. Models such as Nequip [Bat+22b] and MACE [Bat+24] satisfy $\ell_x, \ell_y, \ell_z \leq 4$, so the added register pressure from the operand caching is manageable. We then loop over all nonzero entries of the sparse tensor to execute the tensor contraction. Because the nonzero indices (i,j,k) and entries $\boldsymbol{v}$ of the sparse tensor $\mathcal{P}$ are known at compile-time, we emit the sequence of instructions in the inner loop explicitly using our JIT kernel generator. Finally, the output $\boldsymbol{Z}$ is accumulated to shared memory after multiplication by $\boldsymbol{W}$. When multiple subkernels execute in sequence, we allow values in $\boldsymbol{x}_{\text{reg}}$, $\boldsymbol{y}_{\text{reg}}$, and $\boldsymbol{z}_{\text{reg}}$ to persist if they are reused.

The matrix multiplication by the weights at the end of Algorithm 2 depends on the structure of $\boldsymbol{W}$. When $\boldsymbol{W}$ is square and diagonal (kernel B), multiplication proceeds asynchronously in parallel across all threads. When $\boldsymbol{W}$ is a general dense matrix, we temporarily store $\boldsymbol{z}_{\text{reg}}$ to shared memory and perform a warp-level matrix-multiplication across all threads.

Our kernel generator maximizes instruction-level parallelism, and the output kernels contain long streams of independent arithmetic operations. By contrast, common sparse tensor storage formats (coordinate [Hel+21], compressed-sparse fiber [SK15], etc.) require expensive memory indirections that reduce throughput. Because we compile a single kernel to handle all nonzero blocks of $\mathcal{P}$, we avoid expensive runtime branches and permit data reuse at the shared memory and register level. Such optimizations would be difficult to implement in a traditional statically-compiled library.

For typical applications, b and b' are multiples of 32. When b is greater than 32, the static analysis algorithm in Section 2.3.1 breaks the computation into multiple subkernels with $b \leq 32$, and likewise for b'.

2.3.3 Backward Pass

Like other kernels in physics informed deep learning models [Kar+21], the gradients of the CG tensor product are required during model *inference* as well as training for interatomic force prediction. Suppose $E(\boldsymbol{R}, \boldsymbol{W})$ is the scalar energy prediction emitted by our equivariant model for a configuration of s atoms, where $\boldsymbol{W}$ contains trainable model weights and each row of $\boldsymbol{R} \in \mathbb{R}^{s\times 3}$ is an atom coordinate. Then $\boldsymbol{F}_{\text{pr}} = -\partial E/\partial \boldsymbol{R} \in \mathbb{R}^{s\times 3}$ is the predicted force on each atom. Conveniently, we can compute these forces by auto-differentiating $E(\boldsymbol{R}, \boldsymbol{W})$ in a framework like PyTorch or JAX, but we require a kernel to compute the gradient of the CG tensor product inputs given the gradient of its output.

To implement the backward pass, suppose $\boldsymbol{z} = \text{TP}(\mathcal{P}, \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})$ and we have $\boldsymbol{g}_z = \partial E/\partial \boldsymbol{z}$. Because the CG tensor product is linear in its inputs, the product rule gives

$$\begin{aligned}
\partial E/\partial \boldsymbol{x}\,[i] &= \sum_{(i,j,k)\in \text{nz}(\mathcal{P})} \mathcal{P}\,[ijk] \cdot \boldsymbol{y}\,[j] \cdot \left(\boldsymbol{W}^\top \cdot \boldsymbol{g}_z\right)[k] \\
\partial E/\partial \boldsymbol{y}\,[j] &= \sum_{(i,j,k)\in \text{nz}(\mathcal{P})} \mathcal{P}\,[ijk] \cdot \boldsymbol{x}\,[i] \cdot \left(\boldsymbol{W}^\top \cdot \boldsymbol{g}_z\right)[k] \\
\partial E/\partial \boldsymbol{W}\,[kk'] &= \boldsymbol{g}_z\,[k'] \cdot \sum_{(i,j,k)\in \text{nz}(\mathcal{P})} \mathcal{P}\,[ijk]\, \boldsymbol{x}\,[i]\, \boldsymbol{y}\,[j]
\end{aligned}$$

Notice the similarity between the three equations above and Equation (2.1): all require summation over the nonzero indices of $\mathcal{P}$ and multiplying each value with a pair of elements from distinct vectors. Accordingly, we develop Algorithm 3 with similar structure to Algorithm 2 to compute all three gradients in a single kernel. For simplicity, we list the general case where the submatrix $\boldsymbol{W}$ is a general dense matrix (kernel C).

There are two new key features in Algorithm 3: first, we must perform a reduction over the warp for the gradient vector $\boldsymbol{g}_y$, since each thread calculates a contribution that must be summed. Second: when $\boldsymbol{W}$ is not diagonal, an additional warp-level matrix multiply

Algorithm 3 Subkernel C Warp-Level Backward

Require: $\boldsymbol{X} \in \mathbb{R}^{b' \times (2\ell_x+1)}$, $\boldsymbol{y} \in \mathbb{R}^{2\ell_y+1}$, $\boldsymbol{W} \in \mathbb{R}^{b \times b'}$
Require: $\boldsymbol{G}_Z \in \mathbb{R}^{b \times (2\ell_z+1)}$, sparse tensor $\mathcal{P}^{(\ell_x,\ell_y,\ell_z)}$
  Threads collaboratively compute $\boldsymbol{G}'_Z = \boldsymbol{W}^\top \cdot \boldsymbol{G}_Z$
  for $t = 1, \ldots, b'$ **do**
    Load $\boldsymbol{x}_{\text{reg}} \leftarrow \boldsymbol{X}[t,:]$, $\boldsymbol{y}_{\text{reg}} \leftarrow \boldsymbol{y}$, $\boldsymbol{g}'_{z\text{reg}} \leftarrow \boldsymbol{G}'_Z[t,:]$
    Initialize $\boldsymbol{g}_{\text{x reg}}, \boldsymbol{g}_{\text{y reg}}, \boldsymbol{g}_{\text{w reg}}, \boldsymbol{z}_{\text{reg}}$ to 0

    for $(i,j,k,v) \in \text{nz}(\mathcal{P}^{(\ell_x,\ell_y,\ell_z)})$ **do**
      $\boldsymbol{g}_{\text{x reg}}[i] \mathrel{+}= v \cdot \boldsymbol{y}_{\text{reg}}[j] \cdot \boldsymbol{g}'_{z\text{reg}}[k]$
      $\boldsymbol{g}_{\text{y reg}}[j] \mathrel{+}= v \cdot \boldsymbol{x}_{\text{reg}}[i] \cdot \boldsymbol{g}'_{z\text{reg}}[k]$
      $\boldsymbol{z}_{\text{reg}}[k] \mathrel{+}= v \cdot \boldsymbol{x}_{\text{reg}}[i] \cdot \boldsymbol{y}_{\text{reg}}[j]$

    Store $\boldsymbol{g}_y \leftarrow$ warp-reduce$(\boldsymbol{g}_{\text{y reg}})$
    Store $\boldsymbol{G}_x[t,:] \leftarrow \boldsymbol{g}_{\text{x reg}}$ and $\boldsymbol{Z}'[t,:] \leftarrow \boldsymbol{z}_{\text{reg}}$
  Threads collaboratively compute $\boldsymbol{G}_W = \boldsymbol{G}_Z \cdot (\boldsymbol{Z}')^\top$

is required at the end of the algorithm to calculate $\boldsymbol{G}_W$. We embed Algorithm 3 into a high-level procedure akin to Algorithm 1 to complete the backward pass.

2.3.4 Higher Partial Derivatives

For interatomic potential models, we require higher-order derivatives to optimize force predictions during training [Kar+21], as we explain below. Rather than write new kernels for these derivatives, we provide a novel (to the best of our knowledge) calculation that implements them using the existing forward and backward pass kernels.

As in Section 2.3.3, let $\boldsymbol{F}_{\text{pr}} = -\partial E/\partial \boldsymbol{R} \in \mathbb{R}^{s\times 3}$ be the predicted atomic forces generated by our model. During training, we must minimize a loss function of the form

$$\min_{\boldsymbol{W}} \mathcal{L}(\boldsymbol{R}, \boldsymbol{W}) = \min_{\boldsymbol{W}} \|\boldsymbol{F}_{\text{pr}}(\boldsymbol{R}, \boldsymbol{W}) - \boldsymbol{F}_{\text{gt}}(\boldsymbol{R})\|_F^2$$

where $\boldsymbol{F}_{\text{gt}}(\boldsymbol{R}) \in \mathbb{R}^{s\times 3}$ is a set of ground-truth forces created from a more expensive simulation. The loss function may include other terms, but only the Frobenius norm of the force difference is relevant here. We use a gradient method to perform the minimization and calculate

$$\begin{aligned} \frac{\partial \mathcal{L}}{\partial \boldsymbol{W}} &= 2 \cdot \text{vec}(\boldsymbol{F}_{\text{pr}}(\boldsymbol{R}, \boldsymbol{W}) - \boldsymbol{F}_{\text{gt}}(\boldsymbol{R}))^\top \frac{\partial \boldsymbol{F}_{\text{pr}}}{\partial \boldsymbol{W}} \\ &= -2 \cdot \text{vec}(\boldsymbol{F}_{\text{pr}}(\boldsymbol{R}, \boldsymbol{W}) - \boldsymbol{F}_{\text{gt}}(\boldsymbol{R}))^\top \frac{\partial^2 E}{\partial \boldsymbol{R} \partial \boldsymbol{W}} \end{aligned} \tag{2.3}$$

where "vec" flattens its matrix argument into a vector and $\partial^2 E/(\partial \boldsymbol{R} \partial \boldsymbol{W}) \in \mathbb{R}^{3s \times (\#\text{ weights})}$ is a matrix of second partial derivatives. Equation (2.3) can also be computed by auto-

differentiation, but the second partial derivative requires us to register an autograd formula for our CG tensor product *backward* kernel (i.e. we must provide a "double-backward" implementation).

To avoid spiraling engineering complexity, we will implement the double-backward pass by linearly combining the outputs from existing kernels. Let $\boldsymbol{z} = \mathrm{TP}(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})$ (we omit the sparse tensor argument $\mathcal{P}$ here) and define $\boldsymbol{g}_z = \partial E / \partial \boldsymbol{z}$ for the scalar energy prediction E. Finally, let $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{C}$ be the gradients calculated by the backward pass, given as

$$\begin{aligned}[\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{C}] &= [\partial E/\partial \boldsymbol{x}, \partial E/\partial \boldsymbol{y}, \partial E/\partial \boldsymbol{W}] \\ &= \mathrm{backward}\left(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W}, \boldsymbol{g}_z\right).\end{aligned}$$

Now our task is to compute ($\partial \mathcal{L}/\partial \boldsymbol{x}$, $\partial \mathcal{L}/\partial \boldsymbol{y}$, $\partial \mathcal{L}/\partial \boldsymbol{W}$, $\partial \mathcal{L}/\partial \boldsymbol{g}_z$) given ($\partial \mathcal{L}/\partial \boldsymbol{a}$, $\partial \mathcal{L}/\partial \boldsymbol{b}$, $\partial \mathcal{L}/\partial \boldsymbol{C}$). We dispatch seven calls to the forward and backward pass kernels:

$$\begin{aligned}\mathrm{op1} &= \mathrm{backward}(\partial \mathcal{L}/\partial \boldsymbol{a}, \partial \mathcal{L}/\partial \boldsymbol{b}, \boldsymbol{W}, \boldsymbol{g}_z) \\ \mathrm{op2} &= \mathrm{backward}(\boldsymbol{x}, \boldsymbol{y}, \partial \mathcal{L}/\partial \boldsymbol{C}, \boldsymbol{g}_z) \\ \mathrm{op3} &= \mathrm{TP}(\partial \mathcal{L}/\partial \boldsymbol{a}, \boldsymbol{y}, \boldsymbol{W}) \\ \mathrm{op4} &= \mathrm{backward}(\partial \mathcal{L}/\partial \boldsymbol{a}, \boldsymbol{y}, \boldsymbol{W}, \boldsymbol{g}_z) \\ \mathrm{op5} &= \mathrm{backward}(\boldsymbol{x}, \partial \mathcal{L}/\partial \boldsymbol{b}, \boldsymbol{W}, \boldsymbol{g}_z) \\ \mathrm{op6} &= \mathrm{TP}(\boldsymbol{x}, \partial \mathcal{L}/\partial \boldsymbol{b}, \boldsymbol{W}) \\ \mathrm{op7} &= \mathrm{TP}(\boldsymbol{x}, \boldsymbol{y}, \partial \mathcal{L}/\partial \boldsymbol{C}).\end{aligned} \tag{2.4}$$

By repeatedly applying the product and chain rules to the formulas for $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{C}$ in Section 2.3.3, we can show

$$\begin{aligned}\partial \mathcal{L}/\partial \boldsymbol{x} &= \mathrm{op1}\,[1] + \mathrm{op2}\,[1] \\ \partial \mathcal{L}/\partial \boldsymbol{y} &= \mathrm{op1}\,[2] + \mathrm{op2}\,[2] \\ \partial \mathcal{L}/\partial \boldsymbol{W} &= \mathrm{op4}\,[3] + \mathrm{op5}\,[3] \\ \partial \mathcal{L}/\partial \boldsymbol{g}_z &= \mathrm{op3} + \mathrm{op6} + \mathrm{op7},\end{aligned} \tag{2.5}$$

where op1 [1], op1 [2], and op1 [3] denote the three results calculated by the backward function, and likewise for op2, op4, and op5. Equations (2.4) and (2.5) could be implemented in less than 10 lines of Python and accelerate the double-backward pass without any additional kernel engineering. In practice, we fuse the forward calls into a single kernel by calling Algorithm 2 three times with different arguments in a procedure like Algorithm 1. The backward calls fuse in a similar manner, and we adopt this approach to dramatically reduce memory traffic and kernel launch overhead.

Our approach offers several additional advantages. Equations (2.4) and (2.5) are agnostic to the structure of $\mathcal{P}$, rendering the double-backward algorithm correct as long as the associated forward and backward kernels are correct. Because Equation (2.4) recursively calls operations with registered autograd formulas, an autograd framework can compute *arbitrary* higher

partial derivatives of our model, not just double-backward. Finally, these formulas continue to apply when the CG tensor product is embedded within a graph convolution (see Section 2.3.5), composing seamlessly with the other optimizations we introduce.

2.3.5 Graph Convolution and Kernel Fusion

Figure 2.7 illustrates two typical use cases of the CG tensor product kernel. The first case (2.7A) calls the kernel illustrated in Figure 2.2 several times with unique triples of $(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{W})$ inputs, and we have already addressed its implementation. The second case (2.7B) embeds the CG tensor product into a graph convolution operation [Tho+18; Bat+22b; Bat+22a]. Here, the nodes of a graph typically correspond to atoms in a simulation and edges represent pairwise interactions. For a symmetric, directed graph $G = (V, E)$, let $\boldsymbol{x}_1 ... \boldsymbol{x}_{|V|}$ $\boldsymbol{y}_1 ... \boldsymbol{y}_{|E|}$, and $\boldsymbol{W}_1 ... \boldsymbol{W}_{|E|}$ be node embeddings, edge embeddings, and trainable edge weights, respectively. Then each row $\boldsymbol{z}_j$ of the graph convolution output, $j \in [|V|]$, is given by

$$\boldsymbol{z}_j = \sum_{(j,k,e) \in \mathcal{N}(j)} \mathrm{TP}(\mathcal{P}, \boldsymbol{x}_k, \boldsymbol{y}_e, \boldsymbol{W}_e), \tag{2.6}$$

where $\mathcal{N}(j)$ denotes the neighbor set of node j and $(j, k, e) \in \mathcal{N}(j)$ indicates that edge e connects nodes j and k. Current equivariant message passing networks [Bat+22b; Bat+22a] implement Equation (2.6) by duplicating the node features to form $\boldsymbol{x}'_1, ..., \boldsymbol{x}'_{|E|}$, calling the large batch kernel developed earlier, and then executing a scatter-sum (also called reduce-by-key) to perform aggregation. Unfortunately, duplicating the node features incurs significant memory and communication-bandwidth overhead when $|E| \gg |V|$ (see Table 2.3).

Notice that graph convolution exhibits a memory access pattern similar to sparse-dense matrix multiplication (SpMM) [YBO18]. We provide two procedures for the fused CGTP / graph convolution based on classic SpMM methods. The first, detailed in Algorithm 4, requires row-major sorted edge indices and iterates over the phases of the computation schedule as the outer loop. The latter change enables the algorithm to keep a running buffer $\boldsymbol{z}_{\mathrm{acc}}$ that accumulates the summation in Equation (2.6) for each node. The buffer $\boldsymbol{z}_{\mathrm{acc}}$ is only flushed to global memory when a warp transitions to a new row of the graph adjacency matrix, reducing global memory writes from $O(|E|)$ to $O(|V|)$. To handle the case where two or more warps calculate contributions to the same node, we write the first row processed by each warp to a fixup buffer [YBO18]. We developed a backward pass kernel using a similar SpMM strategy, but a permutation that transposes the graph adjacency matrix is required as part of the input.

The second algorithm, which we omit for brevity, functions almost identically to Algorithm 4, but replaces the fixup / store logic with an atomic accumulation at every inner loop iteration. This *nondeterministic* method performs $O(|E|)$ atomic storebacks, but does not require a sorted input graph or adjacency transpose permutation.

Algorithm 4 Deterministic TP + Graph Convolution

Require: Graph $G = (V, E)$, $E(b) = (i_b, j_b)$
Require: Edges in E sorted by first coordinate
Require: Batch $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_{|V|}$, $\boldsymbol{y}_1, \ldots, \boldsymbol{y}_{|E|}$, $\boldsymbol{W}_1, \ldots, \boldsymbol{W}_{|E|}$
for $\text{seg}_i \in$ schedule **do**
  $(s, t) \leftarrow E[k][0], E[k][1]$
  Set $\boldsymbol{z}_{\text{acc}} \leftarrow 0$
  for $b = 1, \ldots, |E|$ **do**
    $\boldsymbol{x}_{\text{smem}} = \boldsymbol{x}_t \left[\text{seg}_{i\ \text{(x start)}} : \text{seg}_{i\ \text{(x end)}}\right]$
    Load $\boldsymbol{y}_{\text{smem}}, \boldsymbol{W}_{\text{smem}}$ similarly, set $\boldsymbol{z}_{\text{smem}} \leftarrow 0$
    Run kernels as in Algorithm 1
    $\boldsymbol{z}_{\text{acc}}$ += $\boldsymbol{z}_{\text{smem}}$
    if $b = |E|$ or $s < E\,[b+1]\,[0]$ **then**
      if s is the first vertex processed by warp **then**
        Send $\boldsymbol{z}_{\text{acc}}$ to fixup buffer
      else
        -$\boldsymbol{z}_s \left[\text{seg}_{i\ \text{(z start)}} : \text{seg}_{i\ \text{(z end)}}\right]$ += $\boldsymbol{z}_{\text{acc}}$
      $\boldsymbol{z}_{\text{acc}} = 0$
Execute fixup kernel

2.3.6 Analysis and Related Work Comparison

Our JIT-based approach embeds the sparse tensor structure and values into the GPU instruction stream, which is only possible because $\mathcal{P}$ has relatively few nonzero entries in each block. Because GPU instructions must also be fetched from global memory, performance eventually degrades as blocks exceed several thousand nonzero entries. We do not encounter such tensors in practical models.

Kondor et al. [KLT18] used the GElib library [KT24] to implement one of the original O(3)-equivariant deep learning models. Their library also employs fine-grained loop unrolling in CUDA and parallelizes distinct tensor product evaluations across threads. Despite these strong innovations, their statically compiled library does not provide an optimized backward pass, synchronizes at the CTA rather than warp level, and handles each nonzero block of $\mathcal{P}$ with a separate kernel invocation.

2.4 Experiments

Our kernel generator is available online[1] as an installable Python package. We adopted the frontend interface of e3nn [GS22; Gei+22] and used QuTiP [JNN13; Lam+24] to generate CG

[1] `https://github.com/PASSIONLab/OpenEquivariance`

coefficients. We tested correctness against e3nn to ensure that our kernels produce identical results, up to floating point roundoff and a well-defined permutation of the weights $\boldsymbol{W}$ on certain input configurations. In cases where weight reordering is required, we provide a function for easy migration. We use the NVIDIA and AMD HIP Runtime Compilers to compile our generated kernels through a C++ extension to Python. Figure 2.3 illustrates our software stack.

Quantity	Value
FP32 Peak	19.5 TFLOP/s
FP64 SIMT Peak	9.7 TFLOP/s
FP64 Tensor Core Peak	19.5 TFLOP/s
HBM2 Bandwidth	2.04 TB/s

Table 2.1: A100-SXM4-80GB performance characteristics [Cor20].

The majority of experiments were conducted on NVIDIA A100 GPU nodes of NERSC Perlmutter (each equipped with an AMD EPYC 7763 CPU). Table 2.1 lists the advertised maximum memory bandwidth and compute peaks for multiple data types, a yardstick for our results. Section 2.4.4 covers performance on other GPU models.

As baselines, we used the PyTorch versions of e3nn (v0.5.6) [Gei+22] and NVIDIA cuEquivariance (v0.4.0) [Gei+24]. The e3nn implementation was accelerated with `torch.compile` except where prohibited by memory constraints. For Figures 2.8, 2.9, 2.10, and 2.12, we benchmarked all functions through a uniform PyTorch interface and included any overhead in the measured runtime. Figures 2.11 and 2.13 (right) rely on kernel runtime measurements without PyTorch overhead.

cuEquivariance experienced a significant efficiency increase since v0.2.0, the latest version available when our first preprint was released (see Figures 2.11, 2.13). Since that early release, the authors also added JIT capability and fused convolution, although the closed source kernel backend renders the details opaque. Unless otherwise noted, we report all benchmarks against cuE v0.4.0.

2.4.1 Forward / Backward Throughput

We first profiled our kernels on a large collection of model configurations used by Nequip [Bat+22b] and MACE [Bat+22a]. For each model, we selected an expensive representative tensor product to benchmark. Figure 2.8 shows the results of our profiling on configurations that use only Kernel B (see Figure 2.5). Our median FP32 speedup over e3nn was 5.9x, resp. 4.8x, for the forward and backward passes, with a maximum of 9.2x for the forward pass. We observed a median speedup of 1.6x over cuE for the FP32 forward pass, which drops to 1.3x

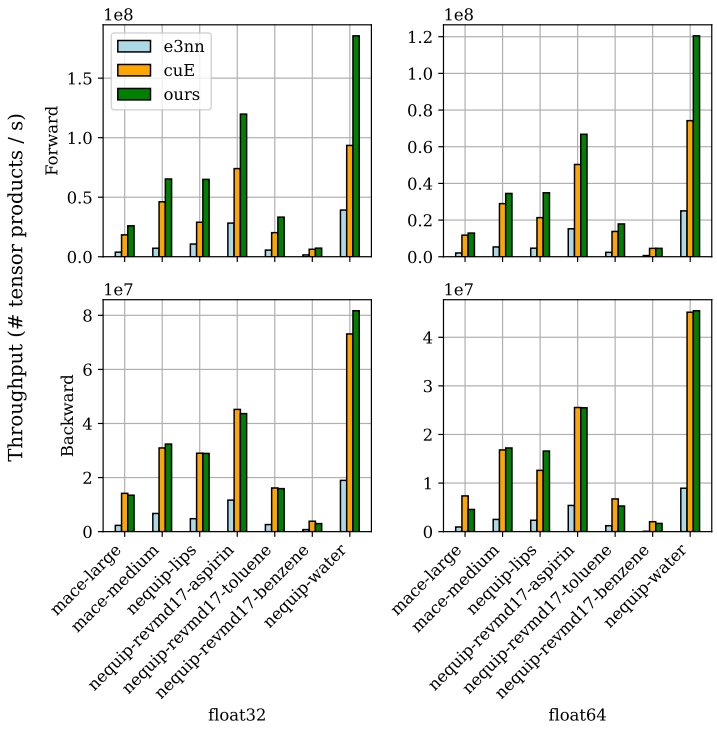


Figure 2.8: Throughput of CG tensor products (batch size 50K), kernel B configurations without SpMM kernel fusion. On difficult configurations like Nequip-benzene with massive output vector lengths, we exhibit more than 10x improvement over e3nn.

in FP64 precision. Our median performance approaches parity with cuE on the backward pass, with a minimum and maximum speedup of 0.72x and 1.32x in FP64 precision.

To benchmark kernel C, we used the Tetris polynomial from e3nn's documentation [Gei+22] and two configurations based on DiffDock [Cor+23]. We exhibit between 1.4x and 2.0x speedup over cuE for both forward and backward passes on DiffDock. Our speedups over e3nn are less dramatic for kernel C, which has a workload dominated by the small dense matrix multiplication in Algorithms 2 and 3.

2.4.2 Second Derivative Performance

We analyzed the double backward pass for the tensor products from the prior section (excluding the Tetris polynomial, which does not require it). Figure 2.10 shows our results. Across data types and model configurations, our speedup ranges from 5.5x to 35x over e3nn and 0.69x-1.69x over cuE. Although our median speedup over cuE is 0.73x in FP32 precision and 0.93x for FP64 precision, we exhibit lower runtime on all DiffDock tensor products and

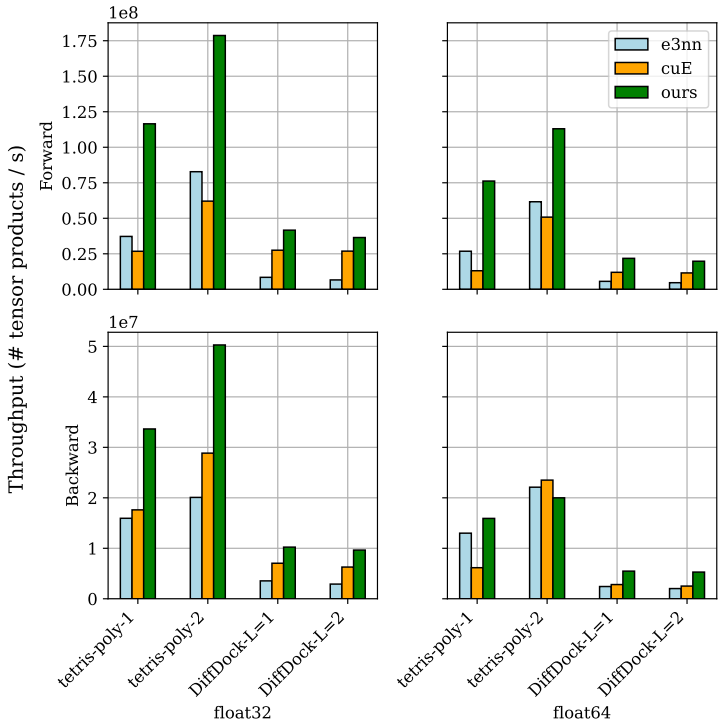


Figure 2.9: Throughput of CG tensor products (batch size 50K), kernel C configurations. We exhibit up to 2x speedup over cuE on the DiffDock tensor products.

several Nequip configurations.

For many Nequip / MACE configurations, the performance gap between our implementation and cuE could likely shrink with some judicious kernel tuning. In particular, we could improve our heuristic selection of the warp count per block, the number of blocks, and the shared memory allotted to each block. We leave tuning these hyperparameters as future work.

2.4.3 Roofline Analysis

We conducted a roofline analysis [WWP09] by profiling our forward / backward pass implementations on varied input configurations. We profiled tensor products with a single "B" subkernel (see Figure 2.5) with FP32 precision, core building blocks for models like Nequip and MACE. The arithmetic intensity of the CG tensor product depends on the structure of the sparse tensor, and we profiled configurations with progressively increasing arithmetic intensity.

Figure 2.11 shows our results, which indicate consistently high compute and bandwidth

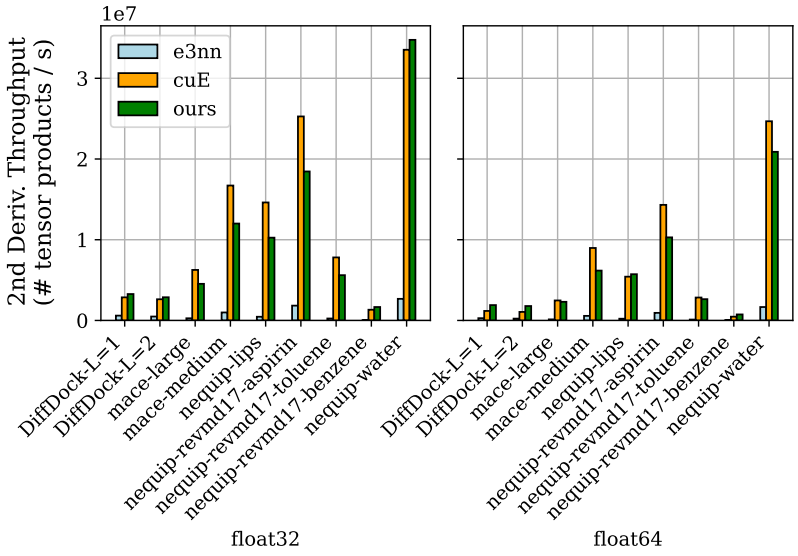


Figure 2.10: Throughput of second derivative kernels (batch size 20K) for chemistry / protein models.

utilization for both our kernels and the latest version of cuE. An earlier version of cuE (v0.2.0, indicated on the plot as cuE-old) exhibited significantly lower efficiency, which has since been corrected by the package authors. The performance of all kernels saturates at 58% of the FP32 peak, likely because Algorithms 2 and 3 contain contain a significant fraction of non fused-multiply-add (FMA) instructions.

2.4.4 Additional GPU Models

We also tested our kernels on the NVIDIA A5000 and a single GPU die of the AMD MI250x. Table 2.2 compares the MACE tensor product runtime across architectures and kernel providers; our codebase contains a more complete set of benchmarks. The A5000 performance matches our expectations given its lower memory bandwidth compared to the A100. While we also saw significant speedup on the MI250x, we detected somewhat lower memory bandwidth utilization than predicted.

2.4.5 Kernel Fusion Benchmarks

We conducted our kernel fusion experiments on three molecular structure graphs listed in Table 2.3. We downloaded the atomic structures of human dihydrofolate deductase (DHFR) and the SARS-COV-2 glycoprotein spike from the Protein Data Bank and constructed a radius-neighbors graph for each using Scikit-Learn [Ped+11]. The carbon lattice was provided to us as a representative workload for MACE [Bat+24].

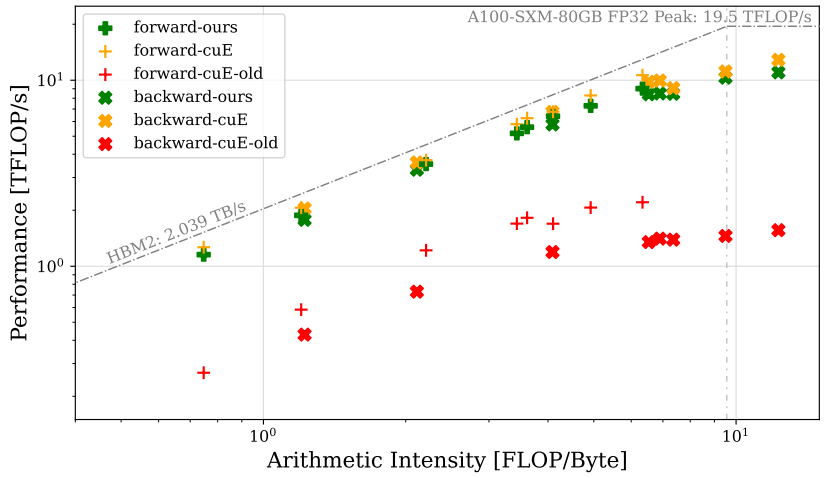


Figure 2.11: Roofline analysis for input configurations of varying arithmetic intensity, batch size 200K. Our kernels and the latest version of cuE closely track the slope of the global memory roofline, indicating high efficiency. An older version of cuE (v0.2.0) is also included to highlight the performance improvement in their package (see top of Section 2.4).

GPU	forward			backward		
	e3nn	cuE	ours	e3nn	cuE	ours
A100	13	2.8	**2.0**	21	**3.5**	3.7
A5000	29	4.2	**3.8**	42	**9.3**	11
MI250x	41	-	**3.0**	128	-	**15**

Table 2.2: MACE-large isolated tensor product runtime (ms), batch size 50K, FP32 unfused.

Graph	Nodes	Adj. Matrix NNZ
DHFR 1DRF	1.8K	56K
COVID spike 6VXX	23K	136K
Carbon lattice	1K	158K

Table 2.3: Molecular graphs for kernel fusion experiments.

Figure 2.12 shows the speedup of fused implementations benchmarked on the most expensive tensor product in the MACE-large model. The baseline, "cuE-scattersum", implements the unfused strategy in Section 2.3.5 by duplicating node embeddings, executing a large batch of tensor products with cuEquivariance, and finally performing row-based reduction by keys. Our deterministic fused algorithm offers the greatest speedup in FP64 precision on the carbon lattice forward pass over cuE (roughly 1.3x speedup). On the other hand, cuE-fused offers

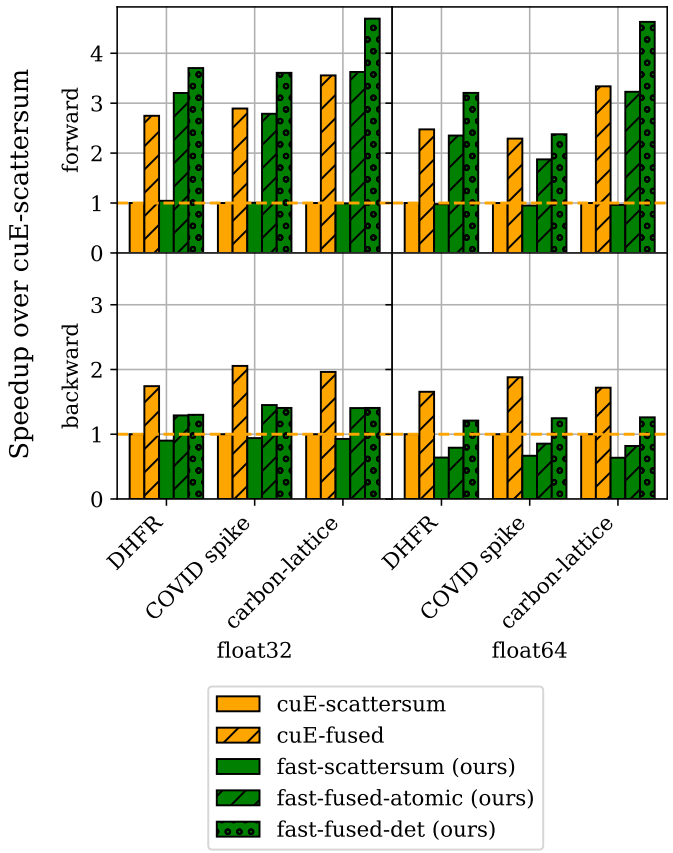


Figure 2.12: Speedup of convolution kernels over cuE-scattersum, which calls cuEquivariance and follows it by a scatter-sum operation. cuE-fused refers to a new kernel introduced in cuE v0.4.0.

1.3-1.4x speedup on the backward pass, and we aim to close this performance gap.

2.4.6 Acceleration of Nequip and MACE

The Nequip [Bat+22b; Tan+25] and MACE [Bat+22a; Bat+24] interatomic potential models implement the equivariant graph neural network architecture in Figure 2.7B. Both have similar message passing structures, while MACE incorporates higher order interactions for its node features. Our first benchmark uses the Nequip-ASE calculator interface to evaluate forces on a large box of water molecules. Due to recent updates to Nequip's software [Tan+25] and time constraints, we only used the nondeterministic fused convolution for our measurements, which appear in Table 2.4. For simplicity, we report speedup with respect to the Nequip Python interface without JITScript or `torch.compile`, although our package is fully compatible with these subsystems.

GPU	Speedup over Unmodified Nequip	
	ours-unfused	ours-fused
A100	6.3x	7.8x
MI250x	3.9x	4.4x

Table 2.4: Force evaluation speedup for our kernels on a 4-layer FP32 Nequip model, 5184-atom water box system.

For MACE, we patched the code to sort nonzero entries of the atomic adjacency matrix according to compressed sparse row (CSR) order. We then substituted our deterministic fused convolution into the model and conducted our benchmark on the carbon lattice in Table 2.3. MACE uses a distinct set of weight matrices for each atomic species, and an inefficiency in the baseline code causes its runtime to increase disproportionately to the useful computation involved. Our model has a species dictionary of eight elements (to trigger the problem in the baseline code, even though the carbon lattice only requires one), and our package includes a module to optimize away the inefficiency.

Figure 2.13 (left) compares the rate of molecular dynamics simulation among the different kernel providers. We benchmarked cuE with the optimal data layout for its irreps and included optimizations for symmetric tensor contraction, linear combination layers, and graph convolution. In FP32 precision, we provide a 5.1x speeedup over e3nn and 1.5x over the older v0.2.0 cuE package, noting that the latter does not provide kernel fusion. A similar speedup exists for the FP64 models. Our implementation, however, achieves 0.73x speedup compared to the latest version of cuE that introduces kernel fusion.

To further investigate these benchmarks, Figure 2.13 (right) breaks down device runtime spent in various kernels. Our time spent on the tensor product (CTP kernels) falls within 2-3 milliseconds of cuE, a highly competitive result. Our performance suffers due to the remaining model components, which contribute to less than 15% of the unoptimized model runtime. To address this, we created a hybrid model (ours-cuE-hybrid) that combines our fused convolution with the linear / symmetric contraction layers offered by cuE. While the hybrid model closes the gap further, the runtime of the remaining kernels is still higher than cuE. This is because the hybrid model preserves the original data layout of MACE layers, whereas cuE transposes several key weight matrices to achieve higher performance. As a consequence, cuE requires a data reordering function for models trained without the package, whereas our kernels have no such restriction.

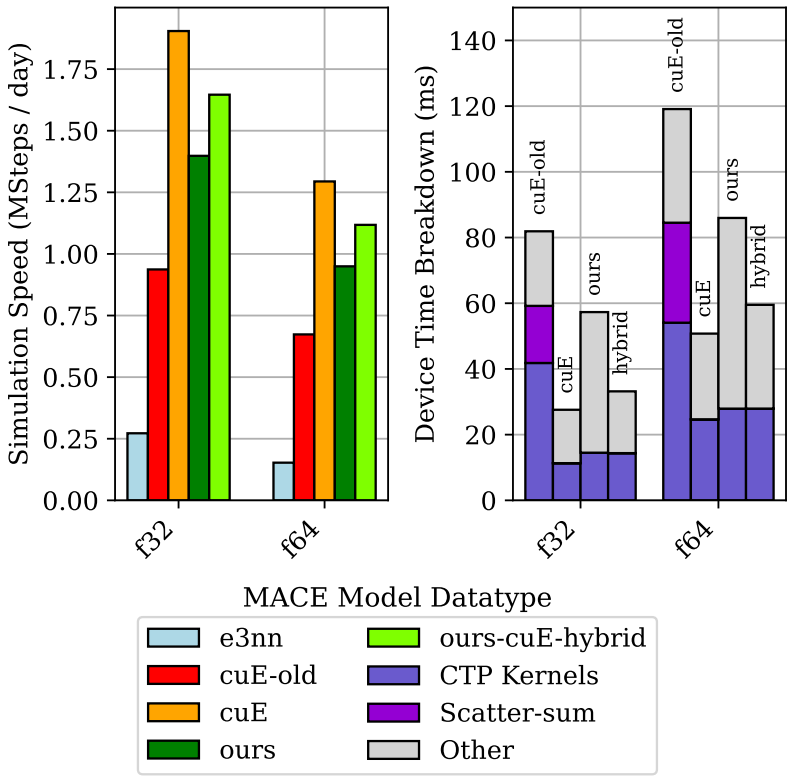


Figure 2.13: Simulation speed of MACE for varying kernel provider (left) and device time breakdown (right). cuE-old refers to version 0.2.0 of their package, while the hybrid implementation combines our fused convolution with other model primitives optimized by cuE.

2.5 Conclusions and Further Work

We have established that our sparse kernels achieve consistently high performance on several common primitives used in O(3)-equivariant deep neural networks. We see several avenues for future progress:

- **Low Precision Data and MMA Support**: Our kernels rely on the single-instruction multiple thread (SIMT) cores for FP32 and FP64 floating point arithmetic. Modern GPUs offer specialized hardware for lower-precision calculation, both using SIMT cores and within matrix-multiply-accumulate (MMA) units. We hope to harness these capabilities in the future.
- **Stable Summation During Convolution**: Our kernel generator allows us to easily extend our methods to use stable (Kahan) summation [Kah65] within fused graph

convolution. Kahan summation reduces numerical roundoff error during feature vector aggregation across node neighborhoods, promoting energy conservation in simulations.

- **Integration into new models**: Our open-source software remains accessible to newcomers while delivering the high performance required for massive workloads. In conjunction with domain experts, we hope to apply our library to train larger, more expressive equivariant deep neural networks.

2.6 Further Details: Passaro and Zitnick's Algorithm

Passaro and Zitnick [PZ23] detail an elegant strategy to accelerate CG tensor contraction that the EquiformerV2 [Lia+24] model subsequently adopted. This section adapts their description to our notation, explains their algorithm, and examines its computational characteristics.

For simplicity, we redefine the CG tensor product in this section to omit the weight matrix from Equation (2.1). Multiplication by the weights can occur after the tensor contraction without affecting the calculation. We seek an efficient algorithm to compute

$$\begin{aligned} \boldsymbol{z}(\boldsymbol{v}) &= \mathrm{TP}(\mathcal{P}, \boldsymbol{x}(\boldsymbol{v}), \boldsymbol{y}(\boldsymbol{v})) \\ &:= \boldsymbol{P} \cdot (\boldsymbol{x}(\boldsymbol{v}) \otimes \boldsymbol{y}(\boldsymbol{v})), \end{aligned} \tag{2.7}$$

where we make explicit that $\boldsymbol{x}$, $\boldsymbol{y}$, and $\boldsymbol{z}$ are functions of some common input $\boldsymbol{v}$. The equivariance assumptions on all three yield the relations

$$\boldsymbol{D}_x(g) \cdot \boldsymbol{x}(\boldsymbol{v}) = \boldsymbol{x}\left(\boldsymbol{D}_{\text{in}}(g) \cdot \boldsymbol{v}\right), \tag{2.8}$$

$$\boldsymbol{D}_y(g) \cdot \boldsymbol{y}(\boldsymbol{v}) = \boldsymbol{y}\left(\boldsymbol{D}_{\text{in}}(g) \cdot \boldsymbol{v}\right), \tag{2.9}$$

$$\boldsymbol{D}_z(g) \cdot \boldsymbol{z}(\boldsymbol{v}) = \boldsymbol{z}\left(\boldsymbol{D}_{\text{in}}(g) \cdot \boldsymbol{v}\right), \tag{2.10}$$

for all group elements $g \in O(3)$. To simplify the analysis further, assume that $\boldsymbol{D}_x, \boldsymbol{D}_y$ and $\boldsymbol{D}_z$ are all irreducible representations.

Suppose $\boldsymbol{y}(\boldsymbol{v})$ is a spherical harmonic function evaluated on a 3D coordinate embedded in $\boldsymbol{v}$ (as is the case for many equivariant neural architectures [Bat+22b; Bat+22a; Lia+24]). Here, Passaro and Zitnick [PZ23] make a key observation: given an arbitrary vector $\boldsymbol{v}$, there exists a group element g^* satisfying

$$\boldsymbol{D}_y(g^*) \cdot \boldsymbol{y}(\boldsymbol{v}) = \boldsymbol{y}\left(\boldsymbol{D}_{\text{in}}(g^*) \cdot \boldsymbol{v}\right) = \boldsymbol{e}_k, \tag{2.11}$$

for any $1 \leq k \leq \dim(\boldsymbol{y})$, where $\boldsymbol{e}_k$ is the k-th standard basis vector. g^* should properly be written as $g^*(\boldsymbol{v})$ to highlight its dependence on $\boldsymbol{v}$, but we abbreviate the notation when the dependence is irrelevant. This fact follows from the surjection property of spherical harmonic

bases when they are viewed as vector-valued functions [PZ23]. Now beginning with Equation (2.10), we substitute g^* and derive

$$\begin{aligned}
\boldsymbol{z}(\boldsymbol{v}) &= \boldsymbol{D}_z(g^*)^{-1} \cdot \boldsymbol{z}\left(\boldsymbol{D}_{\text{in}}(g^*) \cdot \boldsymbol{v}\right) \\
&= \boldsymbol{D}_z(g^*)^{-1} \cdot \boldsymbol{P} \cdot \left(\boldsymbol{x}\left(\boldsymbol{D}_{\text{in}}(g^*) \cdot \boldsymbol{v}\right) \otimes \boldsymbol{y}\left(\boldsymbol{D}_{\text{in}}(g^*) \cdot \boldsymbol{v}\right)\right) \\
&= \boldsymbol{D}_z(g^*)^{-1} \cdot \boldsymbol{P} \cdot \left(\boldsymbol{x}\left(\boldsymbol{D}_{\text{in}}(g^*) \cdot \boldsymbol{v}\right) \otimes \boldsymbol{e}_k\right) \\
&= \boldsymbol{D}_z(g^*)^{-1} \cdot \boldsymbol{P} \cdot \left(\left(\boldsymbol{D}_x(g^*) \cdot \boldsymbol{x}(\boldsymbol{v})\right) \otimes \boldsymbol{e}_k\right) \\
&= \boldsymbol{D}_z(g^*)^{-1} \cdot \boldsymbol{P}_{\text{sparse}} \cdot \boldsymbol{D}_x(g^*) \cdot \boldsymbol{x}(\boldsymbol{v}) \\
&= \boldsymbol{D}_z(g^{*-1}) \cdot \boldsymbol{P}_{\text{sparse}} \cdot \boldsymbol{D}_x(g^*) \cdot \boldsymbol{x}(\boldsymbol{v}).
\end{aligned} \tag{2.12}$$

The first step shifts $\boldsymbol{D}_z(g^*)$ from the left side of Equation (2.10) to the right side. The second step substitutes the definition of $\boldsymbol{z}(\boldsymbol{v})$ from Equation (2.7). The third step and fourth steps use Equations (2.11) and (2.8), respectively. In step 5, observe that multiplying $\boldsymbol{P}$ against a Kronecker product of a general argument and a standard basis vector effectively removes columns of $\boldsymbol{P}$ from the computation. We call this downsampled matrix $\boldsymbol{P}_{\text{sparse}}$. In the last step, we use the properties of the group representation $\boldsymbol{D}_z$ to avoid matrix inversion.

The last line of Equation (2.12) is theoretically faster to compute than Equation 2.7. The tensor-times-multiple-vector kernel has been replaced by a string of matrix-vector multiplications, while $\boldsymbol{P}_{\text{sparse}}$ has asymptotically fewer nonzero elements than $\mathcal{P}$. The only remaining details are computation of g^*, g^{*-1}, $\boldsymbol{D}_z$, and $\boldsymbol{D}_x$. Passaro and Zitnick [PZ23] demonstrate that the desired group elements can be identified efficiently, while e3nn [Gei+22] contains utilities to materialize the Wigner matrices.

To interact representations of high order, Passaro and Zitnick [PZ23] demonstrate significant computational advantage over e3nn's native tensor product. While we leave benchmarking against our implementation as future work, there exist obstacles to high-performance implementation of Equation (2.12). To rely on primitives like (batched) dense matrix multiplication, $\boldsymbol{D}_z$ and $\boldsymbol{D}_x$ must be evaluated and stored for each edge of an atomic graph, as the key group element $g^*(\boldsymbol{v})$ is potentially unique for each edge. Though relatively inexpensive, these evaluations involve matrix exponentiation that is antagonistic to kernel fusion. The string of matrix-vector products also incurs a longer critical path than directly calculating Equation (2.1), which benefits from instruction-level parallelism notwithstanding its higher operation count. For the lower-order representations in practical models, we hypothesize that our optimized kernel package competes effectively against the discussed sparsification approach.

Chapter 3

Fast Khatri-Rao Product Leverage Sampling

In Chapter 2, we leaned heavily on known structure in the tensor kernel to achieve high performance. In this chapter, one of our main tasks is to *discover* structure in a tensor by decomposing it into a set of small matrices. We will use block coordinate descent to optimize each matrix in turn, and the matricized tensor times Khatri-Rao product (MTTKRP) serves as the workhorse kernel to achieve this objective.

As we cannot avail of any specific property of the tensor, we resort instead to downsampling rows of the Khatri-Rao product while preserving its singular value spectrum. The algorithm we devise achieves best-in-class asymptotic performance for alternating PARAFAC decomposition. To bolster these theoretical claims, we demonstrate empirically that our method achieves higher accuracy and lower time to achieve fixed accuracy thresholds compared to recently proposed randomized methods.

Our algorithm achieves 1.5-2.5x speedup over CP-ARLS-LEV [LK22], a state-of-the art approximate MTTKRP sampling algorithm, and reduces the required sample count by more than 50x on certain tensors. The building blocks of our sampling algorithm find broader use as well. In Chapter 5, we use theorems developed here to downsample an entirely different tensor network with minimal new theory.

3.1 Introduction

The Khatri-Rao product (KRP, denoted by $\odot$) is the column-wise Kronecker product of two matrices, and it appears in diverse applications across numerical analysis and machine learning [LT08]. We examine overdetermined linear least squares problems of the form $\min_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_F$, where the design matrix $\boldsymbol{A} = \boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$ is the Khatri-Rao product

of matrices $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$. These problems appear prominently in signal processing [SB02; Tok+25], inverse problems related to partial differential equations [Che+20a], and alternating least squares (ALS) CANDECOMP / PARAFAC (CP) tensor decomposition [KB09]. In this work, we focus on the case where $\boldsymbol{A}$ has moderate column count (several hundred at most). Despite this, the problem remains formidable because the height of $\boldsymbol{A}$ is $\prod_{j=1}^{N} I_j$. For row counts I_j in the millions, it is intractable to even materialize $\boldsymbol{A}$ explicitly.

Several recently-proposed randomized sketching algorithms can approximately solve least squares problems with Khatri-Rao product design matrices [BBK18; JKW20; LK22; Mal22; WZ22]. These methods apply a sketching operator $\boldsymbol{S}$ to the design and data matrices to solve the reduced least squares problem $\min_{\tilde{\boldsymbol{X}}} \left\| \boldsymbol{S}\boldsymbol{A}\tilde{\boldsymbol{X}} - \boldsymbol{S}\boldsymbol{B} \right\|_F$, where $\boldsymbol{S}$ has far fewer rows than columns. For appropriately chosen $\boldsymbol{S}$, the residual of the downsampled system falls within a specified tolerance ε of the optimal residual with high probability $1 - \delta$. In this work, we constrain $\boldsymbol{S}$ to be a *sampling matrix* that selects and reweights a subset of rows from both $\boldsymbol{A}$ and $\boldsymbol{B}$. When the rows are selected according to the distribution of *statistical leverage scores* on the design matrix $\boldsymbol{A}$, only $\tilde{O}\left(R/(\varepsilon\delta)\right)$ samples are required (subject to the assumptions at the end of section 3.2.1). The challenge, then, is to efficiently sample according to the leverage scores when $\boldsymbol{A}$ has Khatri-Rao structure.

We propose a leverage-score sampler for the Khatri-Rao product of matrices with tens of millions of rows each. After construction, our sampler draws each row in time quadratic in the column count, but logarithmic in the total row count of the Khatri-Rao product. Our core contribution is the following theorem.

Theorem 3.1.1 (Efficient Khatri-Rao Product Leverage Sampling)**.** *Given $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ with $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$, there exists a data structure satisfying the following:*

1. *The data structure has construction time $O\left(\sum_{j=1}^{N} I_j R^2\right)$ and requires additional storage space $O\left(\sum_{j=1}^{N} I_j R\right)$. If a single entry in a matrix $\boldsymbol{U}_j$ changes, it can be updated in time $O(R \log(I_j/R))$. If the entire matrix $\boldsymbol{U}_j$ changes, it can be updated in time $O(I_j R^2)$.*
2. *The data structure produces J samples from the Khatri-Rao product $\boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$ according to the exact leverage score distribution on its rows in time*

$$O\left(NR^3 + J\sum_{k=1}^{N} R^2 \log \max\left(I_k, R\right)\right)$$

using $O(R^3)$ scratch space. The structure can also draw samples from the Khatri-Rao product of any subset of $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$.

The efficient update property and ability to exclude one matrix are important in CP decomposition. When the inputs $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ are sparse, an analogous data structure with $O\left(R\sum_{j=1}^{N} \text{nnz}(\boldsymbol{U}_j)\right)$ construction time and $O\left(\sum_{j=1}^{N} \text{nnz}(\boldsymbol{U}_j)\right)$ storage space exists with identical sampling time. Since the output factor matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ are typically dense, we defer the proof to Section 3.6.6. Combined with error guarantees for leverage-score sampling, we achieve an algorithm for alternating least squares CP decomposition with asymptotic complexity lower than recent state-of-the-art methods (see Table 3.1).

Our method provides the most practical benefit on sparse input tensors, which may have dimension lengths in the tens of millions (unlike dense tensors that quickly incur intractable storage costs at large dimension lengths) [Smi+17]. On the Amazon and Reddit tensors with billions of nonzero entries, our algorithm STS-CP can achieve 95% of the fit of non-randomized ALS between 1.5x and 2.5x faster than a high-performance implementation of the state-of-the-art CP-ARLS-LEV algorithm [LK22]. Our algorithm is significantly more sample-efficient; on the Enron tensor, only $\sim 65,000$ samples per solve were required to achieve the 95% accuracy threshold above a rank of 50, which could not be achieved by CP-ARLS-LEV with even 54 times as many samples.

