An Optimized Tensor Completion Library for multiple GPUs

Ming Dun[★], Yunchun Li[★†], Hailong Yang[†], Qingxiao Sun[†], Zhongzhi Luan[†] and Depei Qian[†]
[★]School of Cyber Science and Technology, Beihang University, Beijing, China
[†]School of Computer Science and Engineering, Beihang University, Beijing, China
{dunming0301,lych,hailong.yang,qingxiaosun,07680,depeiq}@buaa.edu.cn

ABSTRACT

Tensor computations are gaining wide adoption in big data analysis and artificial intelligence. Among them, tensor completion is used to predict the missing or unobserved value in tensors. The decomposition-based tensor completion algorithms have attracted significant research attention since they exhibit better parallelization and scalability. However, existing optimization techniques for tensor completion cannot sustain the increasing demand for applying tensor completion on ever larger tensor data. To address the above limitations, we develop the first tensor completion library *cuTC* on multiple Graphics Processing Units (GPUs) with three widely used optimization algorithms such as alternating least squares (ALS), stochastic gradient descent (SGD) and coordinate descent (CCD+). We propose a novel TB-COO format that leverages warp shuffle and shared memory on GPU to enable efficient reduction. In addition, we adopt the auto-tuning method to determine the optimal parameters for better convergence and performance. We compare *cuTC* with state-of-the-art tensor completion libraries on real-world datasets, and the results show *cuTC* achieves significant speedup with similar or even better accuracy.

CCS CONCEPTS

• **Computer systems organization** → *Parallel architectures*; • **Computing methodologies** → *Parallel algorithms.*

KEYWORDS

tensor completion, performance optimization, GPU

ACM Reference Format:
Ming Dun[★], Yunchun Li[★†], Hailong Yang[†], Qingxiao Sun[†], Zhongzhi Luan[†] and Depei Qian[†]. 2021. An Optimized Tensor Completion Library for multiple GPUs. In *2021 International Conference on Supercomputing (ICS '21), June 14–17, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3447818.3460692

1 INTRODUCTION

The multi-way data, also known as *tensor*, has attracted increasing attention in the fields of big data analysis, computer vision and artificial intelligence for its ability to represent interactions among massive variables [37]. Among the tensor computations, tensor completion is one of the most popular problems for research, which is the generalization of the prevailing matrix completion [13]. One important usage of tensor completion is to estimate or restore the unobserved values of a tensor. For instance, since the RGB images are three-dimensional tensors, the tensor completion algorithm can be applied to image completion and image retrieval for building facade images [44]. In addition, tensor completion can be used to help clinical analysis by mining the underlying patterns from massive health records [46]. Moreover, tensor completion can also be applied to predict the preference for new products based on historical customer ratings [34]. Furthermore, tensor completion algorithms can be adopted to predict climatic parameters for different locations or future time [24], which are all unobserved elements in climate records. All the above applications depend on the performance of tensor completion to satisfy the computation demand for processing ever-increasing volume of tensor data.

Tensor completion algorithms can be mainly divided into two categories [41]: decomposition-based approach and trace-norm based approach. In general, the decomposition-based approach receives more attention in the HPC community due to its better parallelization and scalability. Tensor decomposition factorizes a tensor to low-rank representations, which can ease the understanding of the relationships among variables. Whereas, the decomposition-based tensor completion first conducts the tensor factorization based on observed elements in tensor to generate low-rank representations, and then estimates the unobserved elements with factor matrices. The difference between tensor decomposition and decomposition-based tensor completion is that the former considers the unobserved elements as zeros, while the latter attempts to predict the exact values of missing elements.

Recently, accelerating decomposition-based tensor completion has been actively studied [37, 52]. Researchers have been making efforts to optimize tensor completion on multicore CPUs [37] and manycore CPUs (e.g., Intel Knights Landing) [52]. Meanwhile, GPU has become an attractive architecture for improving the performance of linear algebra libraries [26], scientific applications [23] and artificial intelligence [40]. The GPU exceeds traditional CPU in several aspects such as massive parallelism, higher memory bandwidth and higher peak floating-point performance. The above promising properties of GPU make it a profitable candidate for further boosting the performance of tensor completion algorithms, which has not yet been covered by existing research work. Moreover, modern servers are commonly equipped with multiple GPUs (e.g., 8 GPUs within Nvidia DGX-1 [10]). To take advantage of such servers, it is inevitable to parallelize tensor completion across multiple GPUs for exploiting the performance on table.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ICS '21, June 14–17, 2021, Virtual Event, USA
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8335-6/21/06…$15.00
https://doi.org/10.1145/3447818.3460692

