High-Performance Tensor Learning Primitives Using GPU Tensor Cores

Xiao-Yang Liu, *Member, IEEE*, Zeliang Zhang, Zhiyuan Wang, Han Lu, Xiaodong Wang, *Fellow, IEEE*, and Anwar Walid, *Fellow, IEEE*

Abstract—Tensor learning is a powerful tool for big data analytics and machine learning, e.g., gene analysis and deep learning. However, tensor learning algorithms are compute-intensive since their time and space complexities grow exponentially with the order of tensors, which hinders their application. In this paper, we exploit the parallelism of tensor learning primitives using GPU tensor cores and develop high-performance tensor learning algorithms. First, we propose novel hardware-oriented optimization strategies for tensor learning primitives on GPU tensor cores. Second, for big data analytics, we employ the optimized tensor learning primitives to accelerate the CP tensor decomposition and then apply it for gene analysis. Third, we optimize the Tucker tensor decomposition and propose a novel Tucker tensor layer to compress deep neural networks. We employ natural gradients to train the neural networks, which only involve a forward pass without backpropagation and thus are suitable for GPU computations. Compared with TensorLab and TensorLy libraries on an A100 GPU, our third-order CP tensor decomposition achieves up to $16.32\times$ and $32.25\times$ speedups; and $6.09\times$ and $6.72\times$ speedups for our third-order Tucker tensor decomposition. The proposed fourth-order CP and Tucker tensor decompositions achieve up to $30.65\times$ and $5.41\times$ speedups over the TensorLab. Our CP tensor decomposition for gene analysis achieves up to $5.88\times$ speedup over TensorLy. Compared with a conventional fully connected neural network, our Tucker tensor layer neural network achieves an accuracy of 97.9%, a speedup of $4.47\times$, and a compression ratio of 2.92 at the cost of 0.4% drop in accuracy.

Index Terms—Tensor learning, tensor computing, GPU tensor cores, tensor layer, neural network

1 Introduction

As a powerful mathematical tool, tensors play an important role in science development, including physics [3], chemistry [4], and material science [5]. Tensors are natural representations of multi-dimensional data in many fields, such as video processing [6], financial forecasting [7], social networks [8], graph computing [9][10] and federated learning [11]. Deep learning methods leverage the high-order structure of tensors and achieve state-of-the-art performance in various areas [12]. Tensor learning, which combines tensor methods with deep learning, further utilizes the structure of tensors and leads to improved performance in many aspects [13], such as better robustness [14], compact models [15], and computational speedups [16].

- *Xiao-Yang Liu and Xiaodong Wang are with the Department of Electrical Engineering, Columbia University, New York, NY 10027 USA. E-mail: {xl2427, xw2008}@columbia.edu.*
- *Zeliang Zhang and Zhiyuan Wang are with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China. E-mail: {hust0426, vinlee624}@gmail.com.*
- *Han Lu is with the School of Computer Engineering and Science, Shanghai University, Shanghai 201824, China. E-mail: baxlumen@shu.edu.cn.*
- *Anwar Walid is with the Columbia University, New York, NY 10027 USA, and also with the Nokia-Bell Labs, New Providence, NJ 07974 USA. E-mail: anwar.walid@nokia-bell-labs.com.*

Manuscript received 5 October 2021; revised 22 September 2022; accepted 13 November 2022. Date of publication 15 December 2022; date of current version 10 May 2023.
(Xiao-Yang Liu and Zeliang Zhang have contributed Equally.)
(Corresponding author: Xiaodong Wang.)
Recommended for acceptance by M. Annavaram.
This article has supplementary downloadable material available at https://doi.org/10.1109/TC.2022.3222955, provided by the authors.
Digital Object Identifier no. 10.1109/TC.2022.3222955

Tensor learning methods has been widely used in many fields, such as big data analytics [17] [18] and pattern recognition [19] [20]. Wang et al. [18] applied CP tensor decomposition for gene analysis, which helped study the relationships of gene expression between different individuals and tissues parallelly. X.-Y. Liu et al. [21] proposed a tensor generative adversarial net for real-time indoor localization, which exploited the tensor structure to improve generation accuracy. Yin et al. [19] proposed a Tucker tensor layer to compress deep neural networks, which reduced a large number of parameters and save memory consumption. Dai et al. [22] proposed tensor-train faster-RCNN for video segmentation, which has an efficient interference process and a low occupation of memory.

However, computation efficiency is a major obstacle to tensor learning algorithms [23]. Specifically, the time complexities and memory footprint of tensor learning algorithms grow exponentially with the order of tensors. Fortunately, the tensor learning algorithms involve many highly parallel tensor primitives, e.g., tensor product and tensor contraction. We can fully exploit the computing power of GPU tensor cores to optimize the tensor learning algorithms.

Many existing works have been proposed to optimize tensor learning algorithms. X.-Y. Liu et al. [24] composed a book chapter on high-performance tensor learning for neural networks. T. Zhang et al. [25] [26] developed low-tubal-rank tensor learning primitives on GPUs by optimizing data transfer. Zou et al. [27] exploited the inherent parallelism of tensor operations and the high memory bandwidth of GPU to accelerate social network analysis. TensorLab (using MATLAB) [28] and TensorD (using TensorFlow) [29]

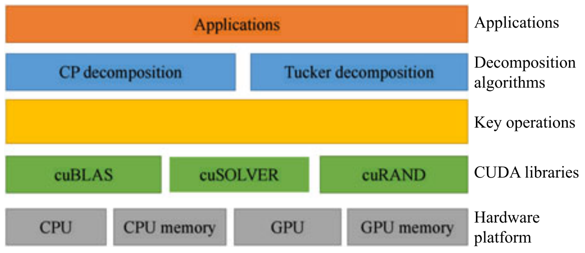


Fig. 1. Architecture of the cuTensorLearning library.

provided GPU implementations of tensor decomposition algorithms, but they can only support medium-scale tensors and cannot meet the demand of big data analysis. TensorLy (Python) [30] provides implementations of tensor methods and deep tensorized neural networks on GPUs, but the library did not fully utilize the computing power of GPU tensor cores and still requires long running time.

GPU tensor cores [31] are specialized hardware that employs mixed-precision computation [32] to accelerate tensor operations and have been studied in many recent works, such as accelerating neural networks [33][34], scientific computing [35][36] and quantum-based molecular dynamics simulations [37]. Tensor learning algorithms are composed of tensor learning primitives. The parallelism of tensor learning primitives shows acceleration potential for GPU tensor cores. Thus, we can map tensor learning primitives onto GPU tensor cores to achieve high performance. Then tensor learning algorithms can be optimized by using the high-performance primitives. However, three major challenges need to be addressed: 1) to reasonably map the tensor learning primitives onto the parallel GPU tensor cores; 2) to coordinate the computing resources with memory resources, and design high-performance schedule strategies for tensor learning algorithms; 3) to support large-scale tensor learning algorithms with limited GPU memory.

In this article, we present efficient primitives using GPU tensor cores to support high-performance tensor learning algorithms. We organize them into a *cuTensorLearning* library, as shown in Fig. 1. The bottom layer is a heterogeneous hardware platform with CPUs and GPUs. Above it are CUDA libraries including cuBLAS, cuSOLVER, cuSPARSE and cuRAND. In the middle layer, we optimize a set of tensor learning primitives, including tensor product and tensor contraction. Using these optimized tensor learning primitives as building blocks, we further optimize two widely used tensor learning algorithms, CP and Tucker tensor decompositions. Finally, the goal of the *cuTensorLearning* library is to accelerate real applications. We demonstrate two applications, i.e., gene analysis using CP tensor decomposition and a Tucker tensor layer for deep neural networks. Our codes are available at https://github.com/XiaoYangLiu-FinRL/cuTensor-CP-Tucker.

Our contributions are summarized as follows:

- We optimize tensor learning primitives using GPU tensor cores, including tensor products and tensor contraction by a matricization-free memory access. We propose a shard mode to support large-scale tensor learning computations which exceed the capacity of GPU memory. We optimize third- and fourth-order CP and Tucker tensor decomposition algorithms using the above primitives.
- We apply optimized tensor learning primitives to two demo applications. We demonstrate the gene analysis using optimized CP tensor decomposition. We present a Tucker tensor layer for highly compact deep neural networks and employ natural gradient for network training.
- We provide extensive performance evaluations. Our third-order CP tensor decomposition achieves $16.32\times$ and $32.25\times$ speedups, compared with TensorLab library and TensorLy, respectively; and $6.09\times$ and $6.72\times$ speedups for our third-order Tucker tensor decomposition. The proposed fourth-order CP and Tucker tensor decompositions achieve up to $30.65\times$ and $5.41\times$ speedups over TensorLab library. We further evaluate two applications on real-world datasets, gene analysis using CP tensor decomposition on the Genotype-Tissue Expression (GTEx) v6 dataset [38] and a Tucker tensor layer for fully connected neural network on handwritten digit database, MNIST [39]. For gene analysis, our CP tensor decomposition implementation achieves up to $5.88\times$ speedup over TensorLy library. Our Tucker tensor layer achieves up to $4.47\times$ speedups over conventional fully connected layer using CUDA cores and has a compression ratio of 2.92 at a cost of 0.4% drop in accuracy.

The remainder of this paper is organized as follows. Section 2 describes tensor learning primitives and briefly summarizes tensor learning algorithms. Section 3 optimizes tensor learning primitives using GPU tensor cores. Section 4 presents an optimized CP tensor decomposition for third- and fourth-order cases and its application to gene analysis. Section 5 presents an optimized Tucker tensor decomposition for third- and fourth-order tensors and proposes a novel Tucker tensor layer for neural networks. In Section 6, we evaluate the performance of tensor learning primitives and algorithms. Section 7 discusses related works. We conclude this paper in Section 8.

2 Tensor Primitives and Algorithms

In this section, we first describe key tensor operations and then two popular algorithms, i.e., CP tensor decomposition and Tucker tensor decomposition.

2.1 Notions and Tensor Primitives

We use uppercase calligraphic letters to denote third-order tensors, e.g., $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, uppercase boldface letters to denote matrices and lowercase boldface letters to denote vectors, e.g., $\boldsymbol{X} \in \mathbb{R}^{I\times J}$ and $\boldsymbol{x} \in \mathbb{R}^{I}$. The notations are given in Table 1.

Rank-One Tensor: A third-order tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ has *rank-one* if it can be represented as the outer product of three vectors. This generalizes to higher-order tensors.