Table 3.1: Asymptotic Complexity to decompose an N-dimensional $I \times ... \times I$ dense tensor via CP alternating least squares. For randomized algorithms, each approximate least-squares solution has residual within $(1+\varepsilon)$ of the optimal value with high probability $1-\delta$. Factors involving $\log R$ and $\log(1/\delta)$ are hidden ($\tilde{O}$ notation). See Section 3.2.2 for details about each algorithm.

Algorithm	Source	Complexity per Iteration
CP-ALS	[KB09]	$N(N+I)I^{N-1}R$
CP-ARLS-LEV	[LK22]	$N(R+I)R^N/(\varepsilon\delta)$
TNS-CP	[MBM22]	$N^3IR^3/(\varepsilon\delta)$
Gaussian TNE	[MS22]	$N^2(N^{1.5}R^{3.5}/\varepsilon^3 + IR^2)/\varepsilon^2$
STS-CP	Ours	$N(NR^3 \log I + IR^2)/(\varepsilon\delta)$

3.2 Preliminaries and Related Work

Notation We require some additional notation beyond the symbols defined in Section 1.1. We use $[N]$ to denote the set $\{1, ..., N\}$ for a positive integer N. $\tilde{O}$ notation indicates the presence of multiplicative terms polylogarithmic in R and $(1/\delta)$ in runtime complexities. For the complexities of our methods, these logarithmic factors are no more than $O(\log(R/\delta))$.

We use angle brackets $\langle \cdot, ..., \cdot \rangle$ to denote a **generalized inner product**. For identically-sized vectors / matrices, it returns the sum of all entries in their elementwise product. For

$\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{m \times n}$,

$$\langle \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \rangle := \sum_{i=1,j=1}^{m,n} \boldsymbol{A}[i,j]\, \boldsymbol{B}[i,j]\, \boldsymbol{C}[i,j].$$

3.2.1 Sketched Linear Least Squares

A variety of random sketching operators $\boldsymbol{S}$ have been proposed to solve overdetermined least squares problems $\min_{\boldsymbol{X}} \|\boldsymbol{A}\boldsymbol{X} - \boldsymbol{B}\|_F$ when $\boldsymbol{A}$ has no special structure [Woo+14; AC09]. When $\boldsymbol{A}$ has Khatri-Rao product structure, prior work has focused on *sampling* matrices [Che+16; LK22], which have a single nonzero entry per row, operators composed of fast Fourier / trigonometric transforms [JKW20], or Countsketch-type operators [Wan+15; Ahl+20]. For tensor decomposition, however, the matrix $\boldsymbol{B}$ may be sparse or implicitly specified as a black-box function. When $\boldsymbol{B}$ is sparse, Countsketch-type operators still require the algorithm to iterate over all nonzero values in $\boldsymbol{B}$. As Larsen and Kolda [LK22] note, operators similar to the FFT induce fill-in when applied to a sparse matrix $\boldsymbol{B}$, destroying the benefits of sketching. Similar difficulties arise when $\boldsymbol{B}$ is implicitly specified. This motivates our decision to focus on row sampling operators, which only touch a subset of entries from $\boldsymbol{B}$. Let $\hat{x}_1, ..., \hat{x}_J$ be a selection of J indices for the rows of $\boldsymbol{A} \in \mathbb{R}^{I \times R}$, sampled i.i.d. according to a probability distribution $q_1, ..., q_I$. The associated sampling matrix $\boldsymbol{S} \in \mathbb{R}^{J \times I}$ is specified by

$$\boldsymbol{S}[j,i] = \begin{cases} \frac{1}{\sqrt{J q_i}}, & \text{if } \hat{x}_j = i \\ 0, & \text{otherwise} \end{cases}$$

where the weight of each nonzero entry corrects bias induced by sampling. When the probabilities q_j are proportional to the *leverage scores* of the rows of $\boldsymbol{A}$, strong guarantees apply to the solution of the downsampled problem.

Leverage Score Sampling The leverage scores of a matrix assign a measure of importance to each of its rows. The leverage score of row i from matrix $\boldsymbol{A} \in \mathbb{R}^{I \times R}$ is given by

$$\ell_i = \boldsymbol{A}[i,:] \left(\boldsymbol{A}^\top \boldsymbol{A}\right)^+ \boldsymbol{A}[i,:]^\top \tag{3.1}$$

for $1 \leq i \leq I$. Leverage scores can be expressed equivalently as the squared row norms of the matrix $\boldsymbol{Q}$ in any reduced $\boldsymbol{QR}$ factorization of $\boldsymbol{A}$ [Dri+12]. The sum of all leverage scores is the rank of $\boldsymbol{A}$ [Woo+14]. Dividing the scores by their sum, we induce a probability distribution on the rows used to generate a sampling matrix $\boldsymbol{S}$. The next theorem has appeared in several works, and we take the form given by Malik et al. [MBM22]. For an appropriate sample count, it guarantees that the residual of the downsampled problem is close to the residual of the original problem.

Theorem 3.2.1 (Guarantees for Leverage Score Sampling)**.** *Given $\boldsymbol{A} \in \mathbb{R}^{I \times R}$ and $\varepsilon, \delta \in (0,1)$, let $\boldsymbol{S} \in \mathbb{R}^{J \times I}$ be a leverage score sampling matrix for $\boldsymbol{A}$. Further define $\tilde{\boldsymbol{X}} =$*

$\arg\min_{\boldsymbol{X}} \|\boldsymbol{SAX} - \boldsymbol{SB}\|_F$. *If* $J \gtrsim R\max\left(\log\left(R/\delta\right), 1/(\varepsilon\delta)\right)$*, then with probability at least* $1-\delta$ *it holds that*

$$\left\|\boldsymbol{A}\tilde{\boldsymbol{X}} - \boldsymbol{B}\right\|_F \leq (1+\varepsilon)\min_{\boldsymbol{X}} \|\boldsymbol{AX} - \boldsymbol{B}\|_F .$$

For the applications considered in this work, R ranges up to a few hundred. As ε and δ tend to 0 with fixed R, $1/(\varepsilon\delta)$ dominates $\log(R/\delta)$. Hence, we assume that the minimum sample count J to achieve the guarantees of the theorem is $\Omega(R/(\varepsilon\delta))$.

3.2.2 Prior Work

Khatri-Rao Product Leverage Score Sampling Well-known sketching algorithms exist to quickly estimate the leverage scores of dense matrices [Dri+12]. These algorithms are, however, intractable for $\boldsymbol{A} = \boldsymbol{U}_1 \odot \ldots \odot \boldsymbol{U}_N$ due to the height of the Khatri-Rao product. Cheng et al. [Che+16] instead approximate each score as a product of leverage scores associated with each matrix $\boldsymbol{U}_j$. Larsen and Kolda [LK22] propose CP-ARLS-LEV, which uses a similar approximation and combines random sampling with a deterministic selection of high-probability indices. Both methods were presented in the context of CP decomposition. To sample from the Khatri-Rao product of N matrices, both require $O(R^N/(\varepsilon\delta))$ samples to achieve the (ε, δ) guarantee on the residual of each least squares solution. These methods are simple to implement and perform well when the Khatri-Rao product has column count up to 20-30. On the other hand, they suffer from high sample complexity as R and N increase. The TNS-CP algorithm by Malik et al. [MBM22] samples from the exact leverage score distribution, thus requiring only $O(R/(\varepsilon\delta))$ samples per least squares solve. Unfortunately, it requires time $O\left(\sum_{j=1}^{N} I_j R^2\right)$ to draw each sample.

Details about Table 3.1 With the background of the prior paragraph in mind, we give details about the algorithms listed in Table 3.1. CP-ALS [KB09] is the standard, non-randomized alternating least squares method given by Algorithm 10 in Section 3.6.7. The least squares problems in the algorithm are solved by exact methods. The per-iteration runtime for both CP-ALS and the next table entry, CP-ARLS-LEV, are re-derived in Appendix C.3 of work by Malik [Mal22] from their original sources. Malik [Mal22] proposed the CP-ALS-ES algorithm (not listed in the table), which is superseded by the TNS-CP algorithm [MBM22]. We report the complexity from Table 1 of the latter work. The algorithm by Ma and Solomonik [MS22] is based on a general method to sketch tensor networks. Our reported complexity is listed in Table 1 for Algorithm 1 in their work.

Table 3.1 does not list the one-time initialization costs for any of the methods. All methods require at least $O(NIR)$ time to randomly initialize factor matrices, and CP-ALS requires no further setup. CP-ARLS-LEV, TNS-CP, and STS-CP all require $O(NIR^2)$ initialization

time. CP-ARLS-LEV uses the initialization phase to compute the initial leverage scores of all factor matrices. TNS-CP uses the initialization step to compute and cache Gram matrices of all factors U_j. STS-CP must build the efficient sampling data structure described in Theorem 3.1.1. The algorithm from Ma and Solomonik requires an initialization cost of $O(I^N m)$, where m is a sketch size parameter on the order $O(NR/\varepsilon^2)$ to achieve the (ε, δ) accuracy guarantee for each least squares solve.

Comparison to Woodruff and Zandieh The most comparable results to ours appear in work by Woodruff and Zandieh [WZ22], who detail an algorithm for approximate ridge leverage-score sampling for the Khatri-Rao product in near input-sparsity time. Their work relies on a prior oblivious method by Ahle et al. [Ahl+20], which sketches a Khatri-Rao product using a sequence of Countsketch / OSNAP operators arranged in a tree. Used in isolation to solve a linear least squares problem, the tree sketch construction time scales as $O\left(\frac{1}{\varepsilon}\sum_{j=1}^{N} \text{nnz}(\boldsymbol{U}_j)\right)$ and requires an embedding dimension quadratic in R to achieve the (ε, δ) solution-quality guarantee. Woodruff and Zandieh use a collection of these tree sketches, each with carefully-controlled approximation error, to design an algorithm with linear runtime dependence on the column count R. On the other hand, the method exhibits $O(N^7)$ scaling in the number of matrices involved, has $O(\varepsilon^{-4})$ scaling in terms of the desired accuracy, and relies on a sufficiently high ridge regularization parameter. Our data structure instead requires construction time quadratic in R. In exchange, we use distinct methods to design an efficiently-updatable sampler with runtime linear in both N and ε^{-1}. These properties are attractive when the column count R is below several thousand and when error as low as $\epsilon \approx 10^{-3}$ is needed in the context of an iterative solver (see Figure 3.6). Moreover, the term $O(R^2 \sum_{j=1}^{N} I_j)$ in our construction complexity arises from symmetric rank-k updates, a highly-optimized BLAS3 kernel on modern CPU and GPU architectures. Section 3.6.1 provides a more detailed comparison between the two approaches.

Kronecker Regression Kronecker regression is a distinct (but closely related) problem to the one we consider. There, $A = \boldsymbol{U}_1 \otimes ... \otimes \boldsymbol{U}_N$ and the matrices $\boldsymbol{U}_i$ have potentially distinct column counts $R_1, ..., R_N$. While the product distribution of leverage scores from $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ provides only an approximation to the leverage score distribution of the Khatri-Rao product [Che+16; LK22], it provides the *exact* leverage distribution for the Kronecker product. Multiple works [Dia+19; FFG22] combine this property with other techniques, such as dynamically-updatable tree-sketches [RSZ22], to produce accurate and updatable Kronecker sketching methods. None of these results apply directly in our case due to the distinct properties of Kronecker and Khatri-Rao products.

3.3 An Efficient Khatri-Rao Leverage Sampler

Without loss of generality, we will prove part 2 of Theorem 3.1.1 for the case where $\boldsymbol{A} = \boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$; the case that excludes a single matrix follows by reindexing matrices $\boldsymbol{U}_k$. We further assume that $\boldsymbol{A}$ is a nonzero matrix, though it may be rank-deficient. Similar to prior sampling works [Mal22; WZ22], our algorithm will draw one sample from the Khatri-Rao product by sampling a row from each of $\boldsymbol{U}_1, \boldsymbol{U}_2, ...$ in sequence and computing their Hadamard product, with the draw from U_j conditioned on prior draws from $\boldsymbol{U}_1, ..., \boldsymbol{U}_{j-1}$.

Let us index each row of $\boldsymbol{A}$ by a tuple $(i_1, ..., i_N) \in [I_1] \times ... \times [I_N]$. Equation (3.1) gives

$$\ell_{i_1,...,i_N} = \boldsymbol{A}\left[(i_1, ..., i_N), :\right] \left(\boldsymbol{A}^\top \boldsymbol{A}\right)^+ \boldsymbol{A}\left[(i_1, ..., i_N), :\right]^\top. \tag{3.2}$$

For $1 \leq k \leq N$, define $\boldsymbol{G}_k := \boldsymbol{U}_k^\top \boldsymbol{U}_k \in \mathbb{R}^{R \times R}$ and $\boldsymbol{G} := \left(\circledast_{k=1}^N \boldsymbol{G}_k\right) \in \mathbb{R}^{R \times R}$; it is a well-known fact that $\boldsymbol{G} = \boldsymbol{A}^\top \boldsymbol{A}$ [KB09]. For a single row sample from $\boldsymbol{A}$, let $\hat{s}_1, ..., \hat{s}_N$ be random variables for the draws from multi-index set $[I_1] \times ... \times [I_N]$ according to the leverage score distribution. Assume, for some k, that we have already sampled an index from each of $[I_1], ..., [I_{k-1}]$, and that the first $k-1$ random variables take values $\hat{s}_1 = s_1, ..., \hat{s}_{k-1} = s_{k-1}$. We abbreviate the latter condition as $\hat{s}_{<k} = s_{<k}$. To sample from I_k, we seek the distribution of $\hat{s}_k$ conditioned on $\hat{s}_1, ...\hat{s}_{k-1}$. Define $\boldsymbol{h}_{<k}$ as the transposed elementwise product[1] of rows already sampled:

$$\boldsymbol{h}_{<k} := \bigcircledast_{i=1}^{k-1} \boldsymbol{U}_i\left[s_i, :\right]^\top. \tag{3.3}$$

Also define $\boldsymbol{G}_{>k}$ as

$$\boldsymbol{G}_{>k} := \boldsymbol{G}^+ \circledast \bigcircledast_{i=k+1}^{N} \boldsymbol{G}_i. \tag{3.4}$$

Then the following theorem provides the conditional distribution of $\hat{s}_k$.

Theorem 3.3.1 ([Mal22], Adapted)**.** *For any $s_k \in [I_k]$,*

$$\begin{aligned} p(\hat{s}_k = s_k \mid \hat{s}_{<k} = s_{<k}) &= C^{-1} \langle \boldsymbol{h}_{<k} \boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k\left[s_k, :\right]^\top \boldsymbol{U}_k\left[s_k, :\right], \boldsymbol{G}_{>k} \rangle \\ &:= \boldsymbol{q}_{\boldsymbol{h}_{<k}, \boldsymbol{U}_k, \boldsymbol{G}_{>k}}\left[s_k\right], \end{aligned} \tag{3.5}$$

where $C = \langle \boldsymbol{h}_{<k} \boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k^\top \boldsymbol{U}_k, \boldsymbol{G}_{>k} \rangle$ is nonzero.

We include the derivation of Theorem 3.3.1 from Equation (3.2) in Section 3.6.2, while Figure 3.1 illustrates the three key objects required in the theorem to compute the conditional

[1] For $a > b$, assume that $\circledast_{i=a}^b (...)$ produces a vector / matrix filled with ones.

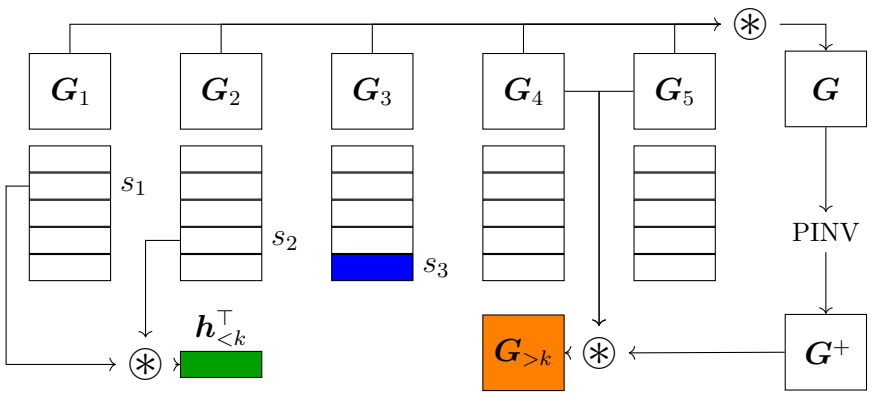


Figure 3.1: Key objects from Theorem 3.3.1 for a Khatri-Rao product of five matrices. The probability of the event $(\hat{s}_3 = s_3)$ conditioned on prior draws depends on the product of rows already selected $(\boldsymbol{h}_{<k}^\top)$; the Gram matrices $\boldsymbol{G}_4$, $\boldsymbol{G}_5$, and $\boldsymbol{G}$ (all of which contribute to $\boldsymbol{G}_{>k}$); and row $\boldsymbol{U}_3[s_3, :]$ itself.

distribution. Computing all entries of the probability vector $\boldsymbol{q}_{\boldsymbol{h}_{<k},\boldsymbol{U}_k,\boldsymbol{G}_{>k}}$ would cost $O(I_j R^2)$ per sample, too costly when $\boldsymbol{U}_j$ has millions of rows. It is likewise intractable (in preprocessing time and space complexity) to precompute probabilities for every possible conditional distribution on the rows of $\boldsymbol{U}_j$, since the conditioning random variable has $\prod_{k=1}^{j-1} I_k$ potential values. Our key innovation is a data structure to sample from a discrete distribution of the form $\boldsymbol{q}_{\boldsymbol{h}_{<k},\boldsymbol{U}_k,\boldsymbol{G}_{>k}}$ *without* materializing all of its entries or incurring superlinear cost in either N or ε^{-1}. We introduce this data structure in the next section and will apply it twice in succession to get the complexity in Theorem 3.1.1.

3.3.1 Efficient Sampling from $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}$

We introduce a slight change of notation in this section to simplify the problem and generalize our sampling lemma. Let $\boldsymbol{h} \in \mathbb{R}^R$ be a vector and let $\boldsymbol{Y} \in \mathbb{R}^{R \times R}$ be a positive semidefinite (p.s.d.) matrix, respectively. Our task is to sample J rows from a matrix $\boldsymbol{U} \in \mathbb{R}^{I \times R}$ according to the distribution

$$\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[s] := C^{-1} \langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}^\top[s, :]\, \boldsymbol{U}[s, :], \boldsymbol{Y} \rangle \tag{3.6}$$

provided the normalizing constant $C = \langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}^\top\boldsymbol{U}, \boldsymbol{Y} \rangle$, is nonzero. We impose that all J rows are drawn with the same matrices $\boldsymbol{Y}$ and $\boldsymbol{U}$, but potentially distinct vectors $\boldsymbol{h}$. The following lemma establishes that an efficient sampler for this problem exists.

Lemma 3.3.2 (Efficient Row Sampler)**.** *Given matrices $\boldsymbol{U} \in \mathbb{R}^{I \times R}, \boldsymbol{Y} \in \mathbb{R}^{R \times R}$ with $\boldsymbol{Y}$ p.s.d., there exists a data structure parameterized by positive integer F that satisfies the following:*

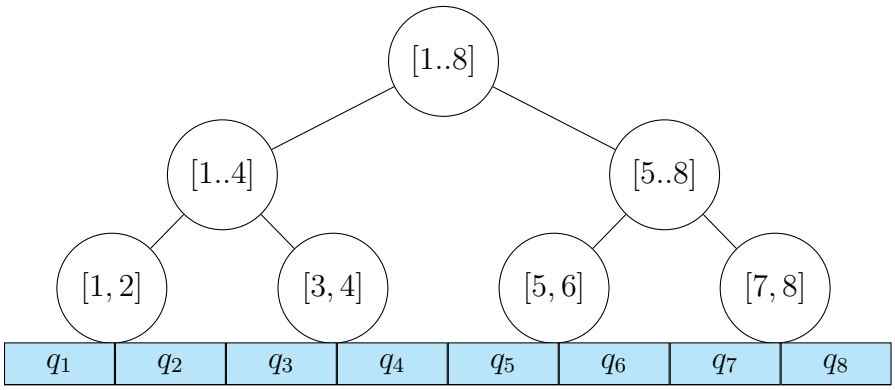


Figure 3.2: A segment tree $T_{8,2}$ and probability distribution $\{q_1, ..., q_8\}$ on $[1, ..., 8]$.

1. *The structure has construction time* $O(IR^2)$ *and storage requirement* $O(R^2\lceil I/F\rceil)$. *If* $I < F$, *the storage requirement drops to* $O(1)$.
2. *After construction, the data structure can produce a sample according to the distribution* $q_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}$ *in time* $O(R^2 \log\lceil I/F\rceil + FR^2)$ *for any vector* $\boldsymbol{h}$.
3. *If* $\boldsymbol{Y}$ *is a rank-1 matrix, the time per sample drops to* $O(R^2 \log\lceil I/F\rceil + FR)$.

This data structure relies on an adaptation of a classic binary-tree inversion sampling technique [Saa+20]. Consider a partition of the interval $[0, 1]$ into I bins, the i-th having width $q_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i]$. We sample $d \sim \text{Uniform}[0, 1]$ and return the index of the containing bin. We locate the bin index through a binary search terminated when at most F bins remain in the search space, which are then scanned in linear time. Here, F is a tuning parameter that we will use to control sampling complexity and space usage.

We can regard the binary search as a walk down a full, complete binary tree $T_{I,F}$ with $\lceil I/F\rceil$ leaves, the nodes of which store contiguous, disjoint segments $S(v) = \{S_0(v)..S_1(v)\} \subseteq [I]$ of size at most F. The segment of each internal node is the union of segments held by its children, and the root node holds $\{1, ..., I\}$. Suppose that the binary search reaches node v with left child $L(v)$ and maintains the interval $[\text{low}, \text{high}] \subseteq [0, 1]$ as the remaining search space to explore. Then the search branches left in the tree iff $d < \text{low} + \sum_{i\in S(L(v))} q_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i]$.

This branching condition can be evaluated efficiently if appropriate information is stored at each node of the segment tree. Excluding the offset "low", the branching threshold takes the form

$$\sum_{i\in S(v)} q_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i] = C^{-1}\langle \boldsymbol{h}\boldsymbol{h}^\top, \sum_{i\in S(v)} \boldsymbol{U}[i,:]^\top \boldsymbol{U}[i,:], \boldsymbol{Y}\rangle := C^{-1}\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{G}^v, \boldsymbol{Y}\rangle. \tag{3.7}$$

Here, we call each matrix $\boldsymbol{G}^v \in \mathbb{R}^{R\times R}$ a *partial Gram matrix*. In time $O(IR^2)$ and space $O(R^2\lceil I/F\rceil)$, we can compute and cache $\boldsymbol{G}^v$ for each node of the tree to construct our data structure. Each subsequent binary search costs $O(R^2)$ time to evaluate Equation (3.7) at each of $\log\lceil I/F\rceil$ internal nodes and $O(FR^2)$ to evaluate $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}$ at the F indices held by each leaf, giving point 2 of the lemma. This cost at each leaf node reduces to $O(FR)$ in case $\boldsymbol{Y}$ is rank-1, giving point 3. A complete proof of this lemma appears in Section 3.6.3.

3.3.2 Sampling from the Khatri-Rao Product

We face difficulties if we directly apply Lemma 3.3.2 to sample from the conditional distribution in Theorem 3.3.1. Because $\boldsymbol{G}_{>k}$ is not rank-1 in general, we must use point 2 of the lemma, where no selection of the parameter F allows us to simultaneously satisfy the space and runtime constraints of Theorem 3.1.1. Selecting $F = R$ results in cost $O(R^3)$ per sample (violating the runtime requirement in point 2), whereas $F = 1$ results in a superlinear storage requirement $O(IR^2)$ (violating the space requirement in point 1, and becoming prohibitively expensive for $I \geq 10^6$). To avoid these extremes, we break the sampling procedure into two stages. The first stage selects a 1-dimensional subspace spanned by an eigenvector of $\boldsymbol{G}_{>k}$, while the second samples according to Theorem 3.3.1 after projecting the relevant vectors onto the selected subspace. Lemma 3.3.2 can be used for *both* stages, and the second stage benefits from point 3 to achieve better time and space complexity.

Below, we abbreviate $\boldsymbol{q} = \boldsymbol{q}_{\boldsymbol{h}_{<k},\boldsymbol{U}_k,\boldsymbol{G}_{>k}}$ and $\boldsymbol{h} = \boldsymbol{h}_{<k}$. When sampling from I_k, observe that $\boldsymbol{G}_{>k}$ is the same for all samples. We compute a symmetric eigendecomposition $\boldsymbol{G}_{>k} = \boldsymbol{V}\boldsymbol{\Lambda}\boldsymbol{V}^\top$, where each column of $\boldsymbol{V}$ is an eigenvector of $\boldsymbol{G}_{>k}$ and $\boldsymbol{\Lambda} = \operatorname{diag}((\lambda_u)_{u=1}^R)$ contains the eigenvalues along the diagonal. This allows us to rewrite entries of $\boldsymbol{q}$ as

$$\boldsymbol{q}[s] = C^{-1}\sum_{u=1}^{R}\lambda_u\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}_k[s,:]^\top \boldsymbol{U}_k[s,:], \boldsymbol{V}[:,u]\,\boldsymbol{V}[:,u]^\top\rangle. \tag{3.8}$$

Define matrix $\boldsymbol{W} \in \mathbb{R}^{I_k\times R}$ elementwise by

$$\boldsymbol{W}[t,u] := \langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}_k[t,:]^\top \boldsymbol{U}_k[t,:], \boldsymbol{V}[:,u]\,\boldsymbol{V}[:,u]^\top\rangle$$

and observe that all of its entries are nonnegative. Since $\lambda_u \geq 0$ for all u ($\boldsymbol{G}_{>k}$ is p.s.d.), we can write $\boldsymbol{q}$ as a mixture of probability distributions given by the normalized columns of $\boldsymbol{W}$:

$$\boldsymbol{q} = \sum_{u=1}^{R}\boldsymbol{w}[u]\,\frac{\boldsymbol{W}[:,u]}{\|\boldsymbol{W}[:,u]\|_1},$$

where the vector $\boldsymbol{w}$ of nonnegative weights is given by $\boldsymbol{w}[u] = (C^{-1}\lambda_u\|\boldsymbol{W}[:,u]\|_1)$. Rewriting $\boldsymbol{q}$ in this form gives us the two stage sampling procedure: first sample a component u of the mixture according to the weight vector $\boldsymbol{w}$, then sample an index in $[I_k]$ according to the

probability vector defined by $\boldsymbol{W}[:,u] / \|\boldsymbol{W}[:,u]\|_1$. Let $\hat{u}_k$ be a random variable distributed according to the probability mass vector w. We have, for C taken from Theorem 3.3.1,

$$\begin{aligned} p(\hat{u}_k = u_k) &= C^{-1}\lambda_{u_k} \sum_{t=1}^{I_k} \boldsymbol{W}[t, u_k] \\ &= C^{-1}\lambda_{u_k} \langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{V}[:,u_k]\,\boldsymbol{V}[:,u_k]^\top, \boldsymbol{G}_k \rangle \\ &= \boldsymbol{q}_{\boldsymbol{h},\sqrt{\boldsymbol{\Lambda}}\boldsymbol{V}^\top,\boldsymbol{G}_k}[u_k]\,. \end{aligned} \tag{3.9}$$

Hence, we can use point 2 of Lemma 3.3.2 to sample a value for $\hat{u}_k$ efficiently. Because $\sqrt{\boldsymbol{\Lambda}}\boldsymbol{V}^\top$ has only R rows with $R \sim 10^2$, we choose the tuning parameter $F = 1$ to achieve lower time per sample while incurring a modest $O(R^3)$ space overhead. Now, introduce a random variable $\hat{t}_k$ with distribution conditioned on $\hat{u}_k = u_k$ given by

$$p(\hat{t}_k = t_k \mid \hat{u}_k = u_k) := \boldsymbol{W}[t_k, u_k] / \|\boldsymbol{W}[:,u_k]\|_1\,. \tag{3.10}$$

This distribution is well-defined, since we suppose that $\hat{u}_k = u_k$ occurs with nonzero probability $e[u_k]$, which implies that $\|\boldsymbol{W}[:,u_k]\|_1 \neq 0$. Our remaining task is to efficiently sample from the distribution above. Below, we abbreviate $\tilde{\boldsymbol{h}} = \boldsymbol{V}[:,u_k] \circledast \boldsymbol{h}$ and derive

$$\begin{aligned} p(\hat{t}_k = t_k \mid \hat{u}_k = u_k) &= \frac{\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}_k[t_k,:]^\top \boldsymbol{U}_k[t_k,:], \boldsymbol{V}[:,u_k]\,\boldsymbol{V}[:,u_k]^\top \rangle}{\|\boldsymbol{W}[:,u_k]\|_1} \\ &= \frac{\langle \tilde{\boldsymbol{h}}\tilde{\boldsymbol{h}}^\top, \boldsymbol{U}_k[t_k,:]^\top \boldsymbol{U}_k[t_k,:], [1] \rangle}{\|\boldsymbol{W}[:,u_k]\|_1} \\ &= \boldsymbol{q}_{\tilde{\boldsymbol{h}},\boldsymbol{U}_k,[1]}[t_k]\,. \end{aligned} \tag{3.11}$$

Based on the last line of Equation (3.11), we apply Lemma 3.3.2 again to build an efficient data structure to sample a row of $\boldsymbol{U}_k$. Since $\boldsymbol{Y} = [1]$ is a rank-1 matrix, we can use point 3 of the lemma and select a larger parameter value $F = R$ to reduce space usage. The sampling time for this stage becomes $O(R^2 \log\lceil I_j/R \rceil)$.

To summarize, Algorithms 5 and 6 give the construction and sampling procedures for our data structure. They rely on the "BuildSampler" and "RowSample" procedures from Algorithms 7 and 8 in Section 3.6.3, which relate to the data structure in Lemma 3.3.2. In the construction phase, we build N data structures from Lemma 3.3.2 for the distribution in Equation (3.11). Construction costs $O\left(\sum_{j=1}^{N} I_j R^2\right)$, and if any matrix $\boldsymbol{U}_j$ changes, we can rebuild Z_j in isolation. Because $F = R$, the space required for Z_j is $O(I_j R)$.

In the sampling phase, Algorithm 6 accepts an optional index j of a matrix to exclude from the Khatri-Rao product. The procedure begins by computing the symmetric eigendecomposition of each matrix $\boldsymbol{G}_{>k}$. The eigendecomposition is computed only once per binary tree structure, and its computation cost is amortized over all J samples. It then creates data structures

E_k for each of the distributions specified by Equation (3.9). These data structures (along with those from the construction phase) are used to draw $\hat{u}_k$ and $\hat{t}_k$ in succession. The random variables $\hat{t}_k$ follow the distribution in Theorem 3.3.1 conditioned on prior draws, so the multi-index $(\hat{t}_k)_{k \neq j}$ follows the leverage score distribution on $\boldsymbol{A}$, as desired. Section 3.6.4 proves the complexity claims in the theorem and provides further details about the algorithms.

Algorithm 5 ConstructKRPSampler($\boldsymbol{U}_1, ..., \boldsymbol{U}_N$)

for $j = 1..N$ **do**
  $Z_j :=$ BuildSampler($\boldsymbol{U}_j, F = R, [1]$)
  $\boldsymbol{G}_j := \boldsymbol{U}_j^\top \boldsymbol{U}_j$

Algorithm 6 KRPSample(j, J)

$\boldsymbol{G} := \circledast_{k \neq j} \boldsymbol{G}_k$
for $k \neq j$ **do**
  $\boldsymbol{G}_{>k} := \boldsymbol{G}^+ \circledast \circledast_{k=j+1}^{N} \boldsymbol{G}_k$
  Decompose $\boldsymbol{G}_{>k} = \boldsymbol{V}_k \boldsymbol{\Lambda}_k \boldsymbol{V}_k^\top$
  $E_k :=$ BuildSampler($\sqrt{\boldsymbol{\Lambda}_k} \cdot \boldsymbol{V}_k^\top, F = 1, \boldsymbol{G}_k$)
for $d = 1..J$ **do**
  $\boldsymbol{h} = [1, ..., 1]^\top$
  for $k \neq j$ **do**
    $\hat{u}_k :=$ RowSample($E_k, \boldsymbol{h}$)
    $\hat{t}_k :=$ RowSample($Z_k, \boldsymbol{h} \circledast (\boldsymbol{V}_k [:, \hat{u}_k])$)
    $\boldsymbol{h} *= \boldsymbol{U}_k \left[\hat{t}_k, :\right]$
  $s_d = (\hat{t}_k)_{k \neq j}$
return $s_1, ..., s_J$

3.3.3 Application to Tensor Decomposition

A tensor is a multidimensional array, and the CP decomposition represents a tensor as a sum of outer products [KB09]. See Section 3.6.7 for an overview. To approximately decompose tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times ... \times I_N}$, the popular alternating least squares (ALS) algorithm begins with randomly initialized factor matrices $\boldsymbol{U}_j$, $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$ for $1 \leq j \leq N$. We call the column count R the **rank** of the decomposition. Each round of ALS solves N overdetermined least squares problems in sequence, each optimizing a single factor matrix while holding the others constant. The j-th least squares problem occurs in the update

$$\boldsymbol{U}_j := \arg\min_{\boldsymbol{X}} \left\| \boldsymbol{U}_{\neq j} \cdot \boldsymbol{X}^\top - \text{mat}(\mathcal{T}, j)^\top \right\|_F$$

where $\boldsymbol{U}_{\neq j} = \boldsymbol{U}_N \odot ... \odot \boldsymbol{U}_{j+1} \odot \boldsymbol{U}_{j-1} \odot ... \odot \boldsymbol{U}_1$ is the Khatri-Rao product of all matrices excluding $\boldsymbol{U}_j$ and mat$(\cdot)$ denotes the mode-j matricization of tensor $\mathcal{T}$. Here, we reverse the

order of matrices in the Khatri-Rao product to match the ordering of rows in the matricized tensor (see Section 3.6.7 for an explicit formula for the matricization). These problems are ideal candidates for randomized sketching [BBK18; JKW20; LK22], and applying the data structure in Theorem 3.1.1 gives us the **STS-CP** algorithm.

Corollary 3.3.3 (STS-CP)**.** *Suppose $\mathcal{T}$ is dense, and suppose we solve each least squares problem in ALS with a randomized sketching algorithm. A leverage score sampling approach as defined in section 3.2 guarantees that with $\tilde{O}(R/(\varepsilon\delta))$ samples per solve, the residual of each sketched least squares problem is within $(1+\varepsilon)$ of the optimal residual with probability $(1-\delta)$. The efficient sampler from Theorem 3.1.1 brings the complexity of ALS to*

$$\tilde{O}\left(\frac{\#it}{\varepsilon\delta}\cdot\sum_{j=1}^{N}\left(NR^3\log I_j + I_jR^2\right)\right)$$

where "#it" is the number of ALS iterations, and with any term $\log I_j$ replaced by $\log R$ if $I_j < R$.

The proof appears in Section 3.6.7 and combines Theorem 3.1.1 with Theorem 3.2.1. STS-CP also works for sparse tensors and likely provides a greater advantage here than the dense case, as sparse tensors tend to have much larger mode size [Smi+17]. The complexity for sparse tensors depends heavily on the sparsity structure and is difficult to predict. Nevertheless, we expect a significant speedup based on prior works that use sketching to accelerate CP decomposition [Che+16; LK22].

3.4 Experiments

Experiments were conducted on CPU nodes of NERSC Perlmutter, an HPE Cray EX supercomputer, and our code is available at `https://github.com/vbharadwaj-bk/fast_tensor_leverage.git`. On tensor decomposition experiments, we compare our algorithms against the random and hybrid versions of CP-ARLS-LEV proposed by Larsen and Kolda [LK22]. These algorithms outperform uniform sampling and row-norm-squared sampling, achieving excellent accuracy and runtime relative to exact ALS. In contrast to TNS-CP and the Gaussian tensor network embedding proposed by Ma and Solomonik (see Table 1), CP-ARLS-LEV is one of the few algorithms that can practically decompose sparse tensors with mode sizes in the millions. In the worst case, CP-ARLS-LEV requires $\tilde{O}(R^{N-1}/(\varepsilon\delta))$ samples per solve for an N-dimensional tensor to achieve solution guarantees like those in Theorem 3.2.1, compared to $\tilde{O}(R/(\varepsilon\delta))$ samples required by STS-CP. Sections 3.6.8, 3.6.9, and 3.6.11 provide configuration details and additional results.

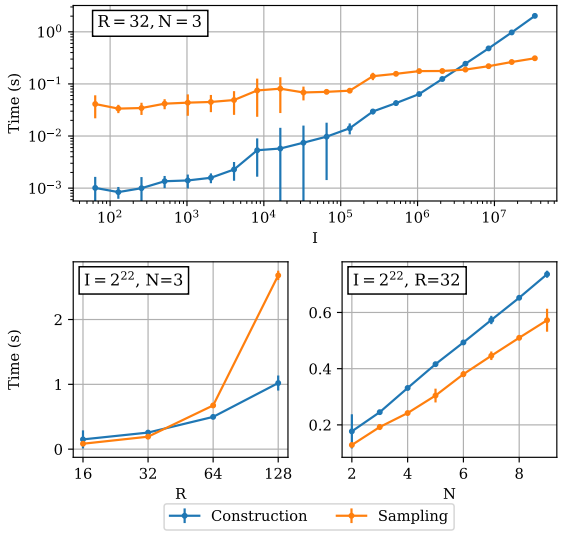


Figure 3.3: Average time (5 trials) to construct our proposed sampler and draw $J = 50,000$ samples from $\boldsymbol{U}_1 \odot \ldots \odot \boldsymbol{U}_N$, with $\boldsymbol{U}_j \in \mathbb{R}^{I \times R}$ $\forall j$. Error bars indicate 3 standard deviations.

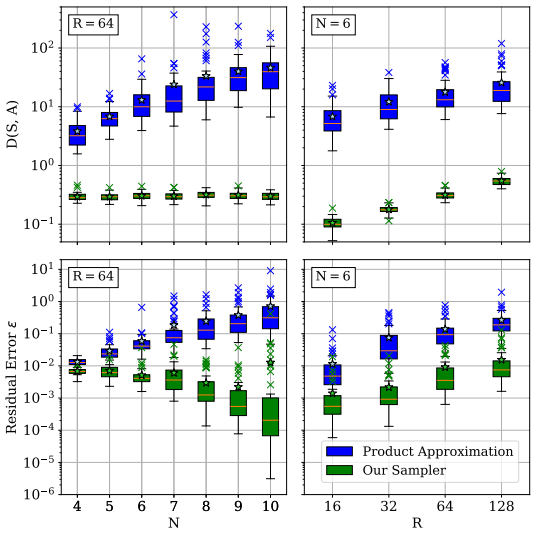


Figure 3.4: $D(\boldsymbol{S}, \boldsymbol{A})$ and residual error (50 trials) for varying R and N on least squares, $I = 2^{16}, J = 5000$. "X" marks indicate outliers 1.5 times the interquartile range beyond the median, stars indicate means.

3.4.1 Runtime Benchmark

Figure 3.3 shows the time to construct our sampler and draw 50,000 samples from the Khatri-Rao product of i.i.d. Gaussian initialized factor matrices. We quantify the runtime impacts of varying N, R, and I. The asymptotic behavior in Theorem 3.1.1 is reflected in our performance measurements, with the exception of the plot that varies R. Here, construction becomes disproportionately cheaper than sampling due to cache-efficient BLAS3 calls during construction. Even when the full Khatri-Rao product has $\approx 3.78 \times 10^{22}$ rows (for $I = 2^{25}, N = 3, R = 32$), we require only 0.31 seconds on average for sampling (top plot, rightmost points).

3.4.2 Least Squares Accuracy Comparison

We now test our sampler on least squares problems of the form $\min_{\boldsymbol{x}} \|\boldsymbol{A}\boldsymbol{x} - \boldsymbol{b}\|$, where $\boldsymbol{A} = \boldsymbol{U}_1 \odot \ldots \odot \boldsymbol{U}_N$ with $\boldsymbol{U}_j \in \mathbb{R}^{I \times R}$ for all j. We initialize all matrices $\boldsymbol{U}_j$ entrywise i.i.d. from a standard normal distribution and randomly multiply 1% of all entries by 10. We choose $\boldsymbol{b}$ as a Kronecker product $\boldsymbol{c}_1 \otimes \ldots \otimes \boldsymbol{c}_N$, with each vector $\boldsymbol{c}_j \in \mathbb{R}^I$ also initialized entrywise from a Gaussian distribution. We assume this form for $\boldsymbol{b}$ to tractably compute the exact solution to the linear least squares problem and evaluate the accuracy of our randomized methods. We **do not** give our algorithms access to the Kronecker form of $\boldsymbol{b}$; they are only permitted

on-demand, black-box access to its entries.

For each problem instance, define

$$D(\boldsymbol{S}, \boldsymbol{A}) = \kappa(\boldsymbol{SQ}) - 1 \tag{3.12}$$

where $\boldsymbol{Q}$ is an orthonormal basis for the column space of $\boldsymbol{A}$ and $\kappa(\boldsymbol{SQ})$ is the condition number of $\boldsymbol{SQ}$. A higher-quality sketch $\boldsymbol{S}$ exhibits a lower condition number $\kappa(\boldsymbol{SQ})$, which quantifies the preservation of distances from the column space of $\boldsymbol{A}$ to the column space of $\boldsymbol{SA}$. Note that this quantity is the numerator of the distortion metric, calculated as $(\kappa(\boldsymbol{SQ})-1)/(\kappa(\boldsymbol{SQ})+1)$ [Mur+23]. We omit the denominator to highlight the rapid condition number growth for the competing sampler. For details about about computing $\kappa(\boldsymbol{SQ})$ efficiently when $\boldsymbol{A}$ is a Khatri-Rao product, see Section 3.6.10. Next, define $\varepsilon = \frac{\text{residual}_{\text{approx}}}{\text{residual}_{\text{opt}}} - 1$, where $\text{residual}_{\text{approx}}$ is the residual of the randomized least squares algorithm. ε is nonnegative and (similar to its role in Theorem 3.2.1) quantifies the quality of the randomized algorithm's solution.

For varying N and R, Figure 3.4 shows the average values of D and ε achieved by our algorithm against the leverage product approximation used by Larsen and Kolda [LK22]. Our sampler exhibits nearly constant $D(\boldsymbol{S}, \boldsymbol{A})$ for fixed rank R and varying N, and it achieves $\varepsilon \approx 10^{-2}$ even when $N = 9$. The product approximation increases both quantities by at least an order of magnitude.

3.4.3 Sparse Tensor Decomposition

We next apply STS-CP to decompose several large sparse tensors from the FROSTT collection [Smi+17] (see Section 3.6.9 for more details on the experimental configuration). Our accuracy metric is the tensor fit. Letting $\tilde{\mathcal{T}}$ be our low-rank CP approximation, the fit with respect to ground-truth tensor $\mathcal{T}$ is $\text{fit}(\tilde{\mathcal{T}}, \mathcal{T}) = 1 - \left\|\tilde{\mathcal{T}} - \mathcal{T}\right\|_F / \|\mathcal{T}\|_F$.

Table 3.2 compares the runtime per ALS round for our algorithm against existing common software packages for sparse tensor CP decomposition. We compared our algorithm against Tensorly version 0.81 [Kos+19] and Matlab Tensor Toolbox version 3.5 [BK08], with dramatic speedups over both.. We compared our algorithm against both non-randomized ALS and a version of CP-ARLS-LEV in Tensor Toolbox.

As demonstrated by Table 3.2, our implementation exhibits more than 1000x speedup over Tensorly and 295x over Tensor Toolbox (non-randomized) for the NELL-2 tensor. STS-CP enjoys a dramatic speedup over Tensorly because the latter explicitly materializes the Khatri-Rao product, which is prohibitively expensive given the large tensor dimensions (see Table 3.3). STS-CP consistently exhibits at least 2.5x speedup over the version of CP-ARLS-LEV in Tensor Toolbox, with more than 10x speedup on the Amazon tensor. To ensure a fair

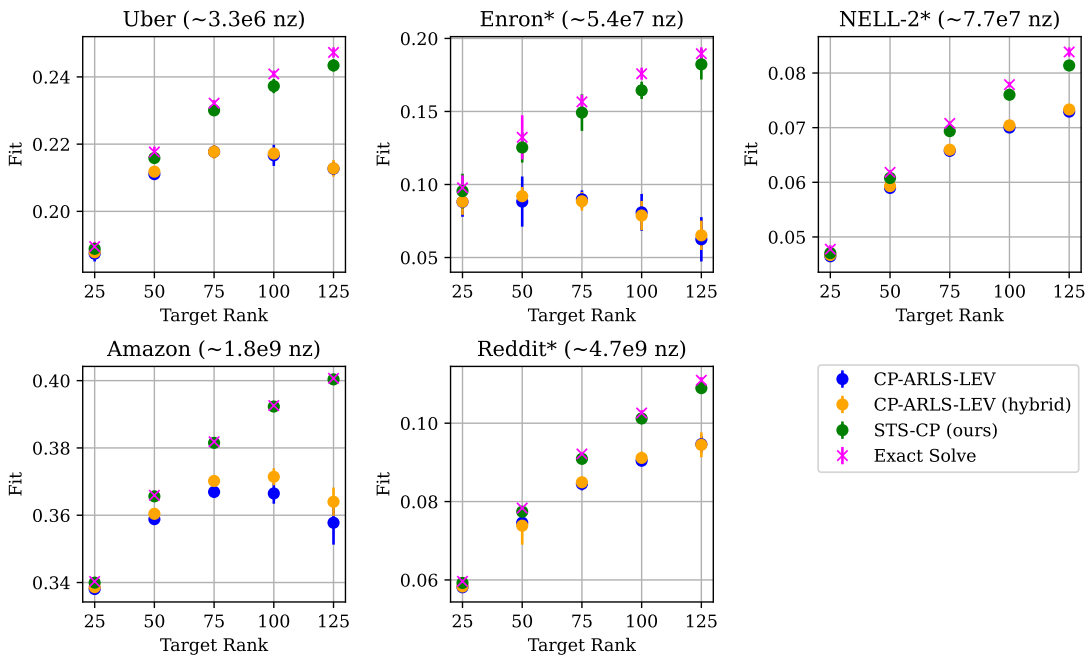


Figure 3.5: Average fits (8 trials) achieved by randomized ($J = 2^{16}$) and exact ALS for sparse tensor CP decomposition. Error bars indicate 3 standard deviations. See Section 3.6.9 for details.

Table 3.2: Average time (seconds) per ALS round for our method vs. standard CP decomposition packages. OOM indicates an out-of-memory error. All experiments were conducted on a single LBNL Perlmutter CPU node. Randomized algorithms were benchmarked with 2^{16} samples per least-squares solve.

Method	Uber	Enron	NELL-2	Amazon	Reddit
Tensorly, Sparse Backend	64.2	OOM	759.6	OOM	OOM
Matlab TToolbox Standard	11.6	294.4	177.4	>3600	OOM
Matlab TToolbox CP-ARLS-LEV	0.5	1.4	1.9	34.2	OOM
STS-CP (ours)	**0.2**	**0.5**	**0.6**	**3.4**	**26.0**

comparison with CP-ARLS-LEV, we wrote an improved implementation in C++ that was used for all other experiments.

As Figure 3.5 shows, the fit achieved by CP-ARLS-LEV compared to STS-CP degrades as the rank increases for fixed sample count. By contrast, STS-CP improves the fit consistently, with a significant improvement at rank 125 over CP-ARLS-LEV. Timings for both algorithms

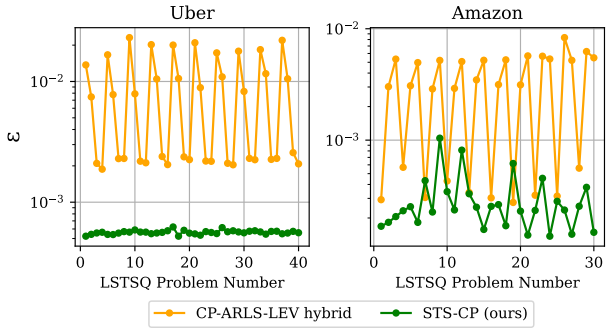


Figure 3.6: Average ε (5 runs) for randomized least squares solves in 10 ALS rounds, $R = 50$.

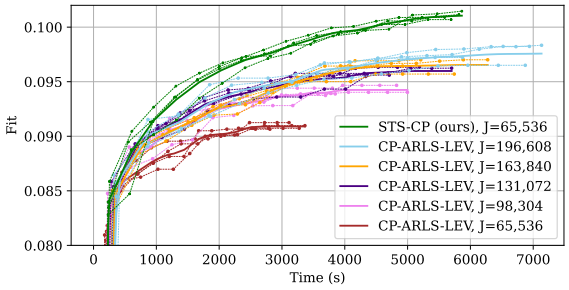


Figure 3.7: Fit vs. time, Reddit tensor, $R = 100$. Thick lines are averages 4 trial interpolations.

are available in Section 3.6.11.4. Figure 3.6 explains the higher fit achieved by our sampler on the Uber and Amazon tensors. In the first 10 rounds of ALS, we compute the exact solution to each least squares problem before updating the factor matrix with a randomized algorithm's solution. Figure 3.6 plots ε as ALS progresses for hybrid CP-ARLS-LEV and STS-CP. The latter consistently achieves lower residual per solve. We further observe that CP-ARLS-LEV exhibits an oscillating error pattern with period matching the number of modes N.

To assess the trade-off between sampling time and accuracy, we compare the fit as a function of ALS update time for STS-CP and random CP-ARLS-LEV in Figure 3.7 (time to compute the fit excluded). On the Reddit tensor with $R = 100$, we compared CP-ARLS-LEV with $J = 2^{16}$ against CP-ARLS-LEV with progressively larger sample count. Even with 2^{18} samples per randomized least squares solve, CP-ARLS-LEV cannot achieve the maximum fit of STS-CP. Furthermore, STS-CP makes progress more quickly than CP-ARLS-LEV. See Section 3.6.11.3 for similar plots for other datasets.

3.5 Discussion and Future Work

Our method for exact Khatri-Rao leverage score sampling enjoys strong theoretical guarantees and practical performance benefits. Especially for massive tensors such as Amazon and Reddit, our randomized algorithm's guarantees translate to faster progress to solution and higher final accuracies. The segment tree approach described here can be applied to sample from tensor networks besides the Khatri-Rao product. In particular, modifications to Lemma 3.3.2 permit efficient leverage sampling from a contraction of 3D tensor cores in ALS tensor train decomposition. We leave the generalization of our fast sampling technique as future work.

3.6 Proofs and Supplementary Results

3.6.1 Further Comparison to Prior Work

In this section, we provide a more detailed comparison of our sampling algorithm with the one proposed by Woodruff and Zandieh [WZ22]. Their work introduces a ridge leverage-score sampling algorithm for Khatri-Rao products with the attractive property that the sketch can be formed in input-sparsity time. For constant failure probability δ, the runtime to produce a $(1 \pm \epsilon)$ ℓ_2-subspace embedding for $\boldsymbol{A} = \boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$ is given in Appendix B of their work (proof of Theorem 2.7). Adapted to our notation, **their runtime** is

$$O\left(\log^4 R \log N \sum_{i=1}^{N} \text{nnz}(U_i) + \frac{N^7 s_\lambda^2 R}{\varepsilon^4} \log^5 R \log N\right)$$

where $s_\lambda = \sum_{i=1}^{R} \frac{\lambda_i}{\lambda_i + \lambda}$, $\lambda_1, ..., \lambda_R$ are the eigenvalues of the Gram matrix G of matrix A, and $\lambda \geq 0$ is a regularization parameter. For comparison, **our runtime** for constant failure probability δ is

$$O\left(R \sum_{i=1}^{N} \text{nnz}(U_i) + \frac{R^3}{\varepsilon} \log\left(\prod_{i=1}^{N} I_i\right) \log R\right).$$

Woodruff and Zandieh's method provides a significant advantage for large column count R or high regularization parameter λ. As a result, it is well-suited to the problem of regularized low-rank approximation when the column count R is given by the number of data points in a dataset. On the other hand, the algorithm has poor dependence on the matrix count N and error parameter ε. For tensor decomposition, R is typically no larger than a few hundred, while high accuracy ($\epsilon \approx 10^{-3}$) is required for certain tensors to achieve a fit competitive with non-randomized methods (see section 3.4.3, Figures 3.5 and 3.6). When λ is small, we have $s_\lambda \approx R$. Here, Woodruff and Zandieh's runtime has an $O(R^3)$ dependence similar to ours. When $R \leq \log^4 R \log N$, our sampler has faster construction time as well.

Finally, we note that our sampling data structure can be constructed using highly cache-efficient, parallelizable symmetric rank-R updates (BLAS3 operation `dSYRK`). As a result, the quadratic dependence on R in our algorithm can be mitigated by dense linear algebra accelerators, including GPUs.

3.6.2 Proof of Theorem 3.3.1

Theorem 3.3.1 appeared in a modified form as Lemma 10 in the work by Malik [Mal22]. This original version used the definition $\tilde{\boldsymbol{G}_{>k}} = \boldsymbol{\Phi} \circledast \circledast_{a=k+1}^{N} \boldsymbol{G}_k$ in place of $\boldsymbol{G}_{>k}$ defined in Equation (3.4), where $\boldsymbol{\Phi}$ was a sketched approximation of $\boldsymbol{G}^+$. Woodruff and Zandieh [WZ22] exhibit a version of the theorem with similar modifications. We prove the version stated in our work below.

