1 INTRODUCTION

Tensors are originated from differential manifold and tensor is used to analyze the change of high dimension space [1]. Due to the amazing representation ability, tensor can capture the relationship between multi-attribute of an entity [2]. Especially, in Machine Learning (ML) community, which relies on effective statistical learning methodology and plenty of data, needs powerful data structure to guarantee the abundant information [3]. Meanwhile, due to abundant data styles in ML, tensor has drawn wide attention to the emerging ML research communities [4].

In ML communities, tensor applications can be divided into the following three classes: (1) In order to capture the general feature of multi-modal data, multi-view learning always combines multi-feature into a tensor space [5], [6], [7], [8]. (2) To project the multi-attribute data into a low complexity and low-rank space, the learning weight variable always be constituted tensor data, etc, Tensor Regression [9], [10], [11]. Support Tensor Machine [12], [13], [14], and Deep Convolutional Neural Networks (DCNN) in TensorFlow and Pytorch framework [15]. (3) Due to the spatiotemporal dynamics and multi-attribute interaction, the forming data is naturally tensor, e.g., in Recommendation Systems [16], Quality of Service (QoS) [17], Network Flow [18], Cyber-Physical-Social (CPS) [19], or Social Networks [20]. The scale of tensor data from the fusion process after the multi-modal feature and weight variables of ML methodologies is far below than the natural tensor data.

High-Order, High-Dimension, and Sparse Tensor (HOHDST) is a mathematic model for the data from Recommendation Systems, QoS, Network Flow, CPS, and Social Networks and high-order and high-dimension mean multi-attribute interaction and multi-entity, respectively [21], [22]. An N-order HOHDST can represent the interaction relationship between N attributes and in reality, each attribute has millions of entities. Thus, this property will result in a substantially high-dimension inherence [4]. Unfortunately, due to data incompleteness, it is non-trivial to obtain the statistic property of the HOHDST data.

The common used method is finding the low-dimension feature via Sparse Tensor Decomposition (STD) and this dimensionality reduction techniques can represent the original HOHST by low-rank or low-dimension space [21], [22], [23], [24]. Tensor Tucker decomposition is one of the most widely used dimensionality reduction methodologies. Through the N-coordinate systems and those systems tangled by a core tensor between each other, tensor Tucker decomposition becomes one of the most used dimensionality reduction methodologies [25]. There are two approaches to find the appropriate core tensor and the N factor matrices: (1) High Order Orthogonal Iterations (HOOI) should find the N orthogonal coordinate systems and this method needs the Singular Value Decomposition (SVD) for the unfolding tensor. However, this method relies on frequent Khatri-Rao and Kronecker products for intermediate matrices [25], [27], [28], [29], [30]. (2) Modern optimization strategy disentangles the tanglement of the core tensor and the N factor matrices and than transfers the non-convex optimization into alternative convex optimization. The above two methods still involve

Index Terms—GPU CUDA Parallelization; Kruskal Approximation; Sparse Tensor Decomposition; Stochastic Strategy; Tensor Computation.

Zixuan Li

Abstract—High-Order, High-Dimension, and Sparse Tensor (HOHDST) data originates from real industrial applications, i.e., social networks, recommender systems, bio-information, and traffic information. Sparse Tensor Decomposition (STD) can project the HOHDST data into low-rank space. In this work, a novel method for STD of Kruskal approximating the core tensor and stochastic strategy for approximating the whole gradient is proposed which comprises of the following two parts: (1) the matrization unfolding order of the Kruskal product for the core tensor follows the multiplication order of the factor matrix and then the proposed theorem can reduce the exponential computational overhead into linear one; (2) stochastic strategy adopts one-step random sampling set, the volume of which is much smaller than original one, to approximate the whole gradient. Meanwhile, this method can guarantee the convergence and save the memory overhead. Due to the compactness of the same order matrix multiplication and parallel access from stochastic strategy, the speed of cuFastTucker can be further reinforced by GPU. Furthermore, a data division and communication strategy of cuFastTucker is proposed for data accommodation on Multi-GPU. cuFastTucker can achieve the fastest speed and keep the same accuracy and much lower memory overhead than the SOTA algorithms, e.g., P-Tucker, Vest, and SGD_Tucker. The code and partial datasets are publically available on "https://github.com/ZixuanLi-China/FastTucker".
high-dimension intermediate matrices, and in order to solve these problems, the main contributions of this work are listed as the following:

1) The space overhead of the intermediate coefficient matrices for updating the core tensor is super huge. A Kruskal approximation strategy is proposed to divide the core tensor into smaller ones. Then, the order of matrix multiplication follows the same matrix multiplication order of the factor matrix. Following the proposed computational Theorems [1] and [2], the computational overhead can be reduced from the exponential overhead into an linear one;

2) By the one step sampling set, on each training iteration, a stochastic strategy is proposed to approximate the whole gradient relying on the whole HOHDST data by partial set. This methodology further reduce the computational overhead and keep the same accuracy;

3) The fine-grain parallelization inherence gives the allocated CUDA thread block the high parallelization and meanwhile, two key steps which are the most time-consuming can be further accelerated by CUDA parallelization (cuFastTucker). Because large-scale HOHDST data cannot be accommodated in a single GPU, a data division and communication strategy of cuFastTucker is proposed for data accommodation on Multi-GPU.

To our best knowledge, the proposed model is the first work that it can take advantage of the Kruakal product to approximate the core tensor with linear computational overhead. In this work, the related work is presented in Section 2. The notations and preliminaries are introduced in Section 3. The proposed model as well as cuFastTucker are showed in Sections 4 and 5 respectively. Experimental results are illustrated in Section 6.

2 RELATED WORKS

ML communities should handle the high-order data and tensor can capture the three or higher-order feature rather than unfolding the high-order data into matrix or vector. When the learning data has high-order feature, i.e., Human Recognition Data, Spatiotemporal Dynamics Data, Tensor Regression [9], [10], [11] can project multi-attribute weather data into the forecasting value, and Support Tensor Machine [12], [13], [14] can find the discrete classification value from multi-attribute data. DCNN plays a key role to learn deep feature from plenty of data, and the tensor decomposition can reduce the parameter complexity [31], [32]. Direct training process for the high-order and high-dimension tensor weight variable will result in over-fitting problem. To avoid the dimension explosion problem, tensor decomposition can reduce the parameter space overhead and the learning process only involves the factor matrices. There are a mass of works try to reduce the parameter complexity. However, those methods cannot solve the dimensionality reduction problem in HOHDST data. In big data era, it is non-trivial to process the HOHDST data. The main problems lie in efficient learning algorithm and the high match process in modern big-data process frameworks, i.e., OpenMP, MPI, CUDA, Hadoop, Spark, and OpenCL, and modern hardware, i.e., GPU, CPU, and embedded platforms.

A distributed CANDECOMP/PARAFAC Decomposition (CPD) [33] is proposed by Ge, et al., and the CPD is a special Tucker Decomposition for HOHDST. In HOHDST data compression community, Shaden, et al., [34] presented a Compressed Sparse Tensors (CST) structure which can improve the access speed and make data compression for HOHDST. Ma et al., [35] optimized the Tensor-Time-Matrix-Chain (TTMc) operation on GPU which is a key part for Tucker Decomposition (TD) and TTMc is a data intensive task. A distributed Non-negative Tucker Decomposition (NTD) is proposed by Chakaravarthy et al., [36] which needs frequent TTMc operations. A parallel strategy of ALS and CD for STD [37], [38] is presented on OpenMP parallelization. A heterogeneous OpenCL parallel version of ALS for STD is proposed on [39]. The current parallel and distributed works [40] mainly focus on divide the whole data into smaller and low-dependence parts and then deploy algorithm rather than fine-grained learning methodologies.

