High Performance Computing Primitives for Tensor Networks Learning Operations on GPUs

Xiao-Yang Liu[1], **Tao Zhang**[2*], **Hao Hong**[2], **Hao Huang**[2], **Han Lu**[2]
[1]Department of Electrical Engineering, Columbia University, USA
[2]School of Computer Engineering and Science, Shanghai University, Shanghai, China
{taozhang, honghao,baxlumen,130210201}@shu.edu.cn, xl2427@columbia.edu

Abstract

Tensor decompositions and tensor networks that factorize multi-dimensional data into latent factors have become a powerful tool for big data analytics and machine learning. However, time and space complexities of the algorithms grow rapidly with the size of tensors. Exploiting parallelisms of tensor network learning operations and accelerating them on many-core GPUs are promising. In this paper, we develop efficient primitives for tensor network learning operations on GPUs by exploiting tensor algorithm parallelism. First, we implement and optimize key operations to improve resource utilization, including tensor matricization and matricized tensor times Khatri-Rao product (MTTKRP), highly-parallel Jacobi-based singular value decomposition (SVD), and batch operations. Second, we fully optimize the data transfer, memory access and reduce memory footprint, even employ more efficient calculation processes with smaller computational complexity. Thirdly, we support several tensor networks learning operations, such a CP, Hierarchical Tucker (HT), tensor-train (TT) and tensor-ring (TR) tensor decompositions. Finally, we evaluate on a Tesla V100 GPU and tested tensors up to 1,200 $\times$ 1,200 $\times$ 1,200. Compared with the TensorLab library, our implementation of CP decomposition achieves up to $5.56\times$ speedup. Compared with GPU baseline implementations, the proposed GPU implementations of HT, TT and TR decompositions achieve up to $4.67\times$, $6.67\times$ and $6.36\times$ speedups, respectively.

1 Introduction

Tensor decompositions, the higher-order analogue to matrix decomposition (e.g., Singular Value Decomposition (SVD), Principal Component Analysis (PCA), non-negative matrix decompoistion, etc.), have became a powerful tool for mining cross-dimension relationships in large-scale multi-dimensional data. Multi-dimensional data arrays in video processing, social networks are naturally represented as tensors, and tensor (multiway) decompositions are employed to perform factor analysis or compression. Tensor decompositions/factorizations have been widely used in big data analysis [11], computer vision [12] [6], pattern recognition and deep learning [7][1][8][4][13], and genetic analysis [5], etc. With the growing needs to process large amount of multi-way data in a real-time manner, designing high-performance tensor decomposition has become a critical task.

Existing research in accelerating CP tensor decomposition has limitations. TensorLab [10] requires long running time, which was not fully optimized for the GPU architecture. In this paper, we develop high performance CP tensor decomposition on GPUs. We use the bottom-up method to optimize the key operations first, and then accelerate the whole algorithm. Our contributions are summarized as follows.

*Corresponding author: Tao Zhang.

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.

Algorithm 1 CP Tensor Decomposition

Input: tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, rank R.
Initialize $\boldsymbol{A} \in \mathbb{R}^{I \times R}, \boldsymbol{B} \in \mathbb{R}^{J \times R}, \boldsymbol{C} \in \mathbb{R}^{K \times R}$,
while convergence criterion is not met **do**
  $\boldsymbol{A} \leftarrow \boldsymbol{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B})(\boldsymbol{C}^\top \boldsymbol{C} .* \boldsymbol{B}^\top \boldsymbol{B})^\dagger$,
  $\boldsymbol{B} \leftarrow \boldsymbol{X}_{(2)}(\boldsymbol{C} \odot \boldsymbol{A})(\boldsymbol{C}^\top \boldsymbol{C} .* \boldsymbol{A}^\top \boldsymbol{A})^\dagger$,
  $\boldsymbol{C} \leftarrow \boldsymbol{X}_{(3)}(\boldsymbol{B} \odot \boldsymbol{A})(\boldsymbol{B}^\top \boldsymbol{B} .* \boldsymbol{A}^\top \boldsymbol{A})^\dagger$,
end while
Output: $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}$.

Algorithm 2 Hierarchical Tucker Tensor Decomposition

Input: Tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, rank r_1, r_2, r_3, r_4.
$1: \boldsymbol{U}_1 \leftarrow$($r_1$ leading left singular vectors) of $\boldsymbol{X}_{(1)}$,
$2: \boldsymbol{U}_2 \leftarrow$($r_2$ leading left singular vectors) of $\boldsymbol{X}_{(2)}$,
$3: \boldsymbol{U}_3 \leftarrow$($r_3$ leading left singular vectors) of $\boldsymbol{X}_{(3)}$,
$4: \boldsymbol{U}_4 \leftarrow$ (r_4 leading left singular vectors) of $\boldsymbol{X}_{(3)}^T$,
$5: \mathcal{U}_4 \leftarrow$ tensorizing of $\boldsymbol{U}_4$,
$6: \mathcal{B}_2 = \mathcal{U}_4 \times_1 \boldsymbol{U}_1^T \times_2 \boldsymbol{U}_2^T$,
$7: \mathcal{B}_1 = \mathcal{X} \times_1 \boldsymbol{U}_4^T \times_2 \boldsymbol{U}_3^T$,
Output: $\mathcal{B}_1, \mathcal{B}_2, \boldsymbol{U}_3, \boldsymbol{U}_2, \boldsymbol{U}_1$.