Khatri-Rao Product: The Khatri-Rao product of matrices $\boldsymbol{A} \in \mathbb{R}^{I\times K}$ and $\boldsymbol{B} \in \mathbb{R}^{J\times K}$ is denoted as $\boldsymbol{A} \odot \boldsymbol{B} \in \mathbb{R}^{IJ\times K}$

$$\boldsymbol{A} \odot \boldsymbol{B} = [\boldsymbol{A}_1 \otimes \boldsymbol{B}_1 \ \boldsymbol{A}_2 \otimes \boldsymbol{B}_2 \ \cdots \ \boldsymbol{A}_K \otimes \boldsymbol{B}_K]. \qquad (1)$$

TABLE 1
Notations

Symbol	Meaning
$\boldsymbol{x}, \boldsymbol{X}, \mathcal{X}$	Vector, matrix, tensor
$*$	Hadamard (element-wise) product
$\circ$	Outer product
$\odot$	Khatri-Rao product
$\otimes$	tensor (Kronecker) product
$^\top$	Matrix transpose
†	Moore-Penrose pseudo-inverse
$[I]$	Set $\{1, 2, \ldots, I\}$
$\boldsymbol{X}(:, j)$ or $\boldsymbol{X}_j$	The j-th column of $\boldsymbol{X}$
$\boldsymbol{X}(i, j)$ or $\boldsymbol{X}_{ij}$	The (i, j)-th entry of $\boldsymbol{X}$
$\boldsymbol{X}(i, j, k)$ or $\mathcal{X}_{ijk}$	(i, j, k)-th entry of $\mathcal{X}$
$\boldsymbol{X}(:, j, k)$, $\boldsymbol{X}(i, :, k)$, $\boldsymbol{X}(i, j, :)$	Mode-1, -2, -3 tube of $\mathcal{X}$
$\boldsymbol{X}(i, :, :)$	i-th horizontal slice of $\mathcal{X}$
$\boldsymbol{X}(:, j, :)$	j-th lateral slice of $\mathcal{X}$
$\boldsymbol{X}(:, :, k)$ or $\mathcal{X}^{(k)}$	k-th frontal slice of $\mathcal{X}$
$\mathcal{X}_{(n)}$	mode-n matricization of $\mathcal{X}$
$\Vert\mathcal{X}\Vert_F$	Frobenius norm of $\mathcal{X}$

Tensor (Kronecker) Product: The tensor product of $\boldsymbol{A} \in \mathbb{R}^{I \times J}$ and $\boldsymbol{B} \in \mathbb{R}^{K \times L}$ is denoted as $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{IK \times JL}$

$$\boldsymbol{A} \otimes \boldsymbol{B} = \begin{bmatrix} A_{11}\boldsymbol{B} & \cdots & A_{1J}\boldsymbol{B} \\ \vdots & \ddots & \vdots \\ A_{I1}\boldsymbol{B} & \cdots & A_{IJ}\boldsymbol{B} \end{bmatrix}. \tag{2}$$

Tensor matricization (tensor unfolding or flattening): Tensor matricization converts a tensor into a matrix. The mode-n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ is denoted by $\mathbf{X}_{(n)}$, $n = 1, 2, 3$, which arranges the mode-n tubes to be the columns of the result matrix. A tensor element (i_1, i_2, i_3) is mapped to matrix element (i_n, j) for $n = 1, 2, 3$, where

$$j = 1 + \sum_{k=1,\, k \neq n}^{3} (i_k - 1) J_k \text{ with } J_k = \prod_{m=1,\, m \neq n}^{k-1} I_m. \tag{3}$$

Tensor contraction: the tensor contractions of tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and matrices $\boldsymbol{A} \in \mathbb{R}^{R \times I}$, $\boldsymbol{B} \in \mathbb{R}^{R \times J}$ and $\boldsymbol{C} \in \mathbb{R}^{R \times K}$ are defined as follows:

$$\begin{aligned} \mathcal{Y} = \mathcal{X} \times_1 \boldsymbol{A}^\top &\leftrightarrow \boldsymbol{Y}_{(1)} = \boldsymbol{A}\boldsymbol{X}_{(1)}, \\ \mathcal{Y} = \mathcal{X} \times_2 \boldsymbol{B}^\top &\leftrightarrow \boldsymbol{Y}_{(2)} = \boldsymbol{B}\boldsymbol{X}_{(2)}, \\ \mathcal{Y} = \mathcal{X} \times_3 \boldsymbol{C}^\top &\leftrightarrow \boldsymbol{Y}_{(3)} = \boldsymbol{C}\boldsymbol{X}_{(3)}. \end{aligned} \tag{4}$$

Matricized tensor times Khatri-Rao product (MTTKRP): Given tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and matrices $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$, the mode-1, mode-2, mode-3 MTTKRP operations are defined as $\boldsymbol{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B})$, $\boldsymbol{X}_{(2)}(\boldsymbol{C} \odot \boldsymbol{A})$ and $\boldsymbol{X}_{(3)}(\boldsymbol{B} \odot \boldsymbol{A})$, respectively.

2.2 CP Tensor Decomposition Algorithm

The CP decomposition factorizes a tensor as a linear combination of rank-one tensors, e.g., given $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and rank R

$$\arg\min_{\widehat{\mathcal{X}}} \Vert\mathcal{X} - \widehat{\mathcal{X}}\Vert_F, \quad \text{where } \widehat{\mathcal{X}} = \sum_{r=1}^{R} \boldsymbol{A}_r \circ \boldsymbol{B}_r \circ \boldsymbol{C}_r, \tag{5}$$

where $\Vert\mathcal{X}\Vert_F = \sqrt{\sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} |\mathcal{X}_{ijk}|^2}$, $\boldsymbol{A} \in \mathbb{R}^{I \times R}$, $\boldsymbol{B} \in \mathbb{R}^{I \times R}$, and $\boldsymbol{C} \in \mathbb{R}^{I \times R}$ are factor matrices.

We adopt the most popular alternating least squares (ALS) algorithm [40] given in Alg. 1:

1) In line 2, the factor matrices are randomly initialized.
2) Lines 4-9 are the iterations to update the factor matrices, namely, perform a least-squares minimization to update a matrix by fixing the two factor matrices

The algorithm terminates when the approximation error converges below a threshold or it reaches a preset maximum number of iterations.

Algorithm 1. CP Tensor Decomposition [40]

```
1: **Input**: $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, rank $R$, maximum iterations $T$.
2: Randomly initialize $\boldsymbol{B}_0 \in \mathbb{R}^{J \times R}, \boldsymbol{C}_0 \in \mathbb{R}^{K \times R}$, and set $t = 1$,
3:   Unfold $\mathcal{X}$ to obtain $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I \times JK}, \boldsymbol{X}_{(2)} \in \mathbb{R}^{J \times IK}$, and $\boldsymbol{X}_{(3)} \in \mathbb{R}^{K \times IJ}$.
4: **while** $t < T$ and *NOT converged* **do**
5:   $\boldsymbol{A}_t \leftarrow \boldsymbol{X}_{(1)}(\boldsymbol{C}_{t-1} \odot \boldsymbol{B}_{t-1})(\boldsymbol{C}_{t-1}^\top \boldsymbol{C}_{t-1} * \boldsymbol{B}_{t-1}^\top \boldsymbol{B}_{t-1})^\dagger$,
6:   $\boldsymbol{B}_t \leftarrow \boldsymbol{X}_{(2)}(\boldsymbol{C}_{t-1} \odot \boldsymbol{A}_t)(\boldsymbol{C}_{t-1}^\top \boldsymbol{C}_{t-1} * \boldsymbol{A}_t^\top \boldsymbol{A}_t)^\dagger$,
7:   $\boldsymbol{C}_t \leftarrow \boldsymbol{X}_{(3)}(\boldsymbol{B}_t \odot \boldsymbol{A}_t)(\boldsymbol{B}_t^\top \boldsymbol{B}_t * \boldsymbol{A}_t^\top \boldsymbol{A}_t)^\dagger$,
8:   $t \leftarrow t + 1$,
9: **end while**
10: **Output**: $\boldsymbol{A}_{t-1}, \boldsymbol{B}_{t-1}, \boldsymbol{C}_{t-1}$.
```

Algorithm 2. Tucker Tensor Decomposition [41]

```
1: **Input**: $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and rank $R_1, R_2, R_3$.
2: $\boldsymbol{A} \leftarrow$ ($R_1$ leading left singular vectors) of $\boldsymbol{X}_{(1)}$,
3: $\boldsymbol{B} \leftarrow$ ($R_2$ leading left singular vectors) of $\boldsymbol{X}_{(2)}$,
4: $\boldsymbol{C} \leftarrow$ ($R_3$ leading left singular vectors) of $\boldsymbol{X}_{(3)}$,
5: Compute $\mathcal{G} = \mathcal{X} \times_1 \boldsymbol{A}^\top \times_2 \boldsymbol{B}^\top \times_3 \boldsymbol{C}^\top$.
6: **Output**: $\mathcal{G}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$.
```

2.3 Tucker Tensor Decomposition Algorithm

Tucker decomposition, as higher-order Principle Component Analysis (PCA), decomposes $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ as follows:

$$\arg\min_{\widehat{\mathcal{X}}} \Vert\mathcal{X} - \widehat{\mathcal{X}}\Vert_F, \quad \widehat{\mathcal{X}} = \mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}, \tag{6}$$

where $\boldsymbol{A} \in \mathbb{R}^{I \times R_1}, \boldsymbol{B} \in \mathbb{R}^{J \times R_2}$ and $\boldsymbol{C} \in \mathbb{R}^{K \times R_3}$ are principal components, $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$ is a core tensor, and the tensor contraction is defined in (4).

We adopt the higher-order singular value decomposition (HOSVD) algorithm [41] given in Alg. 2. There are three major steps:

1) In lines 2-4, obtain tensor matricizations $\boldsymbol{X}_{(1)}$, $\boldsymbol{X}_{(2)}$, $\boldsymbol{X}_{(3)}$.
2) In lines 2-4, compute the left singular vector of $\boldsymbol{X}_{(n)}$ and select R_n leading singular vectors.
3) In line 5, compute the tensor contraction of $\mathcal{X}$ and the transpose of $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$.

3 Tensor Learning Primitives on GPU Tensor Cores

In this section, we provide high-performance implementations of key tensor learning primitives on tensor cores. Our optimization mainly consists of three steps: partition the computation tasks, distribute tasks to tensor cores for computation and fetch the results.

3.1 GPU Tensor Cores

Tensor learning primitives can be efficiently computed in equivalent matrix forms on GPU tensor cores. GPU tensor cores perform mixed-precision matrix-multiply-and-accumulate (MMA) operation on 4×4 matrices. The mixed precision MMA component computes

$$\mathrm{fl}_{32}(\boldsymbol{D}) = \mathrm{fl}_{32}(\boldsymbol{C}) + \mathrm{fl}_{16}(\boldsymbol{A})\ \mathrm{fl}_{16}(\boldsymbol{B}), \tag{7}$$

where $\boldsymbol{A} \in \mathbb{R}^{4\times 4}$ and $\boldsymbol{B} \in \mathbb{R}^{4\times 4}$ are in FP16 format, $\boldsymbol{C} \in \mathbb{R}^{4\times 4}$ and $\boldsymbol{D} \in \mathbb{R}^{4\times 4}$ are in FP32 format.

Each two GPU tensor cores are wrapped together into a warp unit and exposed to CUDA programmers in warp-level APIs for scheduling, which performs an MMA operation on 16×16 matrices. Thus, to map the pipeline for tensor computations onto tensor cores, we consider block size of 16×16.

Assuming I, J, K are multiples of 16, given $\boldsymbol{A} \in \mathbb{R}^{I\times J}$ and $\boldsymbol{B} \in \mathbb{R}^{J\times K}$, for simplicity, we denote the matrix multiplication of two matrices as

$$\boldsymbol{C} = \mathrm{tCore}(\boldsymbol{A}, \boldsymbol{B}), \tag{8}$$

we compute $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B} \in \mathbb{R}^{I\times K}$ in three steps:

1) Reduce $\boldsymbol{A}$ and $\boldsymbol{B}$ to half precision, $\mathrm{fl}_{16}(\boldsymbol{A})$ and $\mathrm{fl}_{16}(\boldsymbol{B})$,
2) Partition $\mathrm{fl}_{16}(\boldsymbol{A})$ and $\mathrm{fl}_{16}(\boldsymbol{B})$ into small blocks, where we denoted each block as $\boldsymbol{A}^b_{ij} \in \mathbb{R}^{16\times 16}$ and $\boldsymbol{B}^b_{jk} \in \mathbb{R}^{16\times 16}$, $i \in [I/16], j \in [J/16], k \in [K/16]$.
3) Each tensor core loads a row of blocks, $\boldsymbol{A}(16(i-1)+1:16i,:)$, and a column of blocks, $\boldsymbol{B}(:,16(j-1)+1:16j)$, $i \in [I/16], j \in [J/16]$. Then, by employing (7), each tensor core computes a portion of result, $\boldsymbol{C}(16(i-1)+1:16i, 16(j-1)+1:16j)$.

