cuFastTuckerPlusTC: A Stochastic Parallel Sparse FastTucker Decomposition Using GPU Tensor Cores

Zixuan Li, Mingxing Duan, Huizhang Luo, Wangdong Yang, Kenli Li, *Senior Member, IEEE*, and Keqin Li, *Fellow, IEEE*

***Abstract*—Sparse tensors are prevalent in real-world applications, often characterized by their large-scale, high-order, and high-dimensional nature. Directly handling raw tensors is impractical due to the significant memory and computational overhead involved. The current mainstream approach involves compressing or decomposing the original tensor. One popular tensor decomposition algorithm is the Tucker decomposition. However, existing state-of-the-art algorithms for large-scale Tucker decomposition typically relax the original optimization problem into multiple convex optimization problems to ensure polynomial convergence. Unfortunately, these algorithms tend to converge slowly. In contrast, tensor decomposition exhibits a simple optimization landscape, making local search algorithms capable of converging to a global (approximate) optimum much faster. In this article, we propose the FastTuckerPlus algorithm, which decomposes the original optimization problem into two non-convex optimization problems and solves them alternately using the Stochastic Gradient Descent method. Furthermore, we introduce cuFastTuckerPlusTC, a fine-grained parallel algorithm designed for GPU platforms, leveraging the performance of tensor cores. This algorithm minimizes memory access overhead and computational costs, surpassing the state-of-the-art algorithms. Our experimental results demonstrate that the proposed method achieves a $2\times$ to $8\times$ improvement in convergence speed and a $3\times$ to $5\times$ improvement in per-iteration execution speed compared with state-of-the-art algorithms.**

***Index Terms*—Sparse tensor decomposition, GPU CUDA parallelization, stochastic gradient descent, tensor cores.**

Received 30 June 2023; revised 14 November 2025; accepted 19 November 2025. Date of publication 24 November 2025; date of current version 19 December 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62321003, Grant 62227808, Grant U21A20461, Grant 62422205, Grant U24A20255, and Grant 62272149, and in part by the Natural Science Foundation of Hunan Province, China under Grant 2025JJ60399. Recommended for acceptance by K. Cameron. *(Corresponding author: Kenli Li.)*

Zixuan Li is with the School of Computer Science & School of Cyberspace Science, Xiangtan University, Xiangtan, Hunan 411105, China, and also with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410012, China (e-mail: zixuanli@hnu.edu.cn).

Mingxing Duan, Huizhang Luo, Wangdong Yang, and Kenli Li are with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410012, China, and also with the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China (e-mail: duanmingxing@hnu.edu.cn; luohuizhang@hnu.edu.cn; yangwangdong@hnu.edu.cn; lkl@hnu.edu.cn).

Keqin Li is with the College of Computer Science and Electronic Engineering, Hunan University, Changsha, Hunan 410012, China, also with the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China, and also with the Department of Computer Science, State University of New York, New Paltz, NY 12561 USA (e-mail: lik@newpaltz.edu).

This article has supplementary downloadable material available at https://doi.org/10.1109/TPDS.2025.3636547, provided by the authors.

Digital Object Identifier 10.1109/TPDS.2025.3636547

I. Introduction

MULTI-ORDER data, represented as tensors, is pervasive in various domains, including social networks [1], [2], recommender systems [3], [4], cryptography [5], bioinformatics [6], and neuroscience [7], [8]. Tensors serve as higher-order generalizations of matrices, capable of capturing complex relationships among multiple entities or variables. Leveraging tensor decomposition techniques provides a powerful framework for uncovering latent structures, reducing dimensionality, and performing feature extraction [9]. Through this decomposition process, a concise representation of the data is obtained, enabling efficient storage, analysis, and interpretation. However, in many real-world applications, tensors often exhibit sparsity, where a significant number of elements are zero or missing. Sparsity is a common characteristic in large-scale datasets due to inherent constraints or limitations during data acquisition. For instance, in social network analysis, connections between individuals may be sparse, while in image analysis, pixels representing background regions are predominantly zero. The growing interest in large-scale sparse tensor decomposition in recent years stems from the exponential growth of data in the aforementioned domains. Nonetheless, the analysis of such tensors poses unique challenges due to their immense size, sparsity patterns, and computational complexity.

Sparse tensors, in contrast to dense tensors, exhibit a substantial number of zero or missing entries, which are commonly encountered in real-world applications due to the inherent sparsity of the underlying phenomena. The presence of sparsity in tensors presents unique challenges and opportunities for data analysis and computational algorithms. Traditional tensor decomposition algorithms, originally designed for dense tensors, face difficulties in effectively scaling to large-scale sparse tensors due to the computational overhead involved in processing numerous zero entries. Consequently, specialized algorithms and techniques have been developed to leverage the sparsity patterns and efficiently factorize sparse tensors. Furthermore, sparse tensor decomposition techniques should possess the ability to accurately recover missing entries and handle inherent data noise.

Tucker decomposition is a widely used technique for factorizing tensors into a core tensor and factor matrices, enabling the analysis of complex relationships and latent structures within tensors. There are two primary categories of methods used for implementing Tucker decomposition. The first category is based on Singular Value Decomposition (SVD) methods, such

TABLE I
TABLE OF ACRONYMS

Acronym	Full form
SVD	Singular Value Decomposition
HOOI	Higher Order Orthogonal Iteration
ALS	Alternating Least Squares
CD	Coordinate Descent
SGD	Stochastic Gradient Descent
RMSE	Root Mean Square Error
SOTA	state-of-the-art

as High Order Singular Value Decomposition [10] and Higher Order Orthogonal Iteration (HOOI) [11]. The second category is comprised of gradient-based methods, including Alternating Least Squares (ALS) [12], Coordinate Descent (CD) [13], and Stochastic Gradient Descent (SGD) [14]. SVD-based methods typically involve two essential computational steps: Tensor-Time-Matrix-chain and SVD. Both of these steps are computationally expensive and require significant memory usage. Gradient-based methods, on the other hand, calculate the gradient of a parameter and update it using gradient descent or least squares. Sparse Tucker decomposition extends this technique to handle large-scale sparse tensors where a substantial number of elements are zero or missing. For easy reference, Table I summarizes the key aronyms utilized throughout the paper.

Parallel computing refers to the utilization of multiple processing units or resources that work simultaneously to solve computational problems more efficiently. By parallelizing the Sparse Tucker decomposition process, the computational time can be significantly reduced, enabling the analysis of larger and more complex sparse tensors. These techniques leverage various parallel computing architectures, including multi-core processors, distributed systems, and GPU accelerators, to achieve high-performance factorization of large-scale sparse tensors. Several parallel algorithms have been developed for Sparse Tucker decomposition. ParTi! [15] is an HOOI-based parallel algorithm designed for GPUs, which utilizes the Balanced Compressed Sparse Fiber format to accelerate Tensor-Time-Matrix-chain calculations. P-tucker [16] is an ALS-based parallel algorithm for multi-core CPUs, reducing memory requirements for updating factor matrices, but it incurs significant computational overhead and exhibits unbalanced load distribution. Vest [17] is a CD-based parallel algorithm for multi-core CPUs, which prunes unimportant entries to reduce calculations, but the pruning process itself is time-consuming. SGD_Tucker [18] is an SGD-based parallel algorithm for multi-core GPUs, dividing high-dimensional intermediate variables into smaller ones to reduce memory overhead during updates. GTA [19] is an ALS-based parallel algorithm designed for heterogeneous platforms and serves as an extended version of P-Tucker. Bigtensor [20] is a distributed Tucker decomposition algorithm designed to process large-scale tensors efficiently across multiple computing nodes. cuTucker [21] is an SGD-based parallel algorithm that runs on multiple GPUs. cuFastTucker [21] is an SGD-based parallel algorithm for multiple GPUs, decomposing the core tensor into multiple core matrices to reduce space and computational overhead. Building upon cuFastTucker [21], cuFasterTucker [22] further reduces the computation of shared and reusable intermediate variables, maximizing the utilization of GPU computing resources, and introduces non-negative constraints.

The aforementioned algorithms are based on convex relaxations of non-convex optimization problems, transforming them into convex optimization problems. While they can achieve convergence in polynomial time, they often exhibit slow convergence in practice [16], [17], [18], [21], [22]. In contrast, local search algorithms have demonstrated fast convergence in practical scenarios [23]. Many non-convex optimization problems are conjectured to possess favorable geometric properties, wherein all local optima are (approximately) global optima [24], [25]. Tensor factorization [14], matrix awareness [26], [27], and matrix completion [28] exhibit good optimization landscapes, where all locally optimal solutions are globally optimal. Similarly, common low-rank matrix factorizations share a unified optimization landscape, wherein all local optima are global optima, and high-order saddle points are absent [29]. Moreover, for an exact standard Tucker decomposition, all locally optimal solutions are also globally optimal [30]. These problems can be effectively addressed using fundamental optimization algorithms such as stochastic gradient descent [14], [31], [32], [33]. From these observations, we infer that FastTucker decomposition also possesses a favorable optimization landscape, where all local optima are (approximately) global optima, and can be effectively solved using stochastic gradient descent.

Tensor Cores have emerged as specialized hardware components that provide powerful acceleration for tensor computations. They leverage the parallelism and computational capabilities of modern GPUs to efficiently perform tensor operations. Tensor Cores incorporate dedicated hardware units and optimized algorithms, enabling rapid and precise tensor computations. The introduction of Tensor Cores has revolutionized the field of tensor computations, delivering significant speedups in various applications, including deep neural network training [34] and complex physical system simulations [35]. The efficient handling of large-scale tensor computations by Tensor Cores has opened up new possibilities for addressing challenging problems in machine learning [36], [37], [38], scientific simulations [39], [40], and data analysis [41]. However, not all algorithms are suitable for acceleration using Tensor Cores.

In this paper, we introduce cuFastTuckerPlusTC, a parallel sparse FastTucker decomposition algorithm designed for the GPU platform with Tensor Cores. Unlike previous convex optimization approaches, cuFastTuckerPlusTC utilizes a non-convex stochastic optimization strategy. The algorithm consists of two parts: simultaneous updates of all factor matrices followed by simultaneous updates of all core matrices. Our contributions can be summarized as follows:

1) *Algorithm:* The proposed method, cuFastTuckerPlusTC, is a stochastic optimization strategy specifically developed for parallel sparse FastTucker decomposition on the GPU platform using Tensor Cores. It addresses the optimization problem by decomposing it into two non-convex optimization subproblems, which are alternately solved to achieve convergence. The algorithm exhibits rapid convergence and effectively utilizes the capabilities of Tensor Cores, resulting in faster performance compared to existing state-of-the-art (SOTA) parallel Tucker decomposition algorithms such as cuFastTucker and cuFasterTucker. Although cuFastTuckerTC and

TABLE II
TABLE OF SYMBOLS

Symbol	Definition
$\boldsymbol{\mathcal{X}}$	The input N-order tensor $\in \mathbb{R}^{I_1 \times \cdots \times I_N}$
$\boldsymbol{\mathcal{G}}$	The N-order core tensor $\in \mathbb{R}^{J_1 \times \cdots \times J_N}$
$x_{i_1,\cdots,i_n}$	$(i_1, \cdots, i_n)$th element of tensor $\mathcal{X}$
$\{N\}$	The index set $\{1, \cdots, N\}$
Ω	The set of non-zero elements in $\boldsymbol{\mathcal{X}}$
$\lvert\Omega\rvert$	The number of non-zero elements in Ω
Ψ	The sample set from Ω
$\boldsymbol{A}^{(n)}$	The nth factor matrix $\in \mathbb{R}^{I_n \times J_n}$
$\boldsymbol{B}^{(n)}$	The nth core matrix $\in \mathbb{R}^{J_n \times R}$
$\boldsymbol{a}^{(n)}_{i_n,:}$	The i_nth row vector $\in \mathbb{R}^{1 \times J_n}$ of $\boldsymbol{A}^{(n)}$
$\boldsymbol{b}^{(n)}_{:,r}$	The rth column vector $\in \mathbb{R}^{J_n \times 1}$ of $\boldsymbol{B}^{(n)}$
$\circ$	Outer product
$\times_{(n)}$	n-Mode Tensor-Matrix product
$\odot$	R Dot Product
$*$	Hadamard Product
$\circledast$	R Hadamard Product

cuFasterTuckerTC are versions of cuFastTucker and cuFasterTucker respectively, accelerated by Tensor Cores, their performance does not match that of cuFastTuckerPlusTC.

2) *Theory:* In comparison to cuFastTucker, our proposed cuFastTuckerPlusTC exhibits a smaller memory access overhead of $(M+R)\sum_{n=1}^{N} J_n$ and a computational overhead of $MR(\sum_{n=1}^{N} J_n + N(N-2))$ for key steps. On the other hand, cuFasterTucker aims to reduce the real-time computation of $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$, which incurs an overhead of $MR\sum_{n=1}^{N} J_n$, by introducing an additional memory access of $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$, resulting in an overhead of $N(N-1)R$. In the case of cuFastTuckerPlusTC, the calculation of $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$ is performed in real-time using Tensor Cores, without introducing any additional memory access overhead. Furthermore, the increased running time of cuFastTuckerPlusTC is lower than the additional memory access time required for $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$. These advantages make cuFastTuckerPlusTC superior to cuFasterTucker in terms of both computation and memory access.

3) *Performance:* The experimental results of cuFastTuckerPlusTC demonstrate its superior convergence speed and efficiency compared to the current SOTA algorithms. Specifically, during the factor matrix update step, cuFastTuckerPlusTC achieves a speedup of $9\times$ to $20\times$ compared to cuFastTucker and $2\times$ to $3\times$ compared to cuFasterTucker. Similarly, during the core matrix update step, cuFastTuckerPlusTC achieves a speedup of $27\times$ to $31\times$ compared to cuFastTucker and $2\times$ to $6\times$ compared to cuFasterTucker. These significant speed improvements indicate the effectiveness and efficiency of cuFastTuckerPlusTC in terms of both computational time and memory access.