3 NOTATIONS AND PRELIMINARIES

We denote scalars by regular lowercase or uppercase, vectors by bold lowercase, matrices by bold uppercase, and tensor by bold lowercase, matrices by bold uppercase, and tensors by bold Euler script letters. Basic symbols and matrix and tensor operations are presented in Tables 2 and 1 respectively.

| Operations | Definition |
|------------|------------|
| The nth matricization of tensor X: | X = \sum_{i=1}^{N} x^{(n)}_{i} |
| The nth column vectorization of tensor X: | \sum_{k=1}^{N} x_{k}^{(n)} |
| R Kruskal Product: | B^{(n)} = \sum_{r=1}^{R} b^{(n)}_{r} |
| n-Mode Tensor-Matrix product: | (X \times (n)) = \sum_{i=1}^{n} x_{i}^{(n)} |

3.1 Basic Definitions

Definition 1 (The n-Rank of a Tensor). The n-Rank of tensor X ∈ R^{I_{1}×⋯×I_{N}} is the rank of n-th matricization X^{(n)}, denoted as rank_{n}(X).

Definition 2 (Tensor Approximation). For a N-order sparse tensor X ∈ R^{I_{1}×⋯×I_{N}}, the Tensor Approximation should find a low-rank tensor \tilde{X} ∈ R^{I_{1}×⋯×I_{N}} such that the noisy tensor E ∈ R^{I_{1}×⋯×I_{N}} should be small enough, where E = X - \tilde{X}.

Definition 3 (Sparse Tucker Decomposition (STD)). Given a N-order HOHDST X ∈ R^{I_{1}×⋯×I_{N}}, the goal of STD is to
train a core tensor $\mathcal{G} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ and $N$ factor matrices $A^{(n)} \in \mathbb{R}^{J_n \times K_{n-1}} \times \cdots \times \mathbb{R}^{J_N \times K_0}$, such that:

$$\mathbf{X} \approx \hat{\mathbf{X}} = \mathcal{G} \times_1 A^{(1)} \times_2 \cdots \times_N A^{(n)} \times_{N+1} \cdots \times_N A^{(N)},$$  

\[ (1) \]

The matricized versions of equation \[ (1) \] are

$$\hat{\mathbf{X}}^{(n)} = A^{(n)} \mathcal{G}^{(n)} (A^{(N)} \otimes \cdots \otimes (A^{(n+1)} \otimes A^{(n-1)} \otimes \cdots \otimes A^{(1)}))^T$$  

\[ (2) \]

where $\hat{\mathbf{X}}^{(n)}$ is nth matricization of tensor $\mathbf{X}$, $\mathcal{G}^{(n)}$ is nth matricization of tensor $\mathcal{G}$ and

$$\hat{\mathbf{X}}^{(n)} = (A^{(N)} \otimes \cdots \otimes (A^{(n+1)} \otimes (A^{(n-1)} \otimes \cdots \otimes A^{(1)}))^{g^{(n)}},$$  

\[ (3) \]

where $\hat{\mathbf{X}}^{(n)}$ is nth matricization of tensor $\mathbf{X}$, $g^{(n)}$ is nth matricization of tensor $\mathcal{G}$.

The basis optimization problem is organized as \[ (1) \] and \[ (2) \] as:

$$\arg\min_{w \in \mathbb{R}^{N}} f(w) = \frac{L}{\mathbf{y}, \mathbf{x}, w} + \lambda w \mathbf{R}(w)$$  

\[ (4) \]

and can obtain appropriate accuracy as:

$$\arg\min_{A^{(n)} \in \mathbb{R}^{N}} f \left( A^{(n)} \right) = \mathbf{X}^{(n)}$$  

\[ (7) \]
A^{(1)} \in \mathbb{R}^{J_N \times I_N \times I_{n-1} \times \cdots \times I_{n-1} \times J_1 \times J_1 \times \cdots \times J_1} \quad \text{and} \quad G^{(n)} \in \mathbb{R}^{J_n \times I_n \times I_{n-1} \times \cdots \times I_{n-1} \times J_1 \times J_1 \times \cdots \times J_1}.

\text{arg min}_{g^{(n)}} \left( g^{(n)} \mid x^{(n)}, \{A^{(n)}\}, g^{(n)} \right)

= \left\| x^{(n)} - \hat{A}^{(n)} \right\|^2_2 + \lambda g^{(n)} \left\| g^{(n)} \right\|^2_2

\tag{8}

where \( \hat{A}^{(n)} = H^{(n)} g^{(n)} \) and the coefficient \( H^{(n)} = A^{(N)} \otimes \cdots \otimes A^{(n+1)} \otimes A^{(n-1)} \otimes \cdots \otimes A^{(1)} \otimes A^{(n)} \in \mathbb{R}^{J_N \times I_N \times I_{n-1} \times \cdots \times I_{n-1} \times J_1 \times J_1 \times \cdots \times J_1} \). The coefficient matrices \( \{ \{G^{(n)}, S^{(n)}\}, \{H^{(n)}\}\} \) of variables \{A^{(n)}, g^{(n)}\}, \( n \in \{N\} \), respectively, are memory-consuming. For the problems (7) and (8), computing on GPU should consider high parallelization, control-orient (CPU). To make better use of hardware resource, the proposed model has the matricized version of \( A \) and the coefficient matrix of a optimization strategy obeying Theorems 4.1 and 4.2 present a novel strategy to make promise of the convergence.

Besides, it is hard for modern hardware to give consideration to both the computation-oriented (GPU) and logic-control-orient (CPU). To make better use of hardware resource, the algorithm design for high performance computing on GPU should consider high parallelization, low memory overhead, and low conflict probability for the operation of memory Read-And-Write. Thus, appropriate sampling for coefficient matrix of an optimization strategy should be considered to maintain low overhead and, meanwhile, comparable accuracy. In the following section, a compact stochastic strategy for STD will be introduced.

4 A Compact Stochastic Strategy for STD

The coefficient matrices \( \{ \{G^{(n)}, S^{(n)}\}, \{H^{(n)}\}\} \) in the optimization problems (7) and (8), respectively, are memory-consuming. Statistic sampling for approximating a full gradient should consider (1) computational convenience, (2) convergence and (3) accuracy. For the problem 1, SGD for STD chooses the elements from randomly one-step sampling set \( \Psi \) which is a subset of index set \( \Omega \). Then, the gradient of the coefficient matrices \( \{ \{G^{(n)}, S^{(n)}\}, \{H^{(n)}\}\} \) just obeys the order from the set \( \Omega \) rather than the whole set \( \Omega \), and, by this way, the construction overhead from one-step partial set \( \Psi \) is much lower than the whole set \( \Omega \).

SGD methods of updating factor matrix and core tensor \( \{ \{A^{(n)}, n \in \{N\}\}, \{\hat{G}\}\} \) are still memory-consuming. Sections 4.3 and 4.4 present a novel strategy obeying Theorems 4.1 and 2 with SGD for the process of updating the factor matrix and core tensor, respectively, and the proposed strategy can further reduce the exponentially increased overhead into linear one. Hence, due to the compactness of matrix multiplication and parallel access, the proposed model has fine-grained parallelization. Section 4.3 will conclude the overall computational and space overheads.