To bridge the gap of providing efficient tensor completion on GPUs, there are several unique challenges to be addressed. Firstly, the unpredictable sparsity pattern in tensor can lead to severe load imbalance and poor cache locality, which is detrimental to performance. Secondly, the formidable memory bandwidth for accessing the massive data when updating the factor matrices also deteriorates the computation efficiency. Thirdly, tensor completion requires more computation and communication resources than traditional tensor decomposition, especially when using Alternating Least Squares (ALS) optimization algorithm. It involves additional batched matrix operations except for the computation hotspot of tensor decomposition, such as matricized tensor times Khatri-Rao product (MTTKRP). Therefore, new optimization approaches for tensor completion need to be developed in order to better adapt to the GPU architecture. Moreover, when scaling to multiple GPUs with discrete memory addresses, a more sophisticated data partition strategy is needed to avoid load imbalance among GPUs, and to reduce data transfers when updating the factor matrices in each computation epoch.

In this paper, we exploit the performance potential of GPUs to accelerate tensor completion by developing a library *cuTC*. In *cuTC*, to facilitate our optimizations on GPUs, we propose a new sparse tensor storage format TB-COO by extending the coordinate format (COO) format with tile and bitmap. Based on TB-COO, we optimize three widely used algorithms in tensor completion on GPUs, including alternating least squares (ALS), stochastic gradient descent (SGD) and coordinate descent (CCD+). Leveraging the efficient design of TB-COO and unique GPU characteristics such as share memory, hierarchical parallelization and warp shuffle, we are able to achieve significant performance speedup compared to the state-of-the-art libraries. To the best of our knowledge, this is the first work to develop an efficient tensor completion library targeting multiple GPUs.

Specifically, this paper makes the following contributions:

- We propose a new sparse tensor storage format TB-COO by extending COO format with tiles and bitmaps. The TB-COO format improves load balance by tiling tensors into entries and eliminates expensive atomic operations by enabling reduction through bitmaps.
- We optimize three commonly used algorithms in tensor completion leveraging the efficient design of TB-COO and unique GPU characteristics. In addition, we design an auto-tuning scheme to determine the parameter settings for optimal performance and algorithm convergence.
- We develop the first tensor completion library *cuTC* on multiple GPUs, and evaluate it on both synthesized and real-world datasets. The experiment results demonstrate *cuTC* achieves significant performance speedup compared to the state-of-the-art libraries. In addition, we analyze the efficiency of *cuTC* through roofline analysis.

The rest of the paper is organized as follows. Section 2 introduces the tensor notations used in this paper and provides a concise description of decomposition-based tensor completion, popular optimization algorithms in tensor completion as well as the existing sparse tensor storage formats. We present the TB-COO format and optimization approaches for ALS, SGD and CCD+ algorithms on GPU in Section 3. The performance results compared to the state-of-the-art libraries are presented in Section 4. Section 5 presents the existing researches related to decomposition-based tensor completion. Section 6 concludes this paper.

2 BACKGROUND

2.1 Decomposition-Based Tensor Completion

Before diving into the details of decomposition-based tensor completion, we will provide the preliminaries and notations for tensors. The multi-dimensional array is commonly denoted as the tensor [19], whose dimensions are denoted as *modes*. Tensor is the generalization of matrices and vectors. Moreover, a *fiber* of the tensor is one of its subarrays, which keeps all but one index of the tensor to be constant, whereas a *slice* of the tensor is one of its subarrays that fixes all but two indices. The important notations and their definitions are summarized in Table 1. To keep the illustration concise, we focus on the three-dimensional tensor and the mode-1 operations to describe the tensor storage format and parallel tensor completion algorithms without loss of generality (the same approach can be applied to the operations on other modes).

Table 1: Key Notations

Notation	Definition
$\mathcal{X}$	A high-dimensional tensor.
$\mathcal{X}_{i,j,k}$	An element in a high dimensional tensor.
$\mathcal{X}_{i,:,:}$	A slice in a high dimensional tensor.
$\mathcal{X}_{i,j,:}$	A fiber in a high dimensional tensor.
$\mathbf{A}$	A matrix.
$\mathbf{A}_{i,j}$	An element in a matrix.
$\mathbf{a}$	An vector.
a_i	An element in a vector.
$\odot$	The symbol for Kronecker product.
$*$	The symbol for Hadamard product.
$\dagger$	The symbol for pesudo-inverse.