Proof of Theorem 3.3.1. We rely on the assumption that the Khatri-Rao product $\boldsymbol{A}$ is a nonzero matrix (but it may be rank-deficient). We begin by simplifying the expression for the leverage score of a row of A corresponding to multi-index $(i_1, ..., i_N)$. Starting with Equation (3.2), we derive

$$\begin{aligned}
&\ell_{i_1,...,i_N} \\
&= \boldsymbol{A}\left[(i_1, ..., i_N), :\right] \boldsymbol{G}^{+} \boldsymbol{A}\left[(i_1, ..., i_N), :\right]^{\top} \\
&= \langle \boldsymbol{A}\left[(i_1, ..., i_N), :\right]^{\top} \boldsymbol{A}\left[(i_1, ..., i_N), :\right], \boldsymbol{G}^{+} \rangle \\
&= \langle \left(\circledast_{a=1}^{N} U_a\left[i_a, :\right] \right)^{\top} \left(\circledast_{a=1}^{N} \boldsymbol{U}_a\left[i_a, :\right] \right), \boldsymbol{G}^{+} \rangle \\
&= \langle \circledast_{a=1}^{N} \boldsymbol{U}_a\left[i_a, :\right]^{\top} \boldsymbol{U}_a\left[i_a, :\right], \boldsymbol{G}^{+} \rangle \\
&= \langle \circledast_{a=1}^{k-1} \boldsymbol{U}_a\left[i_a, :\right]^{\top} \boldsymbol{U}_a\left[i_a, :\right], \boldsymbol{U}_k\left[i_k, :\right]^{\top} \boldsymbol{U}_k\left[i_k, :\right] \circledast \circledast_{a=k+1}^{N} \boldsymbol{U}_a\left[i_a, :\right]^{\top} \boldsymbol{U}_a\left[i_a, :\right], \boldsymbol{G}^{+} \rangle \\
&= \langle \circledast_{a=1}^{k-1} \boldsymbol{U}_a\left[i_a, :\right]^{\top} \boldsymbol{U}_a\left[i_a, :\right], \boldsymbol{U}_k\left[i_k, :\right]^{\top} \boldsymbol{U}_k\left[i_k, :\right], \boldsymbol{G}^{+} \circledast \circledast_{a=k+1}^{N} \boldsymbol{U}_a\left[i_a, :\right]^{\top} \boldsymbol{U}_a\left[i_a, :\right] \rangle.
\end{aligned} \tag{3.13}$$

We proceed to the main proof. To compute $p(\hat{s}_k = s_k \mid \hat{s}_{<k} = s_{<k})$, we marginalize over random variables $\hat{s}_{k+1} ... \hat{s}_N$. Using the definition of $\boldsymbol{h}_{<k}$ from Equation (3.3), we have

$$\begin{aligned}
p(\hat{s}_k = s_k \mid \hat{s}_{<k} = s_{<k}) &\propto \sum_{i_{k+1},...,i_N} p\left((\hat{s}_{<k} = s_{<k}) \wedge (\hat{s}_k = s_k) \wedge \bigwedge_{u=k+1}^{N} (\hat{s}_u = i_u) \right) \\
&\propto \sum_{i_{k+1},...,i_N} \ell_{s_1,...,s_k,i_{k+1},...,i_N}.
\end{aligned} \tag{3.14}$$

The first line above follows by marginalizing over $\hat{s}_{k+1}, ..., \hat{s}_N$. The second line follows because the joint random variable $(\hat{s}_1, ..., \hat{s}_N)$ follows the distribution of statistical leverage scores on

the rows of A. We now plug in Equation (3.13) to get

$$\begin{aligned}
&\sum_{i_{k+1},\ldots,i_N} \ell_{s_1,\ldots,s_k,i_{k+1},\ldots,i_N} \\
&= \sum_{i_{k+1},\ldots,i_N} \langle \circledast_{a=1}^{k-1} \boldsymbol{U}_a[s_a,:]^\top \boldsymbol{U}_a[s_a,:], \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}^+ \circledast \circledast_{a=k+1}^{N} \boldsymbol{U}_a[i_a,:]^\top \boldsymbol{U}_a[i_a,:] \rangle \\
&= \sum_{i_{k+1},\ldots,i_N} \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}^+ \circledast \circledast_{a=k+1}^{N} \boldsymbol{U}_a[i_a,:]^\top \boldsymbol{U}_a[i_a,:] \rangle \\
&= \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}^+ \circledast \circledast_{a=k+1}^{N} \sum_{i_a=1}^{I_a} \boldsymbol{U}_a[i_a,:]^\top \boldsymbol{U}_a[i_a,:] \rangle \\
&= \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}^+ \circledast \circledast_{a=k+1}^{N} \boldsymbol{G}_a \rangle \\
&= \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}_{>k} \rangle.
\end{aligned} \tag{3.15}$$

We compute the normalization constant C for the distribution by summing the last line of Equation (3.15) over all possible values for $\hat{s}_k$:

$$\begin{aligned}
C &= \sum_{s_k=1}^{I_k} \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}_{>k} \rangle \\
&= \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \sum_{s_k=1}^{I_k} \boldsymbol{U}_k[s_k,:]^\top \boldsymbol{U}_k[s_k,:], \boldsymbol{G}_{>k} \rangle \\
&= \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{G}_k, \boldsymbol{G}_{>k} \rangle.
\end{aligned} \tag{3.16}$$

For $k = 1$, we have $h_{<k} = [1, ..., 1]^\top$, so $C = \langle \boldsymbol{G}_k, \boldsymbol{G}_{>k} \rangle$. Then C is the sum of all leverage scores, which is known to be the rank of A [Woo+14]. Since A was assumed nonzero, $C \neq 0$. For $k > 1$, assume that the conditioning event $\hat{s}_{<k} = s_{<k}$ occurs with nonzero probability. This is a reasonable assumption, since our sampling algorithm will never select prior values $\hat{s}_1, ..., \hat{s}_{k-1}$ that have 0 probability of occurrence. Let $\tilde{C}$ be the normalization constant for the conditional distribution on $\hat{s}_{k-1}$. Then we have

$$\begin{aligned}
0 &< p(\hat{s}_{k-1} = s_{k-1} \mid \hat{s}_{<k-1} = s_{<k-1}) \\
&= \tilde{C}^{-1} \langle \boldsymbol{h}_{<k-1}\boldsymbol{h}_{\boldsymbol{<k-1}}^\top, \boldsymbol{U}_{k-1}[s_{k-1},:]^\top \boldsymbol{U}_{k-1}[s_{k-1},:], \boldsymbol{G}_{>k-1} \rangle \\
&= \tilde{C}^{-1} \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{G}_{>k-1} \rangle \\
&= \tilde{C}^{-1} \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{G}_k \circledast \boldsymbol{G}_{>k} \rangle \\
&= \tilde{C}^{-1} \langle \boldsymbol{h}_{<k}\boldsymbol{h}_{<k}^\top, \boldsymbol{G}_k, \boldsymbol{G}_{>k} \rangle \\
&= \tilde{C}^{-1} C
\end{aligned} \tag{3.17}$$

Since $\tilde{C} > 0$, we must have $C > 0$. □

3.6.3 Proof of Lemma 3.3.2

We detail the construction procedure, sampling procedure, and correctness of our proposed data structure. Recall that $T_{I,F}$ denotes the collection of nodes in a full, complete binary tree with $\lceil I/F \rceil$ leaves. Each leaf $v \in T_{I,F}$ holds a segment $S(v) = \{S_0(v)..S_1(v)\} \subseteq \{1..I\}$, with $|S(v)| \leq F$ and $S(u) \cap S(v) = \varnothing$ for distinct leaves u, v. For each internal node v, $S(v) = S(L(v)) \cup S(R(v))$, where $L(v)$ and $R(v)$ denote the left and right children of node v. The root node r satisfies $S(r) = \{1..I\}$.

Construction: Algorithm 7 gives the procedure to build the data structure. We initialize a segment tree $T_{I,F}$ and compute $\boldsymbol{G}^v$ for all leaf nodes $v \in T_{I,F}$ as a sum of outer products of rows from U (lines 1-3). Starting at the level above the leaves, we then compute $\boldsymbol{G}^v$ for each internal node as the sum of $\boldsymbol{G}^{L(v)}$ and $\boldsymbol{G}^{R(v)}$, the partial Gram matrices of its two children. Runtime $O(IR^2)$ is required to compute I outer products across all iterations of the loop on line 3. Our segment tree has $\lceil I/F \rceil - 1$ internal nodes, and the addition in line 6 contributes runtime $O(R^2)$ for each internal node. This adds complexity $O(R^2(\lceil I/F \rceil - 1)) \leq O(IR^2)$, for total construction time $O(IR^2)$.

To analyze the space complexity, observe that we store a matrix $\boldsymbol{G}^v \in \mathbb{R}^{R \times R}$ at all $2\lceil I/F \rceil - 1$ nodes of the segment tree, for asymptotic space usage $O(\lceil I/F \rceil R^2)$. We can cut the space usage in half by only storing $\boldsymbol{G}^v$ when v is either the root or a left child in our tree, since the sampling procedure in Algorithm 8 only accesses the partial Gram matrix stored by left children. We can cut the space usage in half again by only storing the upper triangle of each symmetric matrix $\boldsymbol{G}^v$. Finally, in the special case that $I < F$, the segment tree has depth 1 and the initial binary search can be eliminated entirely. As a result, the data structure has $O(1)$ space overhead, since we can avoid storing any partial Gram matrices $\boldsymbol{G}^v$. This proves the complexity claims in point 1 of Lemma 3.3.2.

Algorithm 7 BuildSampler($\boldsymbol{U} \in \mathbb{R}^{I \times R}$, F, $\boldsymbol{Y}$)

1: Build tree $T_{I,F}$ with depth $d = \lceil \log \lceil I/F \rceil \rceil$
2: **for** $v \in$ leaves$(T_{I,F})$ **do**
3: $\quad \boldsymbol{G}^v := \sum_{i \in S(v)} \boldsymbol{U}[i,:]^\top \boldsymbol{U}[i,:]$
4: **for** $u = d - 2 ... 0$ **do**
5: $\quad$ **for** $v \in$ level$(T_{I,F}, u)$ **do**
6: $\quad\quad \boldsymbol{G}^v := \boldsymbol{G}^{L(v)} + \boldsymbol{G}^{R(v)}$

Sampling: Algorithm 8 gives the procedure to draw a sample from our proposed data structure. It is easy to verify that the normalization constant C for $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}$ is $\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{G}^{\text{root}(T_{I,F})}, \boldsymbol{Y} \rangle$, since $\boldsymbol{G}^{\text{root}(T_{I,F})} = \boldsymbol{U}^\top \boldsymbol{U}$. Lines 8 and 9 initialize a pair of templated procedures $\tilde{m}$ and

$\tilde{\boldsymbol{q}}$, each of which accepts a node from the segment tree. The former is used to compute the branching threshold at each internal node, and the latter returns the probability vector $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[S_0(v) : S_1(v)]$ for the segment $\{S_0(v)..S_1(v)\}$ maintained by a leaf node. To see this last fact, observe for $i \in [I]$ that

$$\begin{aligned}
&\tilde{\boldsymbol{q}}(v)[i - S_0(v)] \\
&= C^{-1}\boldsymbol{U}[i,:] \cdot (\boldsymbol{h}\boldsymbol{h}^\top \circledast \boldsymbol{Y}) \cdot \boldsymbol{U}[i,:]^\top \\
&= C^{-1}\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{U}[i,:]^\top \boldsymbol{U}[i,:], \boldsymbol{Y}\rangle \\
&= \boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i].
\end{aligned} \tag{3.18}$$

The loop on line 12 performs the binary search using the two templated procedures. Line 18 uses the procedure $\tilde{\boldsymbol{q}}$ to scan through at most F bin endpoints after the binary search finishes early.

The depth of segment tree $T_{I,F}$ is $\log\lceil I/F\rceil$. As a result, the runtime of the sampling procedure is dominated by $\log\lceil I/F\rceil$ evaluations of $\tilde{m}$ and a single evaluation of $\tilde{\boldsymbol{q}}$ during the binary search. Each execution of procedure $\tilde{m}$ requires time $O(R^2)$, relying on the partial Gram matrices G^v computed during the construction phase. When $\boldsymbol{Y}$ is a general p.s.d. matrix, the runtime of $\tilde{\boldsymbol{q}}$ is $O(FR^2)$. This complexity is dominated by the matrix multiplication $\boldsymbol{W} \cdot (\boldsymbol{h}\boldsymbol{h}^\top \circledast \boldsymbol{Y})$ on line 5. In this case, the runtime of the "RowSampler" procedure to draw one sample is $O(R^2 \log\lceil I/F\rceil + FR^2)$, satisfying the complexity claims in point 2 of the lemma.

Now suppose $\boldsymbol{Y}$ is a rank-1 matrix with $Y = \boldsymbol{u}\boldsymbol{u}^\top$ for some vector $\boldsymbol{u}$. We have $\boldsymbol{h}\boldsymbol{h}^\top \circledast \boldsymbol{Y} = (\boldsymbol{h} \circledast \boldsymbol{u})(\boldsymbol{h} \circledast \boldsymbol{u})^\top$. This gives

$$\tilde{\boldsymbol{q}}_p(\boldsymbol{h}, C, v) = \mathrm{diag}(\boldsymbol{W} \cdot (\boldsymbol{h}\boldsymbol{h}^\top \circledast \boldsymbol{u}\boldsymbol{u}^\top) \cdot \boldsymbol{W}) = (\boldsymbol{W} \cdot (\boldsymbol{h} \circledast \boldsymbol{u}))^2$$

where the square is elementwise. The runtime of the procedure $\tilde{q}$ is now dominated by a matrix-vector multiplication that costs time $O(FR)$. In this case, we have per-sample complexity $O(R^2 \log\lceil I/F\rceil + FR)$, matching the complexity claim in point 3 of the lemma.

Correctness: Recall that the inversion sampling procedure partitions the interval $[0, 1]$ into I bins, the i-th bin having width $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i]$. The goal of our procedure is to find the bin that contains the uniform random draw d. Since procedure $\tilde{m}$ correctly returns the branching threshold (up to the offset "low") given by Equation (3.7), the loop on line 12 correctly implements a binary search on the list of bin endpoints specified by the vector $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}$. At the end of the loop, c is a leaf node that maintains a collection $S(c)$ of bins, one of which contains the random draw d. Since the procedure $\tilde{\boldsymbol{q}}$ correctly returns probabilities $\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[i]$ for $i \in S(c)$ for leaf node c, (see Equation (3.18)), line 18 finds the bin that contains the random draw d. The correctness of the procedure follows from the correctness of inversion sampling [Saa+20]. □

Algorithm 8 Row Sampling Procedure

Require: Matrices $\boldsymbol{U}, \boldsymbol{Y}$ saved from construction, partial Gram matrices $\{\boldsymbol{G}^v \mid v \in T_{I,F}\}$.

1: **procedure** $m_p(\boldsymbol{h}, C, v)$
2: **return** $C^{-1}\langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{G}^v, \boldsymbol{Y}\rangle$
3: **procedure** $\boldsymbol{q}_p(\boldsymbol{h}, C, v)$
4: $\boldsymbol{W} := \boldsymbol{U}[S(v), :]$
5: **return** $C^{-1}\mathrm{diag}(\boldsymbol{W} \cdot (\boldsymbol{h}\boldsymbol{h}^\top \circledast \boldsymbol{Y}) \cdot \boldsymbol{W}^\top)$
6: **procedure** RowSample(h)
7: $C := \langle \boldsymbol{h}\boldsymbol{h}^\top, \boldsymbol{G}^{\mathrm{root}(T_{I,F})}, \boldsymbol{Y}\rangle$
8: $\tilde{m}(\cdot) := m_p(\boldsymbol{h}, C, \cdot)$
9: $\tilde{\boldsymbol{q}}(\cdot) := \boldsymbol{q}_p(\boldsymbol{h}, C, \cdot)$
10: $c := \mathrm{root}(T_{I,F}), \mathrm{low} = 0.0, \mathrm{high} = 1.0$
11: Sample $d \sim \mathrm{Uniform}(0.0, 1.0)$
12: **while** $c \notin \mathrm{leaves}(T_{I,F})$ **do**
13: $\mathrm{cutoff} := \mathrm{low} + \tilde{m}(L(c))$
14: **if** $\mathrm{cutoff} \geq d$ **then**
15: $c := L(c), \mathrm{high} := \mathrm{cutoff}$
16: **else**
17: $c := R(c), \mathrm{low} := \mathrm{cutoff}$
18: **return** $S_0(v) + \arg\min_{i \geq 0} \left(\mathrm{low} + \sum_{j=1}^{i} \tilde{\boldsymbol{q}}(c)[j] < d\right)$

3.6.4 Cohesive Proof of Theorem 3.1.1

In this proof, we fully explain Algorithms 5 and 6 in the context of the sampling procedure outlined in section 3.3.2. We verify the complexity claims first and then prove correctness.

Construction and Update: For each matrix $\boldsymbol{U}_j$, Algorithm 5 builds an efficient row sampling data structure Z_j as specified by Lemma 3.3.2. We let the p.s.d. matrix $\boldsymbol{Y}$ that parameterizes each sampler be a matrix of ones, and we set $F = R$. From Lemma 3.3.2, the time to construct sampler Z_j is $O(I_j R^2)$. The space used by sampler Z_j is $O(\lceil I_j/F \rceil R^2) = O(I_j R)$, since $F = R$. In case $I_j < R$, we use the special case described in Section 3.6.3 to get a space overhead $O(1)$, avoiding a term $O(R^2)$ in the space complexity.

Summing the time and space complexities over all j proves part 1 of the theorem. To update the data structure if matrix $\boldsymbol{U}_j$ changes, we only need to rebuild sampler Z_j for a cost of $O(I_j R^2)$. The construction phase also computes and stores the Gram matrix $\boldsymbol{G}_j$ for each matrix $\boldsymbol{U}_j$. We defer the update procedure in case a single entry of matrix $\boldsymbol{U}_j$ changes to Section 3.6.5.

Sampling: For all indices k (except possibly j), lines 1-5 from Algorithm 6 compute $\boldsymbol{G}_{>k}$

and its eigendecomposition. Only a single pass over the Gram matrices $\boldsymbol{G}_k$ is needed, so these steps cost $O(R^3)$ for each index k. Line 5 builds an efficient row sampler E_k for the matrix of scaled eigenvectors $\sqrt{\boldsymbol{\Lambda}_k} \cdot \boldsymbol{V}_k$. For sampler k, we set $\boldsymbol{Y} = \boldsymbol{G}_k$ with cutoff parameter $F = 1$. From Lemma 3.3.2, the construction cost is $O(R^3)$ for each index k, and the space required by each sampler is $O(R^3)$. Summing these quantities over all $k \neq j$ gives asymptotic runtime $O(NR^3)$ for lines 2-5.

The loop spanning lines 6-12 draws J row indices from the Khatri-Rao product $\boldsymbol{U}_{\neq j}$. For each sample, we maintain a "history vector" $\boldsymbol{h}$ to write the variables $\boldsymbol{h}_{<k}$ from Equation (3.3). For each index $k \neq j$, we draw random variable $\hat{u}_k$ using the row sampler E_k. This random draw indexes a scaled eigenvector of $\boldsymbol{G}_{>k}$. We then use the history vector $\boldsymbol{h}$ multiplied by the eigenvector to sample a row index $\hat{t}_k$ using data structure Z_k. The history vector $\boldsymbol{h}$ is updated, and we proceed to draw the next index $\hat{t}_k$.

As written, lines 2-5 also incur scratch space usage $O(NR^3)$. The scratch space can be reduced to $O(R^3)$ by exchanging the order of loops on line 6 and line 8 and allocating J separate history vectors $\boldsymbol{h}$, once for each draw. Under this reordering, we perform all J draws for each variable $\hat{u}_k$ and $\hat{t}_k$ before moving to $\hat{u}_{k+1}$ and $\hat{t}_{k+1}$. In this case, only a single data structure E_k is required at each iteration of the outer loop, and we can avoid building all the structures in advance on line 5. We keep the algorithm in the form written for simplicity, but we implemented the memory-saving approach in our code.

From Lemma 3.3.2, lines 9 and 10 cost $O(R^2 \log R)$ and $O\left(R^2 \log \lceil I_k/R \rceil\right)$, respectively. Line 11 costs $O(R)$ and contributes a lower-order term. Summing over all $k \neq j$, the runtime to draw a single sample is

$$O\left(\sum_{k \neq j} (R^2 \log \lceil I_k/R \rceil + R^2 \log R)\right) = O\left(\sum_{k \neq j} R^2 \log \max\left(I_k, R\right)\right).$$

Adding the runtime for all J samples to the runtime of the loop spanning lines 2-6 gives runtime $O\left(NR^3 + J \sum_{k \neq j} R^2 \log \max\left(I_k, R\right)\right)$, and the complexity claims have been proven.

Correctness: We show correctness for the case where $j = -1$ and we sample from the Khatri-Rao product of all matrices $\boldsymbol{U}_k$, since the proof for any other value of j requires a simple reindexing. To show that our sampler is correct, it is enough to prove for $1 \leq k \leq N$,

$$p(\hat{t}_k = t_k \mid \boldsymbol{h}_{<k}) = \boldsymbol{q}_{\boldsymbol{h}_{<k}, \boldsymbol{U}_k, \boldsymbol{G}_{>k}}\left[t_k\right], \tag{3.19}$$

since, by Theorem 3.3.1, $p(\hat{s}_k = s_k \mid \hat{s}_{<k} = s_{<k}) = \boldsymbol{q}_{\boldsymbol{h}_{<k}, \boldsymbol{U}_k, \boldsymbol{G}_{>k}}\left[s_k\right]$. This would imply that the joint random variable $(\hat{t}_1, ..., \hat{t}_N)$ has the same probability distribution as $(\hat{s}_1, ..., \hat{s}_N)$, which by definition follows the leverage score distribution on $\boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$. To prove the condition

in Equation (3.19), we apply Equations (3.9) and (3.11) derived earlier:

$$
\begin{aligned}
&p(\hat{t}_k = t_k \mid \boldsymbol{h}_{<k}) \\
&\quad = \sum_{u_k=1}^{R} p(\hat{t}_k = t_k \mid \hat{u}_k = u_k, \boldsymbol{h}_{<k}) p(\hat{u}_k = u_k \mid \boldsymbol{h}_{<k}) && \text{(Bayes' Rule)} \\
&\quad = \sum_{u_k=1}^{R} \boldsymbol{w}[u_k] \frac{\boldsymbol{W}[t_k, u_k]}{\|\boldsymbol{W}[:, u_k]\|_1} && \text{(Equations (3.9) and (3.11), in reverse)} \\
&\quad = \boldsymbol{q}_{\boldsymbol{h}_{<k}, \boldsymbol{U}_k, \boldsymbol{G}_{>k}}[t_k] .
\end{aligned}
\tag{3.20}
$$

□

3.6.5 Efficient Single-Element Updates

Applications such as CP decomposition typically change all entries of a single matrix $\boldsymbol{U}_j$ between iterations, incurring an update cost $O(I_j R^2)$ for our data structure from Theorem 3.1.1. In case only a single element of $\boldsymbol{U}_j$ changes, our data structure can be updated in time $O(R \log I_j)$.

Proof. Algorithm 9 gives the procedure when the update $\boldsymbol{U}_j[r, c] := \hat{u}$ is performed. The matrices $\boldsymbol{G}^v$ refer to the partial Gram matrices maintained by each node v of the segment trees in our data structure, and the matrix $\tilde{\boldsymbol{U}}_j$ refers to the matrix $\boldsymbol{U}_j$ before the update operation.

Algorithm 9 UpdateSampler$(j, r, c, \hat{u})$

1: Let $u = \tilde{\boldsymbol{U}}_j[r, c]$
2: Locate v such that $r \in S(v)$
3: Update $\boldsymbol{G}^v[c, :] \mathrel{+}= (\hat{u} - u)\tilde{\boldsymbol{U}}_j[r, :]$
4: Update $\boldsymbol{G}^v[:, c] \mathrel{+}= (\hat{u} - u)\tilde{\boldsymbol{U}}_j[r, :]^\top$
5: Update $\boldsymbol{G}^v[c, c] \mathrel{+}= (\hat{u} - u)^2$
6: **while** $v \neq \text{root}(T_{I_j,R})$ **do**
7: $\quad v_{\text{prev}} := v$, $v := A(v)$
8: $\quad$ Update $\boldsymbol{G}^v := \boldsymbol{G}^{\text{sibling}(v_{\text{prev}})} + \boldsymbol{G}^{v_{\text{prev}}}$

Let $T_{I_j,R}$ be the segment tree corresponding to matrix U_j in the data structure, and let $v \in T_{I_j,R}$ be the leaf whose segment contains r. Lines 3-5 of the algorithm update the row and column indexed by c in the partial Gram matrix held by the leaf node.

The only other nodes requiring an update are ancestors of v, each holding a partial Gram matrix that is the sum of its two children. Starting from the direct parent $A(v)$, the loop

on line 6 performs these ancestor updates. The addition on line 8 only requires time $O(R)$, since only row and column c change between the old value of $\boldsymbol{G}^v$ and its updated version. Thus, the runtime of this procedure is $O(R \log\lceil I_j/R \rceil)$ from multiplying the cost to update a single node by the depth of the tree. □

3.6.6 Extension to Sparse Input Matrices

Our data structure is designed to sample from Khatri-Rao products $\boldsymbol{U}_1 \odot ... \odot \boldsymbol{U}_N$ where the input matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ are dense, a typical situation in tensor decomposition. Slight modifications to the construction procedure permit our data structure to handle sparse matrices efficiently as well. The following corollary states the result as a modification to Theorem 3.1.1.

Corollary 3.6.1 (Sparse Input Modification)**.** *When input matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ are sparse, point 1 of Theorem 3.1.1 can be modified so that the proposed data structure has*

$$O\left(R \sum_{j=1}^{N} nnz(\boldsymbol{U}_j)\right)$$

construction time and

$$O\left(\sum_{j=1}^{N} nnz(\boldsymbol{U}_j)\right)$$

storage space. The sampling time and scratch space usage in point 2 of Theorem 3.1.1 does not change. The single-element update time in point 1 is likewise unchanged.

Proof. We will modify the data structure in Lemma 3.3.2. The changes to its construction and storage costs will propagate to our Khatri-Rao product sampler, which maintains one of these data structures for each input matrix.

Let us restrict ourselves to the case $F = R, \boldsymbol{Y} = [1]$ in relation to the data structure in Lemma 3.3.2. These choices for F and $\boldsymbol{Y}$ are used in the construction phase given by Algorithm 5. The proof in Section 3.6.3 constrains each leaf v of a segment tree $T_{I,F}$ to hold a contiguous segment $S(v) \subseteq [I]$ of cardinality at most F. Instead, choose each segment $S(v) = \{S_0(v)..S_1(v)\}$ so that $\boldsymbol{U}[S_0(v) : S_1(v), :]$ has at most R^2 nonzero elements, and the leaf count of the tree is at most $\lceil \text{nnz}(\boldsymbol{U})/R^2 \rceil + 1$ for input matrix $\boldsymbol{U} \in \mathbb{R}^{I \times R}$. Assuming the nonzero entries of $\boldsymbol{U}$ are sorted in row-major order, we can construct such a partition of $[I]$ into segments in time $O(\text{nnz}(\boldsymbol{U}))$ by iterating in order through the nonzero rows and adding each of them to a "current" segment. We shift to a new segment when the current segment cannot hold any more nonzero elements.

This completes the modification to the data structure in Lemma 3.3.2, and we now analyze its updated time / space complexity.

Updated Construction / Update Complexity of Lemma 3.3.2, $F = R, \boldsymbol{Y} = [1]$: Algorithm 7 constructs the partial Gram matrix for each leaf node v in the segment tree. Each nonzero in the segment $\boldsymbol{U}[S_0(v) : S_1(v), :]$ contributes time $O(R)$ during line 3 of Algorithm 7 to update a single row and column of $\boldsymbol{G}^v$. Summed over all leaves, the cost of line 3 is $O(\text{nnz}(\boldsymbol{U})R)$. The remainder of the construction procedure updates the partial Gram matrices of all internal nodes. Since there are at most $O\left(\lceil \text{nnz}(\boldsymbol{U})/R^2 \rceil\right)$ internal nodes and the addition on line 6 costs $O(R^2)$ per node, the remaining steps of the construction procedure cost $O(\text{nnz}(\boldsymbol{U}))$, a lower-order term. The construction time is therefore $O(\text{nnz}(\boldsymbol{U})R)$.

Since we store a single partial Gram matrix of size R^2 at each of $O\left(\lceil \text{nnz}(\boldsymbol{U})/R^2 \rceil\right)$ internal nodes, the space complexity of our modified data structure is $O(\text{nnz}(\boldsymbol{U}))$.

Finally, the data structure update time in case a single element of $\boldsymbol{U}$ is modified does not change from Theorem 3.1.1. Since the depth of the segment tree $\lceil \text{nnz}(\boldsymbol{U})/R^2 \rceil + 1$ is upper-bounded by $\lceil I/R \rceil + 1$, the runtime of the update procedure in Algorithm 9 stays the same.

Updated Sampling Complexity of Lemma 3.3.2, $F = R, \boldsymbol{Y} = [1]$: The procedure "RowSample" in Algorithm 8 now conducts a traversal of a tree of depth $O(\lceil \text{nnz}(\boldsymbol{U})/R^2 \rceil)$. As a result, we can still upper-bound the number of calls to procedure $\tilde{m}$ as $\lceil I/F \rceil$. The runtime of procedure $\tilde{m}$ is unchanged. The runtime of procedure $\tilde{\boldsymbol{q}}$ for leaf node c is dominated by the matrix-vector multiplication $\boldsymbol{U}[S_0(c) : S_1(c), :] \cdot \boldsymbol{h}$. This runtime is $O\left(\text{nnz}\left(\boldsymbol{U}[S_0(c) : S_1(c), :]\right)\right) \leq O\left(R^2\right)$. Putting these facts together, the sampling complexity of the data structure in Lemma 3.3.2 does not change under our proposed modifications for $F = R, \boldsymbol{Y} = [1]$.

Updated Construction Complexity of Theorem 3.1.1: Algorithm 5 now requires $O\left(R \sum_{j=1}^{N} \text{nnz}(\boldsymbol{U}_j)\right)$ construction time and $O\left(\sum_{j=1}^{N} \text{nnz}(\boldsymbol{U}_j)\right)$ storage space, summing the costs for the updated structure from Lemma 3.3.2 over all matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$. The sampling complexity of these data structures is unaffected by the modifications, which completes the proof of the corollary. □

3.6.7 Alternating Least Squares CP Decomposition

CP Decomposition CP decomposition represents an N-dimensional tensor $\tilde{\mathcal{T}} \in \mathbb{R}^{I_1 \times ... \times I_n}$ as a weighted sum of generalized outer products. Formally, let $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ with $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$ be factor matrices with each column having unit norm, and let $\boldsymbol{\sigma} \in \mathbb{R}^R$ be a nonnegative coefficient vector. We call R the rank of the decomposition. The tensor $\tilde{\mathcal{T}}$ that the decomposition represents is given elementwise by

$$\tilde{\mathcal{T}}[i_1, ..., i_N] := \langle \boldsymbol{\sigma}^\top, \boldsymbol{U}_1[i_1, :], ..., \boldsymbol{U}_N[i_N, :] \rangle = \sum_{r=1}^{R} \boldsymbol{\sigma}[r] \boldsymbol{U}_1[i_1, r] \cdots \boldsymbol{U}_N[i_N, r],$$

which is a generalized inner product between $\boldsymbol{\sigma}^\top$ and rows $\boldsymbol{U}_j[i_j, :]$ for $1 \leq j \leq N$. Given an input tensor $\mathcal{T}$ and a target rank R, the goal of approximate CP decomposition is to find a rank-R representation $\tilde{\mathcal{T}}$ that minimizes the Frobenius norm $\left\|\mathcal{T} - \tilde{\mathcal{T}}\right\|_F$.

Definition of Matricization The matricization $\text{mat}(\mathcal{T}, j)$ flattens tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ into a matrix and isolates mode j along the row axis of the output. The output of matricization has dimensions $I_j \times \prod_{k \neq j} I_k$. We take the formal definition below from a survey by Kolda and Bader [KB09]. The tensor entry $\mathcal{T}[i_1, ..., i_N]$ is equal to the matricization entry $\text{mat}(\mathcal{T}, j)[i_N, u]$, where

$$u = 1 + \sum_{\substack{k=1 \\ k \neq j}}^{N} (i_k - 1) \prod_{\substack{m=1 \\ m \neq j}}^{k-1} I_m.$$

Details about Alternating Least Squares Let $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ be factor matrices of a low-rank CP decomposition, $\boldsymbol{U}_k \in \mathbb{R}^{I_k \times R}$. We use $\boldsymbol{U}_{\neq j}$ to denote $\bigodot_{k=N, k \neq j}^{k=1} \boldsymbol{U}_k$. Note the inversion of order here to match indexing in the definition of matricization above. Algorithm 10 gives the non-randomized alternating least squares algorithm CP-ALS that produces a decomposition of target rank R given input tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \ldots \times I_N}$ in general format. The random initialization on line 1 of the algorithm can be implemented by drawing each entry of the factor matrices $\boldsymbol{U}_j$ according to a standard normal distribution, or via a randomized range finder [HMT11]. The vector $\boldsymbol{\sigma}$ stores the generalized singular values of the decomposition. At iteration j within a round, ALS holds all factor matrices except $\boldsymbol{U}_j$ constant and solves a linear-least squares problem on line 6 for a new value for $\boldsymbol{U}_j$. In between least squares solves, the algorithm renormalizes the columns of each matrix $\boldsymbol{U}_j$ to unit norm and stores their original norms in the vector $\boldsymbol{\sigma}$. Section 3.6.9 contains more details about the randomized range finder and the convergence criteria used to halt iteration.

Algorithm 10 CP-ALS($\mathcal{T}$, R)

1: Initialize $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$ randomly for $1 \leq j \leq N$.
2: Renormalize $\boldsymbol{U}_j[:, i] \mathrel{/}= \|\boldsymbol{U}_j[:, i]\|_2$, $1 \leq j \leq N, 1 \leq i \leq R$.
3: Initialize $\boldsymbol{\sigma} \in \mathbb{R}^R$ to [1].
4: **while** not converged **do**
5: **for** $j = 1...N$ **do**
6: $\boldsymbol{U}_j := \arg\min_{\boldsymbol{X}} \left\|\boldsymbol{U}_{\neq j} \cdot \boldsymbol{X}^\top - \text{mat}(\mathcal{T}, j)^\top\right\|_F$
7: $\boldsymbol{\sigma}[i] = \|\boldsymbol{U}_j[:, i]\|_2$, $1 \leq i \leq R$
8: Renormalize $\boldsymbol{U}_j[:, i] \mathrel{/}= \|\boldsymbol{U}_j[:, i]\|_2$, $1 \leq i \leq R$.
9: **return** $[\boldsymbol{\sigma}; \boldsymbol{U}_1, ..., \boldsymbol{U}_N]$.

We obtain a randomized algorithm for sparse tensor CP decomposition by replacing the exact least squares solve on line 6 with a randomized method according to Theorem 3.2.1. Below,

we prove Corollary 3.3.3, which derives the complexity of the randomized CP decomposition algorithm.

Proof of Corollary 3.3.3. The design matrix $\boldsymbol{U}_{\neq j}$ for optimization problem j within a round of ALS has dimensions $\prod_{k\neq j} I_k \times R$. The observation matrix $\text{mat}(\mathcal{T}, j)^\top$ has dimensions $\prod_{k\neq j} I_k \times I_j$. To achieve error threshold $1+\varepsilon$ with probability $1-\delta$ on each solve, we draw $J = \tilde{O}\left(R/(\varepsilon\delta)\right)$ rows from both the design and observation matrices and solve the downsampled problem (Theorem 3.2.1). These rows are sampled according to the leverage score distribution on the rows of $\boldsymbol{U}_{\neq j}$, for which we use the data structure in Theorem 3.1.1. After a one-time initialization cost $O(\sum_{j=1}^{N} I_j R^2))$ before the ALS iteration begins, the complexity to draw J samples (assuming $I_j \geq R$) is

$$O\left(NR^3 + J\sum_{k\neq j} R^2 \log I_k\right) = \tilde{O}\left(NR^3 + \frac{R}{\varepsilon\delta}\sum_{k\neq j} R^2 \log I_k\right).$$

The cost to assemble the corresponding subset of the observation matrix is $O(JI_j) = \tilde{O}(RI_j/(\varepsilon\delta))$. The cost to solve the downsampled least squares problem is $O(JR^2) = \tilde{O}(I_j R^2/(\varepsilon\delta))$, which dominates the cost of forming the subset of the observation matrix. Finally, we require additional time $O(I_j R^2)$ to update the sampling data structure (Theorem 3.1.1 part 1). Adding these terms together and summing over $1 \leq j \leq N$ gives

$$\begin{aligned} &\tilde{O}\left(\frac{1}{\varepsilon\delta}\cdot\sum_{j=1}^{N}\left[I_j R^2 + \sum_{k\neq j} R^3 \log I_k\right]\right) \\ &= \tilde{O}\left(\frac{1}{\varepsilon\delta}\cdot\sum_{j=1}^{N}\left[I_j R^2 + (N-1)R^3 \log I_j\right]\right). \end{aligned} \tag{3.21}$$

Rounding $N-1$ to N and multiplying by the number of iterations gives the desired complexity. When $I_j < R$ for any j, the complexity changes in Theorem 3.1.1 propagate to the equation above. The column renormalization on line 8 of the CP-ALS algorithm contributes additional time $O\left(\sum_{j=1}^{N} I_j R\right)$ per round, a lower-order term.

□

3.6.8 Experimental Platform and Sampler Parallelism

We provide two implementations of our sampler. The first is a slow reference implementation written entirely in Python, which closely mimics our pseudocode and can be used to test correctness. The second is an efficient implementation written in C++, parallelized in shared memory with OpenMP and Intel Thread Building Blocks.

Each Perlmutter CPU node (our experimental platform) is equipped with two sockets, each containing an AMD EPYC 7763 processor with 64 cores. All benchmarks were conducted with our efficient C++ implementation using 128 OpenMP threads. We link our code against Intel Thread Building blocks to call a multithreaded sort function when decomposing sparse tensors. We use OpenBLAS 0.3.21 to handle linear algebra with OpenMP parallelism enabled, but our code links against any linear algebra library implementing the CBLAS and LAPACKE interfaces.

Our proposed data structure samples from the exact distribution of leverage scores of the Khatri-Rao product, thereby enjoying better sample efficiency than alternative approaches such as CP-ARLS-LEV [LK22]. The cost to draw each sample, however, is $O(R^2 \log H)$, where H is the number of rows in the Khatri-Rao product. Methods such as row-norm-squared sampling or CP-ARLS-LEV can draw each sample in time $O(\log H)$ after appropriate preprocessing. Therefore, efficient parallelization of our sampling procedure is required for competitive performance, and we present two strategies below.

1. **Asynchronous Thread Parallelism**: The KRPSampleDraw procedure in Algorithm 6 can be called by multiple threads concurrently without data races. The simplest parallelization strategy divides the J samples equally among the threads in a team, each of which makes calls to KRPSampleDraw asynchronously. This strategy works well on a CPU, but is less attractive on a SIMT processor like a GPU where instruction streams cannot diverge without significant performance penalties.

2. **Synchronous Batch Parallelism** As an alternative to the asynchronous strategy, suppose for the moment that all leaves have the same depth in each segment tree. Then for every sample, STSample makes a sequence of calls to $\tilde{m}$, each updating the current node by branching left or right in the tree. The length of this sequence is the depth of the tree, and it is followed by a single call to the function $\tilde{\boldsymbol{q}}$. Observe that procedure $\tilde{m}$ in Algorithm 8 can be computed with a matrix-vector multiplication followed by a dot product. The procedure $\tilde{\boldsymbol{q}}$ of Algorithm 8 requires the same two operations if $F = 1$ or $\boldsymbol{Y} = [1]$. Thus, we can create a batched version of our sampling procedure that makes a fixed length sequence of calls to batched `gemv` and `dot` routines. All processors march in lock-step down the levels of each segment tree, each tracking the branching paths of a distinct set of samples. The MAGMA linear algebra library provides a batched version of `gemv` [Hai+15], while a batched dot product can be implemented with an ad hoc kernel. MAGMA also offers a batched version of the symmetric rank-k update routine `syrk`, which is helpful to parallelize row sampler construction (Algorithm 7). When all leaves in the tree are not at the same level, the the bottom level of the tree can be handled with a special sequence of instructions making the required additional calls to $\tilde{m}$.

Our CPU code follows the batch synchronous design pattern. To avoid dependency on GPU-based MAGMA routines in our CPU prototype, portions of the code that should be batched BLAS calls are standard BLAS calls wrapped in a `for` loop. These sections can be easily replaced when the appropriate batched routines are available.

3.6.9 Sparse Tensor CP Experimental Configuration

Table 3.3: Sparse tensors from FROSTT collection.

Tensor	Dimensions	Nonzeros	Prep.	Init.
Uber Pickups	183 × 24 × 1,140 × 1,717	3,309,490	None	IID
Enron Emails	6,066 × 5,699 × 244,268 × 1,176	54,202,099	log	RRF
NELL-2	12,092 × 9,184 × 28,818	76,879,419	log	IID
Amazon Reviews	4,821,207 × 1,774,269 × 1,805,187	1,741,809,018	None	IID
Reddit-2015	8,211,298 × 176,962 × 8,116,559	4,687,474,081	log	IID

Table 3.3 lists the nonzero counts and dimensions of sparse tensors in our experiments [Smi+17]. We took the log of all values in the Enron, NELL-2, and Reddit-2015 tensors. Consistent with established practice, this operation damps the effect of a few high magnitude tensor entries on the fit metric [LK22].

The factor matrices for the Uber, Amazon, NELL-2, and Reddit experiments were initialized with i.i.d. entries from the standard normal distribution. As suggested by Larsen and Kolda [LK22], the Enron tensor's factors were initialized with a randomized range finder [HMT11]. The range finder algorithm initializes each factor matrix U_j as $\text{mat}(\mathcal{T}, j)S$, a sketch applied to the mode-j matricization of $\mathcal{T}$ with $S \in \mathbb{R}^{\prod_{k \neq j} I_k \times R}$. Larsen and Kolda [LK22] chose S as a sparse sampling matrix to select a random subset of fibers along each mode. We instead used an i.i.d. Gaussian sketching matrix that was not materialized explicitly. Instead, we exploited the sparsity of $\mathcal{T}$ and noted that at most $\text{nnz}(\mathcal{T})$ columns of $\text{mat}(\mathcal{T}, j)$ were nonzero. Thus, we computed at most $\text{nnz}(\mathcal{T})$ rows of the random sketching matrix S, which were lazily generated and discarded during the matrix multiplication without incurring excessive memory overhead.

ALS was run for a maximum of 40 rounds on all tensors except for Reddit, which was run for 80 rounds. The exact fit was computed every 5 rounds (defined as 1 epoch), and we used an early stopping condition to terminate runs before the maximum round count. The algorithm was terminated at epoch T if the maximum fit in the last 3 epochs did not exceed the maximum fit from epoch 1 through epoch $T - 3$ by tolerance 10^{-4}.

Hybrid CP-ARLS-LEV deterministically includes rows from the Khatri-Rao product whose probabilities exceed a threshold τ. The ostensible goal of this procedure is to improve diversity

in sample selection, as CP-ARLS-LEV may suffer from many repeat draws of high probability rows. We replicated the conditions proposed in the original work by selecting $\tau = 1/J$ [LK22].

Individual trials of non-randomized (exact) ALS on the Amazon and Reddit tensors required several hours on a single Perlmutter CPU node. To speed up our experiments, accuracy measurements for exact ALS in Figure 3.4 were carried out using multi-node SPLATT, The Surprisingly ParalleL spArse Tensor Toolkit [SK16b], on four Perlmutter CPU nodes. The fits computed by SPLATT agree with those computed by our own non-randomized ALS implementation. As a result, Figure 3.4 verifies that our randomized algorithm STS-CP produces tensor decompositions with accuracy comparable to those by highly-optimized, state-of-the-art CP decomposition software. We leave a distributed-memory implementation of our *randomized* algorithms to future work.

3.6.10 Efficient Computation of Sketch Distortion

The variable σ has a definition in this section distinct from the rest of this work. The condition number κ of a matrix $\boldsymbol{M}$ is defined as

$$\kappa(\boldsymbol{M}) := \frac{\sigma_{\max}(\boldsymbol{M})}{\sigma_{\min}(\boldsymbol{M})}$$

where $\sigma_{\min}(\boldsymbol{M})$ and $\sigma_{\max}(\boldsymbol{M})$ denote the minimum and maximum nonzero singular values of $\boldsymbol{M}$. Let $\boldsymbol{A}$ be a Khatri-Rao product of N matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ with $\prod_{j=1}^{N} I_j$ rows, R columns, and rank $r \leq R$. Let $\boldsymbol{A} = \boldsymbol{Q}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ be its reduced singular value decomposition with $\boldsymbol{Q} \in \mathbb{R}^{\prod_j I_j \times r}, \boldsymbol{\Sigma} \in \mathbb{R}^{r \times r}$, and $\boldsymbol{V} \in \mathbb{R}^{r \times R}$. Finally, let $\boldsymbol{S} \in \mathbb{R}^{J \times \prod_j I_j}$ be a leverage score sampling matrix for $\boldsymbol{A}$. Our goal is to compute $\kappa(\boldsymbol{S}\boldsymbol{Q})$ without fully materializing either $\boldsymbol{A}$ or its QR decomposition. We derive

$$\begin{aligned} \kappa(\boldsymbol{S}\boldsymbol{Q}) &= \kappa(\boldsymbol{S}\boldsymbol{Q}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{V}\boldsymbol{\Sigma}^{-1}) \\ &= \kappa(\boldsymbol{S}\boldsymbol{A}\boldsymbol{V}\boldsymbol{\Sigma}^{-1}) \end{aligned} \tag{3.22}$$

The matrix $\boldsymbol{S}\boldsymbol{A} \in \mathbb{R}^{J \times R}$ is efficiently computable using our leverage score sampling data structure. We require time $O(JR^2)$ to multiply by $\boldsymbol{V}\boldsymbol{\Sigma}^{-1}$ and compute the singular value decomposition of the product to get the condition number. Next observe that $\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{V}\boldsymbol{\Sigma}^2\boldsymbol{V}^\top$, so we can recover $\boldsymbol{V}$ and $\boldsymbol{\Sigma}^{-1}$ by eigendecomposition of $\boldsymbol{A}^\top\boldsymbol{A} \in \mathbb{R}^{R \times R}$ in time $O(R^3)$. Finally, recall the formula

$$\boldsymbol{A}^\top\boldsymbol{A} = \circledast_{j=1}^{N} \boldsymbol{U}_j^\top\boldsymbol{U}_j$$

used at the beginning of Section 3.3 that enables computation of $\boldsymbol{A}^\top\boldsymbol{A}$ in time $O\left(\sum_{j=1}^{N} I_j R^2\right)$ without materializing the full Khatri-Rao product. Excluding the time to form $\boldsymbol{S}\boldsymbol{A}$ (which is

given by Theorem 3.1.1), $\kappa(\boldsymbol{SQ})$ is computable in time

$$O\left(JR^2 + R^3 + \sum_{j=1}^{N} I_j R^2\right).$$

3.6.11 Further Experiments

3.6.11.1 Probability Distribution Comparison

Figure 3.8 provides confirmation on a small test problem that our sampler works as expected. For the Khatri-Rao product of three matrices $\boldsymbol{A} = \boldsymbol{U}_1 \odot \boldsymbol{U}_2 \odot \boldsymbol{U}_3$, it plots the true distribution of leverage scores against a normalized histogram of 50,000 draws from the data structure in Theorem 3.1.1. We choose $\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3 \in \mathbb{R}^{8\times 8}$ initialized i.i.d. from a standard normal distribution with 1% of all entries multiplied by 10. We observe excellent agreement between the histogram and the true distribution.

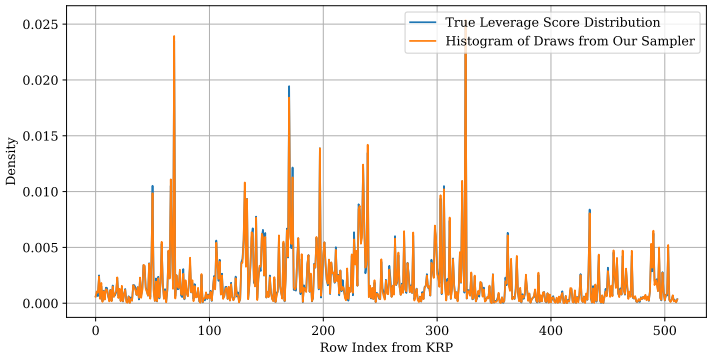


Figure 3.8: Comparison of true leverage score distribution with histogram of 50,000 samples drawn from $\boldsymbol{U}_1 \odot \boldsymbol{U}_2 \odot \boldsymbol{U}_3$.

3.6.11.2 Fits Achieved for $J = 2^{16}$

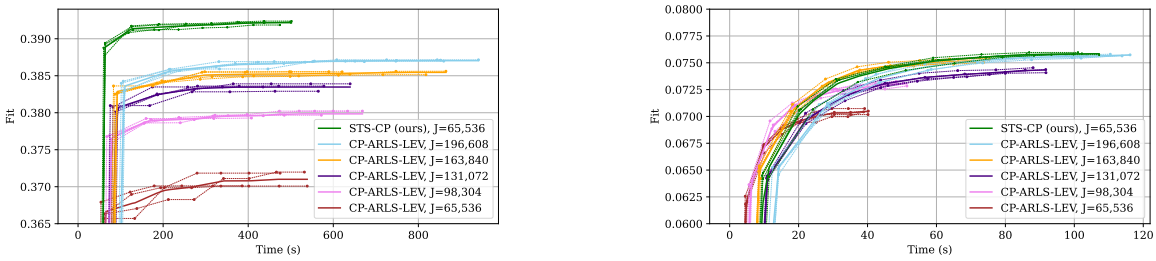
Table 3.4 gives the fits achieved for sparse tensor decomposition for varying rank and algorithm (presented graphically in Figure 3.5). Uncertainties are one standard deviation across 8 runs of ALS.

3.6.11.3 Fit as a Function of Time

Figures 3.9a and 3.9b shows the fit as a function of time for the Amazon Reviews and NELL2 tensors. The hybrid version of CP-ARLS-LEV was used for comparison in both experiments. As in section 3.4.3, thick lines are averages of the running max fit across 4 ALS trials, shown by the thin dotted lines. For Amazon, the STS-CP algorithm makes faster progress than CP-ARLS-LEV at all tested sample counts.

Table 3.4: Fits Achieved by Randomized Algorithms for Sparse Tensor Decomposition, $J = 2^{16}$, and non-randomized ALS. The best result among randomized algorithms is boldfaced. "CP-ARLS-LEV-H" refers to the hybrid version of CP-ARLS-LEV and "Exact" refers to non-randomized ALS.

Tensor	R	CP-ARLS-LEV	CP-ARLS-LEV-H	STS-CP (ours)	Exact
Uber	25	.187 ± 2.30e-03	.188 ± 2.11e-03	**.189** ± 1.52e-03	.190 ± 1.41e-03
	50	.211 ± 1.72e-03	.212 ± 1.27e-03	**.216** ± 1.18e-03	.218 ± 1.61e-03
	75	.218 ± 1.76e-03	.218 ± 2.05e-03	**.230** ± 9.24e-04	.232 ± 9.29e-04
	100	.217 ± 3.15e-03	.217 ± 1.69e-03	**.237** ± 2.12e-03	.241 ± 1.00e-03
	125	.213 ± 1.96e-03	.213 ± 2.47e-03	**.243** ± 1.78e-03	.247 ± 1.52e-03
Enron	25	.0881 ± 1.02e-02	.0882 ± 9.01e-03	**.0955** ± 1.19e-02	.0978 ± 8.50e-03
	50	.0883 ± 1.72e-02	.0920 ± 6.32e-03	**.125** ± 1.03e-02	.132 ± 1.51e-02
	75	.0899 ± 6.10e-03	.0885 ± 6.39e-03	**.149** ± 1.25e-02	.157 ± 4.87e-03
	100	.0809 ± 1.26e-02	.0787 ± 1.00e-02	**.164** ± 5.90e-03	.176 ± 4.12e-03
	125	.0625 ± 1.52e-02	.0652 ± 1.00e-02	**.182** ± 1.04e-02	.190 ± 4.35e-03
NELL-2	25	.0465 ± 9.52e-04	.0467 ± 4.61e-04	**.0470** ± 4.69e-04	.0478 ± 7.20e-04
	50	.0590 ± 5.33e-04	.0593 ± 4.34e-04	**.0608** ± 5.44e-04	.0618 ± 4.21e-04
	75	.0658 ± 6.84e-04	.0660 ± 3.95e-04	**.0694** ± 2.96e-04	.0708 ± 3.11e-04
	100	.0700 ± 4.91e-04	.0704 ± 4.48e-04	**.0760** ± 6.52e-04	.0779 ± 5.09e-04
	125	.0729 ± 8.56e-04	.0733 ± 7.22e-04	**.0814** ± 5.03e-04	.0839 ± 8.47e-04
Amazon	25	.338 ± 6.63e-04	.339 ± 6.99e-04	**.340** ± 6.61e-04	.340 ± 5.78e-04
	50	.359 ± 1.09e-03	.360 ± 8.04e-04	**.366** ± 7.22e-04	.366 ± 1.01e-03
	75	.367 ± 1.82e-03	.370 ± 1.74e-03	**.382** ± 9.13e-04	.382 ± 5.90e-04
	100	.366 ± 3.05e-03	.371 ± 2.53e-03	**.392** ± 6.67e-04	.393 ± 5.62e-04
	125	.358 ± 6.51e-03	.364 ± 4.22e-03	**.400** ± 3.67e-04	.401 ± 3.58e-04
Reddit	25	.0581 ± 1.02e-03	.0583 ± 2.78e-04	**.0592** ± 3.07e-04	.0596 ± 4.27e-04
	50	.0746 ± 1.03e-03	.0738 ± 4.85e-03	**.0774** ± 7.88e-04	.0783 ± 2.60e-04
	75	.0845 ± 1.64e-03	.0849 ± 8.96e-04	**.0909** ± 5.49e-04	.0922 ± 3.69e-04
	100	.0904 ± 1.35e-03	.0911 ± 1.59e-03	**.101** ± 6.25e-04	.103 ± 7.14e-04
	125	.0946 ± 2.13e-03	.0945 ± 3.17e-03	**.109** ± 7.71e-04	.111 ± 7.98e-04



(a) Amazon (b) NELL-2

Figure 3.9: Fit as a function of time, $R = 100$.

