RESEARCH-ARTICLE

Accelerated Constrained Sparse Tensor Factorization on Massively Parallel Architectures

YONGSEOK SOH, University of Oregon, Eugene, OR, United States

RAMAKRISHNAN KANNAN, Oak Ridge National Laboratory, Oak Ridge, TN, United States

PIYUSH SAO, Oak Ridge National Laboratory, Oak Ridge, TN, United States

JEEWHAN CHOI, University of Oregon, Eugene, OR, United States

Open Access Support provided by:

University of Oregon

Oak Ridge National Laboratory

Published: 12 August 2024

ICPP '24: the 53rd International Conference on Parallel Processing
August 12 - 15, 2024
Gotland, Sweden

ICPP '24: Proceedings of the 53rd International Conference on Parallel Processing (August 2024)
https://doi.org/10.1145/3673038.3673128
ISBN: 9798400717932

Accelerating Constrained Sparse Tensor Factorization on Massively Parallel Architectures

Yongseok Soh
ysoh@uoregon.edu
University of Oregon
Eugene, OR, USA

Ramakrishnan Kannan*
ramki@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Piyush Sao*
saopk@ornl.gov
Oak Ridge National Laboratory
Oak Ridge, TN, USA

Jee Choi
jeec@uoregon.edu
University of Oregon
Eugene, OR, USA

ABSTRACT

This study presents the first *constrained* sparse tensor factorization (cSTF) framework that *optimizes* and *fully offloads* computation to massively parallel GPU architectures, and the first *performance characterization* of cSTF on GPU architectures. In contrast to prior work on tensor factorization, where the matricized tensor times Khatri-Rao product (MTTKRP) is the primary performance bottleneck, our systematic analysis of the cSTF algorithm on GPUs reveals that adding constraints creates an additional bottleneck in the update operation for many real-world sparse tensors. While executing the update operation on the GPU brings significant speedup over its CPU counterpart, it remains a significant bottleneck. To further accelerate the update operation, we propose cuADMM, a new update algorithm that leverages algorithmic and code optimization strategies to minimize both computation and data movement on GPUs. As a result, our framework delivers significantly improved performance compared to prior state-of-the-art. On 10 real-world sparse tensors, our framework achieves geometric mean speedup of 5.1× (max 41.59×) and 7.01× (max 58.05×) on the NIVIDA A100 and H100 GPUs, respectively, over the state-of-the-art SPLATT library running on a 26-core Intel Ice Lake Xeon CPU.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; • **Mathematics of computing** → **Mathematical software performance**.

ICPP '24, August 12–15, 2024, Gotland, Sweden

ACM ISBN 979-8-4007-1793-2/24/08
https://doi.org/10.1145/3673038.3673128

KEYWORDS

constrained tensor factorization, accelerated sparse tensor Factorization, high performance, algorithm

ACM Reference Format:
Yongseok Soh, Ramakrishnan Kannan, Piyush Sao, and Jee Choi. 2024. Accelerating Constrained Sparse Tensor Factorization on Massively Parallel Architectures. In *The 53rd International Conference on Parallel Processing (ICPP '24), August 12–15, 2024, Gotland, Sweden.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3673038.3673128

1 INTRODUCTION

Sparse tensor factorization (STF) is an important technique in unsupervised learning for extracting low-dimensional latent features from high-dimensional data, which are typically represented as sparse tensors. STF is becoming increasingly critical in signal analysis[27], anomaly detection[15, 34], cybersecurity[6, 12], and trend analysis[35], as it enables researchers in different domains to extract valuable insights from large, complex, and multi-dimensional datasets. Utilizing sparse tensor analysis methods in practical applications involves unique computational challenges.

The first challenge comes from the highly sparse nature of real-world tensors, which leads to irregular memory access, workload imbalance, and synchronization overhead. These properties make computation *involving sparse tensors* particularly challenging to execute efficiently on massively parallel GPU architectures. This is demonstrated by the matricized tensor times Khatri-Rao product (MTTKRP) operation, which is the primary performance bottlenecks in most *unconstrained* STF algorithms for both CPUs[7, 32] and GPUs[20, 23].

The second challenge comes from constraints inherent in many real-world tensors. For instance, multi-dimensional sensory input data often comprises of *non-negative* values. Therefore, imposing such constraints on the factors typically results in a more interpretable output for domain scientists, making it a crucial aspect of tensor analysis. Various update methods are *used on the dense factor matrices* to impose constraints, and several optimized libraries exist for CPUs[5, 28]. However, as far as we are aware, *no studies have been conducted on optimizing update methods for constrained sparse tensor factorization (cSTF) on massively parallel GPU architectures.*

As such, we draw on prior studies on CPUs for constrained tensor factorization to estimate the performance characteristics of cSTF and the impact of the update methods on the overall performance on

GPUs. Figure 1 (under **DenseTF**) shows a breakdown of execution time for *constrained dense* tensor factorization using three different update methods on a synthetic $400 \times 200 \times 100 \times 50$ tensor with a factorization rank of $R = 32$, obtained using the PLANC library [5]. For dense tensors, the MTTKRP operation dominates the overall execution time. We can attribute this to the proportionally larger size of the dense tensor ($400 \cdot 200 \cdot 100 \cdot 50$ elements) compared to its factor matrices ($(400 + 200 + 100 + 50) \cdot R$ elements in total).

In contrast, real-world sparse tensors demonstrate very high levels of sparsity, where the size of the tensors (i.e., number of non-zero elements) is comparable to the size of the factor matrices (see Table 2 for examples). Therefore, we expect *the update process to be at least as expensive as the MTTKRP operation.* To test our hypothesis, we modified the PLANC library to execute *constrained sparse* tensor factorization on the `Delicious` sparse tensor using the same update method and factorization rank ($R = 32$). Figure 1 (under **SparseTF**) shows a breakdown of execution time. As expected, we see that the update process is *significantly more expensive* than the MTTKRP operation.

From this preliminary study, we hypothesize that offloading the entire cSTF algorithm to the GPU will significantly improve performance compared to CPU-based implementations. The update process operates over the *dense* factor matrices with regular memory access, taking full advantage of the higher memory bandwidth and compute performance of modern GPUs. Offloading the entire end-to-end cSTF computation to the GPU eliminates the need to transfer data between host and GPU over the slower PCIe or NVLink interconnect. While the MTTKRP operation on the sparse tensor is also a significant performance bottleneck for cSTF, existing GPU work [20, 23] can be leveraged to further reduce the end-to-end execution time.