The code for cuFastTuckerPlusTC, as utilized in this paper, along with a toy dataset, is available for reproducibility at https://github.com/ZixuanLi-China/cuFastTuckerPlus. The subsequent sections of this paper are structured as follows. Section II introduces the notations, definitions, and the problem to be addressed, along with its basic algorithm. Section III presents our proposed method, which is a non-convex optimization algorithm called FastTuckerPlus. Section IV describes our proposed fine-grained parallel sparse Tucker algorithm, cuFasterTuckerPlus, designed specifically for the GPU platform with Tensor Cores. Section V showcases the experimental results of cuFastTuckerPlusTC, comparing its performance to SOTA algorithms. Finally, in Section VI, we summarize our work.

II. PRELIMINARIES

In this paper, Section II-A provides a comprehensive description of the notations used. Section II-B presents the fundamental definitions necessary for understanding the concepts discussed. The specific problem formulation is presented in Section II-C, while the supplementary material provides a detailed explanation of the proposed SGD-based convex optimization approach used to address it. For easy reference, Table II summarizes the key notations utilized throughout the paper.

A. Notations

The notation conventions used in this paper are as follows: tensors are represented using bold Euler script letters (e.g., $\boldsymbol{\mathcal{X}}$), matrices are denoted by bold uppercase letters (e.g., $\boldsymbol{A}$), vectors are represented using bold lowercase letters (e.g., $\boldsymbol{a}$), and scalars are denoted by regular lowercase or uppercase letters (e.g., k and N). The elements of a tensor are specified by combining the symbolic name of the tensor with the corresponding indices. For instance, $x_{i_1,\ldots,i_n}$ denotes the element located at indices $(i_1, \ldots, i_n)$ in the tensor $\boldsymbol{\mathcal{X}}$. Additionally, $\boldsymbol{a}_{i,:}$ refers to the i-th row of matrix $\boldsymbol{A}$, and $\boldsymbol{b}_{:,r}$ denotes the r-th column of matrix $\boldsymbol{B}$.

B. Basic Definitions

Definition 1 (n-Mode Tensor-Matrix Product): Given a N-order tensor $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$ and a matrix $\boldsymbol{A} \in \mathbb{R}^{I_n \times J_n}$, n-Mode Tensor-Matrix product projects $\boldsymbol{\mathcal{G}}$ and $\boldsymbol{A}$ to a new tensor $(\boldsymbol{\mathcal{G}} \times_{(n)} \boldsymbol{A}) \in \mathbb{R}^{I_1 \times \cdots \times J_{n-1} \times I_n \times J_{n+1} \times \cdots J_N}$ according to the coordinates, where $(\boldsymbol{\mathcal{G}} \times_{(n)} \boldsymbol{A})_{j_1 \times \cdots \times j_{n-1} \times i_n \times j_{n+1} \times \cdots \times j_N} = \sum_{j_n=1}^{J_n} g_{j_1,\ldots,j_n,\ldots,j_N} \cdot a_{i_n,j_n}$.

Definition 2 (R Kruskal Product): Given N matrices $\boldsymbol{B}^{(1)} \in \mathbb{R}^{J_1 \times R}, \cdots, \boldsymbol{B}^{(N)} \in \mathbb{R}^{J_N \times R}$, R Kruskal Product projects the N matrices $\boldsymbol{B}^{(n)}$ to a new N-order tensor $\widehat{\boldsymbol{\mathcal{G}}} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$ where $\widehat{\boldsymbol{\mathcal{G}}} = \sum_{r=1}^{R} \boldsymbol{b}^{(1)}_{:,r} \circ \cdots \circ \boldsymbol{b}^{(N)}_{:,r}$.

Definition 3 (R Dot Product): Given a matrix $\boldsymbol{A} \in \mathbb{R}^{M \times R}$, and a matrix $\boldsymbol{B} \in \mathbb{R}^{R \times M}$, R Dot Product projects $\boldsymbol{A}$ and $\boldsymbol{B}$ to a new matrice $(\boldsymbol{A} \odot \boldsymbol{B}) \in \mathbb{R}^{M \times 1}$ according to the coordinates, where $(\boldsymbol{A} \odot \boldsymbol{B})_{m,1} = \boldsymbol{a}_{m,:} \cdot \boldsymbol{b}_{:,m}$.

Definition 4 (Hadamard Product): Given a matrix $\boldsymbol{A} \in \mathbb{R}^{M \times N}$ and a matrix $\boldsymbol{B} \in \mathbb{R}^{M \times N}$, Hadamard Product projects $\boldsymbol{A}$ and $\boldsymbol{B}$ to a new matrice $(\boldsymbol{A} * \boldsymbol{B}) \in \mathbb{R}^{M \times N}$ according to the coordinates, where $(\boldsymbol{A} * \boldsymbol{B})_{m,n} = a_{m,n} \cdot b_{m,n}$.

Definition 5 (R Hadamard Product): Given a matrix $\boldsymbol{A} \in \mathbb{R}^{M \times 1}$ and a matrix $\boldsymbol{B} \in \mathbb{R}^{M \times R}$, Hadamard Product projects $\boldsymbol{A}$ and $\boldsymbol{B}$ to a new matrice $(\boldsymbol{A} \circledast \boldsymbol{B}) \in \mathbb{R}^{M \times R}$ according to the coordinates, where $(\boldsymbol{A} \circledast \boldsymbol{B})_{:,r} = \boldsymbol{A} * \boldsymbol{B}_{:,r}$.

C. Problem

Consider a sparse and incomplete N-order tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$. Let Ω denote the set of non-zero elements in $\boldsymbol{\mathcal{X}}$, and $|\Omega|$ represent the number of non-zero elements in Ω. The goal of Tensor Completion is to estimate the missing elements in $\boldsymbol{\mathcal{X}}$ based on the available non-zero elements in Ω. Sparse Tucker Decomposition is a widely used method for tensor completion, which involves finding N factor matrices $\boldsymbol{A}^{(1)} \in \mathbb{R}^{I_1 \times J_1}$, ..., $\boldsymbol{A}^{(N)} \in \mathbb{R}^{I_N \times J_N}$ along with an N-order core tensor $\boldsymbol{\mathcal{G}} \in \mathbb{R}^{J_1 \times \cdots \times J_N}$. The aim is to approximate the elements $x_{i_1,\ldots,i_N}$ of the tensor $\boldsymbol{\mathcal{X}}$ by $\widehat{x}_{i_1,\ldots,i_N}$ given by

$$x_{i_1,\ldots,i_N} \approx \widehat{x}_{i_1,\ldots,i_N} = \boldsymbol{\mathcal{G}} \times_{(1)} \boldsymbol{a}^{(1)}_{i_1,:} \times_{(2)} \cdots \times_{(N)} \boldsymbol{a}^{(N)}_{i_N,:} \quad (1)$$

and the missing elements can be predicted using (1).

Sparse FastTucker Decomposition is a variation of the Tucker Decomposition. It uses N core matrices $\boldsymbol{B}^{(1)} \in \mathbb{R}^{J_1 \times R}$, $\cdots$, $\boldsymbol{B}^{(N)} \in \mathbb{R}^{J_N \times R}$ to approximate the core tensor $\boldsymbol{\mathcal{G}}$ by:

$$\boldsymbol{\mathcal{G}} \approx \widehat{\boldsymbol{\mathcal{G}}} = \sum_{r=1}^{R} \boldsymbol{b}^{(1)}_{:,r} \circ \cdots \circ \boldsymbol{b}^{(N)}_{:,r} \quad (2)$$

which can reduce the space and computational complexity. In sum, the Sparse FastTucker decomposition is to find N factor matrices $\boldsymbol{A}^{(1)} \in \mathbb{R}^{I_1 \times J_1}$, ..., $\boldsymbol{A}^{(N)} \in \mathbb{R}^{I_N \times J_N}$ and N core matrices $\boldsymbol{B}^{(1)} \in \mathbb{R}^{J_1 \times R}$, ..., $\boldsymbol{B}^{(N)} \in \mathbb{R}^{J_N \times R}$, such that

$$\begin{aligned}
&\widehat{x}_{i_1,\ldots,i_N} \\
&= \left(\sum_{r=1}^{R} \boldsymbol{b}^{(1)}_{:,r} \circ \cdots \circ \boldsymbol{b}^{(N)}_{:,r} \right) \times_{(1)} \boldsymbol{a}^{(1)}_{i_1,:} \times_{(2)} \cdots \times_{(N)} \boldsymbol{a}^{(N)}_{i_N,:} \\
&= \sum_{r=1}^{R} (\boldsymbol{a}^{(1)}_{i_1,:} \boldsymbol{b}^{(1)}_{:,r}) \cdots (\boldsymbol{a}^{(N)}_{i_N,:} \boldsymbol{b}^{(N)}_{:,r}) \\
&= \sum_{r=1}^{R} \prod_{n=1}^{N} c^{(n)}_{i_n,r} \approx x_{i_1,\ldots,i_N}
\end{aligned} \quad (3)$$

where $\boldsymbol{C}^{(n)} = \boldsymbol{A}^{(n)} \boldsymbol{B}^{(n)} \in \mathbb{R}^{I_n \times R}$, and $c^{(n)}_{i_n,r}$ is the (i_n, r)th element of the $\boldsymbol{C}^{(n)}$. And we denote $\boldsymbol{A}^{(1)}$, ..., $\boldsymbol{A}^{(N)}$ and $\boldsymbol{B}^{(1)}$, ..., $\boldsymbol{B}^{(N)}$ using $\{\boldsymbol{A}^{(n)}\}$ and $\{\boldsymbol{B}^{(n)}\}$, respectively. And so on for other similar symbols. That is to solve the following non-convex optimization problem:

$$\begin{aligned}
&\underset{\{\boldsymbol{A}^{(n)}\},\{\boldsymbol{B}^{(n)}\}}{\arg\min} f\left(\boldsymbol{\mathcal{X}}, \{\boldsymbol{A}^{(n)}\}, \{\boldsymbol{B}^{(n)}\}\right) \\
&= \frac{1}{2} \sum_{x_{i_1,\ldots,i_N} \in \boldsymbol{\mathcal{X}}} \left\| x_{i_1,\ldots,i_N} - \widehat{x}_{i_1,\ldots,i_N} \right\|_F^2 \\
&\quad + \frac{\lambda_{\boldsymbol{A}}}{2} \sum_{n=1}^{N} \|\boldsymbol{A}^{(n)}\|_F^2 + \frac{\lambda_{\boldsymbol{B}}}{2} \sum_{n=1}^{N} \|\boldsymbol{B}^{(n)}\|_F^2
\end{aligned} \quad (4)$$

which adds regularization terms to prevent over fitting. Both $\lambda_{\boldsymbol{A}}$ and $\lambda_{\boldsymbol{B}}$ are hyperparameters.

III. Proposed Method

In this section, we propose FastTuckerPlus, an SGD-based non-convex optimization algorithm. FastTuckerPlus is a local search algorithm that demonstrates excellent convergence properties. It offers advantages in terms of computational overhead and memory access overhead. Moreover, it exhibits strong adaptability, allowing it to leverage the performance of Tensor Cores effectively. We provide a comprehensive description of the FastTuckerPlus algorithm in Section III-A, highlighting its key details. To optimize the calculation process for Tensor Cores and maximize their performance, we introduce matrixization in Section III-B. Lastly, we conduct a complexity analysis of FastTuckerPlus in the supplementary material.

A. A Non-Convex SGD-Based Sparse FastTucker Decomposition Algorithm

Our proposed FastTuckerPlus algorithm alternately solves two non-convex optimization objectives

$$\begin{aligned}
&\underset{\{\boldsymbol{a}^{(n)}_{i_n,:}\}}{\arg\min} f\left(\{\boldsymbol{a}^{(n)}_{i_n,:}\} \middle| x_{i_1,\ldots,i_N}, \{\boldsymbol{a}^{(n)}_{i_n,:}\}, \{\boldsymbol{B}^{(n)}\}\right) \\
&= \frac{1}{2} \left\| x_{i_1,\ldots,i_N} - \widehat{x}_{i_1,\ldots,i_N} \right\|_F^2 + \frac{\lambda_{\boldsymbol{A}}}{2} \sum_{n=1}^{N} \|\boldsymbol{a}^{(n)}_{i_n,:}\|_F^2
\end{aligned} \quad (5)$$

and

$$\begin{aligned}
&\underset{\{\boldsymbol{B}^{(n)}\}}{\arg\min} f\left(\{\boldsymbol{B}^{(n)}\} \middle| x_{i_1,\ldots,i_N}, \{\boldsymbol{a}^{(n)}_{i_n,:}\}, \{\boldsymbol{B}^{(n)}\}\right) \\
&= \frac{1}{2} \left\| x_{i_1,\ldots,i_N} - \widehat{x}_{i_1,\ldots,i_N} \right\|_F^2 + \frac{\lambda_{\boldsymbol{B}}}{2} \sum_{n=1}^{N} \|\boldsymbol{B}^{(n)}\|_F^2
\end{aligned} \quad (6)$$