For the NELL-2 tensor, STS-CP makes slower progress than CP-ARLS-LEV for sample counts up to $J = 163,840$. On average, these trials with CP-ARLS-LEV do not achieve the same final fit as STS-CP. CP-ARLS-LEV finally achieves a comparable fit to STS-CP when the former uses $J = 196,608$ samples, compared to $J = 65,536$ for our method.

3.6.11.4 Speedup of STS-CP and Practical Usage Guide

Timing Comparisons. For each tensor, we now compare hybrid CP-ARLS-LEV and STS-CP on the time required to achieve a fixed fraction of the fit achieved by non-randomized ALS. For each tensor and rank in the set $\{25, 50, 75, 100, 125\}$, we ran both algorithms using a range of sample counts. We tested STS-CP on values of J from the set $\{2^{15}x \mid 1 \leq x \leq 4\}$ for all tensors. CP-ARLS-LEV required a sample count that varied significantly between datasets to hit the required thresholds, and we report the sample counts that we tested in Table 3.5. Because CP-ARLS-LEV has poorer sample complexity than STS-CP, we tested a wider range of sample counts for the former algorithm.

Table 3.5: Tested Sample Counts for hybrid CP-ARLS-LEV.

Tensor	Values of J Tested
Uber	$\{2^{15}x \mid x \in \{1..13\}\}$
Enron	$\{2^{15}x \mid x \in \{1..7\} \cup \{10, 12, 14, 16, 18, 20, 22, 26, 30, 34, 38, 42, 46, 50, 54\}\}$
NELL-2	$\{2^{15}x \mid x \in \{1..7\}\}$
Amazon	$\{2^{15}x \mid x \in \{1..7\}\}$
Reddit	$\{2^{15}x \mid x \in \{1..12\}\}$

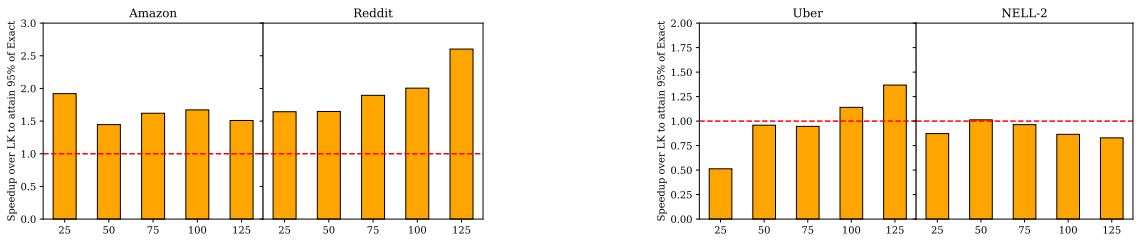
For each configuration of tensor, target rank R, sampling algorithm, and sample count J, we ran 4 trials using the configuration and stopping criteria in Section 3.6.9. The result of each trial was a set of (time, fit) pairs. For each configuration, we linearly interpolated the pairs for each trial and averaged the resulting continuous functions over all trials. The result for

each configuration was a function $f_{\mathcal{T},R,A,J} : \mathbb{R}^+ \to [0,1]$. The value $f_{\mathcal{T},R,A,J}(t)$ is the average fit at time t achieved by algorithm A to decompose tensor $\mathcal{T}$ with target rank R using J samples per least squares solve. Finally, let

$$\text{Speedup}_{\mathcal{T},R,M} := \frac{\min_J \text{argmin}_{t\geq 0} \left[f_{\mathcal{T},R,\text{CP-ARLS-LEV-H},J}(t) > P\right]}{\min_J \text{argmin}_{t\geq 0} \left[f_{\mathcal{T},R,\text{STS-CP},J}(t) > P\right]}$$

be the speedup of STS-CP to over CP-ARLS-LEV (hybrid) to achieve a threshold fit P on tensor $\mathcal{T}$ with target rank R. We let the threshold P for each tensor $\mathcal{T}$ be a fixed fraction of the fit achieved by non-randomized ALS (see Table 3.4).

Figure 3.10 reports the speedup of STS-CP over hybrid CP-ARLS-LEV for $P = 0.95$ on all tensors except Enron. For large tensors with over one billion nonzero elements, we report a significant speedup anywhere from 1.4x to 2.0x for all tested ranks. For smaller tensors with less than 100 million nonzero entries, the lower cost of each least squares solve lessens the impact of the expensive, more accurate sample selection phase of STS-CP. Despite this, STS-CP performs comparably to CP-ARLS-LEV at most ranks, with significant slowdown only at rank 25 on the smallest tensor Uber.



(a) Large Tensors

(b) Small Tensors

Figure 3.10: Speedup of STS-CP over CP-ARLS-LEV hybrid (LK) to reach 95% of the fit achieved by non-randomized ALS. Large tensors have more than 1 billion nonzero entries.

On the Enron tensor, hybrid CP-ARLS-LEV could not achieve the 95% accuracy threshold for any rank above 25 for the sample counts tested in Table 3.5. **STS-CP achieved the threshold accuracy for all ranks tested**. Instead, Figure 3.11 reports the speedup to achieve 85% of the fit of non-randomized ALS on the Enron. Beyond rank 25, our method consistently exhibits more than 2x speedup to reach the threshold.

Guide to Sampler Selection. Based on the performance comparisons in this section, we offer the following guide to CP decomposition algorithm selection. Our experiments demonstrate that **STS-CP offers the most benefit on sparse tensors with billions of nonzeros elements (Amazon and Reddit) at high target decomposition rank**. Here, the runtime of our more expensive sampling procedure is offset by reductions in the least

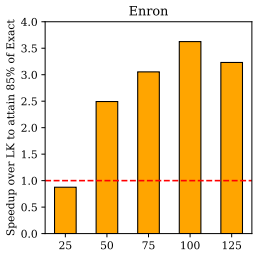


Figure 3.11: Speedup of STS-CP over CP-ARLS-LEV hybrid to reach 85% of the fit achieved by non-randomized ALS, Enron Tensor.

squares solve time. For smaller tensors, our sampler may still offer significant performance benefits (Enron). In other cases (Uber, NELL-2), CP-ARLS-LEV exhibits better performance, but by small margins for rank beyond 50.

STS-CP reduces the cost of each least squares solve through a sample selection process that relies on dense linear algebra primitives (see Algorithms 7 and 8). Because these operations can be expressed as standard BLAS calls and can be carried out in parallel (see Section 3.6.8, we hypothesize that STS-CP is favorable when GPUs or other dense linear algebra accelerators are available.

Because our target tensor is sparse, the least squares solve during each ALS iteration requires a sparse matricized-tensor times Khatri-Rao product (spMTTKRP) operation. After sampling, this primitive can reduced to sparse-matrix dense-matrix multiplication (SpMM). Development of accelerators for these primitives is an active area of research [Wij+23; Son+22]. When such accelerators are available, the lower cost of the spMTTKRP operation reduces the relative benefit provided by the STS-CP sample selection method. We hypothesize that CP-ARLS-LEV, with its faster sample selection process but lower sample efficiency, may retain its benefit in this case. We leave verification of these two hypotheses as future work.

Chapter 4

Distributed Randomized Sparse CP Decomposition

Chapter 3 established the theory of a sketching algorithm to accelerate the MTTKRP. We now focus on the downstream application—randomized Candecomp / PARAFAC decomposition—and consider the practical details of distributing the algorithm on a cluster of processors. Because processor-to-processor data exchange forms the main bottleneck, we devise strategies to avoid communication while obeying local memory constraints on each node. Our deployed methods rely on a theoretical communication analysis and novel strategies for distributing a binary tree traversal.

Our distributed-memory randomized algorithms, d-STS-CP and d-CP-ARLS-LEV, have significant advantages while preserving the accuracy of the final approximation. As Figure 4.2 shows, d-STS-CP computes a rank 100 decomposition of the Reddit tensor (~ 4.7 billion nonzero entries) with a 11x speedup over SPLATT, a state-of-the-art distributed-memory decomposition package. The reported speedup was achieved on 512 CPU cores, with a final fit within 0.8% of non-randomized ALS for the same iteration count. While the distributed algorithm d-CP-ARLS-LEV achieves a lower final accuracy, it makes progress faster than SPLATT and spends less time on sampling (completing 80 rounds in an average of 81 seconds). We demonstrate that it is well-suited to smaller tensors and lower target ranks.

4.1 Introduction

Randomized algorithms for numerical linear algebra have become increasingly popular in the past decade, but their distributed-memory communication characteristics and scaling properties have received less attention. In this work, we examine randomized algorithms to compute the Candecomp / PARAFAC (CP) decomposition, a generalization of the matrix singular-value decomposition to a number of modes $N > 2$. Given a tensor $\mathcal{T} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$

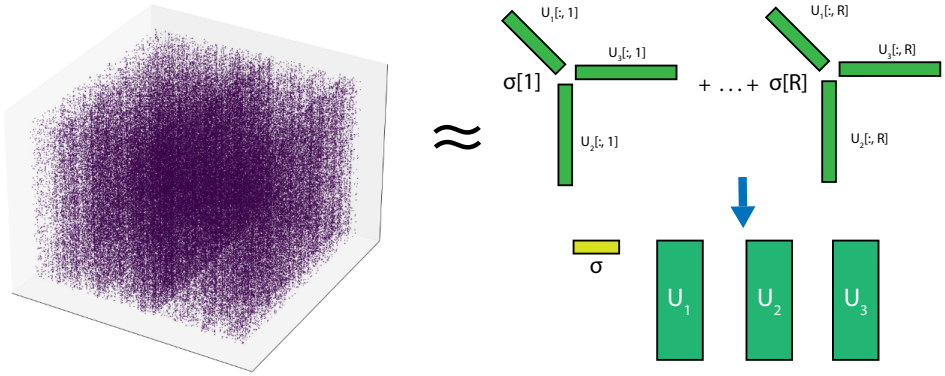


Figure 4.1: A subset of entries from the 3D Amazon Review sparse tensor [Smi+17] and its illustrated CP decomposition.

and a target rank R, the goal of CP decomposition (illustrated in Figure 4.1) is to find a set of *factor matrices* $U_1, ..., U_N, U_j \in \mathbb{R}^{I_j \times R}$ with unit norm columns and a nonnegative vector $\sigma \in \mathbb{R}^R$ satisfying

$$\mathcal{T}[i_1, ..., i_N] \approx \sum_{r=1}^{R} \boldsymbol{\sigma}[r] \boldsymbol{U}_1[i_1, r] ... \boldsymbol{U}_N[i_N, r]. \tag{4.1}$$

We consider real **sparse** tensors $\mathcal{T}$ with $N \geq 3$, all entries known, and billions of nonzero entries. Sparse tensors are a flexible abstraction for a variety of data, such as network traffic logs [Mao+14], text corpora [Smi+17], and knowledge graphs [BAH19].

4.1.1 Motivation

Why is a low-rank approximation of a sparse tensor useful? We can view the sparse CP decomposition as an extension of well-studied sparse matrix factorization methods, which can mine patterns from large datasets [KP07]. Each row of the CP factors is a dense embedding vector for an index $i_j \in [I_j]$, $1 \leq j \leq N$. Because each embedding is a small dense vector while the input tensor is sparse, sparse tensor CP decomposition may incur high relative error with respect to the input and rarely captures the tensor sparsity structure exactly. Nevertheless, the learned embeddings contain valuable information. CP factor matrices have been successfully used to identify patterns in social networks [HKD20; LK22], detect anomalies in packet traces [Mao+14], and monitor trends in internal network traffic [Smi+18]. As Table 4.1 shows, a wealth of software packages exists to meet the demand for sparse tensor decomposition.

One of the most popular methods for computing a sparse CP decomposition, the *Alternating-Least-Squares* (ALS) algorithm, involves repeatedly solving large, overdetermined linear

Software	Source	Notes
SPLATT	[Smi+15]	CA-Distributed Algorithms, CSF Format
BIGTensor	[Par+16]	Hadoop MapReduce
ParTI!	[LMV18]	GPU Support, HiCOO Format
Genten	[PK19]	Kokkos Parellelism and Performance Portability

Table 4.1: Selected well-documented software packages for non-randomized sparse tensor CP decomposition and their major contributions.

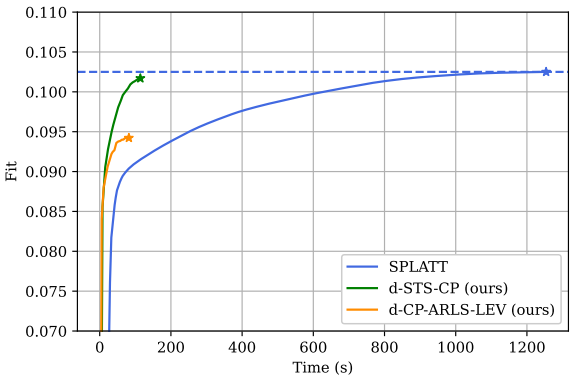


Figure 4.2: Running maximum accuracy over time for SPLATT, a state-of-the-art distributed CP decomposition software package, and our randomized algorithms on the Reddit tensor, target rank $R = 100$, on 512 CPU cores. Curves are averages of 5 trials, 80 ALS rounds.

least-squares problems with structured design matrices [KB09]. High-performance libraries DFacto [CV14], SPLATT [Smi+15], HyperTensor [KU15], and BigTensor [Par+16] distribute these expensive computations to a cluster of processors that communicate through an interconnect. Separately, several works use randomized sampling methods to accelerate the least-squares solves, with prototypes implemented in a shared-memory setting [Che+16; LK22; Mal22; Bha+23]. These randomized algorithms have strong theoretical guarantees and offer significant asymptotic advantages over non-randomized ALS. Unfortunately, prototypes of these methods require hours to run [LK22; Bha+23] and are neither competitive nor scalable compared to existing libraries with distributed-memory parallelism.

4.1.2 Our Contributions

We propose the first distributed-memory parallel formulations of two randomized algorithms, CP-ARLS-LEV [LK22] and STS-CP [Bha+23], with accuracy identical to their shared-memory prototypes. We then provide implementations of these methods that scale to thousands of

CPU cores. We face **dual technical challenges** to parallel scaling. First, sparse tensor decomposition generally has lower arithmetic intensity (FLOPs / data word communicated between processors) than dense tensor decomposition, since computation scales linearly with the tensor nonzero count. Some sparse tensors exhibit nonzero fractions as low as 4×10^{-10} (see Table 4.5), while the worst-case communication costs for sparse CP decomposition remain identical to the dense tensor case [SK16a]. Second, randomized algorithms can save an order of magnitude in computation over their non-randomized counterparts [Mah11; DM16; MT20], but their inter-processor communication costs remain unaltered unless carefully optimized. Despite these compounding factors that reduce arithmetic intensity, we achieve both speedup and scaling through several key innovations, three of which we highlight:

Novel Distributed-Memory Sampling Procedures Random sample selection is challenging to implement when the CP factor matrices and sparse tensor are divided among P processors. We introduce two distinct communication-avoiding algorithms for randomized sample selection from the Khatri-Rao product. First, we show how to implement the CP-ARLS-LEV algorithm by computing an independent probability distribution on the factor block row owned by each processor. The resulting distributed algorithm has minimal compute / communication overhead compared to the other phases of CP decomposition. The second algorithm, STS-CP, requires higher sampling time, but achieves lower error by performing random walks on a binary tree for each sample. By distributing leaf nodes uniquely to processors and replicating internal nodes, we give a sampling algorithm with per-processor communication bandwidth scaling as $O(\log P/P)$ (see Table 4.3).

Communication-Optimized MTTKRP We show that communication-optimal schedules for non-randomized ALS may exhibit disproportionately high communication costs for randomized algorithms. To combat this, we use an "accumulator-stationary" schedule that eliminates expensive `Reduce-scatter` collectives, causing all communication costs to scale with the number of random samples taken. This alternate schedule significantly reduces communication on tensors with large dimensions (Figure 4.8) and empirically improves the computational load balance (Figure 4.11).

Local Tensor Storage Format Existing storage formats developed for sparse CP decomposition [Smi+15; Nis+19] are not optimized for random access into the sparse tensor, which our algorithms require. In response, we use a modified compressed-sparse-column format to store each matricization of our tensor, allowing efficient selection of nonzero entries by our random sampling algorithms. We then transform the selected nonzero entries into compressed sparse row format, which eliminates shared-memory data races in the subsequent sparse-dense matrix multiplication. The cost of the transposition is justified and provides a roughly 1.7x speedup over using atomic operations in a hybrid OpenMP / MPI implementation.

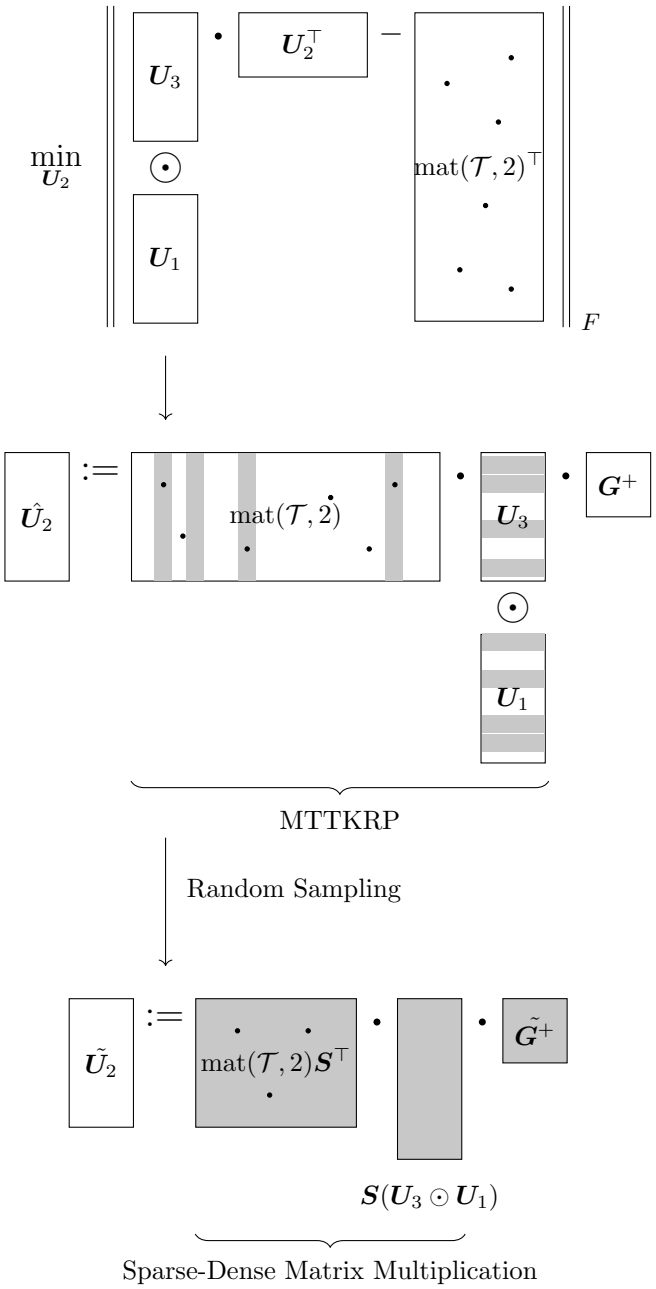


Figure 4.3: Top: the linear least-squares problem to optimize factor matrix $\boldsymbol{U}_2$ during the ALS algorithm for a 3D tensor (column dimension of $\mathrm{mat}(\mathcal{T}, 2)$ not to scale). Middle: the exact solution to the problem using the Matricized Tensor Times Khatri-Rao Product (MTTKRP). Shaded columns of $\mathrm{mat}(\mathcal{T}, 2)$ and rows of $(\boldsymbol{U}_3 \odot \boldsymbol{U}_1)$ are selected by our random sampling algorithm. Bottom: the downsampled linear least-squares problem after applying random sampling matrix $\boldsymbol{S}$.

Symbol	Description
$\mathcal{T}$	Sparse tensor of dimensions $I_1 \times \ldots \times I_N$
R	Target Rank of CP Decomposition
$\boldsymbol{U}_1, \ldots, \boldsymbol{U}_N$	Dense factor matrices, $\boldsymbol{U}_j \in \mathbb{R}^{I_j \times R}$
$\boldsymbol{\sigma}$	Vector of scaling factors, $\sigma \in \mathbb{R}^R$
J	Sample count for randomized ALS
P	Total processor count
$P_1, \ldots, P_N$	Dimensions of processor grid, $\prod_i P_i = P$
$\boldsymbol{U}_i^{(p_j)}$	Block row of U_i owned by processor p_j

Table 4.2: Notation for Chapter 4.

4.2 Notation and Preliminaries

Table 4.2 summarizes our notation for this chapter, and we briefly review the details of alternating least-squares CP Decomposition from Section 3.6.7. Let $\mathcal{T}$ be an N-dimensional tensor indexed by tuples $(i_1, ..., i_N) \in [I_1] \times ... \times [I_N]$, with $\text{nnz}(\mathcal{T})$ as the number of nonzero entries. In this work, sparse tensors are always represented as a collection of $(N+1)$-tuples, with the first N elements giving the indices of a nonzero element and the last element giving the value. We seek a low-rank approximation of $\mathcal{T}$ given by Equation (4.1), the right-hand-side of which we abbreviate as $[\boldsymbol{\sigma}; \boldsymbol{U}_1, ..., \boldsymbol{U}_N]$. By convention, each column of $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ has unit norm. Our goal is to minimize the sum of squared differences between our approximation and the provided tensor:

$$\text{argmin}_{\boldsymbol{\sigma}, \boldsymbol{U}_1, \ldots, \boldsymbol{U}_N} \left\| [\boldsymbol{\sigma}; \boldsymbol{U}_1, ..., \boldsymbol{U}_N] - \mathcal{T} \right\|_{\text{F}}^2 . \tag{4.2}$$

4.2.1 Non-Randomized ALS CP Decomposition

Minimizing Equation (4.2) jointly over $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ is still a non-convex problem (the vector $\boldsymbol{\sigma}$ can be computed directly from the factor matrices by renormalizing each column). Alternating least squares is a popular heuristic algorithm that iteratively drives down the approximation error. The algorithm begins with a set of random factor matrices and optimizes the approximation in rounds, each involving N subproblems. The j-th subproblem in a round holds all factor matrices but $\boldsymbol{U}_j$ constant and solves for a new matrix $\hat{\boldsymbol{U}}_j$ minimizing the squared Frobenius norm error [KB09]. The updated matrix $\hat{\boldsymbol{U}}_j$ is the solution to the overdetermined linear least-squares problem

$$\hat{\boldsymbol{U}}_j := \arg\min_{\boldsymbol{X}} \left\| \boldsymbol{U}_{\neq j} \cdot \boldsymbol{X}^\top - \text{mat}(\mathcal{T}, j)^\top \right\|_F . \tag{4.3}$$

Here, the design matrix is

$$\boldsymbol{U}_{\neq j} := \boldsymbol{U}_N \odot ... \odot \boldsymbol{U}_{j+1} \odot \boldsymbol{U}_{j-1} \odot ... \odot \boldsymbol{U}_1,$$

which is a Khatri-Rao Product (KRP) of the factors held constant. The matrix $\text{mat}(\mathcal{T}, j)$ is a *matricization* of the sparse tensor $\mathcal{T}$, which reorders the tensor modes and flattens it into a matrix of dimensions $I_j \times (\prod_{i \neq j} I_i)$. We solve the problem efficiently using the normal equations. Denoting the Gram matrix by $\boldsymbol{G} = (\boldsymbol{U}_{\neq j})^\top (\boldsymbol{U}_{\neq j})$, we have

$$\hat{\boldsymbol{U}}_j := \text{mat}(\mathcal{T}, j) \cdot \boldsymbol{U}_{\neq j} \cdot \boldsymbol{G}^+, \tag{4.4}$$

where $\boldsymbol{G}^+$ is the Moore-Penrose pseudo-inverse of $\boldsymbol{G}$. Since $\boldsymbol{U}_{\neq j}$ is a Khatri-Rao product, we can efficiently compute $\boldsymbol{G}$ through the well-known [KB09] formula

$$\boldsymbol{G} = \circledast_{k \neq j} (\boldsymbol{U}_k^\top \boldsymbol{U}_k), \tag{4.5}$$

where $\circledast$ denotes elementwise multiplication. Figure 4.3 illustrates each least-squares problem, and Algorithm 10 summarizes the ALS procedure, including a renormalization of factor matrix columns after each solve. We implement the initialization step in line 1 by drawing all factor matrix entries from a unit-variance Gaussian distribution, a standard technique [LK22].

The most expensive component of the ALS algorithm is the operation $\text{mat}(\mathcal{T}, j) \cdot \boldsymbol{U}_{\neq j}$ in Equation (4.4), an MTTKRP kernel. For a sparse tensor $\mathcal{T}$, this kernel has a computational pattern similar to sparse-dense matrix multiplication (SpMM): for each nonzero in the sparse tensor, we compute a scaled Hadamard product between $N - 1$ rows of the constant factor matrices and add it to a row of the remaining factor matrix. The MTTKRP runtime is

$$O(\text{nnz}(\mathcal{T}) N R), \tag{4.6}$$

which is linear in the nonzero count of $\mathcal{T}$. Because $\mathcal{T}$ may have billions of nonzero entries, we seek methods to drive down the cost of the MTTKRP.

4.2.2 Randomized Leverage Score Sampling

As Chapter 3 demonstrates, sketching is a powerful tool to accelerate least squares problems of the form $\min_{\boldsymbol{X}} \|\boldsymbol{A}\boldsymbol{X} - \boldsymbol{B}\|_{\text{F}}$ where $\boldsymbol{A}$ has far more rows than columns [Mah11; DM16; MT20]. We apply a structured sketching matrix $\boldsymbol{S} \in \mathbb{R}^{J \times I}$ to both $\boldsymbol{A}$ and $\boldsymbol{B}$, where the row count of $\boldsymbol{S}$ satisfies $J \ll I$. The resulting problem $\min_{\tilde{\boldsymbol{X}}} \|\boldsymbol{S}(\boldsymbol{A}\tilde{\boldsymbol{X}} - \boldsymbol{B})\|_F$ is cheaper to solve, and the solution $\tilde{\boldsymbol{X}}$ has residual arbitrarily close (for sufficiently high J) to the true minimum with high probability. We seek a sketching operator $\boldsymbol{S}$ with an efficiently computable action on $\boldsymbol{A}$, which is a Khatri-Rao product.

We choose $\boldsymbol{S}$ to be a *sampling* matrix with a single nonzero per row (see Section 4.3.2 for alternatives). This matrix extracts and reweights J rows from both $\boldsymbol{A}$ and $\boldsymbol{B}$, preserving

the sparsity of the matricized tensor $\boldsymbol{B}$. The cost to solve the j-th sketched subproblem is dominated by the downsampled MTTKRP operation $\text{mat}(\mathcal{T}, j)\boldsymbol{S}^\top \boldsymbol{S}\boldsymbol{U}_{\neq j}$, which has runtime

$$O\left(\text{nnz}(\text{mat}(\mathcal{T}, j)\boldsymbol{S}^\top)NR\right). \tag{4.7}$$

As Figure 4.3 (bottom) illustrates, $\text{mat}(\mathcal{T}, j)\boldsymbol{S}^\top$ typically has far fewer nonzeros than $\mathcal{T}$, enabling sampling to reduce the computation cost in Equation (4.6). To select indices to sample, we implement two algorithms that involve the *leverage scores* of the design matrix [Che+16; LK22; Bha+23]. Given a matrix $\boldsymbol{A} \in \mathbb{R}^{I \times R}$, the leverage score of row i is given by

$$\ell_i = \boldsymbol{A}[i,:]\left(\boldsymbol{A}^\top \boldsymbol{A}\right)^+ \boldsymbol{A}[i,:]^\top. \tag{4.8}$$

These scores induce a probability distribution over the rows of matrix $\boldsymbol{A}$, which we can interpret as a measure of importance. As the following theorem from Larsen and Kolda [LK22] (building on similar results by Drineas and Mahoney [DM16]) shows, sampling from either the exact or approximate distribution of statistical leverage guarantees, with high probability, that the solution to the downsampled problem has low residual with respect to the original problem.

Theorem 4.2.1 (Larsen and Kolda [LK22])**.** *Let $\boldsymbol{S} \in \mathbb{R}^{J \times I}$ be a sampling matrix for $\boldsymbol{A} \in \mathbb{R}^{I \times R}$ where each row i is sampled i.i.d. with probability p_i. Let $\beta = \min_{i \in [I]}(p_i R/\ell_i)$. For a sufficiently high universal constant C and any $\varepsilon, \delta \in (0,1)$, let the sample count be*

$$J = \frac{R}{\beta} \max\left(C \log \frac{R}{\delta}, \frac{1}{\varepsilon\delta}\right).$$

If $\tilde{\boldsymbol{X}} = \arg\min_{\tilde{\boldsymbol{X}}} \left\| \boldsymbol{S}\boldsymbol{A}\tilde{\boldsymbol{X}} - \boldsymbol{S}\boldsymbol{B} \right\|_F$, then

$$\left\| \boldsymbol{A}\tilde{\boldsymbol{X}} - \boldsymbol{B} \right\|_F^2 \leq (1+\varepsilon) \min_{\boldsymbol{X}} \|\boldsymbol{A}\boldsymbol{X} - \boldsymbol{B}\|_F^2.$$

with probability at least $1 - \delta$.

Here, $\beta \leq 1$ quantifies deviation of the sampling probabilities from the exact leverage score distribution, with a higher sample count J required as the deviation increases. The STS-CP algorithm samples from the exact leverage distribution with $\beta = 1$, achieving higher accuracy at the expense of increased sampling time. CP-ARLS-LEV samples from an approximate distribution with $\beta < 1$.

Sketching methods for tensor decomposition have been extensively investigated [Che+16; Ahl+20; LK22; Mal22; BBK18], both in theory and practice. Provided an appropriate sketch row count J and assumptions common in the optimization literature, some convergence guarantees for randomized ALS can be derived [GAY20].

4.3 Related Work

4.3.1 High-Performance ALS CP Decomposition

Significant effort has been devoted to optimizing the shared-memory MTTKRP using new data structures for the sparse tensor, cache-blocked computation, loop reordering strategies, and methods that minimize data races between threads [SK15; Nis+19; LMV18; Ngu+22; PK19; WKP23; KS23]. Likewise, several works provide high-performance algorithms for ALS CP decomposition in a distributed-memory setting. Smith and Karypis [SK16a] provide an algorithm that distributes load-balanced chunks of the sparse tensor to processors in an N-dimensional Cartesian topology. Factor matrices are shared among slices of the topology that require them, and each processor computes a local MTTKRP before reducing results with a subset of processors. The SPLATT library [Smi+15] implements this communication strategy and uses the compressed sparse fiber (CSF) format to accelerate local sparse MTTKRP computations on each processor.

Ballard et al. [BHR18]. use a similar communication strategy to compute the MTTKRP involved in dense nonnegative CP decomposition. They further introduce a dimension-tree algorithm that reuses partially computed terms of the MTTKRP between ALS optimization problems. DFacTo [CV14] instead reformulates the MTTKRP as a sequence of sparse matrix-vector products (SpMV), taking advantage of extensive research optimizing the SpMV kernel. Smith and Karypis [SK16a] note, however, that DFacTo exhibits significant communication overhead. Furthermore, the sequence of SpMV operations cannot take advantage of access locality within rows of the dense factor matrices, leading to more cache misses than strategies based on sparse-matrix-times-dense-matrix-multiplication (SpMM). GigaTensor [Kan+12] and BIGTensor [Par+16] use the MapReduce model in Hadoop to scale to distributed, fault-tolerant clusters. Ma and Solomonik [MS21] use pairwise perturbation to accelerate CP-ALS, reducing the cost of MTTKRP computations when ALS is sufficiently close to convergence using information from prior rounds.

Our work investigates variants of the Cartesian data distribution scheme adapted for a downsampled MTTKRP. We face challenges adapting either specialized data structures for the sparse tensor or dimension-tree algorithms. By extracting arbitrary nonzero elements from the sparse tensor, randomized sampling destroys the advantage conferred by formats such as CSF. Further, each least-squares solve requires a fresh set of rows drawn from the Khatri-Rao product design matrix, which prevents efficient reuse of results from prior MTTKRP computations.

Libraries such as the Cyclops Tensor Framework (CTF) [Sol+14] automatically parallelize distributed-memory contractions of both sparse and dense tensors. SpDISTAL [YAK22b] proposes a flexible domain-specific language to schedule sparse tensor linear algebra on a cluster, including the MTTKRP operation. The randomized algorithms investigated here

could be implemented on top of either library, but it is unlikely that current tensor algebra compilers can automatically produce the distributed samplers and optimized communication schedules that we contribute.

4.3.2 Alternate Sketching Algorithms and Tensor Decomposition Methods

Besides leverage score sampling, popular options for sketching Khatri-Rao products include Fast Fourier Transform-based sampling matrices [JKW20] and structured random sparse matrices (e.g. Countsketch) [Ahl+20; Dia+18]. The former method, however, introduces fill-in when applied to the sparse matricized tensor $\text{mat}(\mathcal{T}, j)$. Because the runtime of the downsampled MTTKRP is linearly proportional to the nonzero count of $\text{mat}(\mathcal{T}, j)S^\top$, the advantages of sketching are lost due to fill-in. While Countsketch operators do not introduce fill, they still require access to all nonzeros of the sparse tensor at every iteration, which is expensive when $\text{nnz}(\mathcal{T})$ ranges from hundreds of millions to billions.

Other algorithms besides ALS exist for large sparse tensor decomposition. Stochastic gradient descent (SGD, investigated by Kolda and Hong [KH20]) iteratively improves CP factor matrices by sampling minibatches of indices from $\mathcal{T}$, computing the gradient of a loss function at those indices with respect to the factor matrices, and adding a step in the direction of the gradient to the factors. Gradient methods are flexible enough to minimize a variety of loss functions besides the Frobenius norm error [HKD20], but require tuning additional parameters (batch size, learning rate) and a distinct parallelization strategy. The CCD++ algorithm [Yu+12] extended to tensors keeps all but one rank-1 component of the decomposition fixed and optimizes for the remaining component, in contrast to ALS which keeps all but one factor matrix fixed.

4.4 Distributed-Randomized CP Decomposition

In this section, we distribute Algorithm 10 to P processors when random sampling is used to solve the least-squares problem on line 6. Figure 4.4 (left) shows the initial data distribution of our factor matrices and tensor to processors, which are arranged in a hypercube of dimensions $P_1 \times ... \times P_N$ with $\prod_i P_i = P$. Matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$ are distributed by block rows among the processors to ensure an even division of computation, and we denote by $\boldsymbol{U}_i^{(p_j)}$ the block row of U_i owned by processor $p_j \in [P]$. We impose that all processors can access the Gram matrix $\boldsymbol{G}_i$ of each factor $\boldsymbol{U}_i$, which is computed by an `Allreduce` of the $R \times R$ matrices $\boldsymbol{U}_i^{(p_j)\top}\boldsymbol{U}_i^{(p_j)}$ across $p_j \in [1, ..., P]$. Using these matrices, the processors redundantly compute the overall Gram matrix $\boldsymbol{G}$ through Equation (4.5), and by extension $\boldsymbol{G}^+$.

With these preliminaries, each processor takes the following actions to execute steps 6-8 of Algorithm 10:

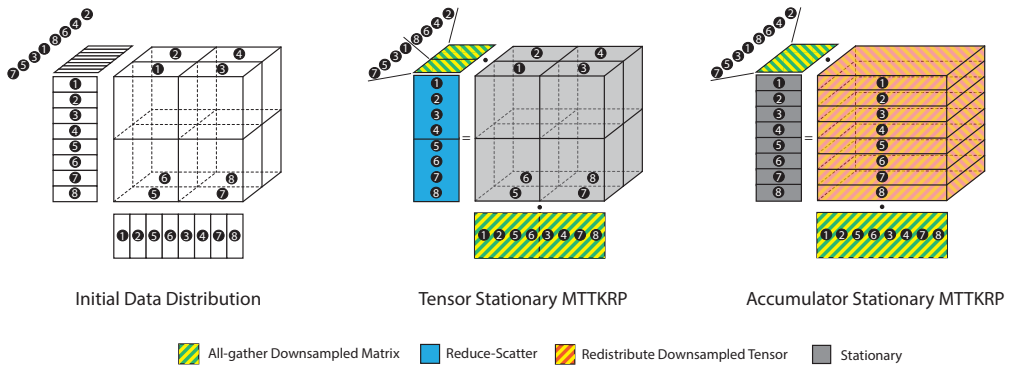


Figure 4.4: Initial data distribution and downsampled MTTKRP data movement for a 3D tensor, $P = 8$ processors. Rectangles along each side of the tensor illustrate factor matrices corresponding to each mode, divided by block rows among processors. Each black circle denotes the processor owning a block of a matrix or tensor; multiple circles on an object indicate replication of a piece of data. Colors / shading indicate communication collectives.

1. **Sampling and Allgather:** Sample rows of $\boldsymbol{U}_{\neq j}$ according to the leverage-score distribution and `Allgather` the rows to processors who require them. For non-randomized ALS, no sampling is required.

2. **Local Computation:** Extract the corresponding nonzeros from the local tensor owned by each processor and execute the downsampled MTTKRP, a sparse-dense matrix multiplication.

3. **Reduction and Postprocessing:** Reduce the accumulator of the sparse-dense matrix multiplication across processors, if necessary, and post-process the local factor matrix slice by multiplying with $\boldsymbol{G}^{+}$. Renormalize the factor matrix columns and update sampling data structures.

Multiple prior works establish the correctness of this schedule [SK16a; BHR18]. We now examine strategies for drawing samples (step 1), communicating factor matrix rows (steps 2 and 3), and performing local computation efficiently (step 2) tailored to the case of randomized least-squares.

4.4.1 New Distributed Sampling Strategies

Table 4.3 gives the asymptotic per-processor computation and communication costs to draw J samples in our distributed versions of CP-ARLS-LEV and STS-CP. We give detailed descriptions, as well as pseudo-code, for each sampling strategy in Sections 4.7.1 and 4.7.2. In this section, we briefly describe the accuracy characteristics and communication / computation

Sampler	Compute	Messages	Words Sent/Recv
d-CP-ARLS-LEV	JN/P	P	JN/P
d-STS-CP	$(JN/P)R^2 \log P$	$NP \log P$	$(J/P)NR \log P$

Table 4.3: Asymptotic Per-Processor Costs to Draw J Samples

Schedule	Words Communicated / Round
Non-Randomized TS	$2NR\left(\prod_{k=1}^{N} I_k/P\right)^{1/N}$
Sampled TS	$NR\left(\prod_{k=1}^{N} I_k/P\right)^{1/N}$
Sampled AS	$JRN(N-1)$

Table 4.4: Communication costs for downsampled MTTKRP.

patterns for each method. Table 4.3 does not include the costs to construct the sampling data structures in each algorithm, which are subsumed asymptotically by the matrix-multiplication $U_i^{(p_j)} \cdot G^+$ on each processor (step 3). The costs of all communication collectives are taken from Chan et al. [Cha+07].

CP-ARLS-LEV: The CP-ARLS-LEV algorithm by Larsen and Kolda [LK22] *approximates* the leverage scores in Equation (4.8) by the product of leverage scores for each factor matrix $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$. The leverage scores of the block row $\boldsymbol{U}_i^{(p_j)}$ owned by processor p_j are approximated by

$$\tilde{\ell}^{(p_j)} = \text{diag}\left(\boldsymbol{U}_i^{(p_j)} \boldsymbol{G}_i^{+} \boldsymbol{U}_i^{(p_j)\top}\right)$$

which, given the replication of $\boldsymbol{G}_i^{+}$, can be constructed independently by each processor in time $O\left(R^2 I_i/P\right)$. The resulting probability vector, which is distributed among P processors, can be sampled in expected time $O(J/P)$, assuming that the sum of leverage scores distributed to each processor is roughly equal (see Section 4.4.4 on load balancing for methods to achieve this). Multiplying by $(N-1)$ to sample independently from each matrix held constant, we get an asymptotic computation cost $O(JN/P)$ for the sampling phase. Processors exchange only a constant multiple of P words to communicate the sum of leverage scores that they hold locally and the exact number of samples they must draw, as well as a cost $O(JN/P)$ to evenly redistribute / postprocess the final sample matrix. While this algorithm is computationally efficient, it requires $J = \tilde{O}(R^{N-1}/(\varepsilon\delta))$ to achieve the (ϵ, δ)-guarantee from Theorem 4.2.1, which may lead to a higher runtime in the distributed-memory MTTKRP. Larsen and Kolda note that CP-ARLS-LEV sampling can be implemented without any communication at all if an entire factor matrix is assigned uniquely to a single processor, which can compute leverage scores and draw samples independently [LK22]. That said, assigning an entire factor

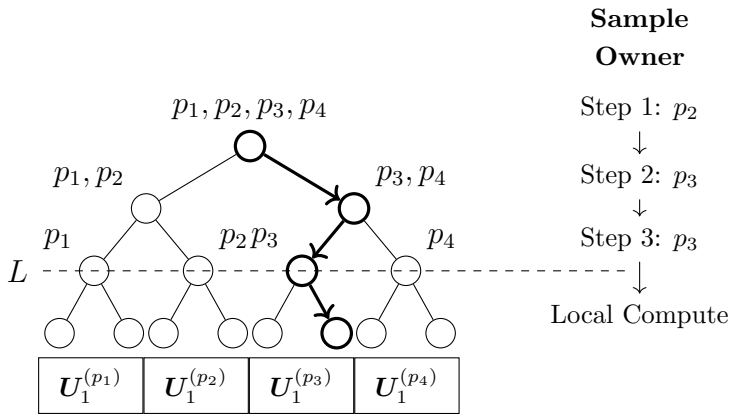


Figure 4.5: Example random walk in STS-CP to draw a single sample index from matrix $\boldsymbol{U}_1$, distributed to $P = 4$ processors. Annotations on the tree (left) indicate processors that share data for each node. The schedule to the right indicates the processor that owns the sample at each stage of the random walk. The sample begins randomly at p_2, then branches left to p_3 (p_4 shares node data and could also have been selected), involving communication of a vector corresponding to the sample from p_2 to p_3. The sample remains at p_3 for the remainder of the walk.

matrix to a single processor incurs higher communication costs in the MTTKRP phase of the algorithm and may be infeasible under tight memory constraints, leading to our adoption of a block-row distribution for the factors.

STS-CP: The STS-CP algorithm from Chapter 3 samples from the exact leverage distribution by executing a random walk on a binary tree data structure once for each of the $N - 1$ factor matrices held constant. Each leaf of the binary tree corresponds to a block of R rows from a factor matrix U_i and holds the $R \times R$ Gram matrix of that block row. Each internal node v holds a matrix $\boldsymbol{G}^v$ that is the sum of the matrices held by its children. Each sample begins with a unique vector $\boldsymbol{h}$ at the root of the tree. At each non-leaf node v, the algorithm computes $(\boldsymbol{h}^\top \boldsymbol{G}^{L(v)} \boldsymbol{h})/(\boldsymbol{h}^\top \boldsymbol{G}^v \boldsymbol{h})$. If this quantity is greater than a random number r unique to each sample, the algorithm sends the sample to the left subtree, and otherwise the right subtree. The process repeats until the random walk reaches a leaf and a row index is selected.

We distribute the data structure and the random walk as shown in Figure 4.5. We assume that P is a power of two to simplify our description, but our implementation makes no such restriction. Each processor p_j owns a subtree of the larger tree that corresponds to their block row $\boldsymbol{U}_i^{(p_j)}$. The roots of these subtrees all occur at the same depth $L = \log P$. Above level L, each node stores $2 \log P$ additional matrices, $\boldsymbol{G}^v$ and $\boldsymbol{G}^{L(v)}$, for each ancestor node v of its subtree.

To execute the random walks, each sample is assigned randomly to a processor which evaluates the branching threshold at the tree root. Based on the direction of the branch, the sample and corresponding vector $\boldsymbol{h}$ are routed to a processor that owns the required node information, and the process repeats until the walk reaches level L. The remaining steps do not require communication.

The replication of node information above level L requires communication overhead $O(R^2 \log P)$ using the classic bi-directional exchange algorithm for `Allreduce` [Cha+07]. For a batch of J samples, each level of the tree requires $O(JR^2)$ FLOPs to evaluate the branching conditions. Under the assumption that the final sampled rows are distributed evenly to processors, the computation and communication at each level are load balanced in expectation. Each processor has expected computation cost $O((J/P)NR^2 \log P)$ over all levels of the tree and all matrices $\boldsymbol{U}_i, 1 \leq i \leq N, i \neq k$. Communication of samples between tree levels is accomplished through `All-to-allv` collective calls, requiring $O(NP \log P)$ messages and $O((J/P)NR \log P)$ words sent / received in expectation by each processor.

4.4.2 A Randomization-Tailored MTTKRP Schedule

The goal of this section is to demonstrate that an optimal communication schedule for non-randomized ALS may incur unnecessary overhead for the randomized algorithm. In response, we will use a schedule where all communication costs scale with the number of random samples taken, enabling the randomized algorithm to decrease *communication costs* as well as computation. Table 4.4 gives lower bounds on the communication required for each schedule we consider, and we derive the exact costs in this section.

The two schedules that we consider are "tensor-stationary", where factor matrix rows are gathered and reduced across a grid, and "accumulator-stationary", where no reduction takes place. These distributions were compared by Smith and Karypis [SK16a] under the names "medium-grained" and "course-grained", respectively. Both distributions exhibit, under an even distribution of tensor nonzero entries and leverage scores to processors, ideal expected computation scaling. Therefore, we focus our analysis on communication. We begin by deriving the communication costs for non-randomized ALS under the tensor-stationary communication schedule, which we will then adapt to the randomized case.

Although our input tensor is sparse, we model the **worst-case** communication costs for the dense factor matrices with standard `Allgather` and `Reduce-scatter` primitives. For non-randomized (exact) ALS, the cost we derive matches that given by Smith and Karypis in their sparse tensor decomposition work [SK16a]. Furthermore consider the extremely sparse Reddit tensor, (nonzero fraction 4×10^{-10} [Smi+17]), which nonetheless exhibits an average of 571 nonzeros per fiber along the longest tensor mode and an average of 26,000 nonzeros per fiber aligned with the shortest tensor mode. The high per-fiber nonzero count induces a practical communication cost comparable to the worst-case bounds, a feature that Reddit

shares with other datasets in Table 4.5.

Exact Tensor-Stationary: The tensor-stationary MTTKRP algorithm is communication-optimal for dense CP decomposition [BHR18] and outperforms several other methods in practice for non-randomized sparse CP decomposition. [SK16a]. The middle image of Figure 4.4 illustrates the approach. During the k-th optimization problem in a round of ALS, each processor does the following:

1. For any $i \neq k$, participates in an `Allgather` of all blocks $U_i^{(p_j)}$ for all processors p_j in a slice of the processor grid aligned with mode k.
2. Executes an MTTKRP with locally owned nonzeros and the gathered row blocks.
3. Executes a `Reduce-scatter` with the MTTKRP result along a slice of the processor grid aligned with mode j, storing the result in $\boldsymbol{U}_k^{(p_j)}$

For non-randomized ALS, the gather step must only be executed once per round and can be cached. Then the communication cost for the `Allgather` and `Reduce-scatter` collectives summed over all $k = 1...N$ is

$$2\sum_{k=1}^{N} I_k R/P_k.$$

To choose the optimal grid dimensions P_k, we minimize the expression above subject to the constraint $\prod_{k=1}^{N} P_k = P$. Straightforward application of Lagrange multipliers leads to the optimal grid dimensions

$$P_k = I_k \left(P/\prod_{i=1}^{N} I_i\right)^{1/N}.$$

These are the same optimal grid dimensions reported by Ballard et al. [BHR18]. The communication under this optimal grid is

$$2NR\left(\prod_{k=1}^{N} I_k/P\right)^{1/N}.$$

Downsampled Tensor-Stationary: As Figure 4.4 illustrates, only factor matrix rows that are selected by the random sampling algorithm need to be gathered by each processor in randomized CP decomposition. Under the assumption that sampled rows are evenly distributed among the processors, the expected cost of gathering rows reduces to $JR(N-1)/P_k$ within slices along mode k. The updated communication cost under the optimal grid dimensions derived previously is

$$\frac{R\left(\prod_{k=1}^{N} I_k\right)^{1/N}}{P^{1/N}}\left[N + \sum_{k=1}^{N} \frac{J(N-1)}{I_k}\right].$$

The second term in the bracket arises from `Allgather` collectives of sampled rows, which is small if $J \ll I_k$ for all $1 \leq k \leq N$. The first term in the bracket arises from the `Reduce-scatter`, which is unchanged by the sampling procedure. Ignoring the second term in the expression above gives the second entry of Table 4.4.

Observe that this randomized method spends the same time on the reduction as the non-randomized schedule while performing significantly less computation, leading to diminished arithmetic intensity. On the other hand, this distribution may be optimal when the tensor dimensions I_k are small or the sample count J is high enough.

Downsampled Accumulator-Stationary: As shown by Smith and Karypis [SK16a], the accumulator-stationary data distribution performs poorly for non-randomized ALS. In the worst case, each processor requires access to all entries from all factors $\boldsymbol{U}_1, ..., \boldsymbol{U}_N$, leading to high communication and memory overheads. On the other hand, we demonstrate that this schedule may be optimal for *randomized* ALS on tensors where the sample count J is much smaller than the tensor dimensions. The rightmost image in Figure 4.4 illustrates the approach, which avoids the expensive `Reduce-scatter` collective. To optimize U_k, we keep the destination buffer for a block row of $\boldsymbol{U}_k$ stationary on each processor while communicating only sampled factor matrix rows and nonzeros of $\mathcal{T}$. Under this distribution, all sampled factor matrix rows must be gathered to all processors. The cost of the gather step for a single round becomes $O(JRN(N-1))$ (for each of N least-squares problems, we gather at most $J(N-1)$ rows of length R). Letting $\boldsymbol{S}_1, ..., \boldsymbol{S}_N$ be the sampling matrices for each ALS subproblem in a round, the number of nonzeros selected in problem j is $\text{nnz}(\text{mat}(\mathcal{T}, j)\boldsymbol{S}_j)$. These selected (row, column, value) triples must be redistributed as shown in Figure 4.4 via an `All-to-allv` collective call. Assuming that the source and destination for each nonzero are distributed uniformly among the processors, the expected cost of redistribution in least-squares problem j is $(3/P)\text{nnz}(\text{mat}(\mathcal{T}, j)\boldsymbol{S}_j^\top)$. The final communication cost is

$$JRN(N-1) + \frac{3}{P}\sum_{j=1}^{N} \text{nnz}(\text{mat}(\mathcal{T}, j)\boldsymbol{S}_j^\top). \tag{4.9}$$

The number of nonzeros sampled varies from tensor to tensor even when the sample count J is constant. That said, the redistribution exhibits perfect scaling (in expectation) with the processor count P. In practice, we avoid redistributing the tensor entries multiple times by storing N different representations of the tensor aligned with each slice of the processor grid, a technique that competing packages (e.g. DFacto [CV14], early versions of SPLATT [SK15]) also employ. This optimization eliminates the second term in Equation (4.9), giving the communication cost in the third row of Table 4.4. More importantly, observe that all communication scales linearly with the sample count J, enabling sketching to improve *both* the communication and computation efficiency of our algorithm. On the other hand, the

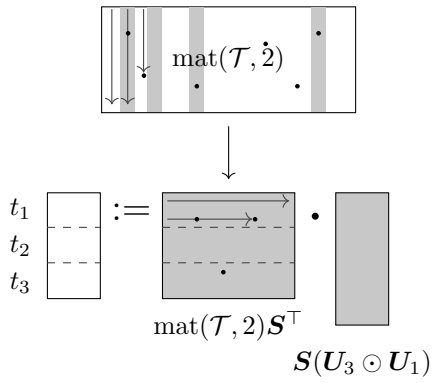


Figure 4.6: Shared-memory parallelization of downsampled MTTKRP procedure. Nonzero sparse coordinates in the sampled gray columns, initially sorted by column, are selected and remapped into a CSR matrix. The subsequent matrix multiplication is parallelized to threads t_1, t_2, t_3 without atomic operations or data races, since each thread is responsible for a unique block of the output.

term $JRN(N-1)$ does not scale with P, and we expect that gathering rows becomes a communication bottleneck for high processor counts.

4.4.3 Tensor Storage and Local MTTKRP

As mentioned in Section 4.4.2, we store different representations of the sparse tensor $\mathcal{T}$ across the processor grid to decrease communication costs. Each corresponds to a distinct matricization $\text{mat}(\mathcal{T}, j)$ for $1 \leq j \leq N$ used in the MTTKRP (see Figure 4.3). For non-randomized ALS, a variety of alternate storage formats have been proposed to reduce the memory overhead and accelerate the local computation. Smith and Karypis support a compressed sparse fiber format for the tensor in SPLATT [SK15; Smi+15], and Nisa et al. [Nis+19] propose a mixed-mode compressed sparse fiber format as an improvement. These optimizations cannot improve the runtime of our randomized algorithms because they are not conducive to sampling random nonzeros from $\mathcal{T}$.

