ACCELERATING SPARSE MATRICIZED TENSOR TIMES KHATRI RAO PRODUCT ON MASSIVELY PARALLEL ARCHITECTURES

by

ANDY NGUYEN

DR. JEE CHOI, ADVISOR

A THESIS

Submitted in partial fulfillment of the requirements
for the degree of Bachelor of Science
in the Department of Computer Science
in the College of Arts and Sciences of
University of Oregon

EUGENE, OREGON

2023

ABSTRACT

Tensor decomposition is a popular data analysis technique that extracts latent information from multi-dimensional data. High performance implementations of tensor decomposition face a number of challenges, such as efficiently storing sparse tensor data, achieving optimally load-balanced parallelism, and addressing the resource limitations of GPU accelerators. This thesis presents Blocked Linearized CoOrdinates (BLCO), a novel sparse tensor format built to accelerate tensor-related computations on massively parallel GPU architectures. In contrast to prior approaches, BLCO enables efficient out-of-memory computation of tensor algorithms using a unified implementation operating on only a single in-memory tensor copy. Our linearization and adaptive blocking strategies not only meet the resource constraints of target GPU devices but also accelerate data indexing, eliminate control-flow and memory-access irregularities, and reduce kernel launch overhead, all while maintaining a high compression rate. To address the substantial synchronization costs on GPUs, we introduce an opportunistic conflict resolution algorithm, in which threads collaborate to discover and resolve conflicting updates on-the-fly, at successive levels of the memory hierarchy. Overall, our tensor decomposition framework delivers superior in-memory performance compared to prior state-of-the-art GPU implementations, and it is the only framework currently capable of processing large scale, out-of-memory tensors. On the latest Intel and NVIDIA GPUs, BLCO achieves 2.12x - 2.6x geometric mean speedup over the prior state-of-the-art (Mixed-Mode Compressed Sparse Fiber) and demonstrates consistent performance across a gamut of reprsentative real-world tensor data.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A *tensor* is a higher order generalization of a matrix, and it provides a natural abstraction for complex and multi-dimensional data in the real world. Tensors appears in a wide discipline of fields, including data mining [1], [2], social network analytics [3], [4], cybersecurity [5], [6], healthcare [7], [8], language learning [9], textual analysis [10], product review analysis [11], and more. As an example, image and image metadata from the image hosting site Flickr[1] can be captured and stored in tuples of user-image-tag-date, which corresponds to a 4D tensor with each dimension representing a separate metadata category. In the real world, such tensors tend to be large and *sparse*, with irregular shape and non-uniform distribution of data. Compressed storage formats and highly parallel algorithms are important to performing efficient tensor operations on a wide variety of tensors.

Tensor decomposition is a data analysis technique used to extract salient information from tensor data. Canonical Polydiac Decomposition (CPD), the decomposition of interest in this thesis, breaks down the original tensor into a low-rank approximation, which facilitates further analysis. A typical implementation of CPD consists of various matrix operations from a linear algebra library alongside a high performance implementation of *Matricized Tensor Times Khatri-Rao Product (MTTKRP)*, which dominates the execution time of CPD [12]. A number of frameworks have been developed over the years to perform CPD, including SPLATT [12], ParTi! [13], and GenTen [14]; each one introduces a novel approach to optimize MTTKRP and CPD.

[1] http://www.flickr.com

A number of works accelerate the MTTKRP computation on GPUs [13]–[17]. The massively parallel execution and high bandwidth capabilities of GPUs makes them a suitable target architecture for optimizing tensor computations. However, a sparse implementation of MTTKRP runs into several performance concerns, namely:

- Irregular memory access: sparse tensors have wide ranges of sparsity patterns and consequently memory access paterns, complicating cache usage
- Low arithmetic intensity of MTTKRP: the heavy emphasis on memory traffic creates latency and bandwidth issues, hindering full device utilization
- Workload imbalance: the data layout of a sparse tensor can lead to severe parallel performance degradation
- Synchronization overhead: the expensive cost of atomic operations dominates execution time, especially on massively parallel accelerators
- Limited on-device memory: out-of-memory (OOM) tensors cannot fit entirely in on-device memory and thus cannot be naïvely operated on

To address these concerns, prior studies primarily focus on designing sparse tensor formats that either compress the tensor to minimize its memory footprint and/or group data-dependent non-zero elements together to reduce atomic operations. However, these strategies result in formats that are *mode-specific*, where non-zero elements are organized and accessed according to a specific dimension. Mode-specific formats typically require keeping multiple tensor copies [15], [16], [18] or creating excessive mapping and scheduling information (i.e. mode-specific flags or arrays) about groups of data-dependent non-zero elements for every mode [15], [17], [18], both of which can significantly increase the overall memory footprint. Additionally, the mode-specific nature makes out-of-memory tensors, which can have billions of non-zero elements, difficult to process due to limited on-device memory. Current GPU frameworks for MTTKRP and CPD are constrained to sparse tensors that can fit in only the limited device memory (i.e. in-memory) and do not support OOM tensors.

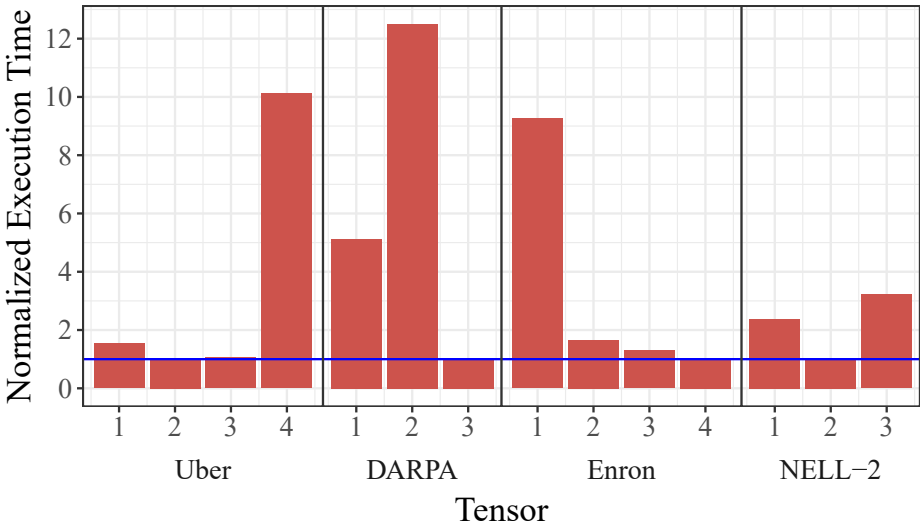


Figure 1.1: The MTTKRP execution time of MM-CSF across all modes on the A100 GPU, normalized by the lowest execution time (denoted by the blue line) for each data set. The decomposition rank is 32. For NELL-2, modes 1 and 3 take 2–3× longer to execute than mode 2, while for Uber and Enron, one mode takes *more than* 9× *longer* to execute. For DARPA, mode 1 and mode 2 take 5× and 12× longer, respectively, than mode 3. Note that the number of FLOPs computed is *identical* across modes for each data set.

Furthermore, mode-specific sparse tensor formats can result in inconsistent compression and performance along different modes of operation. For example, Figure 1.1 illustrates the impact of mode-specific compression on the performance on MTTKRP on each mode, measured using the prior state-of-the-art format mixed-mode compressed sparse fiber (MM-CSF) [17]. The results demonstrate that depending on the data set, the execution time may vary by an *order of magnitude* across modes when the compression favors one particular mode over the others. This issue is compounded with the tree-like nature of data structures based on compressed sparse fiber [12]: different tree traversal algorithms are required depending on the mode, leading to poor scalability and portability.

In summary, current state-of-the-art approaches to accelerating MTTKRP on GPUs rely on mode-specific formats to reduce data movement via compression and to decrease the number of atomic operations by reordering non-zero elements. However, these strategies result in (i) drastic performance degradation along different modes of tensor operation, (ii) significant memory overhead required to keep extra tensor copies or mapping flags/arrays,

(iii) complex algorithms and code implementations to handle different modes, and (iv) limited support for processing real-world tensor datasets on memory-constrained accelerators.

Given these concerns, we propose a novel *mode-agnostic* framework for large-scale sparse tensor decomposition on GPUs. The rest of this thesis is as follows. We first provide a mathematical overview of relevant tensor arithmetic (Chapter 2). Then we analyze prior state-of-the-art sparse tensor formats and parallel MTTKRP algorithms to pinpoint the key performance bottlenecks and limitations (Chapter 3). We introduce the Blocked Linearized CoOrdinate (BLCO) sparse tensor format, discussing index linearization and adaptive blocking techniques used to accelerate MTTKRP with a unified tensor representation (Chapter 4). We present a massively parallel MTTKRP algorithm that eliminates irregularities in control-flow and memory-access and uses various levels of the memory hierarchy (Chapter 5). Then we demonstrate substantial performance improvement compared to prior state-of-the-art approaches, achieving 2.12x-2.6x speedup on a wide range of real-world tensors on Intel and NVIDIA GPU architectures and novel support for out-of-memory tensors (Chapter 6). We end with related works in the sparse tensor decomposition field (Chapter 7) and offer concluding remarks (Chapter 8).

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief overview of tensors, tensor decomposition, and relevant computations. For more detailed information about tensor decomposition, we refer the reader to the work by Kolda and Bader [19], [20].

2.1 Notation

Tensors are higher-order generalizations of matrices. An N-dimensional tensor is referred to as an *mode-N* tensor or as having N *modes*. The following notations are used henceforth:

1. *Scalars* are denoted by lower case letters (e. g., a).
2. *Vectors* are denoted by bold lower case letters (e. g., $\mathbf{a}$).
3. *Matrices* are denoted by bold capital letters (e. g., $\mathbf{A}$). A $I_1 \times I_2$ matrix $\mathbf{A}$ can be denoted as $\mathbf{A} \in \mathbb{R}^{I_1 \times I_2}$.
4. *Higher-order tensors* are denoted by bold Euler script letters (e. g., $\boldsymbol{\mathcal{X}}$). A mode-N tensor $\boldsymbol{\mathcal{X}}$ with dimensions $I_1 \times I_2 \times \cdots \times I_N$ can be denoted as $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$.
5. *Matricization* is the process of reordering the elements of a tensor into a matrix. Mode-n matricization of a tensor $\boldsymbol{\mathcal{X}}$, denoted as $\mathbf{X}_{(n)}$, is a $I_n \times \hat{I}_n$ matrix, where $\hat{I}_n = \prod_{i \neq n} I_n$.
6. *Fibers* are the analogue of matrix rows/columns for higher-order tensors. A mode-n fiber of a tensor $\boldsymbol{\mathcal{X}}$ is any vector formed by fixing all indices of $\boldsymbol{\mathcal{X}}$, *except* the n^{th} index

(e.g., a matrix column is defined by fixing the second index, and is therefore a mode-1 fiber).