Although the above two optimization objectives are non-convex, they exhibit a relatively simple optimization landscape, allowing them to be effectively solved using local search methods by

$$\begin{cases}
\boldsymbol{a}^{(1)}_{i_1,:} \leftarrow \boldsymbol{a}^{(1)}_{i_1,:} + \gamma_{\boldsymbol{A}} \left(e_{i_1,\ldots,i_N} \boldsymbol{d}^{(1)}_{i_1,:} \boldsymbol{B}^{(1)\top} - \lambda_{\boldsymbol{A}} \boldsymbol{a}^{(1)}_{i_1,:} \right) \\
\quad \cdots \\
\boldsymbol{a}^{(N)}_{i_N,:} \leftarrow \boldsymbol{a}^{(N)}_{i_N,:} + \gamma_{\boldsymbol{A}} \left(e_{i_1,\ldots,i_N} \boldsymbol{d}^{(N)}_{i_N,:} \boldsymbol{B}^{(N)\top} - \lambda_{\boldsymbol{A}} \boldsymbol{a}^{(N)}_{i_N,:} \right)
\end{cases} \quad (7)$$

for optimization objective (5) and

$$\begin{cases}
\boldsymbol{B}^{(1)} \leftarrow \quad \boldsymbol{B}^{(1)} + \gamma_{\boldsymbol{B}} \left(e_{i_1,\ldots,i_N} \boldsymbol{a}^{(1)\top}_{i_1,:} \boldsymbol{d}^{(1)}_{i_1,:} - \lambda_{\boldsymbol{B}} \boldsymbol{B}^{(1)} \right) \\
\quad \cdots \\
\boldsymbol{B}^{(N)} \leftarrow \quad \boldsymbol{B}^{(N)} + \gamma_{\boldsymbol{B}} \left(e_{i_1,\ldots,i_N} \boldsymbol{a}^{(N)\top}_{i_N,:} \boldsymbol{d}^{(N)}_{i_N,:} - \lambda_{\boldsymbol{B}} \boldsymbol{B}^{(N)} \right)
\end{cases} \quad (8)$$

for optimization objective (6), where $e_{i_1,\ldots,i_N} = x_{i_1,\ldots,i_N} - \widehat{x}_{i_1,\ldots,i_N}$. And in our approach, we define $\boldsymbol{d}^{(n)}_{i_n,:}$ as the notation for the product of $\boldsymbol{c}^{(1)}_{i_1,:} * \cdots * \boldsymbol{c}^{(n-1)}_{i_{n-1},:} * \boldsymbol{c}^{(n+1)}_{i_{n+1},:} * \cdots * \boldsymbol{c}^{(N)}_{i_N,:}$ for the element $x_{i_1,\ldots,i_N}$. For any given element $x_{i_1,\ldots,i_N} \in \Omega$, the updates of $\{\boldsymbol{a}^{(n)}_{i_n,:}\}$ in (7) are independent of each other. Similarly, the updates of $\{\boldsymbol{B}^{(n)}\}$ in (8) are also independent of each other. This implies that only one element needs to be selected for each update. However, the subset Ψ can still be selected from Ω, and

Algorithm 1: Sparse FastTuckerPlus Algorithm.

Input: Sparse tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, ranks $\{J_n\}$ and R, learning rates $\gamma_{\boldsymbol{A}}$ and $\gamma_{\boldsymbol{B}}$, regularization parameters $\lambda_{\boldsymbol{A}}$ and $\lambda_{\boldsymbol{B}}$, and iterations T.
Output: Factor matrices $\{\boldsymbol{A}^{(n)}\}$ and core matrices $\{\boldsymbol{B}^{(n)}\}$.

```
Initialize factor matrices {A^(n) ∈ R^{I_n × J_n}} and core matrices {B^(n) ∈ R^{J_n × R}}.
for t from 1 to T do
    for randomly takes Ψ from Ω do
        Calculate {c^(n)_{i_n,:}}.
        for n from 1 to N do
            Update a^(n)_{i_n,:} by rule (7).
        end for
    end for
    for randomly takes Ψ from Ω do
        Calculate {c^(n)_{i_n,:}}.
        for n from 1 to N do
            Update B^(n) by rule (8).
        end for
    end for
end for
```

the updates can be performed one by one according to the elements in Ψ. Furthermore, for a given $x_{i_1,\ldots,i_N}$, its corresponding $\{\boldsymbol{c}^{(n)}_{i_n,:}\}$ are the same. Therefore, the $\{\boldsymbol{c}^{(n)}_{i_n,:}\}$ can be precalculated to avoid redundant computations when calculating $\{\boldsymbol{d}^{(n)}_{i_n,:}\}$. The unified update of $\{\boldsymbol{a}^{(n)}_{i_n,:}\}$ or $\{\boldsymbol{B}^{(n)}\}$ ensures that $\{\boldsymbol{c}^{(n)}_{i_n,:}\}$ follows the updates. Therefore, maintaining $\{\boldsymbol{C}^{(n)}\}$ in memory does not reduce the computational overhead. The Sparse FastTuckerPlus Algorithm is described in Algorithm 1.

In Algorithm 1, when updating $\{\boldsymbol{a}^{(n)}_{i_n}\}$ or $\{\boldsymbol{B}^{(n)}\}$ using a single Ψ with M selected samples, the following data need to be read from memory: a matrix set consisting of N matrices of size $J_n \times R$ ($\{\boldsymbol{B}^{(n)}\}$) and a matrix set consisting of N matrices of size $M \times J_n$ ($\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$. A total of $(M+R)\sum_{n=1}^{N} J_n$ parameters are read from memory. And $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$ requires $|\Omega|\sum_{n=1}^{N} J_n R$ multiplications. After calculating all $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$, they are shared by all $\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$ during the computation of $\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$. Hence, the total number of required multiplications in the entire process is $MR(\sum_{n=1}^{N} J_n + N(N-2))$.

B. Matrixization

To leverage the computational capabilities of Tensor Cores, we can express the update rules (7) and (8) in the form of matrix calculations. We randomly select a subset Ψ from Ω containing M elements and represent it as a matrix $\boldsymbol{X}_\Psi \in \mathbb{R}^{M\times 1}$, where $x_{m,1}$ corresponds to the m-th element in Ψ. We use $\Psi^{(n)}$ to denote the index set of elements in Ψ for the n-th order, and $\Psi^{(n)}_i$ to denote the i-th index in $\Psi^{(n)}$. To form the factor sub-matrix $\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}$, we extract the factor vector with index set $\Psi^{(n)}$ from the factor matrix $\boldsymbol{A}^{(n)}$. Then, we define $\boldsymbol{C}^{(n)}_{\Psi^{(n)},:} = \boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\boldsymbol{B}^{(n)} \in \mathbb{R}^{M\times R}$, where the $(\Psi^{(n)}_i, r)$-th element $c^{(n)}_{\Psi^{(n)}_i,r} = \boldsymbol{a}^{(n)}_{\Psi^{(n)}_i,r} \cdot \boldsymbol{b}^{(n)}_{:,r}$ of $\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}$. Similarly to $\boldsymbol{d}^{(n)}_{i_n,:}$, we use $\boldsymbol{D}^{(n)}_{\Psi^{(n)},:} \in \mathbb{R}^{M\times R}$ to denote $\boldsymbol{C}^{(1)}_{\Psi^{(1)},:} * \cdots * \boldsymbol{C}^{(n-1)}_{\Psi^{(n-1)},:} * \boldsymbol{C}^{(n+1)}_{\Psi^{(n+1)},:} * \cdots * \boldsymbol{C}^{(N)}_{\Psi^{(N)},:}$.

Then update rules (7) and (8) can be expressed as follows:

$$\begin{cases} \boldsymbol{A}^{(1)}_{\Psi^{(1)}} \leftarrow \boldsymbol{A}^{(1)}_{\Psi^{(1)},:} + \gamma_A \left((\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast (\boldsymbol{D}^{(1)}_{\Psi^{(1)},:} \cdot \boldsymbol{B}^{(1)^\top}) - \lambda_{\boldsymbol{A}} \boldsymbol{A}^{(1)}_{\Psi^{(1)},:} \right) \\ \quad \cdots \\ \boldsymbol{A}^{(N)}_{\Psi^{(N)}} \leftarrow \boldsymbol{A}^{(N)}_{\Psi^{(N)},:} + \gamma_A \left((\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast (\boldsymbol{D}^{(N)}_{\Psi^{(N)},:} \cdot \boldsymbol{B}^{(N)^\top}) - \lambda_{\boldsymbol{A}} \boldsymbol{A}^{(N)}_{\Psi^{(N)},:} \right) \end{cases} \tag{9}$$

where $\widehat{\boldsymbol{X}}_\Psi = \boldsymbol{A}^{(1)}_{\Psi^{(1)},:} \odot (\boldsymbol{B}^{(1)}\boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:}) \in \mathbb{R}^{M\times 1}$, and

$$\begin{cases} \boldsymbol{B}^{(1)} \leftarrow \boldsymbol{B}^{(1)} + \gamma_B \left(\left((\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast \boldsymbol{A}^{(1)}_{\Psi^{(1)},:} \right)^\top \cdot \boldsymbol{D}^{(1)}_{\Psi^{(1)},:} - \lambda_{\boldsymbol{B}} \boldsymbol{B}^{(1)} \right) \\ \quad \cdots \\ \boldsymbol{B}^{(N)} \leftarrow \boldsymbol{B}^{(N)} + \gamma_B \left(\left((\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast \boldsymbol{A}^{(N)}_{\Psi^{(N)},:} \right)^\top \cdot \boldsymbol{D}^{(N)}_{\Psi^{(N)},:} - \lambda_{\boldsymbol{B}} \boldsymbol{B}^{(N)} \right) \end{cases} \tag{10}$$

where $\widehat{\boldsymbol{X}}_\Psi = \boldsymbol{C}^{(1)}_{\Psi^{(1)},:} \odot \boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:} \in \mathbb{R}^{M\times 1}$. And we use $\boldsymbol{E}^{(n)}_{\Psi^{(n)},:} \in \mathbb{R}^{M\times J_n}$ to denote $(\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast \boldsymbol{A}^{(N)}_{\Psi^{(N)},:}$. The matrix calculation process that can be accelerated by Tensor Cores is as follows: $\boldsymbol{A}^{(n)}_{\Psi^{(n)},:} \cdot \boldsymbol{B}^{(n)}$, $\boldsymbol{D}^{(n)}_{\Psi^{(n)},:} \cdot \boldsymbol{B}^{(n)^\top}$ and $\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:} \cdot \boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$. We also describe, in the supplementary material, how cuFastTucker, cuFasterTuckerCOO, and cuFasterTucker are reformulated into matrix operations to enable acceleration using Tensor Cores.

IV. cuFastTuckerPlusTC on GPU

We present the implementation of our proposed cuFastTuckerPlusTC algorithm on GPUs with Tensor Cores. In Section IV-A, we provide a brief introduction to Tensor Cores. We also explain how to optimize data partitioning to maximize the utilization of Tensor Cores in Section IV-B. Our algorithm is designed to leverage two levels of parallelism: warp parallelism, discussed in Section IV-C, and block parallelism, explained in Section IV-D. Finally, in Section IV-E, we summarize our algorithm and highlight the key techniques used in its implementation.

A. Tensor Core

Tensor Cores are specialized processing cores in NVIDIA GPUs that excel in performing matrix operations. Unlike CUDA Cores, which are more general-purpose, Tensor Cores are specifically designed for high-performance matrix computations. Utilizing Tensor Cores can significantly enhance the peak throughput compared to using CUDA Cores for matrix operations. Tensor Cores perform a fused multiply-add operation. They multiply two 4×4 half-precision float matrices, add the result to a 4×4 half-precision or single-precision matrix, and produce a new $4 \times$

4 half-precision or single-precision matrix. NVIDIA refers to these operations performed by Tensor Cores as mixed-precision math because the input matrices are in half-precision, but the product can be in full-precision. CUDA provides the Warp-level Matrix Multiply and Accumulate (WMMA) API, which allows developers to leverage Tensor Cores on the GPU. Through the WMMA API, developers can calculate $\boldsymbol{D} = \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$ within a warp, where $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$, and $\boldsymbol{D}$ can also be tiles of larger matrices. All threads within the warp cooperate to perform their respective matrix multiply-add operations. The size limit of matrices in WMMA is $M \times N \times K$, where $\boldsymbol{A} \in \mathbb{R}^{M \times K}$, $\boldsymbol{B} \in \mathbb{R}^{N \times K}$, $\boldsymbol{C} \in \mathbb{R}^{M \times N}$, and $\boldsymbol{D} \in \mathbb{R}^{M \times N}$. Currently, CUDA supports matrix multiplication and addition operations of various numerical precisions, including $16 \times 16 \times 16$ half-precision, $16 \times 16 \times 16$ single-precision, and $8 \times 8 \times 4$ double-precision, etc. For larger matrices, the matrix multiply-add operations can be divided into multiple tile-wise operations of suitable sizes. In our paper, without loss of generality, we utilize $16 \times 16 \times 16$ single-precision matrix multiply-add operations.

B. Matrix Partition

To maximize performance and efficiency, we divide $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\}$ and $\{\boldsymbol{B}^{(n)}\}$ into multiple tiles of submatrices with a size of 16×16. Any remaining entries are padded with zeros. It is worth noting that the best performance can be achieved when M, R and $\{J_n\}$ are all multiples of 16. For the sake of convenience and to save memory costs, we set M to be 16. Let $\{J_n = 16P_n\}$, and $R = 16Q$. As a result, $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\}$ can be divided into $\{1 \times P_n\}$ tiles, and $\{\boldsymbol{B}^{(n)}\}$ can be divided into $\{P_n \times Q\}$ tiles. Correspondingly, $\{\boldsymbol{D}^{(n)}_{\Psi^{(n)}}\}$, are divided into $\{1 \times Q\}$ tiles, and $\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)}}\}$ are divided into $\{1 \times P_n\}$ tiles, respectively.

In our notation, $(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}})_{1,p_n}$, represents the $(1, p_n)$-th tile of $\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}$. Similarly, $(\boldsymbol{B}^{(n)})_{p_n,q}$ represents the (p_n, q)-th tile of $\boldsymbol{B}^{(n)}$. The computation of $(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)})_{1,q}$ is given by $\sum_{p_n=1}^{P_n}(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}})_{1,p_n}(\boldsymbol{B}^{(n)})_{p_n,q}$, where $(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)})_{1,q}$ represents the $(1, q)$-th tile of the matrix product $\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)}$. Similarly, the computations $\boldsymbol{D}^{(n)}_{\Psi^{(n)}_{,:}} \cdot \boldsymbol{B}^{(n)^\top}$ and $\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)}_{,:}} \cdot \boldsymbol{B}^{(n)}$ follow the same pattern. Please note that in the paper, we have omitted explicit mention of matrix partitioning for brevity and clarity of the expressions.