The gradient of the optimization problem (6) should be considered to maintain low overhead and, meanwhile, comparable accuracy. In this paper, the problem (6) is turned into:

\[
\text{arg min}_{A^{(n)}, n \in \{N\}} \left( \{A^{(n)}, \hat{G}\} \right)

= \left\| X^{(n)} - X^{(n)} \right\|^2_2 + \lambda \hat{G} \left\| \hat{G} \right\|^2_2 + \lambda A^{(n)} \left\| A^{(n)} \right\|^2_2.

\tag{10}

4.1 Update Process for Factor Matrix

Correspondingly, the problem (7) is turned into:

\[
\text{arg min}_{A^{(n)}, n \in \{N\}} \left( A^{(n)}, X^{(n)}, \{A^{(n)}\}, \hat{G}^{(n)} \right)

= \left\| X^{(n)} - X^{(n)} \right\|^2_2 + \lambda a^{(n)} \left\| a^{(n)} \right\|^2_2.

\tag{11}

and each feature vector \( a^{(n)} \), \( n \in \{N\} \), shares the same coefficient matrix \( G^{(n)} S^{(n)}T \). With the one-step sampling set \( \Psi \), the optimization problem is turned into:

\[
\text{arg min}_{A^{(n)}, n \in \{N\}} \left( a^{(n)}, X^{(n)}, \{a^{(n)}\}, \hat{G}^{(n)} \right)

= \left\| X^{(n)} - X^{(n)} \right\|^2_2 + \lambda a^{(n)} \left\| a^{(n)} \right\|^2_2.

\tag{12}

where \( \hat{X}^{(n)} \) is a \( \{\Psi^{(n)}\} \) row vector \( \mathbb{R}^{I_{n+1} \times I_{n+1} \times \cdots \times I_{n+1} \times \cdots \times I_{n+1}} \) and the approximated gradient from SGD is obtained as:

\[
\partial f (a^{(n)}, X^{(n)}, \{a^{(n)}\}, \hat{G}^{(n)})

= - \frac{X^{(n)}}{\hat{X}^{(n)}} \frac{\partial D^{(n)}}{\partial a^{(n)}} (\Psi^{(n)} a^{(n)})_{\hat{X}^{(n)}} + \lambda a^{(n)} a^{(n)}

\tag{13}

where \( D^{(n)} (\Psi^{(n)} a^{(n)})_{\hat{X}^{(n)}} = \hat{G}^{(n)} S^{(n)} T \). Theorem 1.

There are two row vectors \( x = x^{(N)} \otimes x^{(N-1)} \otimes \cdots \otimes x^{(2)} \otimes x^{(1)} \) and \( y = y^{(N)} \otimes y^{(N-1)} \otimes \cdots \otimes y^{(2)} \otimes y^{(1)} \), \( n \in \{N\} \), where \( x^{(n)} \in \mathbb{R}^{I_{n} 	imes I_{n}} \) and \( y^{(n)} \in \mathbb{R}^{I_{n} \times I_{n}} \), \( n \in \{N\} \). The vector multiplication \( x^{T} y = (x^{(N)} \otimes x^{(N-1)} \otimes \cdots \otimes x^{(2)} \otimes x^{(1)}) (y^{(N)} \otimes y^{(N-1)} \otimes \cdots \otimes y^{(2)} \otimes y^{(1)}) \) can be transformed into \( x^{T} y = (x^{(N)} y^{(N)}) (x^{(N-1)} y^{(N-1)}) \cdots (x^{(2)} y^{(2)}) (x^{(1)} y^{(1)}) \).

\[
\cdots \otimes b^{(n+1)} \otimes b^{(n-1)} \otimes \cdots \otimes b^{(1)}

\]
Proof. The index $(i/j)$ corresponds to a solely index \((i_1, \cdots, i_n, i_N) / (i_1', \cdots, i_n', i_N')\), respectively, where \(i = 1 + \sum_{k=1}^{N} (i_k - 1) \prod_{m=1, m \neq k}^{N} I_m\) \(\div\) \(j = 1 + \sum_{k=1}^{N} (i_k - 1) \prod_{m=1, m \neq k}^{N} I_m\) and \((x_i = \prod_{k=1}^{N} x_i^{(n)} / y_j = \prod_{k=1}^{N} y_j^{(n)})\). \(x y^T = \sum_{i_1 \cdots i_n = 1}^I x_i y_j^T = \sum_{i_1 \cdots i_n = 1}^I \sum_{i_k = 1}^I \sum_{i_n = 1}^I \sum_{i_N = 1}^I (\prod_{k=1}^{N} x_i^{(n)} \prod_{k=1}^{N} x_j^{(n)}) = \prod_{k=1}^{N} (x^{(n)} (y^{(N)})^T). \cdots \prod_{2}^{(n)} (x^{(2)} (y^{(2)})^T) \cdot (x^{(1)} (y^{(1)})^T).

According to Theorem 1 \(\mathbf{G}S_{ij}^{(n)} = \hat{\mathbf{G}}(n)S_{ij}^{(n)T} = \mathbf{B}^{(n)}(\mathbf{B}^{(n)} \otimes \cdots \otimes \mathbf{B}^{(n+1)}), \mathbf{B}^{(n)} \otimes \cdots \otimes \mathbf{B}^{(1)}(\mathbf{a}^{(1)}_{i_1}, \cdots, \mathbf{a}^{(1)}_{i_N}) = \sum_{r_{\text{core}}=1}^{R_{\text{core}}} b^{(n)}_{r_{\text{core}}} (b^{(n)}_{r_{\text{core}}} \otimes \cdots \otimes b^{(1)}_{r_{\text{core}}}) (\mathbf{a}^{(1)}_{i_1}, \cdots, \mathbf{a}^{(1)}_{i_N}) \sum_{k=1}^{N} b^{(n)}_{r_{\text{core}}} (b^{(n)}_{r_{\text{core}}} \otimes \cdots \otimes b^{(1)}_{r_{\text{core}}}) (\mathbf{a}^{(1)}_{i_1}, \cdots, \mathbf{a}^{(1)}_{i_N}), \text{where} \ j = 1 + \sum_{k=1}^{N} b^{(n)}_{r_{\text{core}}} (b^{(n)}_{r_{\text{core}}} \otimes \cdots \otimes b^{(1)}_{r_{\text{core}}}) (\mathbf{a}^{(1)}_{i_1}, \cdots, \mathbf{a}^{(1)}_{i_N}), \text{where} \ (i_1, \cdots, i_n, \cdots, i_N) \in \Psi.

4.2 Update Process for Core Tensor

Meanwhile, the problem (8) is turned into
\[
\begin{align*}
\arg \min_{\hat{g}^{(n)}} & \left\{ \|g^{(n)} - g^{(n)}\|_2^2 \right\} \\
& = \|\mathbf{X}^{(n)} - \mathbf{X}^{(n)}\|_2^2 + \lambda \|\hat{g}^{(n)}\|_2^2 \tag{14}\end{align*}
\]
where \(\mathbf{X}^{(n)} \in \mathbb{R}^{I_n \times \cdots \times I_n \times I_{N-1} \times I_N}\), \(\mathbf{X}^{(n)} \in \mathbb{R}^{I_n \times \cdots \times I_n \times I_{N-1} \times I_N}\), and \(\lambda \|\hat{g}^{(n)}\|_2^2\).