Figure 1: Third-order CP (left) and HT (right) tensor decompositions algorithm.

- We implement key tensor operations, including tensor matricization, matricized tensor times Khatri-Rao product (MTTKRP), and accelerations using tensor cores.
- We propose optimization strategies for memory access, reducing the amount of calculation, reducing memory footprint, improving resource utilization, thereby improving algorithm performance.
- We implement CP, HT, TT and TR tensor decompositions on GPUs, which achieve high performance and the same accuracy as CPU.
- On a Tesla V100 GPU, we perform numerical experiments to evaluate our tensor network learning operations. Compared with the TensorLab-GPU [10], CP decomposition achieves up to $5.56\times$ speedups. Compared with GPU baselines, our HT, TT and TR decompositions achieve $4.67\times$, $6.67\times$ and $6.36\times$ speedups, respectively.

The remainder of this paper is organized as follows. Section 2 describes tensor network learning operations and algorithms. Section 3 presents the implementations and optimizations. In Section 4, we evaluate the performance on GPUs. The conclusions are given in Section 5.

2 Tensor Networks Learning Operations

Note that CP, HT, TT and TR tensor decompositions are typical forms of tensor networks.

2.1 Notions and Key Tensor Operations

We use uppercase boldface letters and uppercase calligraphic letters to denote matrices and tensors, respectively, e.g., $\boldsymbol{X} \in \mathbb{R}^{I \times J}$, and $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. We use $.*$, $\circ$, $\odot$ and $\otimes$ to denote Hadamard (element-wise) product, outer product, Khatri-Rao product and Kronecker (tensor) product. We use $^\top$ and † to the matrix transpose and Moore-Penrose pseudo-inverse, respectively. Indices range from 1 to their capital letters, e.g., $i = 1, ..., I$ or $i \in [I]$. The Frobenius norm of a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ is defined as $\|\mathcal{X}\|_F = \sqrt{\sum_{i=1}^{I} \sum_{j=1}^{J} \sum_{k=1}^{K} |\mathcal{X}_{ijk}|^2}$.

Rank-one tensor: A tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ has rank one if it is the outer product of three vectors.

Tensor matricization (tensor unfolding, flattening): The mode-n matricization of $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ is denoted by $\mathbf{X}_{(n)}$, where the element (i_1, i_2, i_3) is mapped to matrix element (i_n, j) for $n = 1, 2, 3$,

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

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

Algorithm 3 Third-order Tensor-Train Decomposition

Input: Tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, pre-specified accuracy ε.
$\mathcal{C}^{(0)} = \mathcal{A}, r_0 = r_1 = r_2 = r_3 = 1,$
for $k = 1$ to 2 **do**
  $\boldsymbol{C} = \text{reshape}(\mathcal{C}^{(k-1)}, [r_{k-1}n_k, \prod_{i=k+1}^{3} n_i]),$
  Compute SVD: $\boldsymbol{C} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T,$ and $\boldsymbol{s} = \text{diag}(\boldsymbol{S}),$
  $\delta = \frac{\varepsilon}{\sqrt{3-1}}\|\boldsymbol{s}\|_2,\ \gamma = 0,\ r_k = \sharp(\boldsymbol{s}),$
  while $\gamma \leq \delta$ **do**
    $\gamma = \gamma + s_{r_k}^2,\ r_k = r_k - 1,$
  end while
  $r_k = r_k + 1,\ \boldsymbol{U} = \boldsymbol{U}(:, 1:r_k),$
  $\boldsymbol{S} = \boldsymbol{S}(1:r_k, 1:r_k),\ \boldsymbol{V}^T = \boldsymbol{V}^T(1:r_k, :),$
  $\mathcal{G}^{(k)} = \text{reshape}(\boldsymbol{U}, [r_{k-1}, n_k, r_k]),$
  $\mathcal{C}^{(k)} = \text{reshape}(\boldsymbol{S}\boldsymbol{V}^T, [r_k, \prod_{i=k+1}^{3} n_i, r_{k+1}]),$
end for
$\mathcal{G}^{(3)} = \mathcal{C}^{(2)}.$
Output: Cores $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \mathcal{G}^{(3)}, r_0, r_1, r_2, r_3 \in \mathbb{N}^+.$