Using a GPU, a matrix multiplication involves IK calls of a CUDA core to compute fused multiple-add (FMA) operation. A GPU tensor core computes matrix multiplications of two blocks with size 4×4. Using GPU tensor cores, a matrix multiplication only involves of $IJK/64$ calls of a tensor core.

In CUDA programming, we utilize the cuBLAS library APIs cuBlasGemmEx() to compute $\boldsymbol{C} = \boldsymbol{A}\boldsymbol{B}$, while cuBlasGemmEx() automatically employs load_matrix_sync() API in the wmma library to load a row or column of blocks onto a tensor core and wmma_sync() API to perform (7).

3.2 Basic Tensor Operations

3.2.1 Khatri-Rao Product

The Khatri-Rao product of $\boldsymbol{A} \in \mathbb{R}^{I\times K}$ and $\boldsymbol{B} \in \mathbb{R}^{J\times K}$ is $\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B} \in \mathbb{R}^{IJ\times K}$ where $\boldsymbol{C}_k = \boldsymbol{A}_k \otimes \boldsymbol{B}_k$ for $k \in [K]$. As shown in Fig. 2, assuming I, J are multiples of 16, we compute the Khatri-Rao product in the following steps,

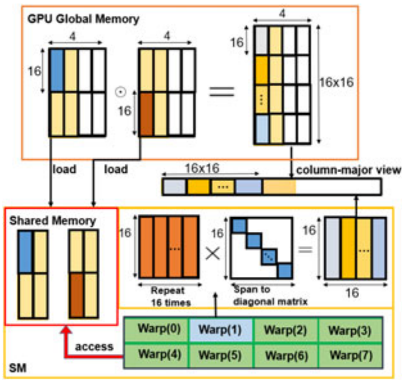


Fig. 2. Khatri-Rao product on GPU tensor cores.

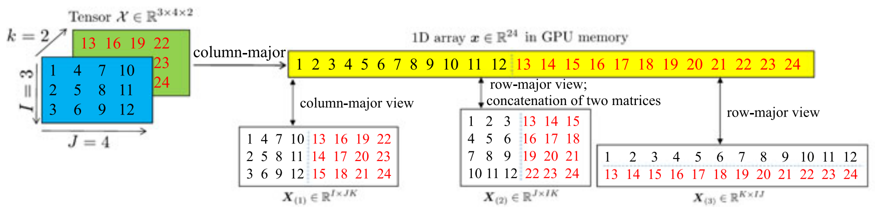
1) First, we partition $\boldsymbol{A}$ into $I/16 \times K$ blocks of size 16×1, where a block is denoted by $\boldsymbol{A}^b_{ik} \in \mathbb{R}^{16\times 1}$, $i \in [I/16]$, $k \in [K]$. Similarly, partition $\boldsymbol{B}$ into $J/16 \times K$ blocks of size 16×1, where $\boldsymbol{B}^b_{jk} \in \mathbb{R}^{16\times 1}$, $j \in [J/16]$, $k \in [K]$.
2) Next, $\boldsymbol{A}^b_{ik}$ is spanned to diagonal matrix $\widehat{\boldsymbol{A}}^b_{ik} \in \mathbb{R}^{16\times 16}$ and $\boldsymbol{B}^b_{jk}$ is replicated 16 times to $\widehat{\boldsymbol{B}}^b_{jk} \in \mathbb{R}^{16\times 16}$. Reduce $\widehat{\boldsymbol{A}}^b_{ik}$ and $\widehat{\boldsymbol{B}}^b_{jk}$ to half precision and load them from GPU global memory to shared memory.
3) Then, we use (8) to obtain $\boldsymbol{C}^b_{lk} = \mathrm{tCore}(\widehat{\boldsymbol{B}}^b_{jk}, \widehat{\boldsymbol{A}}^b_{ik}) \in \mathbb{R}^{16\times 16}$, where $l = (I/16) \times (j-1) + i$. In this process, each SM loads a portion of matrices and warps in each SM can access data from the shared memory. The data of $\boldsymbol{C}^b_{lk}$ corresponds to a portion of one column of $\boldsymbol{C}$, i.e., the $256 \times (l-1) + 1$ th element to the $256 \times l$ th element of $\boldsymbol{C}_k$. The portion of the result will be loaded from shared memory to global memory when all computation tasks in one thread block are completed.

In CUDA programming, we implement warp-level data preprocess kernels to partition and access blocks. Specifically, each kernel invokes a warp consisting of 32 threads to process 16×2 elements in parallel. Each 8 warps, consisting of $16 \times 2 \times 8$ threads, work together to fetch 16×1 elements from the GPU global memory and pad elements of blocks of size 16×16 in parallel. In each SM, there are 64 warps and a total of 8 post-processed blocks are stored in shared memory. Then, we leverage cublasGemmBatchedEx() API to batch the $(I/16) \times (J/16) \times K$ matrix multiplication tasks onto GPU tensor cores of each SM in turn. A total of $(I/16) \times (J/16) \times (K/4)$ SMs are invoked for the computations.

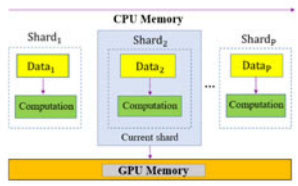
Such an implementation has three major advantages. First, column-major data storage provides continuous data access for two-column vectors, thus ensuring memory access efficiency. Second, each SM loads a portion of matrices and warps in each SM can access data from the shared memory, which reduces the bank conflicts. Third, by reorganizing the data, the computation of the Khatri-Rao product can be mapped onto GPU tensor cores for high performance.

3.2.2 Tensor (Kronecker) Product

The tensor (Kronecker) product of $\boldsymbol{A} \in \mathbb{R}^{I\times J}$ and $\boldsymbol{B} \in \mathbb{R}^{K\times L}$ is $\boldsymbol{C} = \mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{IK\times JL}$. We convert a tensor product into equivalent Khatri-Rao products as follows:


Tensor $\mathcal{X} \in \mathbb{R}^{3\times4\times2}$
$k=2$
$I=3$
$J=4$
column-major
1D array $x \in \mathbb{R}^{24}$ in GPU memory
column-major view
row-major view; concatenation of two matrices
row-major view
$X_{(1)} \in \mathbb{R}^{I\times JK}$
$X_{(2)} \in \mathbb{R}^{J\times IK}$
$X_{(3)} \in \mathbb{R}^{K\times IJ}$

(a) Exploiting the linear storage format in GPU memory to avoid explicit tensor matricization.



(b) Shard mode for tensor decompositions.

Fig. 3. (a) presents exploiting the linear storage format in GPU memory to avoid explicit tensor matricization. (b) presents the shard mode for tensor decompositions.

$$A \otimes B = [A_1 \otimes B, A_2 \otimes B \cdots A_J \otimes B],$$
$$A_j \otimes B = [A_j \odot B_1, A_j \odot B_2 \cdots A_j \odot B_L]. \quad (9)$$

Therefore, the Kronecker product of two matrices can be computed by computing the Khatri-Rao product of columns belonging to A and B respectively, which involves $(I/16) \times (J/16) \times L \times K$ matrix multiplications. Using the same strategies as the optimized Khatri-Rao product, we can batch the $(I/16) \times (J/16) \times L \times K$ matrix multiplication tasks onto GPU tensor cores.

3.3 Tensor Matricization

Tensor matricization, e.g. $X_{(1)}$, $X_{(2)}$ and $X_{(3)}$, is a fundamental operation in Alg. 1 and Alg. 2. In a conventional implementation, GPU allocates additional memory and performs explicit tensor matricization, which results in substantial memory and time cost.

We eliminate explicit tensor matricizations to save computation and memory footprint by exploiting the linear storage format in GPU memory. A third-order tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ is stored in a slice-by-slice and column-major layout as a 1D array x, where $\mathcal{X}_{ijk}$ is stored at $x[(k-1)IJ + (j-1)I + i]$. Fig. 3a illustrates an example tensor of $3 \times 4 \times 2$ and its three matricizations. First, $X_{(1)} \in \mathbb{R}^{3\times8}$ can be directly fetched by viewing $x \in \mathbb{R}^{24}$ as a column-major matrix of size 3×8. Second, viewing $x(1:12)$ and $x(13:24)$ as two column-major matrices of size 3×4 and using cublasSgemm$(\cdot)$ routine in the cuBLAS library, $X_{(2)}$ can be obtained by merging the transpose of two matrices. Third, the row-major storage of $X_{(3)}$ can be directly obtained from x. Therefore, with proper memory access, explicit tensor matricization of $\mathcal{X}$ can be avoided to save computation and GPU memory.

3.4 Tensor Contraction

Tensor contraction is a key operation in tensor learning algorithms. The Tucker decomposition in Alg. 2 takes mode-1, mode-2, and mode-3 tensor contractions. The conventional approach has three steps:

1) Tensor matricization: $\mathcal{X} \rightarrow X_{(1)}, X_{(2)}$ or $X_{(3)}$;
2) Executing matrix multiplication: $Y_{(1)} = AX_{(1)}$, $Y_{(2)} = BX_{(2)}$, or $Y_{(3)} = CX_{(3)}$;
3) Folding the result matrix into a tensor: $Y_{(1)} \rightarrow \mathcal{Y}$, $Y_{(2)} \rightarrow \mathcal{Y}$, or $Y_{(3)} \rightarrow \mathcal{Y}$.

Steps 1) and 2) are conversions between matrices and tensors, which involves expensive time consumption for large matrices and tensors. They can be avoided by using matricization-free memory access as in Section 3.3. In Step 2), we employ GPU tensor cores to compute the matrix multiplication as in Section 3.1.

3.5 Matricized Tensor Times Khatri-Rao Product

The MTTKRP operation can be computed as follows:

1) Matricize a tensor: $\mathcal{X} \rightarrow X_{(1)}, X_{(2)}$, or $X_{(3)}$;
2) Calculate Khatri-Rao product: $C \odot B, C \odot A, B \odot A$;
3) Execute matrix multiplications: $X_{(1)}(C \odot B)$, $X_{(2)}(C \odot A)$, $X_{(3)}(B \odot A)$.

We have proposed techniques to optimize the Khatri-Rao product and tensor matricization in Section 3.2.1 and Section 3.3. The tensor matricization results in a fat matrix whose number of columns is far larger than that of rows, and the Khatri-Rao product results in a tall matrix whose number of rows is far larger than that of columns. The third step is comprised of fat-tall matrix multiplications.

By respectively dividing the fat matrix and tall matrix into smaller block matrices, tensor cores can be used to compute the fat-tall matrix multiplication in a block manner. Therefore, MTTKRP operation is mapped onto tensor cores, which brings great acceleration.