According to Theorem 2, Assume a row vector \(x = x^{(N)} \otimes x^{(N-1)} \cdots \otimes x^{(n)} \otimes \cdots \otimes x^{(2)} \otimes x^{(1)}\) and a matrix \(\mathbf{Y} = \mathbf{Y}^{(N)} \otimes \mathbf{Y}^{(N-1)} \cdots \otimes \mathbf{Y}^{(2)} \otimes \mathbf{Y}^{(1)}\), \(n \in \{N\}\), \(x \in \mathbb{R}^{I_n \times I_n} \times \mathbb{R}^{I_n \times \cdots \times I_n \times I_{N-1} \times I_N}\), \(Y \in \mathbb{R}^{I_n \times I_n \times I_n} \times \mathbb{R}^{I_n \times \cdots \times I_n \times I_{N-1} \times I_N}\), \(n \in \{N\}\). The vector-matrix multiplication \(xY = x^{(N)} \otimes x^{(N-1)} \cdots \otimes x^{(n)} \otimes \cdots \otimes x^{(2)} \otimes x^{(1)}\) \(\mathbf{Y}^{(N)} \otimes \mathbf{Y}^{(N-1)} \cdots \otimes \mathbf{Y}^{(n)} \otimes \cdots \otimes \mathbf{Y}^{(2)} \otimes \mathbf{Y}^{(1)}\). Can be transformed into \(xY\).
and the approximated gradient from SGD is obtained as:

\[
\nabla f (\mathbf{b}^{(n)T}; \psi^{(n)} ; \{A^{(n)}\}, \tilde{g}^{(n)}) \left( \right) \left( \right)
\]

\[
\left( 1 : \text{CoreTensorPart1} \right) \left( \right)
\]

\[
\left( 2 : \text{CoreTensorPart2} \right) \left( \right)
\]

\[
\left( 3 : \text{CoreTensorPart3} \right) \left( \right)
\]

\[
\left( 4 : \text{CoreTensorPart4} \right) \left( \right)
\]

where \( Q^{(n)_r} \in \mathbb{R}^{J_n} \), and \( Q^{(n)_l} \in \mathbb{R}^{J_n} = a^{(n)_l} \odot \cdots \odot a^{(n)_l-1} \odot \cdots \odot a^{(1)_l} \). \( \sum_{k=1}^{N} \left( i_k - 1 \right) \prod_{m=1}^{k-1} m \neq \prod_{m=1}^{k-1} m \), \( (i_1, \ldots, i_n, \ldots, i_N) \in \Psi \), \( n \in \{N\}, r \in \{R_{\text{core}}\} \).

4.3 Complexity Analysis and Comparison

The computational details and computational complexity of the proposed model are concluded in Table 3 and Algorithm 1 respectively. This section also presents the computational complexity of the coordinate method without the Kruskal product for approximating the core tensor (In experimental section, we refer the cFucker as the stochastic strategy for STD without Kruskal product for approximating the core tensor on GPU CUDA programming).

(1) With the Theorems 1 and 2, the computational complexity of \( D^{(n)} \) can be reduced from direct computation \( O\left( \prod_{k=1}^{N} J_k \right) \) to \( D^{(n)} \) = \( O\left( R_{\text{core}} J_n \sum_{k=1}^{N} \prod_{m=1}^{k-1} m \right) \). The computational complexity and space overhead of \( Q^{(n)_r} \) can be reduced from \( O\left( J_n \sum_{k=1}^{N} J_k \right) \) into \( O\left( \prod_{k=1}^{N} J_k \right) \).

(2) Without Kruskal approximation, in the process of updating \( a^{(n)} \), \( a \in \{I_n\}, n \in \{N\}, \) the computational complexity of the intermediate matrices \( \{ \} \) \( = \left( a^{(n)_l} \odot \cdots \odot a^{(n)_l-1} \odot \cdots \odot a^{(1)_l} \odot a^{(0)_l} \} \), \( D^{(n)} \) \( = \left( g^{(n)} \odot \cdots \odot g^{(n)_l+1} \odot \cdots \odot g^{(n)_l-1} \odot \cdots \odot g^{(1)_l} \} \), \( \Psi \) is \( \left( O\left( \prod_{k=1}^{N} J_k \right) , O\left( \prod_{k=1}^{N} J_k \right) \right) \), respectively, and \( O\left( \prod_{k=1}^{N} J_k \right) \) \( \gg \) \( O\left( \prod_{k=1}^{N} J_k \right) \) in the process of updating \( g^{(n)} \), the space overhead complexity of coefficient matrix \( \{ \} \) \( = \left( g^{(n)} \odot \cdots \odot g^{(n)_l+1} \odot \cdots \odot g^{(n)_l-1} \odot \cdots \odot g^{(1)_l} \} \), \( \Psi \) is \( \left( O\left( \prod_{k=1}^{N} J_k \right) , O\left( \prod_{k=1}^{N} J_k \right) \right) \), respectively, and

Algorithm 1: Algorithm for Updating Factor Matrices and Core Tensor.

**Input:** Sparse tensor \( X \), learning rate \( \gamma \), regularization parameter \( \lambda \).

- **Output:** \( A^{(n)} \), \( B^{(n)} \), \( n \in \{N\} \).

1. for \( n \) from 1 to \( N \) do
   2. Set all \( c^{(n)} \leftarrow 0, G^{(n)} \leftarrow 0, n \in \{N\}, r \in \{R_{\text{core}}\} \);
   3. \( j \leftarrow 1 + \sum_{k=1}^{N} m \neq \prod_{m=1}^{k-1} m \), \( m \in \{N\} \);
   4. for \( r \) from 1 to \( R_{\text{core}} \) do
      5. for \( n \) from 1 to \( N \), \( n \neq \prod_{m=1}^{k-1} m \) do
         6. \( c^{(n)} \leftarrow b^{(n)T} a^{(n)} \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
      7. end for
      8. \( G^{(n)} \leftarrow G^{(n)} + b^{(n)} \); %T
   9. end for
10. end for
11. FacMatPart1 \( \leftarrow X_{in,j} G^{(n)} \) of Part (1) in Eq. (13);
12. FacMatPart2 \( \leftarrow \lambda_{in,j} a^{(n)} \) of Part (2) in Eq. (13);
13. InterMX \( \leftarrow a^{(n)T} G^{(n)} \) of Part (3) in Eq. (13);
14. FacMatPart3 \( \leftarrow \text{InterMX} G^{(n)} \) of Part (3) in Eq. (13);
15. Update \( a^{(n)} \) by SGD in Eq. (5) after summing the parts (1)-(3) of Eq. (13).
16. end for
17. for \( n \) from 1 to \( N \) do
      18. \( j \leftarrow 1 + \sum_{k=1}^{N} m \neq \prod_{m=1}^{k-1} m \), \( m \in \{N\} \);
      19. Set all \( Q^{(n)_r} \leftarrow 0, \text{CoreTensorPart4} \leftarrow 0, n \in \{N\}, r \in \{R_{\text{core}}\} \);
      20. for \( r \) from 1 to \( R_{\text{core}} \) do
         21. for \( n \) from 1 to \( R_{\text{core}} \) do
            22. for \( n \) from 1 to \( N \), \( n \neq \prod_{m=1}^{k-1} m \) do
               23. \( c^{(n)} \leftarrow b^{(n)T} a^{(n)} \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            24. \( a^{(n)} \leftarrow \left( \right) \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            25. end for
            26. \( Q^{(n)_r} \leftarrow \left( \right) \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            27. end for
         28. end for
         29. \( \text{CoreTensorPart4} \leftarrow \text{CoreTensorPart4} + \text{InterMX} \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
      30. end for
      31. end for
16. Update \( a^{(n)} \) by SGD in Eq. (5) after summing the parts (1)-(3) of Eq. (13).
16. end for
17. for \( n \) from 1 to \( N \) do
      18. \( j \leftarrow 1 + \sum_{k=1}^{N} m \neq \prod_{m=1}^{k-1} m \), \( m \in \{N\} \);
      19. Set all \( Q^{(n)_r} \leftarrow 0, \text{CoreTensorPart4} \leftarrow 0, n \in \{N\}, r \in \{R_{\text{core}}\} \);
      20. for \( r \) from 1 to \( R_{\text{core}} \) do
         21. for \( n \) from 1 to \( R_{\text{core}} \) do
            22. for \( n \) from 1 to \( N \), \( n \neq \prod_{m=1}^{k-1} m \) do
               23. \( c^{(n)} \leftarrow b^{(n)T} a^{(n)} \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            24. \( a^{(n)} \leftarrow \left( \right) \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            25. end for
            26. \( Q^{(n)_r} \leftarrow \left( \right) \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
            27. end for
         28. end for
         29. \( \text{CoreTensorPart4} \leftarrow \text{CoreTensorPart4} + \text{InterMX} \); %This step can be accelerated by CUDA Warp Shuffle and Memory Coalescing.
      30. end for
      31. end for
38. end for
39. end for
40. Return: \( A^{(n)} \), \( B^{(n)} \), \( n \in \{N\} \).
TABLE 3: Table of Computational Complexity.