Algorithm 4 Third-order Tensor-Ring Decomposition

Require: Tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, pre-specified accuracy ε.
Ensure: Cores $\mathcal{G}^{(1)}, \mathcal{G}^{(2)}, \mathcal{G}^{(3)}, r_0, r_1, r_2, r_3 \in \mathbb{N}^+.$
$\mathcal{C}^{(0)} = \mathcal{A}, m = r_0 = r_1 = r_2 = r_3 = 1,$
Choose the first mode as the start point,
  $\boldsymbol{C} = \text{reshape}(\mathcal{C}^{(0)}, [r_0 n_1, n_2 n_3]),$
$\boldsymbol{C} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^T,\ \boldsymbol{s} = \text{diag}(\boldsymbol{S}),$
$\delta = \frac{\varepsilon}{\sqrt{3-1}}\|\boldsymbol{s}\|_2,\ \gamma = 0,\ m = \sharp(\boldsymbol{s}),$
while $\gamma \leq \delta$ **do**
  $\gamma = \gamma + s_m^2, m = m - 1,$
end while
$m = m + 1,\ \boldsymbol{U} = \boldsymbol{U}(:, 1:m),$
  $\boldsymbol{S} = \boldsymbol{S}(1:m, 1:m),\ \boldsymbol{V}^T = \boldsymbol{V}^T(1:m, :),$
Split ranks r_0, r_1 by
  $\min_{r_0, r_1} |r_0 - r_1|,$ s.t. $r_0 r_1 = m,$
$\mathcal{G}^{(1)} = \text{permute}(\text{reshape}(\boldsymbol{U}, [n_1, r_0, r_1]), [2, 1, 3]),$
$\mathcal{C} = \text{permute}(\text{reshape}(\boldsymbol{S}\boldsymbol{V}^T, [r_0, r_1, n_2 n_3]), [2, 3, 1]),$
$\boldsymbol{C} = \text{reshape}(\mathcal{C}, [r_1 n_2, n_3 r_0]),$
Repeat Line 3 to 8, and set $r_2 = m, r_3 = r_0$
$\mathcal{G}^{(2)} = \text{reshape}(\boldsymbol{U}, [r_1, n_2, r_2]),$
$\mathcal{G}^{(3)} = \text{reshape}(\boldsymbol{S}\boldsymbol{V}^T, [r_2, n_3, r_3]).$

Figure 2: Third-order TT (left) and TR (right) tensor decompositions algorithm.

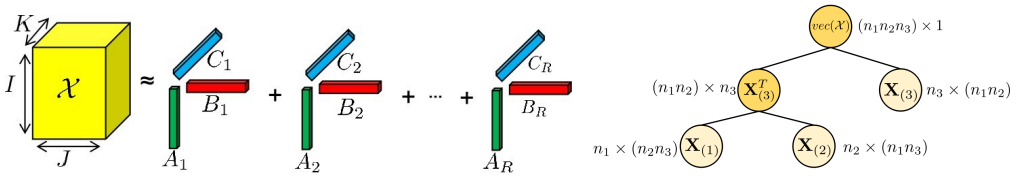

Figure 3: Third-order CP and HT tensor decompositions.

2.2 CP Tensor Model and Algorithm

CP tensor decomposition [2], as shown in Fig. 3, factorizes a tensor into the sum of rank-one tensor components. For a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and target rank R, we have $\mathcal{X} \approx \sum_{r=1}^{R} \boldsymbol{A}_r \circ \boldsymbol{B}_r \circ \boldsymbol{C}_r$, where $\boldsymbol{A}_r \in \mathbb{R}^I,\ \boldsymbol{B}_r \in \mathbb{R}^J,\ \boldsymbol{C}_r \in \mathbb{R}^K$ for $r \in [R]$. Compute a CP tensor decomposition that best approximates $\mathcal{X}$, i.e.,

$$\arg\min_{\widehat{\mathcal{X}}} \|\mathcal{X} - \widehat{\mathcal{X}}\|_F \tag{2}$$

where $\widehat{\mathcal{X}} = \sum_{r=1}^{R} \lambda_r \boldsymbol{A}_r \circ \boldsymbol{B}_r \circ \boldsymbol{C}_r$ and $\boldsymbol{A} \in \mathbb{R}^{I \times R}, \boldsymbol{B} \in \mathbb{R}^{J \times R}, \boldsymbol{C} \in \mathbb{R}^{K \times R}$, respectively.

In Alg. 1, the alternating least square method (ALS) fixes all but one matrix and reduces the CP tensor decomposition problem to a linear least-squares problem. In line 2, the factor matrices are randomly initialized. Lines 3-7 are the iterative process and the algorithm updates the factor matrices (lines 4-6) alternatively. The algorithm terminates when the approximation error is below a pre-specified threshold or it reaches a preset maximum number of iterations.