4 High-Performance CP Tensor Decompositions on GPU Tensor Cores

In this section, we first propose the shard mode to achieve a high-performance schedule for large scale tensor computations. Then, we present the efficient third- or higher-order CP tensor decomposition and apply the shard mode to achieve large-scale CP tensor decomposition. Besides, we demonstrate an interesting application, gene analysis using CP tensor decomposition.

4.1 Shard Mode

Due to the limitation of GPU memory, tensor learning algorithms are not scalable on GPUs. Therefore, we use a shard mode strategy to schedule tensor computations and make tensor learning algorithms more scalable.

As shown in Fig. 3b, we use a shard mode to compute $X_{(1)}(C \odot B)$ in line 5 of Alg. 1 as follows,

- First, transfer part matrix of origin data $X_{(1)}$, from CPU memory to GPU memory, to construct a shard.
- Second, in current shard, multiply the counterpart of $(C \odot B)$ to obtain a small intermediate result.
- Last, assemble a series of intermediate results to obtain $X_{(1)}(C \odot B)$.

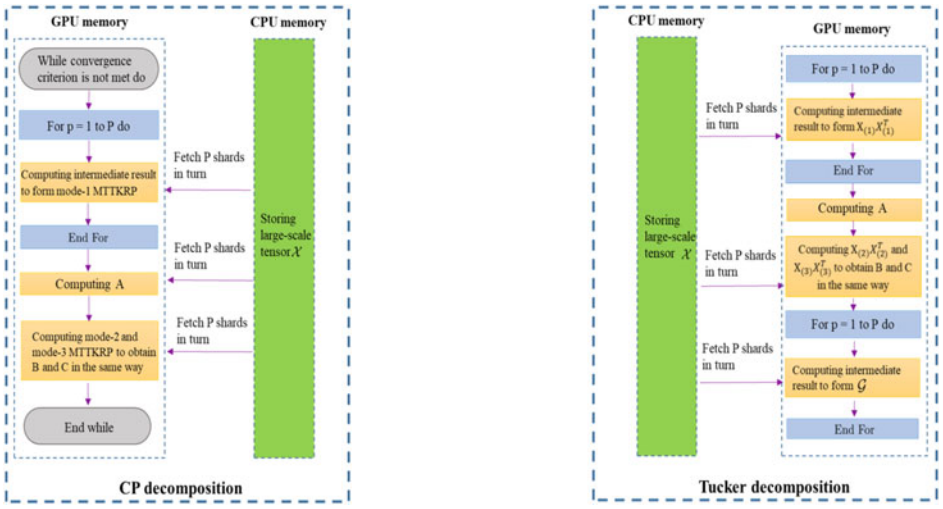

GPU memory
CPU memory
While convergence criterion is not met do
For p = 1 to P do
Computing intermediate result to form mode-1 MTTKRP
End For
Computing A
Computing mode-2 and mode-3 MTTKRP to obtain B and C in the same way
End while
Fetch P shards in turn
Fetch P shards in turn
Fetch P shards in turn
Storing large-scale tensor $\mathcal{X}$
CP decomposition
CPU memory
GPU memory
For p = 1 to P do
Computing intermediate result to form $X_{(1)}X_{(1)}^T$
End For
Computing A
Computing $X_{(2)}X_{(2)}^T$ and $X_{(3)}X_{(3)}^T$ to obtain B and C in the same way
For p = 1 to P do
Computing intermediate result to form $\mathcal{G}$
End For
Storing large-scale tensor $\mathcal{X}$
Fetch P shards in turn
Fetch P shards in turn
Fetch P shards in turn
Tucker decomposition

Fig. 4. Our strategy for high-order CP and Tucker tensor decompositions.

The shard mode exploits the parallelism of tensor computations to improve the utilization of GPU. The tensor learning algorithms can be carefully designed and implemented using a shard mode strategy for large-scale tensors, exceeding the GPU capacity.

4.2 CP Tensor Decomposition

In CP tensor decomposition, singular value decomposition (SVD) is utilized to obtain Matrix Moore-Penrose pseudo inverse (10), which involves much memory consumption and computation cost

$$A \leftarrow X_{(1)}(C \odot B)(C^\top C * B^\top B)^\dagger, \tag{10}$$

where $X_{(1)} \in \mathbb{R}^{I\times JK}, A \in \mathbb{R}^{I\times R}, B \in \mathbb{R}^{J\times R}$, and $C \in \mathbb{R}^{K\times R}$. Because SVD cannot be computed in a block manner, it becomes the bottleneck of applying the shard mode to CP tensor decomposition. Instead, we choose Lower-Upper (LU) decomposition to accelerate the computation. First, we obtain an intermediate matrix M as

$$M = (X_{(1)}(C \odot B)) = A(C^\top C * B^\top B), \tag{11}$$

which corresponds to multiplying $C^\top C * B^\top B$ on both sides of (10). Second, we perform matrix LU factorization [30] of $(C^\top C * B^\top B)$ and solve the following least square minimization

$$M = A(LU) \Rightarrow \begin{cases} M & = YU, \\ Y & = AL. \end{cases} \tag{12}$$

Now computing A in (10) has been replaced by solving a least squares problem. In (12), we obtain an intermediate Y from M and U, and then derive A from Y and L. In this way, we avoid the calculation of matrix pseudo inverse, saving computing time and memory consumption.

Then, we apply shard mode to CP tensor decomposition. Our basic idea is to exploit large CPU memory to store the input tensor $\mathcal{X}$ and split $\mathcal{X}$ into P shards along the third dimension. Our shard strategy on CP tensor decomposition (left part of Fig. 4) is as follows:

1) Fetching one shard from CPU memory to GPU memory and utilizing the separability of computation to compute the intermediate result of mode-1 MTTKRP $(X_{(1)}C \odot B)$.
2) Transferring and calculating the next shard in turn and the intermediate results are added up to obtain the final result of mode-1 MTTKRP.
3) Because of the memory consumption of $(C^\top C * B^\top B)$ is low, we directly allocate memory on GPU to calculate. Then, we use the least square method to solve A.
4) Computing mode-2 and mode-3 MTTKRP to obtain B and C in the same way.

4.2.1 *High-Order CP Tensor Decomposition*

To extend CP tensor decomposition to a high-order tensor, we take the fourth-order case as an example. The key step is

$$A \leftarrow X_{(1)}(D \odot C \odot B)((D^\top D) * (C^\top C) * (B^\top B))^\dagger, \tag{13}$$

where $X_{(1)} \in \mathbb{R}^{I\times JKL}, A \in \mathbb{R}^{I\times R}, B \in \mathbb{R}^{J\times R}, C \in \mathbb{R}^{K\times R}, D \in \mathbb{R}^{L\times R}$. (13) is the fourth-order counterpart of (10) with the following key operations:

(13) is the fourth-order expansion of line 5 of Algorithm 1. So, our previous technique in third-order CP tensor decomposition is also applicable to the fourth-order tensor. Thus, the steps of fourth-order Tucker tensor decomposition are as follows:

1) Utilizing shard mode to compute mode-1 MTTKRP $(X_{(1)}D \odot C \odot B)$.
2) Allocating memory on GPU to calculate $(D^\top D * C^\top C * B^\top B)$ and using the least square method to solve A.
4) Computing mode-2, mode-3, mode-4 MTTKRP to obtain B, C and D in the same way.

4.3 Gene Analysis Using CP Tensor Decomposition

In gene analysis, the data is usually modeled as a third-order tensor with 'individual-tissue-gene'. By employing CP tensor decomposition algorithm to factorize a third-order tensor into a series of matrices, it helps analyze the gene expression in multiple tissues and uncover biological relationships. However, due to the large volume of gene data, the expensive computation complexity of tensor operations usually hinders the application of CP tensor decomposition in gene analysis in high performance. Besides, the limited memory is the bottleneck to factorize large-scale gene data. In this section, we demonstrate our strategy for gene analysis using CP tensor decomposition in high performance.

For a given gene data $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ with rank R, we employ high-performance CP tensor decomposition for gene analysis as follows.

1) First, we pad $\mathcal{X}$ with zero to the size of $I' \times J' \times K'$, where I', J', K' are multiples of 16, to fully exploited the compting power of GPU tensor cores.
2) Then, we divide all elements of $\mathcal{X}$ by the maximum, and reduce $\mathcal{X}$ to half precision.
3) Next, we employ the optimized third-order CP tensor decomposition with shard mode on GPU tensor cores in Section 4 to factorize $\mathcal{X}$ with rank R.

5 High-Performance Tucker Tensor Decompositions on GPU Tensor Cores

In this section, we present the efficient third- and higher-order Tucker tensor decomposition and apply shard mode to achieve large scale Tucker tensor decomposition. Furthermore, we demonstrate a novel application, Tucker tensor layer for deep neural networks.

5.1 Tucker Tensor Decomposition

In Tucker decomposition, SVD is utilized to obtain the singular vectors in line 2-4 of Alg. 2. Likewise, we consider the optimization of SVD

$$\boldsymbol{A} \leftarrow (R_1 \text{ leading singular vectors}) \text{ of } \boldsymbol{X}_{(1)}. \tag{14}$$

For computing SVD on GPUs, we encounter an intermediate data explosion problem. For instance, the SVD decomposition of $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I\times JK}$ will produce a matrix of size $JK \times JK$, which may cause memory explosion and the result SVD decomposition is also time-consuming. Therefore, we provide methods to reduce the number of computations and memory consumption.

We multiply $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I\times JK}$ with its transpose and obtain $\boldsymbol{X}_{\text{temp}} \in \mathbb{R}^{I\times I}$, which is much smaller than $\boldsymbol{X}_{(1)}$. Next, we compute the eigenvalue decomposition of $\boldsymbol{X}_{\text{temp}}$ and extract R_1 leading eigen vectors

$$\begin{aligned} &\boldsymbol{X}_{\text{temp}} = \boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^{\top}, \\ &\quad \boldsymbol{A} \leftarrow (R_1 \text{ leading eigen vectors}) \text{ of } \boldsymbol{X}_{\text{temp}}. \end{aligned} \tag{15}$$

The space complexity of the previous computation (14) is $O(J^2K^2)$, while the space complexity of (15) is $O(I^2)$. Although (15) brings in extra operations of matrix transposition and matrix multiplication, the time and memory consumption overhead are relatively small.

In CUDA, for multiplication between $\boldsymbol{X}_{(1)}$ and $\boldsymbol{X}_{(1)}^{\top}$, we use interval access of $\boldsymbol{X}_{(1)}$ to implicitly obtain matrix transpose $\boldsymbol{X}_{(1)}^{\top}$, avoiding memory allocation for $\boldsymbol{X}_{(1)}^{\top}$. Through (15), we only need to perform eigenvalue decomposition on matrix $(\boldsymbol{X}_{(1)} \times \boldsymbol{X}_{(1)}^{\top}) \in \mathbb{R}^{I\times I}$, thus the amount of computations is reduced and the memory is saved.

Then, we apply shard mode to Tucker decomposition. In lines 2-5 of Alg. 2, we only need the raw data to carry on relative computation. Therefore, our shard strategy on Tucker tensor decomposition (right part of Fig. 4) is as follows:

1) Fetching one shard from CPU memory to GPU memory and utilizing the separability of computation to compute the intermediate result of matrix multiplication $(\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^{\top})$.
2) Transferring and calculating the next shard in turn and the intermediate results are added up to obtain the final result of $(\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^{\top})$.
3) Computing $(\boldsymbol{X}_{(2)}\boldsymbol{X}_{(2)}^{\top})$ and $(\boldsymbol{X}_{(3)}\boldsymbol{X}_{(3)}^{\top})$ to obtain B and C in the same way.
4) Fetching one shard from CPU memory to GPU memory and utilizing the separability of computation to compute the intermediate result of tensor contraction $(\mathcal{X} \times_1 \boldsymbol{A}^{\top} \times_2 \boldsymbol{B}^{\top} \times_3 \boldsymbol{C}^{\top})$.
5) Transferring and calculating the next shard in turn and the intermediate results are added up to obtain the final result of tensor contraction.