| Updating Factor Matrices                      | Computational Complexity |
|-----------------------------------------------|-------------------------|
| $i_n \in \{I_n\}, n \in \{N\}$               |                         |
| $D^{(n)}_{\Psi^{(n)} M i_n} j$               | $O(DCC^{(n)})$;         |
| $D^{(n)}_{\Psi^{(n)} M i_n j}$               | $O(DCC^{(n)})$;         |
| $X^{(n)}_{i_n M i_n j}$                      | $O(DCC^{(n)})$;         |
| $E^{(n)}_{i_n M i_n j}$                      | $O(DCC^{(n)})$;         |
| $A_{i_n M i_n j}$                            | $O(DCC^{(n)})$;         |
| $J_n$                                         |                         |
| Total                                         | $O\left(\sum_{n=1}^{N}(|J_n + J_n^2|)\right)$; |
| $|J_n| + J_n^2$                               |                         |
| for all $n \in \{N\}$                        |                         |
| Updating Core Tensor                         |                         |
| $Q^{(n), r}_{j}^{(n), r}_{\Psi^{(n)} M i_n}$| $O(QCC^{(n)})$;         |
| Part (1)                                      |                         |
| $Q^{(n), r}_{j}^{(n), r}_{\Psi^{(n)} M i_n}$| $O(QCC^{(n)})$;         |
| Part (2)                                      |                         |
| $J_n$                                         | $O(J_n^2)$;             |
| Part (3)                                      |                         |
| Total                                         | $O\left(\sum_{n=1}^{N}(R_{core} + (J_n^2 + J_n^2))\right)$; |
| $|J_n| + J_n^2$                               |                         |
| for all $n \in \{N\}$                        |                         |
| $J_n$                                         |                         |
| $r_{core} \in \{R_{core}\}$                  |                         |

$O(\prod_{n=1}^{N} J_n)$, and the computational complexity of the intermediate matrices $\{H_{j}, H_{j}^{(n)}, g_{j}^{(n)}| n \in \{N\}\}$ are \{ $\prod_{n=1}^{N} J_n, \prod_{n=1}^{N} J_n$ \}, respectively. According to Table 3 and above analysis, we can conclude that the proposed model can reduce the exponential overhead into linear one with the Theorems 1 and 2 and Kruskal approximation strategy.

5 cuFastTucker on GPUs

The Section 4 solves the problem of high computational overhead for the factor matrix and core tensor of STD with Kruskal approximation and Theorems 1 and 2. However, the STD for the HOHDST data still relies on the modern HPC resource to obtain the real-time result. Due to the basic computational part of thread and thread block and fine-grained and high parallelization of the proposed model, the GPU is chosen to further accelerate the proposed model (cuFastTucker). The parallelization strategy is divided into two parts: (1) thread parallelization within a thread block; (2) parallelization of thread block. In this section, data partition and communication on multi-GPUs are also presented.

5.1 CUDA Thread Parallelization within a Thread Block

GPUs is a Single Instruction Multiple Devices architecture, where a thread block can be packed as a thread group, and the current size of scheduling unit (Warp) in CUDA GPUs is 32. Hence, the number of threads within a thread block are a divisor or multiple of 32. This means that when $J_n, n \in \{N\}$ is a divisor or multiple of 32, it has better performance. Fig. 1 illustrates the two key steps $\{GS^{(n)}, Q^{(n), r}\}$ for updating the factor matrix and core tensor in a CUDA thread block. The major optimization techniques in cuFastTucker are concluded as:

Warp Shuffle: as the Fig. 4 illustrate, warp shuffle instructions in cuFastTucker are used to compute the dot product or sum operations $\{GS^{r, (n)}, Q^{(n), r}\}$ (Lines 6 and 23 in Algorithm 1) and then broadcast the result, automatically. The warp shuffle instruction needs additional hardware support, with lower latency and no additional memory resources, and it allows a thread to directly read the register values of other threads in the same thread warp, which has better communication efficiency than reading and writing data through the shared memory.

On-chip Cache: the current GPUs allows programmers...
to control the caching behavior of each memory instruction of the on-chip L1 cache. In the SGD based method, the read-only index and read-only value of the non-zero element may be frequently reused in the near future (temporal reuse) or by other thread blocks (spatial reuse). We use ldg to modify read-only memory to improve access efficiency.

**Memory Coalescing:** in order to utilize the bus bandwidth, GPUs usually coalesce the memory accesses of multiple threads into fewer memory requests. Assuming that the multiple threads access addresses within 32 bytes, their access can be completed through a memory request, which can greatly improve bandwidth utilization. According to the CUDA code, all the variable matrices \( \{ A(n), G(n), B(n) | n \in \{ N \} \} \) are stored as the form of \( \{ A(n) \in \mathbb{R}^{I_n \times J_n}, G(n)^T, B(n)^T \in \mathbb{R}^{R_{core} \times J_n} | n \in \{ N \} \} \) to ensure that consecutive threads access consecutive memory addresses.

**Register Usage:** the register file is the fastest storage unit on GPUs, so we save every reusable variable in registers. Although the total number of registers on GPUs is fixed, our algorithm only needs to use a small number of registers. The current number of GPUs is completely sufficient.

**Shared Memory:** the threads within a thread block use the shared memory which is much faster than global memory, and the register is not suitable for storing continuous vectors. The frequent used intermediate vectors lie in shared memory, and these vectors will be used in the next process.