C. Warp Parallelization

The warp serves as the scheduling unit in NVIDIA GPUs, with all threads in a warp executing the same instructions. In the current mainstream NVIDIA GPU architecture, a warp consists of 32 threads. The GPU assigns 32 consecutive threads from a block to form a warp. Even if the number of remaining threads is less than 32, they are still organized into a warp, leaving the excess threads idle. Therefore, proper task distribution is crucial to avoid idle threads within a warp. In the cuFastTuckerPlusTC framework, we utilize a warp to process a specific Ψ. When updating the factor matrices, the warp updates the $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\}$. This warp handles a tile of size 16×16 in each iteration. The calculations involving $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)}\}$ and $\{\boldsymbol{D}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)^\top}\}$ can be performed using either CUDA Cores or Tensor Cores. In the case of Tensor Cores, a warp is responsible for multiplying two tiles of size 16×16. The warp computes the elements $(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}})_{1,p_n}(\boldsymbol{B}^{(n)})_{p_n,q}$, and then computes $(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)})_{1,q} = \sum_{p_n=1}^{P_n}(\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}})_{1,p_n}(\boldsymbol{B}^{(n)})_{p_n,q}$. This process completes the calculation of $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)}\}$. The calculations involving $\{\boldsymbol{D}^{(n)}_{\Psi^{(n)}_{,:}}\boldsymbol{B}^{(n)^\top}\}$ follow a similar pattern. For CUDA Cores, we divide a warp into two workers, with each worker consisting of 16 threads. When multiplying two tiles of size 16×16, one worker calculates the upper half (8×16) of the resulting tile, while the other worker computes the lower half (8×16) of the tile. Other tile operations with a size of 16×16, which do not involve matrix multiplication, are also performed using CUDA Cores. Similar to matrix multiplication, each worker is responsible for half of the tile. When updating the core matrices, the usage of CUDA Cores and Tensor Cores aligns with the update process for the factor matrices. The distinction is that we do not update $\{\boldsymbol{B}^{(n)}\}$, within the warp. Instead, we accumulate the gradients of $\{\boldsymbol{B}^{(n)}\}$ for all Ψ in Ω and subsequently update $\{\boldsymbol{B}^{(n)}\}$.

D. Block Parallelization

A block consists of multiple threads that can synchronize and share shared memory, enabling communication between threads. When the number of threads in a block is a multiple of 32, meaning the block consists of multiple warps, all threads can be fully utilized without any idle threads. In cuFastTuckerPlusTC, since each Ψ can be processed independently and has the same number of elements, we evenly distribute all sampled Ψ to different warps to achieve load balancing. In hardware, all threads within a block do not execute in parallel simultaneously. Instead, a block is divided into multiple warps allocated to the same Streaming Multiprocessor and queued for execution on the Streaming Processors. The GPU's hardware architecture allows a large number of warps to reside in the SM concurrently. When a warp encounters a wait condition during execution, such as memory read and write delays, the Warp Scheduler swiftly switches to the next eligible warp for execution to maintain high instruction throughput. This architectural design is a fundamental characteristic of GPUs, where numerous warps are employed to hide latency. The abundance of warps enables the GPU to achieve exceptional data throughput.

E. Overview

cuFastTuckerPlusTC is divided into two main parts: updating the factor matrices and updating the core matrices, as described in Algorithms 2 and 3, respectively. In these algorithms, we prioritize storing the parameters in the fastest memory available and maximizing memory reuse. We ensure that shared memory and registers are not excessively occupied, allowing for a sufficient

Algorithm 2: Update Factor Matrices of cuFastTuckerPlusTC.

$\mathcal{G}\{parameter\}$: parameters in global memory.
$\mathcal{R}\{parameter\}$: parameters in register memory.
Input: Sparse tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, initialized factor matrices $\{\boldsymbol{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}\}$ and core matrices $\{\boldsymbol{B}^{(n)} \in \mathbb{R}^{J_n \times R}\}$, learning rate $\gamma_{\boldsymbol{A}}$, regularization parameter $\lambda_{\boldsymbol{A}}$.
Output: Factor matrices $\{\boldsymbol{A}^{(n)}\}$.

for n from 1 to N **do**
  $\mathcal{R}\{\boldsymbol{B}^{(n)}\} \leftarrow \mathcal{G}\{\boldsymbol{B}^{(n)}\}$
end for
for *Warp Parallelization* **do**
  for randomly take Ψ from Ω **do**
    for n from 1 to N **do**
      $\mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow 1$
    end for
    for n from 1 to N **do**
      $\mathcal{R}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{G}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$
      $\mathcal{S}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \cdot \mathcal{R}\{\boldsymbol{B}^{(n)}\}$ by Tensor Cores
      for k from 1 to N and $k \neq n$ **do**
        $\mathcal{R}\{\boldsymbol{D}^{(k)}_{\Psi^{(k)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{D}^{(k)}_{\Psi^{(k)},:}\} * \mathcal{S}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$
      end for
    end for
    for n from 1 to N **do**
      $\mathcal{R}\{\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$
      $\mathcal{S}\{\boldsymbol{B}^{(n)}\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{B}^{(n)}\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{B}^{(n)}\} \cdot \mathcal{R}\{\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}\}$ by Tensor Cores
    end for
    $\mathcal{R}\{\widehat{\boldsymbol{X}}_\Psi\} \leftarrow \mathcal{S}\{\boldsymbol{A}^{(1)}_{\Psi^{(1)},:}\} \odot \mathcal{S}\{\boldsymbol{B}^{(1)}\boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:}\}$ - $\mathcal{G}\{\boldsymbol{X}_\Psi\}$
    for n from 1 to N **do**
      $\mathcal{G}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} - \gamma_{\boldsymbol{A}} \cdot (\ \mathcal{R}\{\widehat{\boldsymbol{X}}_\Psi\} \circledast \mathcal{S}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\boldsymbol{B}^{(n)^\top}\} + \lambda_{\boldsymbol{A}} \cdot \mathcal{S}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}\ )$
    end for
  end for
end for

Algorithm 3: Update Core Matrices of cuFastTuckerPlusTC.

$\mathcal{G}\{parameter\}$: parameters in global memory.
$\mathcal{R}\{parameter\}$: parameters in register memory.
$Grad(\boldsymbol{B}^{(n)})$: the gradient of $\boldsymbol{B}^{(n)}$.
Input: Sparse tensor $\boldsymbol{\mathcal{X}} \in \mathbb{R}^{I_1 \times \cdots \times I_N}$, initialized factor matrices $\{\boldsymbol{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}\}$ and core matrices $\{\boldsymbol{B}^{(n)} \in \mathbb{R}^{J_n \times R}\}$, learning rate $\gamma_{\boldsymbol{B}}$, regularization parameter $\lambda_{\boldsymbol{B}}$.
Output: Core matrices $\{\boldsymbol{B}^{(n)}\}$.

for n from 1 to N **do**
  $\mathcal{R}\{\boldsymbol{B}^{(n)}\} \leftarrow \mathcal{G}\{\boldsymbol{B}^{(n)}\}$
  $\mathcal{G}\{Grad(\boldsymbol{B}^{(n)})\} \leftarrow 0$
end for
for *Warp Parallelization* **do**
  for randomly take Ψ from Ω **do**
    for n from 1 to N **do**
      $\mathcal{R}\{Grad(\boldsymbol{B}^{(n)})\} \leftarrow 0$
      $\mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow 1$
    end for
    for n from 1 to N **do**
      $\mathcal{R}\{\boldsymbol{A}^{(n)^\top}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{G}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$
      $\mathcal{S}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)}}\} \leftarrow \mathcal{R}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\} \cdot \mathcal{R}\{\boldsymbol{B}^{(n)}\}$ by Tensor Cores
      for k from 1 to N and $k \neq n$ **do**
        $\mathcal{R}\{\boldsymbol{D}^{(k)}_{\Psi^{(k)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{D}^{(k)}_{\Psi^{(k)},:}\} * \mathcal{S}\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$
      end for
    end for
    $\mathcal{R}\{\widehat{\boldsymbol{X}}_\Psi\} \leftarrow \mathcal{S}\{\boldsymbol{C}^{(1)}_{\Psi^{(1)},:}\} \odot \mathcal{R}\{\boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:}\}$ - $\mathcal{G}\{\boldsymbol{X}_\Psi\}$
    for n from 1 to N **do**
      $\mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$
      $\mathcal{R}\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\} \leftarrow \mathcal{S}\{\boldsymbol{E}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\widehat{\boldsymbol{X}}_\Psi\} \circledast \mathcal{S}\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$
      $\mathcal{S}\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\} \leftarrow \mathcal{R}\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\} \cdot \mathcal{R}\{\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$ by Tensor Cores
      $\mathcal{R}\{Grad(\boldsymbol{B}^{(n)})\} \leftarrow \mathcal{R}\{Grad(\boldsymbol{B}^{(n)})\} + \mathcal{S}\{\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\}$
    end for
  end for
  for n from 1 to N **do**
    $\mathcal{G}\{Grad(\boldsymbol{B}^{(n)})\} \leftarrow \mathcal{G}\{Grad(\boldsymbol{B}^{(n)})\} + \mathcal{R}\{Grad(\boldsymbol{B}^{(n)})\}$
  end for
end for
for n from 1 to N **do**
  $\mathcal{G}\{\boldsymbol{B}^{(n)}\} \leftarrow \mathcal{G}\{\boldsymbol{B}^{(n)}\} - \gamma_{\boldsymbol{B}} \cdot (\ \mathcal{G}\{Grad(\boldsymbol{B}^{(n)}) / |\Omega| + \lambda_{\boldsymbol{B}} \cdot \mathcal{G}\{\boldsymbol{B}^{(n)}\}\ )$
end for

number of active thread blocks during runtime to enhance parallel efficiency. Furthermore, we strive to minimize unnecessary calculations throughout the computation process. Additionally, cuFastTuckerPlusTC utilizes a load-balanced sampling method. In summary, the main optimization techniques employed by cuFastTuckerPlusTC are as follows:

Memory Coalescing: Global Memory access latency on GPUs can be a significant performance bottleneck, often taking hundreds of clock cycles. To mitigate this issue, GPU architectures employ parallelization techniques to enhance memory access throughput. When a specific location in global memory is accessed, a sequence of consecutive locations that includes the target location is fetched in a single operation. This approach leverages the fact that threads within a warp execute the same instructions simultaneously. In cuFastTuckerPlusTC, each warp is assigned the responsibility of handling one or more tiles. To optimize memory access patterns, $\{\boldsymbol{A}^{(n)}\}$ and $\{\boldsymbol{B}^{(n)}\}$ are stored in global memory in row-major order and column-major order, respectively. This arrangement ensures that when a warp reads or writes $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$ or $\{\boldsymbol{B}^{(n)}\}$, respectively, it minimizes the number of memory requests required.

Warp Shuffle: The Warp Shuffle instruction is a feature that enables direct data exchange between threads within the same warp. It utilizes dedicated hardware, has low latency, and does not require additional memory space. Compared to using shared memory for data exchange, utilizing Warp Shuffle is more efficient. Furthermore, the reduced shared memory usage allows for allocating the saved resources to the on-chip L1 cache, enhancing overall performance. However, it's important to note that data exchange between different warps within the same block still necessitates the use of shared memory. In cuFastTuckerPlusTC, we leverage the Warp Shuffle instruction for matrix multiplication and dot product calculations. For example, we utilize it in expressions such as $\boldsymbol{A}^{(1)}_{\Psi^{(1)},:} \odot (\boldsymbol{B}^{(1)}\boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:}) \in \mathbb{R}^{M\times 1}$ (line 20 of Algorithm 2), $\boldsymbol{C}^{(1)}_{\Psi^{(1)},:} \odot \boldsymbol{D}^{(1)^\top}_{\Psi^{(1)},:}$ (line 18 of Algorithm 3), $(\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast (\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}\boldsymbol{B}^{(n)^\top})$ (line 22 of Algorithm 2) and $(\boldsymbol{X}_\Psi - \widehat{\boldsymbol{X}}_\Psi) \circledast \boldsymbol{A}^{(n)}_{\Psi^{(n)},:}$ (line 21 of Algorithm 3).

WMMA API: The utilization of the WMMA API enables us to leverage Tensor Cores for performing matrix multiplication computations within the warp. Initially, the matrices are read from global memory or shared memory into the registers of the warp. However, it's important to note that access to the entire matrix can only be made at the warp level, and accessing specific elements within the matrix is not possible at the thread level. Once the Tensor Cores complete their calculations, the results stored in registers can be written back to global memory or shared memory. In cuFastTuckerPlusTC, we leverage Tensor Cores to compute the following matrix multiplications: $\boldsymbol{A}^{(n)}_{\Psi^{(n)},:} \cdot \boldsymbol{B}^{(n)}$ (line 11 of Algorithm 2 and line 13 of Algorithm 3), $\boldsymbol{B}^{(n)} \cdot \boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}$ (line 18 of Algorithm 2) and $\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:} \cdot \boldsymbol{B}^{(n)}$ (line 22 of Algorithm 3). These matrix multiplication operations are efficiently performed using the Tensor Cores. However, it's worth mentioning that these operations can also be carried out using Warp Shuffle with CUDA Cores. When employing CUDA Cores for tile multiplication, two workers are responsible for calculating the upper and lower parts of the tile, and each worker computes a dot product at a time.