In summary, we make the following contributions:

(1) We present the first GPU-accelerated framework for end-to-end constrained sparse tensor factorization (cSTF). We integrate the alternating optimization with alternating direction method of multipliers (AO-ADMM) update method, a fast and robust algorithm for applying constraints, to fully operate on the GPU.
(2) We analyze performance bottlenecks of the ADMM algorithm on GPUs and propose cuADMM, an optimized ADMM algorithm that eliminates redundant computation and data movement. cuADMM achieves a 1.8× geometric mean speedup over generic ADMM on an NVIDIA H100 GPU across 10 real-world sparse tensors.
(3) By offloading cSTF entirely to the GPU, our framework outperforms state-of-the-art CPU-based libraries, such as SPLATT [32]. We evaluate our framework on 10 real-world tensors and show that it achieves a geometric mean speedup (and maximum speedup) of 5.1× (41.59×) and 7.01× (58.05×) on the latest NVIDIA A100 and H100 GPUs, respectively.
(4) We also demonstrate our framework's flexibility by incorporating the state-of-the-art sparse MTTKRP GPU kernel [20] and two additional GPU-based non-negativity constraint algorithms, HALS and MU, which achieve significant speedups over their CPU counterparts.

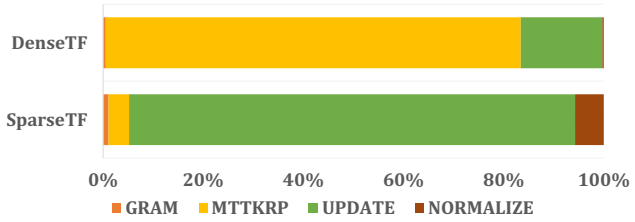

Figure 1: Execution time breakdown of constrained tensor factorization for dense (DenseTF) and sparse (SparseTF) tensors using the ADMM update methods from the PLANC library [5]. For dense tensors, MTTKRP dominates the overall execution time. However, for sparse tensors, the ADMM update process for Non-negative CP factorization takes significantly longer than MTTKRP. The dense tensor is a synthetic $400 \times 200 \times 100 \times 50$ tensor, while the sparse tensor is the `Delicious` tensor (Table 2). A factorization rank of $R = 32$ is used for both.

2 BACKGROUND

We present a brief overview of tensor notations and factorization methods. For an in-depth discussion on tensor factorization algorithms and applications, we refer the readers to the seminal work by Kolda and Bader [13].

2.1 Tensor Notations and Operations

Tensors are multi-dimensional generalization of matrices and arrays. A N-mode (or N-order) tensor is an array with N modes or dimensions. The following notations are used in this paper:

(1) *Scalars* are written with lowercase letters (e.g., a).
(2) *Vectors* are written with bold lowercase letters (e.g., $\mathbf{a} \in \mathbb{R}^I$).
(3) *Matrices* are written with bold capital letters (e.g., $\mathbf{A} \in \mathbb{R}^{I \times J}$). The $(i, j)^{th}$ entry of $\mathbf{A} \in \mathbb{R}^{I \times J}$ is denoted $a_{i,j}$.
(4) *Higher-order tensors* are written with Euler script letters (e.g., $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$). The $(i_1, \ldots, i_N)^{th}$ entry of the N-order tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$ is denoted $x_{i_1,\ldots,i_N}$.
(5) *Fibers* are the analogue of matrix rows/columns for higher-order tensors. A mode-n fiber of a tensor $\mathcal{X}$ is any vector formed by fixing all indices of $\mathcal{X}$, *except* the n^{th} index (e.g., a matrix column is defined by fixing the second index, and is therefore a mode-1 fiber). We denote fibers using a colon for the variable index (e.g., the j^{th} column of a matrix A is denoted by $\mathbf{a}_{:,j}$).
(6) *Hadamard* product is an element-wise product between two vectors or matrices, and is denoted by "$*$".
(7) *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

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

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

2.2 Canonical Polyadic Decomposition

The canonical polyadic decomposition (CPD) is a widely used *model* for tensor factorization that approximates a mode-N tensor $\mathcal{X}$ as a sum of R outer products of N vectors. Each outer product is a *rank*-1 *tensor*, and in tensor analysis, each rank-1 tensor corresponds to a *latent feature* of the data. The R vectors corresponding to each of the N modes can be combined to form a *factor matrix* for the corresponding mode by arranging them as column vectors of the matrix.

For example, the factorization of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ can be written in terms of factor matrices $\mathbf{A} \in \mathbb{R}^{I \times R}$, $\mathbf{B} \in \mathbb{R}^{J \times R}$, and $\mathbf{C} \in \mathbb{R}^{K \times R}$, where the columns of $\mathbf{A}$ (and respectively, $\mathbf{B}$ and $\mathbf{C}$) are the vectors used in forming the R outer products along mode-1 (and respectively, mode-2 and mode-3).

Formally, an unconstrained CPD for the three mode $\mathcal{X}$ with factors **A**,**B** and **C** can be defined as

$$\underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\operatorname{argmin}} \|\mathcal{X} - \sum_{r=1}^{R} \mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r)\| \tag{1}$$

, where $\mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r)$ is the outer product of the r^{th} vector that yields a rank-1 one tensor.

To calculate a CPD factorization of a tensor $\mathcal{X}$, the CANDECOMP/PARAFAC Alternating Least Squares (CP-ALS) algorithm is mostly commonly used due to its fast execution time and the simplicity of its algorithm. CP-ALS is an Block Coordinate Descent (BCD) type iterative algorithm, where in each iteration, the factor matrices are updated one mode at a time. When calculating the factor matrix of mode n, the factor matrices of the remaining modes are fixed, and a linear least squares problem is solved to update the factor matrix for mode n.

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

(1) *Tensor matricization* is the process by which a tensor is *unfolded* into a matrix. The mode-n matricization of a tensor $\mathcal{X}$, denoted $\mathbf{X}_{(n)}$, is obtained by laying out the mode-n fibers of $\mathcal{X}$ as the columns of $\mathbf{X}_{(n)}$.

(2) *Khatri-Rao product* [18] is the "matching column-wise" Kronecker product between two matrices. That is, given matrices $\mathbf{B} \in \mathbb{R}^{J \times R}$ and $\mathbf{C} \in \mathbb{R}^{K \times R}$, their Khatri-Rao product $\mathbf{K}$, denoted $\mathbf{K} = \mathbf{B} \odot \mathbf{C}$, where $\mathbf{K}$ is a $(J \cdot K) \times R$ matrix, is defined as: $B \odot C = [\mathbf{b}_1 \otimes \mathbf{c}_1 \; \mathbf{b}_2 \otimes \mathbf{c}_2 \ldots \mathbf{b}_R \otimes \mathbf{c}_R]$.