2.3 HT Tensor Model and Algorithm

Generally, the HT format is stored in the form of a binary tree $\mathcal{T}$, where each branch is a hierarchical division of the tensor mode set. For example the binary tree of a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ is given in Fig. 1. HT decomposition is to perform the singular value decomposition (SVD) on each child node of the tree structure. Among these nodes, each leaf node stores the left singular vectors matrix. For non-leaf nodes t, it stores a tensor $\mathcal{B}_t$ called transfer tensor, which satisfies:

$$\mathcal{B}_t = \boldsymbol{U}_t \times_1 \boldsymbol{U}_{t_l}^T \times_2 \boldsymbol{U}_{t_r}^T, \tag{3}$$

where the $\boldsymbol{U}_t$ is the left singular vectors matrix of node t, and $\boldsymbol{U}_{t_l}$, $\boldsymbol{U}_{t_r}$ are left singular vectors matrices of the children of t. The $\times_1$ and $\times_2$ are tensor times matrix (TTM) or tensor contraction.

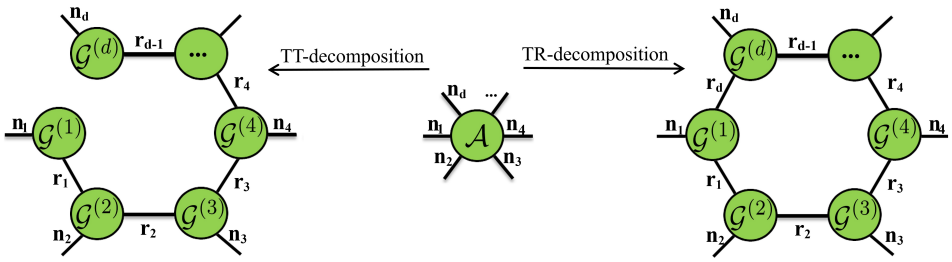

Figure 4: An illustration of TT and TR decompositions for a d-th order tensor.

In Alg. 2, we first perform matricization on the tensor in different mode sets and use the matrix $\boldsymbol{X}_{(t)}$ to get the left singular vectors matrix $\boldsymbol{U}_t \in \mathbb{R}^{n_t \times r_t}$ in lines 1-4. In line 5, tensorizing means it reshapes the matrix into a tensor. Secondly, we use (3) to calculate the transfer tensor $\mathcal{B}$ stored in the non-leaf node. The algorithm is described in Alg. 1 proposed by [3] and we set $r_t = 0.2 \times n_t$.

2.4 TT Tensor Model and Algorithm

TT decomposition [9] expresses a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ as contractions of three core tensors:

$$\mathcal{A} = \mathcal{G}^{(1)} \circ \mathcal{G}^{(2)} \circ \mathcal{G}^{(3)}, \tag{4}$$

where $\mathcal{G}^{(k)} \in \mathbb{R}^{r_{k-1} \times n_k \times r_k}$ is the k-th core tensor. The auxiliary indices $[r_0, r_1, r_2, r_3]$ are the tensor-train ranks (TT-ranks), and $r_0 = r_3 = 1$ for third-order tensors.

Alg. 3 describes the procedures of the third-order TT decomposition [9]. As shown in Fig. 2, the connection "leg" between two circles represents the contraction operation of two tensors. Through the contraction of every small tensors, we can get the original tensor $\mathcal{A}$. The $\boldsymbol{C} = \text{reshape}(\mathcal{C}^{(k-1)}, [r_{k-1}n_k, \prod_{i=k+1}^{3} n_i])$ operation changes the tensor $\mathcal{C}^{(k-1)}$ to a matrix $\boldsymbol{C}$ with $r_{k-1}n_k$ rows and $\prod_{i=k+1}^{3} n_i$ columns. The $\mathcal{G}^{(k)} = \text{reshape}(\boldsymbol{U}, [r_{k-1}, n_k, r_k])$ operation changes the matrix $\boldsymbol{U}$ to a tensor $\mathcal{G}^{(k)}$ with r_{k-1} rows, n_k columns, and r_k in the third direction.

2.5 TR Tensor Model and Algorithm

TR decomposition [14], as shown in Fig. 2, represents a tensor $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$ with three third-order latent tensors $\mathcal{G}^{(k)} \in \mathbb{R}^{r_k \times n_k \times r_{k+1}}, k = 1, 2, 3$:

$$\mathcal{A} = \mathcal{G}^{(1)} \circ \mathcal{G}^{(2)} \circ \mathcal{G}^{(3)},$$

where we also calculate the contraction between $\mathcal{G}^{(1)}$ and $\mathcal{G}^{(3)}$. Auxiliary indices $[r_1, r_2, r_3]$ are the tensor-ring ranks (TR-ranks). Because of the trace characteristic of TR-format tensor, $r_4 = r_1$, which leads to the main difference between the TT decomposition and the TR decomposition.

Alg. 4 describes the procedures of the third-order TR decomposition. Because of the ring-shaped feature, the third-order TR decomposition needs to choose a start point. The $\mathcal{C} = \text{permute}(\text{reshape}(\boldsymbol{S}\boldsymbol{V}^T, [r_1, r_2, n_2 n_3]), [2, 3, 1])$ operation transposes the tensor dimensions from $[r_1, r_2, n_2 n_3]$ to $[r_2, n_2 n_3, r_1]$.

