cuTensor-HT: High Performance Third-order Hierarchical Tucker Tensor Decomposition on GPUs

Hao Huang[1] , **Tao Zhang**[1,3,*] , **Xiao-Yang Liu**[2]
[1]School of Computer Engineering and Science, Shanghai University, Shanghai, China
[2]Department of Electrical Engineering, Columbia University, USA
[3]Shanghai Institute for Advanced Communication and Data Science, Shanghai, China
baxlumen@shu.edu.cn, taozhang@shu.edu.cn, xl2427@columbia.edu

Abstract

Extracting effective information from large-scale multi-dimensional data has become a hot issue, where the hierarchical Tucker (HT) tensor decomposition is a widely used tool. However, the HT tensor decomposition is a computationally intensive task since the time complexity increases rapidly with the dimension and size of the tensor. In this paper, we implement the HT decomposition on the GPU, and propose optimization strategies to improve resource utilization, including efficient memory access, reducing computation consumption and batched operations. For tensors of various sizes, the optimized GPU implementation achieves $4.67\times$ speedups over the unoptimized GPU baseline.

Keywords— GPU, hierarchical Tucker tensor decomposition, parallelization

1 Introduction

In many real-world applications, there are high dimensional approximation problems, which are intractable when the dimension grows beyond 10 [1]. The major reason is that as the dimension grows, the storage space and computation will grow exponentially. Several methods based on approximation of low rank tensors are proposed and have important applications [2] [3] in many fields. For example, CANDECOMP/PARAFAC (CP) decomposition [4] is a simple low-rank approximation method, but it lacks of flexibility in applications. Tucker decomposition [5] can overcome the problem of insufficient flexibility by setting different ranks in different tensor modes. But in the case of high order tensors, the size of the core tensor generated by Tucker decomposition will also increase exponentially with the dimension. Hierarchical Tucker (HT) tensor decomposition [6] can perform well in the case of large-scale tensors. It has a high degree of parallelism [1], flexibility, and can effectively use the properties of low rank tensors, significantly reduce the amount of storage.

However, tensor decomposition workloads are compute-intensive and the computation grows rapidly with the size and dimensions of tensors. GPUs have higher throughput, memory access bandwidth, and better energy efficiency than CPUs [7], therefore they have been widely used in many computationally intensive tasks [8], [9]. Moreover, the high parallelism of HT decomposition is well suited for GPUs. In this paper, we map the algorithmic flow of the HT decomposition onto the GPU architecture. In addition, we propose optimization strategies based on the GPU algorithm and achieve significant acceleration.

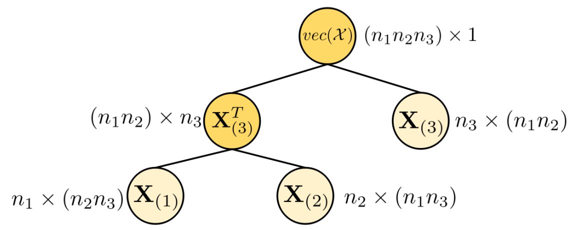
$vec(\mathcal{X})$ $(n_1n_2n_3)\times 1$
$(n_1n_2)\times n_3$ $\mathbf{X}_{(3)}^T$
$\mathbf{X}_{(3)}$ $n_3\times(n_1n_2)$
$n_1\times(n_2n_3)$ $\mathbf{X}_{(1)}$
$\mathbf{X}_{(2)}$ $n_2\times(n_1n_3)$

Figure 1: The binary tree generated by a third-order tensor.

The major contributions of this paper are as follows. First, we implement the HT decomposition on GPUs. Second, we propose optimization strategies for memory access, reducing the amount of calculation and improving resource utilization, thereby improving algorithm performance. Third, we conduct experiments to evaluate the performance of HT decomposition algorithm on GPUs and we achieved $4.67\times$ speedups over the unoptimized GPU baseline.