Instead, we adopt the approach shown in Figure 4.6. The coordinates in each tensor matricization are stored in sorted order of their column indices, an analogue of compressed-sparse-column (CSC) format. With this representation, the random sampling algorithm efficiently selects columns of $\text{mat}(\mathcal{T}, j)$ corresponding to rows of the design matrix. The nonzeros in these columns are extracted and remapped to a compressed sparse row (CSR) format through a "sparse transpose" operation. The resulting CSR matrix participates in the sparse-dense matrix multiplication, which can be efficiently parallelized without data races on a team of shared-memory threads.

Tensor	Dimensions	NNZ	Prep.
Uber	$183 \times 24 \times 1.1K \times 1.7K$	3.3M	-
Amazon	$4.8M \times 1.8M \times 1.8M$	1.7B	-
Patents	$46 \times 239K \times 239K$	3.6B	-
Reddit	$8.2M \times 177K \times 8.1M$	4.7B	log

Table 4.5: Sparse Tensor Datasets from FROSTT.

The key to efficiency in the sparse matrix transpose is that the sampling process extracts only a small fraction of nonzero entries from the entire tensor. We leave reducing the memory footprint of our randomized algorithms as future work.

4.4.4 Load Balance

To ensure load balance among processors, we randomly permute the sparse tensor indices along each mode, a technique also used by SPLATT [SK16a]. These permutations ensure that each processor holds, in expectation, an equal fraction of nonzero entries from the tensor and an equal fraction of sampled nonzero entries. For highly-structured sparse tensors, random permutations do not optimize processor-to-processor communication costs, which packages such as Hypertensor [KU15] minimize through hypergraph partitioning. As Smith and Karypis [SK16a] demonstrate empirically, hypergraph partitioning is slow and memory-intensive on large tensors. Because our randomized implementations require just minutes on massive tensors to produce decompositions comparable to non-randomized ALS, the overhead of partitioning outweighs the modest communication reduction it may produce.

4.5 Experiments

Experiments were conducted on CPU nodes of NERSC Perlmutter, a Cray HPE EX supercomputer. Each node has 128 physical cores divided between two AMD EPYC 7763 (Milan) CPUs. Nodes are linked by an HPE Slingshot 11 interconnect.

Our implementation is written in C++ and links with OpenBLAS 0.3.21 for dense linear algebra. We use a simple Python wrapper around the C++ implementation to facilitate benchmarking. We use a hybrid of MPI message-passing and OpenMP shared-memory parallelism in our implementation, which is available online at `https://github.com/vbharadwaj-bk/rdist_tensor`.

Our primary baseline is the SPLATT , the Surprisingly Parallel Sparse Tensor Toolkit [SK16a; Smi+15]. SPLATT is a scalable CP decomposition package optimized for both communication costs and local MTTKRP performance through innovative sparse tensor storage structures.

As a result, it remains one of the strongest libraries for sparse tensor decomposition in head-to-head benchmarks against other libraries [RSK19; Nis+19; KS23]. We used the default medium-grained algorithm in SPLATT and adjusted the OpenMP thread count for each tensor to achieve the best possible performance to compare against.

Table 4.5 lists the sparse tensors used in our experiments, all sourced from the Formidable Repository of Open Sparse Tensors and Tools (FROSTT) [Smi+17]. Besides Uber, which was only used to verify accuracy due to its small size, the Amazon, Patents, and Reddit tensors are the only members of FROSTT at publication time with over 1 billion nonzero entries. These tensors were identified to benefit the most from randomized sampling since the next largest tensor in the collection, NELL-1, has 12 times fewer nonzeros than Amazon. We computed the logarithm of all values in the Reddit tensor, consistent with established practice [LK22].

4.5.1 Correctness at Scale

Table 4.6 gives the average fits (5 trials) of decompositions produced by our distributed-memory algorithms. The fit [LK22] between the decomposition $\tilde{\mathcal{T}} = [\boldsymbol{\sigma}; \boldsymbol{U}_1, ..., \boldsymbol{U}_N]$ and the ground-truth $\mathcal{T}$ is defined as

$$\text{fit}(\tilde{\mathcal{T}}, \mathcal{T}) = 1 - \frac{\left\| \tilde{\mathcal{T}} - \mathcal{T} \right\|_F}{\|\mathcal{T}\|_F}.$$

A fit of 1 indicates perfect agreement between the decomposition and the input tensor. We used $J = 2^{16}$ for our randomized algorithms to test our implementations on configurations identical to those in prior work [LK22; Bha+23]. To test both the distributed-memory message passing and shared-memory threading parts of our implementation, we used 32 MPI ranks and 16 threads per rank across 4 CPU nodes. We report accuracy for the accumulator-stationary versions of our algorithms and checked that the tensor-stationary variants produced the same mean fits. The "Exact" column gives the fits generated by SPLATT. ALS was run for 40 rounds on all tensors except Reddit, for which we used 80 rounds.

The accuracy of both d-CP-ARLS-LEV and d-STS-CP match the shared-memory prototypes in the original works [LK22; Bha+23]. As theory predicts, the accuracy gap between d-CP-ARLS-LEV and d-STS-CP widens at higher rank. The fits of our methods improves by increasing the sample count J at the expense of higher sampling and MTTKRP runtime.

4.5.2 Speedup over Baselines

Figure 4.7 shows the speedup of our randomized distributed algorithm per ALS round over SPLATT at 4 nodes and 16 nodes. We used the same configuration and sample count for each tensor as Table 4.6. On Amazon and Reddit at rank 25 and 4 nodes, d-STS-CP achieves

Tensor	R	d-CP-ARLS-LEV	d-STS-CP	Exact
Uber	25	0.187	0.189	0.190
	50	0.211	0.216	0.218
	75	0.218	0.230	0.232
Amazon	25	0.338	0.340	0.340
	50	0.359	0.366	0.366
	75	0.368	0.381	0.382
Patents	25	0.451	0.451	0.451
	50	0.467	0.467	0.467
	75	0.475	0.475	0.476
Reddit	25	0.0583	0.0592	0.0596
	50	0.0746	0.0775	0.0783
	75	0.0848	0.0910	0.0922

Table 4.6: Average Fits, $J = 2^{16}$, 32 MPI Ranks, 4 Nodes

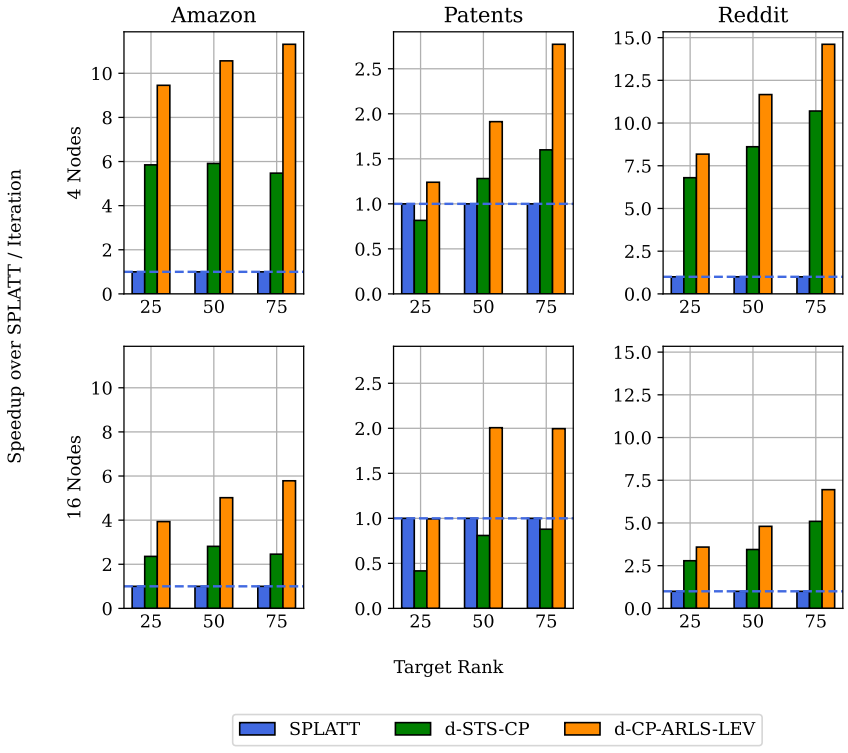


Figure 4.7: Average speedup per ALS iteration of our distributed randomized algorithms over SPLATT (5 trials, $J = 2^{16}$).

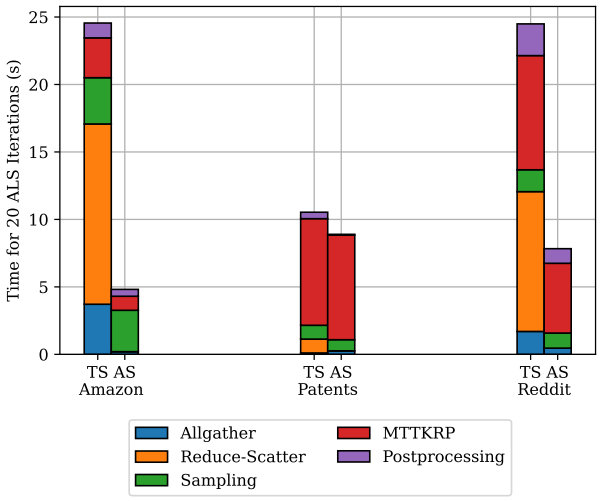


Figure 4.8: Average runtime (5 trials, $R = 25$) per activity for tensor-stationary and accumulator-stationary distributions with 32 MPI ranks over 4 nodes.

a speedup in the range 5.7x-6.8x while d-CP-ARLS-LEV achieves between 8.0-9.5x. We achieve our most dramatic speedup at rank 75 on the Reddit tensor, with d-STS-CP achieving 10.7x speedup and d-CP-ARLS-LEV achieving 14.6x. Our algorithms achieve less speedup compared to SPLATT on the denser Patents tensor. Here, a larger number of nonzero entries are selected by randomized sampling, with a significant computation bottleneck in the step that extracts and re-indexes the tensor entries. The bottom half of Figure 4.7 shows that d-STS-CP maintains at least 2x speedup over SPLATT even at 16 nodes / 2048 CPU cores on Amazon and Reddit, but has less advantage on the Patents tensor. Table 4.6 quantifies the accuracy sacrificed for the speedup, which can be changed by adjusting the sample count at each least-squares solve. As Figure 4.2 shows, both of our randomized algorithms make faster progress than SPLATT, with d-STS-CP producing a comparable rank-100 decomposition of the Reddit tensor in under two minutes.

4.5.3 Comparison of Communication Schedules

Figure 4.8 breaks down the runtime per phase of the d-STS-CP algorithm for the tensor-stationary and accumulator-stationary schedules on 4 nodes. To illustrate the effect of sampling on the row gathering step, we gather all rows (not just those sampled) for the tensor-stationary distribution, a communication pattern identical to SPLATT. Observe that the `Allgather` collective under the accumulator-stationary schedule is significantly cheaper for Amazon and Reddit, since only sampled rows are communicated. As predicted, the `Reduce-scatter` collective accounts for a significant fraction of the runtime for the tensor-stationary distribution on Amazon and Reddit, which have tensor dimensions in the millions.

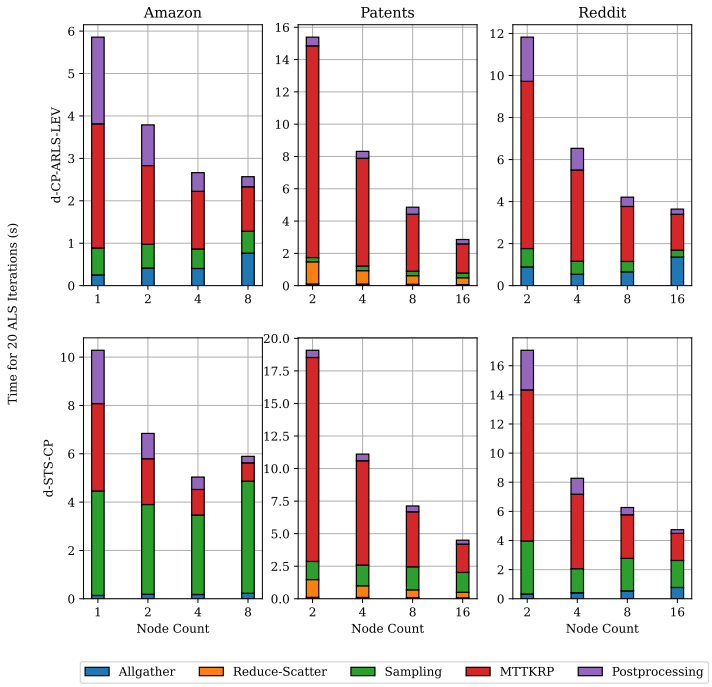


Figure 4.9: Average runtime (5 trials) per activity vs. CPU core count, $R = 25$. Each node has 128 CPU cores, and 8 MPI ranks were used per node.

On both tensors, the runtime of this collective is greater than the time required by all other phases combined in the accumulator-stationary schedule. By contrast, both schedules perform comparably on Patents. Here, the `Reduce-scatter` cost is marginal due to the smaller dimensions of the tensor.

We conclude that sparse tensors with large dimensions can benefit from the accumulator-stationary distribution to reduce communication costs, while the tensor-stationary distribution is optimal for tensors with higher density and smaller dimensions. The difference in MTTKRP runtime between the two schedules is further explored in Section 4.5.6.

4.5.4 Strong Scaling and Runtime Breakdown

Figure 4.9 gives the runtime breakdown for our algorithms at varying core counts. Besides the `Allgather` and `Reduce-scatter` collectives used to communicate rows of the factor matrices, we benchmark time spent in each of the three phases identified in Section 4.4: sample identification, execution of the downsampled MTTKRP, and post-processing factor matrices.

With its higher density, the Patents tensor has a significantly larger fraction of nonzeros

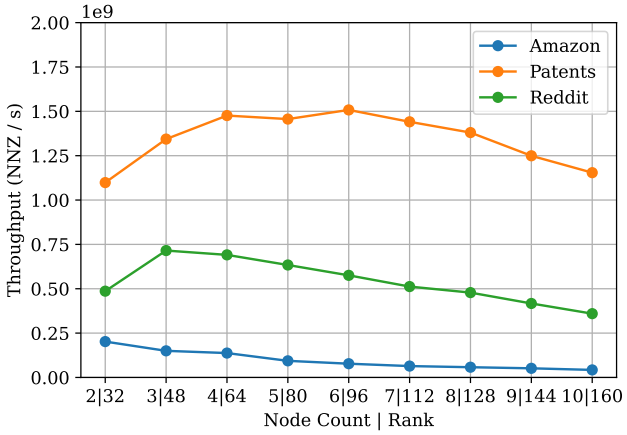


Figure 4.10: Average throughput (3 trials per data point) of the d-STS-CP algorithm vs. increasing node count and rank, measured as the average number of nonzeros iterated over in the MTTKRP per second of total algorithm runtime (higher is better). Ideal scaling is a horizontal line. The ratio of node count to rank was kept constant at 16. d-STS-CP was chosen to preserve decomposition accuracy at high ranks.

randomly sampled at each linear least-squares solve. As a result, most ALS runtime is spent on the downsampled MTTKRP. The Reddit and Amazon tensors, by contrast, spend a larger runtime portion on sampling and post-processing the factor matrices due to their larger mode sizes. Scaling beyond 1024 cores for the Amazon tensor is impeded by the relatively high sampling cost in d-STS-CP, a consequence of repeated `All-to-allv` collective calls. The high sampling cost is because the Amazon tensor has side-lengths in the millions along all tensor modes, leading to deeper trees for the random walks in STS-CP.

4.5.5 Weak Scaling with Target Rank

We measure weak scaling for our randomized algorithms by recording the throughput (nonzero entries processed in the MTTKRP per second of total algorithm runtime) as both the processor count and target rank R increase proportionally. We keep the ratio of node count to rank R constant at 16. We use a fixed sample count $J = 2^{16}$, and we benchmark the d-STS-CP algorithm to ensure minimal accuracy loss as the rank increases.

Although the FLOP count of the MTTKRP is linearly proportional to R (see Equation (4.6)), we expect the efficiency of the MTTKRP to *improve* with increased rank due to spatial cache access locality in the longer factor matrix rows, a well-documented phenomenon [Akt+14]. On the other hand, the sampling runtime of the d-STS-CP algorithm grows quadratically with the rank R (see Table 4.3). The net impact of these competing effects is determined by the density and dimensions of the sparse tensor.

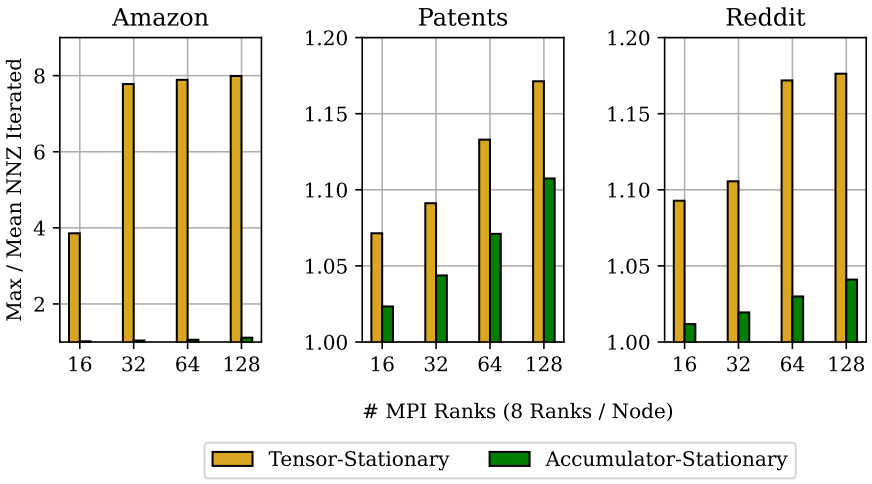


Figure 4.11: Avg. Load imbalance, defined as maximum / mean nonzeros iterated over by any MPI process, for d-STS-CP. 8 MPI Ranks / Node, 5 Trials.

Figure 4.10 shows the results of our weak scaling experiments. Because ALS on the Amazon tensor spends a large fraction of time drawing samples (see Figure 4.9), its throughput suffers with increasing rank due to the quadratic cost of sampling. At the other extreme, our algorithm spends little time sampling from the Patents tensor with its smaller dimensions, enabling throughput to increase due to higher cache spatial locality in the factor matrices. The experiments on Reddit follow a middle path between these extremes, with performance dropping slightly at high rank due to the cost of sampling.

4.5.6 Load Imbalance

Besides differences in the communication times of the tensor-stationary and accumulator-stationary schedules, Figure 4.8 indicates a runtime difference in the downsampled MTTKRP between the two schedules. Figure 4.11 offers an explanation by comparing the load balance of these methods. We measure load imbalance (averaged over 5 trials) as the maximum number of nonzeros processed in the MTTKRP by any MPI process over the mean of the same quantity.

The accumulator-stationary schedule yields better load balance over all tensors, with a dramatic difference for the case of Amazon. The latter exhibits a few rows of the Khatri-Rao design matrix with high statistical leverage and corresponding fibers with high nonzero counts, producing the imbalance. The accumulator-stationary distribution (aided by the load balancing random permutation) distributes the nonzeros in each selected fiber across all P processors, correcting the imbalance.

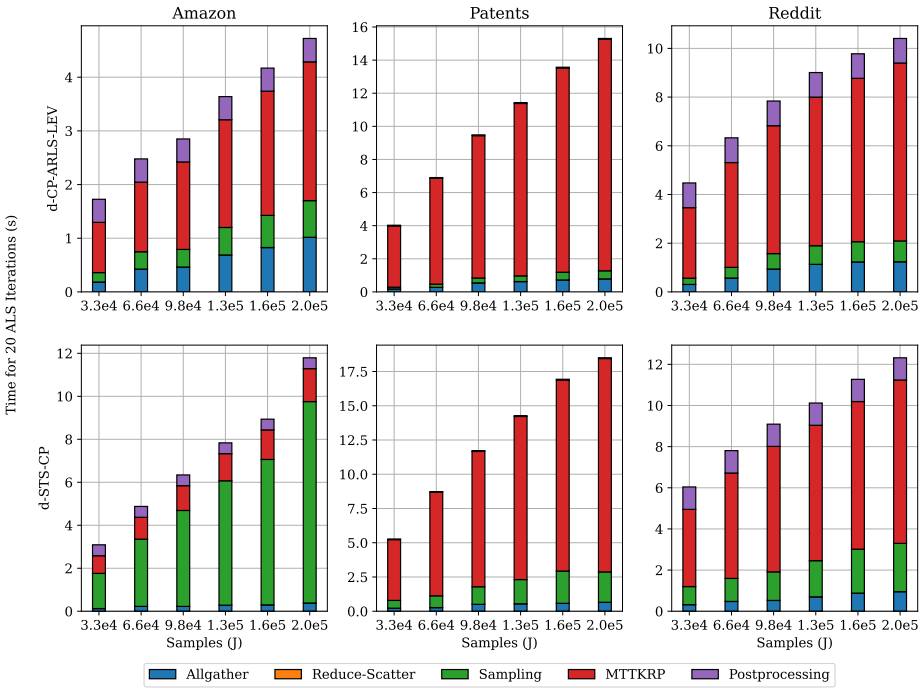


Figure 4.12: Runtime breakdown vs. sample count, $R = 25$. 512 CPU cores, 5 trials, accumulator-stationary distribution.

4.5.7 Impact of Sample Count

In prior sections, we used the sample count $J = 2^{16}$ to establish a consistent comparison with prior work. Figure 4.12 demonstrates the runtime impact of increasing the sample count for both of our algorithms on all three tensors. For all experiments but one, the MTTKRP component of the runtime increases the most as J gets larger. For d-STS-CP on the Amazon tensor, the runtime increase owes primarily to the higher cost of sample selection. The higher sampling time for d-STS-CP on Amazon is explained in Section 4.5.4. Figure 4.13 gives the final fits after running our randomized algorithms for varying sample counts. The increase in accuracy is minimal beyond $J = 2^{16}$ for d-STS-CP on Amazon and Reddit. Both algorithms perform comparably on Patents. These plots suggest that sample count as low as $J = 2^{16}$ is sufficient to achieve competitive performance with libraries like SPLATT on large tensors.

4.6 Conclusions and Further Work

We have demonstrated in this work that randomized CP decomposition algorithms are competitive at the scale of thousands of CPU cores with state-of-the-art, highly-optimized non-randomized libraries for the same task. Future work includes improving the irregular

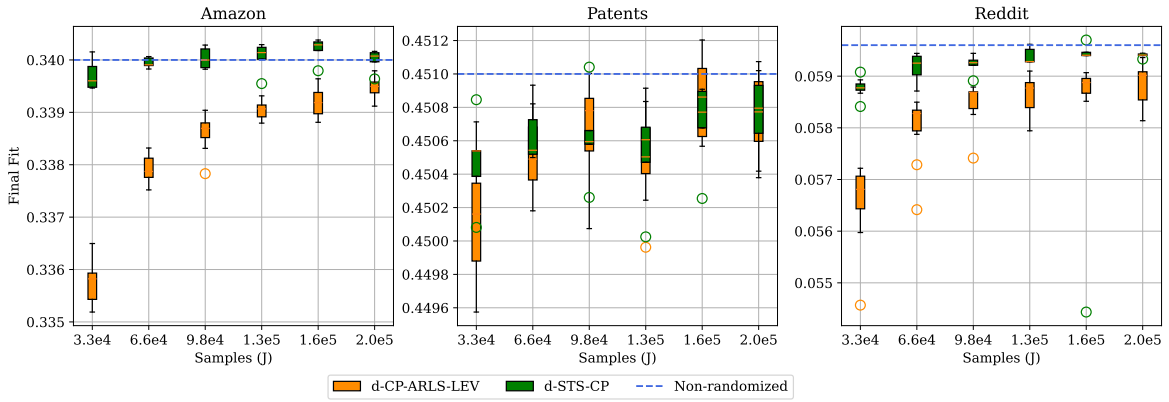


Figure 4.13: Final fit of randomized CP decomposition for varying sample count J. Horizontal dashed lines indicate the fit produced by SPLATT. ALS was run for 40 iterations on Amazon and Patents, 80 iterations on Reddit, and for 10 trials each. All other experimental configuration is identical to Figure 4.12.

communication pattern of the d-STS-CP algorithm, as well as deploying our algorithm on massive real-world tensors larger than those offered by FROSTT.

4.7 Full Algorithm Descriptions

4.7.1 Distributed CP-ARLS-LEV Sampling

Let $i_1, ..., i_{N-1}$ denote row indices from factor matrices $\boldsymbol{U}_1, ..., \boldsymbol{U}_{N-1}$ that uniquely identify a row from the Khatri-Rao product $\boldsymbol{U}_{\neq N}$. To efficiently sample according to an *approximate leverage score distribution* on the rows of $\boldsymbol{U}_{\neq N}$, the CP-ARLS-LEV algorithm by Larsen and Kolda [LK22] weights each row by

$$\tilde{\ell}_{i_1,...,i_{N-1}} := \prod_{k=1}^{N-1} \boldsymbol{U}_k \left[i_k, :\right] \boldsymbol{G}_k^{+} \boldsymbol{U}_k \left[i_k, :\right]^\top$$

where $\boldsymbol{G}_k := \boldsymbol{U}_k^\top \boldsymbol{U}_k$ for all k. Because each weight in the distribution above is a product of scores from each factor, we can draw $i_1, ..., i_{N-1}$ independently and concatenate the indices to assemble one row sample. Given that the factors are distributed by block rows among processors, the main challenge is to sample without gathering the probability weight vector for each $\boldsymbol{U}_k$ to a single processor.

Algorithms 11 and 12 give full procedures to build the distributed CP-ARLS-LEV data structure and draw samples from it, respectively. The build algorithm is called for all $\boldsymbol{U}_i$, $1 \leq i \leq N$, before the ALS algorithm begins. It is also called each time a matrix $\boldsymbol{U}_i$ is

Algorithm 11 CP-ARLS-LEV-build $\left(\boldsymbol{U}_i^{(p_j)}\right)$)

1: $\boldsymbol{G}_i := \textbf{Allreduce}\left(\boldsymbol{U}_i^{(p_j)\top}\boldsymbol{U}_i^{(p_j)}\right)$
2: $\textbf{dist}_i^{(p_j)} := \text{diag}\left(\boldsymbol{U}_i^{(p_j)}\boldsymbol{G}_i^{+}\boldsymbol{U}_i^{(p_j)\top}\right)$
3: $C_i^{(p_j)} := \left\|\textbf{dist}_i^{(p_j)}\right\|_1$
4: $\textbf{dist}_i^{(p_j)} /= C_i^{(p_j)}$
5: **Postcondition:** $\boldsymbol{G}_i^{+}$, $\textbf{dist}_i^{(p_j)}$, and $C_i^{(p_j)}$ are initialized on each processor.

updated in an ALS round. Each procedure is executed synchronously by all processors p_j, $1 \leq j \leq P$. Recall further that we define $\boldsymbol{U}_i^{(p_j)}$ as the block row of the i-th factor matrix uniquely owned by processor p_j. Algorithm 11 allows all processors to redundantly compute the Gram matrix $\boldsymbol{G}_i$ and the normalized local leverage score distribution on the block row $\boldsymbol{U}_i^{(p_j)}$.

Algorithm 12 enables each processor to draw samples from the Khatri-Rao product $\boldsymbol{U}_{\neq k}$. For each index $i \neq k$, each processor determines the fraction of J rows drawn from its local block using a consistent multinomial sample according to the weights $C_i^{(p_j)}$, $1 \leq j \leq P$. By *consistent*, we mean that each processor executes the multinomial sampling using a pseudorandom number generator with a common seed that is shared among all processors. The result of this operation is a vector $\textbf{SC}^{\text{loc}} \in \mathbb{Z}^P$ which gives the sample count each processor should draw locally. Each processor then samples from its local distribution. At the end of the algorithm, each row of $\boldsymbol{X}$ contains a sample drawn according to the approximate leverage score distribution.

We note that the sampling algorithm, as presented, involves the `Allgather` of a $J \times N$ sampling matrix followed by a random permutation. We use this procedure in our code, since we found that the communication cost $O(JN)$ was negligible for the range of sample counts we used. However, this cost can be reduced to $O(JN/P)$, in expectation, with an `All-to-allv` communication pattern that permutes the indices without gathering them to a single processor.

4.7.2 Distributed STS-CP Sampling

We use the same variables defined at the beginning of Section 4.7.1. To draw samples from the *exact leverage score distribution*, the STS-CP algorithm conditions each row index draw i_k on draws $i_1, ..., i_{k-1}$. To formalize this, let $\hat{i}_1, ..., \hat{i}_{N-1}$ be random variables for each index that jointly follow the exact leverage distribution. Suppose we have already sampled $\hat{i}_1 = i_1, ..., \hat{i}_{k-1} = i_{k-1}$, and let $\boldsymbol{h} = \circledast_{j=1}^{k-1} \boldsymbol{U}_j\left[i_j, :\right]^\top$ be the product of these sampled rows.

Algorithm 12 CP-ARLS-LEV-sample (k, J)

Require: Vectors $\mathbf{dist}_i^{(p_j)}$, and normalization constants $C_i^{(p_j)}$.
Initialize sample matrix $X \in \mathbb{Z}^{J \times N}$ on all processors.
for $i = 1...N$, $i \neq k$ **do**
  $\boldsymbol{C} := \mathbf{Allgather}\left(C_i^{(p_j)}\right)$
  $W = \sum_{\ell=1}^{P} C\left[\ell\right]$
  $\mathbf{SC}^{\text{loc}} :=$ consistent-multinomial$(\left[\boldsymbol{C}\left[1\right]/W, ..., \boldsymbol{C}\left[P\right]/W\right], J)$
  $\mathbf{samples}^{\text{loc}} :=$ sample $\left(\mathbf{dist}_i^{(p_j)}, \mathbf{SC}^{\text{loc}}\left[j\right]\right)$
  $\boldsymbol{X}\left[:, i\right] :=$ `Allgather` $\left(\mathbf{samples}^{\text{loc}}\right)$ //See note
  Perform a consistent random permutation of $\boldsymbol{X}\left[:, i\right]$
return $\boldsymbol{X}$, a set of samples from the Khatri-Rao product $\boldsymbol{U}_{\neq k}$.

conditional probability of $\hat{i}_k = i_k$ is

$$p(\hat{i}_k = i_k \mid \hat{i}_{<k} = i_{<k}) \propto \left(\boldsymbol{U}_k\left[i_k, :\right]^\top \circledast \boldsymbol{h}\right)^\top \boldsymbol{G}_{>k} \left(\boldsymbol{U}_k\left[i_k, :\right]^\top \circledast \boldsymbol{h}\right)$$

where $\boldsymbol{G}_{>k} = \boldsymbol{G}^+ \circledast \circledast_{j=k}^{N-1} \boldsymbol{G}_i$ (see Section 3.3). The STS-CP algorithm exploits this formula to efficiently sample from the exact leverage distribution.

Algorithms 13 and 14 give procedures to build and sample from the distributed data structure for STS-CP, which are analogues of Algorithms 11 and 12 for CP-ARLS-LEV. To simplify our presentation, we assume that the processor count P is a power of two. The general case is a straightforward extension (see Chan et. al. [Cha+07]), and our implementation makes no restriction on P. The build procedure in Algorithm 13 computes the Gram matrix G_i for each matrix U_i using a the bidirectional exchange algorithm for Allreduce [Cha+07]. The difference is that each processor caches the intermediate matrices that arise during the reduction procedure, each uniquely identified with internal nodes of the binary tree in Figure 4.5.

In the sampling algorithm, the cached matrices are used to determine the index of a row drawn from U_i via binary search. The matrix of sample indices X and sampled rows H are initially distributed by block rows among processors. Then for each matrix U_i, $i \neq k$, a random number is drawn uniformly in the interval $[0, 1]$ for each sample. By stepping down levels of the tree, J binary searches are computed in parallel to determine the containing bin of each random draw. At each level, the cached matrices tell the program whether to branch left or right by computing the branching threshold T, which is compared to the random draw r. The values in each column of $X^{(p_j)}$ hold the current node index of each sample at level ℓ of the search. At level $L = \log_2 P$, the algorithm continues the binary search locally on each processor until a row index is identified (a procedure we denote as "local-STS-CP". For more

Algorithm 13 STS-CP-build $\left(\boldsymbol{U}_i^{(p_j)}\right)$)

$\tilde{\boldsymbol{G}}_{\log_2 P} := \boldsymbol{U}_i^{(p_j)\top}\boldsymbol{U}_i^{(p_j)}$
for $\ell = \log_2 P ... 2$ **do**
  Send $\tilde{\boldsymbol{G}}_\ell$ to sibling of ancestor at level ℓ, and **receive** the corresponding matrix $\tilde{\boldsymbol{G}}_{\text{sibling}}$.
  Assign $\tilde{\boldsymbol{G}}_{\ell-1} = \tilde{\boldsymbol{G}}_{\text{sibling}} + \tilde{\boldsymbol{G}}_\ell$
  if Ancestor at level ℓ is a left child **then**
    $\tilde{\boldsymbol{G}}^L_{\ell-1} := \tilde{\boldsymbol{G}}_\ell$
  else
    $\tilde{\boldsymbol{G}}^L_{\ell-1} := \tilde{\boldsymbol{G}}_{\text{sibling}}$
Assign $\boldsymbol{G}_i := \tilde{\boldsymbol{G}}_1$
Postcondition: Each processor stores a list of *partial gram matrices* $\tilde{\boldsymbol{G}}_\ell$ and $\boldsymbol{G}^L_\ell$, from the root to its unique tree node. $\boldsymbol{G}_i$ is initialized.

details, see Chapter 3. At the end of the algorithm, the sample indices in X are correctly drawn according to the exact leverage scores of $U_{/=} k$. The major communication cost of this algorithm stems from the **All-to-allv** collective between levels of the binary search. Because a processor may not have the required matrices $\tilde{G}_\ell, \tilde{G}^L_\ell$ to compute the branching threshold for a sample, the sample must be routed to another processor that owns the information.

Algorithm 14 STS-CP-sample (k, J))

Initialize $\boldsymbol{X} \in \mathbb{Z}^{J \times N}$, $\boldsymbol{H} \in \mathbb{R}^{J \times (R+1)}$ distributed by block rows. Let $\boldsymbol{X}^{(p_j)}, \boldsymbol{H}^{(p_j)}$ be the block rows assigned to p_j.
$\boldsymbol{X}^{(p_j)} := [0], \boldsymbol{H}^{(p_j)} := [1]$
for $i = 1...N, i \neq k$ **do**
  $\boldsymbol{G}_{>k} := \boldsymbol{G}^{+} \circledast \circledast_{\ell=k+1}^{N} \boldsymbol{G}_i$
  $\boldsymbol{H}^{(p_j)}[:, R+1] :=$ uniform-samples($[0, 1]$)
  for $\ell = 1... \log P - 1$ **do**
    $J^{\text{loc}} =$ row-count $\left(\boldsymbol{X}^{(p_j)}\right)$
    for $k = 1...J^{\text{loc}}$ **do**
      $r := \boldsymbol{H}^{(p_j)}[k, R+1]$
      $\boldsymbol{h} := \boldsymbol{H}^{(p_j)}[k, 1:R]$
      $\boldsymbol{X}^{(p_j)}[i, k] * = 2$
      $T = \boldsymbol{h}^{\top} \left(\tilde{\boldsymbol{G}}_{\ell}^{L} \circledast \boldsymbol{G}_{>k}\right) \boldsymbol{h} / \left(\boldsymbol{h}^{\top} (\tilde{\boldsymbol{G}}_{\ell} \circledast \boldsymbol{G}_{>k}) \boldsymbol{h}\right)$
      if $r \geq T$ **then**
        $\boldsymbol{X}^{(p_j)}[i, k]$ += 1
        $r := (r - T)/(1 - T)$
      else
        $r := r/T$
      $\boldsymbol{H}^{(p_j)}[k, R+1] := r$
    Execute an **All-to-allv** call to redistribute $\boldsymbol{X}^{(p_j)}$, $\boldsymbol{H}^{(p_j)}$ according to the binary-tree data structure.
  $J^{\text{loc}} =$ row-count $\left(X^{(p_j)}\right)$
  for $k = 1...J^{\text{loc}}$ **do**
    idx := local-STS-CP($\boldsymbol{H}^{(p_j)}[k, 1:R], \tilde{\boldsymbol{G}}_{\log 2P}, \boldsymbol{G}_{>k}, r$)
    $\boldsymbol{X}^{(p_j)}[i, k] :=$ idx
    $\boldsymbol{H}^{(p_j)}[k, 1:R] * = \boldsymbol{U}_i^{(p_j)}[\text{idx} - I_i p_j / P, :]$
return $\boldsymbol{X}^{(p_j)}, \boldsymbol{H}^{(p_j)}$

Chapter 5

Sketches for Orthonormal Core Chains

We now examine subspace embeddings of a tensor structure distinct from the Khatri-Rao product. Specifically, we consider a linear chain of tensor cores that are related by *tensor contraction*, a generalization of standard matrix-matrix multiplication. These structures arise in computations that involve the tensor train decomposition, which appears in domains spanning quantum physics to machine learning [Per+07; Ose11]. We focus on the special case where the matrix flattening of each tensor core is *orthonormal*, in which case the matricization of the entire core chain is also orthonormal.

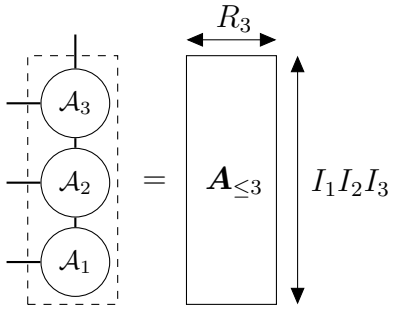


Figure 5.1: A tensor core chain with a dangling edge and its corresponding matrix representation. The height of the *core chain matricization* is generally exponential in the core count N.

While the target matrix now has an ideal condition number, constructing a subspace embedding via sampling is still a formidable task. As Dereziński and Mahoney [DM21] note, random uniform sampling requires a sample count proportional to the matrix *coherence* rather than condition number. The former, defined as the ratio between the largest and average row leverage scores of the target, inflates the worst-case sample count significantly compared to row-norm and leverage score sampling.

Without much additional difficulty, we will use Lemma 3.3.2 to construct an efficient row

sampler for orthonormal tensor core chains. Rows are sampled proportional to the squared row norms of the matricization of the core chain, which are equal to the row leverage scores. The resulting sampler selects rows with asymptotic runtime only a log factor worse than uniform random sampling, but with significantly higher accuracy.

Subspace embeddings of the Khatri-Rao product immediately yield an accelerated randomized algorithm to compute the CP decomposition via alternating least-squares. Likewise, subspace embeddings of orthonormal core chains yield efficient ALS algorithms for the *tensor train* decomposition, and we demonstrate the utility of our approach by decomposing sparse tensors from the FROSTT collection [Smi+17]. Our randomized methods exhibit up to 26x speedup over a simple baseline code to decompose sparse tensors from Chapter 3, achieving up to 99% of the fit of non-randomized TT-ALS.

While our method is a viable and competitive choice to decompose massive sparse tensors, the components of tensor trains are more difficult to interpret than those of the CP decomposition. The relative error of tensor train decomposition was also higher than CP decomposition in our experiments. With these disclaimers, we believe that our sampling data structure is interesting in its own right. For comparison, drawing samples from a Kronecker product of orthonormal matrices is much easier, only requiring simple independent sampling from each matrix. Meanwhile, multiple nontrivial optimizations are needed to efficiently sample from an orthonormal core chain. Our core chain sampler also exhibits a creative application of Lemma 3.3.2 and may find use outside tensor decomposition problems.

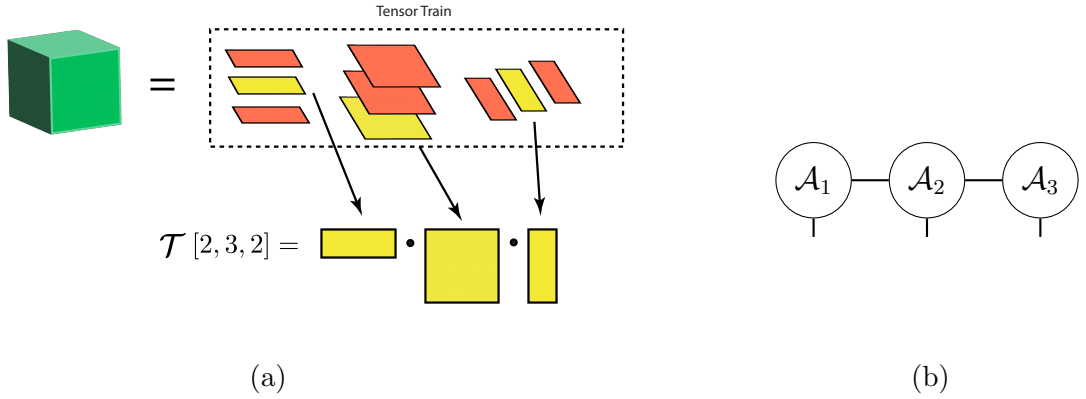
5.1 Introduction

Let $\mathcal{A}_1, ..., \mathcal{A}_j$ be a set of three-dimensional tensors with $\mathcal{A}_k \in \mathbb{R}^{R_{k-1} \times I_k \times R_k}$ for $1 \leq k \leq j$. We call each tensor a *core*, and we impose $R_0 = 1$. Consider the matrix $\boldsymbol{A}^L_{\leq j} \in \mathbb{R}^{\prod_{k=1}^{j} I_k \times R_j}$ where each row is specified by

$$\boldsymbol{A}^L_{\leq j}\left[(i_1, ..., i_j), :\right] = \mathcal{A}_1\left[:, i_1, :\right] \cdot ... \cdot \mathcal{A}_j\left[:, i_j, :\right]. \tag{5.1}$$

Figure 5.1 illustrates the matrix and its relationship with the 3D cores. Here, we identify each multi-index $(i_1, ..., i_j)$ uniquely with an index in $\left[1 ... \prod_{k=1}^{j} I_k\right]$. Each row of $\boldsymbol{A}^L_{\leq j}$ is a product of matrix slices from the core tensors. We call the set of cores $\mathcal{A}_1, ..., \mathcal{A}_j$ related in his manner a *core chain*, and we refer to $\boldsymbol{A}^L_{\leq j}$ as the matricization of the chain. Figure 5.1 illustrates the relationship between the two structures. Core chains arise naturally in computations involving the tensor train decomposition [Ose11] (also called the matrix-product state by the physics community [Per+07]). Figure 5.2 provides two complementary illustrations of a three-dimensional tensor train, which we discuss further in Section 5.2.

How can we efficiently compute a column subspace embedding of a core chain matricization? Sketches of $\boldsymbol{A}^L_{\leq j}$ can be used to reduce the rank of an existing tensor train decomposition



(a) (b)

Figure 5.2: Two illustrations of the tensor train decomposition. Subfigure (a) shows each 3D tensor train core as a stack of vectors or matrices. To evaluate an entry of the represented tensor, the highlighted items from each stack are gathered and multiplied together. Subfigure (b) shows the same structure using tensor network notation.

[Al +23]. In addition, they can accelerate an alternating least-squares algorithm analogous to the one studied in Chapters 3 and 4 [Mal22]. We focus on the special case where the *left-matricization* of each input core is orthonormal. That is, each matrix

$$\boldsymbol{A}_k^L = \mathrm{mat}(\mathcal{A}_k, 3)$$

has orthonormal columns (see Section 3.6.7 for the definition of the matricization operator). By assuming that the input arrives in orthonormal form, the sketching algorithm we devise cannot speed up tensor train rounding, but proves useful for other applications. Our main contribution is the following theorem:

Theorem 5.1.1 (Row-norm-squared sampling for orthonormal core chains)**.** *Let $\boldsymbol{A}_{\leq j}$ be the left-matricization of a core chain, specified row-wise by Equation* (5.1)*. Suppose each core $\mathcal{A}_1, ..., \mathcal{A}_j$ is orthonormal. Then there exists a data structure to randomly sample rows from $\boldsymbol{A}_{\leq j}$ according to the distribution of its squared row norms with the following properties:*

1. *The data structure has construction time $O\left(\sum_{n=1}^{j} I_n R_{n-1} R_n^2\right)$. When $R = R_1 = ... = R_j$ and $I = I_1 = ... = I_j$, the runtime is $O(jIR^3)$. The space overhead of the data structure is linear in the sizes of the input cores.*
2. *The data structure produces a single row sample from $\boldsymbol{A}_{\leq j}$ according to the distribution of its squared row norms in time $O\left(\sum_{k=1}^{j} \log\left(I_k R_{k-1}/R_k\right) R_k^2\right)$. When all ranks R_k and physical dimensions I_k are equal, this complexity is $O(jR^2 \log I)$.*

For simplicity, take $R_1 = ... = R_j = R$ and $I_1 = ... = I_j = I$. Then the complexity to draw a row from the core chain is only a factor $\log I$ worse than the time to materialize that row using Equation (5.1).

5.2 Context and Related Work

Tensor trains are attractive because they can represent tensors of extremely high dimension with only a modest parameter count. An N-dimensional tensor with side-length I decomposes with only $O(NIR^2)$ parameters, where R is the rank of the decomposition. Such a decomposition exists for any provided tensor, but the rank R may be exponential in N in the worst case [Ose11]. Contrast this with PARAFAC and Tucker decompositions [BK25]: the latter cannot avoid exponential parameter growth in the core tensor, while the former is empirically less expressive [Nov+15]. As a result, tensor trains have found particular success in high-dimensional function approximation [Ose11; OT10]. They have also successfully replaced dense matrices in linear layers of deep neural networks [Nov+15], providing steep discounts in parameter count and a regularizing effect that sometimes leads to accuracy *gain*.

We can view a tensor train as a compact representation of a vector of length I^N. With appropriate modifications, it can also represent an $I^N \times I^N$ matrix that can act efficiently on a tensor train vector. This property makes tensor trains effective for dynamic low rank approximation, where a system state represented by a TT-vector evolves due to the action of a TT-matrix [GKT13; YL24]. Likewise, TT-vectors are used in Krylov subspace methods where the operator is given either as a Kronecker product or a TT-matrix [Al +23].

5.2.1 Orthonormalizing a Core Chain

If each core matricization $\boldsymbol{A}_k^L$ is orthonormal for $1 \leq k \leq j$, it is a well-known fact that $\boldsymbol{A}_{\leq j}^L$ is also orthonormal. We formalize the statement below.

Proposition 5.2.1 (Lemma 3.1 from Oseledets [Ose11], Adapted)**.** *Suppose* $\boldsymbol{A}_k^{L\top} \boldsymbol{A}_k^L = \boldsymbol{I}$ *for* $1 \leq k \leq j$*. Then* $\boldsymbol{A}_{\leq j}^{L\top} \boldsymbol{A}_{\leq j}^L = \boldsymbol{I}$*.*

Proposition 5.2.1 allows us to orthonormalize a core chain matricization by orthonormalizing each core in isolation. We refer the reader to Oseledets [Ose11] for a proof and note that subsequent results in this chapter rely on similar proof techniques. The orthonormalization property is required for the applications we describe next.

5.2.2 Alternating Core Optimization

We revisit the tensor fitting task from Chapters 3 and 4: given tensor $\mathcal{T}$, the goal is to fit a tensor train decomposition $[\mathcal{A}_1, ..., \mathcal{A}_N]$. The core ranks are either selected in advance or

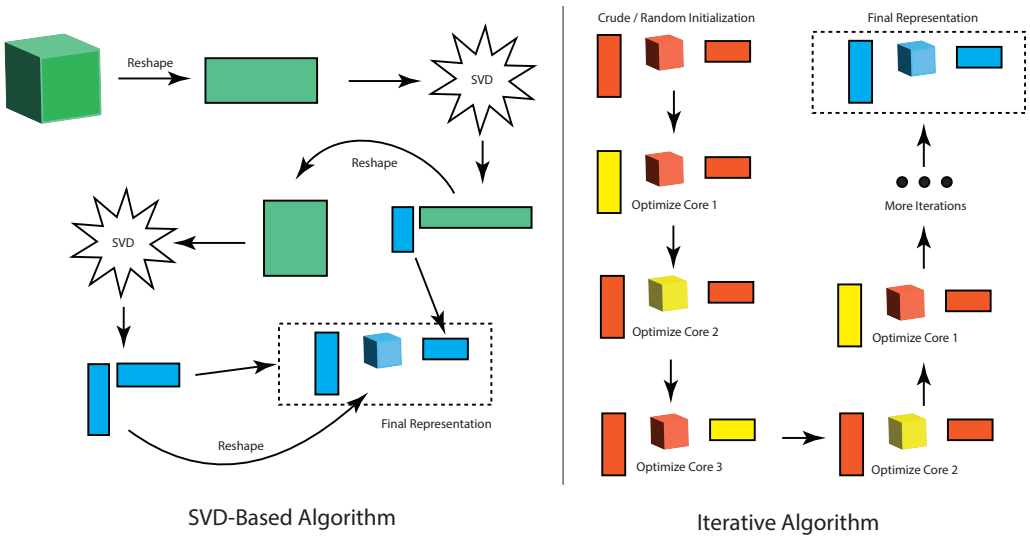


Figure 5.3: Sequential vs. iterative algorithms for tensor train decomposition. Left: the SVD-based algorithm begins with the full tensor flattened into a matrix. It alternates between reshaping intermediate objects and computing a truncated SVD, producing a single core of the final decomposition at each step. Right: the iterative algorithm begins with a crude initialization for the tensor train decomposition and optimizes one core at a time while keeping the remainder fixed.

determined adaptively as the algorithm progresses. Figure 5.3 illustrates two approaches to this problem and highlights a contrast to Candecomp / PARAFAC decomposition: namely, the existence of an SVD-based algorithm that guarantees successful decomposition. When the tensor is specified explicitly in memory, Figure 5.3 illustrates a process of iteratively flattening the tensor, computing a singular value decomposition, reshaping one of the outputs, and repeating these steps [Ose11].

Unfortunately, the runtime of singular value decomposition on large tensors may be prohibitive. Adding to the problem, the target tensor $\mathcal{T}$ may be too large to represent in memory; tensor train decompositions are routinely computed on inputs with 100+ dimensions [OT10]. Figure 5.3 (right) illustrates an alternate approach: cores are initialized randomly and optimized iteratively until convergence. This alternating scheme is related to both alternating least squares [KB09; BK25] and the Density Matrix Renormalization Group (DMRG) algorithms proposed by White [Whi92]. In the latter approach, a tensor train represents a high-dimensional quantum state, and alternating optimization produces the minimum energy eigenstate of a Hamiltonian (which is also specified as a tensor train matrix, or "matrix product operator").

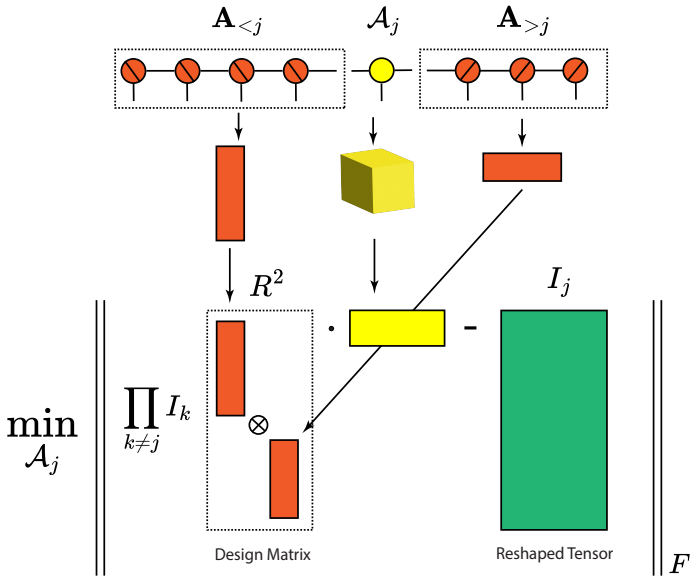


Figure 5.4: Illustrated linear least squares problem from Proposition 5.2.2.

The core update rules proposed by Oseledets and Tyrtyshnikov [OT10] rely on variants of matrix skeleton decomposition to optimize each core, rather than linear least-squares. Consequently, these methods are highly efficient for smooth functional tensors with sufficient structure, but struggle on noisy (high-error) decomposition tasks. We concentrate on alternating least squares updates:

Proposition 5.2.2 (Tensor Train ALS Update Rule)**.** *Given tensor train* $[\mathcal{A}_1, ..., \mathcal{A}_N]$ *and target tensor* $\mathcal{T}$*,*

$$\min_{\mathcal{A}_j} \|[\mathcal{A}_1, ..., \mathcal{A}_N] - \mathcal{T}\|_F = \min_{\boldsymbol{A}_j = \mathrm{mat}(\mathcal{A}_j, 2)^\top} \left\| \left(\boldsymbol{A}_{<j} \otimes \boldsymbol{A}_{>j}^\top\right) \cdot \boldsymbol{A}_j - \mathrm{mat}(\mathcal{T}, j)^\top \right\|_F,$$

where $\boldsymbol{A}_{<j}$ *is the matricization of the core chain comprising* $\mathcal{A}_1, ..., \mathcal{A}_{j-1}$ *and* $\boldsymbol{A}_{>j}$ *is the matricization of the core chain composed of* $\mathcal{A}_{j+1}, ..., \mathcal{A}_N$*.*

Figure 5.4 illustrates the key linear least squares problem, where the design matrix is a Kronecker product of a pair of core chains. We omit the proof of Proposition 5.2.2, which follows from the linearity of tensor contraction. From Chapter 3, it follows that a fast sampler for the row leverage scores of $\boldsymbol{A}_{<j}$ produces an efficient randomized algorithm for tensor train ALS updates.