For a mode-3 tensor $\mathcal{X}$, the mode-1 MTTKRP operation can be expressed as $\mathbf{X}_{(1)}\,(\mathbf{B} \odot \mathbf{C})$. Note that for sparse tensors, the MTTKRP operation does not explicitly calculate the Khatri-Rao product, which typically results in a very tall and skinny matrix, as described above. Instead, the row of the Khatri-Rao product matrix required for each non-zero element in the sparse tensor is calculated *on-the-fly* using the corresponding rows of the factor matrices. Figure 2 illustrates the sparse MTTKRP operation for a single non-zero element. Note that the key distinction between sparse MTTKRP and dense MTTKRP is in the granularity of the MTTKRP computation (a non-zero element vs. large matrix derived from matricized tensor)

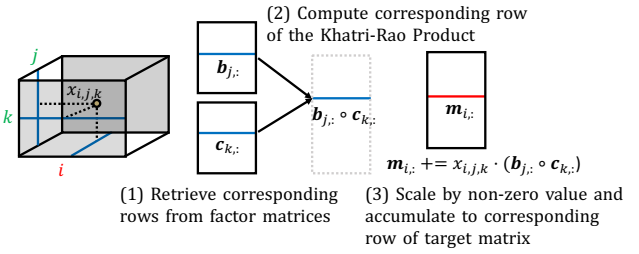

Figure 2: Illustration of the mode-1 sparse MTTKRP operation for a 3-mode tensor. For each non-zero element $x_{i,j,k}$, the j^{th} row and k^{th} row from factor matrices B and C, respectively, are loaded from memory. These are denoted by $\mathbf{b}_{j,:}$ and $\mathbf{c}_{k,:}$. A Hadamard product between the two rows are computed (which corresponds to calculating the Khatri-Rao product on-the-fly), and then the resulting vector is scaled by the non-zero value of $x_{i,j,k}$. This vector is then accumulated to the i^{th} row of a temporary matrix M (i.e., $\mathbf{m}_{i,:}$), which is later used to calculate the factor matrix A. Note that the computational workload for a sparse MTTKRP operation is bounded by the number of non-zero elements, while for a dense MTTKRP operation, it is determined by the product of the dimensions across each tensor mode.

2.3 Sparse Tensor Formats

The efficient representation and manipulation of sparse tensors are foundational to the performance of the sparse MTTKRP operation, particularly when accelerated on massively parallel GPU architectures. Various sparse tensor formats have been proposed to store and process the non-zero elements efficiently on GPUs [17, 20, 22, 25]. The state-of-the-art sparse tensor format and MTTKRP implementation for GPUs is the blocked linearized coordinate ordering (BLCO) [20]. We leverage this work to accelerate the sparse MTTKRP kernel in our cSTF framework.

2.4 Constrained Tensor Factorization

Imposing constraints on factorization, such as non-negativity, can benefit data analysis by retaining the property of the tensor data specific to the application domain. As such, there has been a wide variety of work dealing with imposing various types of constraints in matrix and tensor factorization algorithms. [8, 9, 11, 19, 29]

From an optimization problem standpoint, it simply extends Equation 1 by adding the target constraint (e.g., non-negativity) to yield the following optimization problem:

$$\begin{aligned} \underset{\mathbf{A},\mathbf{B},\mathbf{C}}{\operatorname{argmin}} \quad & \|\mathcal{X} - \sum_{r=1}^{R} \mathbf{A}(:,r) \circ \mathbf{B}(:,r) \circ \mathbf{C}(:,r)\| \\ \text{subject to} \quad & \mathbf{A}, \mathbf{B}, \mathbf{C} \geq 0. \end{aligned} \tag{2}$$

In this study, we focus on the alternating optimization with alternating direction method of multipliers (AO-ADMM) update method [9], as it demonstrates fast convergence in practice, even for non-smooth functions and weak convexity conditions. However,

Algorithm 1 Computing constrained tensor factorization for the CPD model using AO-ADMM

Input: $\mathcal{X}$ is $I_1 \times \cdots \times I_N$ tensor, R is approximation rank

1: % Initialize the factor matrices ($\mathbf{H}$)
2: **for** $n = 1$ to N **do**
3: Initialize $\mathbf{H}^{(n)}$
4: $\mathbf{G}^{(n)} = \mathbf{H}^{(n)T}\mathbf{H}^{(n)}$
5: **end for**
6: **repeat**
7: **for** $n = 1$ to N **do**
8: $\mathbf{S}^{(n)} = \mathbf{G}^{(1)} * \cdots * \mathbf{G}^{(n-1)} * \mathbf{G}^{(n+1)} * \cdots * \mathbf{G}^{(N)}$
9: $\mathbf{M}^{(n)} = \text{MTTKRP}\left(\mathcal{X}, \mathbf{H}^{(\mathbb{K}|\mathbb{K}=[1,N]-\{n\})}\right)$
10: $\mathbf{H}^{(n)} = \text{ADMM}(\mathbf{S}^{(n)}, \mathbf{M}^{(n)})$
11: $\mathbf{H}^{(n)}, \lambda = \text{normalize}(\mathbf{H}^{(n)}, \lambda)$
12: $\mathbf{G}^{(n)} = \mathbf{H}^{(n)T}\mathbf{H}^{(n)}$
13: **end for**
14: **until** convergence is reached

Output: $\mathcal{X} \approx [\![\mathbf{H}^{(1)}, \ldots, \mathbf{H}^{(N)}]\!]$

our framework can seamlessly incorporate a wide variety of update methods, such as HALS and MU, as we demonstrate in Section 5.4.

3 ACCELERATING CONSTRAINED SPARSE TENSOR FACTORIZATION ON GPUS

In this section, we present a high-level motivation for accelerating cSTF on massively parallel GPU architecture. We begin by describing the overall cSTF algorithm, and then provide a detailed description of how ADMM is used within the cSTF algorithm. We then use our analysis of the computational and data movement cost of ADMM to motivate the use of GPUs for accelerating cSTF.

3.1 Overview of the cSTF Algorithm

Algorithm 1 illustrates the pseudo-code of the cSTF algorithm, and the update of a single factor matrix is described in lines 8-12. The computation of the factor matrix for a given mode consists of four primary operations—Gram (lines 8 and 12), MTTKRP (line 9), ADMM update (line 10), and normalize (line 11)—which is representative of how constraints are applied in general. In this study, we focus primarily on the *update* part, which we show in later sections to be a significant performance bottleneck in cSTF that will benefit from GPU acceleration.

3.2 Overview of the ADMM algorithm

ADMM is a popular and effective method for solving large-scale convex optimization problems with several favorable characteristics [2]. It decomposes the problem into smaller sub-problems that can be solved more easily and converges to a solution quickly. In contrast to other algorithms that are limited to one particular constraint, ADMM supports various types of constraints, such as sparsity (L1 norm) and smoothness.

ADMM has been recognized for its effectiveness in solving constrained tensor factorization problems. Instead of directly applying ADMM to address the non-convex optimization problem as defined in Equation 2, recent advancements have led to the development of

Algorithm 2 ADMM Algorithm

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

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

alternating optimization with ADMM (AO-ADMM). This method applies the alternating least squares (ALS) approach to ADMM, where the algorithm alternately shifts its focus between variables (i.e., modes) to solve each sub-problem using ADMM [9, 29]. In essence, the cSTF algorithm described in Algorithm 1 is using AO-ADMM, as it is using ADMM (line 10) for computing the factor matrix for each mode. In the rest of this paper, we will use AO-ADMM and ADMM interchangeably.