The remainder of this paper is organized as follows. Section 2 introduces the HT tensor decomposition algorithm. In section 3, we present the design, implementation and optimization of the HT tensor decomposition algorithm on GPUs. The experiment settings and results are presented in Section 4. Section 5 concludes this paper.

2 Hierarchical Tucker Decomposition

The hierarchical Tucker tensor decomposition is a typical special form of the tensor networks. The HT decomposition is introduced in [6], we briefly summarize the key concepts.

2.1 The Principle of HT Decomposition

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. For convenience of notation,

*Corresponding author: Tao Zhang. Tao Zhang is supported by Science and Technology Committee of Shanghai Municipality under grant No. 19511121002 and No. 19DZ2252600.

Algorithm 1 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: $\mathcal{B}_1, \mathcal{B}_2, \boldsymbol{U}_3, \boldsymbol{U}_2, \boldsymbol{U}_1$.
```

we assume that each element in the left child of a node in the binary tree is smaller than any element in the right child. There is only a single mode in all leaf nodes, and a set of modes is included in the non-leaf nodes. By observing the binary tree $\mathcal{T}$, we can know it satisfies that:

- All nodes are non-empty subsets of the mode set.
- The set of root node is $t_{root} = \{1, 2, 3\}$.
- The set of left and right children of each non-leaf node are disjoint.

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:

$$\mathcal{B}_t = \boldsymbol{U}_t \times_1 \boldsymbol{U}_{t_l}^T \times_2 \boldsymbol{U}_{t_r}^T, \tag{1}$$

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 left and right children of t, respectively. The $\times_1$ and $\times_2$ are tensor times matrix (TTM).

2.2 Overview of HT Decomposition Algorithm

The algorithm of HT decomposition is roughly consists of two steps. First, we perform matricization operations 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 (1) to calculate the transfer tensor $\mathcal{B}$ stored in the non-leaf node. The algorithm is described in Alg. 1 proposed by [6] and we set $r_t = 0.2 \times n_t$.

3 Parallel HT Decomposition on GPUs

As mentioned in [1], HT decomposition has strong parallelism. By organizing data and transplanting the algorithm to GPUs, we obtain the baseline (unoptimized) GPU implementation. To improve performance, we propose optimization strategies for efficient memory access, reducing computations and improving resource utilization. We implement these optimizations on GPUs and obtain the optimized GPU implementation.

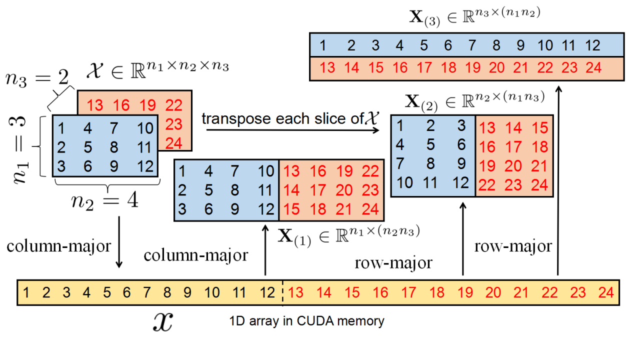
$\mathbf{X}_{(3)} \in \mathbb{R}^{n_3 \times (n_1 n_2)}$
$n_3 = 2$
$\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$
$n_1 = 3$
$n_2 = 4$
transpose each slice of $\mathcal{X}$
$\mathbf{X}_{(2)} \in \mathbb{R}^{n_2 \times (n_1 n_3)}$
$\mathbf{X}_{(1)} \in \mathbb{R}^{n_1 \times (n_2 n_3)}$
column-major
column-major
row-major
row-major
$\mathcal{X}$
1D array in CUDA memory

Figure 2: The diagarms of three modes of data stored in CUDA.

3.1 Design and Implementation

The main calculation of HT decomposition includes two parts. The first part is the SVD at lines 1-4 as described in Alg. 1 and the second part is the TTM at lines 6-7. In the process of updating the left singular vectors matrix, there is no dependency between the data used in the step and this feature is suitable for parallel execution on many-core GPUs.

3.1.1 Data Storage

In the GPU algorithm, all matrices and tensors are stored in memory in a column-major layout. During program execution, the data is always accessed in units of columns, which ensures continuous memory access. In addition, we use the official routines of NVIDIA CUDA, and the data of these routines is stored in columns-major format by default.

3.1.2 Implementation Details

The first part of the algorithm is to obtain the left singular vectors matrix $\boldsymbol{U}$. We use the routine gesvd$(\cdot)$ in the NVIDIA cuSOLVER library for this part. This routine requires that the number of rows in the input matrix is greater than the number of columns. However, the number of rows of the $\boldsymbol{X}_{(t)}$ is usually less than the number of columns. Therefore, we first perform QR decomposition on the matrix $\boldsymbol{X}_{(t)}$ and decompose the matrix into an orthogonal matrix $\boldsymbol{Q}$ and an upper triangular square matrix R, then use the routine gesvd$(\cdot)$ on the $\boldsymbol{R}$. After running this routine, we multiply the obtained left singular vectors matrices and $\boldsymbol{Q}$, and the result is the matrix $\boldsymbol{U}_t$ that we need.

The second part of the algorithm is to calculate $\mathcal{B}_2$, $\mathcal{B}_1$. This calculation process is a matrix multiplication operation. We use the matrix multiplication routine in NVIDIA CUBLAS to update.

3.2 Optimization Strategies

As mentioned above, SVD and TTM are the two most important parts of the GPU algorithm. The essence of the TTM calculation process is matrix multiplication, but it contains many matricization operations, which increases unnecessary time complexity. The performance of the baseline algorithm is not perform well, primarily due to the low resource utilization. We optimize the GPU algorithm on three aspects: 1) memory access; 2) reducing computations; 3) improving resource utilization by batching multiple operations.

Algorithm 2 Optimized hierarchical Tucker tensor decomposition on GPUs