Loop Unrolling: Loop unrolling is an instruction-level optimization technique that enhances code performance by sacrificing programming complexity. By unrolling loops in CUDA, we can reduce instruction costs and introduce more independently dispatched instructions. This approach increases the number of concurrent operations in the pipeline, leading to improved instruction and memory bandwidth. In our implementation, we utilize the $\#pragma\ unroll$ directive to perform loop unrolling, which doesn't significantly increase the program's complexity. Additionally, it's important to note that CUDA does not support register arrays, but fixed-length short arrays can be stored in registers. When an array is too long or has an indeterminate index, the compiler may store it in global memory. However, by unrolling the loop, we can eliminate the indeterminate index of the array, allowing the compiler to allocate the fixed-length short array into registers. This optimization technique enables faster access to arrays stored in registers rather than relying on shared memory for certain computational tasks.

Shared Memory: Shared memory is a high-bandwidth and low-latency on-chip memory located on the SM. It is allocated in blocks and shared by all threads within a block, persisting throughout the lifetime of the block. Shared memory serves multiple purposes, including reducing access to global memory by storing frequently accessed parameters (hotspot parameters) and facilitating communication among threads in different warps within the same block. In cuFastTuckerPlusTC, we allocate exclusive shared memory for each warp in the block. This shared memory is used to store various parameters, such as $\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}$ (line 10 of Algorithm 2 and line 12 of Algorithm 3), $\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}$ (line 11 of Algorithm 2 and line 13 of Algorithm 3), $\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$ (line 17 of Algorithm 2 and line 20 of Algorithm 3), $\boldsymbol{B}^{(n)}\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}$ (line 18 of Algorithm 2) and $\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$ (line 22 of Algorithm 3). These parameters are utilized as inputs and outputs of Tensor Cores. As these parameters are shared across different time steps, they do not consume a significant amount of shared memory.

Register: Registers are the fastest memory in the GPU, offering quick access to data for calculations, reading, and writing. It is crucial to allocate parameters efficiently to registers to maximize performance while ensuring enough warps survive in the Streaming Multiprocessor (SM) to maintain high data throughput. In cuFastTuckerPlusTC, we prioritize frequently used parameters and store them in registers without occupying excessive register space. Intermediate variables and the results of computing $\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$ (lines 7, 13 of Algorithm 2 and lines 9, 15 of Algorithm 3) are stored in registers. We also use registers to store and accumulate the gradient of $\boldsymbol{B}^{(n)}$ (line 8 of Algorithm 3), which is later written to global memory to minimize global memory writes. Furthermore, we utilize registers with Tensor Cores to store $\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}$ (line 10 of Algorithm 2 and line 12 of Algorithm 3), $\boldsymbol{B}^{(n)}$ (line 2 of Algorithm 2 and line 2 of Algorithm 3) and $\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$. Correspondingly, the calculated $\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}$ (line 11 of Algorithm 2 and line 13 of Algorithm 3), $\boldsymbol{B}^{(n)}\boldsymbol{D}^{(n)^\top}_{\Psi^{(n)},:}$ (line 18 of Algorithm 2) and $\boldsymbol{E}^{(n)^\top}_{\Psi^{(n)},:}\boldsymbol{D}^{(n)}_{\Psi^{(n)},:}$ (line 22 of Algorithm 3) are also stored in registers with Tensor Cores. Additionally, other intermediate processes within the calculation process are performed in registers to take advantage of their speed and minimize memory access.

Read-only Data Cache: In cuFastTuckerPlusTC, we take advantage of the fact that the values of $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$ remain unchanged between the warp reading and updating stages. To optimize memory access and reduce shared memory usage, we store $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$ (line 10 of Algorithm 2 and line 12 of Algorithm 3) in the read-only cache. The read-only cache, also known as the constant cache, is a specialized cache in NVIDIA GPUs designed for storing read-only data. By storing $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)},:}\}$ in the read-only cache, we enable faster memory access for the warp reading stage. To optimize read-only memory access, we utilize the $\_\_ldg$ modifier in NVIDIA GPUs. The $\_\_ldg$ modifier is a CUDA compiler intrinsic that hints the compiler to use the read-only cache for memory loads. It allows for more efficient and optimized access to read-only memory, improving performance in scenarios where the data remains unchanged. By leveraging the read-only cache and utilizing the $\_\_ldg$ modifier, cuFastTuckerPlusTC enhances memory access efficiency when

reading $\{\boldsymbol{A}^{(n)}_{\Psi^{(n)}_{,:}}\}$ contributing to overall performance optimization.

V. Experiments

We present experimental results to answer the following questions.

1) *Convergence of the algorithms:* Does the non-convex SGD algorithm cuFastTuckerPlusTC exhibit superior convergence properties compared to convex optimization algorithms?
2) *Accuracy and Runtime under Varying Numerical Precisions:* What is the impact of different numerical precisions on the accuracy and running time of cuFastTuckerPlusTC?
3) *Single Iteration Running Time:* Compared with convex optimization SGD algorithms, does the non-convex SGD algorithm cuFastTuckerPlusTC require less computation time per iteration?
4) *Memory Access Overhead:* Compared with convex optimization SGD algorithms, does the non-convex SGD algorithm cuFastTuckerPlusTC offer advantages in terms of memory access overhead?
5) *Tensor Cores' Impact:* How does cuFastTuckerPlusTC's acceleration performance fare when utilizing Tensor Cores? Additionally, what is the acceleration performance of other algorithms when employing Tensor Cores?
6) *Calculation or Storage:* Which approach yields superior performance: precomputing and storing the intermediate matrices $\{\boldsymbol{C}^{(n)}\}$ in advance, or computing $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)}_{,:}}\}$ temporarily when needed?

We provide a detailed description of the datasets and experimental settings in Section V-A. The questions posed in this paper are addressed as follows: the convergence of the algorithms is investigated in Section V-B, the accuracy and runtime of cuFastTuckerPlusTC under various numerical precisions are analyzed in Section V-C, the single iteration running time of the algorithms is analyzed in Section V-D, the memory access overhead is evaluated in Section V-E, the effect of the Tensor Cores on cuFastTuckerPlusTC and other algorithms is examined in Section V-F, the choice between calculation and storage approaches is discussed in Section V-G.

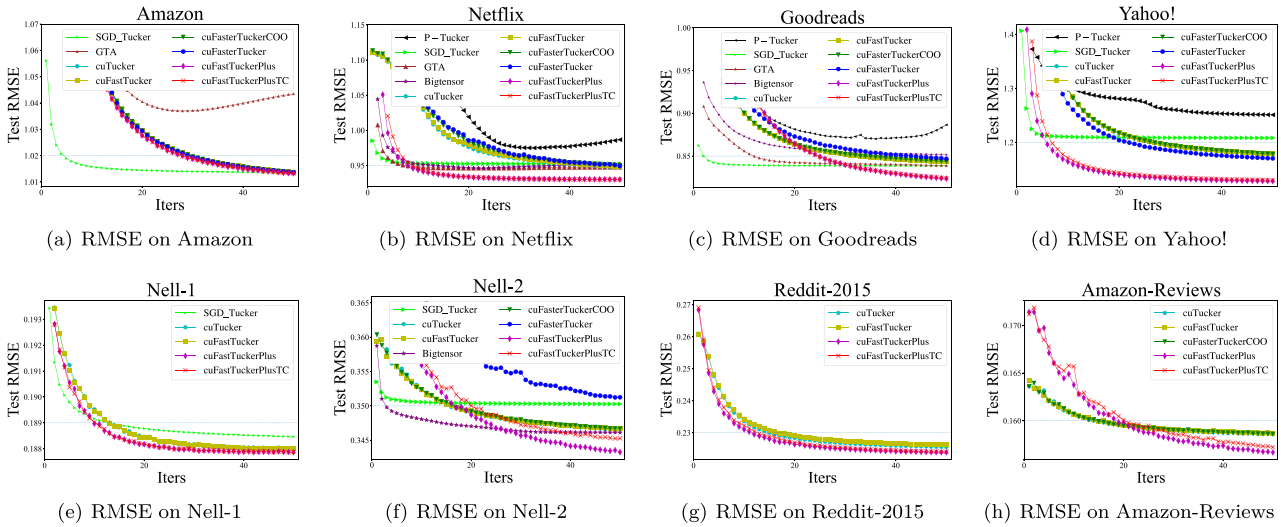
A. Experimental Settings

1) *Datasets:* We utilize a combination of Real-World datasets and synthetic datasets to evaluate the convergence and performance of the algorithms. For Real-World datasets, we evaluate our method on eight widely used sparse tensor datasets, encompassing four recommendation system datasets, two knowledge graph datasets, and two text processing datasets. The recommendation system datasets include: Amazon[1] [42], Netflix[2] , Goodreads[3] [43], [44] and Yahoo![4] . The knowledge graph datasets include: Nell-1[5] [45] and Nell-2[6] [45]. And the text processing datasets include: Amazon-Reviews [7] [46] and Reddit-2015[8] [47]. All datasets are divided into a trainset, denoted as Ω, and a testset, denoted as Γ. We train the parameters on the trainset Ω and evaluate their performance on the testset Γ. For further details on the characteristics of the real-world datasets, please refer to Tables III. Real-world datasets are employed to evaluate the algorithm's capability and efficiency in processing large-scale sparse tensors in applications such as recommender systems, knowledge graphs, and text processing. In addition to the Real-World datasets, we construct synthetic datasets consisting of 8 sparse tensors with orders ranging from 3 to 10. Each tensor contains a total of 100,000,000 elements, with each dimension having a size of 10,000. Synthetic datasets are employed to evaluate the algorithm's capability and efficiency in processing high-order sparse tensors.
2) *Contrasting algorithms:* In this paper, we compare the proposed cuFastTuckerPlusTC algorithm with sparse Tucker decomposition algorithms, including P-Tucker [16], Vest [17], SGD_Tucker [18], ParTi! [15], GTA [19], Bigtensor [20] and cuTucker [21]. In addition, we evaluate cuFastTuckerPlusTC against state-of-the-art sparse FastTucker decomposition algorithms, such as cuFastTucker [21], cuFasterTucker [22], and cuFasterTuckerCOO [22]. To enable a fair comparison, we implemented three Tensor Core–accelerated variants: cuFastTuckerTC, cuFasterTuckerTC, and cuFasterTuckerCOOTC, which enhance cuFastTucker, cuFasterTucker, and cuFasterTuckerCOO, respectively. Similarly, we implemented the FastTuckerPlus algorithm (cuFastTuckerPlus) using CUDA cores.
3) *Environment:* The implementation of cuFastTuckerPlus/cuFastTuckerPlusTC is done in C/C++ using CUDA. The experiments for P-Tucker, Vest, SGD_Tucker were conducted on an *Intel Core i7-12700 K CPU (5.00 GHz, 20 threads)* with 64 GB RAM. In contrast, the experiments for ParTi!, cuTucker, cuFastTucker, cuFasterTucker, cuFasterTuckerCOO, cuFastTuckerPlus, cuFastTuckerTC, cuFasterTuckerTC, cuFasterTuckerCOOTC, and cuFastTuckerPlusTC were conducted on an *NVIDIA GeForce RTX 3080Ti GPU*, which is equipped with 10,240 CUDA cores, 320 Tensor Cores, and 12 GB of graphics memory. The GTA and Bigtensor algorithms were evaluated on a heterogeneous platform consisting of the above CPU and GPU. In terms of numerical precision, P-Tucker, Vest, SGD_Tucker, and ParTi! utilize double-precision, while GTA and BigTensor use single-precision. The computational platforms and precision of these algorithms are constrained by their available open-source implementations. The *Intel Core i7-12700 K* and *NVIDIA*

[1] https://nijianmo.github.io/amazon/index.html
[2] https://www.netflixprize.com/
[3] https://mengtingwan.github.io/data/goodreads
[4] https://webscope.sandbox.yahoo.com/
[5] https://frostt.io/tensors/nell-1/
[6] https://frostt.io/tensors/nell-2/
[7] https://frostt.io/tensors/amazon-reviews/
[8] https://frostt.io/tensors/reddit-2015/

TABLE III
REAL-WORLD DATASETS

	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2	Reddit-2015	Amazon-Reviews
I_1	2, 628, 801	480, 189	816, 371	1, 000, 990	2, 902, 330	12, 092	8, 211, 298	4, 821, 207
I_2	4, 529, 425	17, 770	2, 325, 541	624, 961	2, 143, 368	9, 184	176, 962	1, 774, 269
I_3	7, 813	2, 182	4, 774	3, 075	25, 495, 389	28, 818	8, 116, 559	1, 805, 187
$\|\Omega\|$	111, 240, 135	99, 072, 112	103, 500, 736	250, 272, 286	142, 166, 596	76, 109, 392	4, 640, 599, 395	1, 724, 388, 676
$\|\Gamma\|$	1, 135, 628	1, 408, 395	1, 050, 813	2, 527, 989	1, 432, 956	770, 027	46, 874, 686	17, 420, 342
Max	5	5	5	5	6.102893	6.542442	7.238994	4.929572
Min	1	1	1	0.025	1	1	1	1



(a) RMSE on Amazon (b) RMSE on Netflix (c) RMSE on Goodreads (d) RMSE on Yahoo!

(e) RMSE on Nell-1 (f) RMSE on Nell-2 (g) RMSE on Reddit-2015 (h) RMSE on Amazon-Reviews

Fig. 1. The convergence curves of cuFastTuckerPlusTC and other algorithms on the Real-World datasets.

RTX 3080 Ti represent the same generation of high-end hardware and are highly compatible within heterogeneous computing systems. Using this combination as the test platform ensures a relatively fair comparison. Due to hardware limitations, the algorithm can access up to 64 GB of CPU memory and 16 GB of GPU memory. Additionally, we restrict the execution time of a single iteration to within one hour. Unless otherwise specified, GPU-based algorithms such as cuTucker, cuFastTucker, cuFasterTucker, cuFasterTuckerCOO, and cuFastTuckerPlus are executed in single-precision, whereas cuFastTuckerTC, cuFasterTuckerTC, cuFasterTuckerCOOTC, and cuFastTuckerPlusTC utilize Tensor Cores with $16 \times 16 \times 16$ single-precision matrix operations. All algorithms utilize optimal parameters and are executed without interference from other tasks.