Note a critical detail: we can cheaply compute the QR decomposition of both $\boldsymbol{A}_{<j}$ and $\boldsymbol{A}_{>j}^\top$ via Proposition 5.2.1. The $\boldsymbol{R}$-factors are irrelevant for the optimization problem in

Proposition 5.2.2, so we take $\boldsymbol{A}_{<j}$ and $\boldsymbol{A}_{>j}^{\top}$ to be orthonormal without loss of generality. This canonical form leads to an (ideally) conditioned linear least squares problem, as well as structure that we exploit for leverage score sampling. While the design matrix is orthonormal, the runtime bottleneck to solve each linear least squares problem is computation of $(\boldsymbol{A}_{<j} \otimes \boldsymbol{A}_{>j}^{\top})^{\top} \cdot \mathrm{mat}(\mathcal{T}, j)^{\top}$. This cost remains unchanged by the canonical form, and we will reduce it through randomization.

5.2.3 Tensor Train Rounding

Before turning to leverage score sampling, we discuss the limitations of our approach for a second important tensor train primitive. Multiple applications, including dynamic low-rank approximation [GKT13; YL24] and tensor train Krylov methods [Al +23], maintain a tensor train with ranks that increase as the algorithm progresses. The rank inflation is a typically a consequence of adding multiple vectors (represented as tensor trains) or applying a matrix product operator to an evolving matrix product state. Oseledets [Ose11] describes a rank reduction procedure to combat this issue and preserve accuracy of the decomposition. The tensor train is first converted to left-orthonormal canonical form with only the rightmost core as a general tensor. Successive singular value decompositions are then executed from right to left for each core, with the lowest singular values dropped to reduce the core ranks.

The bottleneck in *tensor train rounding* is converting the representation with inflated ranks into orthonormal canonical form. Al Daas et al. [Al +23] propose a randomized sketch to accelerate orthonormalization, while Ma and Solomonik [MS22] prove matching lower bounds on the runtime of sketching-based approaches. By contrast, the leverage score sketch that we propose requires that the core chain arrive in orthonormal form. While the cost to orthonormalize each core is negligible in ALS, it is prohibitively large for tensor train rounding compared to the remaining computation. In this case, our methods do not provide a benefit.

5.3 An Efficient Orthonormal Core Chain Sampler

We now sketch the proof of Theorem 5.1.1 by exhibiting a data structure that samples a row from the matrix left chain matricization $\boldsymbol{A}_{\leq j}$ with probability proportional to its squared row norm. With straightforward modifications, such a data structure also draws samples from a right chain matricization $\boldsymbol{A}_{\geq j}$. Our proof closely mirrors that of the corresponding claim in Chapter 3, but key modifications are required to adapt the procedure to a tensor core chain.

Let random variables $\hat{s}_1, ..., \hat{s}_j$ be random variables **defined** so that the multi-index $(\hat{s}_1, ..., \hat{s}_j)$ is drawn with probability proportional to the squared norm of the corresponding row from

$\boldsymbol{A}_{\leq j}$. Each variable $\hat{s}_k$ takes on values in the set $[I_k]$, and by definition we have

$$p(\hat{s}_1 = s_1 \wedge ... \wedge \hat{s}_j = s_j) := \frac{1}{R_j} \left(\boldsymbol{A}_{\leq j} \left[(s_1, ..., s_j), : \right] \cdot \boldsymbol{A}_{\leq j} \left[(s_1, ..., s_j), : \right]^\top \right). \tag{5.2}$$

In the equation above, we take $\boldsymbol{A}_{\leq j} \left[(s_1, ..., s_j), : \right]$ as a row vector and its transpose as a column vector. Our sampling procedure will draw a slice from each core starting from $\mathcal{A}_j$ and ending with $\mathcal{A}_1$, corresponding to a single row from $\boldsymbol{A}_{\leq j}$. By starting the sampling procedure at the j-th core, we can exploit the left-orthonormality property to derive the conditional distribution on variable $\hat{s}_k$.

Lemma 5.3.1 (Conditional distribution for $\hat{s}_k$)**.** *Consider the event $\hat{s}_j = s_j, ..., \hat{s}_{k+1} = s_{k+1}$, which we abbreviate as $\hat{s}_{>k} = s_{>k}$. Then*

$$p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) = \frac{1}{\|\boldsymbol{H}_{>k}\|_F^2} \mathrm{Tr} \left[\boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k \left[:, s_k, : \right]^\top \cdot \mathcal{A}_k \left[:, s_k, : \right] \cdot \boldsymbol{H}_{>k} \right]$$

where $\boldsymbol{H}_{>k} := \mathcal{A}_{k+1} \left[:, s_{k+1}, : \right] \cdot ... \cdot \mathcal{A}_j \left[:, s_j, : \right]$. When $k = j$, we take $\hat{s}_{>k} = s_{>k}$ as a trivial event with probability 1 and $\boldsymbol{H}_{>k} = \boldsymbol{I} \in \mathbb{R}^{R_j \times R_j}$.

The proof appears in Section 5.6.1. Here, $\boldsymbol{H}_{>k}$ functions as a "history matrix" as the product of slices already selected by our sampler. Unfortunately, this distribution is expensive to sample. The cost to even update $\boldsymbol{H}_{>k}$ through matrix multiplication as indices are selected is $O(R^3)$ (assuming $R_1 = ... = R_j = R$), which cannot satisfy the complexity requirement in Theorem 5.1.1.

Instead, we make the following observation: let $\boldsymbol{q}$ be the distribution of squared row norms on the rows of $\boldsymbol{A}_{\leq j}$. Then

$$\boldsymbol{q} := \frac{1}{R_j} \left(\boldsymbol{A}_{\leq j} \left[:, 1 \right]^2 + ... + \boldsymbol{A}_{\leq j} \left[:, R_j \right]^2 \right),$$

where $\boldsymbol{A}_{\leq j} \left[:, ... \right]^2$ denotes the componentwise square of each column vector. In other words, $\boldsymbol{q}$ takes the form of a *mixture distribution*. Since each column of $\boldsymbol{A}_{\leq j}$ has the same norm, it suffices to select a single column uniformly at random and restrict the sampling procedure to that column. More concretely, define the random variable $\hat{r}$ as uniform on $[1, .., R_j]$ and random variables $\hat{t}_1, ..., \hat{t}_j$ by the conditional distribution

$$p(\hat{t}_k = t_k \mid \hat{t}_{k+1} = t_{k+1} \wedge ... \wedge \hat{t}_j = t_j \wedge \hat{r} = r) = \frac{1}{\|\boldsymbol{h}_{>k}\|^2} \left\| \mathcal{A}_k \left[:, t_k, : \right] \cdot \boldsymbol{h}_{>k} \right\|^2 \tag{5.3}$$

where $\boldsymbol{h}_{>k} = \mathcal{A}_{k+1} \left[:, t_{k+1}, : \right] \cdot ... \cdot \mathcal{A}_j \left[:, t_j, : \right] \cdot \boldsymbol{e}_r$ and $\boldsymbol{e}_r$ is the r-th standard basis vector in $\mathbb{R}^{R_j}$. Then we have the following lemma:

Lemma 5.3.2 (Conditional Distribution of $\hat{t}_k$)**.** *Fix $s_j = t_j, s_{j-1} = t_{j-1}, ..., s_k = t_k$. After marginalizing over $\hat{r}$, the conditional distribution on $\hat{t}_k$ satisfies*

$$p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k}) = p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}).$$

As a consequence of Lemma 5.3.2, the joint random variable $(\hat{t}_1, ..., \hat{t}_j)$ follows the desired squared row-norm distribution on the rows of $\boldsymbol{A}_{\leq j}$ after marginalizing over $\hat{r}$. The proof appears in Section 5.6.2. Notice that the "history matrix" $\boldsymbol{H}_{>k}$ has been replaced by a vector $\boldsymbol{h}_{>k}$. This modification allows us to control sampling complexity, as the history vector can be updated through matrix-vector (rather than matrix-matrix) multiplication.

Our final task is to draw each sample from the distribution in Equation (5.3) in time sublinear in the dimension I_k (after appropriate preprocessing). Letting $\boldsymbol{A}_k^L$ be the left matricization of core $\mathcal{A}_k$, we have

$$\begin{aligned} &p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k} \wedge \hat{r} = r) \\ &= \frac{1}{\|\boldsymbol{h}_{>k}\|^2} \left\| \boldsymbol{A}_k^L \left[t_k R_{k-1} : (t_k + 1) R_{k-1}, : \right] \cdot \boldsymbol{h}_{>k} \right\|_2^2 \\ &= \frac{1}{\|\boldsymbol{h}_{>k}\|^2} \sum_{i=0}^{R_{k-1}} \left(\boldsymbol{A}_k^L \left[t_k R_{k-1} + i, : \right] \cdot \boldsymbol{h}_{>k} \right)^2 \end{aligned} \tag{5.4}$$

The probability of selecting the slice s_k is proportional to the sum of R_{k-1} consecutive entries from the probability vector $(\boldsymbol{A}_k^L \cdot \boldsymbol{h}_{>k})^2$. As a result, we can sample $\hat{t}_k$ by first sampling an index in the range $[1..I_k R_{k-1}]$ given by $(\boldsymbol{A}_k^L \cdot \boldsymbol{h}_{>k})^2$, then performing integer division by R_{k-1} to obtain the corresponding slice index t_k. Note that we already have a powerful tool to sample from the weight vector $(\boldsymbol{A}_k^L \cdot \boldsymbol{h}_{>k})^2$ given arbitrary $\boldsymbol{h}_{>k}$: Lemma 3.3.2. The following corollary formalizes the claim.

Corollary 5.3.3 (Narrowing of Lemma 3.3.2)**.** *Given a matrix $\boldsymbol{A} \in \mathbb{R}^{M \times R}$, there exists a data structure with construction time $O(MR^2)$ and space usage $O(MR)$. Given any vector $\boldsymbol{h} \in \mathbb{R}^R$, it can draw a single sample from the un-normalized distribution of weights $(\boldsymbol{A} \cdot \boldsymbol{h})^2$ in time $O(R^2 \log(M/R))$.*

Proof. Construct the row sampling data structure in Lemma 3.3.2 with $\boldsymbol{U} = \boldsymbol{A}_k^L$ with $\boldsymbol{Y} = [1]$, a matrix of all ones, and set $F = R$ and $C = \|\boldsymbol{h}_{>k}\|^2$. Then

$$\boldsymbol{q}_{\boldsymbol{h},\boldsymbol{U},\boldsymbol{Y}}[s] = C^{-1} \boldsymbol{U}[s, :] \left(\boldsymbol{h}\boldsymbol{h}^\top \right) \boldsymbol{U}[s, :]^\top = C^{-1} (\boldsymbol{U}[s, :] \cdot \boldsymbol{h})^2$$

This is the target distribution, and the runtime to draw each sample is $O(R^2 \log(M/R) + R^2) = O(R^2 \log(M/R))$. The choice $F = R$ also induces space usage $O(MR)$, linear in the size of the input. □

Corollary 5.3.3 enables us to efficiently draw samples according to the distribution in Equation 5.4, and therefore gives us a procedure to sample from the entire core chain. Constructing the data structure above for each matrix $\boldsymbol{A}_k^L$, $1 \leq k \leq j$, costs $O(IR_{k-1}R_k^2)$ with a linear space overhead in the input core sizes. Drawing a sample from the k-th data structure requires time $O(R_k^2 \log(I_k R_{k-1}/R_k))$. Summing up this runtime over $1 \leq k \leq j$ gives the stated complexity in Theorem 5.1.1.

Algorithm 15 ConstructChainSampler($\mathcal{A}_1, ..., \mathcal{A}_N$)

for $k = 1..N$ **do**
  $Z_k :=$ BuildSampler($\boldsymbol{A}_k^L$)

Algorithm 16 ChainSampleLeft(J, j)

for $d = 1..J$ **do**
  $\hat{r} :=$ Uniform-sample($[1...R_j]$)
  $h := \boldsymbol{e}_{\hat{r}}$
  for $k = j...1$ **do**
    $\hat{t}_k :=$ RowSample($Z_k, \boldsymbol{h}$)$//R_{k-1}$
    $\boldsymbol{h} = \boldsymbol{h} \cdot \mathcal{A}_k \left[:, \hat{t}_k, :\right]$
  $t_d = (\hat{t}_k)_{k \leq j}$
return $t_1, ..., t_J$

Algorithms 15 and 16 summarize the procedures to efficiently draw J samples from a left-orthogonal core chain. The construction procedure builds a set of data structures of the form given by Lemma 3.3.2 on the left-matricization of each tensor core. For each of J rows to draw, the sampling algorithm selects a column $\hat{t}$ uniformly at random from the left matricization $\boldsymbol{A}_{<j}^L$. It then initializes the history vector h and successively samples indices $\hat{s}_{j-1}, ..., \hat{s}_1$ according to the conditional distribution, updating the history vector at each step. Section 5.6.3 provides a rigorous proof of the correctness of the procedure sketched in this section.

While the proof sketched above shares similarities with its analogue in Section 3.3, key adaptations are required to sample from a tensor train core chain. The factors of a Khatri-Rao product can be sampled in any order, since the Khatri-Rao product of several matrices is commutative up to a permutation of its rows. Our sampling procedure **requires** us to sample from core $\mathcal{A}_j$ down to $\mathcal{A}_1$, since Lemma 5.3.1 exploits the left-orthogonality of the each core in its derivation. Starting the sampling procedure at $\mathcal{A}_j$ leads to a "history matrix" to keep track of prior draws instead of the vector that would arise starting from core $\mathcal{A}_1$. Here, our second innovation of uniform random column selection brings down sample complexity.

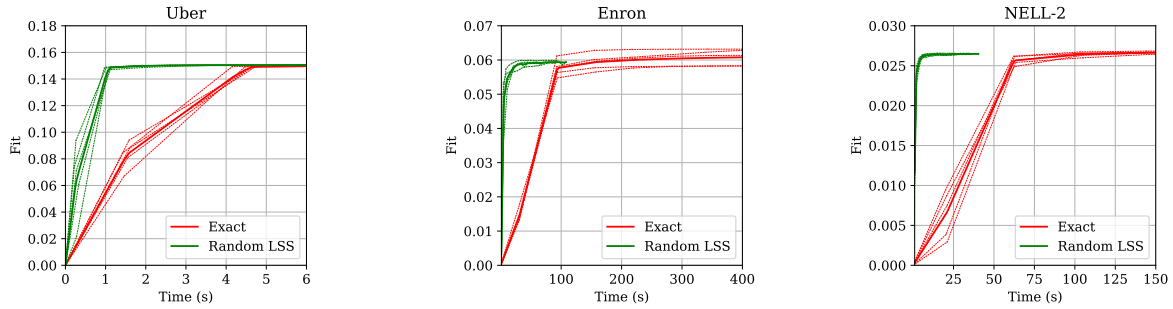


Figure 5.5: Fit as a function of time for three FROSTT tensors, $R = 6$, $J = 2^{16}$ for rTT-ALS. Thick lines are averages of 5 fit-time traces, shown by thin dotted lines.

	Uber			Enron			NELL-2		
R	rTT-ALS	TT-ALS	Speedup	rTT-ALS	TT-ALS	Speedup	rTT-ALS	TT-ALS	Speedup
4	0.1332	0.1334	4.0x	0.0498	0.0507	17.8x	0.0213	0.0214	26.0x
6	0.1505	0.1510	3.5x	0.0594	0.0611	12.4x	0.0265	0.0269	22.8x
8	0.1646	0.1654	3.0x	0.0669	0.0711	10.5x	0.0311	0.0317	22.2x
10	0.1747	0.1760	2.4x	0.0728	0.0771	8.5x	0.0350	0.0359	20.5x
12	0.1828	0.1846	1.5x	0.0810	0.0856	7.4x	0.0382	0.0394	15.8x

Table 5.1: Average fits and speedup, $J = 2^{16}$ for ALS algorithms, 40 iterations. The speedup is the average per-iteration runtime for a single exact ALS sweep divided by the average time for a single randomized sweep.

5.4 Experiments

Experiments were conducted on CPU nodes of NERSC Perlmutter. We reused the core sampling infrastructure from Chapter 3, and our code is available online at `https://github.com/vbharadwaj-bk/ortho_tt_subspace_embedding`. To test our sampling procedure, we focus on sparse tensor decomposition via alternating least squares.

5.4.1 Approximate Sparse Tensor Train Decomposition

We apply randomized TT-ALS to three large sparse tensors from FROSTT [Smi+17]. Table 5.1 gives the fits achieved by our method to decompose these tensors. The largest of these tensors, NELL-2, has around 77 million nonzero entries with mode sizes in the tens of thousands. Fits for sparse tensor decomposition are typically low, but the factors of the resulting decomposition have successfully been mined for patterns [LK22]. For these experiments, we chose all decomposition ranks equal with $R_1 = ... = R_N = R$ and tested over a range of values for R.

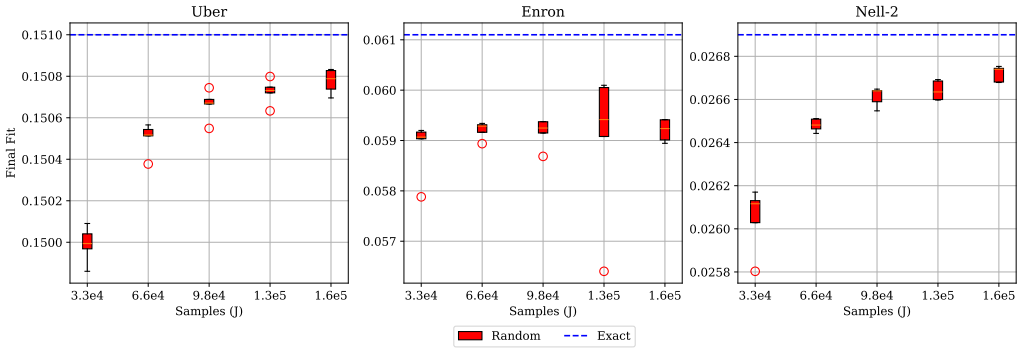


Figure 5.6: Final fit of sparse tensor decomposition for varying sample counts. Each box plot reports statistics for 5 trials. The blue dashed lines show the fit for non-randomized ALS.

The fits produced by rTT-ALS match those produced by the non-randomized ALS method up to variation in the third significant figure for Uber and NELL-2, with slightly higher errors on the Enron tensor. We kept the sample count for our randomized algorithms fixed at $J = 2^{16}$ throughout this experiment. As a result, the gap between the fit of the randomized and exact methods grows as the target rank increases, which our theory predicts.

Table 5.1 also reports the average speedup per ALS sweep of rTT-ALS over the exact algorithm. On the NELL-2 sparse tensor with target rank 12, the non-randomized ALS algorithm requires an average of 29.4 seconds per ALS sweep, while rTT-ALS requires only 1.87 seconds. Figure 5.5 shows that our method makes faster progress than its non-randomized counterpart across all three tensors. Because we could not find a well-documented, high-performance library for sparse tensor train decomposition, we wrote a fast multithreaded implementation in C++, which serves as the baseline method in these figures and tables.

Figure 5.6 shows the impact of varying the sample count on the final fit. We find modest increases in accuracy for both Uber and NELL-2 as the sample count increases by a factor of 5 (starting from $J = 2^{15}$). Increasing J has a smaller impact for the Enron tensor, which is more difficult to decompose with i.i.d. random factor initialization [LK22].

5.5 Conclusions and Future Work

We have established, both empirically and theoretically, that our sampling algorithm correctly draws samples according to the leverage scores of orthonormal core chains. The method accelerates sparse tensor train decomposition; furthermore. our results testify to the generalization power of Theorem 3.3.2, which was originally designed to sample from the Khatri-Rao product.

5.6 Complete proofs

5.6.1 Proof of Lemma 5.3.1

In the proof below, we assume that the conditioning event $\hat{s}_{>k} = s_{>k}$ occurs with nonzero probability and abbreviate $C = p(\hat{s}_{>k} = s_{>k})$. We write

$$\begin{aligned}
&p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) \\
&\quad = p(\hat{s}_k = s_k \wedge \hat{s}_{>k} = s_{>k}) p(\hat{s}_{>k} = s_{>k})^{-1} \\
&\quad = C^{-1} \sum_{s_1,\ldots,s_{k-1}} p(\hat{s}_1 = s_1 \wedge \ldots \wedge \hat{s}_j = s_j) \\
&\quad = C^{-1} \sum_{s_1,\ldots,s_{k-1}} \frac{1}{R_j} \left(\boldsymbol{A}_{\leq j}\left[(s_1,\ldots,s_j),:\right] \cdot \boldsymbol{A}_{\leq j}\left[(s_1,\ldots,s_j),:\right]^\top \right) \\
&\quad = C^{-1} \sum_{s_1,\ldots,s_{k-1}} \frac{1}{R_j} \mathrm{Tr}\left[\boldsymbol{A}_{\leq j}\left[(s_1,\ldots,s_j),:\right]^\top \cdot \boldsymbol{A}_{\leq j}\left[(s_1,\ldots,s_j),:\right] \right] \\
&\quad = C^{-1} \frac{1}{R_j} \sum_{s_1,\ldots,s_{k-1}} \mathrm{Tr}\left[\mathcal{A}_j\left[:,s_j,:\right]^\top \cdot \ldots \cdot \mathcal{A}_1\left[:,s_1,:\right]^\top \cdot \mathcal{A}_1\left[:,s_1,:\right] \cdot \ldots \cdot \mathcal{A}_j\left[:,s_j,:\right] \right] \\
&\quad = C^{-1} \frac{1}{R_j} \sum_{s_2,\ldots,s_{k-1}} \mathrm{Tr}\left[\mathcal{A}_j\left[:,s_j,:\right]^\top \cdot \ldots \cdot \left(\sum_{s_1} \mathcal{A}_1\left[:,s_1,:\right]^\top \cdot \mathcal{A}_1\left[:,s_1,:\right] \right) \cdot \ldots \cdot \mathcal{A}_j\left[:,s_j,:\right] \right] \\
&\quad = C^{-1} \frac{1}{R_j} \sum_{s_2,\ldots,s_{k-1}} \mathrm{Tr}\left[\mathcal{A}_j\left[:,s_j,:\right]^\top \cdot \ldots \cdot \mathcal{A}_2\left[:,s_2,:\right]^\top \cdot \boldsymbol{I} \cdot \mathcal{A}_2\left[:,s_2,:\right] \cdot \ldots \cdot \mathcal{A}_j\left[:,s_j,:\right] \right].
\end{aligned} \tag{5.5}$$

In the expressions above, the summation over each variable s_t, $1 \leq t \leq k$, is taken over the range $1 \ldots I_t$. The first step is Bayes' rule, and the second step follows by marginalizing over random variables $\hat{s}_1, \ldots, \hat{s}_{k-1}$. The third step follows from the Equation 5.2. The fourth step rewrites an inner product of two vectors as the trace of their outer product. The fifth step follows from the definition of $\boldsymbol{A}_{\leq j}$. The sixth step follows from the linearity of the trace by moving the summation over s_1 into the product expression. The last step follows from the definition of the left-orthonormality property on $\mathcal{A}_1$; that is, $\sum_{s_1} \mathcal{A}_1\left[:,s_1,:\right]^\top \cdot \mathcal{A}_1\left[:,s_1,:\right] = \boldsymbol{A}_1^{L\top} \boldsymbol{A}_1^L = \boldsymbol{I}$. By successively moving summation operators into the product expression to repeat the last step (exploiting the left-orthonormality of each core in the process), we find

$$\begin{aligned}
&p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) \\
&\quad = \frac{1}{CR_j} \mathrm{Tr}\left[\mathcal{A}_j\left[:,s_j,:\right]^\top \cdot \ldots \cdot \mathcal{A}_k\left[:,s_k,:\right]^\top \cdot \mathcal{A}_k\left[:,s_k,:\right] \cdot \ldots \cdot \mathcal{A}_j\left[:,s_j,:\right] \right] \\
&\quad = \frac{1}{CR_j} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k\left[:,s_k,:\right]^\top \cdot \mathcal{A}_k\left[:,s_k,:\right] \cdot \boldsymbol{H}_{>k} \right],
\end{aligned} \tag{5.6}$$

where the last line follows from the definition of $\boldsymbol{H}_{>k}$. Our final task is to compute $C = p(\hat{s}_{>k} = s_{>k})$:

$$\begin{aligned} 1 &= \sum_{s_k} p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) \\ &= \frac{1}{CR_j} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \sum_{s_k} \left(\mathcal{A}_k\left[:, s_k, :\right]^\top \cdot \mathcal{A}_k\left[:, s_k, :\right] \right) \cdot \boldsymbol{H}_{>k} \right] \\ &= \frac{1}{CR_j} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \boldsymbol{I} \cdot \boldsymbol{H}_{>k} \right], \\ &= \frac{\|\boldsymbol{H}_{>k}\|_F^2}{CR_j}, \end{aligned} \tag{5.7}$$

where the third step again exploits left-orthonormality of $\mathcal{A}_k$. Therefore $C = \|\boldsymbol{H}_{>k}\|_F^2 / R_j$, and substituting back yields

$$\begin{aligned} &p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) \\ &\quad = \frac{1}{CR_j} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k\left[:, s_k, :\right]^\top \cdot \mathcal{A}_k\left[:, s_k, :\right] \cdot \boldsymbol{H}_{>k} \right] \\ &\quad = \frac{1}{\|\boldsymbol{H}_{>k}\|_F^2} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k\left[:, s_k, :\right]^\top \cdot \mathcal{A}_k\left[:, s_k, :\right] \cdot \boldsymbol{H}_{>k} \right]. \end{aligned} \tag{5.8}$$

□

5.6.2 Proof of Lemma 5.3.2

We use induction to simultaneously prove the following two statements from $k = j$ down to $k = 1$:

$$\begin{aligned} p(\hat{r} = r \mid \hat{t}_{>k} = t_{>k}) &= \frac{\|\boldsymbol{h}_{>k}\|^2}{\|\boldsymbol{H}_{>k}\|_F^2} \\ p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) &= p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k}). \end{aligned} \tag{5.9}$$

The induction base case is $k = j$; recall here that we take $\hat{s}_{>j} = s_{>j}$ and $\hat{t}_{>j} = t_{>j}$ as trivial events that occur with probability 1 and do not change the probability of any other event. Likewise when $k = j$, we take $\boldsymbol{H}_{>k} = \boldsymbol{I} \in \mathbb{R}^{R_j \times R_j}$. Then we have

$$\begin{aligned} p(\hat{r} = r \mid \hat{t}_{>j} = t_{>j}) &= p(\hat{r} = r) \\ &= \frac{1}{R_j} \\ &= \frac{\|\boldsymbol{I} \cdot \boldsymbol{e}_r\|^2}{\|\boldsymbol{I}\|_F^2} \\ &= \frac{\|\boldsymbol{h}_{>j}\|^2}{\|\boldsymbol{H}_{>j}\|_F^2}. \end{aligned} \tag{5.10}$$

The first step removes conditioning on the trivial event. The second step uses the fact that $\hat{r}$ is uniform random on $[R_j]$. The third step rewrites the numerator and denominator, and the fourth step uses the definitions of $\boldsymbol{h}_{>j}$ and $\boldsymbol{H}_{>j}$.

Likewise for the second statement,

$$\begin{aligned}
p(\hat{s}_j = s_j \mid \hat{s}_{>j} = s_{>j}) &= \frac{1}{\|\boldsymbol{H}_{>j}\|_F^2} \mathrm{Tr}\left[\boldsymbol{H}_{>j}^\top \cdot \mathcal{A}_j[:, s_j, :]^\top \cdot \mathcal{A}_j[:, s_j, :] \cdot \boldsymbol{H}_{>j}\right] \\
&= \frac{1}{R_j} \mathrm{Tr}\left[\mathcal{A}_j[:, s_j, :]^\top \cdot \mathcal{A}_j[:, s_j, :]\right] \\
&= \frac{1}{R_j} \sum_{r=1}^{R_j} \left(\boldsymbol{e}_r^\top \cdot \mathcal{A}_j[:, s_j, :]^\top \cdot \mathcal{A}_j[:, s_j, :] \cdot \boldsymbol{e}_r\right) \\
&= \frac{1}{R_j} \sum_{r=1}^{R_j} \|\mathcal{A}_j[:, s_j, :] \cdot \boldsymbol{e}_r\|^2 \\
&= \sum_{r=1}^{R_j} p(\hat{t}_j = t_j \mid \hat{t}_{>j} = t_{>j} \wedge \hat{r} = r) p(\hat{r} = r) \\
&= p(\hat{t}_j = t_j \mid \hat{t}_{>j} = t_{>j}).
\end{aligned} \tag{5.11}$$

The first step uses Lemma 5.3.1. The second step substitutes $\boldsymbol{H}_{>j} = \boldsymbol{I}$. The third and fourth steps use properties of the matrix trace. The fifth step uses uniformity of $\hat{r}$ and substitutes the conditional distribution for $\hat{t}_j$ defined in Equation (5.3). The final step uses the law of total probability to complete the proof of the base case.

To prove the inductive step, assume both statements hold for $k+1$. Then for k, we have

$$\begin{aligned}
&p(\hat{r} = r \mid \hat{t}_{>k} = t_{>k}) \\
&= p(\hat{r} = r \mid \hat{t}_{k+1} = t_{k+1} \wedge \hat{t}_{>k+1} = t_{>k+1}) \\
&= p(\hat{t}_{k+1} = t_{k+1} \mid \hat{r} = r \wedge \hat{t}_{>k+1} = t_{>k+1}) \cdot \frac{p(\hat{r} = r \mid \hat{t}_{>k+1} = t_{>k+1})}{p(\hat{t}_{k+1} = t_{k+1} \mid \hat{t}_{>k+1} = t_{>k+1})} \\
&= \frac{\|\mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \boldsymbol{h}_{>k+1}\|^2}{\|\boldsymbol{h}_{>k+1}\|^2} \cdot \frac{\|\boldsymbol{h}_{>k+1}\|^2}{\|\boldsymbol{H}_{>k+1}\|_F^2} \cdot \frac{\|\boldsymbol{H}_{>k+1}\|_F^2}{\mathrm{Tr}\left[\boldsymbol{H}_{>k+1} \cdot \mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \boldsymbol{H}_{>k+1}\right]} \\
&= \frac{\|\mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \boldsymbol{h}_{>k+1}\|^2}{\mathrm{Tr}\left[\boldsymbol{H}_{>k+1} \cdot \mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \mathcal{A}_{k+1}[:, s_{k+1}, :] \cdot \boldsymbol{H}_{>k+1}\right]} \\
&= \frac{\|\boldsymbol{h}_{>k}\|^2}{\|\boldsymbol{H}_{>k}\|_F^2}.
\end{aligned} \tag{5.12}$$

The first step splits off the conditioning event $\hat{t}_{k+1} = t_{k+1}$. The second step uses Bayes' rule with the events $\hat{t}_{k+1} = t_{k+1}$ and $\hat{r} = r$, both conditioned on $\hat{t}_{>k+1} = t_{>k+1}$. At this point, the

three probabilities in the equation (left quantity, numerator of fraction, and denominator of fraction) are known from Equation (5.3), statement 1 of the induction hypothesis, and statement 2 of the induction hypothesis combined with Lemma 5.3.1, respectively. The third step makes these substitutions, keeping in mind that $s_{>k+1} = t_{>k+1}$. The fourth steps cancels terms, and the last step uses the definitions of $\boldsymbol{h}_{>k}$ and $\boldsymbol{H}_{>k}$. This completes the induction step for the first statement.

To prove the induction step for the second statement, we have

$$\begin{aligned}
p(\hat{s}_k = s_k \mid \hat{s}_{>k} = s_{>k}) &= \frac{1}{\|\boldsymbol{H}_{>k}\|_F^2} \mathrm{Tr}\left[\boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k\left[:, s_k, :\right]^\top \mathcal{A}_k\left[:, s_k, :\right] \cdot \boldsymbol{H}_{>k}\right] \\
&= \frac{1}{\|\boldsymbol{H}_{>k}\|_F^2} \sum_{r=1}^{R_j} \left(\boldsymbol{e}_r^\top \cdot \boldsymbol{H}_{>k}^\top \cdot \mathcal{A}_k\left[:, s_k, :\right]^\top \mathcal{A}_k\left[:, s_k, :\right] \cdot \boldsymbol{H}_{>k} \cdot \boldsymbol{e}_r\right) \\
&= \frac{1}{\|\boldsymbol{H}_{>k}\|_F^2} \sum_{r=1}^{R_j} \left(\boldsymbol{h}_{>k}^\top \cdot \mathcal{A}_k\left[:, s_k, :\right]^\top \mathcal{A}_k\left[:, s_k, :\right] \cdot \boldsymbol{h}_{>k}\right) \\
&= \sum_{r=1}^{R_j} p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k} \wedge \hat{r} = r) \cdot \frac{\|\boldsymbol{h}_{>k}\|^2}{\|\boldsymbol{H}_{>k}\|_F^2} \\
&= \sum_{r=1}^{R_j} p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k} \wedge \hat{r} = r) p(\hat{r} = r \mid \hat{t}_{>k} = t_{>k}) \\
&= p(\hat{t}_k = t_k \mid \hat{t}_{>k} = t_{>k}).
\end{aligned} \tag{5.13}$$

The first three steps are similar to the corresponding manipulations in the proof of the base case. The fourth step substitutes the conditional distribution of $\hat{t}_k$ from Equation (5.3), and the fifth step uses statement 1 from the claim (already proven for k). The sixth step uses the law of total probability, and the claim follows by induction. □

5.6.3 Proof of Theorem 5.1.1

We provide a short end-to-end proof that shows that Algorithms 15 and 16 correctly draw samples from $\boldsymbol{A}_{\leq j}^L$ (the matricization of the left-orthogonal core chain) according to the distribution of its squared row norms while meeting the runtime and space guarantees of Theorem 5.1.1.

Construction Complexity: The cost of Algorithm 15 follows from 3.3.2 with $M = IR_{k-1}$, the row count of $\boldsymbol{A}_k^L$ for $1 \leq k < j$. Using this lemma, construction of each sampling data structure Z_k requires time $O(I_k R_{k-1} R_k^2)$. The space required by sampler Z_k is $O(I_k R_{k-1} R_k)$; summing over all indices $1 \leq k \leq j$ matches the construction claim in Theorem 5.1.1.

Sampling Complexity: The complexity to draw samples in Algorithm 16 is dominated by calls to the RowSample procedure, which as discussed in Section 5.3 is $O(R_k^2 \log(I_k R_{k-1}/R_k))$ Summing the complexity over indices $1 \leq k \leq j$ yields the cost claimed by Theorem 5.1.1 to draw a single sample. The complexity of calling the RowSample procedure repeatedly dominates the complexity to update the history vector h over all loop iterations, which is $O\left(\sum_{k=1}^{j} R_{k-1} R_k\right)$ for each sample.

Correctness: Our task is to show in Algorithm 16, each sample t_d, $1 \leq d \leq J$, is a multi-index that follows the squared row norm distribution on $\boldsymbol{A}_{\leq j}$. To do this, we rely on lemmas proven earlier. For each sample, the $\hat{r}$ is a uniform random draw from $[1, ..., R_j]$, and $\boldsymbol{h}$ is initialized to the corresponding basis vector. By Equation (5.4) and Lemma 3.3.2, Line 5 from Algorithm 16 draws each index $\hat{t}_k$ correctly according to the probability distribution specified by Equation (5.3). The history vector is updated by Line 6 of the algorithm so that subsequent draws past iteration k of the loop are also drawn correctly according to Equation (5.3). Lemma 5.3.2 (relying on Lemma 5.3.1) shows that the multi-index $(\hat{t}_1, ..., \hat{t}_j)$ drawn according to Equation (5.3) follows the same distribution as $(\hat{s}_1, ..., \hat{s}_j)$, which was defined to follow the squared norm distribution on the rows of $\boldsymbol{A}_{\leq j}$. □

Chapter 6

Distributed Sparse Kernels for Machine Learning

Our penultimate technical chapter examines a kernel called Sampled Dense-Dense Matrix Multiplication (SDDMM): matrix multiplication with a mask on the output. We optimize the kernel for the distributed-memory parallel setting and revisit several techniques and applications from prior chapters. Graph neural networks (Chapter 2) and distributed matrix completion (related to tensor factorization from Chapter 4) make reappearances as use-cases for SDDMM.

Echoing techniques used in those chapters, we employ kernel fusion and optimizations that target imbalanced communication costs for different matrices. We also exploit a key duality property between distributed memory SDDMM and sparse-dense matrix multiplication, which has already been exploited in the shared-memory case [Nis+18]. Our communciation-avoiding algorithms achieve 3-5x speedup over the comparable routines in PETSc and scale up to 17,000 CPU cores of Cori, a Cray supercomputer at Lawrence Berkeley National Lab.

6.1 Introduction

Sampled Dense-Dense Matrix Multiplication and Sparse-Times-Dense Matrix Multiplication (SpMM) have become workhorse kernels in a variety of computations. Their use in matrix completion [CZ13] and document similarity computation [TP21] is well documented, and they are the main primitives used in graph learning [Vel+18]. The recent interest in Graph Neural Networks (GNNs) with self-attention has led libraries such as Amazon Deep Graph Library (DGL) [Wan+19] to expose SDDMM and SpMM primitives to users. Typical applications make a call to an SDDMM operation and feed the sparse output to an SpMM operation, repeating the pair several times with the same nonzero pattern (but possibly different values) for the sparse matrix. We refer to the back-to-back sequence of an SDDMM and SpMM as

FusedMM.

Several works optimize SDDMM and SpMM kernels in shared memory environments, or on accelerators such as GPUs [Hon+19; Nis+18; JHA20]. Separately, there have been prior efforts [Koa+16; Sel+21; ASA16] to optimize distributed memory SpMM by minimizing processor to processor communication. There is no significant work, however, on distributed algorithms for SDDMM or FusedMM.

We make three main contributions. First, we show that every sparsity-agnostic distributed-memory algorithm for SpMM can be converted into a distributed memory algorithm for SDDMM that uses the same input / output data distribution and has identical communication cost. Second, we give two methods to elide communication when executing an SDDMM and SpMM in sequence (FusedMM): replication reuse, which elides a second replication of an input matrix, and local kernel fusion, which allows local SDDMM and SpMM operations to execute on the same processor without intervening communication. These methods not only eliminate unnecessary communication rounds, they also enable algorithms that replicate dense matrices to scale to higher or lower replication factors (depending on which of the two methods used). Third, we demonstrate, both in theory and practice, these algorithms that replicate or shift sparse matrices perform best when the ratio of nonzeros in the sparse matrix to the number of nonzeros in the dense matrices is sufficiently low. When the number of nonzeros in the sparse matrix approaches the number of nonzeros in either of the dense matrices, algorithms that shift or replicate dense matrices become favorable.

Our work gives, to the best of our knowledge, the first benchmark of 1.5D SpMM and SDDMM algorithms that cyclically shift sparse matrices and replicate a dense matrix, which enables efficient communication scaling when the input dense matrices are tall and skinny. It also gives the first head-to-head comparison between sparse shifting and dense shifting 1.5D algorithms, which we show outperform each other depending on the problem setting.

6.2 Definitions

Symbol	Definition
$\boldsymbol{S}, \boldsymbol{R}$	$m \times n$ sparse matrix
$\boldsymbol{A}$	$m \times r$ dense matrix
$\boldsymbol{B}$	$n \times r$ dense matrix
ϕ	The ratio $\text{nnz}(\boldsymbol{S})/nr$
p	Total processor count
c	Replication Factor

Table 6.1: Notation for Chapter 6.

Let $\boldsymbol{A} \in \mathbb{R}^{m \times r}, \boldsymbol{B} \in \mathbb{R}^{n \times r}$ be dense matrices and let $\boldsymbol{S} \in \mathbb{R}^{m \times n}$ be a sparse matrix. The SDDMM operation produces a sparse matrix with sparsity structure identical to $\boldsymbol{S}$ given by

$$\text{SDDMM}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{S}) := \boldsymbol{S} \circledast (\boldsymbol{A} \cdot \boldsymbol{B}^\top) \tag{6.1}$$

Computing the output with a dense matrix multiplication followed by element-wise multiplication with $\boldsymbol{S}$ is inefficient, since we only need to compute the output entries at the nonzero locations of $\boldsymbol{S}$.

For clarity when describing our distributed memory algorithms, we distinguish between the SpMM operation involving $\boldsymbol{S}$ that takes $\boldsymbol{B}$ as an input vs. the operation involving $\boldsymbol{S}^\top$ that takes $\boldsymbol{A}$ as an input. Specifically, define

$$\text{SpMMA}(\boldsymbol{S}, \boldsymbol{A}) := \boldsymbol{S} \cdot \boldsymbol{B}$$
$$\text{SpMMB}(\boldsymbol{S}, \boldsymbol{B}) := \boldsymbol{S}^\top \cdot \boldsymbol{A}$$

where the suffix A or B on each operation refers to the matrix with the same shape as the output. The distinction is useful for applications such as alternating least squares and graph attention networks, which require both operations at different points in time. Finally, we borrow notation from prior works on the SDDMM-SpMM sequence [RSA21] and use FusedMMA, FusedMMB to denote operations that are compositions of SDDMM with SpMMA or SpMMB, given as

$$\text{FusedMMA}(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{B}) := \text{SpMMA}(\text{SDDMM}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{S}), \boldsymbol{B})$$
$$\text{FusedMMB}(\boldsymbol{S}, \boldsymbol{A}, \boldsymbol{B}) := \text{SpMMB}(\text{SDDMM}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{S}), \boldsymbol{A})$$

6.3 Related Work

6.3.1 Shared Memory Optimization

Local SpMM and SDDMM operations are bound by memory bandwidth compared to dense matrix multiplication. Accelerating either SpMM or SDDMM in a shared memory environment, such as a single CPU node or GPU, typically involves blocking a loop over the nonzeros of $\boldsymbol{S}$ to optimize cache reuse of the dense matrices [Nis+18]. For blocked SDDMM and SpMM kernels, the traffic between fast and slow memory is exactly modeled by the edgecut-1 metric of a hypergraph partition induced on $\boldsymbol{S}$ (treating the rows of $\boldsymbol{S}$ as hyper-edges and the columns as vertices, with nonzeros indicating pins). Jiang et al. [JHA20] reorder the sparse matrix to minimize the connectivity metric, thereby reducing memory traffic. Instead of reordering $\boldsymbol{S}$, Hong et al. [Hon+19] adaptively choose a blocking shape tuned to the sparsity structure to optimize performance. Both optimizations require expensive processing steps on the sparse matrix $\boldsymbol{S}$, which are typically amortized away by repeated calls to the kernel.

6.3.2 Distributed Sparsity-Aware Algorithms

We can divide distributed-memory implementations for both SpMM and SDDMM into two categories: sparsity-aware algorithms, and sparsity-agnostic bulk communication approaches. In the former category, the dense input matrices are partitioned among processors along with the sparse matrix nonzeros (i, j), such that each processor owns at least one of $\boldsymbol{A}_{i:}$ or $\boldsymbol{B}_{j:}$. When processing (i, j), if a processor does not own one of the two dense rows needed, it fetches the embedding from another processor that owns it [ASA16]. Such approaches work well when $\boldsymbol{S}$ is very sparse. They also benefit from graph / hypergraph partitioning to reorder the sparse matrix, which can reduce the number of remote fetches that each processor must make while maintaining load balance. Such approaches suffer, however, from the overhead of communicating the specific embeddings requested by processors, which typically requires round-trip communication. As $\boldsymbol{S}$ gets denser, processors are better off broadcasting all of their embeddings.

6.3.3 Sparsity Agnostic Bulk Communication Algorithms

Sparsity agnostic algorithms resemble distributed dense matrix multiplication algorithms by broadcasting, shifting, and reducing block rows and block columns of $\boldsymbol{A}, \boldsymbol{B}$, and $\boldsymbol{S}$. These methods cannot significantly benefit from graph partitioning and they often rely on a random permutation of the sparse matrix to load balance among processors.

Such algorithms are typically described as 1D, 1.5D, 2D, 2.5D, or 3D. 1D and 2D algorithms are memory-optimal, with processors requiring no more aggregate memory than the storage required for inputs and outputs (up to a small constant factor for communication buffering). 1.5D, 2.5D, and 3D algorithms increase collective memory consumption of at least one of the three operands to asymptotically decrease communication costs. In this work, we will consider only 1.5D and 2.5D algorithms, since 1D, 2D, and 3D algorithms are special cases of these two.

Koanantakool et al. show when $\boldsymbol{A}, \boldsymbol{B}$ and $\boldsymbol{S}$ are all square, 1.5D SpMM algorithms that cyclically shift the sparse matrix yield superior performance [Koa+16]. They only benchmark multiplication of all square matrices, which does not cover the more common case where $r \ll m, n$. In graph embedding problems, r is typically between 64 and 512, whereas m and n range up to hundreds of millions. For this case, Tripathy et al. [TYB20] introduced CAGNET, which trains graph neural networks on hundreds of GPUs using 1.5D and 2.5D distributed SpMM operations. They, along with Selvitopi et al. [Sel+21], showed that 2D algorithms for SpMM suffer from diminished arithmetic intensity as processor count increases. In contrast, both demonstrate that 1.5D algorithms communicating dense matrices exhibit excellent scaling with processor count. Selvitopi et al. did not consider 2.5D algorithms, however, and neither work benchmarked 1.5D algorithms that cyclically shift sparse matrices. In addition, the 2.5D algorithms in CAGNET only replicate the dense matrix, whereas it is

also possible to construct implementations that only replicate the sparse matrix.

6.3.4 Background on Dense Distributed Linear Algebra

Our sparsity-agnostic algorithms resemble 1.5D and 2.5D variants of the the Cannon and SUMMA distributed dense GEMM algorithms [Can69], [GW95]. In the 2.5D Cannon-like algorithm to compute compute the matrix product $\boldsymbol{X} = \boldsymbol{YZ}$, the submatrix domains of the output $\boldsymbol{X}$ are **replicated** among processors [SD11]. E.g., for every entry $\boldsymbol{X}_{ij}$, different processors compute different parts of the inner product $\boldsymbol{Y}_{i:} \cdot \boldsymbol{Y}_{:j}$ and reduce their results at the end. The inputs are both **propagated** during the algorithm: submatrices of $\boldsymbol{A}$ and $\boldsymbol{B}$ are cyclically shifted between processors in stages. The SUMMA algorithm replaces the stages of cyclic shifts with broadcast collectives. 1.5D variants of these algorithms keep at least one of the three matrices **stationary**: submatrices of a stationary matrix are distributed among processors, but they are not broadcast, reduced, or shifted. It is possible to modify these algorithms so that an input matrix is replicated rather than an output.

Our algorithm design is dictated by choosing which submatrices we keep stationary, replicate, and propagate. While these choices do not matter for dense GEMM if all matrices are square, they impact our kernels due to the sparsity of $\boldsymbol{S}$ and the extremely rectangular shapes of $\boldsymbol{A}$ and $\boldsymbol{B}$.

Kwasniewski et al. [Kwa+19] recently proposed COSMA, which uses the classic red-blue pebbling game to design an optimal parallelization scheme for distributed dense GEMM with matrices of varying shapes. Their communication-minimizing algorithms achieve excellent performance relative to SCALAPACK and recent high performance matrix multiplication work. COSMA, however, does not account for sparsity in either inputs or outputs. Our work, furthermore, focuses on the FusedMM computation pattern commonly found in applications, which provides further opportunities for communication minimization than considering the SpMM and SDDMM kernels in isolation.

6.4 Distributed Memory Algorithms for SDDMM and FusedMM

This section highlights the connection between SpMM and SDDMM and gives a high-level procedure to convert between algorithms that compute each one. This enables us to use a single input distribution to compute SDDMM, SpMMA, and SpMMB, at the cost of possibly replicating the sparse matrix $\boldsymbol{S}$ by a factor of 2 to store its transpose. Each kernel requires the same amount of communication. Next, we give two *communication elision* strategies when we execute an SDDMM and an SpMM operation in sequence, with the output of the SDDMM feeding into the SpMM (a FusedMM operation). Optimizing for the sequence of the two kernels reduces communication overhead compared to simply executing

one distributed algorithm followed by the other. Subsequently, we detail the implementation of those high-level strategies.

6.4.1 The Connection between SDDMM and SpMM

SDDMM and SpMM have identical data access patterns, which becomes clear when we compare their serial algorithms (take SpMMA here). Letting sparse matrix $\boldsymbol{R}$ be the SDDMM output, we have $\boldsymbol{R}_{ij} = \boldsymbol{S}_{ij} \left(\boldsymbol{A}_{i:} \cdot \boldsymbol{B}_{j,:}^\top \right)$ with $\boldsymbol{R}$ set to zero where $\boldsymbol{S}$ is 0. Compare this equation to each step required to compute $\boldsymbol{A}$ += SpMMA($\boldsymbol{S}, \boldsymbol{B}$): for every nonzero (i, j) of $\boldsymbol{S}$, we perform the update $\boldsymbol{A}_{i:}$ += $\boldsymbol{S}_{ij}\boldsymbol{B}_{j:}$. For both SDDMM and SpMMA, each nonzero of $\boldsymbol{S}$ results in an interaction between a row of $\boldsymbol{A}$ and a row of $\boldsymbol{B}$.

Now consider any distributed memory algorithm for SpMMA that does not replicate its input or output matrices during computation. For each nonzero (i, j) and for every index $k \in [1, r]$, this algorithm must co-locate $\boldsymbol{S}_{ij}$, $\boldsymbol{A}_{ik}$, and $\boldsymbol{B}_{jk}$ on some processor and compute $\boldsymbol{A}_{ik}$ += $\boldsymbol{S}_{ij}\boldsymbol{B}_{jk}$. Transform the algorithm as follows:

1. Change the input sparse matrix $\boldsymbol{S}$ to an output matrix initialized to 0.
2. Change $\boldsymbol{A}$ from an output matrix to an input matrix.
3. Have each processor execute local update $\boldsymbol{S}_{ij}$ += $\boldsymbol{A}_{ik}\boldsymbol{B}_{jk}$.

Then for all nonzeros (i, j) and every index $k \in [1, r]$, the processors collectively execute computations to overwrite $\boldsymbol{S}$ with $\boldsymbol{A}\boldsymbol{B}^\top$ masked at the nonzeros of $\boldsymbol{S}$. This is exactly the SDDMM operation up to multiplication of $\boldsymbol{A}\boldsymbol{B}^\top$ with the values initially in $\boldsymbol{S}$. If the output distribution of $\boldsymbol{S}$ is identical to its input distribution after execution, then the post-multiplication with the initial values in $\boldsymbol{S}$ does not require additional communication. A similar transformation converts an algorithm for SpMMB into an algorithm for SDDMM.

We can extend this transformation procedure to algorithms that replicate input and output matrices. Typically, inputs are replicated via broadcast, while output replication requires a reduction at the end of the algorithm to sum up temporary buffers across processors. Since we interchange in the input / output role between matrices $\boldsymbol{A}$ and $\boldsymbol{S}$, we convert broadcasts of the values of $\boldsymbol{S}$ in SpMMA to reductions of its values in SDDMM, and reductions of $\boldsymbol{A}$ to broadcasts.

Algorithms 17 and 18 illustrate the transformation procedure outlined above by giving unified algorithms to compute SDDMM, SpMMA, and SpMMB. The local update executed by each processor changes based on the kernel being executed, and initial broadcasts / terminal reductions depend on whether matrices function as inputs or outputs.

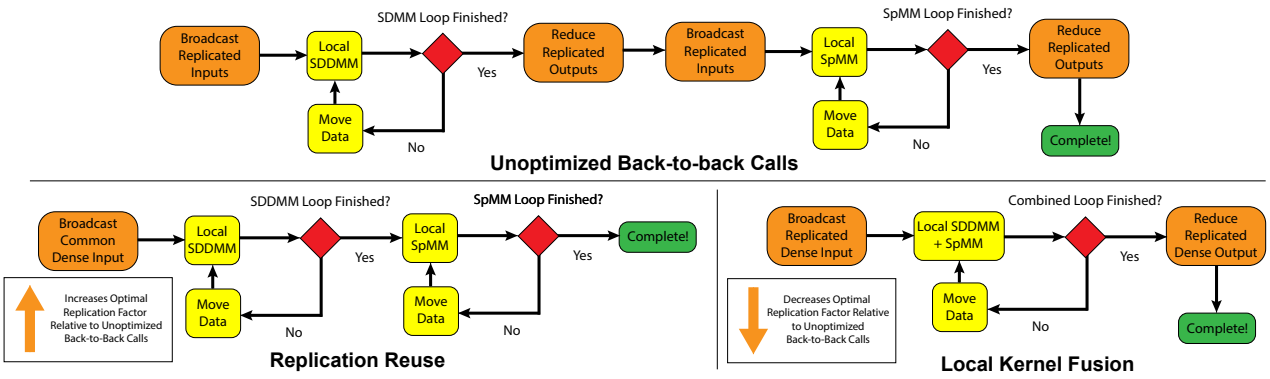


Figure 6.1: Communication eliding strategies illustrated. The benefit from replication reuse and local kernel fusion arises from an increased or decreased optimal replication factor, not just the elimination of communication phases.