```
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: for $i = 1$ to 3 do
2:     $\boldsymbol{H}_i = \boldsymbol{X}_{(i)} \boldsymbol{X}_{(i)}^T$
3: end for
4: for $j = 1$ to 3 in parallel do
5:     $\boldsymbol{U}_j \leftarrow$($r_j$ leading eigen vectors) of $\boldsymbol{H}_j$
6: end for
7: $\boldsymbol{U}_4 \leftarrow$($r_4$ leading left singular vectors) of $\boldsymbol{X}_{(3)}^T$
8: $\mathcal{B}_2 = \mathcal{U}_4 \times_1 \boldsymbol{U}_1^T \times_2 \boldsymbol{U}_2^T$,
9: $\mathcal{B}_1 = \mathcal{X} \times_1 \boldsymbol{U}_4^T \times_2 \boldsymbol{U}_3^T$,
Output: $\mathcal{B}_1, \mathcal{B}_2, \boldsymbol{U}_3, \boldsymbol{U}_2, \boldsymbol{U}_1$.
```

3.2.1 Reducing Computations

In the part of performing TTM calculations at lines 6-7 in Alg. 1, we need to convert the tensor into a matrix according to mode-1, and then convert it back to a tensor after matrix multiplication. After that, repeat this process according to mode-2. There are eight times matricization. Obviously, matricization increases the computational complexity of this part. In CUDA, matrices and tensors are in column-major, and stored in memory as a one-dimensional array. As shown in Fig. 2, we get the matrix $\boldsymbol{X}_{(t)}$ by different ways of organizing numbers, so we avoid the matricization. For a tensor $\mathcal{X} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, the $\boldsymbol{X}_{(1)}$ only needs to take data from memory according to $n_1 \times (n_2 n_3)$; transpose the front and back parts of $\boldsymbol{X}_{(1)}$ separately to get $\boldsymbol{X}_{(2)}$; get $\boldsymbol{X}_{(3)}$ by fetching data according to $n_3 \times (n_1 n_2)$ with row-major. This process is easy to implement in the multiplication function library CUBLAS, thus avoiding the extra calculation brought by matricization.

3.2.2 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 $\boldsymbol{X}_{(t)}^T$ to multiply, 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, as shown in Fig. 3, 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.

The pseudo code of the optimized HT tensor decomposition algorithm on GPUs is described in Alg. 2. The $\boldsymbol{X}_{(t)}$ is directly obtained according to the above optimization method. The matrix multiplication at line 2 will reduce accuracy if executed in parallel, and the time complexity will not be significantly reduced. Lines 4-6 are batch eigen decomposition and get larger r_t eigenvalues. When calculating $\boldsymbol{U}_4$ at line 7, we use the routine gesvda$(\cdot)$, which solves the problem of excessive matrix rows. The $\mathcal{U}_4$ is directly obtained from the $\boldsymbol{U}_4$ without calculation, as shown in Fig. 2, at line 8.