5.1.1 *High-Order Tucker Tensor Decomposition*

To extend a Tucker tensor decomposition to the high-order tensor case, we take fourth-order case as an example. For fourth-order Tucker tensor decomposition, the key step is

$$\mathcal{G} \approx \mathcal{X} \times_1 \boldsymbol{A}^{\top} \times_2 \boldsymbol{B}^{\top} \times_3 \boldsymbol{C}^{\top} \times_4 \boldsymbol{D}^{\top}. \tag{16}$$

where $\mathcal{X} \in \mathbb{R}^{I\times J\times K\times L}, \boldsymbol{A} \in \mathbb{R}^{I\times R_1}, \boldsymbol{B} \in \mathbb{R}^{J\times R_2}, \boldsymbol{C} \in \mathbb{R}^{K\times R_3}, \boldsymbol{D} \in \mathbb{R}^{L\times R_4}$. (16) is the fourth-order expansion of line 5 of Alg. 2. So, our previous technique in third-order Tucker tensor decomposition is also applicable to the fourth-order tensor. Furthermore, we can adopt the strategy of (15) which replaces matrix singular value decomposition with matrix eigenvalue decomposition to reduce memory consumption and computation. Thus, the steps of fourth-order Tucker tensor decomposition are as follows:

1) utilizing a shard mode to compute matrix multiplication $(\boldsymbol{X}_{(1)}\boldsymbol{X}_{(1)}^{\top})$.
2) Computing $(\boldsymbol{X}_{(2)}\boldsymbol{X}_{(2)}^{\top})$, $(\boldsymbol{X}_{(3)}\boldsymbol{X}_{(3)}^{\top})$ and $(\boldsymbol{X}_{(4)}\boldsymbol{X}_{(4)}^{\top})$ to obtain $\boldsymbol{B}, \boldsymbol{C}$ and $\boldsymbol{D}$ in the same way.
3) utilizing a shard mode to compute tensor contraction $(\mathcal{X} \times_1 \boldsymbol{A}^{\top} \times_2 \boldsymbol{B}^{\top} \times_3 \boldsymbol{C}^{\top} \times_4 \boldsymbol{D}^{\top})$.

5.2 Tucker Tensor Layer for Deep Neural Network

Tensor layer for deep neural network can be used to compress the network while maintaining the accuracy of approximation. For example., as shown in Fig. 5, one can use a Tucker tensor layer to replace the conventional fully connected layer and convolutional layer to reduce the number of parameters. We demonstrate the application of the Tucker tensor layer for a fully connected layer in this section. However, due to the computation complexity of tensor computations, the time consumption of training the Tucker tensor layer is large. Natural gradient can be estimated without backpropagation and used to update

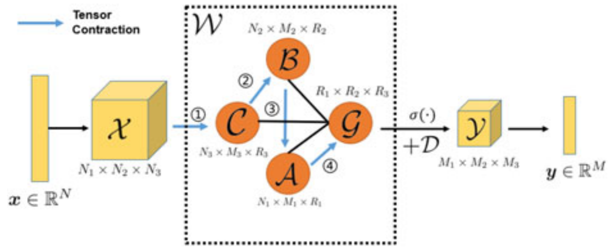

Tensor Contraction
𝒲
$N_2 \times M_2 \times R_2$
ℬ
②
③
$R_1 \times R_2 \times R_3$
𝒞
𝒢
①
$N_3 \times M_3 \times R_3$
𝒜
④
$N_1 \times M_1 \times R_1$
𝒳
$N_1 \times N_2 \times N_3$
$x \in \mathbb{R}^N$
$\sigma(\cdot)$
$+\mathcal{D}$
𝒴
$M_1 \times M_2 \times M_3$
$y \in \mathbb{R}^M$

Fig. 5. The forward pass of a Tucker tensor layer. The input $\boldsymbol{x} \in \mathbb{R}^N$ is first reorganized into a 3-order tensor $\mathcal{X}$. Then, $\mathcal{X}$ is contracted with weight tensors, $\{\mathcal{C}, \mathcal{B}, \mathcal{G}, \mathcal{A}\}$, activated by $\sigma(\cdot)$, and added by bias tensor $\mathcal{D}$ to obtain a result tensor $\mathcal{Y}$. The final output $\boldsymbol{y}$ is a vectorization of $\mathcal{Y}$.

the neural networks. Besides, the estimation process of natural gradient has a high degree of parallelism. Therefore, we adopt the natural gradient to train Tucker tensor layer for deep neural networks using GPU tensor cores.

For a given input $\boldsymbol{x} \in \mathbb{R}^N$, a conventional fully connected layer applies a linear transformation as follows:

$$\boldsymbol{y} = \sigma(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{d}), \tag{17}$$

where we denote the activation function as $\sigma(\cdot)$, the weight matrix as $\boldsymbol{W} \in \mathbb{R}^{M \times N}$, the bias vector as $\boldsymbol{d} \in \mathbb{R}^M$ and the output of the layer as $\boldsymbol{y} \in \mathbb{R}^M$.

Based on Section 2.3, we construct the corresponding Tucker format of the fully connected neural network and (17) is expressed as

$$\mathcal{Y} = \sigma((\mathcal{G} \times_1 \boldsymbol{A} \times_2 \boldsymbol{B} \times_3 \boldsymbol{C}) \cdot \mathcal{X} + \mathcal{D}), \tag{18}$$

where we have $\mathcal{G} \in \mathbb{R}^{R_1 \times R_2 \times R_3}$, $\boldsymbol{A} \in \mathbb{R}^{N_1 M_1 \times R_1}, \boldsymbol{B} \in \mathbb{R}^{N_2 M_2 \times R_2}, \boldsymbol{C} \in \mathbb{R}^{N_3 M_3 \times R_3}$, $M = M_1 M_2 M_3, N = N_1 N_2 N_3$ and R_1, R_2, R_3 are the ranks in Algorithm 2. Compared with conventional fully connected layer, the number of parameters in Tucker tensor layer is reduced from $MN + M$ to $M_1 N_1 \times R_1 + M_2 N_2 \times R_2 + M_3 N_3 \times R_3 + M$.

To make full use of the optimized tensor primitives, we utilize natural gradients to update the Tucker tensor layer parameters $\boldsymbol{\theta}$, where $\boldsymbol{\theta} = (\mathcal{G}, \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$. The natural gradient can be estimated as follows:

$$\nabla \boldsymbol{\theta} \approx \frac{1}{n\sigma} \sum_{i=1}^{n} L(\boldsymbol{\theta} + \sigma \epsilon_i) \boldsymbol{\epsilon_i}. \tag{19}$$

where $L(\cdot)$ is the loss function, $\epsilon_i \sim \mathcal{N}(0, \boldsymbol{I})$ is the noise, σ is the standard deviation, $i = 1, 2, \ldots, n$. The update process is described as:

1) Generate noise ϵ_i that has the same size with θ and add $\sigma\epsilon_i$ to $\boldsymbol{\theta}$.
2) Perform n forward passes and obtain the n loss values in parallel.
3) Estimate the natural gradient $\nabla\boldsymbol{\theta}$ using (19).
4) Update $\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha \nabla \boldsymbol{\theta}$, where α is the learning rate.

6 Performance Evaluations

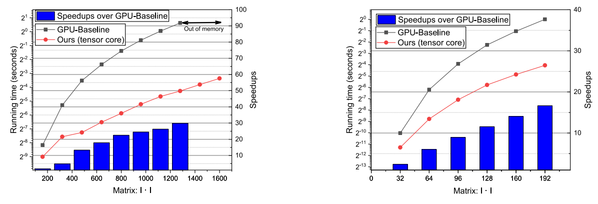
6.1 Experiment Settings

We run all experiments on a DGX-2 server with 8 NVIDIA A100 GPUs and dual Intel Xeon E5-2640 V4 CPUs. Each A100 GPU has 40 GB memory, 6,912 CUDA cores and 432 tensor cores. Each CPU has 10 cores running at 2.4GHz. The operating system of the server is 64-bit Ubuntu 20.04.

We randomly generate tensors of varying sizes to evaluate the performance of tensor decompositions. For an $I \times I \times I$ input tensor, we set the rank $R = R_1 = R_2 = R_3 = 0.1I$. For the key tensor operations, i.e., Khatri-Rao product, tensor product, tensor contraction, and MTTKRP, we report the running time, which does not include the data transfer time between CPUs and GPUs. Because these tensor operations are normally called during the computation process of the tensor decomposition algorithms and the data is already in the GPU device memory. For the CP and Tucker decomposition algorithms, we measure the entire running time, including data transfer time between CPUs and GPUs. To determine the iteration number for the CP and Tucker tensor decomposition algorithms, we make different implementations to have similar errors. Each experiment is repeated ten times and we report the average result.

We use running time and relative squared error as performance metrics:

- *Running time*: We vary the tensor size and test the running time of different CPU/GPU implementations. The speedup of our GPU implementation over a reference GPU implementation is calculated as: (running time of a reference GPU implementation) / (running time of our GPU implementation).
- *Relative squared error*: For CP and Tucker decompositions, we report the relative squared error (RSE) that is calculate as $\|\widehat{\mathcal{T}} - \mathcal{T}\|_F / \|\mathcal{T}\|_F$, where $\widehat{\mathcal{T}}$ is recovered from the factors $\mathcal{G}$, $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$.
- *Compression Ratio*: We denote the compression ratio of data as the ratio between the uncompressed size and the compressed size, i.e., Compression ratio = Uncompressed Size / Compressed Size.



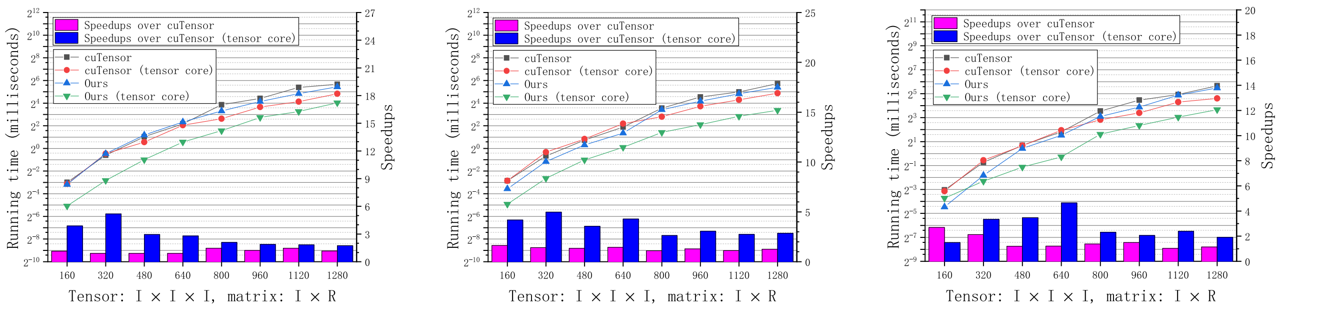
(a) Khatri-Rao product. (b) Tensor product.

Fig. 6. Running time and speedups of Khatri-Rao product and tensor product on GPU.