Algorithm 2 illustrates the pseudocode for ADMM in the context of constrained tensor factorization. ADMM requires the output of the MTTKRP operation $\mathbf{M}$, the factor matrix of interest $\mathbf{H}$, which are both of size $I \times R$, and the Hadamard product of Gram matrices $\mathbf{S}$ of size $R \times R$ as input. After initializing the dual variable $\mathbf{U}$ and preconditioning the variable ρ, it iteratively updates $\mathbf{H}$ and $\mathbf{U}$ until convergence. From an optimization perspective, this algorithm is designed to solve the augmented Lagrangian problem through iterative updates of primal ($\mathbf{H}$) and dual ($\mathbf{U}$) variables. The goal is to balance achieving the lowest possible value of the objective function while adhering to the constraints. The dual variable update plays a crucial role in guiding the primal variable update to remain within the acceptable bounds set by the constraints.

3.3 Computation and Data Movement Analysis for ADMM

In the first step of ADMM, the Cholesky decomposition of $\mathbf{S} + \rho\mathbf{I}$ is computed (line 3). Since this computation is continuously used in the iterative process, it is typically computed initially and reused. The second step involves computing $\tilde{\mathbf{H}}$, which is the sum of the primal and dual variables scaled by the step size ρ, and then added with the MTTKRP output $\mathbf{M}$. $\tilde{\mathbf{H}}$ is then updated through a Cholesky solve, which usually requires a sequence of forward and backward substitutions (line 6). The third step applies the constraint to $\tilde{\mathbf{H}} - \mathbf{U}$ via the proximity operator r (line 7). The choice of r depends on the specific constraints or regularization used, providing algorithmic flexibility. Finally, the inner iteration updates the dual variable and checks for the convergence of both the primal and dual variables (line 9).

To predict the performance characteristics of ADMM, we can estimate the computational (W) and memory access (Q) costs for a single iteration, as shown in Equations 3 and 4, respectively.

Algorithm 3 GPU Optimized cuADMM Algorithm with *Preinversion* and *Operation fusion*. With *Preinversion*, an explicit inverse can be calculated using a Cholesky solve outside the inner loop (line 4), and the multiple triangular solve can be replaced with a DGEMM operation (line 7).

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

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

$$W = 19IR + 2IR^2 \text{ flops} \tag{3}$$

$$Q = 22IR + R^2 \text{ words} \tag{4}$$

$$I = \frac{W}{Q} = \frac{19 + 2R}{(22 + \frac{R}{I})8} \tag{5}$$

The computational cost totals $19IR + 2IR^2$ flops, with $19IR$ derived from matrix-matrix addition operations and $2IR^2$ from the Cholesky solve. The memory access cost totals $22IR + R^2$ words, where the predominant component ($22IR$) involves the read and write operations for matrices $\mathbf{H}, \mathbf{U}, \mathbf{M}$, each sized $I \times R$, and storing intermediate products.

The arithmetic intensity (I, in units of flop/byte) of ADMM is shown in Equation 5, assuming double-precision data. Assuming $I \gg R$, the arithmetic intensity can be approximated as $\frac{19+2R}{176}$, which yields arithmetic intensities of 0.29, 0.47, and 0.83 for ranks $R = 16$, 32, and 64, respectively.

The low arithmetic intensity suggests that ADMM performance is limited by memory bandwidth, and therefore can benefit from High Bandwidth Memory (HBM) available on the latest GPUs. The highly regular data access pattern offered by ADMM operating on dense factor matrices suggests that data access can easily be coalesced on GPUs to fully utilize the available bandwidth. Therefore, we hypothesize that *significant speedup can be achieved by offloading the entire cSTF workload onto GPUs.*

4 OUR FULLY GPU-RESIDENT CSTF FRAMEWORK

We modified the CPU-based PLANC library [5], which supports various constrained update methods for *dense* tensor factorization, to handle sparse tensors and their factorization. This was achieved by incorporating the state-of-the-art MTTKRP algorithm for CPUs [7] that utilizes the adaptive linearized tensor order (ALTO) sparse tensor format. This adaptation forms the basis for our initial analysis in Section 1 and lays the groundwork for subsequent GPU optimization. Our GPU cSTF framework leverages the blocked linearized cooradinate (BLCO) sparse tensor format [21] for the MTTKRP operation, which represents the state-of-the-art on GPUs. The remaining three operations—computing the Gram matrix, ADMM update, and normalization of the factor matrices—are implemented using our own CUDA implementations and cuBLAS kernels.

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

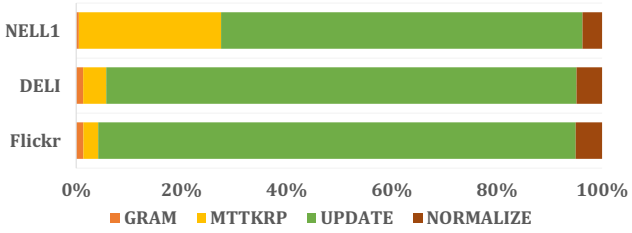

Figure 3: Execution time breakdown of cSTF on three tensors with the largest number of non-zero elements—`Flickr`**,** `Delicious` **(denoted** `DELI`**), and** `NELL1`**. Their execution time breakdown is representative of the tensors used in our evaluation, and illustrates that ADMM update dominates the overall execution time.**

4.1 Profiling the baseline cSTF implementation

We used our modified PLANC library to analyze the performance of cSTF on 10 real-world sparse tensors (Table 2). Our analysis in Figure 1 shows a significant change in the performance bottleneck when dealing with sparse tensors, with the ADMM update phase becoming the main bottleneck instead of the MTTKRP phase, which is the case for dense tensor factorization. Figure 3 illustrates the execution time breakdown for three largest sparse tensors—`Flickr`, `Delicious`, and `NELL1`. This execution time breakdown is representative of the tensors used in our evaluation, providing further evidence that ADMM update dominates the overall execution time for cSTF, and highlighting the importance of optimizing the update algorithm for cSTF.

4.2 cuADMM - GPU-optimized ADMM algorithm

Every operation in the ADMM algorithm, described in Section 3.2 can be implemented using the highly optimized cuBLAS kernels (e.g., DGEMM, DGEMV, and DGEAM) on GPUs. Therefore, it is easy to assume that the baseline ADMM algorithm will deliver close to peak peak performance on GPUs, especially given the simplicity of the operations.

However, when these operations compute over large factor matrices, significant memory traffic can occur while reading and writing

the *intermediate data* between kernel calls. This fact can easily be overlooked, leading to less-than-efficient implementations.

In shared memory systems, attempts have been made to optimize ADMM updates in the tensor factorization domain through blockwise reformulation, enhancing the temporal locality of matrices [29, 33]. However, these techniques are not effective on GPU architectures as GPU architectures are optimized for large, uniform memory access patterns and parallel workloads, rather than the segmented, blockwise computations.