6.4.2 Strategies for Distributed Memory FusedMM

FusedMMA computes each row of its output as

$$\text{FusedMMA}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{S})_{i:} := \sum_{j \mid (i,j) \in nnz(\boldsymbol{S})} \boldsymbol{S}_{ij} \langle \boldsymbol{A}_{i:}, \boldsymbol{B}_{j:} \rangle \boldsymbol{B}_{j:}$$

which is a sum of rows j of matrix $\boldsymbol{B}$ weighted by the dot products between rows j of $\boldsymbol{B}$ and rows i of $\boldsymbol{A}$. The analogous equation for FusedMMA replaces $\boldsymbol{S}_{ij}$ with $\boldsymbol{S}_{ij}^\top = \boldsymbol{S}_{ji}$. The simplest distributed implementation for FusedMMA computes the intermediate SDDMM, stores it temporarily, and feeds the result directly to SpMMA, exploiting the common input / output data layouts in the previous section. The communication cost for this implementation is twice that of either a single SpMMA operation or SDDMM operation. Such an algorithm for FusedMMA takes the following structure:

1. Replicate dense matrices $\boldsymbol{A}, \boldsymbol{B}$ in preparation for SDDMM (if either $\boldsymbol{A}$ or $\boldsymbol{B}$ is replicated)
2. Propagate matrices, compute SDDMM
3. Reduce SDDMM output $\boldsymbol{R}$ (if sparse matrix is replicated)
4. Replicate dense matrix $\boldsymbol{B}$ in preparation for SpMMA (if $\boldsymbol{B}$ is replicated)
5. Propagate matrices, compute SpMMA
6. Reduce output matrix $\boldsymbol{A}$ (if $\boldsymbol{A}$ is replicated)

In the outline above, propagation refers to any communication that excludes replication of inputs / outputs (for example, cyclic shifts of buffers in Cannon's algorithm). Increased

replication factors allow us to decrease propagation cost, but increase the required memory and result in a higher cost to perform the replication. Choosing the optimal replication factor involves minimizing the sum of the communication overhead paid in replication and propagation. We can save communication in two ways: first by reducing the replication cost, and second by reducing the propagation cost. Figure 6.1 illustrates both.

(1) Replication Reuse: This approach performs only a single replication of an input matrix in both the SDDMM and SpMM computations. If we replicate the dense matrix $\boldsymbol{B}$, before the SDDMM operation, we don't require a second replication before the SpMM phase, nor do we need a reduction of the output buffer.

(2) Local Kernel Fusion: This approach combines the two propagation steps, 2 and 5, into a single phase while only replicating matrix $\boldsymbol{A}$. Using locally available data at each propagation step, it requires performing a local SDDMM and SpMM in sequence without intermediate communication. Thus, local kernel fusion with any data distribution that divides $\boldsymbol{A}$ and $\boldsymbol{B}$ by columns would yield an incorrect result. From the standard definition of FusedMMA, we must compute the dot product $\langle \boldsymbol{A}_{i:}, \boldsymbol{B}_{j:} \rangle$ that scales every row of $\boldsymbol{B}$ before aggregating those rows, requiring us to complete the SDDMM before performing any aggregation. The 1.5D algorithm that replicates and shifts dense matrices (Section 6.5) co-locates entire rows of $\boldsymbol{A}$ and $\boldsymbol{B}$ on the same processor during the stages of computation, and is the only candidate that can take advantage of local kernel fusion. Besides the communication savings that they offer, optimized local FusedMM functions (e.g., [RSA21]) can improve performance by eliding intermediate storage of the SDDMM result.

Although applying either method would provide moderate communication savings without modifying any other aspect of the algorithm, their utility really lies in allowing us to change the degree of replication for any algorithm that replicates a dense matrix. Algorithms that employ replication reuse can achieve lower communication overhead at *higher* replication factors compared to an unoptimized sequence of calls. Intuitively, increasing the replication factor enables the algorithm to trade away more overhead in the propagation phase before the cost of replication becomes overwhelming. By contrast, algorithms that employ local kernel fusion can achieve lower communication overhead at *lower* replication factors. The algorithm requires less replication to balance off the lower cost of propagation. Note that these strategies are mutually exclusive; applying them both to 1.5D dense shifting algorithms would require propagating a separate accumulation buffer in the propagation phase, which destroys the benefit of local kernel fusion.

While we have discussed strategies for FusedMMA, we obtain algorithms for FusedMMB by interchanging the roles of $\boldsymbol{A}$ and $\boldsymbol{B}$ and replacing matrix $\boldsymbol{S}$ with its transpose $S^\top$. In practice, this amounts to storing two copies of the sparse matrix across all processors, one with the coordinates transposed.

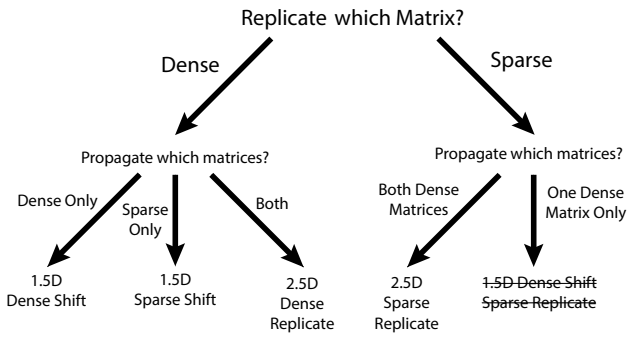


Figure 6.2: Design choices in SpMM algorithms. We do not consider the 1.5D sparse replicating, dense shifting algorithm, since it is inferior to the 2.5D sparse replicating algorithm.

6.5 Algorithm Descriptions

We consider formulations for SpMM detailed previously [Koa+16; TYB20; Sel+21], but some with key modifications. We arrive at each algorithm by deciding which of the three matrices $\boldsymbol{A}, \boldsymbol{B}$, and $\boldsymbol{S}$ to replicate, propagate, or keep stationary (see figure 6.2). We consider formulations where only a single matrix is replicated enabling us to scale to higher replication factors and take advantage of both communication-eliding strategies described above.

Let p be the total processor count. We list the input distributions of all matrices in Table 6.2. Each matrix is partitioned by blocks into a grid with the specified dimension. The product of these dimensions may exceed the total processor count because processors can own multiple non-contiguous blocks, as in block row / column cyclic distributions. The third column of Table 6.2 gives the processor that initially owns a block (i, j) as an (u, v) or (u, v, w) tuple, which identifies the processor position within a 2D / 3D grid. For the 2.5D sparse replicating algorithm, "*" means all processors along the third axis of the grid share the coordinates of block (i, j). Figure 6.3 illustrates the data movement in our algorithms for 8 processors with a replication factor of 2. We will refer to the grid axis along which inputs are reduced or gathered as the "fiber axis". It is the second dimension of the computational grid for 1.5D algorithms and the third dimension for 2.5D algorithms.

To analyze our algorithms, we use the α-β-γ model where α is the per-message latency, β is the inverse-bandwidth, and γ is the cost per FLOP performed locally on each processor. Since our algorithms communicate blocks of matrices, they exchange at most a small multiple of p messages, each of which is a large block of a matrix. Therefore, we focus on minimizing the number of data words communicated by each processor, the coefficient of inverse-bandwidth in our model. In the analysis that follows, we use "communication cost" to mean the maximum amount of time that any processor spends sending and receiving messages.

We assume sends and receives can make independent progress on each node and use the costs for collectives in the literature [Cha+07]. To aid the analysis, assume $m \approx n$, and let ϕ be the ratio of the number of nonzeros in $\boldsymbol{S}$ to the number of nonzeros of dense matrix $\boldsymbol{B}$, i.e. $\phi = \text{nnz}(\boldsymbol{S})/nr$.

Matrix	Grid	Owner of Block (i, j)
1.5D Dense Shifting		
$\boldsymbol{A}$	$p \times 1$	$(i/c, i\%c)$
$\boldsymbol{B}$	$p \times 1$	$(i/c, i\%c)$
$\boldsymbol{S}, \boldsymbol{R}$	$(p/c) \times p$	$(i, j\%c)$
1.5D Sparse Shifting		
$\boldsymbol{A}$	$p \times (p/c)$	$(j, i\%c)$
$\boldsymbol{B}$	$p \times (p/c)$	$(j, i\%c)$
$\boldsymbol{S}, \boldsymbol{R}$	$1 \times p$	$(j/c, j\%c)$
2.5D Dense Replicating		
$\boldsymbol{A}$	$\sqrt{pc} \times \sqrt{p/c}$	$(i/c, j, i\%c)$
$\boldsymbol{B}$	$\sqrt{pc} \times \sqrt{p/c}$	$(i/c, j, i\%c)$
$\boldsymbol{S}, \boldsymbol{R}$	$\sqrt{p/c} \times \sqrt{pc}$	$(i, j/c, j\%c)$
2.5D Sparse Replicating		
$\boldsymbol{A}$	$p \times \sqrt{p/c}$	$(i/c, j, i\%c)$
$\boldsymbol{B}$	$\sqrt{p/c} \times p$	$(i, j/c, j\%c)$
$\boldsymbol{S}, \boldsymbol{R}$	$\sqrt{p/c} \times \sqrt{p/c}$	$(i, j, *)$

Table 6.2: Input Matrix Distributions Before Replication

6.5.1 1.5D Dense Shifting, Dense Replicating

1.5D algorithms operate on a $(p/c) \times c$ processor grid, where $c \geq 1$ is the replication factor. We can interpret the 1.5D dense shifting, dense replicating algorithm as c layers of concurrently executing 1D algorithms. To decrease communication as the processor count increases, the 1.5D dense shifting, dense replicating algorithm replicates one of the two dense matrix inputs, propagates the other dense matrix, and keeps the remaining sparse matrix stationary on each processor.

The procedure is detailed in Algorithm 17 and illustrated in Figure 6.3. The sparse matrix $\boldsymbol{S}$ is stored with a column block cyclic distribution across the grid layers. The processors begin by allocating a buffer $\boldsymbol{T}$ for the replicated matrix $\boldsymbol{A}$, keeping $\boldsymbol{T}$ initially 0 if $\boldsymbol{A}$ is an output buffer and otherwise gathering blocks of $\boldsymbol{A}$ of size $(n/p) \times r$ within each fiber if $\boldsymbol{A}$ is an input (replication). For p/c phases, algorithms cyclically shift their local blocks of the matrix B (propagation). Finally, if $\boldsymbol{A}$ is the output of the computation, the buffer $\boldsymbol{T}$ is reduced

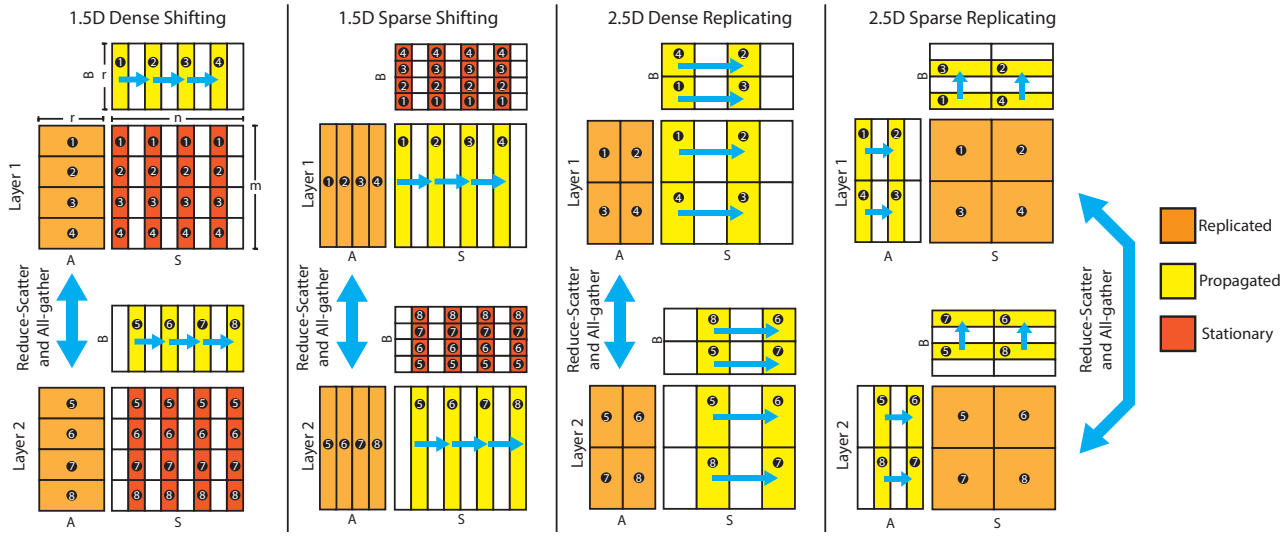


Figure 6.3: Data movement after replication, illustrated for $p = 8$ processors and replication factor $c = 2$. Within each layer, submatrices shown in yellow are cyclically shifted from processor to processor (propagation) in the direction indicated by single-headed blue arrows. Submatrices in orange are replicated across layers and participate in reduce-scatter / allgather operations outside the propagation phase. Submatrices in red remain stationary on each processor. Black numbered circles give the processor owning each submatrix at the beginning of each algorithm (note the initial skew for 2.5D algorithms).

and scattered to processors within the fiber. Increasing the replication factor c decreases communication costs incurred within each layer due to cyclic shifts of $\boldsymbol{B}$, but increases the communication cost of allgather / reduce-scatter primitives between layers.

Communication analysis, No Elision: Consider a pair of SDDMM and SpMMA operations that execute as two sequential calls to Algorithm 17 with no intervening communication elision. The allgather and reduce-scatter primitives operate on dense block of size nrc/p within each fiber, which contains c processors. The allgather within SDDMM and reduce-scatter in SpMMA communicate $2((c-1)/c)(nrc/p)$ words. Each layer contains p/c processors and executes $2p/c$ cyclic shifts of dense blocks with size nr/p. Each cyclic shift communicates nr/p words. Multiplying by the number of phases and adding to the cost of communication along fibers gives a communication cost $nr(2(c-1)/p + 2/c)$. Differentiating this expression and setting equal to 0, we find that $c = \sqrt{p}$ minimizes the cost.

Communication analysis, Replication Reuse: If we apply replication reuse to optimize FusedMMA (by interchanging the roles of $\boldsymbol{A}$ and $\boldsymbol{B}$, replacing $\boldsymbol{S}$ with $\boldsymbol{S}^\top$, and performing an SpMMB), we eliminate the terminal reduce-scatter operation, yielding a communication cost $nr((c-1)/p + 2/c)$. The optimal replication factor c becomes $c = \sqrt{2p}$. Since we have less overall communication within each fiber, we can afford to increase replication further to drive

Algorithm 17 Unified 1.5D Algorithm Moving Dense Matrices for SpMMA, SpMMB, SDDMM

Require: Mode $\in$ {SDDMM, SpMMA, SpMMB}, Dense Matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and sparse matrices $\boldsymbol{S}$, $\boldsymbol{R}$ distributed on a $\frac{p}{c} \times c$ grid

Ensure: One of $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{R}$ filled with the output of an SpMMA, SpMMB, or SDDMM computation, depending on Mode

$\boldsymbol{T}$:= Zeros($cm/P, R$)
if Mode $\in$ {SDDMM, SpMMB} **then**
  Allgather($\boldsymbol{A}_{\text{loc}}$, $\boldsymbol{T}$, fiber-axis)
for $i = 1$ **to** $\frac{p}{c}$ **do**
  if Mode == SDDMM **then**
    $\boldsymbol{R}_{\text{loc}}$ += SDDMM($\boldsymbol{T}, \boldsymbol{B}_{\text{loc}}, S_{\text{loc},i}$)
  else if Mode == SpMMA **then**
    $\boldsymbol{T}$ += SpMMA($\boldsymbol{S}_{\text{loc},i}, \boldsymbol{B}_{\text{loc}}$)
  else
    $\boldsymbol{B}_{\text{loc}}$ += SpMMB($\boldsymbol{S}_{\text{loc},i}, \boldsymbol{T}$)
  Cyclic Shift $\boldsymbol{B}_{\text{loc}}$ within layer
if Mode == SpMMA **then**
  Reduce-Scatter($\boldsymbol{T}$, $\boldsymbol{A}_{\text{loc}}$, fiber-axis)
return One of $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{R}$ filled with the computed result

down the cost of cyclic shifts within each layer. The ratio of the communication cost to the version without FusedMM elision for the optimal choice of c in each is $(1 - 2\sqrt{2p})/(2 - 4\sqrt{p})$, which for $p \to \infty$ tends to $1/\sqrt{2}$. Thus, we save roughly 30% of communication compared to executing two kernels in sequence.

Communication analysis, Local Kernel Fusion: If we use the local kernel fusion strategy to optimize FusedMMA, we need only a single round of p/c cyclic shifts instead of two. We still require an initial allgather and terminal reduce-scatter, giving $nr(2(c-1)/p + 1/c)$ words communicated. The smaller communication cost from cyclic shifts yields an optimal replication factor $c = \sqrt{p/2}$, since communication within each fiber costs more relative to the cyclic shifts. The ratio of the communication cost of local kernel fusion to the case without any communication elision for the optimal choice of replication factors approaches $1/\sqrt{2}$ as $p \to \infty$, meaning that either communication eliding strategy produces the same reduction in communication. Local kernel fusion, however, is the only algorithm that permits local FusedMMA operations on each processor, since it does not divide row and column embeddings along the short r-axis.

6.5.2 1.5D Sparse Shifting, Dense Replicating

1.5D algorithms that shift the sparse matrix operate analogously to the dense shifting, dense replicating algorithm. In this case, the sparse matrix is propagated while the non-replicated dense matrix is stationary on each processor. Koanantakool et al. [Koa+16] showed that 1.5D SpMM algorithms are more efficient than 2.5D algorithms when $m{=}n{=}r$. Their work, however, both replicates and shifts the sparse matrix to decrease latency at high processor counts and improve local SpMM throughput. Their approach was also motivated by working with all square matrices, which means that any dense replication proves prohibitively expensive in comparison to the sparse matrix communication. By contrast, our dense matrices are tall-skinny. We cyclically shift the sparse matrix and replicate the dense input matrix to reduce communication at high processor counts. This is advantageous when $\text{nnz}(\boldsymbol{S})$ is significantly smaller than nr. The c layers of the grid still participate in scatters / gathers of an input matrix, but within each layer, block rows of the sparse matrix $\boldsymbol{S}$ are cyclically shifted, and we now divide $\boldsymbol{A}$ and $\boldsymbol{B}$ by block columns rather than block rows. Dividing the dense matrices by columns, however, can significantly hurt local kernel throughput on some architectures [TYB20].

Communication Analysis, FusedMM: We again apply replication reuse to optimize the FusedMM operation. The algorithm incurs the communication cost of $2p/c$ cyclic shifts of an average of $\text{nnz}(\boldsymbol{S})/p$ nonzeros each; each nonzero consists of three words when the sparse matrix is in coordinate format. We add this to a single allgather operation on blocks of size nr/p across the c processes in each fiber to produce an aggregate cost

$$6\left(\frac{\text{nnz}(\boldsymbol{S})}{c}\right) + \frac{nr(c-1)}{p} \tag{6.2}$$

where the first term arises from cyclic shifts and the second term arises from the allgather. For ease of analysis, we replace $\text{nnz}(\boldsymbol{S})$ with ϕnr (see the earlier definition of ϕ) and optimize for c to get $c = \sqrt{6p\phi}$. When ϕ is low and $c < 1$, we interpret this as indicating that no amount of replication is favorable. When $c < 1$, the optimal communication cost is $6\phi nr$, and when $c > 1$ (which occurs at higher processor counts), the communication cost is

$$\frac{nr}{\sqrt{p}}\left(2\sqrt{6\phi} - \frac{1}{\sqrt{p}}\right) \tag{6.3}$$

Ignoring the lower order $1/\sqrt{p}$ term, Equation 6.3 indicates that when ϕ is low, the sparse shifting algorithm performs better than the 1.5D dense shifting algorithm. As ϕ increases, the dense shifting 1.5D algorithm performs better.

6.5.3 2.5D Dense Replicating Algorithms

2.5D algorithms operate on a $\sqrt{p/c} \times \sqrt{p/c} \times c$ grid. We can interpret each layer as executing a concurrent version of SUMMA / Cannon on a square 2D grid. This 2.5D algorithm replicates

a dense matrix and cyclically shifts a sparse matrix and the remaining dense matrix within each layer. Algorithm 18 gives pseudocode of the procedure; processors within each layer cyclically shift both $\boldsymbol{S}$ and $\boldsymbol{B}$ along processor row, resp. column, axes, while blocks of $\boldsymbol{A}$ are reduce-scattered or gathered along the fiber-axis at the beginning (and end). As written, the algorithm requires an initial shift of its inputs to correctly index blocks of the matrices. In practice, applications do not need to perform this initial shift if they fill the input and output buffers appropriately. Thus, we don't include the initial shift in our communication analysis.

Communication Analysis, FusedMM: We can only apply replication reuse when using 2.5D dense replicating algorithms, as the input dense matrices are divided by columns among processors. The communication analysis is similar to the 1.5D FusedMM algorithms, so we omit the details for brevity. The optimal replication factor is $c = p^{1/3}(1+3\phi)^{2/3}$. The resulting optimal cost is

$$\frac{nr}{p^{2/3}}\left(\frac{2+6\phi}{(1+3\phi)^{1/3}} + (1+3\phi)^{2/3} - \frac{1}{p^{1/3}}\right) = O\left(\frac{nr\phi^{2/3}}{p^{2/3}}\right)$$

Notice the factor $p^{2/3}$ in the denominator, as opposed to $p^{1/2}$ for the 1.5D algorithms. As with the 1.5D sparse shifting algorithm, replication is less favorable when ϕ is low and becomes more favorable as ϕ increases.

6.5.4 2.5D Sparse Replicating Algorithms

2.5D sparse replicating algorithms operate similarly to the dense replicating version, except that both dense matrices cyclically shift within each layer and the nonzeros of the sparse matrix are reduce-scattered / gathered along the fiber axis. The algorithm has the attractive property that only the nonzero values need to be communicated along the fiber axis, since the nonzero coordinates do not change between function calls. In contrast to the dense replicating algorithm, the 2.5D sparse replicating algorithm divides the dense embedding matrices into successively more block columns as c increases. Because this algorithm does not replicate dense matrices, it cannot benefit from communication elision when performing a FusedMM operation.

Communication Analysis, No Communication Elision To execute an SDDMM and SpMM in sequence, the sparse replicating 2.5D algorithm executes an initial allgather to accumulate the sparse matrix values at each layer of the processor grid, and an all-reduce (reduce-scatter + allgather) between the SDDMM and SpMM calls. Over both propagation steps, it executes $4\sqrt{pc}$ cyclic shifts of dense blocks containing nr/p words. The optimal replication factor is $c = p^{1/3}\left(2/(3\phi)\right)^{2/3}$, and the resulting optimal communication cost is

$$\frac{nr\phi^{1/3}}{p^{2/3}}\left(\sqrt[3]{(2^5)3\phi} + \sqrt[3]{(2^2)3} - \frac{3\phi^{2/3}}{p^{1/3}}\right) = O\left(\frac{nr\phi^{1/3}}{p^{2/3}}\right)$$

Algorithm 18 Unified 2.5D Dense Replicating Algorithm for SpMMA, SpMMB, SDDMM

Require: Mode $\in$ {SDDMM, SpMMA, SpMMB}; Dense matrices $\boldsymbol{A}$, $\boldsymbol{B}$ and sparse matrices $\boldsymbol{S}$, $\boldsymbol{R}$ distributed on a $\sqrt{p/c} \times \sqrt{p/c} \times c$ grid (per Table 6.2)

Ensure: One of $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{R}$ filled with the output of an SpMMA, SpMMB, or SDDMM computation, depending on Mode

if Input Matrices are not shifted **then**
  Cyclic Shift $\boldsymbol{S}_{\text{loc}}$ to processor *row-rank* $-$ *col-rank*
  Cyclic Shift $\boldsymbol{B}_{\text{loc}}$ to processor *col-rank* $-$ *row-rank*
$\boldsymbol{T}$:= Zeros$(cm/P, R)$
if Mode $\in$ {SDDMM, SpMMB} **then**
  Allgather($\boldsymbol{A}_{\text{loc}}$, $\boldsymbol{T}$, fiber-axis)
for $i = 1$ **to** $\sqrt{p/c}$ **do**
  if Mode == SDDMM **then**
    $\boldsymbol{R}_{\text{loc}}$ += SDDMM$(\boldsymbol{T}, \boldsymbol{B}_{\text{loc}}, S_{\text{loc}})$
  else if Mode == SpMMA **then**
    $\boldsymbol{T}$ += SpMMA$(\boldsymbol{S}_{\text{loc}}, \boldsymbol{B}_{\text{loc}})$
  else
    $\boldsymbol{B}_{\text{loc}}$ += SpMMB$(\boldsymbol{S}_{\text{loc}}, \boldsymbol{T})$
  Cyclic Shift $\boldsymbol{S}_{\text{loc}}$ by 1 within row clockwise
  Cyclic Shift $\boldsymbol{B}_{\text{loc}}$ by 1 within column clockwise
if Mode == SpMMA **then**
  Reduce-Scatter($\boldsymbol{T}$, $\boldsymbol{A}_{\text{loc}}$, fiber-axis)
return One of $\boldsymbol{A}$, $\boldsymbol{B}$, or $\boldsymbol{R}$ filled with the computed result

Note the factor $\phi^{1/3}$ instead of $\phi^{2/3}$; for $\phi > 1$, this is an improvement, and when $\phi < 1$ but is sufficiently far away from 0, the difference is subsumed by the constant factors in front of the expression. Note also that the optimal value of c has $\phi^{2/3}$ in its denominator, indicating that a sparser input $\boldsymbol{S}$ benefits from higher replication.

6.5.5 Summary

Table 6.3 summarizes the analysis in the previous sections by giving communication and latency costs for each of the algorithms above embedded in the FusedMM procedure. It also lists the dimensions of the matrices used in each local call to either SDDMM or SpMM. Table 6.4 gives the optimal replication factors for our algorithms.

Our theory predicts that 1.5D algorithms with correctly tuned replication factors will marginally outperform the 2.5D algorithms over a range of processor counts, sparse matrix densities, and dense matrix widths. The choice to use a dense shifting or sparse shifting 1.5D algorithm depends on the value of ϕ in the specific problem instance. Figure 6.6 (Section 6.6) illustrates and evaluates these predictions.

Algorithm	Messages	Words Communicated	Local Dim, S	Local Dim, B
1.5D Dense Shift, RR	$\frac{2p}{c} + (c-1)$	$nr\left(\frac{2}{c} + \frac{(c-1)}{p}\right)$	$\frac{nc}{p} \times \frac{n}{p}$	$\frac{n}{p} \times r$
1.5D Dense Shift, LKF	$\frac{p}{c} + 2(c-1)$	$nr\left(\frac{1}{c} + \frac{2(c-1)}{p}\right)$	$\frac{nc}{p} \times \frac{n}{p}$	$\frac{n}{p} \times r$
1.5D Sparse Shift, RR	$\frac{2p}{c} + (c-1)$	$nr\left(\frac{6\phi}{c} + \frac{c-1}{p}\right)$	$\frac{nc}{p} \times n$	$n \times \frac{r}{p}$
2.5D Dense Replicate, RR	$4\sqrt{\frac{p}{c}} + (c-1)$	$\frac{nr}{\sqrt{pc}}\left(6\phi + 2 + \frac{c^{3/2}}{\sqrt{p}} - \frac{\sqrt{c}}{\sqrt{p}}\right)$	$\frac{n\sqrt{c}}{\sqrt{p}} \times \frac{n}{\sqrt{pc}}$	$\frac{n}{\sqrt{pc}} \times \frac{r\sqrt{c}}{\sqrt{p}}$
2.5D Sparse Replicate, NKE	$4\sqrt{\frac{p}{c}} + 3(c-1)$	$\frac{nr}{\sqrt{p}}\left(\frac{4}{\sqrt{c}} + \frac{3\phi(c-1)}{\sqrt{p}}\right)$	$\frac{n\sqrt{c}}{\sqrt{p}} \times \frac{n\sqrt{c}}{\sqrt{p}}$	$\frac{n}{\sqrt{pc}} \times \frac{r\sqrt{c}}{\sqrt{p}}$

Table 6.3: Latency, Bandwidth, and Matrix Dimensions in Local Kernel Calls for FusedMM Algorithms. RR=Replication Reuse, LKF=Local Kernel Fusion, NE=No Kernel Elision.

Algorithm	Best Replication Factor
1.5D Dense Shift, No Elision	$\sqrt{p}$
1.5D Dense Shift, Replication Reuse	$\sqrt{2p}$
1.5D Dense Shift, Local Kernel Fusion	$\sqrt{p/2}$
1.5D Sparse Shift, Replication Reuse	$\sqrt{6p\phi}$
2.5D Dense Replicate, No Elision	$\sqrt[3]{p\frac{(1+3\phi)^2}{4}}$
2.5D Dense Replicate, Replication Reuse	$\sqrt[3]{p(1+3\phi)^2}$
2.5D Sparse Replicate, No Elision	$\sqrt[3]{\frac{p}{(2\phi/3)^2}}$

Table 6.4: Optimal replication factors for FusedMM algorithms.

6.6 Experiments

We ran experiments on Cori, a Cray XC40 system at Lawrence Berkeley National Laboratory, on 256 Xeon Phi Knights Landing (KNL) CPU nodes. Each KNL node is single socket CPU containing 68 cores running at 1.4 GHz with access to 96 GiB of RAM [Cori]. KNL nodes communicate through an Aries interconnect with a Dragonfly topology.

Our implementation employs a hybrid OpenMP / MPI programming model, with a single MPI rank and 68 OpenMP threads per node. We use the `MPI_ISend` and `MPI_Irecv` primitives for point-to-point communication, as well as the blocking collectives `MPI_Reduce_scatter` and `MPI_Allgather`. To load balance among the processors, we randomly permute the rows and columns of sparse matrices that we read in.

We use the Intel Math Kernel Library (MKL version 18.0.1.163) to perform local SpMM computations. Because the MKL sparse BLAS does not yet include an SDDMM function, we wrote a simple implementation that uses OpenMP to parallelize the collection of independent dot products required in the computation. We rely on CombBLAS [Aza+21] for sparse matrix IO and to generate distributed Erdős-Rényi random sparse matrices. We use Eigen [GJ+10] as a wrapper around matrix buffers to handle local dense linear algebra. Our code is available online[1].

6.6.1 Baseline Comparisons

Our work presents, to the best of our knowledge, the first distributed-memory implementation of SDDMM for general matrices. There is no comparable library to establish a baseline. Among the PETSc, Trilinos, and libSRB libraries, only PETSc offers a distributed-memory SpMM implementation as a special case of the `MatMatMult` routine [Bal+21], which we compare against. Since PETSc does not support hybrid OpenMP / MPI parallelism [PETScThr], we ran benchmarks with 68 MPI ranks per node (1 per core). To ensure a fair benchmark

[1] https://github.com/PASSIONLab/distributed_sddmm

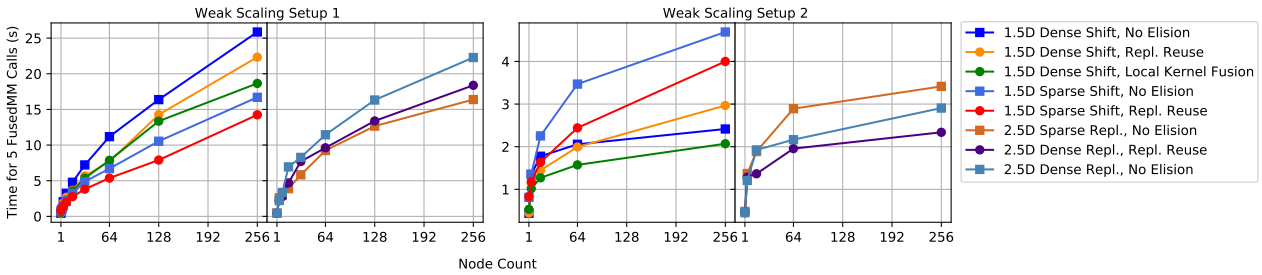


Figure 6.4: Weak scaling experiments with $r = 256$. The horizontal axis gives the node count p for each experiment. In setup 1, p processors run FusedMM on S of side-length $2^{16}p$ with 32 nonzeros per row. In setup 2, p processors execute FusedMM on S of side length $2^{16}p^{1/2}$ with $32p^{1/2}$ nonzeros per row.

against FusedMM algorithms that make a call to both SDDMM and SpMM, we compare our algorithms against two back-to-back SpMM calls from the PETSc library. Since SDDMM and SpMM have identical FLOP counts and communication requirements, using two back-to-back SpMM calls offers a reasonable performance surrogate for FusedMM.

Figure 6.8 compares the strong scaling performance of our algorithms to Cray PETSc (v3.13.3.0, 64-bit). The library requires a 1D block row distribution for all matrices and does not perform any replication, resulting in poor communication scaling. Due to exceptionally poor performance from PETSc on sparse matrices with many nonzeros, we omit the baseline benchmark on the two larger strong scaling workloads.

6.6.2 Weak Scaling on Erdős-Rényi Random Matrices

To benchmark the weak scaling of our algorithms, we keep the FLOP count assigned to each node constant (assuming a load-balanced sparse matrix) while increasing both the processor count and the "problem size". We investigated two methods of doubling the problem size, each giving different performance characteristics.

Setup 1: In these experiments, node counts double from experiment to experiment as we double the side-length of the sparse matrix, keeping the number of nonzeros in each row and the embedding dimension r constant. We begin with sparse matrix dimensions $65,536 \times 65,536$ on a single node with 32 nonzeros per row, and we use $r = 256$ as the embedding dimension. We scale to 256 nodes that collectively process a $2^{24} \times 2^{24}$ matrix with 500 million nonzeros.

While FLOPs per processor remains constant from experiment to experiment, communication for our 1.5D algorithms scales as $O(n/\sqrt{p})$. Doubling both n and p results in an expected increase in communication time of $\sqrt{2}$, giving a projected $\sqrt{p}$-scaling in communication

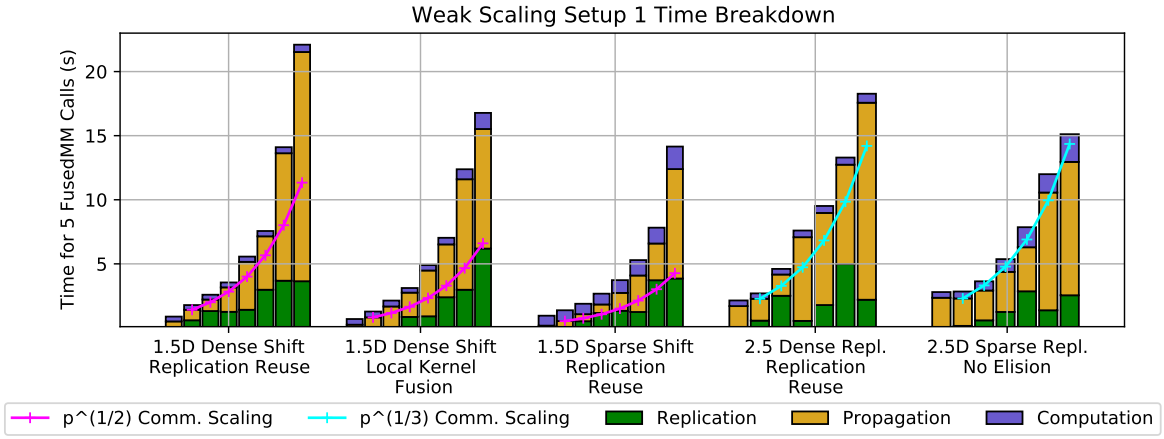


Figure 6.5: Weak scaling time breakdown, setup 1. Each section of bars indicates the time breakdown for successively doubling processor counts, from $p = 2$ on the left of each subsection to $p = 256$ on the right. We expect $p^{1/2}$-communication scaling for 1.5D algorithms and $p^{1/3}$ for 2.5D algorithms.

time with processor count. Similarly, our 2.5D algorithms have communication scaling of $O(n/\sqrt[3]{p^2})$, yielding a projected $\sqrt[3]{p}$ scaling in communication time with processor count. Notice also under this strategy that ϕ remains constant at $32/256 = 1/8$, while the percentage of nonzeros in S decays exponentially. Figure 6.4 (left) gives the results of these experiments, for the best observed replication factor at each processor count. Even though each processor handles the same number of nonzeros across experiments, the communication time (detailed in figure 6.5) quickly dominates the computation time as we double the processor count. Among 1.5D algorithms, the sparse shifting, dense replicating algorithm exhibits the best overall performance. We attribute this to the low, constant value of $\phi = \text{nnz}(\boldsymbol{S})/nr$ across the experiments. At 256 nodes, replication reuse allows the replication reusing sparse shifting 1.5D algorithm to run 1.15 times faster than the 2.5D sparse replicating algorithm, while the variant without communication elision runs 2% slower than the 2.5D sparse replicating algorithm. For 1.5D dense shifting algorithms, both FusedMM elision strategies have roughly the same gain over the unoptimized back-to-back kernel sequence until the 256 node experiment. At 256 nodes, the 1.5D algorithm with local kernel fusion exhibits 1.38x speedup over its non-eliding counterpart, and replication reuse gives 1.16x speedup.

Setup 2: Beginning with the same conditions for a single node as setup 1, node counts quadruple from experiment to experiment, as we both double the side-length of the sparse matrix and double its nonzero count per row. The FLOPs per processor again remains constant. Under this setup, however, the percentage of nonzeros of $\boldsymbol{S}$ remains constant while $\text{nnz}(\boldsymbol{S})/nr$ doubles from experiment to experiment. Setup 2 provides insight into the scaling of the 1.5D sparse shifting algorithm as the ratio ϕ successively doubles. Since the communication cost for 1.5D dense shifting algorithms does not depend on ϕ and scales as $O(n/\sqrt{p})$, its communication cost should remain constant across experiments, while

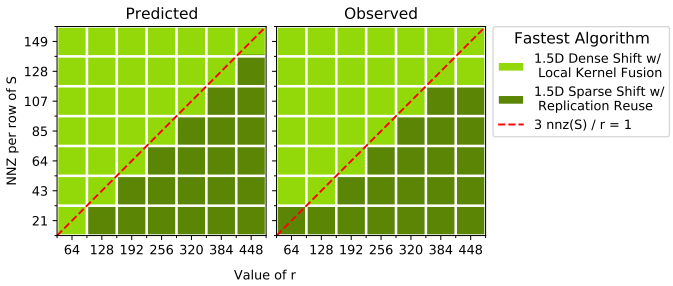


Figure 6.6: Predicted and observed optimal algorithms for $p = 32, m = 2^{22}$. 2.5D algorithms were also benchmarked, but were neither predicted nor observed as best. The best observed replication factor in each configuration was used.

the $O(n/\sqrt[3]{p^2})$ communication time scaling for 2.5D algorithms even implies a decrease in communication time. In practice, the decrease in node locality caused by scaling to high node counts renders a decrease in overall time unlikely.

Figure 6.4 (right half) shows the results, all of which take less than five seconds due to better communication scaling compared to setup 1. Inverting the results from the first weak scaling benchmark, the 1.5D sparse shifting algorithm performs progressively worse as node count increases compared to the best performer, the 1.5D dense shifting algorithm under local kernel fusion. At 256 nodes, the 1.5D local kernel fusion algorithm is 1.94 times as fast as the sparse shifting algorithm with replication reuse. For almost all cases, employing replication reuse or local kernel fusion results in nontrivial performance gains over an unoptimized sequence of SDDMM and SpMM.

6.6.3 Effect of Embedding Width r

Compiled from 740 trials over different configurations, figure 6.6 shows the best algorithm out of the four that employ communication elision (along with the 2.5D sparse replicating algorithm) on a range of r-values and sparse matrix nonzero counts. As predicted, the 1.5D dense shifting algorithm performs better when $\phi = \text{nnz(S)}/(nr)$ is high, while the 1.5D sparse shifting algorithm performs better for low ϕ. Both 1.5D algorithms outperform the 2.5D algorithms in theory as well as practice by margins comparable to those in figure 6.4. From figure 6.6, we note that the optimal algorithm choice is always a 1.5D sparse shifting or dense shifting algorithm depending on the value of ϕ, a conclusion that carries some caveats. Specifically, the performance of the 2.5D algorithms is hurt at $p = 32$ since the replication factor is constrained to be either 2 or 8. Our strong scaling experiments indicate that at high enough node counts, 2.5D algorithms approach (and sometimes outperform) the 1.5D algorithms.

For weak scaling setup 1, figure 6.7 gives the predicted and observed optimal replication factor as a function of the processor count for 1.5D dense shifting algorithms. As predicted,

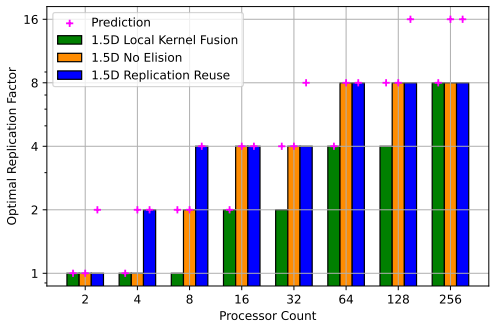


Figure 6.7: Predicted vs. observed optimal replication factors for weak scaling experiments.

the optimal replication factor c for the 1.5D algorithm with replication reuse is at least the optimal replication factor for the unfused algorithm. The latter, in turn, is at least the optimal replication factor of the local kernel fusion algorithm. The plot experimentally confirms that our fused algorithms save communication by changing the optimal replication factor in addition to decreasing the number of communication rounds.

Our predictions of optimal replication factor match the observed optimal values on most experiments. When they disagree, our theory tends to overestimate for two reasons: first, we did not test replication factors higher than 8 for our weak scaling experiments due to memory constraints, leading to the gap at the right end of the figure. Second, we used an ordering of MPI ranks that maximized locality within each layer of the processor grid. As a result, communication within the fiber axis (i.e. replication costs) are more expensive due to lack of node locality, a fact that we verified by comparing against an MPI rank order that optimized for locality along each fiber.

6.6.4 Strong Scaling on Real-World Matrices

We conduct strong scaling experiments on up to 256 KNL nodes with $r = 128$ on five real-world matrices given in table 6.5 containing up to ≈ 1.5 billion nonzeros. amazon-large.mtx, uk-2002.mtx, arabic-2005.mtx, and twitter7.mtx were taken from the Suitesparse matrix collection [DH11], while eukarya.mtx contains protein sequence alignment information for eukaryotic genomes [Aza+18]. With ≈ 360 million nonzeros but only ≈ 3 million vertices, eukarya is the most dense at 111 nonzeros per row, compared with roughly 16 nonzeros per row for both uk-2002 and amazon-large. Twitter7 and Arabic-2005 fall between the two extremes at 28-35 nonzeros per row, but have significantly more total nonzeros compared to the other matrices. We expect that the 1.5D sparse shifting and 2.5D sparse replicating algorithms will exhibit better communication performance on the sparse Amazon matrix, while the 1.5D dense shifting and 2.5D dense replicating algorithms become optimal for eukarya and and twitter7. Because we choose $r = 128$ conservatively to avoid allocating large amounts of RAM at small node counts, we enforce a minimum replication factor of 2 for the

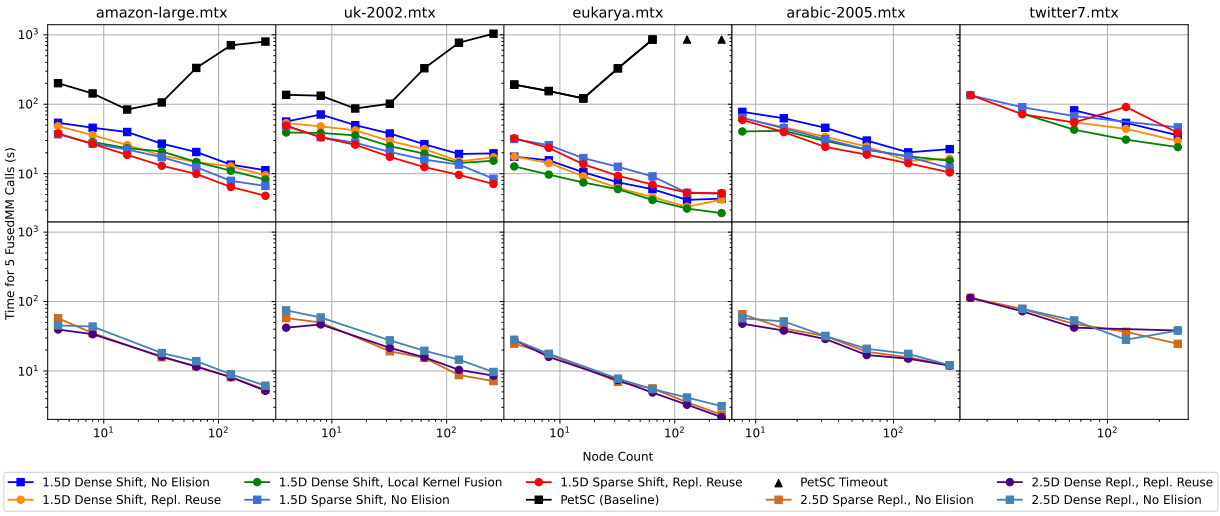


Figure 6.8: Strong scaling experiments, $r = 128$. We benchmarked our algorithms on 5 FusedMM calls and the baseline implementation PETSc with 10 SpMM calls. Black triangles indicate that PETSc took longer than three hours to complete the trial. We benchmarked with a minimum of 4 nodes for the smaller workloads, 8 nodes for Arabic-2005, and 16 nodes for twitter7.

Matrix	Rows	Columns	Nonzeros
amazon-large.mtx	14,249,639	14,249,639	230,788,269
uk-2002.mtx	18,484,117	18,484,117	298,113,762
eukarya.mtx	3,243,106	3,243,106	359,744,161
arabic-2005.mtx	22,744,080	22,744,080	639,999,458
twitter7.mtx	41,652,230	41,652,230	1,468,365,182

Table 6.5: Matrices in strong scaling experiments.

1.5D sparse shifting algorithm at 256 nodes (since we cannot divide 128 into more than 256 parts for $c = 1$).

Figure 6.8 gives the results of the strong scaling experiments; the performance of each algorithm is its best runtime over replication factors from 1 through 16. As we expect, the 1.5D sparse shifting algorithm with replication reuse performs best on the amazon-large and uk-2002 matrices, while it is the second worst on eukarya. With 256 nodes on uk-2002, the 1.5D sparse shifting algorithm with replication reuse performs 1.19x faster than the version without communication elision, and it performs 2.1x faster than the dense shifting fused algorithm with local kernel fusion. On eukarya with 256 nodes, the dense shifting algorithm with local kernel fusion performs 1.6x faster than the version without communication elision and 1.9x faster than the sparse shifting algorithm. Among 2.5D algorithms, the dense replicating

algorithm with replication reuse and the sparse replicating algorithm have similar performance, and both outperform the 2.5D dense replicating algorithm with no communication elision. As predicted by our theory, the dense replicating algorithm has slightly better performance on eukarya.mtx at high node counts, even outperforming all of our 1.5D algorithms.

6.6.5 Applications

Here, we plug in our distributed memory algorithms to machine learning applications to benchmark their performance. We focus on two: collaborative filtering with alternating least squares, and graph neural networks with self-attention.

Collaborative Filtering with ALS: Collaborative filtering attempts to factor a matrix $\boldsymbol{C} \in \mathbb{R}^{m \times n}$ as $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}^\top$, for $\boldsymbol{A} \in \mathbb{R}^{m \times r}, \boldsymbol{B} \in \mathbb{R}^{n \times r}$; however, we only have access to a set of sparse observations of $\boldsymbol{C}$, denoted as the sparse matrix $\tilde{\boldsymbol{C}}$ with nonzero indicators $\boldsymbol{S}$. We iteratively minimize the loss, which is the Frobenius norm of $\tilde{\boldsymbol{C}} - \text{SDDMM}(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{S})$. The SDDMM kernel in the loss also appears (along with SpMM) in the iterative update equations.

The ALS method alternately keeps $\boldsymbol{A}$ or $\boldsymbol{B}$ fixed and, for each row $\boldsymbol{x}$ of the matrix to optimize, solves a least squares problem of the form $\boldsymbol{M}\boldsymbol{x} = \boldsymbol{b}$. These least squares problems are distinct for each row due to the varying placement of nonzeros within each column of $\boldsymbol{S}$. If we use Conjugate Gradients (CG) as the least squares solver, Canny and Zhao [CZ13] exhibit the technique of batching computation of the query vectors $\boldsymbol{M}\boldsymbol{x}$ for all rows at once using a FusedMM operation.

2.5D sparse replicating and 2.5D dense replicating algorithms suffer slight penalties for this application compared to 1.5D algorithms, as the output distributions of the dense matrices are shifted and transposed, respectively, compared to the input distributions. Since the output query vectors become (after some additional manipulation) inputs to the next CG iteration, 2.5D algorithms must pay to shift the input and output distributions at each step. We benchmark 20 CG iterations with our distributed algorithms embedded: 10 to optimize the matrix $\boldsymbol{A}$, and 10 to optimize the matrix $\boldsymbol{B}$.

Graph Attention Network (Forward Pass Workload): Consider a graph with adjacency matrix $\boldsymbol{S} \in \{0, 1\}^{n \times n}$. Conventional GNNs contain a series of layers, with an r-length vector of features associated with each node as the input to a layer. These node embeddings are held in an $n \times r$ matrix $\boldsymbol{A}$. The GNN layer applies a small linear transformation $\boldsymbol{W} \in \mathbb{R}^{r \times r'}$ to the embedding matrix before performing a convolution to sum the feature vectors at neighbors of any node x. The sum is the new feature vector of x. Application of a nonlinear activation σ typically follows the convolution, giving the final layer output $A' = \sigma(\boldsymbol{S}\boldsymbol{A}\boldsymbol{W})$.

The subsequent GNN layer takes $\boldsymbol{A}'$ as an input, with the final layer generating a node-level or graph-level prediction. Graph attention networks (GATs) modify this architecture by

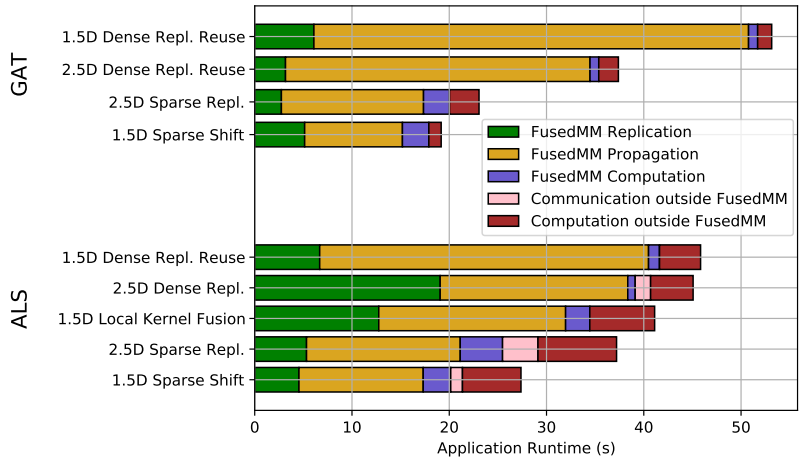


Figure 6.9: Alternating least squares and graph attention network forward pass performance on amazon.mtx. We use 256 nodes and $r = 128$. The 1.5D algorithm with local kernel fusion was not benchmarked for GATs, as it is incompatible with softmax regularization of learned edge weights.

weighting edges with a self-attention score computed using the embeddings of the incident nodes. A single self-attention head [Vel+18] replaces $\boldsymbol{S}$ with $\boldsymbol{S}' = \sigma_{\text{LeakyReLU}}(\boldsymbol{S} \circledast (\boldsymbol{A} \cdot_{GAT} \boldsymbol{A}^\top))$. Here, $\boldsymbol{A} \cdot_{GAT} \boldsymbol{A}^\top$ is an $n \times n$ matrix with entries given by $(\boldsymbol{A} \cdot_{GAT} \boldsymbol{A}^\top)_{ij} = \boldsymbol{a}^\top(\boldsymbol{A}_{i:} || \boldsymbol{A}_{j:})$, where $||$ denotes concatenation and $\boldsymbol{a}^\top$ is a trainable vector. The computation of $\boldsymbol{S}'$ involves a slight modification of Eq. 6.1 and has an identical communication pattern to SDDMM. A multi-head GAT concatenates the outputs of several attention heads with distinct trainable weight matrices $\boldsymbol{W}$ and weight vectors $\boldsymbol{a}$. We simulate the forward pass workload of this multi-head GAT architecture using random weight matrices to focus on the communication reduction and scaling of our distributed-memory algorithms.

Figure 6.9 shows the time breakdown of our applications, both inside the FusedMM kernels and in the rest of the application. The ALS application exhibits some variation in communication and computation time spent outside FusedMM. The variation is only partially due to additional processor to processor communication to compute distributed dot products, which is higher for the sparse replicating and shifting algorithms. More significantly, at the high replication factors (8 and 16) used in the experiment, the row-major local dense matrices are extremely tall-skinny for the 1.5D sparse shifting and 2.5D sparse replicating algorithms compared to the other variants. Careful analysis of the CG solver revealed that the dense batch dot product operation requires a long sequence of poorly performing dot product calls on short vectors. Since the sequence grows linearly with local matrix height, hundreds of thousands of these calls slow performance for the 1.5D sparse shifting and 2.5D sparse replicating algorithms. Calling an optimized batched BLAS library would fix this issue, which we leave as future work.

6.7 Conclusions and Further Work

We gave a theoretical communication analysis of distributed memory sparsity agnostic approaches for SDDMM and FusedMM. Our theory predicted communication savings using two distinct approaches to combine the two kernels, and we observed those benefits in our strong and weak scaling experiments. In both theory and practice, 1.5D sparse shifting and 2.5D sparse replicating algorithms perform better when $\boldsymbol{S}$ has far fewer nonzeros compared to either dense matrix. When $\boldsymbol{S}$ has a higher nonzero count, dense shifting 1.5D and dense replicating 2.5D algorithms win out.