### 5.2 CUDA Thread Block Parallelization

From Algorithm 4, the computational step for each feature vector \( a^{(n)}_{i_1, \ldots, i_n} \in \{ I_n \}, n \in \{ N \} \) is independent which has fine-grain parallelization. Meanwhile, Kruskal approximation vectors \( b^{(n)}_{i, r} \in \{ N \}, r \in \{ R_{core} \} \) are dependent. Thus, the vectors \( b^{(n)}_{i, r}, n \in \{ N \}, r \in \{ R_{core} \} \) should be updated, simultaneously. The allocated \( NR_{core} \) thread blocks pre-compute the gradient following the line 23 of Algorithm 4 and then update \( b^{(n)}_{i, r}, n \in \{ N \}, r \in \{ R_{core} \} \), simultaneously. There are two strategies to allocate the TB thread blocks to update the feature vector \( a^{(n)}_{i_1, \ldots, i_n} \in \{ I_n \}, n \in \{ N \} \) feature vectors are allocated to the TB thread blocks; (2) each thread block selects a index from the one-step sampling set \( \Psi \) and each thread block compute the gradient following the line 6 of Algorithm 4.

### 5.3 Workload Partitioning

The scale of data that can be processed by a single GPU is limited, and otherwise, the time required to process the data is not acceptable. The HOHDST data is divided so that it can be processed simultaneously on multiple GPUs. Given an \( N \)-order tensor \( X \in \mathbb{R}^{I_1 \times \cdots \times I_N} \) and \( M \) GPUs, we averagely cut each order of the tensor into \( M \) parts, so the tensor is evenly divided into \( M^N \) blocks \( \{ X_{1,1}, \ldots, X_{1,1 \cdots 2}, \ldots, X_{M,M}, \ldots, X_{M^N,1}, X_{M^N,2}, \ldots, X_{M^N,1} \} \) in the same period of time, each GPU is responsible for processing one of these \( M^N \) blocks. In order to avoid data conflicts between multiple GPUs, the indexes of the same order of the blocks that are responsible for different GPUs at the same time are different. For example, block \( X_{1,1 \cdots 1} \) and block \( X_{2,2 \cdots 1} \) cannot be processed at the same time.

As shown in Figure 2, two GPUs are used to process a 3-dimensional tensor, which is equally divided into \( 2 \times 2 \times 2 \) blocks. When updating the factor matrices, GPU 1 and GPU 2 update \( \{ A^{(1)}_1, A^{(2)}_1, A^{(3)}_1 \}, \{ A^{(1)}_2, A^{(2)}_2, A^{(3)}_2 \}, \{ A^{(1)}_3, A^{(2)}_3, A^{(3)}_3 \}, \{ A^{(1)}_4, A^{(2)}_4, A^{(3)}_4 \} \) and \( \{ A^{(1)}_5, A^{(2)}_5, A^{(3)}_5 \} \) respectively. In this process, the processing of GPU 1 and GPU 2 does not conflict, and after GPU 1 and GPU 2 update blocks \( \{ A^{(1)}_1, A^{(2)}_1, A^{(3)}_1, A^{(2)}_2 \}, \{ A^{(1)}_2, A^{(2)}_2, A^{(3)}_2 \} \) and \( \{ A^{(1)}_3, A^{(2)}_3, A^{(3)}_3 \} \) to each other. When updating the core tensor, it is only necessary to update the core tensor after accumulating all the gradients.

![Fig. 2: An example of workload partitioning of a tensor.](image-url)

**TABLE 4: Real World Datasets**

| Dataset   | Netflix | Yahoo!Music | Amazon Reviews |
|-----------|---------|-------------|---------------|
| Netflix   | 480,189 | 1,000,990   | 4,821,207     |
| I2        | 17,770  | 624,961     | 1,774,269     |
| I3        | 2,182   | 3,075       | 1,805,187     |
| (I)       | 99,072  | 112,250     | 2,722,286     |
| (Ω)       | 1,408,395 | 2,527,989     |               |
| Max Value | 5       | 5           |               |
| Min Value | 1       | 0.025       |               |

**TABLE 5: Synthesis Datasets**

| Dataset   | Order-3 | Order-4 | Order-5 | Order-6 & Order-10 |
|-----------|---------|---------|---------|-----------------|
| Order-3   | 10,000  | 10,000  | 10,000  | 10,000          |
| Order-4   | 1G      | 800M    | 600M    | 100M            |
| Order-5   | 5       | 5       | 5       | 5               |
| Order-6 & Order-10 | 1 | 1 | 1 | 1 |

**TABLE 6: The initial learning rate and regularization parameters of cuTucker on Netflix and Yahoo!Music datasets.**

| Dataset      | Netflix | Yahoo!Music |
|--------------|---------|-------------|
| \( \alpha \) | 0.0005  | 0.0045      |
| \( \beta \)  | 0.0025  | 0.0005      |
| \( \gamma \) | 0.01    | 0.01        |
| \( \delta \) | 0.0025  | 0.0005      |
Fig. 3: The accuracy influence of the baseline algorithm (cuTucker) and cuFastTucker in various parameters $R_{core} = \{8, 16, 32\}$ with fixed $J_n = \{8, 16, 32\}$, $n \in \{N\}$.

TABLE 7: The initial learning rate and regularization parameters of cuFastTucker on Netflix and Yahoo!Music datasets.

| $J$ | $R_{core}$: Netflix | $R_{core}$: Yahoo!Music |
|-----|---------------------|-------------------------|
| 8   | 0.0009              | 0.0000                   |
| 16  | 0.005               | 0.005                    |
| 32  | 0.005               | 0.005                    |

TABLE 8: The time (Seconds) overhead influence for updating the core tensor of cuFastTucker with the intermediate matrix on shared memory and global memory.

| $J$ | Netflix | Yahoo!Music |
|-----|---------|-------------|
| 4   | 0.274793| 3.213333    |
| 8   | 0.707503| 8.155324    |

TABLE 9: The time (Seconds) overhead influence for updating the factor matrices of cuFastTucker with the core tensor on shared memory and global memory (NVIDIA Tesla P100 GPU).

| $J$ | Netflix | Yahoo!Music |
|-----|---------|-------------|
| 4   | 0.180897| 0.415549    |
| 8   | 0.382566| 2.751087    |

TABLE 10: The time (Seconds) overhead influence for updating the core tensor of cuFastTucker with the core tensor on shared memory and global memory (NVIDIA Tesla P100 GPU).

| $J$ | Netflix | Yahoo!Music |
|-----|---------|-------------|
| 4   | 0.230354| 0.509430    |
| 8   | 0.642367| 1.272383    |

TABLE 11: The time (Seconds) overhead influence for updating the factor matrices of cuFastTucker with the core tensor on shared memory and global memory (NVIDIA TITAN RTX GPU).

| $J$ | Netflix | Yahoo!Music |
|-----|---------|-------------|
| 4   | 0.294781| 0.578777    |
| 8   | 0.707503| 1.272383    |

6 EXPERIMENTS

This section mainly answers the following main questions: (1) the influence of the parameters $\{J_n, R_{core}|n \in \{N\}\}$,
and the computational overhead of each part of cuFastTucker (Section 6.1); (2) the accuracy performance of cuFastTucker (Section 6.2); (3) the scalability of cuFastTucker and the performance of cuFastTucker on multi-GPUs (Section 6.3). In this section, cuTucker, P-tucker [46], Vest [47] and the computational overhead presented in Theorems 1 and 2.