An additional consideration for optimizing performance on GPU architectures is addressing the degradation caused by Cholesky or triangular solves. This computation type involves solving systems of equations with triangular matrices (either lower or upper) through sequential forward or backward substitutions. The inherently serialized nature of triangular solves conflicts with GPU capabilities, which excel in parallel processing across multiple data elements (i.e., SIMD or vectorized computation). This mismatch can lead to significant GPU core idle times and suboptimal performance.

In the following subsection, we outline two performance optimization strategies to address these issues. These impact of these two optimization strategies are evaluated in Section 5.2.

4.3 Optimized ADMM updates Implementation

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

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

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

Table 1: Hardware and software setup.

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

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

4.3.2 Pre-inversion. In iterative update frameworks such as ADMM, we often need to solve large linear systems denoted by $\mathbf{AX} = \mathbf{B}$ during each iteration. This process involves using Cholesky decomposition to extract the $\mathbf{L}$ and $\mathbf{L}^\mathbf{T}$ factors of the matrix $\mathbf{S} + \rho\mathbf{I}$ (line 3 in Algorithm 2). We then perform backward and forward substitutions (line 6) to apply the inverse of $\mathbf{S} + \rho\mathbf{I}$. However, the dense triangular solve process is inherently sequential, leading to low computational throughput.

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

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

5 EXPERIMENTS AND PERFORMANCE

5.1 Hardware specification and Datasets

The hardware specifications for our experiments are detailed in Table 1. Our study utilizes a range of real-world sparse tensor datasets of varying sizes and sparsity levels, sourced from the FROSTT repository [30], as shown in Table 2. We conduct tensor factorization using ranks {16, 32, 64} and take the average of 10 execution times

Table 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	$165.4K \times 11.4K \times 2$	$26M$	7.8×10^{-07}
Enron	$6K \times 5.7K \times 244.3K \times 1.2K$	$54.2M$	5.5×10^{-09}
NELL2	$12.1K \times 9.2K \times 28.8K$	$76.9M$	2.4×10^{-05}
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}
NELL1	$2.9M \times 2.1M \times 25.5M$	$143.6M$	9.1×10^{-13}
Amazon	$4.8M \times 1.8M \times 1.8M$	$1.7B$	1.1×10^{-10}

for each configuration. We also fix the number of ADMM iterations to 10 for a fair performance comparison across the different frameworks, and since ADMM converges in approximately 10 iterations for all practical purposes.

5.2 Optimized ADMM performance analysis

We evaluate the impact of our two optimization strategies—*operation fusion* (OF) (Section 4.3.1) and *pre-inversion* (PI) (Section 4.3.2)—by comparing their performance against the basic ADMM GPU implementation. The basic ADMM implementation uses the highly optimized cuBLAS kernels for its computation. Figure 4 displays the speedup achieved when our optimization strategies are applied separately (shown as orange and yellow bars for OF and PI, respectively) and applied together (shown as green bars) for a single ADMM iteration across a set of representative real-world sparse tensors for a rank-32 factorization.

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

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

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

The current state-of-the-art in high-performance cSTF that utilizes ADMM is the *CPU-only* SPLATT library [31]. There are very few studies in cSTF and ADMM on GPUs, and they are not focused on performance and/or the code is not available to the public. As such, we believe SPLATT represents the best comparison for this study, and comparing against SPLATT helps illustrate the benefit of GPU acceleration for cSTF.

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

Figures 5 and 6 shows the per-iteration end-to-end speedup of our framework against SPLATT using ADMM across 10 real-world datasets on the NVIDIA A100 and H100 GPUs, respectively. We achieve a geometric mean speedup of 5.10× and 7.01×, and the speedup for each tensor ranges from 1.47–41.59× and 1.22–58.05× on the two GPUs, respectively. We achieve higher speedup on tensors with larger mode lengths, as having more elements to apply constraints to on the factor matrices corresponds to more parallel work that benefits from GPU's massively parallel execution model.

The H100 GPU demonstrates higher performance compared to the A100 GPU, despite the two devices having the same memory bandwidth. Our analysis of the ADMM algorithm (Section 3.3) reveals a low arithmetic intensity, which suggests that ADMM performance is bounded by the memory bandwidth of the device. The MTTKRP algorithm demonstrates a similar behavior [3]. The higher performance on the H100 GPU comes from the higher overall cache size on the H100 GPU (20.3MB vs. 28.5MB L1D and 40MB vs. 50MB L2 on the A100 and H100, respectively). This result demonstrates the *importance of data reuse and caches* in the performance of cSTF.

Finally, we compare the speedup achieved by our GPU framework's MTTKRP and ADMM kernels *against each other*. Figure 7 and 8 displays the speedup divided into two components— *MTTKRP* and *ADMM*—for the A100 and H100 GPUs, respectively. It can be seen that speedup for MTTKRP is *approximately indirectly proportional to the speedup* for ADMM, with the VAST tensor being the only exception. When the mode lengths are long (i.e., sparse tensors on the upper left corner), ADMM benefits more from GPU acceleration, as described in Section 5.3 (i.e., higher level of parallelism). However, longer mode lengths also result in higher sparsity, which lowers the performance achieved by the MTTKRP kernel, as higher sparsity generally lowers data reuse. Conversely, when the mode lengths are short (i.e., sparse tensors on the lower right corner), higher speedup is achieved for MTTKRP while lower speedup is achieved for ADMM. This results illustrates the *importance of optimizing both kernels*, as neglecting MTTKRP may result in no improvement in performance for sparse tensors with short mode lengths, and neglecting ADMM may result in no improvement in performance for sparse tensors with long mode lengths.

5.4 Accelerating additional non-negativity constraint algorithms

To demonstrate the flexibility of our GPU cSTF framework, we integrate two additional constraint update schemes—multiplicative update (MU) [14] and hierarchical alternating least square (HALS) [4] for non-negativity—and compare its performance against the CPU-based PLANC library that we modified to support sparse tensor factorization, as described in Section 4. Figures 9 and 10 show the achieved speedup. Our framework achieves geometric mean speedup of 6.42× and 5.90× for MU and HALS, respectively,

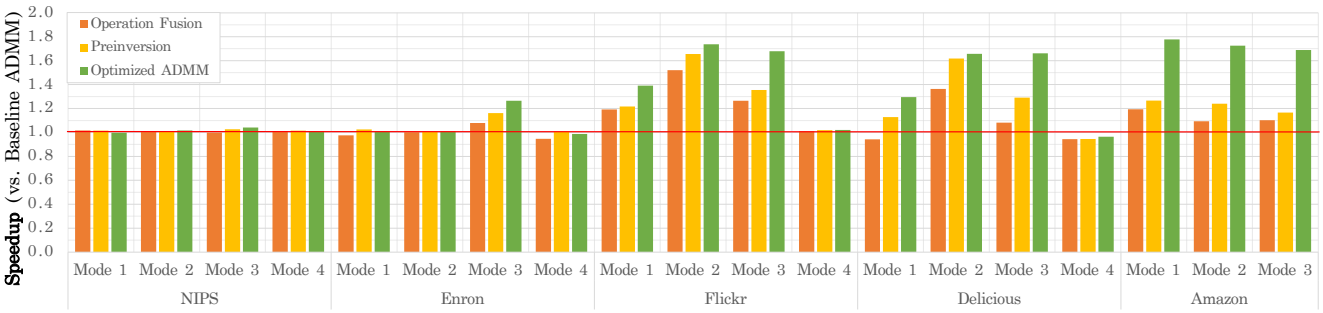

Figure 4: Speedup achieved by cuADMM over the baseline ADMM implementation that uses cuBLAS and cuSOLVER kernels. The orange, yellow, and green bars correspond to the speedup achieved by operation fusion (OF), pre-inversion (PI), and both OF and PI, respectively. The datasets are categorized into three groups based on factor matrix size: small (NIPS), medium (Enron), and large (Flickr, Delicious, and Amazon). Speedup is correlated to the factor matrix size, and can be as high as 1.8×.