B. Convergence of the Algorithms

We utilize the Root Mean Square Error (RMSE) on the test datasets Γ to evaluate the accuracy of all algorithms. To ensure fairness, all methods are initialized with the same random seed, and RMSE values are recorded after each iteration. For consistency, we set the Tucker core sizes to $\{J_n = 16\}$ and the rank to $R = 64$ across all algorithms. Each algorithm is executed for a total of 50 iterations. Fig. 1 presents the convergence curves of cuFastTuckerPlusTC and other baseline algorithms on the real-world datasets. Table IV reports the time required for cuFastTuckerPlusTC and other baseline algorithms to reach the baseline RMSE on real-world datasets (subject to hardware constraints: 12 GB of GPU memory, 64 GB of RAM, and a one-hour time limit). The baseline test RMSE for each dataset are as follows: Amazon: 1.02, Netflix: 0.95, Goodreads: 0.85, Yahoo!: 1.20, NELL-1: 0.189, NELL-2: 0.35, Reddit-2015: 0.23, Amazon-Reviews: 0.16. It is important to note that only algorithms which converge to the baseline RMSE are included in Fig. 1. Notably, cuFastTuckerPlus and cuFastTuckerPlusTC exhibit faster convergence (Table IV) compared to the other algorithms, while also achieving greater accuracy (Fig. 1). Compared with baseline algorithms, cuFastTuckerPlus achieves speedups ranging from $2\times$ to $8\times$ on real-world datasets. These results indicate that Sparse FastTucker decomposition benefits from a simple convergence landscapes, and the non-convex optimization-based SGD approach leads to faster convergence. While our approach uses single-precision arithmetic to improve computational efficiency, some of the compared baselines adopt double precision. Although this difference could lead to minor numerical discrepancies, we observed that the impact on the overall convergence trend and reconstruction quality is negligible in our experiments.

C. Accuracy and Running Time in Various Numerical Precisions

The running time and accuracy of cuFastTuckerPlusTC are influenced by the choice of numerical precision. Under the

TABLE IV
TIME (IN SECONDS) REQUIRED FOR SPARSE TUCKER DECOMPOSITION ALGORITHMS TO CONVERGE TO THE BENCHMARK RMSE ON THE REAL-WORLD DATASETS, OR THE REASON FOR FAILURE TO OBTAIN RESULTS. HERE, 'NC' DENOTES 'NON-CONVERGENCE', 'OOT' DENOTES 'OUT OF TIME', AND 'OOM' DENOTES 'OUT OF MEMORY'

Algorithm	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2	Redddit-2015	Amazing-Reviews
P-Tucker	NC	NC	NC	NC	OOM	OOM	OOM	OOM
Vest	OOT	OOT	OOT	OOT	OOT	OOT	OOT	OOT
SGD_Tucker	384.50 (62.41×)	NC	140.40 (25.00×)	NC	1288.30 (407.08×)	NC	NC	NC
GTA	NC	693.21 (365.52×)	2107.49 (375.21×)	OOM	OOM	NC	OOM	OOM
Bigtensor	NC	166.40 (87.74×)	NC	OOM	OOM	27.84 (7.87×)	OOM	OOM
ParTi!	OOM	NC	NC	OOM	OOM	NC	OOM	OOM
cuTucker	283.42 (46.00×)	395.87 (208.74×)	289.39 (51.52×)	571.04 (144.60×)	168.89 (53.37×)	102.85 (29.08×)	7045.95 (42.39×)	2448.26 (16.79×)
cuFastTucker	104.29 (16.93×)	146.51 (77.25×)	107.40 (19.12×)	212.34 (53.77×)	57.63 (18.21×)	40.40 (11.42×)	2887.64 (17.37×)	987.31 (6.77×)
cuFasterTuckerCOO	18.48 (3.00X)	NC	17.55 (3.13×)	32.47 (8.22×)	OOM	5.40 (1.53×)	OOM	171.03 (1.17×)
cuFasterTucker	15.84 (2.57×)	12.45 (6.56×)	17.77 (3.16×)	15.47 (3.92×)	OOM	NC	OOM	OOM
cuFastTuckerPlus	92.46 (15.01×)	21.33 (11.25×)	70.17 (12.49×)	40.16 (10.17×)	38.55 (12.18×)	35.64 (10.08×)	1665.12 (10.02×)	1008.63 (6.92×)
cuFastTuckerPlusTC	6.16	1.90	5.62	3.95	3.16	3.54	166.21	145.81

TABLE V
THE TEST RMSE AFTER 50 ITERATIONS FOR CUFASTTUCKERPLUSTC WITH VARIOUS NUMERICAL PRECISION ON THE REAL-WORLD DATASETS

Matrix A	Matrix B	Accumulator	Matrix Size	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
half	half	half	$16 \times 16 \times 16$	1.0132	0.9300	0.8246	1.1314	0.1879	0.3452
half	half	float	$16 \times 16 \times 16$	1.0133	0.9299	0.8243	1.1301	0.1879	0.3453
bf16	bf16	float	$16 \times 16 \times 16$	1.0132	0.9298	0.8248	1.1315	0.1879	0.3450
tf32	tf32	float	$16 \times 16 \times 8$	1.0129	0.9315	0.8346	1.1317	0.1879	0.3445
double	double	double	$8 \times 8 \times 4$	1.0128	0.9322	0.8329	1.1309	0.1878	0.3417

TABLE VI
THE SINGLE ITERATION TIME (IN SECONDS) FOR CUFASTTUCKERPLUSTC WITH VARIOUS NUMERICAL PRECISION ON THE REAL-WORLD DATASETS

Matrix A	Matrix B	Accumulator	Matrix Size	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
half	half	half	$16 \times 16 \times 16$	0.49	0.42	0.47	1.14	0.66	0.33
half	half	float	$16 \times 16 \times 16$	0.44	0.42	0.45	1.14	0.57	0.35
bf16	b16	float	$16 \times 16 \times 16$	0.52	0.44	0.48	1.18	0.67	0.34
tf32	tf32	float	$16 \times 16 \times 8$	0.68	0.58	0.62	1.53	0.88	0.43
double	double	double	$8 \times 8 \times 4$	7.22	6.59	6.89	16.64	9.25	4.91

same parameter settings and dataset, higher numerical precision results in longer running times for cuFastTuckerPlusTC. We set the parameters to $\{J_n = 16\}$ and $R = 64$, and run cuFastTuckerPlusTC using five different numerical precisions supported by Tensor Cores. Table VI reports the single iteration runtime of cuFastTuckerPlusTC on the Real-World datasets under these five precision settings. Table V shows the final test RMSE obtained by cuFastTuckerPlusTC after 50 iterations at each precision level. It is worth noting that the initial values are consistent across all runs; only the numerical precision varies. As shown in Table V, the accuracy of cuFastTuckerPlusTC across the five precision levels on the real-world datasets is comparable. Meanwhile, Table VI reveals that the single iteration times for the three precisions with matrix sizes of $16 \times 16 \times 16$ are similar. The runtime corresponding to the TF32 data type with a matrix size of $16 \times 16 \times 8$ increases slightly, whereas the runtime for double precision with a matrix size of $8 \times 8 \times 4$ increases significantly. The use of double precision results in more than a tenfold increase in computational time compared to single precision, primarily due to the lack of native FP64 support for Tensor Cores on consumer-grade GPUs. Therefore, cuFastTuckerPlusTC employs single precision with a matrix size of $16 \times 16 \times 16$ as the most appropriate setting.

We further evaluate the computation time and final test RMSE of cuFastTuckerPlus implemented with CUDA Cores using both single and double precision. Additionally, we assess the performance of cuTucker, cuFastTucker, and cuFasterTuckerCOO under the same precision settings. Table VII presents the final test RMSE of these algorithms on the real-world datasets at different precision levels, while Table VIII reports their single-iteration runtimes. As shown in Table VII, using double precision yields higher final accuracy on some datasets compared to single precision. However, this improvement comes at the cost of increased computation time, as reflected in Table VIII. For these algorithms, single precision is generally sufficient for numerical computations. Double precision may be adopted when higher numerical accuracy is required, albeit with a trade-off in performance.

TABLE VII
THE TEST RMSE AFTER 50 ITERATIONS FOR CUFASTTUCKERPLUS WITH VARIOUS NUMERICAL PRECISION ON THE REAL-WORLD DATASETS

Algorithm	Precision	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
cuTucker	float	1.0137	0.9473	0.8430	1.1773	0.1880	0.3466
	double	1.0137	0.9464	0.8421	1.1801	0.1880	0.3460
cuFastTucker	float	1.0137	0.9488	0.8432	1.1788	0.1880	0.3466
	double	1.0137	0.9455	0.8424	1.1788	0.1880	0.3460
cuFasterTuckerCOO	float	1.0136	0.9502	0.8437	1.1784	-	0.3467
	double	1.0138	0.9432	0.8421	1.1780	-	0.3448
cuFastTuckerPlus	float	1.0133	0.9301	0.8451	1.1669	0.1879	0.3471
	double	1.0164	0.9294	0.8164	1.1278	0.1888	0.3388

TABLE VIII
THE SINGLE ITERATION TIME (IN SECONDS) FOR CUFASTTUCKERPLUS WITH VARIOUS NUMERICAL PRECISION ON THE REAL-WORLD DATASETS

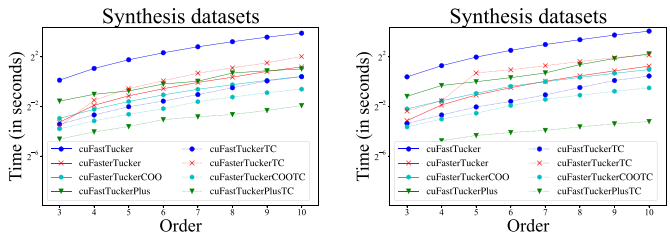
Algorithm	Precision	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
cuTucker	float	19.12	17.63	17.55	42.37	24.18	12.90
	double	32.91	27.99	29.63	71.01	43.25	21.85
cuFastTucker	float	6.96	6.23	6.50	15.77	8.88	4.77
	double	34.38	30.38	31.98	77.17	45.32	24.10
cuFasterTuckerCOO	float	1.22	0.92	1.03	2.40	-	0.66
	double	9.03	7.54	8.08	19.16	-	5.63
cuFastTuckerPlus	float	6.32	5.39	5.69	13.70	7.96	4.09
	double	26.58	22.81	23.99	57.91	33.54	17.01

TABLE IX
THE RUNNING TIME (IN SECONDS) OF A SINGLE ITERATION FOR CUFASTTUCKERPLUSTC AND OTHER ALGORITHMS ON THE REAL-WORLD DATASETS, AND THE SPEEDUP ACHIEVED BY CUFASTTUCKERPLUSTC

	Algorithm	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
The process of updating the factor matrices	cuFastTucker	0.91 (9.52×)	0.82 (17.52×)	0.85 (12.92×)	2.06 (13.80×)	1.12 (10.56×)	0.60 (20.83×)
	cuFasterTuckerCOO	0.17 (1.77×)	0.12 (2.52×)	0.14 (2.08×)	0.31 (2.11×)	0.23 (2.22×)	0.10 (3.33×)
	cuFasterTucker	0.20 (2.14×)	0.11 (2.42×)	0.16 (2.40×)	0.32 (2.12×)	0.28 (2.65×)	0.06 (2.06×)
	cuFastTuckerPlus	0.34 (3.57×)	0.29 (6.25×)	0.31 (4.79×)	0.96 (6.46×)	0.42 (3.95×)	0.22 (7.71×)
	cuFastTuckeTC	0.18 (1.86×)	0.11 (2.41×)	0.14 (2.18×)	0.29 (1.95×)	0.20 (1.86×)	0.07 (2.37×)
	cuFasterTuckerCOOTC	0.15 (1.58×)	0.08 (1.72×)	0.11 (1.73×)	0.26 (1.72×)	0.18 (1.73×)	0.05 (1.83×)
	cuFasterTuckerTC	0.24 (2.56×)	0.14 (3.03×)	0.19 (2.85×)	0.38 (2.57×)	0.34 (3.19×)	0.07 (2.37×)
	cuFastTuckerPlusTC	0.10	0.05	0.07	0.15	0.11	0.03
The process of updating the core matrices	cuFastTucker	1.06 (27.25×)	0.95 (27.20×)	0.99 (27.04×)	2.40 (27.31×)	1.37 (31.54×)	0.74 (31.33×)
	cuFasterTuckerCOO	0.20 (5.05×)	0.14 (4.09×)	0.16 (4.41×)	0.38 (4.30×)	0.26 (5.94×)	0.11 (4.57×)
	cuFasterTucker	0.18 (4.64×)	0.10 (2.95×)	0.14 (3.93×)	0.29 (3.33×)	0.27 (6.17×)	0.06 (2.67×)
	cuFastTuckerPlus	0.88 (22.81×)	0.76 (21.89×)	0.84 (23.10×)	2.14 (24.38×)	0.78 (17.95×)	0.41 (17.44×)
	cuFastTuckerTC	0.15 (3.86×)	0.11 (3.28×)	0.12 (3.40×)	0.28 (3.23×)	0.18 (4.10×)	0.08 (3.30×)
	cuFasterTuckerCOOTC	0.17 (4.30×)	0.11 (3.15×)	0.14 (3.79×)	0.31 (3.55×)	0.21 (4.94×)	0.06 (2.71×)
	cuFasterTuckerTC	0.44 (11.46×)	0.23 (6.63×)	0.34 (9.26×)	0.66 (7.46×)	0.49 (11.33×)	0.10 (4.34×)
	cuFastTuckerPlusTC	0.04	0.03	0.04	0.09	0.04	0.02

D. Single Iteration Running Time of the Algorithm

To evaluate the efficiency of the algorithms, we utilize the single iteration running time as a metric. We set the parameters as $\{J_n = 16\}$ and $R = 16$. Table IX details the single-iteration running times of cuFastTuckerPlusTC (non-convex optimization SGD algorithm) and other SOTA algorithms (convex optimization SGD algorithms) on the Real-World datasets. It is observed that cuFastTuckerPlus exhibits a longer running time compared to cuFasterTucker and cuFasterTuckerCOO. However, it still achieves a significant speedup of approximately 3× compared to cuFasterTucker. Additionally, cuFastTuckerPlusTC exhibits the lowest running time among all the algorithms. In terms of the synthetic datasets, as depicted in Fig. 2, cuFastTuckerPlus lags behind cuFasterTucker and cuFasterTuckerCOO in terms of running time. However, cuFastTuckerPlusTC showcases significantly lower running times compared to all other algorithms.