7. *Slices* are sub-tensors that are formed by fixing one particular index. For example, we can form a time slice (i.e., mode-N-1 subtensor) by fixing the time index to a specific value (e.g., 0 for the first time slice) for a mode-N tensor. That is, a slice is a mode-(N-1) tensor that can be formed by fixing the time mode index to 10 for a mode-N tensor. This is also known as a *time slice*.

8. The *Hadamard product*, or the element-wise product between two matrices, is denoted by $\circledast$.

9. The *Kronecker* product between two matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{K \times L}$ produces the matrix $\mathbf{C} \in \mathbb{R}^{IJ \times KL}$, where

$$\mathbf{C} = \begin{bmatrix} \mathrm{a}_{1,1}\mathbf{B} & \mathrm{a}_{1,2}\mathbf{B} & \cdots & \mathrm{a}_{1,J}\mathbf{B} \\ \mathrm{a}_{2,1}\mathbf{B} & \mathrm{a}_{2,2}\mathbf{B} & \cdots & \mathrm{a}_{2,J}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ \mathrm{a}_{I,1}\mathbf{B} & \mathrm{a}_{I,2}\mathbf{B} & \cdots & \mathrm{a}_{I,J}\mathbf{B} \end{bmatrix}$$

   and is denoted $\mathbf{A} \otimes \mathbf{B}$.

10. The *Khatri-Rao product*, or the column-wise Kronecker product between two matrices, is denoted by $\odot$. The Khatri-Rao product between two matrices $\mathbf{A} \in \mathbb{R}^{I_1 \times K}$ and $\mathbf{B} \in \mathbb{R}^{I_2 \times K}$ yields the matrix $\mathbf{C} \in \mathbb{R}^{I_1 I_2 \times K}$

2.2 CPD

The Canonical Polyadic Decomposition (CPD) is a widely used tensor factorization model in data analysis. CPD approximates an N-order tensor $\boldsymbol{\mathcal{X}}$ by the sum of R outer products of N appropriately sized vectors, for some preselected R (c.f. Figure 2.1). Each of the

N vectors in a given outer product corresponds to a particular tensor mode. The outer products are called the rank-1 tensors of the decomposition and the quantity R is called the decomposition *rank*. By arranging the R vectors corresponding to a particular mode as the columns of a matrix, we obtain the *factor matrix* associated with that mode. The decomposition of $\boldsymbol{\mathcal{X}}$ can then be written in terms of its factor matrices. For example, if $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I \times J \times K}$ is a third-order tensor, the CPD of $\boldsymbol{\mathcal{X}}$ may be written in terms of three factor matrices $\mathbf{A}^{(1)} \in \mathbb{R}^{I \times R}$, $\mathbf{A}^{(2)} \in \mathbb{R}^{J \times R}$, and $\mathbf{A}^{(3)} \in \mathbb{R}^{K \times R}$.

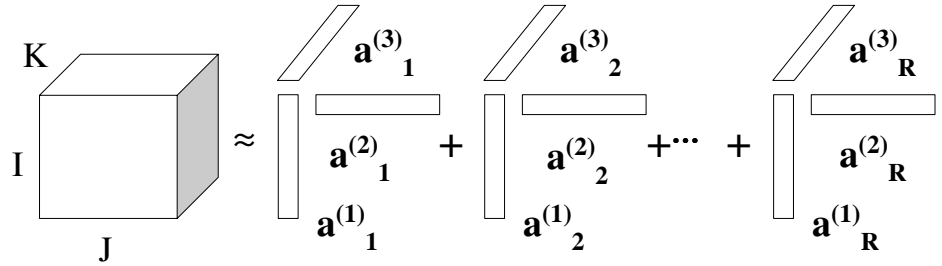


Figure 2.1: Rank-R CPD of a third-order tensor. The factor matrix $\mathbf{A}^{(1)}$ consists of vectors $\mathbf{a}_1^{(1)}, \mathbf{a}_2^{(1)}, \cdots, \mathbf{a}_R^{(1)}$.

The CANDECOMP/PARAFAC Alternating Least Squares (CP-ALS) algorithm is a popular iterative method for calculating the CPD. During each CP-ALS iteration, we update one factor matrix corresponding to a given tensor mode by solving a linear least squares problem, while fixing the remaining factor matrices; this is done exactly once for each factor matrix per iteration. For example, to perform the update for the first factor matrix $\mathbf{A}$, we fix the remaining matrices $\mathbf{B}$ and $\mathbf{C}$ and solve for $\mathbf{A}$ in the least squares problem:

$$\min_{\mathbf{A}} \left\| \mathbf{X}_{(1)} - \mathbf{A} \left(\mathbf{C} \odot \mathbf{B} \right)^T \right\|_F^2 \tag{2.1}$$

The least squares problem is minimized by

$$\hat{\mathbf{A}} = \mathbf{X}_{(1)} \left(\mathbf{C} \odot \mathbf{B} \right) \left(\mathbf{C}^T \mathbf{C} \circledast \mathbf{B}^T \mathbf{B} \right)^{\dagger} \tag{2.2}$$

where † indicates a pseudoinverse. We repeat this twice more for $\mathbf{B}$ and $\mathbf{C}$ to conclude one

iteration. Iterations are repeated until a convergence criteria is met, such as a fixed number of iterations or until the approximation ceases to improve beyond some error threshold. The CP-ALS algorithm for an N-order tensor is detailed in Algorithm 1. Line 4 shows the MTTKRP computations.

Algorithm 1 The CP-ALS algorithm.

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

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

2.3 Matricized Tensor Times Khatri-Rao Product

In Equation 2.2, the intermediate calculation $\mathbf{M} = \mathbf{X}_{(1)}(\mathbf{C} \odot \mathbf{B})$ is referred to as the *Matricized Tensor Times Khatri-Rao Product*, or as simply *MTTKRP*. It consists of a matrix multiplication between the target tensor's matricization with a Khatri-Rao product of the two factor matrices. For sparse tensors (or any tensor with large mode lengths), explicitly calculating the Khatri-Rao product is prohibitively expensive, in terms of both time and memory. In practice, all state-of-the-art tensor formats [12], [15]–[17], [20], [21] instead calculate the rows of the Khatri-Rao product matrix *on-the-fly*, constructing the corresponding row for each tensor non-zero element only as needed (c.f. Figure 2.2). This creates a number of performance optimization challenges (discussed in Chapter 1); on modern systems, the most expensive operation of CP-ALS, as well as many other tensor algorithms, is MTTKRP [12].

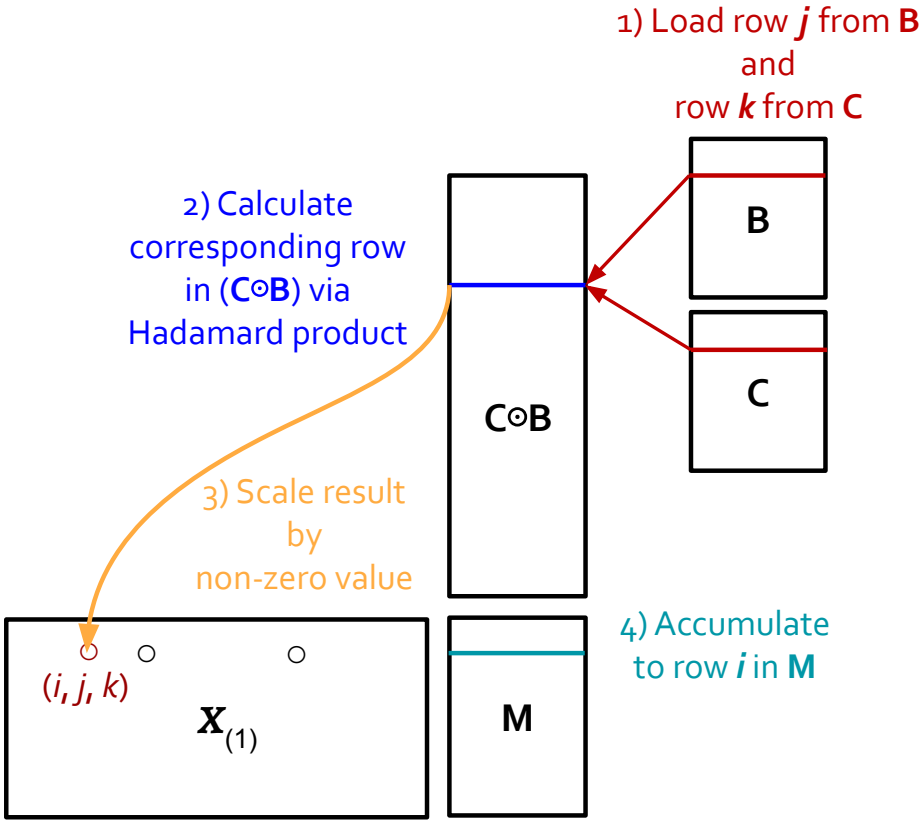


Figure 2.2: Mode-1 MTTKRP operation for a third-order sparse tensor. For each non-zero element with index (i_1, i_2, i_3), rows i_2 and i_3 from factor matrices $\mathbf{A}^{(2)}$ and $\mathbf{A}^{(3)}$ are fetched (1) and their Hadamard product (element-wise product) is calculated (2). The result is scaled by the non-zero element's value (3), and then accumulated to matrix row i_1 from matrix $\mathbf{M}$ (4), which is later used to calculate $\mathbf{A}^{(1)}$.

CHAPTER 3

SPARSE TENSOR FORMATS FOR GPUS

The state-of-the-art tensor formats for massively parallel GPUs use list-based [14], [15], [18] or tree-based [16], [17] data structures to store high-dimensional sparse data in a *mode-specific* form. In this section, we provide an overview of the Flagged CoOrdinate (F-COO) and Mixed-Mode Compressed Sparse Fiber (MM-CSF) formats, which are representative of the two main format categories for GPU architectures, respectively. In contrast to sparse linear algebra, higher-order tensor algorithms typically perform tensor operations on *every mode orientation*. Using MTTKRP as a case study, we illustrate the challenges in optimizing and efficiently executing sparse tensor operations on GPUs.