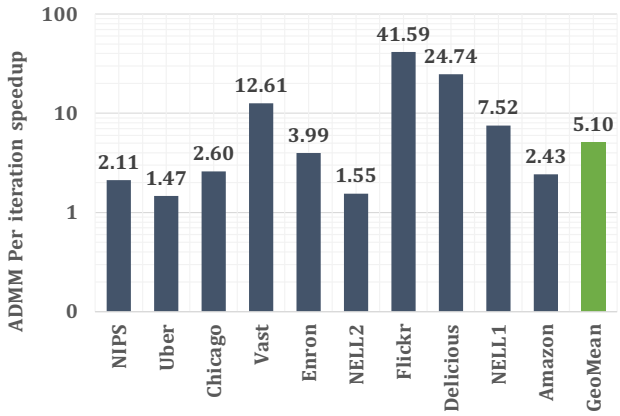

Figure 5: End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on A100 GPU.

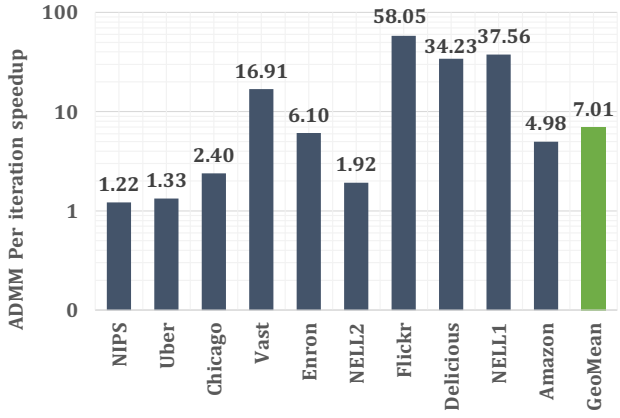

Figure 6: End-to-end per iteration speedup vs. SPLATT using ADMM update for rank $R = 32$ on H100 GPU.

on A100, and 8.89× and 7.78× on H100. This is comparable to the speedup achieved when using the ADMM update scheme.

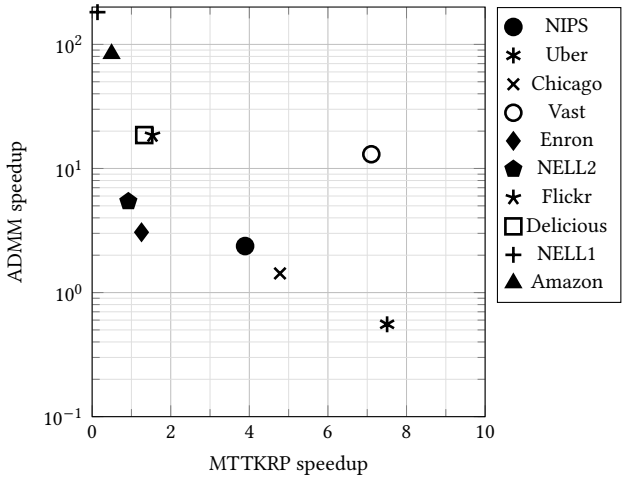

Figure 7: Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the A100 GPU

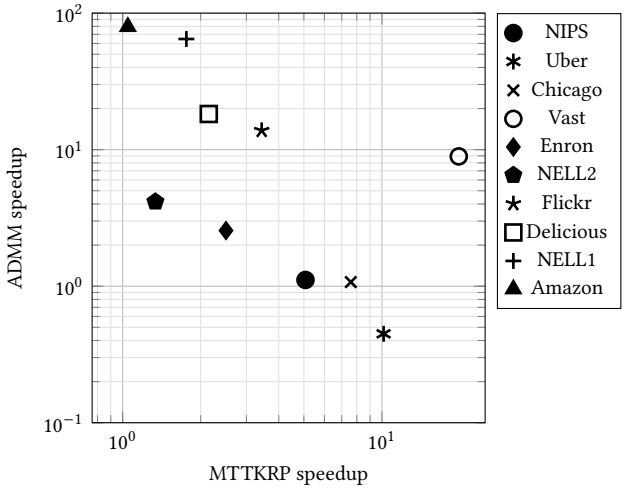

Figure 8: Comparison of speedup achieved for MTTKRP and ADMM across 10 real-world sparse tensors on the H100 GPU

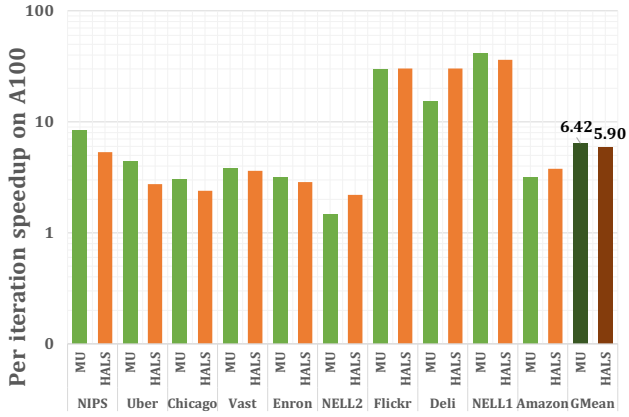

Figure 9: Speedup achieved on the A100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library that supports MU and HALS.

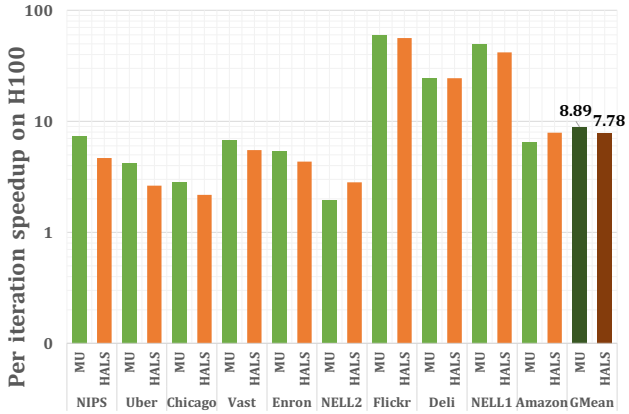

Figure 10: Speedup achieved on the H100 GPU by our GPU cSTF framework using the MU and HALS update schemes against PLANC, the state-of-the-art CPU-based cSTF library that supports MU and HALS.