3 Optimization of Key Tensor Operations

The horizontal, lateral, and frontal slices of a third-order tensor $\mathcal{X}$ are denoted as $\mathcal{X}(i, :, :)$, $\mathcal{X}(:, j, :)$, and $\mathcal{X}(:, :, k)$, respectively. Alternatively, the k-th frontal slice $\mathcal{X}(:, :, k)$ is denoted as $\mathcal{X}^{(k)}$.

3.1 Tensor Matricization

Tensor matricization is a fundamental operation in CP tensor decomposition. In Alg. **??**, lines 4-6 need to compute different mode matricizations $\boldsymbol{X}_{(1)}, \boldsymbol{X}_{(2)}$ and $\boldsymbol{X}_{(3)}$. In conventional implementation,

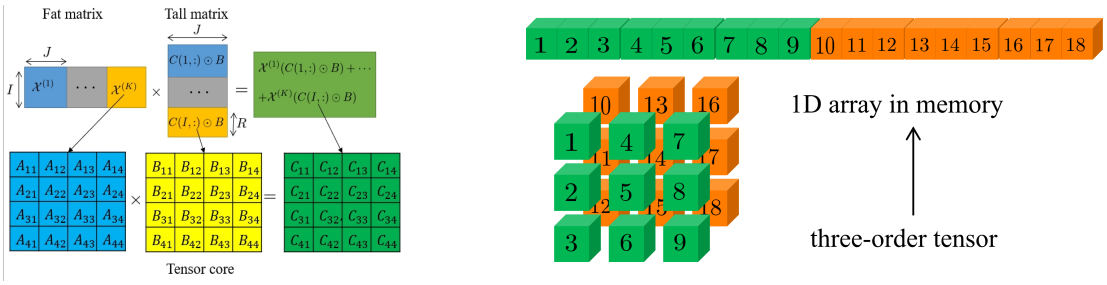

Figure 5: Block computation for the MTTKRP operation and tensor data access.

GPU allocates additional memory and performs explicit tensor matricization, which introduces substantial memory and time cost.

A third-order tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ is stored in GPU memory in slice-by-slice, column-major layout using a 1D array $\boldsymbol{x}$, where $\mathcal{X}_{ijk}$ is stored at $\boldsymbol{x}[(k-1)IJ + (j-1)I + i]$. In the traditional method, in order to implement tensor contractions, we need to change the storage order of each element of the tensor in the GPU, which needs extra time and space cost. We observe that different tensor matricization can be obtained through different data access methods. First, the column-major storage of mode-1 matricization $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I \times JK}$ in memory is exactly the same as $\boldsymbol{x}$. Second, by transposing each slice of $\mathcal{X}$, we obtain $\boldsymbol{X}_{(2)}$. Third, the row-major storage of mode-3 matricization $\boldsymbol{X}_{(3)}$ is exactly the same as $\boldsymbol{x}$. By exploiting these property, we are able to use $\mathcal{X}$ to represent the result of tensor matrixization directly and avoid explicit tensor matricization to save computation and GPU memory.

In CUDA programming, we obtain the three matricizations $\boldsymbol{X}_{(1)}, \boldsymbol{X}_{(2)}, \boldsymbol{X}_{(3)}$ by different ways of fetching the 1D data in the memory. In the cuBLAS library, cublasSgemm() and cublasSgemmBatchedStrided() are interfaces of matrix multiplication and parallel matrix multiplication. For mode-1 matricization $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I \times JK}$, we use cublasSgemm() and set the main dimension as I. For mode-2 matricization $\boldsymbol{X}_{(2)} \in \mathbb{R}^{J \times IK}$, we use cublasSgemmBatchedStrided() and set the leading dimension to J, and the number of matrix multiplications to K. For mode-3 matricization $\boldsymbol{X}_{(3)} \in \mathbb{R}^{K \times IJ}$, we find that the column-major storage of $\mathcal{X}$ in 1D array is same as the transpose of $\boldsymbol{X}_{(3)}$. Therefore, we use cublasSgemm() and set the leading dimension as K. In general, through the access strategy, we cleverly regard physical storage data as the form of logical storage we require.

3.2 Matricized Tensor Times Khatri-Tao Product (MTTKRP)

MTTKRP is a basic operation in tensor computing. The convention approach to compute MTTKRP include the following three steps:

- Matricizing a tensor into a matrix: $\mathcal{X} \rightarrow \boldsymbol{X}_{(1)}, \boldsymbol{X}_{(2)}, \boldsymbol{X}_{(3)}$;
- Calculating Khatri-Rao product and obtain $(\boldsymbol{C} \odot \boldsymbol{B})$, $(\boldsymbol{C} \odot \boldsymbol{A})$, $(\boldsymbol{B} \odot \boldsymbol{A})$;
- Executing matrix multiplication: $(\boldsymbol{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B}))$, $(\boldsymbol{X}_{(2)}(\boldsymbol{C} \odot \boldsymbol{A}))$, $(\boldsymbol{X}_{(3)}(\boldsymbol{B} \odot \boldsymbol{A}))$.