6.2 Tensor Learning Primitives

In this subsection, our experiments are running on an A100 GPU. Running time is shown on a log scale. We compare the performance of basic tensor operations in Section 3 between GPU baseline which is not optimized, cuTensor by NVIDIA [42] and our GPU implementation which utilize the proposed optimization techniques in Section 3. We compare the following GPU implementations:

- *GPU-Baseline*: We implement the tensor learning primitives using cuBLAS and cuSOLVER libraries.
- *cuTensor* [42]: cuTensor is a high-performance CUDA library for tensor primitives.



(a) mode-1 tensor contraction. (b) mode-2 tensor contraction. (c) mode-3 tensor contraction.

Fig. 7. Running time and speedups of tensor contraction in three modes on GPU, respectively.

- *Ours (CUDA core)*: We employ the proposed optimization techniques to tensor learning primitives using GPU CUDA cores
- *Ours (tensor core)*: We employ the proposed optimization techniques to tensor learning primitives using GPU tensor cores.

Fig. 6a shows the running time and speedups of the Khatri-Rao product $\boldsymbol{C} = \boldsymbol{A} \odot \boldsymbol{B}$. The input two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ have the same size $I \times I$ and the output matrix $\boldsymbol{C}$ has size $I^2 \times I$. Compared with the GPU baseline implementation, our GPU implementation achieves an average of $4.57\times$ speedup. Exploiting the continuous memory access and batched outer product techniques, our GPU implementation significantly outperforms the GPU baseline implementation.

Fig. 6b shows the running time and speedups of the tensor product $\boldsymbol{C} = \boldsymbol{A} \otimes \boldsymbol{B}$. The input two matrices $\boldsymbol{A}$ and $\boldsymbol{B}$ have the same size $I \times I$ and the output matrix $\boldsymbol{C}$ has size $I^2 \times I^2$. Since the result matrix has a much large size, we are able to test the input matrices for I varying from 20 to 180. With the optimization techniques to convert a tensor product into multiple Khatri-Rao products in Section 3.2.2, the optimized GPU implementation exhibits much higher parallelism and achieves an average of $1.68\times$ speedup versus the GPU baseline implementation.

Fig. 7 shows the running time and speedups of mode-1 tensor contraction, mode-2 tensor contraction, and mode-3 tensor contraction with varying tensor and matrix sizes, respectively. Our input is a third-order tensor $I \times I \times I$ and a matrix $I \times R$, which R is set to $0.1 \times I$ and the corresponding outputs are three tensors of size $R \times I \times I$, $I \times R \times I$ and $I \times I \times R$, respectively.

In Fig. 7a, with the optimization techniques in Section 3.4 to avoid tensor matricization, the optimized GPU implementations using CUDA cores and tensor cores achieve up to $1.47\times$ and $5.18\times$ speedups versus the NVIDIA cuTensor library [42] using CUDA cores and tensor cores, respectively.

In Fig. 7b with the optimization techniques in Section 3.4 to avoid tensor matricization and employ batched matrix multiplications, the optimized GPU implementations using CUDA cores and tensor cores achieve up to $1.63\times$ and $4.98\times$ speedups versus the NVIDIA cuTensor library [42] using CUDA cores and tensor cores, respectively.

In Fig. 7c, with the optimization techniques in Section 3.4, the optimized GPU implementations using CUDA cores and tensor cores achieve up to $2.69\times$ and $4.66\times$ speedups versus the NVIDIA cuTensor library [42] using CUDA cores and tensor cores, respectively.

Fig. 8 shows the running time and speedups of the mode-1 MTTKRP, mode-2 MTTKRP and mode-3 MTTKRP with varying tensor and matrix sizes, respectively. Our input is a third-order tensor $I \times I \times I$ and two matrices of the same size $I \times R$, which R is set to $0.1 \times I$ and the corresponding outputs are three tensors of the same size $I \times R$. As described in Section 3.5, by utilizing the optimized tensor matricization and Khatri-Rao operations, we improve the performance of the MTTKRP operation.

In Fig. 8a, compared with GPU Basline, our mode-1 MTTKRP using CUDA cores and tensor cores achieves up to $1.84\times$ and $5.34\times$ speedups, respectively.

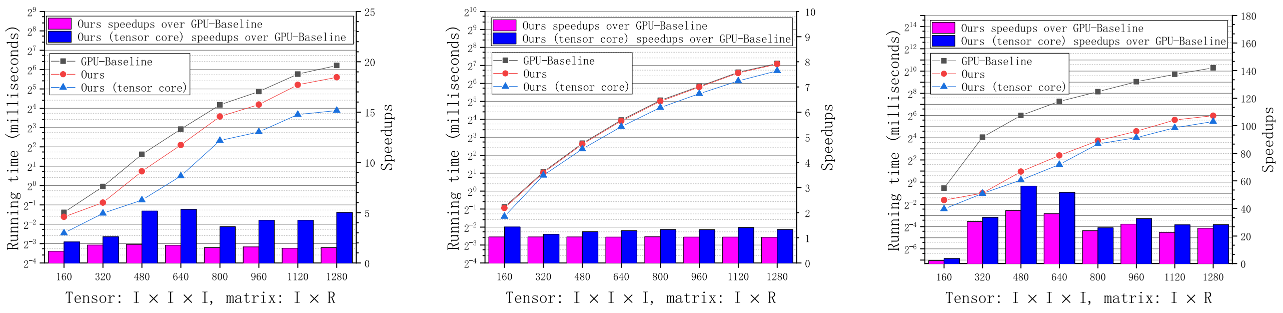
In Fig. 8b, our mode-2 MTTKRP using CUDA cores and tensor cores achieves up to $1.04\times$ and $1.43\times$ speedups, respectively. The reason for the low performance improvement than mode-1 MTTKRP is that mode-2 MTTKPR cannot fully avoid the tensor matricization operation.

In Fig. 8c, our mode-3 MTTKRP using CUDA cores and tensor cores achieves up to $38.55\times$ and $56.21\times$ speedups, respectively. The reason is that the original tensor matricization in the operation of mode-3 MTTKPR breaks the continuity of data access seriously. Eliminating such a tensor matricization operation leads to this high-performance improvement.

6.3 Tensor Decompositions

In this subsection, we compare the performance of third-order CP and Tucker tensor decompositions between GPU baseline which is not optimized, TensorLab-GPU [28], TensorD-GPU [29] and our GPU implementation which utilize the optimization techniques in Section 4. We compare the following GPU implementations:

- *cuTensor* [42]: We utilize cuTensor library to implement CP and Tucker tensor decomposition.
- *TensorLab-GPU* [28]: TensorLab-GPU provides high-performance CP and Tucker tensor decomposition algorithms on GPU.
- *Ours (CUDA core)*: We implement the optimized CP and Tucker tensor decomposition algorithms using GPU CUDA cores.
- *Ours (tensor core)*: We implement the optimized CP and Tucker tensor decomposition algorithms using GPU tensor cores.



(a) mode-1 MTTKRP. (b) mode-2 MTTKRP. (c) mode-3 MTTKRP.

Fig. 8. Running time and speedups of MTTKRP in three modes on GPU, respectively.

Fig. 9 shows the running time and speedups of the CP and Tucker tensor decompositions with varying tensor sizes, respectively. Five GPU-based implementations are compared: our GPU implementation (default with cuda core), our GPU implementation with tensor cores, TensorLab-GPU [28], TensorD-GPU [29], and GPU baseline. Our GPU with tensor cores implementation achieved the best performance among all implementations, while the TensorD-GPU [28] achieved the worst performance. Due to limited GPU memory capacity of a single A100 GPU, the maximum tensor size can be tested by our GPU implementation and the TensorLab-GPU [28] is about $1,280 \times 1,280 \times 1,280$. The tensor size supported by the GPU baseline is smaller due to the higher memory cost. The performance of the TensorD-GPU [29] is the least. In consideration of the performance and scale limitations of TensorD-GPU, We do not perform performance analysis with it.

In Fig. 9a, on varying tensor sizes from $160 \times 160 \times 160$ to $1,280 \times 1,280 \times 1,280$, our GPU implementation with tensor cores achieved average of $16.32\times$ speedup versus the TensorLab-GPU [28] and maximum $1.81\times$ speedup versus our GPU implementation. On varying tensor sizes from $160 \times 160 \times 160$ to $960 \times 960 \times 960$, our GPU implementation with tensor cores achieved average of $4.34\times$ speedup versus the GPU baseline.

In Fig. 9c, on varying tensor sizes from $160 \times 160 \times 160$ to $1,280 \times 1,280 \times 1,280$, our GPU implementation with tensor cores achieved average of $6.09\times$ speedup versus the TensorLab-GPU [28] and maximum $2.19\times$ speedup versus our GPU implementation. With varying tensor sizes from $160 \times 160 \times 160$ to $960 \times 960 \times 960$, our GPU implementation with tensor cores achieved average of $14.44\times$ speedup versus the GPU baseline. The amount of multiplication in the Tucker decomposition is higher than that in CP decomposition which reflects in the performance difference between our GPU implementation with tensor cores and our GPU implementation.

Figs. 9b and 9d shows the running time and speedups of fourth-order CP and Tucker decomposition running on the A100 GPU and two CPUs, respectively. We compare our implementation with TensorLab-GPU [28]. Since the dimensions of the tensor have been expanded, the maximum tensor size can be tested on the GPU is $160 \times 160 \times 160 \times 160$. We tested CP decompostion on tensors from $20 \times 20 \times 20 \times 20$ to $160 \times 160 \times 160 \times 160$. In Fig. 9b, the optimized GPU CP decomposition algorithm achieves up to $30.65\times$ speedup over TensorLab-GPU [28]. In Fig. 9d, the optimized GPU Tucker decomposition algorithm achieves an average of $5.41\times$ speedup over TensorLab-GPU [28].

6.4 Gene Analysis Using CP Decomposition

Following [18], we use CP tensor decomposition to factorize GTEx v6 [38], a popular large scale gene expression data, to demonstrate our application. We compare the following GPU implementations: 1) *TensorLy* [30]: Tensorly implements CP tensor decomposition algorithm on GPU. 2) *Ours (tensor core)*: Our implementations employs the optimized CP tensor decomposition algorithm on GPU tensor cores.

The gene data has size $40490 \times 449 \times 44$. In Fig. 10a, on varying the rank of tensor from 100 to 1000, our GPU implementation using tensor cores achieved average of $5.8\times$ speedup over TensorLy. In Fig. 10b, our GPU implementation achieved nearly the same performance on the relative error with TensorLy.

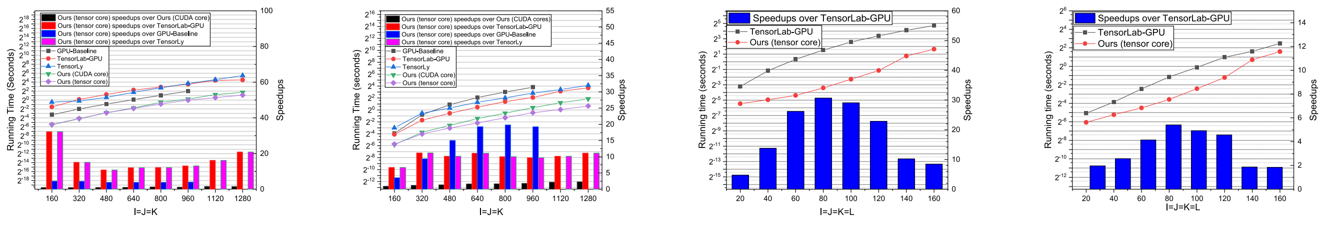
6.5 Tensor Layer for Deep Neural Networks

In this section, we utilize the MNIST dataset to evaluate the performance of our Tucker tensor layer of deep neural network. We compare the three neural networks GPU implementations:

- *Conventional layer*: We build a fully connected neural network in PyTorch [43] and employ natural gradients to train the network on GPUs.
- *Tensor layer*: We utilize the Tucker decomposition to compress the conventional layer and employ natural gradients to train the network on GPUs.
- *Tensor layer (ours)*: We utilize the Tucker decomposition to compress the conventional layer and employ natural gradients to train the network on GPUs, where the forward pass are fully optimized using GPU tensor cores, as given in previous sections.