6 RELATED WORK

In the realm of constrained tensor factorization, several frameworks and methodologies have been explored to optimize performance and accuracy. The functionality of supporting generalized loss functions is central to the work of both [8] and [24], which focus on computing CP-based tensor factorization for dense static and streaming tensors, respectively. This approach is characterized by its versatility in handling a range of loss functions pertinent to various distributions and its ability to manage corresponding constraints.

[5, 10] introduces a holistic constrained factorization framework that supports various optimization algorithms for dense matrices and tensors as well as sparse matrices. It alleviates the cost of MTTKRP by reusing partial MTTKRP products and actively offloading the computations as GEMV and GEMM calls for the GPU. It also provides scalability for distributed systems by offering parallelization strategies that optimize load balance, minimize communication, and efficiently distribute results.

AO-ADMM, initially introduced as an update method for constrained matrix and tensor factorization by Huang et al. [9], has seen further advancements. Smith et al. [29] proposed an accelerated C implementation for sparse tensor factorization, employing blocked ADMM to enhance thread parallelism and improve convergence properties. Soh et al. [33] built upon this by further accelerating ADMM updates for streaming sparse tensor factorization, utilizing techniques such as data blocking, operation fusion, and parallel reduction in residual computations.

[36] solves non-negative dense tensor factorization using accelerated proximal gradient (APG) method. It also reduces computational complexity by computing the MTTKRP through low-rank approximation.

GPU systems has also been widely adopted for constrained tensor factorization frameworks due to its high throughput and parallel processing capabilities. Specifically, [11] optimizes sparse non-negative matrix factorization (MF) by employing a randomized partitioning strategy to distribute workloads efficiently across multiple GPUs. Also, it reduces the overall communication volume through the adoption of a point-to-point communication exploiting the sparsity of the matrix.

For non-negative dense tensor factorizations, [1] accelerates alternating rank-wise column updates of factor matrices using a gradient descent based method. To compute the *update rate* for each rank, it decomposes the tensor into smaller tiles where it computes partial sums that is later reduced for rank updates.

For non-negative sparse tensor factorizations, [16] proposes a fine-grained update scheme that computes a gradient descent-based, constrained elementwise update for each factor matrix element. The update imposes non-negativity by setting the learning rate where the negative values is cancelled out. Using compressed sparse fibers(CSF) for sparse tensor storage, it spatially partitions subtensors and corresponding factor matrices rows onto a grid of GPU processors.

The decomposability features of Alternating Direction Method of Multipliers (ADMM) make it particularly suitable for GPU acceleration. Schubiger et al. [26] introduced an accelerated GPU implementation for constrained quadratic programming (QP) problems using ADMM. The iterative updating of primal and dual variables in the ADMM process, aimed at satisfying the Karush-Kuhn-Tucker (KKT) optimality conditions, gains significant efficiency from the incorporation of a parallelizable Preconditioned Conjugate Gradient (PCG) method.

7 CONCLUSION AND FUTURE WORK

In this study, we introduce the first ever software framework for accelerating constrained sparse tensor factorization specifically designed for fully offloading computation to GPUs. Although previous research has extensively explored constrained dense/sparse matrix factorization and unconstrained sparse tensor factorization for GPUs, the domain of constrained sparse tensor factorization remains comparatively under-explored.

Our work represents a pioneering effort in the field of constrained sparse tensor factorization on GPUs, introducing novel algorithmic optimizations specific to massively parallel GPU architectures. By fully offloading the computation to the GPU, we

reduce unnecessary communication overhead between the host and the GPU, leading to substantial performance improvements. Empirical evaluations demonstrate that our GPU-based framework achieves geometric mean speedup of 5.10× (max 41.59×) and 7.01× (max 58.05×) over state-of-the-art CPU-based SPLATT library on the latest NVIDIA A100 and H100 GPUs, respectively.

For future work, we plan to create decision models to dynamically determine whether to execute computations on the CPU, on the GPU, or on both (heterogeneously), providing flexibility and maximizing the overall performance and resource utilization based on the characteristics of the data. We also plan to extend our framework to support multi-GPU and distributed-memory computation.

REFERENCES

[1] J Antikainen, J Havel, R Josth, A Herout, P Zemcik, and M Hauta-Kasari. 2011. Nonnegative Tensor Factorization Accelerated Using GPGPU. *IEEE Trans. Parallel Distrib. Syst.* 22, 7 (July 2011), 1135–1141. https://doi.org/10.1109/TPDS.2010.194

[2] Stephen Boyd. 2010. Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers. *FNT in Machine Learning* 3, 1 (2010), 1–122. https://doi.org/10.1561/2200000016

[3] J. Choi, X. Liu, S. Smith, and T. Simon. 2018. Blocking Optimization Techniques for Sparse Tensor Computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 568–577. https://doi.org/10.1109/IPDPS.2018.00066

[4] Andrzej Cichocki and Anh-Huy Phan. 2009. Fast Local Algorithms for Large Scale Nonnegative Matrix and Tensor Factorizations. *IEICE Trans. Fundamentals* E92-A, 3 (2009), 708–721. https://doi.org/10.1587/transfun.E92.A.708

[5] Srinivas Eswar, Koby Hayashi, Grey Ballard, Ramakrishnan Kannan, Michael A. Matheson, and Haesun Park. 2021. PLANC: Parallel Low-rank Approximation with Nonnegativity Constraints. *ACM Trans. Math. Softw.* 47, 3 (Sept. 2021), 1–37. https://doi.org/10.1145/3432185

[6] Hadi Fanaee-T and João Gama. 2016. Tensor-based anomaly detection: An interdisciplinary survey. *Knowledge-Based Systems* 98 (2016), 130 – 147. https://doi.org/10.1016/j.knosys.2016.01.027

[7] Ahmed E. Helal, Jan Laukemann, Fabio Checconi, Jesmin Jahan Tithi, Teresa Ranadive, Fabrizio Petrini, and Jeewhan Choi. 2021. ALTO: Adaptive Linearized Storage of Sparse Tensors. In *Proceedings of the ACM International Conference on Supercomputing*. 404–416. https://doi.org/10.1145/3447818.3461703 arXiv:2102.10245 [cs].

[8] David Hong, Tamara G. Kolda, and Jed A. Duersch. 2020. Generalized Canonical Polyadic Tensor Decomposition. *SIAM Rev.* 62, 1 (Jan. 2020), 133–163. https://doi.org/10.1137/18M1203626 arXiv:1808.07452 [cs, math].