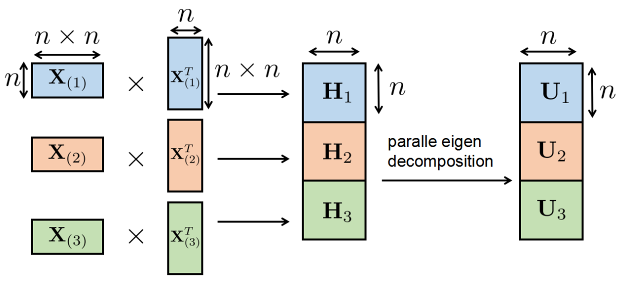

Figure 3: Batch processing in GPUs.

3.2.3 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 Evaluation

All experiments were performed on dual Intel Xeon E5-2640 V4 CPUs and an NVIDIA Tesla V100 GPU, respectively. Each CPU has 10 cores running at 2.4GHz and supports 20 threads through hyper-threading technology. The GPU has 32GB device memory and 5120 CUDA cores. The CPU algorithm uses Matlab 2017b htcuker toolbox to complete and the GPU algorithm run on CUDA 10.1.

We generate tensors of different sizes for evaluation. These tensors are obtained by multiplying several matrices according to the properties of low rank. We use running time and relative squared error as performance evaluation metrics. For the running time, we will execute HT decomposition on CPU and GPU separately and record the running time. The speedup is calculated as (CPU running time) / (GPU running time). The error is defined as:

$$\text{RSE} = \|\mathcal{X} - \widehat{\mathcal{X}}\|_F / \|\mathcal{X}\|_F,$$

where $\mathcal{X}$ is the original tensor and $\widehat{\mathcal{X}}$ is the tensor recovered by multiplying by the factor matrix.

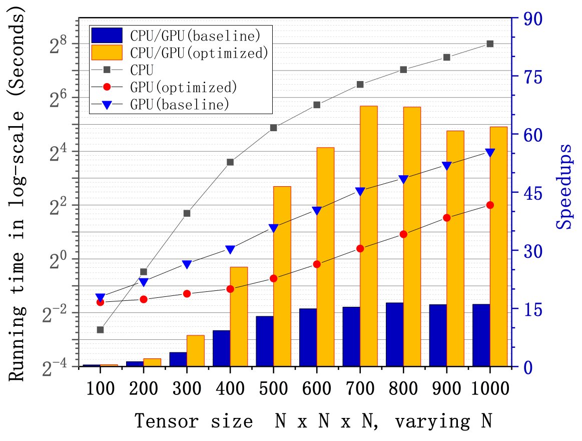

Figure 4: Running time and speedups of HT decomposition.

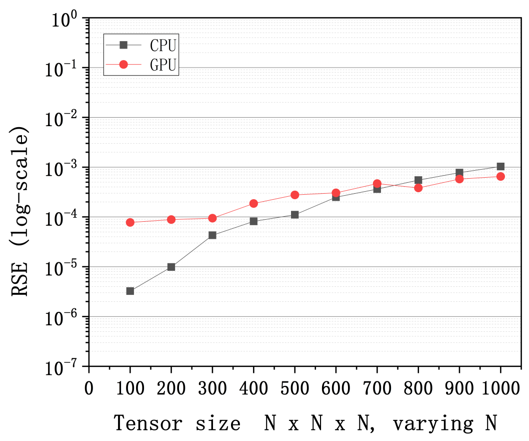

Figure 5: Error of HT decomposition.

We performed performance evaluations on CPUs and the GPU separately, and the maximum tensor size is $1000 \times 1000 \times 1000$, which also reached the limit of device memory. The data here is all single-precision. Three versions of the algorithm are executed for 5 times for comparison: the CPU algorithm, the baseline GPU algorithm, and the optimized GPU algorithm.

Fig. 4 shows the running time and speedups. As the size increases, the acceleration effect will get better and better. The baseline GPU algorithm achieves an average of $6.52\times$ and up to $9.43\times$ speedups versus two Xeon CPUs. In contrast, the optimized GPU algorithm achieves an average of $39.59\times$ and up to $67.21\times$ speedups versus two CPUs.