We eliminate the tensor matricization in the first step through the technique in Section 3.1. Since in the third step, tensor matricization results in a fat matrix and the Khatri-Rao product results in a tall matrix, we use tensor core to accelerate this matrix multiplication.

3.2.1 Batching Block Computations onto Tensor Cores

In Alg. 1, the main while-loop in lines 4-6 performs three MTTKRP operations. This process is time-consuming and becomes a bottleneck. As matrix multiplication can be calculated in a block manner, we divide the large matrices into smaller matrices and batch the block matrix multiplications onto tensor cores.

Tensor cores are novel computing units introduced in latest NVIDIA GPU architectures including Volta and Turing. Compared with conventional CUDA cores, tensor cores are especially efficient for accelerating the computation of block matrix multiplications. Tesla V100 GPU has 640 tensor

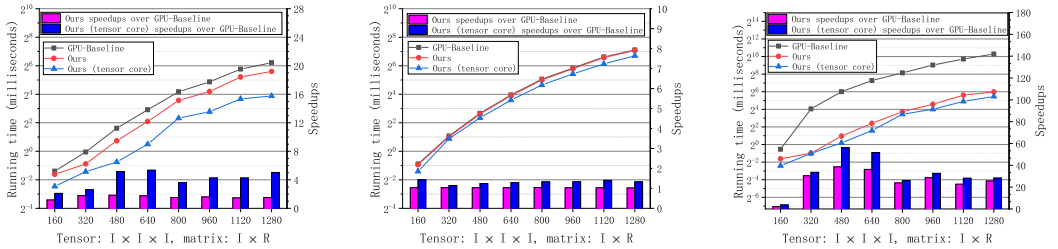

Figure 6: Running time and speedups of MTTKRP in three modes, respectively.

cores in total. The matrix multiplication of two 4×4 matrices can be calculated on a tensor core at one time. Fig. 5 illustrates the computing of $\boldsymbol{X}_{(1)}(\boldsymbol{C} \odot \boldsymbol{B})$, where $\boldsymbol{X}_{(1)} \in \mathbb{R}^{I \times JK}$, $\boldsymbol{C} \in \mathbb{R}^{K \times R}$ and $\boldsymbol{B} \in \mathbb{R}^{J \times R}$. We divide the fat matrix $\boldsymbol{X}_{(1)}$ and the tall matrix $\boldsymbol{C} \odot \boldsymbol{B}$ into $(IJK)/16$ and $(JKR)/16$ small blocks, respectively. Then, we carry out a total of $I/4 \times (JK)/4 \times (R/4)$ block multiplications and batch them onto 640 tensor cores. When one is calculated, the blocks in the queue will immediately occupy the computing resources.

3.3 Batch Operations

In the HT decomposition algorithm, when updating the left singular values matrix $\boldsymbol{U}_t$ stored in the leaf node, the updating steps of each node are the same. All we need is the left singular vectors matrix. So we use the method of eigen decomposition to solve, because it has fewer parameters and time consumption. First we calculate the intermediate matrix $\boldsymbol{H}_t$ with $\boldsymbol{X}_{(t)}$ and $\boldsymbol{X}_{(t)}^T$, and then use the eigen decomposition to get $\boldsymbol{U}_t$. In this step, the data used in each execution process is independent of each other and has no dependencies. To improve GPUs utilization and algorithm performance, we perform eigen decomposition parallelly through batch processing.

We need to perform eigen decomposition on each matrix $\boldsymbol{H}_t$, and this process is repeated three times. In the regular routine, these decompositions are conducted one by one sequentially on GPUs. However, this way does not fully utilize hardware threads on GPUs. Instead, we use the routine syevjBatched$(\cdot)$ which performs eigen decomposition using the Jacobi method for each $\boldsymbol{H}_t$. The parallelism of Jacobi method gives the GPU better performance on small and medium size matrices. Moreover we configure the parameters in this routine to improve accuracy.