Further performance improvement may be possible by overlapping communication in the propagation phase of any of our algorithms with local computation. Such an implementation might require one-sided MPI or a similar protocol for remote direct memory access (RDMA) without CPU involvement.

Chapter 7

Recent Advances and Open Problems

We conclude by discussing recent progress on the problems explored by this dissertation. Research on graph / tensor problems is prolific, and we only cover a small tranche of an exciting body of work.

7.1 Advances in Chemistry Foundation Models

Since the release of OpenEquivariance, the MACE [Bat+24] and Nequip [Bat+22b] models officially introduced support for our accelerated kernels. Lee et al. [Lee+25] also released the FlashTP accelerated kernel package, creating a robust marketplace of equivariant model backends. Both FlashTP and OpenEquivariance provide an interface identical to e3nn [Gei+22], facilitating integration into new models.

Tan et al. [Tan+25] also explored Triton to accelerate tensor products in the Allegro [Mus+23] model. They composed custom kernels with PyTorch model compilation tools (e.g., ahead-of-time-inductor) to reduce the significant overhead of eager execution. Our kernel package composes with these tools as well, although we do not offer the specific kernels that accelerate Allegro.

Lin et al. [Lin+25] recently explored a fascinating crossover between tensor decomposition and equivariant deep neural networks. They replace the sparse tensor of Clebsch-Gordon coefficients from Chapter 2 with a PARAFAC approximation, boosting computational efficiency without sacrificing rotational equivariance. The work highlights the versatility of CP decomposition and the importance of fast tensor kernels to create and evaluate such structures.

Foundation models for chemistry offer tantalizing possibilities for materials discovery. Their capabilities are likely to improve with the recent release of several large training datasets for

molecular property prediction [Cha+21; Tra+23; Lev+25]. Scientists are actively debating the optimal architecture for such models, including the importance of a conservative force field [BLC25] and the expressive capability of equivariant features [LGM24a]. Our work in Chapter 2 empowers researchers to answer these questions while making full use of the latest hardware.

7.2 Open Problems Related to Leverage Scores

Ghadiri et al. [Gha+25] recently adapted our tensor sketching methods for the tensor *completion* problem, analogous to the matrix completion methods from Chapter 6. Along a related line, Hayashi et al. [Hay+25] investigate leverage score sampling to accelerate nonnegative matrix factorization. Compared to tensor factorization, leverage score sampling for matrix completion is more expensive relative to other stages of the alternating solver. Hayashi et al. [Hay+25] still find advantages to the technique and use a hybrid deterministic / random sampling procedure similar to one proposed by Larsen and Kolda [LK22].

Despite the strong theoretical guarantees offered by leverage score sampling, there are convincing arguments that the method is uncompetitive in many cases compared to oblivious sketches (see Martinsson and Tropp [MT20], Section 9.6.4). Their point is well-taken: computing exact leverage scores for a general matrix is expensive, while leverage approximation relies on intermediate oblivious sketches that could be applied directly to the original problem. We argue, however, that the exploitable structure in the Khatri-Rao product renders it uniquely well-suited for leverage score sampling.

An oblivious sketch can be constructed without prior knowledge of the target matrix. As far as we are aware, there is a theoretical gap between the best oblivious sketches (that have reasonable runtime) and leverage score samplers for the Khatri-Rao product. Omitting other relevant variables, the oblivious tree sketch by Ahle et al. [Ahl+20] requires $O(R^2)$ samples to satisfy the subspace embedding property. By contrast, the non-oblivious leverage score sketch by Woodruff and Zandieh [WZ22] achieves the optimal $\Theta(R)$ target row count for the same residual threshold.

Are there faster oblivious sketching algorithms for Khatri-Rao products that preserve column space geometry, require only $O(R)$ rows in the target matrix, and run in nearly input-sparsity time? Such an algorithm would provide a faster alternative to TensorSketch [PP13]. Refuting the existence of such an algorithm, on the other hand, requires a lower bound proof involving both sketch computational complexity and the sketch embedding dimension. To the best of our knowledge, the existence (or disproof thereof) of a faster oblivious sketch for Khatri-Rao products remains an open problem.

7.3 Sparsity-Aware SpMM and SDDMM

Our 1.5D and 2.5D distributed sparse matrix kernels are similar to those explored by Koanantakool et al. [Koa+16]. These kernels move contiguous chunks of the dense and sparse matrices from processor to processor, relying on random permutations for nonzero load balance. We call such methods "sparsity-oblivious". Sparsity-aware reordering strategies, such as graph partitioning, [Kar11], can drastically reduce communication, but they are costly to run. In addition, sparsity-aware distributed kernels (which pack and send only the required entries for sparse computation) are more complex to program and model analytically.

Nevertheless, recent works show that sparsity-aware methods for SpMM and SDDMM are worth these trade-offs. Block et al. [Blo+24] hybridize sparsity-aware and sparsity-oblivious methods to reduce communication in sparse matrices with non-uniform nonzero density. In a similar vein, Gianinazzi et al. [Gia+24] rearrange sparse matrix entries into a block diagonal form, with two small vertical and horizontal strips containing nonzero entries that span the entire matrix. The pattern resembles an arrow. The enhanced locality along the block diagonal reduces communication, while the "arrowheads" relax the overly restrictive conditions of true graph partitioning. Abubaker and Hoefler [AH24] also explore sparsity-aware 3D algorithms for SpMM and SDDMM, exhibiting impressive speedups over our own package. In addition, Mukhopadhyay et al. [Muk+24] demonstrated advantages to sparsity-aware SpMM in distributed graph neural networks.

Bibliography

[AC09] Nir Ailon and Bernard Chazelle. "The fast Johnson–Lindenstrauss transform and approximate nearest neighbors". In: *SIAM Journal on computing* 39.1 (2009), pp. 302–322.

[ACS90] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. "Communication complexity of PRAMs". In: *Theoretical Computer Science* 71.1 (1990), pp. 3–28. ISSN: 0304-3975. DOI: https://doi.org/10.1016/0304-3975(90)90188-N.

[Aga+95] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. "A three-dimensional approach to parallel matrix multiplication". In: *IBM Journal of Research and Development* 39.5 (1995), pp. 575–582. DOI: 10.1147/rd.395.0575.

[AH24] Nabil Abubaker and Torsten Hoefler. *SpComm3D: A Framework for Enabling Sparse Communication in 3D Sparse Kernels*. 2024. arXiv: 2404.19638 [cs.DC]. URL: https://arxiv.org/abs/2404.19638.

[AHK19] Brandon Anderson, Truong-Son Hy, and Risi Kondor. "Cormorant: covariant molecular neural networks". In: *Proceedings of the 33rd International Conference on Neural Information Processing Systems*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[Ahl+20] Thomas D. Ahle, Michael Kapralov, Jakob B. T. Knudsen, Rasmus Pagh, Ameya Velingker, David P. Woodruff, and Amir Zandieh. "Oblivious Sketching of High-Degree Polynomial Kernels". In: *Proceedings of the 2020 ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 2020, pp. 141–160. DOI: 10.1137/1.9781611975994.9.

[Akt+14] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. "Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 1213–1222. DOI: 10.1109/IPDPS.2014.125.

[Al +23] Hussam Al Daas, Grey Ballard, Paul Cazeaux, Eric Hallman, Agnieszka Miedlar, Mirjeta Pasha, Tim W. Reid, and Arvind K. Saibaba. "Randomized Algorithms for Rounding in the Tensor-Train Format". In: *SIAM Journal on Scientific Computing* 45.1 (2023), A74–A95. DOI: 10.1137/21M1451191.

[Alm+25] Josh Alman, Ran Duan, Virginia Vassilevska Williams, Yinzhan Xu, Zixuan Xu, and Renfei Zhou. "More Asymmetry Yields Faster Matrix Multiplication". In: *Proceedings of the 2025 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 2025, pp. 2005–2039. DOI: 10.1137/1.9781611978322.63.

[AMT10] Haim Avron, Petar Maymounkov, and Sivan Toledo. "Blendenpik: Supercharging LAPACK's Least-Squares Solver". In: *SIAM Journal on Scientific Computing* 32.3 (2010), pp. 1217–1236.

[ASA16] Seher Acer, Oguz Selvitopi, and Cevdet Aykanat. "Improving performance of sparse matrix dense matrix multiplication on large-scale parallel systems". en. In: *Parallel Computing*. Theory and Practice of Irregular Applications 59 (Nov. 2016), pp. 71–96. ISSN: 0167-8191. DOI: 10.1016/j.parco.2016.10.001. (Visited on 10/13/2021).

[Aza+18] Ariful Azad, Georgios A Pavlopoulos, Christos A Ouzounis, Nikos C Kyrpides, and Aydin Buluç. "HipMCL: a high-performance parallel implementation of the Markov clustering algorithm for large-scale networks". In: *Nucleic Acids Research* 46.6 (Jan. 2018), e33–e33. ISSN: 0305-1048. DOI: 10.1093/nar/gkx1313.

[Aza+21] Ariful Azad, Oguz Selvitopi, Md Taufique Hussain, John Gilbert, and Aydin Buluç. "Combinatorial BLAS 2.0: Scaling combinatorial algorithms on distributed-memory systems". In: *IEEE TPDS* (2021), pp. 1–1. ISSN: 1558-2183. DOI: 10.1109/TPDS.2021.3094091.

[BAH19] Ivana Balazevic, Carl Allen, and Timothy Hospedales. "TuckER: Tensor Factorization for Knowledge Graph Completion". In: *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*. Hong Kong, China: Association for Computational Linguistics, Nov. 2019, pp. 5185–5194. DOI: 10.18653/v1/D19-1522.

[Bal+21] Satish Balay et al. *PETSc/TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.16. Argonne National Laboratory, 2021.

[Bat+22a] Ilyes Batatia, David Peter Kovacs, Gregor N. C. Simm, Christoph Ortner, and Gabor Csanyi. "MACE: Higher Order Equivariant Message Passing Neural Networks for Fast and Accurate Force Fields". In: *Advances in Neural Information Processing Systems*. 2022.

[Bat+22b] Simon Batzner, Albert Musaelian, Lixin Sun, Mario Geiger, Jonathan P. Mailoa, Mordechai Kornbluth, Nicola Molinari, Tess E. Smidt, and Boris Kozinsky. "E(3)-equivariant graph neural networks for data-efficient and accurate interatomic potentials". In: *Nature Communications* 13.1 (May 2022), p. 2453. ISSN: 2041-1723. DOI: 10.1038/s41467-022-29939-5.

[Bat+24] Ilyes Batatia et al. *A foundation model for atomistic materials chemistry*. 2024. arXiv: 2401.00096 [physics.chem-ph]. URL: https://arxiv.org/abs/2401.00096.

[BBD22] V. Bharadwaj, A. Buluc, and J. Demmel. "Distributed-Memory Sparse Kernels for Machine Learning". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, June 2022, pp. 47–58. DOI: 10.1109/IPDPS53621.2022.00014.

[BBK18] Casey Battaglino, Grey Ballard, and Tamara G. Kolda. "A Practical Randomized CP Tensor Decomposition". In: *SIAM Journal on Matrix Analysis and Applications* 39.2 (2018), pp. 876–901. DOI: 10.1137/17M1112303.

[Bha+23] Vivek Bharadwaj, Osman Asif Malik, Riley Murray, Laura Grigori, Aydin Buluc, and James Demmel. "Fast Exact Leverage Score Sampling from Khatri-Rao Products with Applications to Tensor Decomposition". In: *Advances in Neural Information Processing Systems*. Ed. by A. Oh, T. Neumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine. Vol. 36. Curran Associates, Inc., 2023, pp. 47874–47901.

[Bha+24a] Vivek Bharadwaj, Osman Asif Malik, Riley Murray, Laura Grigori, Aydın Buluç, and James Demmel. "Distributed-Memory Randomized Algorithms for Sparse Tensor CP Decomposition". In: *Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures*. SPAA '24. Nantes, France: Association for Computing Machinery, 2024.

[Bha+24b] Vivek Bharadwaj, Beheshteh T. Rakhshan, Osman Asif Malik, and Guillaume Rabusseau. "Efficient Leverage Score Sampling for Tensor Train Decomposition". In: *Advances in Neural Information Processing Systems*. Vol. 37. Curran Associates, Inc., 2024, pp. 73726–73744.

[Bha+25] Vivek Bharadwaj, Austin Glover, Aydın Buluç, and James Demmel. "An Efficient Sparse Kernel Generator for O(3)-Equivariant Deep Networks". In: *Proceedings of the 3rd SIAM Conference on Applied and Discrete Computational Algorithms*. July 2025.

[BHR18] Grey Ballard, Koby Hayashi, and Kannan Ramakrishnan. "Parallel nonnegative CP decomposition of dense tensors". In: *2018 IEEE 25th International Conference on High Performance Computing (HiPC)*. IEEE, 2018, pp. 22–31.

[Big+23] Filippo Bigi, Guillaume Fraux, Nicholas J. Browning, and Michele Ceriotti. "Fast evaluation of spherical harmonics with sphericart". In: *J. Chem. Phys.* 159 (6 2023), p. 064802.

[BK08] Brett W. Bader and Tamara G. Kolda. "Efficient MATLAB Computations with Sparse and Factored Tensors". In: *SIAM Journal on Scientific Computing* 30.1 (2008), pp. 205–231. DOI: 10.1137/060676489.

[BK25] Grey Ballard and Tamara G. Kolda. *Tensor Decompositions for Data Science*. eng. Cambridge University Press, 2025. ISBN: 9781009471671.

[BKR18] Grey Ballard, Nicholas Knight, and Kathryn Rouse. "Communication Lower Bounds for Matricized Tensor Times Khatri-Rao Product". In: *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2018, pp. 557–567. DOI: 10.1109/IPDPS.2018.00065.

[BLC25] Filippo Bigi, Marcel F. Langer, and Michele Ceriotti. "The dark side of the forces: assessing non-conservative force models for atomistic machine learning". In: *Forty-second International Conference on Machine Learning*. 2025.

[Blo+24] Charles Block, Gerasimos Gerogiannis, Charith Mendis, Ariful Azad, and Josep Torrellas. "Two-Face: Combining Collective and One-Sided Communication for Efficient Distributed SpMM". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 1200–1217. DOI: 10.1145/3620665.3640427.

[Can69] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". AAI7010025. PhD thesis. USA, 1969.

[Cha+07] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. "Collective communication: theory, practice, and experience: Research Articles". In: *Concurrency and Computation: Practice & Experience* 19.13 (Sept. 2007), pp. 1749–1783. ISSN: 1532-0626.

[Cha+21] Lowik Chanussot et al. "Open Catalyst 2020 (OC20) Dataset and Community Challenges". In: *ACS Catalysis* 11.10 (2021), pp. 6059–6072. DOI: 10.1021/acscatal.0c04525.

[Che+16] Dehua Cheng, Richard Peng, Yan Liu, and Ioakeim Perros. "SPALS: Fast Alternating Least Squares via Implicit Leverage Scores Sampling". In: *Advances in Neural Information Processing Systems*. Vol. 29. Curran Associates, Inc., 2016. (Visited on 01/06/2023).

[Che+18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "TVM: an automated end-to-end optimizing compiler for deep learning". In: *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 9781931971478.

[Che+20a] Ke Chen, Qin Li, Kit Newton, and Stephen J. Wright. "Structured Random Sketching for PDE Inverse Problems". In: *SIAM Journal on Matrix Analysis and Applications* 41.4 (2020), pp. 1742–1770. DOI: 10.1137/20M1310497.

[Che+20b] Yiran Chen, Yuan Xie, Linghao Song, Fan Chen, and Tianqi Tang. "A Survey of Accelerator Architectures for Deep Neural Networks". In: *Engineering* 6.3 (2020), pp. 264–274. ISSN: 2095-8099. DOI: https://doi.org/10.1016/j.eng.2020.01.007.

[Che+25] Yifan Chen, Ethan N. Epperly, Joel A. Tropp, and Robert J. Webber. "Randomly pivoted Cholesky: Practical approximation of a kernel matrix with few entry evaluations". In: *Communications on Pure and Applied Mathematics* 78.5 (2025), pp. 995–1041. DOI: https://doi.org/10.1002/cpa.22234.

[Cor+23] Gabriele Corso, Hannes Stärk, Bowen Jing, Regina Barzilay, and Tommi S. Jaakkola. "DiffDock: Diffusion Steps, Twists, and Turns for Molecular Docking". In: *The Eleventh International Conference on Learning Representations*. 2023.

[Cor20] NVIDIA Corporation. *NVIDIA A100 Tensor Core GPU Architecture*. Tech. rep. NVIDIA, 2020. URL: https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[Cori] *Cori - NERSC Documentation*. URL: https://docs.nersc.gov/systems/cori/ (visited on 10/14/2021).

[CV14] Joon Hee Choi and S. Vishwanathan. "DFacTo: Distributed Factorization of Tensors". In: *Advances in Neural Information Processing Systems*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger. Vol. 27. Curran Associates, Inc., 2014.

[CZ13] John Canny and Huasha Zhao. "Big Data Analytics with Small Footprint: Squaring the Cloud". en. In: *KDD*. 2013.

[Dao+22] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. "FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness". In: *Proceedings of the 36th International Conference on Neural Information Processing Systems*. NIPS '22. New Orleans, LA, USA: Curran Associates Inc., 2022. ISBN: 9781713871088.

[Dav19] Timothy A. Davis. "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* 45.4 (Dec. 2019). ISSN: 0098-3500. DOI: 10.1145/3322125.

[Dee+25] Ewa Deelman, Jack Dongarra, Bruce Hendrickson, Amanda Randles, Daniel Reed, Edward Seidel, and Katherine Yelick. "High-performance computing at a crossroads". In: *Science* 387.6736 (2025), pp. 829–831. DOI: 10.1126/science.adu0801.

[Dem13] James Demmel. "Communication-avoiding algorithms for linear algebra and beyond". In: *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 2013, pp. 585–585. DOI: 10.1109/IPDPS.2013.123.

[Dem97] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997. DOI: 10.1137/1.9781611971446.

[DH11] Timothy A. Davis and Yifan Hu. "The University of Florida Sparse Matrix Collection". In: *ACM Trans. Math. Softw.* 38.1 (Dec. 2011). ISSN: 0098-3500. DOI: 10.1145/2049662.2049663.

[Dia+18] Huaian Diao, Zhao Song, Wen Sun, and David Woodruff. "Sketching for kronecker product regression and p-splines". In: *International Conference on Artificial Intelligence and Statistics*. PMLR. 2018, pp. 1299–1308.

[Dia+19] Huaian Diao, Rajesh Jayaram, Zhao Song, Wen Sun, and David Woodruff. "Optimal Sketching for Kronecker Product Regression and Low Rank Approximation". In: *Advances in Neural Information Processing Systems*. Vol. 32. Curran Associates, Inc., 2019.

[DKM06] Petros Drineas, Ravi Kannan, and Michael W. Mahoney. "Fast Monte Carlo Algorithms for Matrices I: Approximating Matrix Multiplication". In: *SIAM Journal on Computing* 36.1 (2006), pp. 132–157. DOI: 10.1137/S0097539704442684.

[DM16] Petros Drineas and Michael W. Mahoney. "RandNLA: Randomized Numerical Linear Algebra". In: *Commun. ACM* 59.6 (May 2016), pp. 80–90. ISSN: 0001-0782. DOI: 10.1145/2842602.

[DM21] Michał Dereziński and Michael W. Mahoney. "Determinantal point processes in randomized numerical linear algebra". In: *Notices of the American Mathematical Society* 68.1 (2021), pp. 34–45.

[Don+03] Jack Dongarra, Ian Foster, Geoffrey Fox, William Gropp, Ken Kennedy, Linda Torczon, and Andy White, eds. *Sourcebook of parallel computing*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003. ISBN: 1558608710.

[Dri+12] Petros Drineas, Malik Magdon-Ismail, Michael W. Mahoney, and David P. Woodruff. "Fast Approximation of Matrix Coherence and Statistical Leverage". In: *J. Mach. Learn. Res.* 13.1 (Dec. 2012), pp. 3475–3506. ISSN: 1532-4435.

[FFG22] Matthew Fahrbach, Gang Fu, and Mehrdad Ghadiri. "Subquadratic Kronecker Regression with Applications to Tensor Decomposition". In: *Advances in Neural Information Processing Systems*. Vol. 35. Curran Associates, Inc., 2022, pp. 28776–28789.

[Fuc+20] Fabian Fuchs, Daniel Worrall, Volker Fischer, and Max Welling. "SE(3)-Transformers: 3D Roto-Translation Equivariant Attention Networks". In: *Advances in Neural Information Processing Systems*. Ed. by H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin. Vol. 33. Curran Associates, Inc., 2020, pp. 1970–1981.

[GAY20] Alex Gittens, Kareem Aggour, and Bülent Yener. "Adaptive Sketching for Fast and Convergent Canonical Polyadic Decomposition". In: *Proceedings of the 37th International Conference on Machine Learning*. Vol. 119. Proceedings of Machine Learning Research. PMLR, July 2020, pp. 3566–3575.

[Gei+22] Mario Geiger et al. *Euclidean neural networks: e3nn*. Version 0.5.0. Atomic Architects (Smidt Research Group), Apr. 2022. DOI: 10.5281/zenodo.6459381. URL: https://github.com/e3nn/e3nn/releases/tag/0.5.0.

[Gei+24] Mario Geiger, Emine Kucukbenli, Becca Zandstein, and Kyle Tretina. *Accelerate Drug and Material Discovery with New Math Library NVIDIA cuEquivariance*. NVIDIA Technical Blog. Nov. 2024. URL: https://developer.nvidia.com/blog/accelerate-drug-and-material-discovery-with-new-math-library-nvidia-cuequivariance/ (visited on 12/02/2024).

[Gha+25] Mehrdad Ghadiri, Matthew Fahrbach, Yunbum Kook, and Ali Jadbabaie. "Fast Tensor Completion via Approximate Richardson Iteration". In: *Forty-second International Conference on Machine Learning*. 2025.

[Gia+24] Lukas Gianinazzi, Alexandros Nikolaos Ziogas, Langwen Huang, Piotr Luczynski, Saleh Ashkboosh, Florian Scheidl, Armon Carigiet, Chio Ge, Nabil Abubaker, Maciej Besta, Tal Ben-Nun, and Torsten Hoefler. "Arrow Matrix Decomposition: A Novel Approach for Communication-Efficient Sparse Matrix Multiplication". In: *Proceedings of the 29th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*. PPoPP '24. Edinburgh, United Kingdom: Association for Computing Machinery, 2024, pp. 404–416. DOI: 10.1145/3627535.3638496.

[GJ+10] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. 2010. URL: http://eigen.tuxfamily.org.

[GKT13] Lars Grasedyck, Daniel Kressner, and Christine Tobler. "A literature survey of low-rank tensor approximation techniques". In: *GAMM-Mitteilungen* 36.1 (2013), pp. 53–78. DOI: https://doi.org/10.1002/gamm.201310004.

[GS22] Mario Geiger and Tess Smidt. *e3nn: Euclidean Neural Networks*. 2022. DOI: 10.48550/ARXIV.2207.09453. URL: https://arxiv.org/abs/2207.09453.

[GV13] Gene H. Golub and Charles F. Van Loan. *Matrix Computations - 4th Edition*. Philadelphia, PA: Johns Hopkins University Press, 2013. DOI: 10.1137/1.9781421407944.

[GW95] Robert A. van de Geijn and Jerrell Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Tech. rep. USA, 1995.

[Hai+15] Azzam Haidar, Tingxing Dong, Stanimire Tomov, Piotr Luszczek, and Jack Dongarra. "Framework for Batched and GPU-resident Factorization Algorithms to Block Householder Transformations". In: *ISC High Performance*. Springer. Frankfurt, Germany: Springer, July 2015.

[Hay+18] Koby Hayashi, Grey Ballard, Yujie Jiang, and Michael J. Tobia. "Shared-memory parallelization of MTTKRP for dense tensors". In: *SIGPLAN Not.* 53.1 (Feb. 2018), pp. 393–394. ISSN: 0362-1340. DOI: 10.1145/3200691.3178522.

[Hay+25] Koby Hayashi, Sinan G. Aksoy, Grey Ballard, and Haesun Park. "Randomized Algorithms for Symmetric Nonnegative Matrix Factorization". In: *SIAM Journal on Matrix Analysis and Applications* 46.1 (2025), pp. 584–625. DOI: 10.1137/24M1638355.

[Hel+21] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. "ALTO: adaptive linearized storage of sparse tensors". In: *Proceedings of the 35th ACM International Conference on Supercomputing*. ICS '21. Virtual Event, USA: Association for Computing Machinery, 2021, pp. 404–416. ISBN: 9781450383356. DOI: 10.1145/3447818.3461703.

[HK81] Jia-Wei Hong and H. T. Kung. "I/O complexity: The red-blue pebble game". In: *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*. STOC '81. Milwaukee, Wisconsin, USA: Association for Computing Machinery, 1981, pp. 326–333. ISBN: 9781450373920. DOI: 10.1145/800076.802486.

[HKD20] David Hong, Tamara G. Kolda, and Jed A. Duersch. "Generalized Canonical Polyadic Tensor Decomposition". In: *SIAM Review* 62.1 (2020), pp. 133–163. DOI: 10.1137/18M1203626.

[HMT11] N. Halko, P. G. Martinsson, and J. A. Tropp. "Finding Structure with Randomness: Probabilistic Algorithms for Constructing Approximate Matrix Decompositions". In: *SIAM Review* 53.2 (2011), pp. 217–288. DOI: 10.1137/090771806.

[Hon+19] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. "Adaptive sparse tiling for sparse matrix multiplication". In: *PPOPP*. ACM, 2019, pp. 300–314. ISBN: 978-1-4503-6225-2. DOI: 10.1145/3293883.3295712. (Visited on 10/12/2021).

[ITT04] Dror Irony, Sivan Toledo, and Alexander Tiskin. "Communication lower bounds for distributed-memory matrix multiplication". In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2004.03.021.

[JHA20] Peng Jiang, Changwan Hong, and Gagan Agrawal. "A novel data transformation and execution strategy for accelerating sparse matrix multiplication on GPUs". In: *PPOPP*. 2020, pp. 376–388. DOI: 10.1145/3332466.3374546.

[JKW20] Ruhui Jin, Tamara G Kolda, and Rachel Ward. "Faster Johnson–Lindenstrauss transforms via Kronecker products". In: *Information and Inference: A Journal of the IMA* 10.4 (Oct. 2020), pp. 1533–1562. ISSN: 2049-8772. DOI: 10.1093/imaiai/iaaa028.

[JNN13] J.R. Johansson, P.D. Nation, and F. Nori. "QuTiP 2: A Python framework for the dynamics of open quantum systems". In: *Computer Physics Communications* 184.4 (Apr. 2013), pp. 1234–1240. DOI: 10.1016/j.cpc.2012.11.019.

[Joh93] S.Lennart Johnsson. "Minimizing the communication time for matrix multiplication on multiprocessors". In: *Parallel Computing* 19.11 (1993), pp. 1235–1257. ISSN: 0167-8191. DOI: https://doi.org/10.1016/0167-8191(93)90029-K.

[Jum+21] John Jumper et al. "Highly accurate protein structure prediction with AlphaFold". In: *Nature* 596.7873 (Aug. 2021), pp. 583–589. ISSN: 1476-4687. DOI: 10.1038/s41586-021-03819-2.

[Kah65] W. Kahan. "Pracniques: further remarks on reducing truncation errors". In: *Commun. ACM* 8.1 (Jan. 1965), p. 40. ISSN: 0001-0782. DOI: 10.1145/363707.363723.

[Kan+12] U. Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. "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*. KDD '12. New York, NY, USA: Association for Computing Machinery, Aug. 2012, pp. 316–324. ISBN: 978-1-4503-1462-6. DOI: 10.1145/2339530.2339583.

[Kar+21] George Em Karniadakis, Ioannis G. Kevrekidis, Lu Lu, Paris Perdikaris, Sifan Wang, and Liu Yang. "Physics-informed machine learning". In: *Nature Reviews Physics* 3.6 (June 2021), pp. 422–440. ISSN: 2522-5820. DOI: 10.1038/s42254-021-00314-5.

[Kar11] George Karypis. "METIS and ParMETIS". In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 1117–1124. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_500.

[KB09] Tamara G. Kolda and Brett W. Bader. "Tensor Decompositions and Applications". In: *SIAM Review* 51.3 (Aug. 2009). Publisher: Society for Industrial and Applied Mathematics, pp. 455–500. ISSN: 0036-1445. DOI: 10.1137/07070111X. (Visited on 12/24/2022).

[KH20] Tamara G. Kolda and David Hong. "Stochastic Gradients for Large-Scale Tensor Decomposition". en. In: *SIAM Journal on Mathematics of Data Science* 2.4 (Jan. 2020), pp. 1066–1095. ISSN: 2577-0187. DOI: 10.1137/19M1266265. (Visited on 09/15/2022).

[Kjo+17] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The Tensor Algebra Compiler". In: *Proc. ACM Program. Lang.* 1.OOPSLA (Oct. 2017), 77:1–77:29. ISSN: 2475-1421. DOI: 10.1145/3133901.

[KLT18] Risi Kondor, Zhen Lin, and Shubhendu Trivedi. "Clebsch–Gordan nets: a fully Fourier space spherical convolutional neural network". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 10138–10147.

[Koa+16] Penporn Koanantakool, Ariful Azad, Aydin Buluç, Dmitriy Morozov, Sang-Yun Oh, Leonid Oliker, and Katherine Yelick. "Communication-Avoiding Parallel Sparse-Dense Matrix-Matrix Multiplication". In: *IPDPS*. 2016, pp. 842–853. DOI: 10.1109/IPDPS.2016.117.

[Kok+24] Teddy Koker, Keegan Quigley, Eric Taw, Kevin Tibbetts, and Lin Li. "Higher-order equivariant neural networks for charge density prediction in materials". In: *npj Computational Materials* 10.1 (July 2024), p. 161. ISSN: 2057-3960. DOI: 10.1038/s41524-024-01343-1.

[Kok24] Teddy Koker. *e3nn.c*. Version 0.1.0. Nov. 2024. DOI: 10.5281/zenodo.14183951. URL: https://github.com/teddykoker/e3nn.c.

[Kos+19] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. "TensorLy: Tensor Learning in Python". In: *Journal of Machine Learning Research (JMLR)* 20.26 (2019).

[KP07] Hyunsoo Kim and Haesun Park. "Sparse non-negative matrix factorizations via alternating non-negativity-constrained least squares for microarray data analysis". In: *Bioinformatics* 23.12 (May 2007), pp. 1495–1502. ISSN: 1367-4803. DOI: 10.1093/bioinformatics/btm134.

[KR68] C. G. Khatri and C. Radhakrishna Rao. "Solutions to Some Functional Equations and Their Applications to Characterization of Probability Distributions". In: *Sankhyā: The Indian Journal of Statistics, Series A (1961-2002)* 30.2 (1968), pp. 167–180. ISSN: 0581572X. (Visited on 04/16/2025).

[KS23] Raghavendra Kanakagiri and Edgar Solomonik. *Minimum Cost Loop Nests for Contraction of a Sparse Tensor with a Tensor Network*. 2023. arXiv: 2307.05740 [cs.DC].

[KT24] Risi Kondor and Erik Henning Thiede. *GElib*. Kondor Research Group, UChicago, July 2024. URL: https://github.com/risi-kondor/GElib.

[KU15] Oguz Kaya and Bora Uçar. "Scalable sparse tensor decompositions in distributed memory systems". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–11. DOI: 10.1145/2807591.2807624.

[Kwa+19] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. "Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication". In: *SC'19*. Denver, Colorado: ACM, 2019. ISBN: 9781450362290.

[Lam+24] Neill Lambert et al. *QuTiP 5: The Quantum Toolbox in Python*. 2024. DOI: 10.48550/arXiv.2412.04705. arXiv: 2412.04705 [quant-ph]. URL: https://arxiv.org/abs/2412.04705.

[Lan87] Serge Lang. "Determinants". In: *Linear Algebra*. New York, NY: Springer New York, 1987, pp. 140–179. ISBN: 978-1-4757-1949-9. DOI: 10.1007/978-1-4757-1949-9_6.

[LCK24] Shengjie Luo, Tianlang Chen, and Aditi S. Krishnapriyan. "Enabling Efficient Equivariant Operations in the Fourier Basis via Gaunt Tensor Products". In: *The Twelfth International Conference on Learning Representations*. 2024.

[Lee+25] Seung Yul Lee, Hojoon Kim, Yutack Park, Dawoon Jeong, Seungwu Han, Yeonhong Park, and Jae W. Lee. "FlashTP: Fused, Sparsity-Aware Tensor Product for Machine Learning Interatomic Potentials". In: *Forty-second International Conference on Machine Learning*. 2025.

[Lev+25] Daniel S. Levine et al. *The Open Molecules 2025 (OMol25) Dataset, Evaluations, and Models*. 2025. arXiv: 2505.08762 [physics.chem-ph]. URL: https://arxiv.org/abs/2505.08762.

[LGM24a] Kin Long Kelvin Lee, Mikhail Galkin, and Santiago Miret. "Deconstructing equivariant representations in molecular systems". In: *AI for Accelerated Materials Design - NeurIPS 2024*. 2024.

[LGM24b] Kin Long Kelvin Lee, Mikhail Galkin, and Santiago Miret. "Scaling Computational Performance of Spherical Harmonics Kernels with Triton". In: *AI for Accelerated Materials Design - Vienna 2024*. 2024.

[Li+21] Mingzhen Li, Yi Liu, Xiaoyan Liu, Qingxiao Sun, Xin You, Hailong Yang, Zhongzhi Luan, Lin Gan, Guangwen Yang, and Depei Qian. "The Deep Learning Compiler: A Comprehensive Survey". In: *IEEE Transactions on Parallel and Distributed Systems* 32.3 (2021), pp. 708–727. DOI: 10.1109/TPDS.2020.3030548.

[Lia+24] Yi-Lun Liao, Brandon M Wood, Abhishek Das, and Tess Smidt. "EquiformerV2: Improved Equivariant Transformer for Scaling to Higher-Degree Representations". In: *The Twelfth International Conference on Learning Representations*. 2024.

[Lin+25] Yuchao Lin, Cong Fu, Zachary Krueger, Haiyang Yu, Maho Nakata, Jianwen Xie, Emine Kucukbenli, Xiaofeng Qian, and Shuiwang Ji. *Tensor Decomposition Networks for Fast Machine Learning Interatomic Potential Computations*. 2025. arXiv: 2507.01131 [cs.LG]. URL: https://arxiv.org/abs/2507.01131.

[LK22] Brett W. Larsen and Tamara G. Kolda. "Practical Leverage-Based Sampling for Low-Rank Tensor Decomposition". In: *SIAM J. Matrix Analysis and Applications* 43.3 (Aug. 2022), pp. 1488–1517. DOI: 10.1137/21M1441754.

[LMV18] Jiajia Li, Yuchen Ma, and Richard Vuduc. *ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs*. Last updated: Jan 2020. Oct. 2018. URL: http://parti-project.org.

[LN23] Lek-Heng Lim and Bradley J Nelson. "What is... an Equivariant Neural Network?" In: *Notices of the American Mathematical Society* 70.4 (Apr. 2023), pp. 619–624. ISSN: 0002-9920, 1088-9477. DOI: 10.1090/noti2666.

[LS23] Yi-Lun Liao and Tess Smidt. "Equiformer: Equivariant Graph Attention Transformer for 3D Atomistic Graphs". In: *International Conference on Learning Representations*. 2023.

[LT08] Shuangzhe Liu and Götz Trenkler. "Hadamard, Khatri-Rao, Kronecker and other matrix products". In: *International Journal of Information and Systems Sciences* 4.1 (2008), pp. 160–177.

[Mah11] Michael W. Mahoney. "Randomized Algorithms for Matrices and Data". In: *Foundations and Trends® in Machine Learning* 3.2 (2011), pp. 123–224. ISSN: 1935-8237. DOI: 10.1561/2200000035.

[Mal22] Osman Asif Malik. "More Efficient Sampling for Tensor Decomposition With Worst-Case Guarantees". In: *Proceedings of the 39th International Conference on Machine Learning*. Vol. 162. Proceedings of Machine Learning Research. PMLR, July 2022, pp. 14887–14917.

[Mao+14] Hing-Hao Mao, Chung-Jung Wu, Evangelos E. Papalexakis, Christos Faloutsos, Kuo-Chen Lee, and Tien-Cheu Kao. "MalSpot: Multi2 Malicious Network Behavior Patterns Analysis". en. In: *Advances in Knowledge Discovery and Data Mining*. Ed. by Vincent S. Tseng, Tu Bao Ho, Zhi-Hua Zhou, Arbee L. P. Chen, and Hung-Yu Kao. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2014, pp. 1–14. ISBN: 978-3-319-06608-0. DOI: 10.1007/978-3-319-06608-0_1.

[MBM22] Osman Asif Malik, Vivek Bharadwaj, and Riley Murray. *Sampling-Based Decomposition Algorithms for Arbitrary Tensor Networks*. arXiv:2210.03828 [cs, math]. Oct. 2022. (Visited on 01/06/2023).

[MS21] Linjian Ma and Edgar Solomonik. "Efficient parallel CP decomposition with pairwise perturbation and multi-sweep dimension tree". In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. May 2021, pp. 412–421. DOI: 10.1109/IPDPS49936.2021.00049.

[MS22] Linjian Ma and Edgar Solomonik. "Cost-efficient Gaussian tensor network embeddings for tensor-structured inputs". In: *Advances in Neural Information Processing Systems*. Vol. 35. Curran Associates, Inc., 2022, pp. 38980–38993.

[MT20] Per-Gunnar Martinsson and Joel A. Tropp. "Randomized numerical linear algebra: Foundations and algorithms". In: *Acta Numerica* 29 (2020), pp. 403–572. DOI: 10.1017/S0962492920000021.

[Muk+24] Ujjaini Mukhopadhyay, Alok Tripathy, Oguz Selvitopi, Katherine Yelick, and Aydin Buluc. "Sparsity-Aware Communication for Distributed Graph Neural Network Training". In: *Proceedings of the 53rd International Conference on Parallel Processing*. ICPP '24. Gotland, Sweden: Association for Computing Machinery, 2024, pp. 117–126. DOI: 10.1145/3673038.3673152.

[Mur+23] Riley Murray, James Demmel, Michael W. Mahoney, N. Benjamin Erichson, Maksim Melnichenko, Osman Asif Malik, Laura Grigori, Piotr Luszczek, Michał Dereziński, Miles E. Lopes, Tianyu Liang, Hengrui Luo, and Jack Dongarra. *Randomized Numerical Linear Algebra : A Perspective on the Field With an Eye to Software*. 2023. arXiv: 2302.11474 [math.NA].

[Mus+23] Albert Musaelian, Simon Batzner, Anders Johansson, Lixin Sun, Cameron J. Owen, Mordechai Kornbluth, and Boris Kozinsky. "Learning local equivariant representations for large-scale atomistic dynamics". In: *Nature Communications* 14.1 (Feb. 2023), p. 579. ISSN: 2041-1723. DOI: 10.1038/s41467-023-36329-y.

[Ngu+22] Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi. "Efficient, out-of-Memory Sparse MTTKRP on Massively Parallel Architectures". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS '22. Virtual Event: Association for Computing Machinery, 2022. ISBN: 9781450392815. DOI: 10.1145/3524059.3532363.

[Nis+18] Israt Nisa, Aravind Sukumaran-Rajam, Sureyya Emre Kurt, Changwan Hong, and P. Sadayappan. "Sampled Dense Matrix Multiplication for High-Performance Machine Learning". In: *HiPC*. Dec. 2018, pp. 32–41. DOI: 10.1109/HiPC.2018.00013.

[Nis+19] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. "An Efficient Mixed-Mode Representation of Sparse Tensors". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver,

Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356216.

[Nov+15] Alexander Novikov, Dmitry Podoprikhin, Anton Osokin, and Dmitry Vetrov. "Tensorizing neural networks". In: *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 1*. NIPS'15. Montreal, Canada: MIT Press, 2015, pp. 442–450.

[Ose11] Ivan V Oseledets. "Tensor-train decomposition". In: *SIAM Journal on Scientific Computing* 33.5 (2011), pp. 2295–2317.

[OT10] Ivan Oseledets and Eugene Tyrtyshnikov. "TT-cross approximation for multidimensional arrays". In: *Linear Algebra and its Applications* 432.1 (2010), pp. 70–88. ISSN: 0024-3795. DOI: https://doi.org/10.1016/j.laa.2009.07.024.

[Par+16] Namyong Park, Byungsoo Jeon, Jungwoo Lee, and U Kang. "BIGtensor: Mining Billion-Scale Tensor Made Easy". In: *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*. CIKM '16. Indianapolis, Indiana, USA: Association for Computing Machinery, 2016, pp. 2457–2460. ISBN: 9781450340731. DOI: 10.1145/2983323.2983332.

[Ped+11] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[Per+07] D. Perez-Garcia, F. Verstraete, M. M. Wolf, and J. I. Cirac. "Matrix product state representations". In: *Quantum Info. Comput.* 7.5 (July 2007), pp. 401–430. ISSN: 1533-7146.

[PETScThr] *Threads and PETSc — PETSc 3.16.2 documentation*. URL: https://petsc.org/release/miscellaneous/threads/ (visited on 01/05/2022).

[PK19] Eric T Phipps and Tamara G Kolda. "Software for sparse tensor decomposition on emerging computing architectures". In: *SIAM Journal on Scientific Computing* 41.3 (2019), pp. C269–C290.

[PP13] Ninh Pham and Rasmus Pagh. "Fast and scalable polynomial kernels via explicit feature maps". In: *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '13. Chicago, Illinois, USA: Association for Computing Machinery, 2013, pp. 239–247. ISBN: 9781450321747. DOI: 10.1145/2487575.2487591.

[PZ23] Saro Passaro and C. Lawrence Zitnick. "Reducing SO(3) Convolutions to SO(2) for Efficient Equivariant GNNs". In: *Proceedings of the 40th International Conference on Machine Learning*. Ed. by Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett. Vol. 202. Proceedings of Machine Learning Research. PMLR, July 2023, pp. 27420–27438.

[Rag+13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 519–530. ISSN: 0362-1340. DOI: 10.1145/2499370.2462176.

[RSA21] Md. Khaledur Rahman, Majedul Haque Sujon, and Ariful Azad. "FusedMM: A Unified SDDMM-SpMM Kernel for Graph Embedding and Graph Neural Networks". In: *IPDPS*. ISSN: 1530-2075. May 2021, pp. 256–266. DOI: 10.1109/IPDPS49936.2021.00034.

[RSK19] Thomas B. Rolinger, Tyler A. Simon, and Christopher D. Krieger. "Performance considerations for scalable parallel tensor decomposition". In: *Journal of Parallel and Distributed Computing* 129 (2019), pp. 83–98. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2017.10.013.

[RSZ22] Aravind Reddy, Zhao Song, and Lichen Zhang. "Dynamic Tensor Product Regression". In: *Advances in Neural Information Processing Systems*. Vol. 35. Curran Associates, Inc., 2022, pp. 4791–4804.

[RT08] Vladimir Rokhlin and Mark Tygert. "A fast randomized algorithm for overdetermined linear least-squares regression". In: *Proceedings of the National Academy of Sciences* 105.36 (2008), pp. 13212–13217. DOI: 10.1073/pnas.0804869105.

[Saa+20] Feras A. Saad, Cameron E. Freer, Martin C. Rinard, and Vikash K. Mansinghka. "Optimal Approximate Sampling from Discrete Probability Distributions". In: *Proceedings of the ACM on Programming Languages* 4.POPL (Jan. 2020), pp. 1–31. ISSN: 2475-1421. DOI: 10.1145/3371104. (Visited on 10/24/2022).

[Sab20] Amit Sabne. *XLA : Compiling Machine Learning for Peak Performance*. 2020.

[SB02] N.D. Sidiropoulos and R.S. Budampati. "Khatri-Rao space-time codes". In: *IEEE Transactions on Signal Processing* 50.10 (2002), pp. 2396–2407. DOI: 10.1109/TSP.2002.803341.

[SD11] Edgar Solomonik and James Demmel. "Communication-Optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms". In: *Euro-Par'11*. Bordeaux, France: Springer-Verlag, 2011, pp. 90–109. ISBN: 9783642233968.

[Sel+21] Oguz Selvitopi, Benjamin Brock, Israt Nisa, Alok Tripathy, Katherine Yelick, and Aydın Buluç. "Distributed-memory parallel algorithms for sparse times tall-skinny-dense matrix multiplication". In: *ICS*. ACM, 2021, pp. 431–442. ISBN: 978-1-4503-8335-6. DOI: 10.1145/3447818.3461472. (Visited on 10/12/2021).

[SK15] Shaden Smith and George Karypis. "Tensor-Matrix Products with a Compressed Sparse Tensor". In: *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. IAAA '15. Austin, Texas: Association for Computing Machinery, 2015. ISBN: 9781450340014. DOI: 10.1145/2833179.2833183.

[SK16a] Shaden Smith and George Karypis. "A Medium-Grained Algorithm for Sparse Tensor Factorization". In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. ISSN: 1530-2075. May 2016, pp. 902–911. DOI: 10.1109/IPDPS.2016.113.

[SK16b] Shaden Smith and George Karypis. *SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit*. https://github.com/ShadenSmith/splatt. Version 1.1.1. 2016.

[Smi+15] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. ISSN: 1530-2075. May 2015, pp. 61–70. DOI: 10.1109/IPDPS.2015.27.

[Smi+17] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. 2017. URL: http://frostt.io/.

[Smi+18] Shaden Smith, Kejun Huang, Nicholas D. Sidiropoulos, and George Karypis. "Streaming Tensor Factorization for Infinite Data Sources". In: *Proceedings of the 2018 SIAM International Conference on Data Mining (SDM)*. SIAM, 2018, pp. 81–89. DOI: 10.1137/1.9781611975321.10.

[Sol+14] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. "A massively parallel tensor contraction framework for coupled-cluster computations". In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Publisher: Academic Press, pp. 3176–3190.

[Son+22] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. "Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication". In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA '22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77. ISBN: 9781450391498. DOI: 10.1145/3490422.3502357.

[SUG21] Kristof Schütt, Oliver Unke, and Michael Gastegger. "Equivariant message passing for the prediction of tensorial properties and molecular spectra". In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 9377–9388.

[Tan+25] Chuin Wei Tan, Marc L. Descoteaux, Mit Kotak, Gabriel de Miranda Nascimento, Seán R. Kavanagh, Laura Zichi, Menghang Wang, Aadit Saluja, Yizhong R. Hu, Tess Smidt, Anders Johansson, William C. Witt, Boris Kozinsky, and Albert Musaelian. *High-performance training and inference for deep equivariant interatomic potentials*. 2025. arXiv: 2504.16068 [physics.comp-ph]. URL: https://arxiv.org/abs/2504.16068.

[Tho+18] Nathaniel Thomas, Tess Smidt, Steven Kearnes, Lusann Yang, Li Li, Kai Kohlhoff, and Patrick Riley. *Tensor field networks: Rotation- and translation-equivariant neural networks for 3D point clouds*. May 18, 2018. arXiv: 1802.08219[cs]. URL: https://arxiv.org/abs/1802.08219 (visited on 10/02/2024).

[TKC19] Philippe Tillet, H. T. Kung, and David Cox. "Triton: an intermediate language and compiler for tiled neural network computations". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*. MAPL 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 10–19. ISBN: 9781450367196. DOI: 10.1145/3315508.3329973.

[Tok+25] Neriman Tokcan, Shakir Showkat Sofi, Van Tien Pham, Clémence Prévost, Sofiane Kharbech, Baptiste Magnier, Thanh Phuong Nguyen, Amirhamzeh Khoshnam, Yassine Zniyed, and Lieven de Lathauwer. "Tensor Decompositions for Signal Processing: Theory, Advances, and Applications". In: *Signal Processing* (June 2025).

[TP21] Jesmin Jahan Tithi and Fabrizio Petrini. "An Efficient Shared-memory Parallel Sinkhorn-Knopp Algorithm to Compute the Word Mover's Distance". In: *arXiv:2005.06727 [cs, stat]* (Mar. 2021). arXiv: 2005.06727. (Visited on 06/28/2021).

[Tra+23] Richard Tran et al. "The Open Catalyst 2022 (OC22) Dataset and Challenges for Oxide Electrocatalysts". In: *ACS Catalysis* 13.5 (2023), pp. 3066–3084. DOI: 10.1021/acscatal.2c05426.

[TYB20] Alok Tripathy, Katherine Yelick, and Aydın Buluç. "Reducing communication in graph neural network training". In: *SC'20*. IEEE, 2020, pp. 1–17. ISBN: 978-1-72819-998-6. (Visited on 10/12/2021).

[UM24] Oliver T. Unke and Hartmut Maennel. *E3x:* E(3)*-Equivariant Deep Learning Made Easy*. 2024. arXiv: 2401.07595 [cs.LG]. URL: https://arxiv.org/abs/2401.07595.

[Vel+18] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. "Graph Attention Networks". In: *arXiv:1710.10903 [cs, stat]* (Feb. 2018). arXiv: 1710.10903. (Visited on 07/08/2021).

[Wan+15] Yining Wang, Hsiao-Yu Tung, Alexander J Smola, and Anima Anandkumar. "Fast and guaranteed tensor decomposition via sketching". In: *Advances in neural information processing systems* 28 (2015).

[Wan+19] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. "Deep Graph Library: A Graph-Centric, Highly-Performant Package for Graph Neural Networks". In: *arXiv preprint arXiv:1909.01315* (2019).

[Wan+25] Lei Wang, Yu Cheng, Yining Shi, Zhengju Tang, Zhiwen Mo, Wenhao Xie, Lingxiao Ma, Yuqing Xia, Jilong Xue, Fan Yang, and Zhi Yang. *TileLang: A Composable Tiled Programming Model for AI Systems*. 2025. arXiv: 2504.17577 [cs.LG]. URL: https://arxiv.org/abs/2504.17577.

[Wei+18] Maurice Weiler, Mario Geiger, Max Welling, Wouter Boomsma, and Taco Cohen. "3D steerable CNNs: learning rotationally equivariant features in volumetric data". In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. NIPS'18. Montréal, Canada: Curran Associates Inc., 2018, pp. 10402–10413.

[Whi92] Steven R. White. "Density matrix formulation for quantum renormalization groups". In: *Phys. Rev. Lett.* 69 (19 Nov. 1992), pp. 2863–2866. DOI: 10.1103/PhysRevLett.69.2863.

[Wij+23] Sasindu Wijeratne, Ta-Yang Wang, Rajgopal Kannan, and Viktor Prasanna. "Accelerating Sparse MTTKRP for Tensor Decomposition on FPGA". In: *Proceedings of the 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*. FPGA '23. Monterey, CA, USA: Association for Computing Machinery, 2023, pp. 259–269. ISBN: 9781450394178. DOI: 10.1145/3543622.3573179.

[WKP23] Sasindu Wijeratne, Rajgopal Kannan, and Viktor Prasanna. *Dynasor: A Dynamic Memory Layout for Accelerating Sparse MTTKRP for Tensor Decomposition on Multi-core CPU*. 2023. arXiv: 2309.09131 [cs.DC].

[Woo+14] David P Woodruff et al. "Sketching as a tool for numerical linear algebra". In: *Foundations and Trends® in Theoretical Computer Science* 10.1–2 (2014), pp. 1–157.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. "Roofline: an insightful visual performance model for multicore architectures". In: *Commun. ACM* 52.4 (Apr. 2009), pp. 65–76. ISSN: 0001-0782. DOI: 10.1145/1498765.1498785.

[WZ22] David Woodruff and Amir Zandieh. "Leverage Score Sampling for Tensor Product Matrices in Input Sparsity Time". In: *Proceedings of the 39th International Conference on Machine Learning*. Vol. 162. Proceedings of Machine Learning Research. PMLR, July 2022, pp. 23933–23964.

[Xie+24] YuQing Xie, Ameya Daigavane, Mit Kotak, and Tess Smidt. "The Price of Freedom: Exploring Tradeoffs between Expressivity and Computational Efficiency in Equivariant Tensor Products". In: *ICML 2024 Workshop on Geometry-grounded Representation Learning and Generative Modeling*. June 2024. (Visited on 10/02/2024).

[YAK22a] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. "DISTAL: the distributed tensor algebra compiler". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 286–300. ISBN: 9781450392655. DOI: 10.1145/3519939.3523437.

[YAK22b] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. "SpDISTAL: Compiling Distributed Sparse Tensor Computations". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022.

[YBO18] Carl Yang, Aydın Buluç, and John D. Owens. "Design Principles for Sparse Matrix Multiplication on the GPU". In: *Euro-Par 2018: Parallel Processing: 24th International Conference on Parallel and Distributed Computing, Turin, Italy, August 27 - 31, 2018, Proceedings*. Turin, Italy: Springer-Verlag, 2018, pp. 672–687. ISBN: 978-3-319-96982-4. DOI: 10.1007/978-3-319-96983-1_48.

[YL24] Erika Ye and Nuno F. Loureiro. "Quantized tensor networks for solving the Vlasov–Maxwell equations". In: *Journal of Plasma Physics* 90.3 (2024), p. 805900301. DOI: 10.1017/S0022377824000503.

[YPP10] Yao Yu, Athina P. Petropulu, and H. Vincent Poor. "MIMO Radar Using Compressive Sampling". In: *IEEE Journal of Selected Topics in Signal Processing* 4.1 (Feb. 2010), pp. 146–163. DOI: 10.1109/JSTSP.2009.2038973. arXiv: 0911.4752 [cs.IT].

[Yu+12] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. "Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems". In: *2012 IEEE 12th International Conference on Data Mining*. ISSN: 2374-8486. Dec. 2012, pp. 765–774. DOI: 10.1109/ICDM.2012.168.