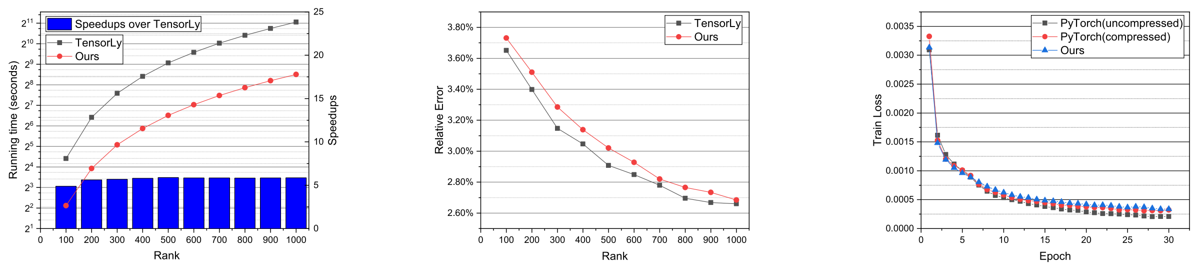
In our experiment, the neural network has three hidden layers of which the number of neurons are 512, 512 and 256 respectively. To fully exploit tensor cores, we pad the images of MNIST from the size of 756 to 1024. We use cross entropy as the loss function and ReLU as the activation function. We employ the Tucker tensor decomposition to compress the weight matrices with rank 128, 64, 64 respectively. The neural networks are trained for 30 epochs, and the curve of training loss can be shown in Fig. 10c. It can be shown that all of the models have been converged.

We evaluate the compression ratio, accuracy of classification and training time in our experiments. The results are



(a) 3D CP decomposition. (b) 3D Tucker decomposition. (c) 4D CP decomposition. (d) 4D Tucker decomposition.

Fig. 9. Running time and speedups of CP and Tucker tensor decomposition on GPU.



(a) Running time and speedups. (b) Relative error. (c) Training process.

Fig. 10. Applications of tensor learning algorithms. (a) and (b) present the gene analysis using CP tensor decomposition. (c) presents the training process of tucker tensor layer on MNIST dataset.

summarized in Table 2. The number of parameters in uncompressed neural network is 921,354 while that in compressed neural network is only 315,146, which reduced 65.80%. The neural network with conventional layer achieves an accuracy of 98.3% with running time 41,153 seconds. The neural network with tensor layer in CUDA version achieves an accuracy of 98.1% at a cost of 0.2%, and has a running time 54,967 which is $1.33\times$ slower versus conventional layer. It can be shown that the complexity of tensor computations hinders the efficiency of training tensor layer. And the neural network with tensor layer in our implementation achieves an accuracy of 97.9% at a cost of 0.4% versus conventional layer, but has a training time 9,203 seconds, which achieves $4.47\times$ speedups versus conventional layer.

7 Related Works

In this section, we discuss related works on efficient tensor learning computations and applications.

7.1 High-Performance Computing Architecture

Many high-performance computing architectures have been proposed to accelerate the data- and computation-intensive programs. Open Multi-Processing (OpenMP) [44] supports multiprocessing programming using shared memory which has been generally used by various operating systems and applications. Xiao [45] proposed a plasticity-on-chip design that exploits the self-similarity of high-level programs to offer efficient data communications. Ma [10] proposed a distributed graph-theoretic framework for automatic Parallelization in Multi-core systems. The above works mainly employ the multi-core processor to perform parallel computing. On the other hand, Graphics Processing Units (GPUs) are electronic circuits designed by NVIDIA to perform efficient graph computation in natural parallel. GPU tensor cores [31] are specialized hardware for accelerating tensor operations. In our work, we focus on the design of high-performance of tensor learning primitives using GPU tensor cores.

7.2 GPU Accelerated Tensor Computations

There are many works focused on optimizing tensor computations on GPU. Li et al. [46] proposed a parallel algorithm and implementation of sparse TTM on GPUs via parallelizing the algorithm on fibers. [47] design a high-performance GEMM-like Tensor-Tensor Multiplication. The sparse MTTKRP algorithm are optimized on GPUs in ParTI [48]. The performance of their algorithm is limited by the overhead of atomic operations. Shi et al. [49] implemented TTM using NVIDIA cuBLAS library on a single node. While this library provides a convenient solution for matrix multiplication on small matrices, their work does not offer any insight into reducing communication overhead.

GPU tensor cores are specialized hardware for accelerating tensor operations, and the rise of GPU tensor cores provides more opportunities to fully utilize the parallelism of tensor operations and achieve high-performance computation. Zachariadis et al. [50] optimize the sparse matrix multiplication on GPU tensor cores via partitioning the matrix into tiles and batching the computations onto the tensor cores. B. Feng et al. employs GPU tensor cores to accelerate the neural networks[33] and the scientific computing[35]. Our work differs from the above works that we develop efficient primitives using GPU tensor cores to support more algorithms and applications, including tensor decompositions, training neural networks, and gene analysis.

7.3 Tensor Learning Libraries

TensorLab [28] is a popular MATLAB toolbox for tensor decompositions that supports CP and Tucker tensor decompositions. TensorLy [30] uses NumPy as a backend and was developed to make tensor learning more convenient.

TABLE 2
Running Time and Compression Ratio of a Tucker Tensor Layer

Nets	Parameters	Accuracy	Training Time (s)
Conventional layer	921,354	98.3%	41,153
Tensor layer	315,146	98.1%	54,967
Tensor layer (**ours**)	315,146	97.9%	9,203

TensorLy keeps compatibility with python environment. However, both Tensorlab and TensorLy suffer from a high computation time and only support tensors of limited size. And aforementioned libraries are not optimized for GPU tensor cores. Tao Zhang et al. [25] proposed a transform-based tensor learning library and optimized the operations for GPU tensor cores. However, compared with tubal-rank tensor learning, CP/Tucker tensor decomposition has been more widely used in various applications, such as gene analysis and deep learning. In our work, we focus on optimizing CP/Tucker tensor decomposition on GPU tensor cores to support novel applications.

7.4 Gene Analysis Using Tensor Learning

In gene analysis, the data is usually organized as a third-order tensor with 'individual-tissue-gene' format. Tensor decomposition has been widely used in factorizing the gene data to help analyze the gene expression. V. Hore et al. [17] first employed CP tensor decomposition to factorize the 3-D gene data into a latent of components which uncover the biological relationships between multiple issues. M. Wang [18] proposed to use semi-nonnegative CP tensor decomposition to investigate the transcriptome variation between individuals and tissues. However, because of the large volume of gene data, the expensive computation complexity of tensor computations hinders the efficiency of gene analysis using tensor decomposition. The running time of factorizing the popular gene data GTEx v6, of which size is $18482 \times 544 \times 53$, is up to several of days on CPU, which is usually unbearable. In our work, we employ the optimized CP tensor decomposition on GPU tensor cores to factorize the GTEx v6 dataset in high performance, which only needs about 6 minutes on A100.

7.5 Tensor Layer for Deep Neural Network

By employing Tucker tensor decomposition to the weight matrix, the deep neural networks (DNNs) can be represented in a more compact format. Yin et al. [19] employed Tucker tensor decomposition to compress the Recurrent Neural Networks (RNNs). Song et al. proposed to compress the convolution layer with Tucker format. However, the aforementioned works did not fully exploit the parallelism of tensor operations, thus they did not achieve high performance for the training and inference processes. D. Povey et al. [51] employed the natural gradient to train the fully connected neural network. And K. Osawa et al. [52] developed an efficient GPU implementation of training a neural network with natural gradient. In our work, we employ the natural gradient to train the Tucker tensor layer of DNNs on GPU tensor cores.

8 Discussion and Conclusion

In this article, we presented high-performance tensor learning algorithms using GPU tensor cores and optimized tensor learning primitives including tensor product and tensor contraction. We implement optimized tensor learning primitives into four tensor learning algorithms, namely, third-order and high-order CP/Tucker tensor decomposition, gene analysis using CP tensor decomposition and a Tucker tensor layer for deep neural network. Compared with the TensorLab library running on an A100 GPU, our implementation of CP decomposition achieves an average of $16.32\times$ speedup, while our implementation of Tucker decomposition achieves an average of $6.09\times$ speedup. We include the tensor learning primitives and fashion tensor learning algorithms, CP/Tucker decomposition, two novel applications, gene analysis and Tucker tensor layer, in a library called *cuTensor-CP/Tucker* that will support various big data and machine learning applications.

There remain substantial improvements to be made. The parallelism of training Tucker tensor layer for deep neural networks is not fully utilized. It remains to design a more efficient schedule strategy to accelerate the training process.

In the future, we will extend this high-performance tensor learning library to take advantage of single-GPU, multi-GPU, and even GPU cluster to handle exascale tensors. Besides, we will accommodate more tensor learning algorithms, such as tensor-train, tensor-ring and hierarchical Tucker tensor decomposition, and support more interesting applications, such as quantum machine learning and image generation using generate adversarial networks(GANs). Furthermore, we will integrate our *cuTensorLearning* library as a module to TensorLy [30] and cuTensor [25][26].

References

[1] X.-Y. Liu, H. Lu, and T. Zhang, "cuTensor-CP: High performance third-order CP tensor decompositions on GPUs," in *Proc. Int. Joint Conf. Artif. Intell. Workshop Tensor Netw. Represenations Mach. Learn.*, 2020, pp. 1–4.

[2] X.-Y. Liu, T. Zhang, H. Hong, H. Huang, and H. Lu, "High-performance computing primitives for tensor networks learning operations on GPUs," in *Proc. Int. Conf. Neural Inf. Process. Syst. Workshop Quantum Tensor Netw. Mach. Learn.*, 2020, pp. 1–9.

[3] G. Dahl, J. M. Leinaas, J. Myrheim, and E. Ovrum, "A tensor product matrix approximation problem in quantum physics," *Linear Algebra Appl.*, vol. 420, no. 2/3, pp. 711–725, 2007.

[4] V. Khoromskaia and B. N. Khoromskij, "Tensor numerical methods in quantum chemistry," in *Tensor Numerical Methods in Quantum Chemistry*. Berlin, Germany: Walter De Gruyter, 2018.

[5] A. Malecki et al., "X-ray tensor tomography (a)," *Europhysics Lett.*, vol. 105, no. 3, 2014, Art. no. 38002.

[6] Y. Zhang, X.-Y. Liu, B. Wu, and A. Walid, "Video synthesis via transform-based tensor neural network," in *Proc. ACM Int. Conf. Multimedia*, 2020, pp. 2454–2462.

[7] G. G. Calvi, V. Lucic, and D. P. Mandic, "Support tensor machine for financial forecasting," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process.*, 2019, pp. 8152–8156.