6.1 Experimental Setup

The experiments are run on Intel(R) Xeon(R) Silver 4110 CPU @ 2.10GHz with 32 processors and 4 NVIDIA Tesla P100 GPUs with CUDA version 10.0. The experimental datasets are divided into real and synthesis sets in Tables 4 and 5 respectively. The 3 real world datasets are listed as: Netflix¹, Yahoo!Music² and Amazon Reviews³. The Netflix and Yahoo!Music datasets are used to get the baseline accuracy, and the Amazon Reviews dataset is used to test the ability of cuFastTucker on large-scale data. The 8 synthesis datasets are produced to test the overall performance of cuFastTucker. The accuracy is measured by RMSE as $\sqrt{\frac{1}{J} \sum_{(i,j) \in \Gamma} (v_{i,j} - \hat{v}_{i,j})^2}$, and MAE as $\frac{1}{J} \sum_{(i,j) \in \Gamma} |v_{i,j} - \hat{v}_{i,j}|$, where $\Gamma$ is the test dataset.

The dynamic learning rate of cuTucker and cuFastTucker uses the strategy in [49] as $\gamma_t = \frac{\alpha}{1 + \beta t}$, where the parameters $\{\alpha, \beta, t, \gamma_t\}$ represent the initial learning rate, adjusting parameter of the learning rate, the number of current iterations, and the learning rate at $t$ iterations, respectively. The parameters in cuTucker and cuFastTucker are listed in Table 6 and Table 7 respectively. $\{\alpha_a, \beta_a, \lambda_a\}$ are denoted as the parameters for updating the feature matrix in cuTucker and cuFastTucker, and $\{\alpha_g, \beta_g, \lambda_g\}$ and $\{\alpha_b, \beta_b, \lambda_b\}$ are denoted as the parameters for updating the core tensor in cuTucker and cuFastTucker, respectively.
cuFastTucker in various parameters $R_{core} = \{8, 16, 32\}$ with fixed $J_n = \{8, 16, 32\}, n \in \{N\}$. Fig. 3 shows the accuracy influence of the baseline algorithm (cuTucker) and cuFastTucker in various parameters $J_n = R_{core} = \{8, 16, 32\}, n \in \{N\}$. Fig. 4 depicts the training time overhead according to the varying of the value of $J_n, n \in \{N\}$ and $R_{core}$. The influence of access time for shared memory and global memory on GPU is presented in Tables 8-10. The conclusion is listed as the following 2 parts:

1. As the Figs. 3 and 4 show, both increasing the value of $R_{core}$ and $J_n$ can increase the accuracy (decrease the value of RMSE and MAE) and the value of $J_n, n \in \{N\}$ plays more influence on bigger dataset (Yahoo!Music) than smaller one (Netflix). Fig. 5 shows that when $R_{core} = J_n, n \in \{N\}$, the accuracy performance (RMSE and MAE) of cuFastTucker will overwhelm the cuTucker, which means that the core tensor has low-rank inference and the compression rate is $(\sum_{n=1}^{N} R_{core}/n)/\prod_{n=1}^{N} J_n$. Fig. 4 also illustrates the accuracy performance (RMSE and MAE) of updating factor matrix with core tensor (Factor+Core) and factor matrix only (Factor). In bigger dataset (Yahoo!Music) the accuracy gap between the curves ‘Factor+Core’ and ‘Factor’ is much small than smaller volume one (Netflix).

2. Figs. 5(a) and 5(b) compare the time overhead of cuTucker and cuFastTucker. The time overhead comprises of the overhead of updating factor matrix (Factor) and core tensor (Core). As the Figs. 5(a) and 5(b) show, the time overhead of both cuTucker (Factor) and cuTucker (Core) is much higher than cuFastTucker (Factor) and cuFastTucker (Core). The reason is that the Kruskal approximation and overhead reduction by Theorems 1 and 2 can reduce the computational overhead of cuFastTucker. As the Figs. 5(a) and 5(b) the computational overhead of cuFastTucker is increased linearly with the increasing of the value of $R_{core}$ and $J_n$. Shared memory and global memory are the main memory classes in GPUs.

Due to Kruskal approximation for core tensor of cuFastTucker, the approximation matrix $B^{(n)}, n \in \{N\}$ rather than the core tensor $G^{(n)}$ and unfolding matrices $G^{(n)}, n \in \{N\}$ can be accessed on shared memory. As the Tables 8 and 12 show, memory accessing speed on shared memory is slightly faster than global memory. cuTucker has a huge intermediate matrix when updating the core tensor, which cannot be stored in shared memory when $J_n, n \in \{N\}$ is greater than 8. While cuFastTucker can store larger core tensor in shared memory, this situation is more obvious when the order is larger. Further, cuFastTucker makes it easier to put memory hotspot core tensor into shared memory, which will further reduce the computation time. Since the shared memory and the L1 cache share a piece of on-chip memory, the increase of the shared memory will lead to a decrease in the size of L1 cache, resulting in a slight decrease in program performance. This
is more noticeable on the NVIDIA TITAN RTX GPU with larger caches.

**TABLE 13**: The time overhead (Seconds) to update the low rank factor matrices in a single iteration of STOA algorithms.

|          | Netflix       | Yahoo/Music  |
|----------|---------------|---------------|
| P-Tucker | 20.393240(106.73X) | 122.95439(197.57X) |
| Vest     | 75.57325(392.74X)       | 503.045186(747.54X)  |
| SGD_Tucker | 12.111656(62.94X) | 29.152721(43.32X) |
| cuTucker | 0.696915(3.62X)          | 1.761734(2.61X)       |
| cuFastTucker | 0.192428       | 0.672929        |

**6.3 Comparison with STOA Approaches**

Fig. 6 and Table 13 depict the comparison of the convergence and accuracy performances and the running time per iteration is presented on Table 13. P-Tucker [46] and Vest [47] and SGD_Tucker [48] are CPU based methods. Meanwhile, cuTucker and cuFastTucker are GPUs based methodologies. To ensure the running fairness, the CPU and GPU run independently without the interference of other works. All comparison methodologies run on \( J_n = 4, n \in \{ N \} \), and in cuFastTucker runs on \( R_{\text{core}} = 4 \). Some algorithms lack the update of the core tensor, and we only compare the update of the factor matrix here. As the Fig. 6 show, P-Tucker has the fastest RMSE decreasing speed at the beginning but it goes slower in the later stage. P-Tucker runs in a bit of unstable. All the methods can obtain the same overall accuracy after 20 iterations. SGD_Tucker runs much faster than P-Tucker and Vest but a bit slower than cuTucker and cuFastTucker. The convergence speed and accuracy of cuTucker and cuFastTucker overwhelm the other three algorithms. As shown in Table 13, cuFastTucker and cuTucker get the top-2 and due to Kruskal approximation and the overhead reduction of vector multiplication in Theorems 1 and 2, cuFastTucker obtain 3.62X and 2.61X speedup than cuTucker on Netflix and Yahoo!Music datasets, respectively.

**6.4 Scalability on Large-scale Data and Speedup on Multi-GPUs**

Fig. 7(a) illustrates the computational time of updating core tensor and factor matrix of cuTucker and cuFastTucker. Both the cuTucker and cuFastTucker have the scalability with the order of \( \{ 5, 6, 7, 8, 9, 10 \} \). Whatever updating the core tensor and factor matrix, cuTucker spends much longer time than cuFastTucker. Figs. 7(b) and 7(c) present the speedup on \( \{ 2, 4, 5 \} \) GPUs and both the cuTucker and cuFastTucker can obtain the near linear speedup. Fig. 8 shows that with fixed order, cuFastTucker can obtain more stable speedup on the high non-zero entries datasets. From the results of Figs. 7(b) and 7(c) and 8(a) and 8(c), cuFastTucker can get near the linear speedup on synthesis datasets. For the very large dataset amazn, cuFastTucker runs perfectly on 4 P100s. When \( R_{\text{core}} = J_n = 4, n \in \{ N \} \), the time for a single update of the factor matrix and core tensor are 10.769747 seconds and 12.953006 seconds, respectively.