3.1 List-based Sparse Tensor Formats

The simplest form of this category is the coordinate (COO) format [14], [20], which keeps N+1 lists for an N-order tensor. N lists are for the indices, and one list is for the non-zero values. While COO is a mode-agnostic format, it suffers from substantial *synchronization overhead* due to update conflicts across threads. For example, Figure 2.2 depicts the mode-1 MTTKRP operation, where every thread that is processing non-zero elements with mode-1 index of i_1 (red and black circles) accumulates its partial result to row i_1 of matrix $\mathbf{M}$ (step (4)). This results in chains of read-after-write (RAW) data hazards, which require expensive locks or atomic operations to resolve; on massively parallel GPUs, this can quickly become a severe performance bottleneck because of the large number of concurrent threads and the high-latency memory system.

i_1	i_2	i_3	v
1	1	1	1.0
1	1	2	2.0
1	3	3	3.0
2	1	2	4.0
2	1	3	5.0
3	1	2	6.0
3	4	4	7.0
4	2	1	8.0
4	2	2	9.0
4	3	3	10.0
4	3	4	11.0
4	4	4	12.0

(a) COO.

	bf	i_2	i_3	v
sf=1	1	1	1	1.0
	1	1	2	2.0
	0	3	3	3.0
	1	1	2	4.0
sf=1	0	1	3	5.0
	1	1	2	6.0
	0	4	4	7.0
	1	2	1	8.0
sf=0	1	2	2	9.0
	1	3	3	10.0
	1	3	4	11.0
	0	4	4	12.0

(b) F-COO for mode-1 MTTKRP.

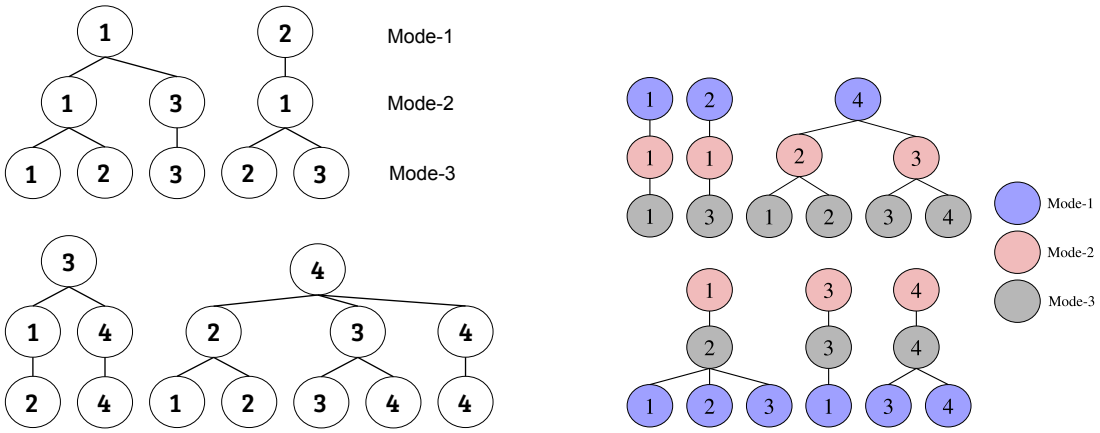
Figure 3.1: Comparison between COO (3.1a) and F-COO (3.1b) sparse tensor formats.. In F-COO, the target mode index (i_1) is replaced by bf (bit flag) that goes from 1 to 0 when the index changes. The sf (start flag) stores a bit for each non-zero group, where 1 indicates that a new target index has been encountered by the group.

To address this issue, the F-COO [15] sparse tensor format uses a sorted *mode-specific* form to group non-zero elements with the same i_1 index together (same i_1 for mode-1 MTTKRP, same i_2 for mode-2 MTTKRP, etc.) This allows for the local accumulation of partial results using segmented scan [22], [23] and global accumulation using atomic operations when group boundaries are crossed (at the thread-block and grid levels, respectively). Additionally, F-COO keeps extra mapping/scheduling information (flags) for synchronization to delineate the end of a non-zero group. As a result, the F-COO format needs to keep N tensor copies in the GPU memory for an N-order tensor; while F-COO reduces the number of global atomic operations, it has high memory usage due to extra data. Figure 3.1 shows an example sparse tensor in the COO (3.1a) and F-COO (3.1b) representations.

3.2 Tree-based Sparse Tensor Formats

The original CSF format [12], [24] extends traditional compressed matrix formats, such as the compressed sparse row (CSR) format, by storing a tensor as a collection of index subtrees with mode-specific ordering. Figure 3.2a shows the CSF representation with a mode

ordering of 1-2-3, where 1 is the root mode and 3 is the leaf mode, for the sparse tensor from Figure 3.1a. Given a CSF representation with a mode ordering of 1-2-3, where 1 is the root mode and 3 is the leaf mode, the root node of each sub-tree represents the factor matrix row that will be updated during mode-1 MTTKRP, and the leaf nodes represent the non-zero elements that contribute to that update. For example, the sub-tree on the top right in Figure 3.2a represents the two non-zero elements with indices (2,1,2) and (2,1,3) that contribute to calculating row 2 of the mode-1 factor matrix (i.e., matrix **M** from Figure 2.2). Since the two non-zero elements share the same mode-1 and mode-2 indices, CSF encodes these elements using 4 index nodes instead of 6, thereby compressing the indexing metadata.



(a) CSF tree for mode-1.

(b) MM-CSF trees.

Figure 3.2: CSF (3.2a) and MM-CSF (3.2b) representations for the sparse tensor from Figure 3.1a. Unlike CSF, where every sub-tree has the same mode orientation, MM-CSF identifies fibers with the highest number of non-zero elements, and then constructs sub-trees with different mode orientations.

In order to calculate MTTKRP for all modes, two strategies can be used. One requires multiple CSF copies with different root modes, which increases the memory footprint by a factor of N, where N is the number of modes. Alternatively, the index sub-trees can be traversed both bottom-up and top-down, meeting at the tree level with the target mode for MTTKRP. While the second approach allows computing MTTKRP for all modes with one copy of the CSF format, it requires expensive synchronization to avoid update conflicts as well as separate tree traversal implementations. Moreover, regardless of the strategy used,

CSF suffers from *workload imbalance* due to the variable size of each sub-tree.

Balanced CSF (B-CSF) [16] improves on CSF by creating sub-trees that are more workload balanced for GPUs, but still requires N copies of the tensor. The state-of-the-art MM-CSF [17] improves upon B-CSF by using a single copy of the tensor. This is achieved by analyzing fiber density (i.e., number of non-zero elements in a fiber) and creating sub-trees with fibers that are as dense as possible. However, the root of these sub-trees can come from any mode; hence, different traversal methods and parallel algorithms are required to perform MTTKRP on different modes, leading to drastic performance variations across different modes as shown in Figure 1.1. Furthermore, the complexity of the tree-based structure requires different implementations for each tensor order (i.e., number of modes) and it also restricts MM-CSF to tensors that can fit in the limited GPU memory. As a result, the current MM-CSF implementation only supports 3- and 4-dimensional tensors and cannot handle out-of-memory (OOM) tensors. Figure 3.2b shows an example of the MM-CSF format for the tensor from Figure 3.1a.

CHAPTER 4

THE BLCO FORMAT

To address the limitations of prior GPU-based formats, we propose the Blocked Linearized CoOrdinate (BLCO) format, a new sparse tensor representation devised for massively parallel architectures. BLCO linearizes and aggregates non-zero elements into coarse-grained blocks that meet the resource constraints of target GPUs to (i) minimize data movement, (ii) accelerate indexing, (iii) enable a unified tensor representation and algorithmic implementation, and (iv) support out-of-memory tensor computation. Figure 4.1 depicts an example of the BLCO format for the sparse tensor in Figure 3.1a. Generating a BLCO tensor consists of two stages: tensor linearization (Section 4.1) and adaptive blocking (Section 4.2).

4.1 Tensor Linearization

The BLCO format leverages *index linearization* [21], [25] to map multi-dimensional space onto one-dimensional linear space, such that a point in N-D space, represented by N coordinates, can be mapped to a point on an encoding line, represented by a *single* index. The length of the encoding line determines the number of bits required to represent the linearized indices. During computation (e.g. MTTKRP), the linear indices are *de-linearized* to recover the original coordinates. Fast de-linearization is important for high-performance execution of tensor algorithms using linearized formats.

To efficiently encode multi-dimensional spaces with irregular shape (which typically arise in higher-order sparse data) in a mode-agnostic way, prior state-of-the-art linear format ALTO [21] recursively partitions the multi-dimensional space and traverses this space lin-

l		v
0	$(000000)_2$	1.0
4	$(000100)_2$	2.0
5	$(000101)_2$	4.0
10	$(001010)_2$	8.0
12	$(001100)_2$	6.0
15	$(001111)_2$	9.0
33	$(100001)_2$	5.0
48	$(110000)_2$	3.0
57	$(111001)_2$	10.0
61	$(111101)_2$	11.0
62	$(111110)_2$	7.0
63	$(111111)_2$	12.0

(a) Initial linearization.

b	l		v
0	0	$(00000)_2$	1.0
	16	$(10000)_2$	2.0
	17	$(10001)_2$	4.0
	6	$(00110)_2$	8.0
	18	$(10010)_2$	6.0
	23	$(10111)_2$	9.0
1	1	$(00001)_2$	5.0
	8	$(01000)_2$	3.0
	11	$(01011)_2$	10.0
	27	$(11011)_2$	11.0
	30	$(11110)_2$	7.0
	31	$(11111)_2$	12.0

(b) BLCO tensor.

Figure 4.1: An example of the BLCO format for the sparse tensor in Figure 3.1a, where the bits of linearized indices are color-coded to indicate different modes. First, BLCO linearizes the tensor (4.1a) based on a recursive partitioning of the multi-dimensional space [21]. Next, it aggregates non-zero elements into blocks (4.1b) according to accelerators' resource constraints, namely on-device memory, length of encoding line, and native support for bit manipulation. For simplicity, we assume that the maximum length of the encoding line is 32 (2^5) and each block has no more than 6 non-zero elements. Note that BLCO re-encodes the linearized index (l) to allow the use of bitwise mask/shift (which are efficiently supported on GPUs) for de-linearization instead of bit-level gather/scatter.

early using a compact space-filling curve. When the multi-dimensional space is regular, i.e., all modes have the same length, the resulting space-filling curve resembles Morton-Z ordering [26]. This linearization of non-zero elements has shown to outperforms state-of-the-art tree-, list-, and block-based formats [21] on CPU-based platforms, and therefore, we adopt its ordering for BLCO.