[8] V. N. Ioannidis, A. G. Marques, and G. B. Giannakis, "Tensor graph convolutional networks for multi-relational and robust learning," *IEEE Trans. Signal Process.*, vol. 68, pp. 6535–6546, 2020.

[9] Q. Gao, W. Xia, Z. Wan, D. Xie, and P. Zhang, "Tensor-SVD based graph learning for multi-view subspace clustering," in *Proc. AAAI Conf. Artif. Intell.*, 2020, pp. 3930–3937.

[10] G. Ma, Y. Xiao, T. Willke, N. Ahmed, S. Nazarian, and P. Bogdan, "A distributed graph-theoretic framework for automatic parallelization in multi-core systems," *Proc. Mach. Learn. Syst.*, vol. 3, pp. 550–568, 2021.

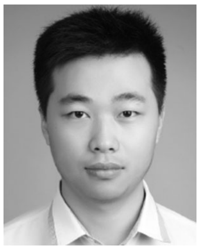
[11] L. Kong, X.-Y. Liu, H. Sheng, P. Zeng, and G. Chen, "Federated tensor mining for secure industrial Internet of Things," *IEEE Trans. Ind. Informat.*, vol. 16, no. 3, pp. 2144–2153, Mar. 2020.

[12] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, no. 3/4, pp. 197–387, 2014.
[13] J. Kossaifi, "NVIDIA research: Tensors are the future of deep learning," *NVIDIA Developer*, 2021, pp. 278–291, [Online]. Available:: https://developer.nvidia.com/blog/nvidia-research-tensors-are-the-future-of-deep-learning/
[14] A. Kolbeinsson et al., "Tensor dropout for robust learning," *IEEE J. Sel. Topics Signal Process.*, vol. 15, no. 3, pp. 630–640, Apr. 2021.
[15] A. Novikov, D. Podoprikhin, A. Osokin, and D. P. Vetrov, "Tensorizing neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2015, pp. 442–450.
[16] P. Springer and P. Bientinesi, "Design of a high-performance GEMM-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, pp. 1–29, 2018.
[17] V. Hore et al., "Tensor decomposition for multiple-tissue gene expression experiments," *Nature Genet.*, vol. 48, no. 9, pp. 1094–1100, 2016.
[18] M. Wang, J. Fischer, and Y. S. Song, "Three-way clustering of multi-tissue multi-individual gene expression data using semi-nonnegative tensor decomposition," *Ann. Appl. Statist.*, vol. 13, no. 2, 2019, Art. no. 1103.
[19] M. Yin, S. Liao, X.-Y. Liu, X. Wang, and B. Yuan, "Towards extremely compact RNNs for video recognition with fully decomposed hierarchical tucker structure," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2021, pp. 12085–12094.
[20] H. Ding, K. Chen, and Q. Huo, "Compressing CNN-DBLSTM models for OCR with teacher-student learning and tucker decomposition," *Pattern Recognit.*, vol. 96, 2019, Art. no. 106957.
[21] X.-Y. Liu and X. Wang, "Real-time indoor localization for smartphones using tensor-generative adversarial nets," *IEEE Trans. Neural Netw. Learn. Syst.*, vol. 32, no. 8, pp. 3433–3443, Aug. 2021.
[22] C. Dai et al., "Video scene segmentation using tensor-train faster-RCNN for multimedia IoT systems," *IEEE Internet Things J.*, vol. 8, no. 12, pp. 9697–9705, Jun. 2021.
[23] F. Shang, Y. Liu, and J. Cheng, "Generalized higher-order tensor decomposition via parallel ADMM," in *Proc. AAAI Conf. Artif. Intell.*, 2014, pp. 1279–1285.
[24] X.-Y. Liu, Y. Fang, L. Yang, Z. Li, and A. Walid, "High performance tensor decompositions for compressing and accelerating deep neural networks," in *Book Tensors for Data Processing*, Amsterdam, The Netherlands: Elsevier, 2021.
[25] T. Zhang, X.-Y. Liu, X. Wang, and A. Walid, "cuTensor-Tubal: Efficient primitives for tubal-rank tensor learning operations on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 595–610, Mar. 2020.
[26] T. Zhang, X.-Y. Liu, and X. Wang, "High performance GPU tensor completion with tubal-sampling pattern," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 7, pp. 1724–1739, Jul. 2020.
[27] B. Zou, C. Li, L. Tan, and H. Chen, "GPUTENSOR: Efficient tensor factorization for context-aware recommendations," *Inf. Sci.*, vol. 299, pp. 159–177, 2015.
[28] N. Vervliet, O. Debals, L. Sorber, M. Van Barel, and L. De Lathauwer, "Tensorlab 3.0," 2016. [Online]. Available: www.tensorlab.net
[29] L. Hao, S. Liang, J. Ye, and Z. Xu, "TensorD: A tensor decomposition library in tensorflow," *Neurocomputing*, vol. 318, pp. 196–200, 2018.
[30] J. Kossaifi, Y. Panagakis, A. Anandkumar, and M. Pantic, "TensorLy: Tensor learning in python," *J. Mach. Learn. Res.*, vol. 20, no. 1, pp. 925–930, 2019.
[31] J. Choquette, W. Gandhi, O. Giroux, N. Stam, and R. Krashinsky, "NVIDIA A100 tensor core GPU: Performance and innovation," *IEEE Micro*, vol. 41, no. 2, pp. 29–35, Mar./Apr. 2021.
[32] P. Blanchard, N. J. Higham, F. Lopez, T. Mary, and S. Pranesh, "Mixed precision block fused multiply-add: Error analysis and application to GPU tensor cores," *SIAM J. Sci. Comput.*, vol. 42, no. 3, pp. C124–C141, 2020.
[33] 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.
[34] A. Mishra et al., "Accelerating sparse deep neural networks," 2021, *arXiv:2104.08378*.
[35] B. Feng, Y. Wang, G. Chen, W. Zhang, Y. Xie, and Y. Ding, "EGEMM-TC: Accelerating scientific computing on tensor cores with extended precision," in *Proc. 26th ACM SIGPLAN Symp. Princ. Pract. Parallel Program.*, 2021, pp. 278–291.
[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.*, to be published, doi: 10.1109/TC.2022.3172895.
[37] J. Finkelstein et al., "Quantum-based molecular dynamics simulations using tensor cores," *J. Chem. Theory Comput.*, vol. 17, no. 10, pp. 6180–6192, 2021.
[38] J. Lonsdale et al., "The genotype-tissue expression (GTEx) project," *Nat. Genet.*, vol. 45, no. 6, pp. 580–585, 2013.
[39] Y. LeCun, C. Cortes, and C. Burges, "Mnist handwritten digit database," *ATT Labs*, vol. 2, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist
[40] J. D. Carroll and J.-J. Chang, "Analysis of individual differences in multidimensional scaling via an n-way generalization of "eckart-young" decomposition," *Psychometrika*, vol. 35, no. 3, pp. 283–319, 1970.
[41] 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.
[42] NVIDIA-Corporation, "NVIDIA cuTensor library," 2020. [Online]. Available: https://developer.nvidia.com/cutensor
[43] A. Paszke et al., "PyTorch: An imperative style, high-performance deep learning library," *Adv. Neural Inf. Process. Syst.*, vol. 32, pp. 8026–8037, 2019.
[44] L. Dagum and R. Menon, "OpenMP: An industry standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, no. 1, pp. 46–55, First Quarter 1998.
[45] Y. Xiao, S. Nazarian, and P. Bogdan, "Plasticity-on-chip design: Exploiting self-similarity for data communications," *IEEE Trans. Comput.*, vol. 70, no. 6, pp. 950–962, Jun. 2021.
[46] J. Li, Y. Ma, C. Yan, and R. Vuduc, "Optimizing sparse tensor times matrix on multi-core and many-core architectures," in *Proc. 6th Workshop Irregular Appl.: Architect. Algorithms*, 2016, pp. 26–33.
[47] P. Springer, J. Hammond, and P. Bientinesi, "TTC: A high-performance compiler for tensor transpositions," *ACM Trans. Math. Softw.*, vol. 44, no. 2, pp. 15:1–15:21, 2017.
[48] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, "Optimizing sparse tensor times matrix on GPUs," *J. Parallel Distrib. Comput.*, vol. 129, pp. 99–109, 2019.
[49] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka, "Tensor contractions with extended BLAS kernels on CPU and GPU," in *Proc. IEEE Int. Conf. High Perform. Comput.*, 2016, pp. 193–202.
[50] 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.
[51] D. Povey, X. Zhang, and S. Khudanpur, "Parallel training of DNNs with natural gradient and parameter averaging," in *Proc. Int. Conf. Learn. Representations*, 2015, pp. 1–28.
[52] K. Osawa, Y. Tsuji, Y. Ueno, A. Naruse, C.-S. Foo, and R. Yokota, "Scalable and practical natural gradient for large-scale deep learning," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 1, pp. 404–415, Jan. 2022.

Xiao-Yang Liu (Member, IEEE) received the graduate degree in PhD program from the Department of Computer Science and Engineering, Shanghai Jiao Tong University, 2017, and the MS degree in electrical engineering from Columbia University, New York, NY, USA, in 2018. He is currently working toward the PhD degree with the Department of Electrical Engineering, Columbia University, New York, NY, USA. His research interests include tensor and tensor networks, high-performance tensor computing, deep reinforcement learning, non-convex optimization, big data analysis and IoT applications.

Zeliang Zhang is currently working toward the graduate degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. His research interests include high-performance tensor computing and deep learning.

Zhiyuan Wang is currently working toward the graduate degree with the School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China. His research interests include high-performance tensor computing and deep learning.



Han Lu received the BEng degree in computer science and technology from the Hebei University of Engineering, China, in 2017. He is currently working toward the graduate degree with the School of Computer Engineering and Science, Shanghai University, China. His research interests include GPU parallel computing, tensor computing, and optimization acceleration of algorithms.

Xiaodong Wang (Fellow, IEEE) received the PhD degree in electrical engineering from Princeton University. He is currently a professor of electrical engineering with Columbia University, New York NY, USA. His research interests fall in the general areas of computing, signal processing, and communications. He has authored the book entitled Wireless Communication Systems: Advanced Techniques for Signal Reception, (Prentice Hall, 2003). He served as an associate editor of the *IEEE Transactions on Communications*, *IEEE Transactions on Wireless Communications*, *IEEE Transactions on Signal Processing*, and *IEEE Transactions on Information Theory*. He is an ISI Highly Cited Author. He received the 1999 NSF CAREER Award, the 2001 IEEE Communications Society and Information Theory Society Joint Paper Award, and the 2011 IEEE Communication Society Award for Outstanding Paper on New Communication Topics.

Anwar Walid (Fellow, IEEE) received the BS degree in electrical engineering from New York University, and the PhD degree in electrical engineering from Columbia University, New York. He is the head of the mathematics of Systems Research Department, Nokia Bell Labs. He received awards from the ACM and IEEE, including 2017 IEEE Communication Society William R. Bennett Prize and 2019 ACM SIGCOMM Networking Systems Award. He served as an associate editor of *IEEE/ACM Transactions on Networking (ToN)*, *IEEE Transactions on Cloud Computing*, and *IEEE Network Magazine*. He is an adjunct professor with the Electrical Engineering Department, Columbia University. He is a elected member of Tau Beta Pi National Engineering Honor Society and IFIP Working Group 7.3.