**7 CONCLUSION**

High-Order, High-Dimension, and Sparse Tensor (HOHDST) is a widely used data form in ML community, etc, spatiotemporal dynamics social networks and recommender systems, and network flow prediction. Thus, it is non-trivial to find an efficient and low computational overhead methodologies for Sparse Tensor Decomposition (STD) to get the key feature of HOHDST data. To solve this problem, cuFastTucker is proposed which comprise of Kruskal core tensor and Theorems 1 and 2 to reduce the computational overhead. Meanwhile, low data-dependence gives the cuFastTucker with fine-grained parallelization on CUDA GPU. The experimental results show that cuFastTucker has linear computational time and space overheads and cuFastTucker runs at least 2.62X and at most 747.54X faster than STODA approaches. In the future works, we will explore how to take advantages of cuFastTucker to accelerate and compress modern Deep Neural Networks, etc, CNN, LSTM, RNN, and Transformer.

**ACKNOWLEDGMENTS**

This work has also been partly funded by the Program of National Natural Science Foundation of China (Grant No. XXXXXXXXXX), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. XXXXXXXXXX).

**REFERENCES**

[1] C. Skordis, “The tensor-vector-scalar theory and its cosmology,” *Classical and Quantum Gravity*, vol. 26, no. 14, p. 143001, 2009.

[2] T. G. Kolda and B. W. Bader, “Tensor decompositions and applications,” *SIAM review*, vol. 51, no. 3, pp. 455–500, 2009.

[3] A. Cichocki, N. Lee, I. Oseledeets, A.-H. Phan, Q. Zhao, and D. P. Mandic, “Tensor networks for dimensionality reduction and large-scale optimization: Part 1 low-rank tensor decompositions,” *Foundations and Trends® in Machine Learning*, vol. 9, no. 4-5, pp. 249–429, 2016.

[4] A. Cichocki, A. Phan, I. Oseledeets, Q. Zhao, M. Sugiyama, N. Lee, and D. Mandic, “Tensor networks for dimensionality reduction and large-scale optimizations: Part 2 applications and future perspectives,” *Foundations and Trends in Machine Learning*, vol. 9, no. 6, pp. 431–673, 2017.

[5] T. Wang, X. Xu, Y. Yang, A. Hanjalic, H. T. Shen, and J. Song, “Matching images and text with multi-modal tensor fusion and re-ranking,” in *Proceedings of the 27th ACM international conference on multimedia*. ACM, 2019, pp. 12–20.

[6] M. Hou, J. Tang, J. Zhang, W. Kong, and Q. Zhao, “Deep multi-modal multilinear fusion with high-order polynomial pooling,” in *Advances in Neural Information Processing Systems*, 2019, pp. 12113–12122.

[7] P. P. Liang, Z. Liu, Y.-H. H. Tsai, Q. Zhao, R. Salakhutdinov, and L.-P. Morency, “Learning representations from imperfect time series data via tensor rank regularization,” in *Proceedings of the Annual Meeting of the Association for Computational Linguistics*, 2019.

[8] Y. Liu, L. He, B. Cao, S. Y. Philip, A. B. Ragan, and A. D. Leow, “Multi-view multi-graph embedding for brain network clustering analysis,” in *AAAI Conference on Artificial Intelligence*, 2018.

[9] J. Liu, C. Zhu, and Y. Liu, “Smooth compact tensor ring regression,” *IEEE Transactions on Knowledge and Data Engineering*, 2020.

[10] J. Kossaifi, C. Z. Lipton, A. Koolenboon, A. Khanna, T. Furlanello, and A. Anandkumar, “Tensor regression networks,” *Journal of Machine Learning Research*, vol. 21, pp. 1–21, 2020.

[11] R. Yu and Y. Liu, “Learning from multiway data: Simple and efficient tensor regression,” in *International Conference on Machine Learning*. PMLR, 2016, pp. 373–381.

[12] Z. Hao, L. He, B. Chen, and X. Yang, “A linear support higher-order tensor machine for classification,” *IEEE Transactions on Image Processing*, vol. 22, no. 7, pp. 2911–2920, 2013.

[13] G. G. Calvi, V. Lucic, and D. P. Mandic, “Support tensor machine for financial forecasting,” in *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 8152–8156.
Fig. 6: Comparison with STOA approaches.

(a) RMSE on Netflix
(b) MAE on Netflix
(c) RMSE on Yahoo! Music
(d) MAE on Yahoo! Music

Fig. 7: (a) Scalability on synthesis datasets; (b)-(c) speedup on Netflix and Yahoo! Music datasets, respectively.

Fig. 8: Scale up to multiple GPUs
“A novel stochastic optimization strategy for parallel machine learning,” in 12th USENIX Symposium on operating systems design and implementation, 2016, pp. 265–283.

[16] V. N. Ioannidis, A. S. Zamzam, G. B. Giannakis, and N. D. Sidiropoulos, “Coupled graph and tensor factorization for recommender systems and community detection,” IEEE Transactions on Knowledge and Data Engineering, 2019.

[17] X. Luo, H. Wu, H. Yuan, and M. Zhou, “Temporal pattern-aware qos prediction via biased non-negative latent factorization of tensors,” IEEE transactions on cybernetics, 2019.

[18] K. Xie, X. Li, X. Wang, G. Xie, J. Wen, and D. Zhang, “Active sparse mobile crowd sensing based on matrix completion,” in Proceedings of the International Conference on Management of Data, 2019, pp. 195–206.

[19] X. Wang, L. T. Yang, X. Xie, J. Jin, and M. I. Deen, “A cloud-edge computing framework for cyber-physical-social services,” IEEE Communications Magazine, vol. 55, no. 11, pp. 80–85, 2017.

[20] 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 Transactions on Network Science and Engineering, 2019.

[21] Y. Luo, P. Yao, K. Ramamohanarao, C. X. and Y. Wen, “Tensor canonical correlation analysis for multi-view dimension reduction,” IEEE transactions on Knowledge and Data Engineering, vol. 27, no. 11, pp. 3111–3124, 2015.

[22] X. Liu, X. Zhu, M. Li, L. Wang, E. Zhu, T. Liu, M. Kloft, D. Shen, J. Yin, and W. Gao, “Multiple kernel learnability for incomplete kernels,” IEEE transactions on pattern analysis and machine intelligence, 2019.

[23] L. van Der Maaten, E. Postma, and J. Van den Herik, “Dimensionality reduction: a comparative,” Journal of Machine Learning Research, vol. 10, no. 66-71, p. 13, 2009.

[24] P. Symeonidis, A. Nanopoulos, and Y. Manolopoulos, “Tag recommendations based on tensor dimensionality reduction,” in Proceedings of the ACM conference on Recommender systems, 2019, pp. 5188–5197.

[25] Y. Nesterov, Introductory lectures on convex optimization: A basic course. Springer Science & Business Media, 2013, vol. 87.