[9] Kejun Huang, Nicholas D. Sidiropoulos, and Athanasios P. Liavas. 2016. A Flexible and Efficient Algorithmic Framework for Constrained Matrix and Tensor Factorization. *IEEE Trans. Signal Process.* 64, 19 (Oct. 2016), 5052–5065. https://doi.org/10.1109/TSP.2016.2576427

[10] Ramakrishnan Kannan, Grey Ballard, and Haesun Park. 2018. MPI-FAUN: An MPI-Based Framework for Alternating-Updating Nonnegative Matrix Factorization. *IEEE Transactions on Knowledge and Data Engineering* 30, 3 (2018), 544–558. https://doi.org/10.1109/TKDE.2017.2767592

[11] Oguz Kaya, Ramakrishnan Kannan, and Grey Ballard. 2018. Partitioning and Communication Strategies for Sparse Non-negative Matrix Factorization. In *Proceedings of the 47th International Conference on Parallel Processing*. ACM, Eugene OR USA, 1–10. https://doi.org/10.1145/3225058.3225127

[12] Teruyoshi Kobayashi, Anna Sapienza, and Emilio Ferrara. 2018. Extracting the multi-timescale activity patterns of online financial markets. *Scientific Reports* 8, 1 (2018), 1–11. https://doi.org/10.1038/s41598-018-29537-w

[13] Tamara G. Kolda and Brett W. Bader. 2009. Tensor Decompositions and Applications. *SIAM Rev.* 51, 3 (Aug. 2009), 455–500. https://doi.org/10.1137/07070111X

[14] Daniel D. Lee and H. Sebastian Seung. 1999. Learning the parts of objects by non-negative matrix factorization. *Nature* 401, 6755 (Oct. 1999), 788–791. https://doi.org/10.1038/44565

[15] Dimitri Leggas, Christopher Coley, and Teresa Ranadive. 2021. Knowledge-guided Tensor Decomposition for Baselining and Anomaly Detection. In *2021 IEEE High Performance Extreme Computing Conference*. 1–7. https://doi.org/10.1109/HPEC49654.2021.9622859

[16] Hao Li, Kenli Li, Jiyao An, and Keqin Li. 2018. CUSNTF: A Scalable Sparse Non-negative Tensor Factorization Model for Large-scale Industrial Applications on Multi-GPU. In *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*. ACM, Torino Italy, 1113–1122. https://doi.org/10.1145/3269206.3271749

[17] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. 2017. A Unified Optimization Approach for Sparse Tensor Operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*. 47–57. https://doi.org/10.1109/CLUSTER.2017.75

[18] Shangzhi Liu and Götz Trenkler. 2008. Hadamard, Khatri-Rao, Kronecker, and Other Matrix Products. *International Journal of Information and Systems Sciences* 4, 1 (2008), 160–177. https://doi.org/10.1155/2016/8301709

[19] Gordon E. Moon, J. Austin Ellis, Aravind Sukumaran-Rajam, Srinivasan Parthasarathy, and P. Sadayappan. 2020. ALO-NMF: Accelerated Locality-Optimized Non-negative Matrix Factorization. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. ACM, Virtual Event CA USA, 1758–1767. https://doi.org/10.1145/3394486.3403227

[20] Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi. 2022. Efficient, out-of-Memory Sparse MTTKRP on Massively Parallel Architectures. In *Proceedings of the 36th ACM International Conference on Supercomputing* (Virtual Event) *(ICS '22)*. Association for Computing Machinery, New York, NY, USA, Article 26, 13 pages. https://doi.org/10.1145/3524059.3532363

[21] Andy Nguyen, Ahmed E. Helal, Fabio Checconi, Jan Laukemann, Jesmin Jahan Tithi, Yongseok Soh, Teresa Ranadive, Fabrizio Petrini, and Jee W. Choi. 2022. Efficient, Out-of-Memory Sparse MTTKRP on Massively Parallel Architectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*. 1–13. https://doi.org/10.1145/3524059.3532363 arXiv:2201.12523 [cs].

[22] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-Mode Representation of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 49, 25 pages. https://doi.org/10.1145/3295500.3356216

[23] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. 2019. An Efficient Mixed-Mode Representation of Sparse Tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* (Denver, Colorado) *(SC '19)*. Association for Computing Machinery, New York, NY, USA, Article 49, 25 pages. https://doi.org/10.1145/3295500.3356216

[24] Eric Phipps, Nick Johnson, and Tamara G. Kolda. 2021. Streaming Generalized Canonical Polyadic Tensor Decompositions. http://arxiv.org/abs/2110.14514 arXiv:2110.14514 [cs, math].

[25] Eric T. Phipps and Tamara G. Kolda. 2019. Software for Sparse Tensor Decomposition on Emerging Computing Architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290. https://doi.org/10.1137/18M1210691

[26] Michel Schubiger, Goran Banjac, and John Lygeros. 2020. GPU acceleration of ADMM for large-scale quadratic programming. *J. Parallel and Distrib. Comput.* 144 (2020), 55–67. https://doi.org/10.1016/j.jpdc.2020.05.021

[27] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. 2017. Tensor Decomposition for Signal Processing and Machine Learning. *IEEE Transactions on Signal Processing* 65, 13 (2017), 3551–3582. https://doi.org/10.1109/TSP.2017.2690524

[28] Shaden Smith, Alec Beri, and George Karypis. 2017. Constrained Tensor Factorization with Accelerated AO-ADMM. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Bristol, United Kingdom, 111–120. https://doi.org/10.1109/ICPP.2017.20

[29] Shaden Smith, Alec Beri, and George Karypis. 2017. Constrained Tensor Factorization with Accelerated AO-ADMM. In *2017 46th International Conference on Parallel Processing (ICPP)*. IEEE, Bristol, United Kingdom, 111–120. https://doi.org/10.1109/ICPP.2017.20

[30] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. *FROSTT: The Formidable Repository of Open Sparse Tensors and Tools*. http://frostt.io/

[31] Shaden Smith and George Karypis. 2017. SPLATT: The Surprisingly ParalleL spArse Tensor Toolkit. https://github.com/ShadenSmith/splatt. Online; accessed 24 January 2021.

[32] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. 2015. SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication. (2015), 61–70. https://doi.org/10.1109/IPDPS.2015.27

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

[34] Ananda Streit, Gustavo H.A. Santos, Rosa M.M. Leão, Edmundo de Souza e Silva, Daniel Menasché, and Don Towsley. 2021. Network Anomaly Detection Based on Tensor Decomposition. *Computer Networks* 200 (2021), 108503. https://doi.org/10.1016/j.comnet.2021.108503

[35] Yanqing Zhang, Xuan Bi, Niansheng Tang, and Annie Qu. 2021. Dynamic Tensor Recommender Systems. *J. Mach. Learn. Res.* 22, 1 (Jan. 2021). Publisher: JMLR.org.

[36] Yu Zhang, Guoxu Zhou, Qibin Zhao, Andrzej Cichocki, and Xingyu Wang. 2016-07. Fast nonnegative tensor factorization based on accelerated proximal gradient and low-rank approximation. *Neurocomputing* 198 (2016-07).