(a) The process of updating the factor matrices

(b) The process of updating the core matrices

Fig. 2. The running time (in seconds) of cuFastTuckerPlus and other algorithms on synthesis datasets.

E. Memory Access Overhead

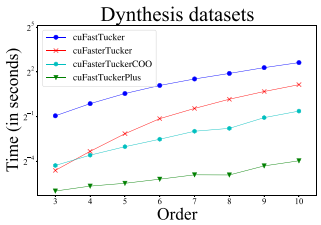
To evaluate the memory access overhead of the algorithm, we use the single iteration memory access time as the indicator.

TABLE X
THE MEMORY ACCESS TIME (IN SECONDS) FOR CUFASTTUCKERPLUS AND OTHER ALGORITHMS ON THE REAL-WORLD DATASETS, AND THE SPEEDUP ACHIEVED BY CUFASTTUCKERPLUS

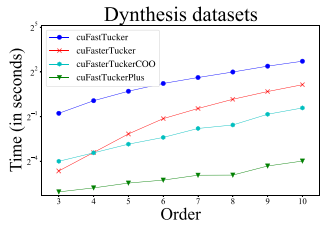
	Algorithm	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
The process of updating the factor matrices	cuFastTucker	0.72 (17.56×)	0.52 (23.49×)	0.65 (21.68×)	1.54 (22.06×)	0.70 (16.85×)	0.37 (24.09×)
	cuFasterTuckerCOO	0.13 (3.17×)	0.05 (2.29×)	0.09 (3.14×)	0.13 (1.87×)	0.17 (3.99×)	0.05 (3.09×)
	cuFasterTucker	0.10 (2.43×)	0.03 (1.29×)	0.06 (2.04×)	0.08 (1.20×)	0.13 (3.09×)	0.03 (2.16×)
	cuFastTuckerPlus	0.04	0.02	0.03	0.07	0.04	0.02
The process of updating the core matrices	cuFastTucker	0.70 (17.05×)	0.65 (31.10×)	0.61 (20.83×)	1.45 (20.99×)	0.71 (16.61×)	0.37 (23.31×)
	cuFasterTuckerCOO	0.17 (4.02×)	0.05 (2.40×)	0.12 (4.11×)	0.13 (1.89×)	0.20 (4.74×)	0.05 (3.34×)
	cuFasterTucker	0.14 (3.45×)	0.07 (3.20×)	0.10 (3.54×)	0.20 (2.87×)	0.20 (4.73×)	0.03 (1.92×)
	cuFastTuckerPlus	0.04	0.02	0.03	0.07	0.04	0.02

TABLE XI
THE SPEEDUP ACHIEVED BY CUFASTTUCKERPLUSTC AND OTHER ALGORITHMS WHEN UTILIZING TENSOR CORES ON THE REAL-WORLD DATASETS

	Algorithm	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
The process of updating the factor matrices	cuFastTucker	5.13×	7.27×	5.93×	7.08×	5.69×	8.80×
	cuFasterTuckerCOO	1.12×	1.47×	1.20×	1.22×	1.28×	1.82×
	cuFasterTucker	0.83×	0.80×	0.84×	0.83×	0.83×	0.87×
	cuFastTuckerPlus	3.57×	6.25×	4.79×	6.46×	2.63×	2.77×
The process of updating the core matrices	cuFastTucker	7.05×	8.29×	7.96×	8.45×	6.97×	10.83×
	cuFasterTuckerCOO	1.17×	1.30×	1.16×	1.21×	1.41×	2.04×
	cuFasterTucker	0.40×	0.45×	0.42×	0.45×	0.80×	0.92×
	cuFastTuckerPlus	22.81×	21.89×	23.10×	24.38×	2.12×	3.57×

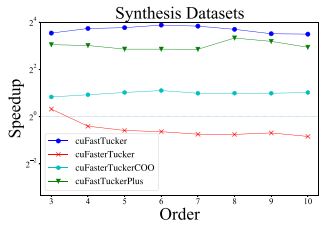


(a) The process of updating the factor matrices

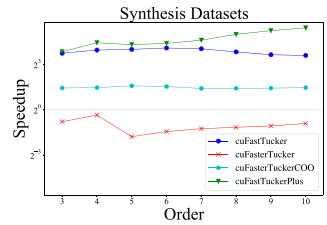


(b) The process of updating the core matrices

Fig. 3. The memory access time (in seconds) for cuFastTuckerPlus and other algorithms on synthesis datasets.



(a) The process of updating the factor matrices



(b) The process of updating the core matrices

Fig. 4. The speedup achieved by cuFastTuckerPlusTC and other algorithms when utilizing Tensor Cores on the synthesis datasets.

We set the parameters to $\{J_n = 16\}$ and $R = 16$. For the same algorithm, whether using Tensor Cores or CUDA Cores, the memory access mode is the same. Therefore, only cuFastTucker, cuFasterTucker, cuFasterTuckerCOO, and cuFastTuckerPlus are compared here. Table X presents the time required for cuFastTuckerPlus and other algorithms to read parameters from global memory on the Real-World datasets. Furthermore, Fig. 3 illustrates the time taken by cuFastTuckerPlus and other algorithms to read parameters from global memory on synthetic datasets. The observations from both the table and the figure demonstrate that cuFastTucker exhibits the longest memory access time. For sparse tensors of 3-order, cuFasterTucker requires less memory access time compared to cuFasterTuckerCOO, while for sparse tensors of 4-order and above, it takes longer. This discrepancy can be attributed to the higher sparsity of higher-order sparse tensors in the synthetic dataset. Notably, cuFastTuckerPlus showcases the shortest memory access time. Furthermore, as the order of the sparse tensor increases, the memory access time of cuFastTuckerPlus exhibits the slowest growth rate.

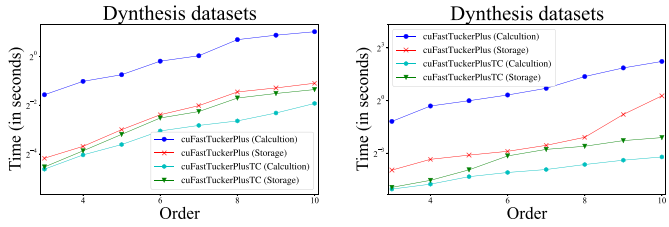
F. The Speedup of Algorithms by Tensor Cores

Table XI provides the speedup achieved by cuFastTuckerPlusTC and other algorithms after employing Tensor Cores on the Real-World datasets. Additionally, Fig. 4 illustrates the speedup attained by cuFastTuckerPlusTC and other algorithms with Tensor Cores on the synthetic datasets. The speedup is calculated as the ratio of the single iteration time required by the algorithm using CUDA Cores to the single iteration time required by the algorithm accelerated by Tensor Cores. Analysis of Table XI and Fig. 4 reveals significant acceleration for cuFastTuckerTC and cuFastTuckerPlusTC. Conversely, cuFasterTuckerTC experiences an increase in runtime, while cuFasterTuckerCOOTC achieves a slight speedup. Notably, when updating the factor matrices, cuFastTuckerTC demonstrates higher speedup compared to cuFastTuckerPlusTC. This discrepancy arises due to the higher computational complexity of cuFastTucker, with a larger portion of matrix calculations eligible for acceleration by Tensor Cores. On the other hand, when updating the core matrices, cuFastTuckerPlus outperforms

TABLE XII
THE RUNNING TIME (IN SECONDS) OF CUFASTTUCKERPLUSTC AND CUFASTTUCKERPLUS WITH VARIOUS STRATEGIES ON THE REAL-WORLD DATASETS

	Algorithm	Amazon	Netflix	Goodreads	Yahoo!	Nell-1	Nell-2
The process of updating the factor matrices	cuFastTuckerPlus (Calcultion)	0.34	0.29	0.31	0.96	0.42	0.22
	cuFastTuckerPlus (Storage)	0.15	0.08	0.11	0.27	0.18	0.04
	cuFastTuckerPlusTC (Calcultion)	0.09	0.05	0.06	0.15	0.11	0.03
	cuFastTuckerPlusTC (Storage)	0.11	0.06	0.09	0.20	0.14	0.03
The process of updating the core matrices	cuFastTuckerPlus (Calcultion)	0.88	0.76	0.84	2.14	0.78	0.41
	cuFastTuckerPlus (Storage)	0.11	0.08	0.10	0.20	0.14	0.05
	cuFastTuckerPlusTC (Calcultion)	0.04	0.03	0.04	0.09	0.04	0.02
	cuFastTuckerPlusTC (Storage)	0.07	0.04	0.05	0.12	0.08	0.02

cuFastTuckerTC in terms of speedup. This distinction is attributed to cuFastTuckerPlus storing more parameters than cuFastTucker, leading to performance degradation in cuFastTuckerPlus and thereby making the acceleration more pronounced. Both cuFasterTucker and cuFasterTuckerCOO primarily rely on reading $\{\boldsymbol{c}^{(n)}_{i_n,:}\}$ from global memory rather than utilizing Tensor Cores for calculations. Consequently, the matrix calculations suitable for Tensor Cores acceleration are minimal. Moreover, cuFasterTuckerTC experiences imbalanced performance due to the use of Tensor Cores, resulting in slower performance compared to cuFasterTuckerCOOTC.



(a) The process of updating the factor matrices

(b) The process of updating the core matrices

Fig. 5. The running time (in seconds) of cuFastTuckerPlus and cuFastTuckerPlusTC in various strategies on the synthesis datasets.

G. Replace Memory Access With Calculation

When updating the factor matrices in cuFastTuckerPlus and cuFastTuckerPlusTC, the factor matrices $\{\boldsymbol{A}^{(n)}\}$ are updated dynamically, which means that $\{\boldsymbol{c}^{(n)}_{\Psi^{(n)},:}\}$ can only be computed in real-time during the update process. However, when updating the core matrices, since $\{\boldsymbol{B}^{(n)}\}$ is updated at the end after accumulating gradients, it is possible to pre-compute $\{\boldsymbol{c}^{(n)}_{\Psi^{(n)},:}\}$ and store it in memory. Then, during the gradient accumulation process, $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$ can be directly read from memory, avoiding redundant computations. To explore this further, we consider two schemes in cuFastTuckerPlus and cuFastTuckerPlusTC: one scheme involves pre-computing $\{\boldsymbol{C}^{(n)}\}$ and then reading $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$, while the other scheme involves real-time computation of $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$. We denote these schemes as $cuFastTuckerPlus\ (Storage)$ and $cuFastTuckerPlus\ (Calculation)$ for cuFastTuckerPlus, and $cuFastTuckerPlusTC\ (Storage)$ and $cuFastTuckerPlusTC\ (Calculation)$ for cuFastTuckerPlusTC. We set the parameters as $\{J_n = 16\}$ and $R = 16$. Table XII presents the running time of the aforementioned schemes for cuFastTuckerPlus and cuFastTuckerPlusTC on the Real-World datasets., while Fig. 5 illustrates their running time on synthesis datasets. The results indicate that, in the absence of Tensor Cores acceleration, the scheme involving pre-computing $\{\boldsymbol{C}^{(n)}\}$ and then reading $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$ demonstrates better efficiency. However, when Tensor Cores acceleration is utilized, the real-time calculation of $\{\boldsymbol{C}^{(n)}_{\Psi^{(n)},:}\}$ becomes more efficient. cuFasterTucker reduces computational overhead by introducing additional memory access. Howover, Tensor Cores expedite these calculations, making them faster compared to reading from memory. This observation highlights that cuFastTuckerPlusTC outperforms cuFasterTucker and cuFasterTuckerTC across various aspects, resulting in overall superior performance.

VI. CONCLUSION

We propose the FastTuckerPlus decomposition algorithm, which tackles the entire optimization problem by dividing it into two non-convex optimization subproblems. This approach enables us to leverage the advantages of local search algorithms based on SGD, which exhibit faster convergence compared to global search algorithms used in convex optimization. Additionally, we introduce the cuFastTuckerPlusTC algorithm, which leverages fine-grained parallelism on GPUs and fully utilizes the capabilities of Tensor Cores. Our experimental results demonstrate that cuFastTuckerPlusTC achieves significantly faster convergence compared to existing SOTA algorithms. Specifically, its single iteration time is $3\times$ to $5\times$ faster than the SOTA algorithm, while simultaneously exhibiting smaller memory access overhead. Our algorithm excels at handling high-order and large-scale sparse tensors, exhibiting lower memory access and computational overhead compared to existing algorithms. Moreover, it outperforms SOTA algorithms in terms of convergence speed and load balancing.