Such a *mode-agnostic* linearized encoding results in an *interleaving* of bits from different mode indices, as shown in Figure 4.1a. To quickly process indices encoded in this manner, efficient support for bit-level *scatter* and *gather* operations are needed. However, GPUs lack native support for advanced bit manipulation, and emulating these instructions can be prohibitive,[1] leading to inefficient tensor processing operations. To address this issue, the BLCO format arranges the non-zero elements using ALTO ordering, but *re-encodes* the linearized indices to allow the use of bitwise shift and mask (AND) instructions, which are natively supported on accelerators, to de-linearize the indices. As illustrated in Figure 4.1b, BLCO achieves this goal by rearranging the encoding bits of linearized indices (l) into contiguous mode sets that can be quickly extracted on GPUs during tensor computations. This re-encoding resembles the index concatenation encoding of the LCO format [25], instead with bitwise shift and mask used for de-linearization in place of arithmetic division and modulo. The additional bitwise operations introduce additional overhead, especially for smaller tensors that do not require block indices at all (that is, they have no metadata bits). However, since MTTKRP is generally a memory-bounded algorithm [27], the computational overhead is minimal and serves to increase arithmetic intensity instead, alleviating global memory pressure overall.

[1]For a third-order tensor, we estimate that a naïve emulation would require 276 bitwise operations to de-linearize each non-zero element.

4.2 Adaptive Blocking

To map a tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ into a linearized form, an encoding line of length $I_1 \times \cdots \times I_N$ is required. As the number of modes and their lengths increase, the encoding line and the range of linear index values can significantly expand. Hence, large-scale tensors may require more than 64 bits to encode a linear index. Since GPUs do not provide native support for large integer (more than 64 bits) operations, a custom implementation[2] of large integer arithmetic is needed, which are typically not as efficient as native instructions. In addition, many tensors only require a slightly higher bit resolution (only a *few* additional bits) than 64 bits, so using large integers for linearized indices may lead to wasted memory for indices.

Most importantly, the state-of-the-art libraries for sparse MTTKRP require the *entire* tensor to be in the *limited* GPU memory before computation can begin, as a consequence of their compressed and/or mode-specific tensor formats. In particular, it is challenging to stream the data in small chunks that can be partially processed using tree- (e.g., B-CSF [16] and MM-CSF [17]) and block-based (e.g., HiCOO [28]) formats, where the granularity of tensor chunks (e.g., sub-trees and HiCOO blocks) are difficult to control. List-based formats (e.g., GenTen [14] and F-COO [15]) are more amenable to streaming but will require additional information for synchronization across these data chunks.

To address these issues, we propose *adaptive blocking* to aggregate non-zero elements into coarse-grained blocks by partitioning the tensor into smaller sub-tensors. Blocking the tensor to exploit density structures and to compress the data has been previously explored by the HiCOO [28] format. However, such a compression comes at the cost of severe workload imbalance across blocks, due to the irregular spatial distributions of sparse data [21], [29], and it requires expensive tuning to find the best block size. In contrast, the proposed blocking technique aims to meet the requirements of memory-constrained GPUs, while at the same time generate the largest possible blocks that can efficiently utilize these throughput-oriented

[2]https://github.com/curtisseizert/CUDA-uint128

accelerators with massive parallelism.

To that end, our BLCO format first uses the *uppermost bits* from every mode of linearized indices that exceeds the target integer size to form the initial blocks, which allows the subspaces spanned by these blocks to naturally adapt to the underlying tensor space. For example, if a tensor requires 72 bits for linearized indices and the size of target integers is 64 bits, a total of 8 $(72 - 64)$ uppermost bits across all the modes are stripped from the linear indices and used as a *key* to group the non-zero elements into blocks; then, these bits are stored as metadata along with each block (b), as depicted in Figure 4.1b. This strategy does *not* require expensive tuning to create the blocks, and compared to using longer encoding lines, it reduces the memory footprint and leverages more efficient native integer (64-bit) instructions. Next, BLCO further splits the initial blocks, if needed, based on the available on-device memory to ensure that each block has no more than the maximum number of non-zero elements that can fit in the target device.

While the resulting BLCO format may still suffer from variance in the number of non-zero elements across blocks, it exposes the fine-grained parallelism within a block, due to its linearized list-based form. Specifically, it allows workload to be partitioned at the granularity of a non-zero element rather than a compressed block. Hence, a GPU hardware scheduler can automatically hide the execution and memory-access latency as well as balance the workload across threads, as long as there are enough non-zero elements to be processed in the GPU memory. In addition, our massively parallel MTTKRP algorithm (Section 5), with opportunistic conflict resolution, allows BLCO blocks to be processed independently. Therefore, once the GPU has processed a block, it will fetch the next available block to automatically occupy any available GPU resources. These properties enable our BLCO format and MTTKRP algorithm to seamlessly handle OOM tensors, in contrast to prior work, while maintaining high GPU occupancy throughout computation.

Our implementation uses device queues (SYCL queues or CUDA streams) to launch BLCO blocks, allowing the computation of active blocks to be done asynchronously with the

transmission of pending blocks. Each device queue has reserved memory, corresponding to the number of non-zero elements that can be processed at one time, which is reused across BLCO blocks assigned to the same queue. In our experiments, we use up to 8 device queues and set the maximum number of non-zero elements per block to 2^{27} to fill the GPU; these parameters allow for enough concurrency and further tuning them has negligible performance impact.

Hypersparse tensors may generate several BLCO blocks that can fit in the same memory reservation of a single device queue. Launching these blocks across multiple queues can potentially incur kernel launch overhead on some GPU architectures. To address this, we batch all BLCO blocks that can be processed by one device queue into a GPU kernel and explicitly store block mappings and element offsets at the work-group (thread block) boundaries. This additional batching information is calculated during format construction and thus creates minimal overhead during tensor computations.

CHAPTER 5

BLCO-BASED SPARSE MTTKRP

On massively parallel systems such as GPUs, atomic updates to global memory can be expensive because of the sheer number of concurrently executing threads that could conflict and the long data access latency. Therefore, prior studies focused on reducing the number of necessary atomic operations by explicitly exposing parallelism [16], [17], storing the data in a mode-specific form [15]–[18] to accumulate partial results locally, and/or keeping auxiliary mode-specific mapping/scheduling information [14], [15], [18]. However, these strategies lead to drastic performance variation across modes, substantial memory overhead for extra tensor copies, and/or complex algorithms and implementations (c.f. Section 3). Here, we describe a novel massively parallel MTTKRP algorithm that eliminates control-flow and memory-access irregularities while resolving update conflicts (RAW hazards) using an *opportunistic on-the-fly* update mechanism, which reduces atomic operations without requiring extra tensor copies or mode-specific information.

5.1 Hierarchical Conflict Resolution

Figure 5.1 illustrates our massively parallel algorithm for mode-1 MTTKRP kernel on the third-order sparse tensor from Figure 3.1a. Note that the non-zero elements are linearized and grouped according to the BLCO format (Figure 4.1b). For simplicity, we assume a work-group (thread-block) size of 12 and tile size of 6 work-items (threads) in the figure.

To eliminate control-flow and memory-access irregularities, the proposed algorithm has two phases: processing and computing. In the processing phase, steps (1) – (3), each thread is

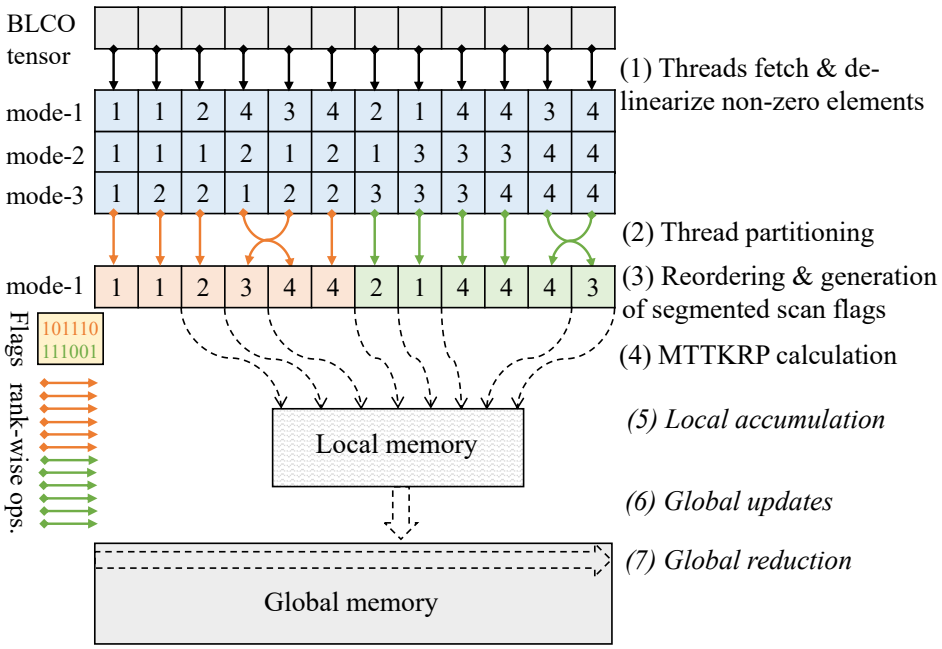


Figure 5.1: Steps (1) – (7) illustrates mode-1 MTTKRP with hierarchical conflict resolution for the BLCO tensor from Figure 4.1b. For register-based conflict resolution, step (6) is the last one, where the results are written directly, bypassing the local accumulation in step (5), to the factor matrix in global memory using atomic updates, as opposed to writing to temporary factor matrix copies. In all conflict resolution mechanisms, steps (1) – (4) are common and writing the updates happens at segment boundaries.

assigned to a non-zero element and threads collaborate to perform *on-the-fly* de-linearization and reordering of non-zero elements as well as generation of segmented scan flags. In the computing phase, steps (4) – (7), threads are reassigned to perform the rank-wise MTTKRP computations and merge conflicting updates at the register, local memory, and global memory levels. In each phase, the threads and their global memory accesses are coalesced.

5.1.1 Processing Phase

In step (1), a group of threads (work-group or thread-block) is assigned to a certain number of non-zero elements and they first collaborate so that each thread loads a linearized non-zero element (i.e., a linear index and its corresponding value) from the global memory in a coalesced manner. Each thread then proceeds to de-linearize the linear index and recover the multi-dimensional coordinates, where each coordinate can be calculated independently to expose more instruction-level parallelism. In step (2), the threads are partitioned into

multiple tiles, where a tile size is no more than the sub-group (warp) size. Next, in step (3), the threads within a tile collaborate to reorder the non-zero elements into groups (segments) according to their target mode indices (mode-1 in the example) and then generate segmented scan flags, where 1 indicates the start of a new segment. To minimize thread divergence and data access latency, we implement the reordering of non-zero elements via parallel histogram and prefix sum, using sub-group (warp-level) data exchange primitives (e.g., *shuffle* operations), and broadcast/store the segmented scan flags across threads in a low-latency register. After each thread completes the on-the-fly processing phase, the non-zero elements are stored, according to their new order, in local memory for later access.