3.4 Memory Access Optimization

In general, the $\boldsymbol{H}_t$ is stored in the global memory for the eigen decomposition. However, compared with the shared memory inside the streaming multiprocessor (SM) on GPUs, the GPU global memory has higher latency and lower bandwidth. In order to batch the eigen decomposition, we need to merge the matrix $\boldsymbol{H}_t$ into a large matrix. The $\boldsymbol{H}_t$ needs to be accessed multiple times, which causes excessive time complexity. To improve performance, we use low-latency shared memory instead of global memory to storage the $\boldsymbol{H}_t$. We launch $3 \times n$ blocks at the same time and transfer the $\boldsymbol{H}_t$ from the global memory to the shared memory. When combining the matrix $\boldsymbol{H}_t$, the algorithm accesses the shared memory $3 \times n$ times. Compared to global memory, using shared memory is much faster

4 Performance Evaluations

We run all experiments on a server with an NVIDIA Tesla V100 GPU and dual Intel Xeon E5-2640 V4 CPUs. The Tesla V100 GPU has 32 GB device memory, $5,120$ CUDA cores and 640 tensor cores. Each CPU has 10 cores running at 2.4GHz. The operating system of the server is 64-bit Ubuntu 18.04.

We use running time as performance metric. 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). The compared GPU implementations are listed out as follows

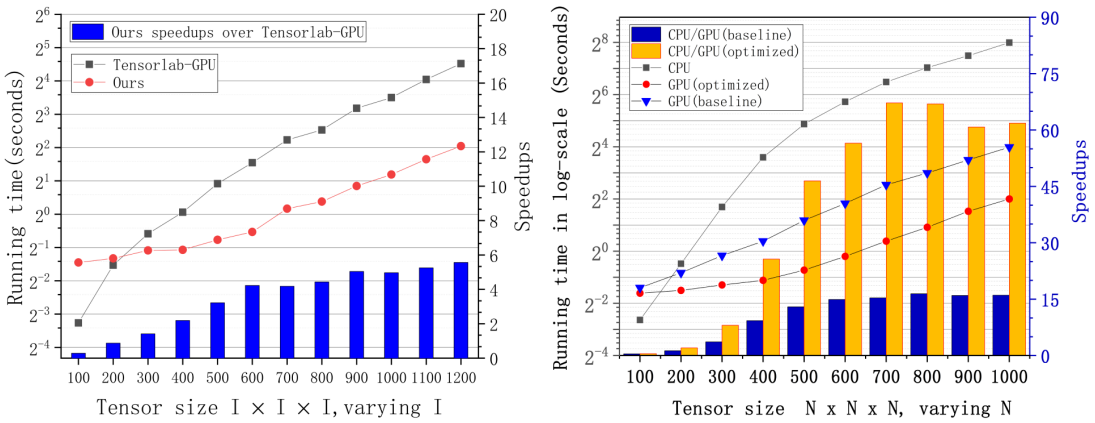

Figure 7: Running time and speedups of CP (left) and HT (right) decomposition.

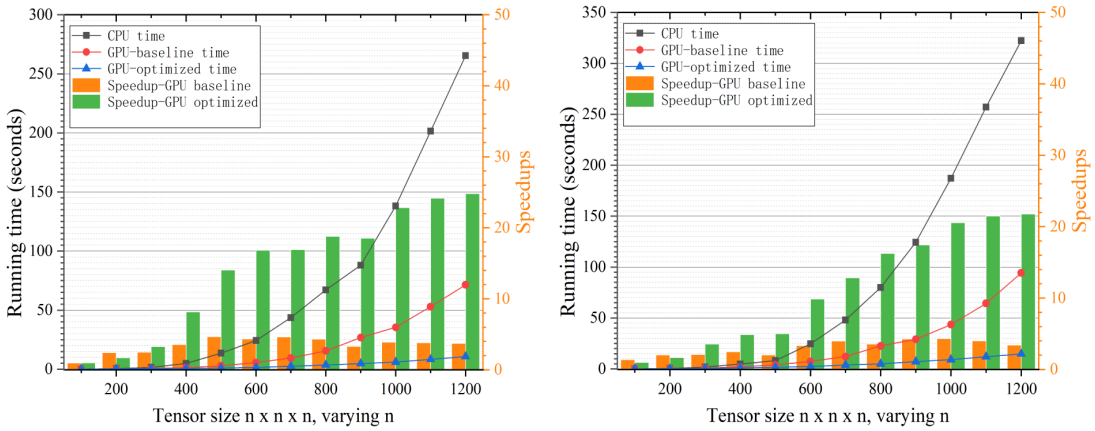

Figure 8: Running time and speedups of TT (left) and TR (right) decomposition.

- **GPU Baseline**: We provide baseline implementation on GPU using the BLAS and cuSOLVER libraries. This implementation does not utilize the optimization techniques in Section 3.
- **TensorLab-GPU** [10]: TensorLab is a well-maintained MATLAB toolbox for tensor computations. We run TensorLab on GPUs.
- **Our GPU implementation (Ours)**: For the key tensor operations and CP tensor decomposition algorithm, our implementations employ the optimization techniques in Section 3.

4.1 Key Tensor Operations

In this subsection, our experiments are running on Tesla V100 GPU. Running time is measured in log scale. For the key tensor operation (MTTKRP), we report the kernel running time that do not include the data transfer time between CPU and GPU. Because the tensor operation is normally called during the computation process of the tensor decomposition algorithms and the data are already in the GPU device memory.