REFERENCES

[1] M. Yin, Y. Liu, X. Zhou, and G. Sun, "A tensor decomposition based collaborative filtering algorithm for time-aware POI recommendation in LBSN," *Multimedia Tools Appl.*, vol. 80, no. 30, pp. 36215–36235, 2021.
[2] J. Wang, W. Jiang, K. Li, and K. Li, "Reducing cumulative errors of incremental CP decomposition in dynamic online social networks," *ACM Trans. Knowl. Discov. Data*, vol. 15, no. 3, pp. 1–33, 2021.
[3] P. Wang, L. T. Yang, G. Qian, J. Li, and Z. Yan, "HO-OTSVD: A novel tensor decomposition and its incremental decomposition for cyber–physical–social networks (CPSN)," *IEEE Trans. Netw. Sci. Eng.*, vol. 7, no. 2, pp. 713–725, Second Quarter 2020.

[4] Z. Chen, Z. Xu, and D. Wang, "Deep transfer tensor decomposition with orthogonal constraint for recommender systems," in *Proc. AAAI Conf. Artif. Intell.*, 2021, pp. 4010–4018.
[5] Y. Shin and S. S. Woo, "PasswordTensor: Analyzing and explaining password strength using tensor decomposition," *Comput. Secur.*, vol. 116, 2022, Art. no. 102634.
[6] F. Huang, X. Yue, Z. Xiong, Z. Yu, S. Liu, and W. Zhang, "Tensor decomposition with relational constraints for predicting multiple types of microRNA-disease associations," *Brief. Bioinf.*, vol. 22, no. 3, 2021, Art. no. bbaa140.
[7] Y. Zhu, X. Li, T. Ristaniemi, and F. Cong, "Measuring the task induced oscillatory brain activity using tensor decomposition," in *Proc. 2019 IEEE Int. Conf. Acoust. Speech Signal Process.*, 2019, pp. 8593–8597.
[8] S. Mirzaei and H. Soltanian-Zadeh, "Overlapping brain community detection using Bayesian tensor decomposition," *J. Neurosci. Methods*, vol. 318, pp. 47–55, 2019.
[9] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Rev.*, vol. 51, no. 3, pp. 455–500, 2009.
[10] L. De Lathauwer, B. De Moor, and J. Vandewalle, "A multilinear singular value decomposition," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1253–1278, 2000.
[11] L. De Lathauwer, B. De Moor, and J. Vandewalle, "On the best rank-1 and rank-(r 1, r 2,..., rn) approximation of higher-order tensors," *SIAM J. Matrix Anal. Appl.*, vol. 21, no. 4, pp. 1324–1342, 2000.
[12] P. Comon, X. Luciani, and A. L. De Almeida, "Tensor decompositions, alternating least squares and other tales," *J. Chemometrics: J. Chemometrics Soc.*, vol. 23, no. 7/8, pp. 393–405, 2009.
[13] Y. Xu and W. Yin, "A block coordinate descent method for regularized multiconvex optimization with applications to nonnegative tensor factorization and completion," *SIAM J. Imag. Sci.*, vol. 6, no. 3, pp. 1758–1789, 2013.
[14] R. Ge, F. Huang, C. Jin, and Y. Yuan, "Escaping from saddle points—online stochastic gradient for tensor decomposition," in *Proc. Int. Conf. Learn. Theory*, 2015, pp. 797–842.
[15] I. Nisa, J. Li, A. Sukumaran-Rajam, R. Vuduc, and P. Sadayappan, "Load-balanced sparse MTTKRP on GPUs," in *Proc. 2019 IEEE Int. Parallel Distrib. Process. Symp.*, 2019, pp. 123–133.
[16] S. Oh, N. Park, S. Lee, and U. Kang, "Scalable tucker factorization for sparse tensors-algorithms and discoveries," in *Proc. IEEE 34th Int. Conf. Data Eng.*, 2018, pp. 1120–1131.
[17] M. Park, J.-G. Jang, and L. Sael, "VEST: Very sparse tucker factorization of large-scale tensors," in *Proc. 2021 IEEE Int. Conf. Big Data Smart Comput.*, 2021, pp. 172–179.
[18] H. Li, Z. Li, K. Li, J. S. Rellermeyer, L. Chen, and K. Li, "SGD_Tucker: A novel stochastic optimization strategy for parallel sparse tucker decomposition," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1828–1841, Jul. 2021.
[19] S. Oh, N. Park, J.-G. Jang, L. Sael, and U. Kang, "High-performance tucker factorization on heterogeneous platforms," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 10, pp. 2237–2248, Oct. 2019.
[20] N. Park, B. Jeon, J. Lee, and U. Kang, "BIGtensor: Mining billion-scale tensor made easy," in *Proc. 25th ACM Int. Conf. Inf. Knowl. Manage.*, 2016, pp. 2457–2460.
[21] Z. Li, Y. Hu, M. Li, W. Yang, and K. Li, "cuFastTucker: A novel sparse FastTucker decomposition for HHLST on multi-GPUs," *ACM Trans. Parallel Comput.*, vol. 11, no. 2, Jun. 2024, Art. no. 12. [Online]. Available: https://doi.org/10.1145/3661450
[22] Z. Li, Y. Qin, Q. Xiao, W. Yang, and K. Li, "cuFasterTucker: A stochastic optimization strategy for parallel sparse FastTucker decomposition on GPU platform," *ACM Trans. Parallel Comput.*, vol. 11, no. 2, Jun. 2024, Art. no. 8. [Online]. Available: https://doi.org/10.1145/3648094
[23] T. Ma, "Why do local methods solve nonconvex problems?," in *Beyond the Worst-Case Analysis of Algorithms*. Cambridge, U.K.: Cambridge Univ. Press, 2021.
[24] Y. N. Dauphin, R. Pascanu, C. Gulcehre, K. Cho, S. Ganguli, and Y. Bengio, "Identifying and attacking the saddle point problem in high-dimensional non-convex optimization," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 2933–2941.
[25] A. Choromanska, M. Henaff, M. Mathieu, G. B. Arous, and Y. LeCun, "The loss surfaces of multilayer networks," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2015, pp. 192–204.
[26] D. Park, A. Kyrillidis, C. Carmanis, and S. Sanghavi, "Non-square matrix sensing without spurious local minima via the Burer-Monteiro approach," in *Proc. Int. Conf. Artif. Intell. Statist.*, 2017, pp. 65–74.
[27] S. Bhojanapalli, B. Neyshabur, and N. Srebro, "Global optimality of local search for low rank matrix recovery," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 3880–3888.
[28] R. Ge, J. D. Lee, and T. Ma, "Matrix completion has no spurious local minimum," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2016, pp. 2981–2989.
[29] R. Ge, C. Jin, and Y. Zheng, "No spurious local minima in nonconvex low rank problems: A unified geometric analysis," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1233–1242.
[30] A. Frandsen and R. Ge, "Optimization landscape of tucker decomposition," *Math. Program.*, vol. 193, no. 2, pp. 687–712, 2022.
[31] Y. Carmon, J. C. Duchi, O. Hinder, and A. Sidford, "Accelerated methods for nonconvex optimization," *SIAM J. Optim.*, vol. 28, no. 2, pp. 1751–1772, 2018.
[32] N. Agarwal, Z. Allen-Zhu, B. Bullins, E. Hazan, and T. Ma, "Finding approximate local minima for nonconvex optimization in linear time," 2016, *arXiv:1611.01146*.
[33] C. Jin, R. Ge, P. Netrapalli, S. M. Kakade, and M. I. Jordan, "How to escape saddle points efficiently," in *Proc. Int. Conf. Mach. Learn.*, 2017, pp. 1724–1732.
[34] B. Feng, Y. Wang, T. Geng, A. Li, and Y. Ding, "APNN-TC: Accelerating arbitrary precision neural networks on Ampere GPU tensor cores," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2021, pp. 1–13.
[35] J. Finkelstein et al., "Quantum-based molecular dynamics simulations using tensor cores," *J. Chem. Theory Computation*, vol. 17, no. 10, pp. 6180–6192, 2021.
[36] H. Huang, X.-Y. Liu, W. Tong, T. Zhang, A. Walid, and X. Wang, "High performance hierarchical tucker tensor learning using GPU tensor cores," *IEEE Trans. Comput.*, vol. 72, no. 2, pp. 452–465, Feb. 2023.
[37] A. Li and S. Su, "Accelerating binarized neural networks via bit-tensor-cores in turing GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 7, pp. 1878–1891, Jul. 2021.
[38] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse tensor core: Algorithm and hardware co-design for vector-wise sparse neural networks on modern GPUs," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, pp. 359–371.
[39] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix–matrix multiplication with GPU tensor cores," *Comput. Elect. Eng.*, vol. 88, 2020, Art. no. 106848.
[40] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, "DGEMM using tensor cores, and its accurate and reproducible versions," in *Proc. 35th Int. Conf. High Perform. Comput.*, Springer, 2020, pp. 230–248.
[41] J. S. Firoz, A. Li, J. Li, and K. Barker, "On the feasibility of using reduced-precision tensor core operations for graph analytics," in *Proc. 2020 IEEE High Perform. Extreme Comput. Conf.*, 2020, pp. 1–7.
[42] J. Ni, J. Li, and J. McAuley, "Justifying recommendations using distantly-labeled reviews and fine-grained aspects," in *Proc. 2019 Conf. Empirical Methods Natural Lang. Process. 9th Int. Joint Conf. Natural Lang. Process.*, 2019, pp. 188–197.
[43] M. Wan and J. J. McAuley, "Item recommendation on monotonic behavior chains," in *Proc. 12th ACM Conf. Recommender Syst.*, 2018, pp. 86–94. [Online]. Available: https://doi.org/10.1145/3240323.3240369
[44] M. Wan, R. Misra, N. Nakashole, and J. J. McAuley, "Fine-grained spoiler detection from large-scale review corpora," in *Proc. 57th Conf. Assoc. Comput. Linguistics*, 2019, pp. 2605–2610. [Online]. Available: https://doi.org/10.18653/v1/p19-1248
[45] 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 *Proc. AAAI Conf. Artif. Intell.*, 2010, Art. no. 3.
[46] J. McAuley and J. Leskovec, "Hidden factors and hidden topics: Understanding rating dimensions with review text," in *Proc. 7th ACM Conf. Recommender Syst.*, 2013, pp. 165–172.
[47] J. Baumgartner, "Reddit comment dataset," 2015. [Online]. Available: https://www.reddit.com/r/datasets/comments/3bxlg7/i_have_every_publicly_available_reddit_comment/

Zixuan Li received the BS degree from the School of Mathematics, Hunan University, China, in 2017, and the MS and PhD degrees from the College of Computer Science and Electronic Engineering, Hunan University, China, in 2020 and 2024, respectively. He is currently a lecturer with Xiangtan University, China. His research interests mainly include high performance computing, parallel and distributed systems, tensor decomposition, and tensor completion.

Mingxing Duan received the PhD degree from the School of Computer Science, National University of Defense Technology, China. He is now an associate professor with the School of Information Science and Engineering, Hunan University. He has been a postdoctoral fellow with the Department of Computer Engineering, College of Computer Science and Electronic Engineering, Hunan University, and also a postdoctoral fellow with the Department of Computing, Hong Kong Polytechnic University. He has been supported by the Chinese Hong Kong Scholars Program. His research interests include Big Data and machine learning. He has served on the Editorial Board of the *American Journal of the Neural Networks and Applications*.

Huizhang Luo received the BS and PhD degrees from the Department of Computer Science, Chongqing University, China, in 2012 and 2017, respectively. He is now an associate professor with Hunan University, China. Before joining Hunan University, he was a postdoctoral research fellow with the HPC Lab, New Jersey Institute of Technology. His research interests mainly include high-performance computing, parallel and distributed systems, and memory systems. His work on scientific data reduction had received the Best Paper Award Nomination at IPDPS 2018.

Wangdong Yang received the PhD degree in computer science from Hunan University, China. He is a professor of computer science and technology with Hunan University, China. His research interests include modeling and programming for heterogeneous computing systems, parallel and distributed computing, and numerical computation. He has published more than 60 papers international conferences and journals.

Kenli Li (Senior Member, IEEE) received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He is currently a full professor of computer science and technology with Hunan University and the deputy director of the National Supercomputing Center, Changsha. He has published more than 300 papers in international conferences and journals. His research interests include parallel computing, cloud computing, and Big Data computing. He is a fellow of CCF. He serves on the editorial boards of the *IEEE Transactions on Computers*, *IEEE Transactions on Industrial Informatics*, *IEEE Transactions on Services Computing*, and *International Journal of Pattern Recognition and Artificial Intelligence*.

Keqin Li (Fellow, IEEE) received the BS degree in computer science from Tsinghua University, in 1985, and the PhD degree in computer science from the University of Houston, in 1990. He is a SUNY distinguished professor with the State University of New York and a National distinguished professor with Hunan University, China. He has authored or co-authored more than 1200 journal articles, book chapters, and refereed conference papers. He holds nearly 80 patents announced or authorized by the Chinese National Intellectual Property Administration. He is among the world's top few most influential scientists in parallel and distributed computing, regarding single-year impact (ranked #2) and career-long impact (ranked #3) based on a composite indicator of the Scopus citation database. He is listed in Scilit Top Cited Scholars (2023-2025) and is among the top 0.02% out of more than 20 million scholars worldwide based on top-cited publications in the last ten years. He is listed in ScholarGPS Highly Ranked Scholars (2022-2024) and is among the top 0.002% out of more than 30 million scholars worldwide based on a composite score of three ranking metrics for research productivity, impact, and quality in the recent five years. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He was a recipient of the 2022–2023 International Science and Technology Cooperation Award and the 2023 Xiaoxiang Friendship Award of Hunan Province, China. He is a member of the SUNY Distinguished Academy. He is an AAAS fellow, an AAIA fellow, an ACIS fellow, and an AIIA fellow. He is a member of the European Academy of Sciences and Arts. He is a member of Academia Europaea (Academician of the Academy of Europe).