5.1.2 Computing Phase

Once the data has been processed, we reassign the threads to calculate the rank-wise MTTKRP computations in step (4), where each thread is responsible for one or more elements along the decomposition rank (i.e., threads process the same non-zero element and perform rank-wise operations in parallel). The algorithm iterates over the non-zero elements using the segmented scan flags, so that each thread accumulates its partial results to a register as long as the target mode index remains the same. At the end of each segment, i.e., when the index changes, each thread writes the accumulated result to the stash (a software-controlled cache) in local memory (shown as "step (5)" in the figure). In step (6), when all the non-zero elements have been processed, the results are copied from the stash (local memory) to one of the multiple copies of the factor matrix in global memory. By using multiple copies of the factor matrix, we can minimize the probability of conflict when multiple thread blocks are copying their results back to global memory at the same time. Finally, in step (7), the multiple factor matrix copies are merged in global memory to produce the final result.

5.2 Register-based Conflict Resolution

Our register-based conflict resolution is a subset of the hierarchical algorithm, discussed above. It bypasses the local memory entirely and writes the accumulated result in registers

to global memory, without the need for a final global reduction. Specifically, after each thread calculates and accumulates the result in its register and reaches a segment boundary, it bypasses step (5) and completes execution at step (6) after writing the updates to the final factor matrix, as opposed to one of its copies, using atomic operations.

5.3 Adaptation Heuristic

The proposed synchronization mechanisms use different number of atomic operations to perform MTTKRP. Register-based conflict resolution requires more atomic operations, as updates at each segment boundary need atomic add operations to the factor matrix in global memory. Hierarchical conflict resolution uses fewer atomic operations, as updates at each segment boundary are copied first to the local-memory stash, and then atomic operations are used only at the end of the work-group (thread-block) execution to write the accumulated result in the stash to the factor matrix in global memory. Furthermore, multiple copies of the factor matrix can be used to reduce the probability of an update conflict, at the cost of a final global reduction.

We propose a simple heuristic for selecting the best conflict resolution mechanism based on the characteristics of target modes and GPU devices. In modern GPUs [30], [31], the execution units (EUs) are aggregated into subslices or streaming multi-processors (SMs). Multiple subslices are grouped into a GPU slice or a graphics processing cluster (GPC). Our heuristic selects the hierarchical conflict resolution, when the target mode length is less than the number of subslices (SMs). At this mode length, the contention from atomic operations to global memory is severe, and using a local-memory stash and factor matrix copies (one copy for each slice or GPC) alleviates this contention. For all other cases, we use the register-based conflict resolution.

CHAPTER 6

EXPERIMENTS

We evaluate the proposed sparse MTTKRP and BLCO format[1] using a representative set of in-memory and out-of-memory tensors with different characteristics. We compare our performance to the state-of-the-art sparse tensor frameworks for massively parallel GPUs, namely, MM-CSF [17], GenTen [14], and F-COO [15]. While the other frameworks only support tensors that can fit in the device memory, BLCO can process both in- and out-of-memory tensors and delivers superior performance across all in-memory tensors.

6.1 Evaluation Setup

6.1.1 Test Platform

We conduct the experiments on the latest Intel discrete GPU with a single-tile (denoted *Intel Device1*) and two NVIDIA GPUs from the most recent micro-architecture generations: A100 (*Ampere*) and V100 (*Volta*). For the format generation, we use a dual-socket AMD Epyc 7662 CPU system and employ 128 threads. Our prototype is implemented in Data Parallel C++ (DPC++) [32] as well as CUDA [33]. Since the current sparse tensor frameworks lack portability across GPU architectures, we ported the state-of-the-art MM-CSF to DPC++ using the Intel DPC++ Compatibility Tool [34]. Due to confidentiality requirements, Table 6.1 summarizes the publicly available specifications of the target hardware and software environment.

[1] Available at https://github.com/jeewhanchoi/blocked-linearized-coordinate

Table 6.1: Hardware and software setup.

	CPU	GPU	
Model	AMD EPYC 7662	NVIDIA A100	NVIDIA V100
μ-arch	Zen 2	Ampere	Volta
Frequency	3.3 GHz	1.41 GHz	1.38 GHz
Cores	64 ($\times$2 sockets)	108 (SM[1]) 6912 (CC[2])	80 (SM) 5120 (CC)
Caches	2MB L1, 32MB L2, 256MB L3	15MB L1D, 40MB L2	10MB L1D, 6MB L2
DRAM (Bandwidth)	256 GB	40 GB (1555 GB/s)	32 GB (900 GB/s)
Interconnect	PCI-e Gen 4	NVLink 3.0	NVLink 2.0
OS/Driver (version)	RHEL (8.3)	Driver (470.42.01)	Driver (440.33.01)
Compiler	gcc 9.3.0	nvcc 11.4	nvcc 11.0

[1] Streaming Multiprocessor [2] CUDA Cores

6.1.2 Data Sets

We consider 14 *real-world* tensor data sets from the FROSTT [35] and HaTen2 [36] open-source repositories that cover a wide range of tensor properties and sparsity structures—number of modes, mode lengths, number of non-zero elements, and density. Table 6.2 lists the sparse tensor data sets used for evaluation, ordered by increasing number of non-zero elements (NNZs). The large-scale tensors with billions of non-zero elements (namely, Amazon, Patents, and Reddit) are considered out-of-memory as they fail to execute on current tensor decomposition frameworks, which need to keep the entire tensor and factor matrices in device memory, producing memory allocation errors on target GPUs.

6.1.3 Configurations

The experiments use a decomposition rank of 32 for MTTKRP, as per prior work [17]. Using double-precision values and 64-bit integers, we report the performance as an average over 25 iterations. We tune each sparse tensor framework to the best of our abilities. For the state-of-the-art MM-CSF, we exhaustively tune the number of warps per fiber and the thread-block size and report the best execution time. While BLCO can benefit from tuning,

Table 6.2: The sparse tensor data sets used for evaluation, ordered by the number of non-zero elements.

Tensor	Dimensions	NNZs	Density
NIPS	$2.5K \times 2.9K \times 14K \times 17$	$3.1M$	1.8×10^{-06}
Uber	$183 \times 24 \times 1.1K \times 1.7K$	$3.3M$	3.8×10^{-04}
Chicago	$6.2K \times 24 \times 77 \times 32$	$5.3M$	1.5×10^{-02}
Vast-2015	$165.4K \times 11.4K \times 2$	$26M$	7.8×10^{-07}
DARPA	$22.5K \times 22.5K \times 23.8M$	$28.4M$	2.4×10^{-09}
Enron	$6K \times 5.7K \times 244.3K \times 1.2K$	$54.2M$	5.5×10^{-09}
NELL-2	$12.1K \times 9.2K \times 28.8K$	$76.9M$	2.4×10^{-05}
FB-M	$23.3M \times 23.3M \times 166$	$99.6M$	1.1×10^{-09}
Flickr	$319.7K \times 28.2M \times 1.6M \times 731$	$112.9M$	1.1×10^{-14}
Delicious	$532.9K \times 17.3M \times 2.5M \times 1.4K$	$140.1M$	4.3×10^{-15}
NELL-1	$2.9M \times 2.1M \times 25.5M$	$143.6M$	9.1×10^{-13}
Amazon	$4.8M \times 1.8M \times 1.8M$	$1.7B$	1.1×10^{-10}
Patents	$46 \times 239.2K \times 239.2K$	$3.6B$	1.4×10^{-03}
Reddit	$8.2M \times 177K \times 8.1M$	$4.7B$	4.0×10^{-10}

we use our adaptation heuristic (Section 5.3) and set the thread-coarsening factor (NNZs per thread) for the hierarchical conflict resolution mechanism (Section 5.1) to 4 and 2 on Intel and NVIDIA GPUs, respectively.

6.2 Comparison Against TD Frameworks

We first compare our BLCO-based MTTKRP against the other popular sparse tensor decomposition frameworks. Figure 6.1 shows the MTTKRP execution time for all modes, across the GPU frameworks, normalized by the execution time of the state-of-the-art MM-CSF. The results demonstrate that BLCO consistently outperforms all other frameworks, achieving a geometric mean speedup between 2.12× and 2.6× over MM-CSF across the different GPU devices. For the other CUDA frameworks, GenTen has comparable performance to MM-CSF, outperforming MM-CSF on six out of 11 data sets, whereas F-COO has lower performance on average compared to MM-CSF, especially on the V100 GPU. The missing data points for F-COO is due to its limited support for higher-order tensors (i.e., 3-D only) and "segfault" errors.

Our performance analysis indicates that the performance of MM-CSF (as well as Gen-

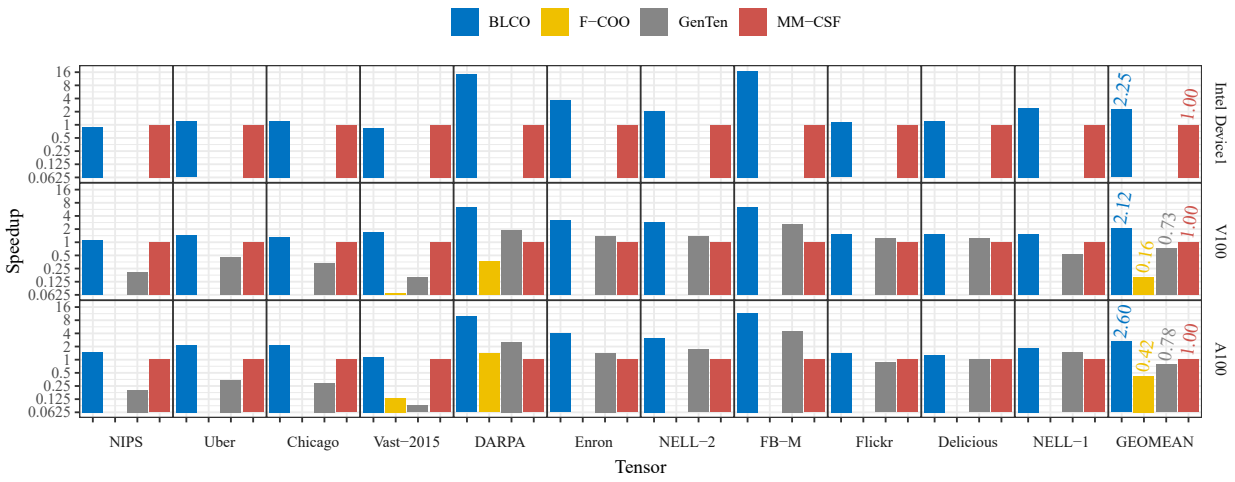


Figure 6.1: Comparison of BLCO-based MTTKRP against popular GPU frameworks on the data sets that fit in GPU memory. Each bar represents the speedup obtained against MM-CSF for computing MTTKRP on all modes. The right-most group of bars show the geometric mean speedup.