Canonical polyadic decomposition (CPD) is one of the most popular methods for tensor decomposition [37] due to its high computation efficiency. Given a three-dimensional tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ and rank F, the CPD algorithm generates three factor matrices to model the tensor: $\mathbf{A} \in \mathbb{R}^{I \times F}$, $\mathbf{B} \in \mathbb{R}^{J \times F}$ and $\mathbf{C} \in \mathbb{R}^{K \times F}$. In other words, the CPD algorithm constructs $\mathcal{X}$ as the summation of F rank-one tensors. The element-wise description of CPD is defined as $\mathcal{X}_{i,j,k} = \sum_{f=1}^{F} a_{if} b_{jf} c_{kf}$.

Decomposition-based tensor completion is the most widely used approach for scalable tensor completion [41]. Particularly, the tensor completion using CPD can be written as a non-convex optimization problem as shown in Equation 1, where the $\mathcal{X}_{:,:,:}$ only consists of the observed values, λ is the regularization parameter and $\mathcal{L}(\cdot)$ is the loss function which can be computed using $\mathcal{L}_{i,j,k} = \mathcal{X}_{i,j,k} - \sum_{f=1}^{F} a_{if} b_{jf} c_{kf}$. The adoption of regularization parameter is to prevent overfitting. To solve this non-convex optimization problem, three algorithms have gained wide popularity, including Alternating Least Square (ALS), Stochastic Gradient Descent (SGD) and Coordinate Descent (CCD+).

$$\min_{\mathbf{A},\mathbf{B},\mathbf{C}} \frac{1}{2}\sum_{\mathcal{X}_{:,:,:}} \mathcal{L}^2_{i,j,k} + \frac{\lambda}{2}(\|\mathbf{A}\|_F^2 + \|\mathbf{B}\|_F^2 + \|\mathbf{C}\|_F^2) \quad (1)$$

2.2 Alternating Least Square (ALS)

During each epoch, the ALS algorithm cyclically updates one factor matrix with all the other factor matrices fixed. For instance, the update for i^{th} row of mode-1 matrix $\mathbf{A}$ can be calculated using Equation 2, where $\text{vec}(\mathcal{X}_{i,:,:})$ is the vectorization of the i^{th} slice of $\mathcal{X}$. Meanwhile, $\mathbf{H}$ is a buffer matrix for storing the result of Hadamard product and $\mathbf{H}_i \in \mathbb{R}^{|\text{vec}(\mathcal{X}_{i,:,:})|\times F}$ stores all the corresponding Hadamard products in the i^{th} slice of $\mathcal{X}$. If the n^{th} element in $\text{vec}(\mathcal{X}_{i,:,:})$ is $\mathcal{X}_{i,j,k}$, then $\mathbf{H}_{l,:}$ can be computed as $\mathbf{B}_{j,:} * \mathbf{C}_{k,:}$. It is obvious that $\mathbf{H}_i^T\mathbf{H}_i + \lambda\mathbf{I}$ is symmetric positive-definite, thus its inversion can be calculated through a Cholesky factorization and forward/backward substitutions. The computation of all $\mathbf{H}_i$ is in the complexity of $O(F^2 nnz)$, where nnz is the number of non-zero elements in tensor $\mathcal{X}$. Moreover, calculating a row for matrix inversions is in the complexity of $O(F^3)$, thus updating the factor matrix $\mathbf{A}$ in ALS is in the complexity of $O(F^2 nnz + IF^3)$.

$$\mathbf{A}_{i,:} = (\mathbf{H}_i^T\mathbf{H}_i + \lambda\mathbf{I})^{-1}\mathbf{H}_i^T\text{vec}(\mathcal{X}_{i,:,:}) \quad (2)$$

2.3 Stochastic Gradient Descent (SGD)

During each epoch, the SGD algorithm takes several steps, each of which updates the factor matrices based on the gradient at a randomly-selected element $\mathcal{X}_{i,j,k}$. The updates are shown in Equation 3, where η denotes the learning rate parameter. One popular parallel implementation of SGD is to exploit the independence between non-zero values with disjoint coordinates [8]. For instance, partitioning the tensor $\mathcal{X}$ into blocks along the diagonal. Moreover, researchers have been devoting to design parallel strategies, which allows the parallelization of elements with overlapped coordinates. Hogwild [32] enables the shuffled non-zeros to be processed in parallel without stratification and synchronization, based on the observation that the input is sparse and conflict rarely happens. ASGD [8] processes all the non-zeros in a distributed manner and combines the local updates with weighted summation during each epoch.