Fig. 5 shows the performance of the two platforms in terms of error. The error of the GPU algorithm differs from the CPU algorithm by an order of magnitude when the tensor size is small. Then as the size increases, the accuracy of the two differs by no more than an order of magnitude. The error of the GPU algorithm is 3.11×10^{-4} on average However, the error of the CPU algorithm is 3.21×10^{-4} on average and it grows faster as the increase of the tensor size, and when the scale is $1000 \times 1000 \times 1000$, the error reaches 0.00103.

5 Conclusion

In this paper, we analyzed the principle of HT decomposition, designed GPU-based optimization strategies, and evaluated the performance of the GPU algorithm through experiments. In addition, significant acceleration has been achieved on the GPU platform:up to $4.67\times$ speedups over the unoptimized GPU algorithm. In terms of error, the average error of the GPU algorithm is 3.11×10^{-4}, while the error of the CPU algorithm increased to 1×10^{-3} in the case of large-scale tensors. Our future work will incorporate this implementation into the cuTensor library [10].

References

[1] Grasedyck and Lars, "Hierarchical singular value decomposition of tensors," *SIAM Journal on Matrix Analysis & Applications*, vol. 31, no. 4, pp. 2029–2054, 2009.

[2] Y. Zhang, X.-Y. Liu, B. Wu, and A. Walid, "Video synthesis via transform-based tensor neural networks.," in *ACM Multimedia*, 2020.

[3] X. Han, B. Wu, Z. Shou, X.-Y. Liu, Y. Zhang, and L. Kong, "Tensor FISTA-Net for real-time snapshot compressive imaging.," in *AAAI*, pp. 10933–10940, 2020.

[4] 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.

[5] L. R. Tucker, "Implications of factor analysis of three-way matrices for measurement of change," *Problems in Measuring Change*, vol. 15, pp. 122–137, 1963.

[6] W. Hackbusch and S. Kühn, "A new scheme for the tensor representation," *Journal of Fourier Analysis and Applications*, vol. 15, no. 5, pp. 706–722, 2009.

[7] T. Zhang, J. Zhang, W. Shu, M.-Y. Wu, and X. Liang, "Efficient graph computation on hybrid CPU and GPU systems," *The Journal of Supercomputing*, vol. 71, no. 4, pp. 1563–1586, 2015.

[8] T. Zhang, X.-Y. Liu, and X. Wang, "High performance GPU tensor completion with tubal-sampling pattern," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 7, pp. 1724–1739, 2020.

[9] T. Zhang, H. Lu, and X.-Y. Liu, "High-performance homomorphic matrix completion on multiple GPUs," *IEEE Access*, vol. 8, pp. 25395–25406, 2020.

[10] T. Zhang, X.-Y. Liu, X. Wang, and A. Walid, "cuTensor-tubal: Efficient primitives for tubal-rank tensor learning operations on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 3, pp. 595–610, 2020.