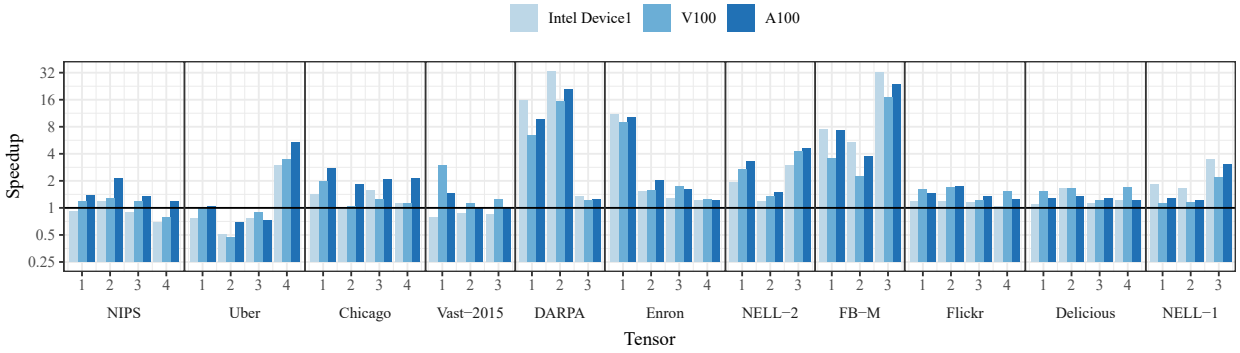


Figure 6.2: The per-mode speedup of BLCO-based MTTKRP against MM-CSF across *every* tensor mode for the data sets that fit in GPU memory.

Ten) can substantially decrease with higher synchronization cost, which leads to lower average performance than BLCO on the GPU devices with more expensive synchronization. Since MM-CSF reorders non-zero elements to increase tensor compression, its performance is sensitive to the number of non-zero elements per fiber. On large-scale data sets with low fiber density, such as DARPA, Enron, and FB-M, MM-CSF has lower compression, leading to significant performance degradation compared to BLCO.

6.3 Comparison Against State of the Art

For a comprehensive evaluation against the state-of-the-art MM-CSF, Figure 6.2 demonstrates the speedup achieved by our BLCO-based MTTKRP for every mode of the tensors that can fit in the device memory of target GPUs. The results demonstrate that our BLCO format achieves better or comparable performance to MM-CSF for every mode (up to 33.35× speedup) across all data sets, except Uber and NIPS. These data sets are not only small, but they also have exceptionally short modes, allowing the data to fit in cache; thus, the higher compression achieved by MM-CSF, due to its mode-specific nature (c.f. Table 6.3), translates to better performance. Yet, such a mode-specific compression leads to substantial performance variations across different modes, as shown in Figure 1.1, and as a result, BLCO still outperforms MM-CSF for all-mode MTTKRP (c.f. Figure 6.1).

6.4 Memory Traffic Analysis

Since sparse tensor decomposition is a memory-bound workload, its performance is largely limited by the data volume and the effective memory throughput. Hence, we provide detailed analysis of these memory metrics across both in- and out-of-memory tensors.

6.4.1 In-Memory Tensors

Table 6.3 details the memory metrics of BLCO-based MTTKRP compared to MM-CSF on the A100 GPU. The metrics are collected using the Nsight Compute profiler [37]. The

memory analysis shows that MM-CSF achieves higher compression than BLCO thanks to its tree-like data structure, and the total volume of data fetched from memory (as shown in column "Vol") is lower in most cases. However, due to the irregular memory access and expensive synchronization associated with traversing a tree-like tensor representation, MM-CSF under-utilizes the memory system compared to BLCO and has lower memory throughput (as shown in column "TP"). In addition, both the memory volume and throughput of MM-CSF vary significantly across modes because of its mode-specific traversal and processing of tensors, which leads to substantial performance variations (c.f. Figure 1.1). In contrast, while BLCO requires more data volume because of its mode-agnostic form, it achieves higher memory throughput by eliminating memory-access irregularities, exploiting data locality, and merging conflicting updates across threads in low-latency registers and memories. Thereby, BLCO fetches/writes more data from/to higher levels of the memory hierarchy in a coalesced way, leading to improved performance by up to an order of magnitude compared to MM-CSF.

Table 6.3: Comparison of the memory related metrics between BLCO and MM-CSF for MTTKRP on the A100 GPU.

Data Set	Format	n	Vol[1]	TP[2]	Data Set	Format	n	Vol[1]	TP[2]
Uber	BLCO	1	2.78	3.60	Enron	BLCO	1	44.82	4.11
		2	2.75	3.61			2	46.23	4.62
		3	2.75	3.53			3	47.88	4.92
		4	2.73	2.77			4	47.22	4.70
	MM-CSF	1	1.68	1.68		MM-CSF	1	41.39	0.31
		2	1.33	2.03			2	62.83	3.16
		3	1.33	1.93			3	37.15	2.29
		4	2.12	0.32			4	37.05	3.01
Vast-2015	BLCO	1	16.91	3.92	NELL-1	BLCO	1	107.5	2.44
		2	16.73	3.77			2	104.5	2.32
		3	13.92	2.90			3	110.7	2.39
	MM-CSF	1	9.19	1.19		MM-CSF	1	123.1	2.21
		2	8.36	1.57			2	118.5	2.19
		3	8.36	1.45			3	122.1	0.86

[1] Memory volume in GB, measured by *l1tex__t_bytes.sum* in Nsight Compute [37]

[2] Memory throughput in TB/s, calculated by (Vol / total execution time)

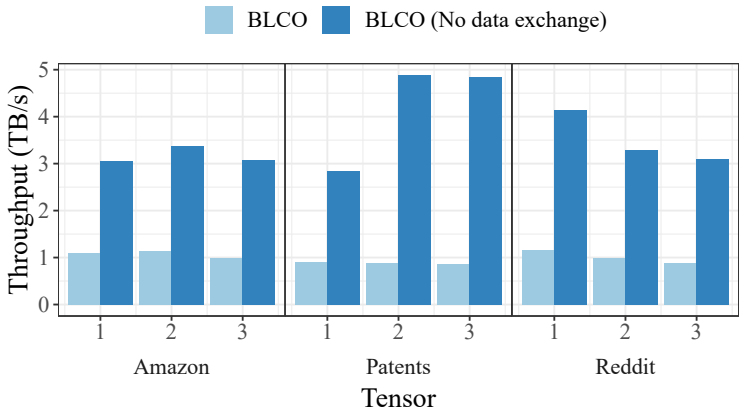


Figure 6.3: The memory throughput of BLCO-based MTTKRP (with and without host-device data exchange) for out-of-memory tensors across every mode on the A100 GPU.

6.4.2 Out-of-Memory Tensors

Figure 6.3 demonstrates the memory throughput of BLCO-based MTTKRP for out-of-memory tensors (Amazon, Patents, and Reddit) on the A100 GPU. Since no other GPU framework supports these tensors, a direct comparison is not possible; instead, we report the overall throughput as well as the throughput without host-device data exchange (in-memory throughput) of our BLCO-based MTTKRP, as measured using the Nsight Systems profiler [38]. The overall throughput is measured based on the total execution time (both MTTKRP computations and host-device data transfers), while the in-memory throughput only includes MTTKRP computations. Without the overhead of host-device communication resulting from the limited GPU memory, the in-memory throughput is on par with the performance observed for the tensors that can fit in the GPU (c.f. Table 6.3). However, the overall throughput is lower due to the limited bandwidth of the host-device interconnect compared to the device memory bandwidth. While BLCO achieves perfect overlap between host-device transfers and MTTKRP computations, the communication overhead still dominates the execution time, leading to lower overall throughput (57%–75% of the memory bandwidth). This result demonstrates that our framework handles out-of-memory tensors as efficiently as in-memory tensors, but the overall performance for out-of-memory tensors is limited by the bandwidth of the host-device interconnect.

6.5 Format Construction

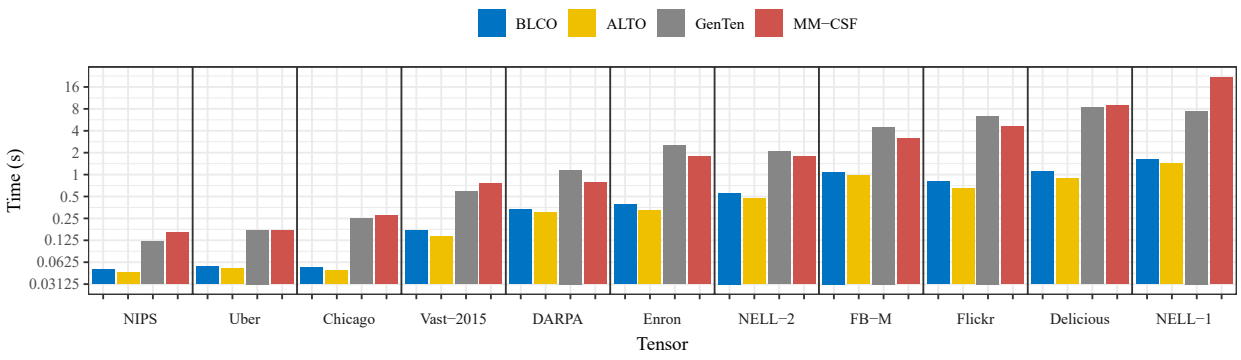


Figure 6.4: Comparison of the BLCO construction/generation cost against popular GPU formats as well as the CPU-based ALTO format on the data sets that fit in GPU memory.