Fig. 6 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 be $0.1 \times I$ and the corresponding outputs are three tensors of the same size $I \times R$.

In Fig. 6(a), 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. 6(b), 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. 6(c), 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.

4.2 Third-order CP Tensor Decomposition

Fig .7 (left) shows the running time and speedups of the CP tensor decomposition. We test the third-order tensor of size $I \times I \times I$ varying from $100 \times 100 \times 100$ to $1,200 \times 1,200 \times 1,200$. Two implementations are compared: our GPU implementation and TensorLab-GPU [10]. Our GPU implementation achieves up to $5.56\times$ speedup versus the TensorLab-GPU [10].

4.3 Third-order HT Tensor Decomposition

Fig. 7 (right) shows the running time and speedups of the HT tensor decomposition. We test the third-order tensor of size $N \times N \times N$ varying from $100 \times 100 \times 100$ to $1,000 \times 1,000 \times 1,000$. The optimized GPU implementation achieves $4.67\times$ speedups over the unoptimized GPU baseline.

4.4 Third-order TT and TR Tensor Decomposition

Fig. 8 shows the running time and speedups. We test the third-order tensor of size $N \times N \times N$ varying from $100 \times 100 \times 100$ to $1,200 \times 1,200 \times 1,200$. Compared with the unoptimized GPU baseline, the optimized GPU implementation achieves $6.67\times$ and $6.36\times$ speedups for third-order TT and TR decompositions, respectively.

5 Conclusions

Tensor operations have attracted lots of attention in recent years. Tensor decompositions have been widely used in big data analysis, computer vision, pattern recognition, and deep learning, etc. However, due to the high computational complexity, existing implementations are not satisfactory in terms of running time and memory consumption. In this paper, we optimized the computations of CP tensor decomposition on many-core GPUs. We proposed optimization strategies for reduced memory consumption to accelerate tensor operations. Compared with TensorLab-GPU running on a Tesla V100 GPU, our implementation of CP tensor decomposition achieves up to a $5.56\times$ speedup.

References

[1] Yunpeng Chen, Xiaojie Jin, Bingyi Kang, Jiashi Feng, and Shuicheng Yan. Sharing residual units through collective tensor factorization to improve deep neural networks. In *IJCAI*, pages 635–641, 2018.

[2] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle. A multilinear singular value decomposition. *SIAM Journal on Matrix Analysis and Applications*, 21(4):1253–1278, 2000.

[3] Wolfgang Hackbusch and Stefan Kühn. A new scheme for the tensor representation. *Journal of Fourier Analysis and Applications*, 15(5):706–722, 2009.

[4] Xiaochen Han, Bo Wu, Zheng Shou, Xiao-Yang Liu, Yimeng Zhang, and Linghe Kong. Tensor FISTA-Net for real-time snapshot compressive imaging. In *AAAI*, pages 10933–10940, 2020.

[5] Victoria Hore, Ana Viñuela, Alfonso Buil, Julian Knight, Mark I McCarthy, Kerrin Small, and Jonathan Marchini. Tensor decomposition for multiple-tissue gene expression experiments. *Nature Genetics*, 48(9):1094, 2016.

[6] Fei Jiang, Xiao-Yang Liu, Hongtao Lu, and Ruimin Shen. Efficient multi-dimensional tensor sparse coding using t-linear combination. In *Thirty-Second AAAI Conference on Artificial Intelligence*, 2018.

[7] Vadim Lebedev, Yaroslav Ganin, Maksim Rakhuba, Ivan Oseledets, and Victor Lempitsky. Speeding-up convolutional neural networks using fine-tuned CP-decomposition. *ICLR*, 2015.

[8] Jiawei Ma, Xiao-Yang Liu, Zheng Shou, and Xin Yuan. Deep tensor ADMM-Net for snapshot compressive imaging. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 10223–10232, 2019.

[9] Ivan V Oseledets. Tensor-train decomposition. *SIAM Journal on Scientific Computing*, 33(5):2295–2317, 2011.

[10] Nico Vervliet, Otto Debals, Laurent Sorber, Marc Van Barel, and Lieven De Lathauwer. Tensorlab 3.0. *available online, URL: www. tensorlab. net*, 2016.

[11] Yuto Yamaguchi and Kohei Hayashi. Tensor decomposition with missing indices. In *IJCAI*, pages 3217–3223, 2017.

[12] Yinchong Yang, Denis Krompass, and Volker Tresp. Tensor-train recurrent neural networks for video classification. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3891–3900. JMLR. org, 2017.

[13] Yimeng Zhang, Xiao-Yang Liu, Bo Wu, and Anwar Walid. Video synthesis via transform-based tensor neural networks. In *ACM Multimedia*, 2020.

[14] Qibin Zhao, Guoxu Zhou, Shengli Xie, Liqing Zhang, and Andrzej Cichocki. Tensor ring decomposition. *arXiv preprint arXiv:1606.05535*, 2016.