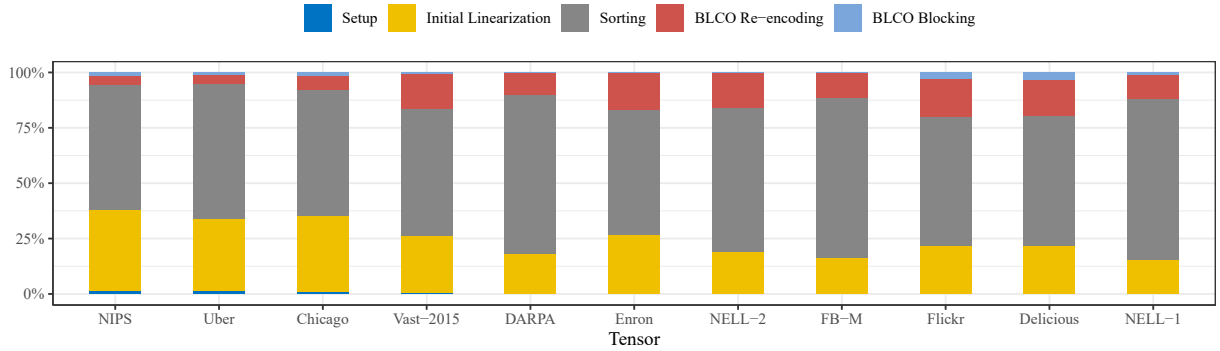


Figure 6.5: Breakdown of the BLCO construction cost for the data sets that fit in GPU memory. Compared to ALTO, BLCO requires additional blocking and re-encoding to enable efficient execution on GPUs.

Figure 6.4 shows the generation time of the GPU sparse formats, namely, BLCO, GenTen, and MM-CSF, as well as the CPU-based ALTO format. Furthermore, Figure 6.5 details the run-time distribution across the different format generation stages of BLCO. The tensor formats are generated from raw data, in the COO representation, on the host CPU listed in Table 6.1. By adopting a mode-agnostic linearized form rather than a multi-dimensional representation, BLCO significantly decreases the format generation cost, which is typically dominated by sorting and clustering non-zero elements. Additionally, BLCO does not require extra mode-specific mapping or scheduling information for reducing synchronization. As a

result, BLCO is several times (up to 13.6×) cheaper to generate than the state-of-the-art MM-CSF format. On the A100 GPU, BLCO needs approximately 12 full (all-mode) MTTKRP iterations on average to amortize its format generation time, while the other GPU formats require up to an order of magnitude more iterations to amortize their construction cost.

Compared to ALTO, BLCO enables efficient execution of tensor operations on massively parallel architectures by (i) re-encoding linearized indices for fast decoding on GPUs, and (ii) blocking the tensor into smaller chunks that fit in limited device memory and require at most 64 bits for indices. However, these additional stages typically consume less than 25% of the overall format construction cost, as shown in Figure 6.5.

CHAPTER 7

RELATED WORK

Optimizing sparse tensor decomposition and MTTKRP operations has been the subject of several prior studies, which propose various sparse tensor formats along with parallel algorithms to process and analyze the multi-modal data on CPU- and GPU-based hardware architectures. List-based formats, such as F-COO [15], GenTen [14], and TB-COO [18], explicitly store the multi-dimensional coordinates of each non-zero element. To reduce atomic operations, these formats keep multiple mode-specific copies of the tensor and/or extra scheduling information, which substantially increases their memory footprint. Tree-based formats, including CSF [12], [24], B-CSF [16], and MM-CSF [17], extend the compressed sparse row (CSR) matrix format to higher-order tensors. While these mode-specific formats can compress the sparse data, they have load imbalance issues and significant performance variation across various modes of execution.

Block-based formats (e.g., HiCOO [28]) cluster non-zero elements to compress the sparse tensor and to exploit data locality. However, as the number of tensor modes and sparsity increase, the majority of HiCOO blocks consist of a few non-zero elements, leading to more memory usage than COO [21], [28]. In addition, the irregular spatial distributions of sparse data result in severe load imbalance and synchronization issues across HiCOO blocks [21], [28], and as a result, it has no GPU implementation [13]. In contrast, BLCO addresses these limitations by generating coarse-grained blocks that fit in the GPU memory, while efficiently utilizing the GPU resources by encoding the non-zero elements within each block in a fine-grained linearized form, amenable to caching and parallel execution.

CPU-based tensor linearization approaches, such as the LCO [25] and ALTO [21] formats, compress the tensor by mapping the multi-dimensional coordinates of a non-zero element into a single index. In particular, ALTO enables high-performance tensor operations by (i) leveraging the efficient bit-level scatter/gather instructions and large integers on CPUs, and (ii)using adaptive atomic- and reduction-based conflict resolution algorithms, tailored for architectures with coarse-grained cores/threads. However, massively parallel GPUs lack native support for the bit manipulation instructions and large integer arithmetic that are needed for efficient processing of prior linearized formats. Additionally, GPUs suffer from limited device memory and require sophisticated conflict resolution, due to their massive fine-grained parallelism and substantial synchronization overhead. BLCO directly addresses these issues to allow efficient execution of large-scale tensors on accelerators.

Segmented scan and reduction [22], [23] have been used to reduce the synchronization cost of sparse workloads [15], [18], [39]–[41] on parallel architectures. Prior studies apply these primitives to *mode-specific* formats with delineated and/or sorted groups of non-zero elements according to the target mode. In contrast, we devise an opportunistic algorithm that leverages the *mode-agnostic* BLCO format to reduce synchronization by discovering conflicts and performing segmented scan on-the-fly and without needing sorted non-zero elements or keeping extra scheduling information. In addition, our novel conflict resolution algorithm eliminates control-flow and memory-access irregularities by specializing threads to perform different operations at each execution phase.

The TACO compiler [42] automatically generates various sparse matrix and tensor algebra kernels, including sparse MTTKRP. However, prior work showed that hand-optimized implementations of CSF-based formats (namely, B-CSF) still outperform the auto-generated TACO code on GPU architectures, even with extensive auto-scheduling and optimization [43].

A wealth of work perform MTTKRP on distributed-memory platforms using MPI [44]–[48], or the MapReduce [49], [50] framework. Other studies [51]–[53] explore format selection based on machine learning models to efficiently leverage existing sparse formats.

CHAPTER 8

CONCLUSION

Tensors and tensor decomposition are growing increasingly popular in the advent of big data. Efficient and scalable implementations of tensor algorithms are paramount in addressing the varying sizes of real world tensors. To enable high-performance sparse tensor decomposition on massively parallel GPU architectures, this work proposes the BLCO format. In contrast to prior approaches, which have been restricted to in-memory tensors, BLCO allows efficient processing of both in-memory and out-of-memory tensors. By discovering and merging conflicting updates on-the-fly, without any mode-specific information or ordering of non-zero elements, our BLCO-based MTTKRP demonstrated substantial speedup (up to 33.35×) over the state-of-the-art MM-CSF. We leave for future work the consideration of other tensor decomposition algorithms such as CP-APR and the Tucker decomposition, and support for even larger tensors through a distributed or heterogeneous GPU implementation.

References

[1] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *2008 Eighth IEEE International Conference on Data Mining*, 2008, pp. 363–372. DOI: 10.1109/ICDM.2008.89.

[2] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Tensors for data mining and data fusion: Models, applications, and scalable algorithms," vol. 8, no. 2, Oct. 2016, ISSN: 2157-6904. DOI: 10.1145/2915921. [Online]. Available: https://doi.org/10.1145/2915921.

[3] A. Rettinger, H. Wermser, Y. Huang, and V. Tresp, "Context-aware tensor decomposition for relation prediction in social networks," *Social Network Analysis and Mining*, vol. 2, no. 4, pp. 373–385, 2012. DOI: 10.1007/s13278-012-0069-5. [Online]. Available: https://doi.org/10.1007/s13278-012-0069-5.

[4] S. Fernandes, H. Fanaee-T, and J. Gama, "Evolving social networks analysis via tensor decompositions: From global event detection towards local pattern discovery and specification," in *Discovery Science*, P. Kralj Novak, T. Šmuc, and S. Džeroski, Eds., Cham: Springer International Publishing, 2019, pp. 385–395, ISBN: 978-3-030-33778-0.

[5] H. Fanaee-T and J. Gama, "Tensor-based anomaly detection: An interdisciplinary survey," *Knowl. Based Syst.*, vol. 98, pp. 130–147, 2016. DOI: 10.1016/j.knosys.2016.01.027. [Online]. Available: https://doi.org/10.1016/j.knosys.2016.01.027.

[6] D. Bruns-Smith, M. M. Baskaran, J. Ezick, T. Henretty, and R. Lethin, "Cyber security through multidimensional data decompositions," in *2016 Cybersecurity Symposium (CYBERSEC)*, 2016, pp. 59–67. DOI: 10.1109/CYBERSEC.2016.017.

[7] J. C. Ho, J. Ghosh, and J. Sun, "Marble: High-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization," in *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '14, New York, New York, USA: Association for Computing Machinery, 2014, pp. 115–124, ISBN: 9781450329569. DOI: 10.1145/2623330.2623658. [Online]. Available: https://doi.org/10.1145/2623330.2623658.

[8] H. He, J. Henderson, and J. C. Ho, "Distributed tensor decomposition for large scale health analytics," in *The World Wide Web Conference*, ser. WWW '19, San Francisco, CA, USA: Association for Computing Machinery, 2019, pp. 659–669, ISBN: 9781450366748. DOI: 10.1145/3308558.3313548. [Online]. Available: https://doi.org/10.1145/3308558.3313548.

[9] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell, "Toward an architecture for never-ending language learning.," in *AAAI*, vol. 5, 2010, p. 3.

[10] B. W. Bader, M. W. Berry, and M. Browne, "Discussion tracking in enron email using parafac," in *Survey of Text Mining II: Clustering, Classification, and Retrieval*, M. W. Berry and M. Castellanos, Eds. London: Springer London, 2008, pp. 147–163, ISBN: 978-1-84800-046-9. DOI: 10.1007/978-1-84800-046-9_8. [Online]. Available: https://doi.org/10.1007/978-1-84800-046-9_8.

[11] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proceedings of the 7th ACM conference on Recommender systems*, ACM, 2013, pp. 165–172.

[12] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis, "Splatt: Efficient and parallel sparse tensor-matrix multiplication," in *2015 IEEE International Parallel and Distributed Processing Symposium*, 2015, pp. 61–70. DOI: 10.1109/IPDPS.2015.27.

[13] J. Li, Y. Ma, and R. Vuduc, *ParTI! : A parallel tensor infrastructure for multicore cpus and gpus*, Last updated: Jan 2020, Oct. 2018. [Online]. Available: http://parti-project.org.

[14] E. Phipps and T. G. Kolda, "Software for Sparse Tensor Decomposition on Emerging Computing Architectures," en, *SIAM Journal on Scientific Computing*, vol. 41, no. 3, pp. C269–C290, Jan. 2019, arXiv: 1809.09175, ISSN: 1064-8275, 1095-7197. DOI: 10.1137/18M1210691. [Online]. Available: http://arxiv.org/abs/1809.09175 (visited on 01/13/2021).

[15] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A Unified Optimization Approach for Sparse Tensor Operations on GPUs," *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 47–57, Sep. 2017, arXiv: 1705.09905. DOI: 10.1109/CLUSTER.2017.75. [Online]. Available: http://arxiv.org/abs/1705.09905 (visited on 01/13/2021).

[16] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-Balanced Sparse MTTKRP on GPUs," en, *arXiv:1904.03329 [cs]*, Apr. 2019, arXiv: 1904.03329. [Online]. Available: http://arxiv.org/abs/1904.03329 (visited on 01/13/2021).

[17] I. Nisa, J. Li, A. Sukumaran-Rajam, P. S. Rawat, S. Krishnamoorthy, and P. Sadayappan, "An efficient mixed-mode representation of sparse tensors," en, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Denver Colorado: ACM, Nov. 2019, pp. 1–25, ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356216. [Online]. Available: https://dl.acm.org/doi/10.1145/3295500.3356216 (visited on 01/13/2021).

[18] M. Dun, Y. Li, H. Yang, Q. Sun, Z. Luan, and D. Qian, "An optimized tensor completion library for multiple gpus," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 417–430, ISBN: 9781450383356. DOI: 10.1145/3447818.3460692. [Online]. Available: https://doi.org/10.1145/3447818.3460692.

[19] T. G. Kolda and B. W. Bader, "Tensor Decompositions and Applications," en, *SIAM Review*, vol. 51, no. 3, pp. 455–500, Aug. 2009, ISSN: 0036-1445, 1095-7200. DOI: 10.1137/07070111X. [Online]. Available: http://epubs.siam.org/doi/10.1137/07070111X (visited on 01/13/2021).

[20] B. W. Bader and T. G. Kolda, "Efficient matlab computations with sparse and factored tensors," *SIAM Journal on Scientific Computing*, vol. 30, no. 1, pp. 205–231, 2008. DOI: 10.1137/060676489. [Online]. Available: https://doi.org/10.1137/060676489.

[21] A. E. Helal, J. Laukemann, F. Checconi, J. J. Tithi, T. Ranadive, F. Petrini, and J. Choi, "Alto: Adaptive linearized storage of sparse tensors," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '21, Virtual Event, USA: Association for Computing Machinery, 2021, pp. 404–416, ISBN: 9781450383356. DOI: 10.1145/3447818.3461703. [Online]. Available: https://doi.org/10.1145/3447818.3461703.

[22] S. Sengupta, M. Harris, M. Garland, *et al.*, "Efficient parallel scan algorithms for gpus," *NVIDIA, Santa Clara, CA, Tech. Rep. NVR-2008-003*, vol. 1, no. 1, pp. 1–17, 2008.

[23] S. Yan, G. Long, and Y. Zhang, "Streamscan: Fast scan algorithms for gpus without global barrier synchronization," in *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2013, pp. 229–238.

[24] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," en, in *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*, Austin Texas: ACM, Nov. 2015, pp. 1–7, ISBN: 978-1-4503-4001-4. DOI: 10.1145/2833179.2833183. [Online]. Available: https://dl.acm.org/doi/10.1145/2833179.2833183 (visited on 01/13/2021).

[25] A. P. Harrison and D. Joseph, "High Performance Rearrangement and Multiplication Routines for Sparse Tensor Arithmetic," *arXiv:1802.02619 [cs]*, Feb. 2018, arXiv: 1802.02619. [Online]. Available: http://arxiv.org/abs/1802.02619 (visited on 01/13/2021).

[26] G. M. Morton, "A computer oriented geodetic data base; and a new technique in file sequencing," IBM Ltd., 150 Laurier Ave., Ottawa, Ontario, Canada, Tech. Rep., 1966. [Online]. Available: https://dominoweb.draco.res.ibm.com/reports/Morton1966.pdf.

[27] J. Choi, X. Liu, S. Smith, and T. Simon, "Blocking Optimization Techniques for Sparse Tensor Computation," en, in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, Vancouver, BC: IEEE, May 2018, pp. 568–577, ISBN: 978-1-5386-4368-6. DOI: 10.1109/IPDPS.2018.00066. [Online]. Available: https://ieeexplore.ieee.org/document/8425210/ (visited on 01/31/2021).

[28] J. Li, J. Sun, and R. Vuduc, "Hicoo: Hierarchical storage of sparse tensors," ser. SC '18, Dallas, Texas: IEEE Press, 2018. DOI: 10.1109/SC.2018.00022. [Online]. Available: https://doi.org/10.1109/SC.2018.00022.

[29] J. Li, B. Uçar, Ü. V. Çatalyürek, J. Sun, K. Barker, and R. Vuduc, "Efficient and effective sparse tensor reordering," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 227–237, ISBN: 9781450360791. DOI: 10.1145/3330345.3330366. [Online]. Available: https://doi.org/10.1145/3330345.3330366.

[30] D. Blythe, "The xe gpu architecture," in *2020 IEEE Hot Chips 32 Symposium (HCS)*, IEEE Computer Society, 2020, pp. 1–27.

[31] "Nvidia a100 80gb pcie gpu," Nvidia Corp., Tech. Rep., Jun. 2021.

[32] J. Reinders, B. Ashbaugh, J. Brodman, M. Kinsner, J. Pennycook, and X. Tian, *Data Parallel C++: Mastering DPC++ for Programming of Heterogeneous Systems using C++ and SYCL*. Springer Nature, 2021.

[33] D. B. Kirk, "Nvidia cuda software and gpu parallel computing architecture," 2006.

[34] *Intel DPC++ Compatibility Tool*, https://software.intel.com/en-us/get-started-with-intel-dpcpp-compatibility-tool, Online; accessed 14 May 2022, 2022.

[35] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. "FROSTT: The formidable repository of open sparse tensors and tools." (2017), [Online]. Available: http://frostt.io/.

[36] I. Jeon, E. E. Papalexakis, U. Kang, and C. Faloutsos, "HaTen2: Billion-scale tensor decompositions," en, in *2015 IEEE 31st International Conference on Data Engineering*, Seoul, South Korea: IEEE, Apr. 2015, pp. 1047–1058, ISBN: 978-1-4799-7964-6. DOI: 10.1109/ICDE.2015.7113355. [Online]. Available: http://ieeexplore.ieee.org/document/7113355/ (visited on 01/31/2021).

[37] *Nsight compute command line interface*, https://docs.nvidia.com/nsight-compute/pdf/NsightComputeCli.pdf, Online; accessed 14 May 2022, 2022.

[38] *Nsight systems user guide*, https://docs.nvidia.com/nsight-systems/pdf/UserGuide.pdf, Online; accessed 14 May 2022, 2022.

[39] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*, 2009, pp. 1–11.

[40] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2010, pp. 127–136.

[41] S. Yan, C. Li, Y. Zhang, and H. Zhou, "Yaspmv: Yet another spmv framework on gpus," in *Proceedings of the 19th ACM SIGPLAN symposium on Principles and practice of parallel programming*, 2014, pp. 107–118.

[42] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe, "The Tensor Algebra Compiler," *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–29, 2017.

[43] R. Senanayake, C. Hong, Z. Wang, A. Wilson, S. Chou, S. Kamil, S. Amarasinghe, and F. Kjolstad, "A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–30, 2020.

[44] J. H. Choi and S. V. N. Vishwanathan, "DFacTo: Distributed Factorization of Tensors," en, *arXiv:1406.4519 [stat]*, Jun. 2014, arXiv: 1406.4519. [Online]. Available: `http://arxiv.org/abs/1406.4519` (visited on 01/31/2021).

[45] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," en, in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, Austin Texas: ACM, Nov. 2015, pp. 1–11, ISBN: 978-1-4503-3723-6. DOI: `10.1145/2807591.2807624`. [Online]. Available: `https://dl.acm.org/doi/10.1145/2807591.2807624` (visited on 01/13/2021).

[46] E. Solomonik, D. Matthews, J. Hammond, and J. Demmel, "Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, IEEE, 2013, pp. 813–824.

[47] S. Smith and G. Karypis, "A medium-grained algorithm for sparse tensor factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2016, pp. 902–911. DOI: `10.1109/IPDPS.2016.113`.

[48] K. Shin and U. Kang, "Distributed methods for high-dimensional and large-scale tensor factorization," in *2014 IEEE International Conference on Data Mining*, IEEE, 2014, pp. 989–994.

[49] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos, "GigaTensor: Scaling tensor analysis up by 100 times - algorithms and discoveries," en, in *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining - KDD '12*, Beijing, China: ACM Press, 2012, p. 316, ISBN: 978-1-4503-1462-6. DOI: `10.1145/2339530.2339583`. [Online]. Available: `http://dl.acm.org/citation.cfm?doid=2339530.2339583` (visited on 01/31/2021).

[50] Z. Blanco, B. Liu, and M. M. Dehnavi, "CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms," en, in *Proceedings of the 47th International Conference on Parallel Processing*, Eugene OR USA: ACM, Aug. 2018, pp. 1–10, ISBN: 978-1-4503-6510-9. DOI: `10.1145/3225058.3225133`. [Online]. Available: `https://dl.acm.org/doi/10.1145/3225058.3225133` (visited on 01/31/2021).

[51] Y. Zhao, J. Li, C. Liao, and X. Shen, "Bridging the gap between deep learning and sparse matrix format selection," in *Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '18, Vienna, Austria: Association for Computing Machinery, 2018, pp. 94–108, ISBN: 9781450349826. DOI: `10.1145/3178487.3178495`. [Online]. Available: `https://doi.org/10.1145/3178487.3178495`.

[52] Z. Xie, G. Tan, W. Liu, and N. Sun, "Ia-spgemm: An input-aware auto-tuning framework for parallel sparse matrix-matrix multiplication," in *Proceedings of the ACM International Conference on Supercomputing*, ser. ICS '19, Phoenix, Arizona: Association for Computing Machinery, 2019, pp. 94–105, ISBN: 9781450360791. DOI: 10.1145/3330345.3330354. [Online]. Available: https://doi.org/10.1145/3330345.3330354.

[53] Q. Sun, Y. Liu, M. Dun, H. Yang, Z. Luan, L. Gan, G. Yang, and D. Qian, "Sptfs: Sparse tensor format selection for mttkrp via deep learning," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '20, Atlanta, Georgia: IEEE Press, 2020, ISBN: 9781728199986. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/3433701.3433724.