$$\mathbf{A}_{i,:} = \mathbf{A}_{i,:} + \eta(\mathcal{L}_{i,j,k}\mathbf{B}_{j,:} * \mathbf{C}_{k,:} - \lambda\mathbf{A}_{i,:}), \quad (3)$$

2.4 Coordinate Descent (CCD+)

In contrast to ALS and SGD, the CCD+ sequentially updates the columns of factor matrices based on $\mathbf{A}_{if} = \frac{\alpha_i}{\lambda+\beta_i}$, where the α_i and β_i is formulated as $\alpha_i = \sum_{\mathcal{X}_{i,:,:}} \mathcal{L}_{i,:,:}\mathbf{B}_{jf}\mathbf{C}_{kf}$ and $\beta_i = \sum_{\mathcal{X}_{i,:,:}} (\mathbf{B}_{jf}\mathbf{C}_{kf})^2$, respectively. The f^{th} columns in $\mathbf{B}$ and $\mathbf{C}$ are updated after that in $\mathbf{A}$ is updated. To improve the performance of parallel CCD+ algorithm, the loss function $\mathcal{L}(\cdot)$ is only computed once in each epoch and reused for F columns [50].

2.5 Sparse Tensor Storage Format

Recently, several sparse tensor storage formats have been developed to improve the performance and reduce memory consumption of sparse tensor decomposition. These formats can be primarily divided into two categories: COO-based formats and CSF-based formats. Since the CSF format [36] has a tree-like structure, the CPD algorithm with CSF format is implemented recursively and does not fit for GPU architecture [22]. Therefore, the new tensor storage format (Sec 3.1) proposed in this paper is based on COO, and we will focus on the discussion of COO-based formats. The illustration of COO-based formats is shown in Figure 1. COO [17] format directly stores the indices and the values of non-zero elements in a tensor, which suffers from massive memory consumption.

When computing the new rows for the factor matrix $\mathbf{A}$, only rows in matrices $\mathbf{B}$ and $\mathbf{C}$ are needed. Based on the above observation, F-COO [22] only stores the indices in mode-2 and mode-3 to reduce memory consumption. Furthermore, the start flag (sf) and bit flag (bf) arrays are used to represent the variation of mode-1 indices of non-zero elements. The sf array is compressed to unsigned int with 32 bit and the bf array is with $uint8\_t$ to further reduce memory footprint. For CPD, zero in bf array denotes change of mode-1 indices of an element, and one in sf array denotes change of mode-1 indices of a block. Moreover, F-COO can enable segment scan [48] to eliminate atomic operations. However, it is difficult to retrieve the mode-1 indices with F-COO since they are not provided explicitly.

HiCOO [21] is another variant of COO format. To convert a tensor from COO to HiCOO, the non-zero elements are sorted in Z-Morton order, and then the sorted elements are partitioned to blocks according to the given block size B. The block pointers are stored in $bptr$ array. Besides, the block indices are denoted as bi, bj and bk, whereas the element indices are denoted as ei, ej and ek, respectively. The original indices in COO can be calculated as $i = bi * B + ei$, $j = bj * B + ej$ and $k = bk * B + ek$. HiCOO is mode-generic, and able to reduce memory consumption through advanced compression. In addition, the atomic operations can be eliminated by the privatization method. However, the complicated structure of HiCOO format makes it inefficient when applied on GPU.

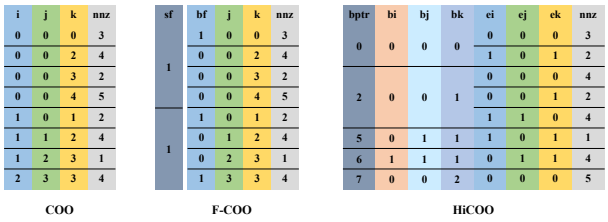


Figure 1: The illustration of COO-based sparse tensor storage formats.

3 METHODOLOGY AND IMPLEMENTATION

In this section, we introduce the novel sparse tensor storage format TB-COO on GPU. In addition, we provide the methodology and implementation details for the three optimization algorithms for tensor completion: ALS, SGD and CCD+. Moreover, we present the auto-tuning approach for further improving the performance and convergence of our tensor completion library *cuTC*.

3.1 TB-COO Storage Format

Figure 2 shows the design of our proposed TB-COO format, where T and B stands for *Tile* and *Bitmap*, respectively. To convert a tensor from COO format to TB-COO format, firstly the tensor needs to be sorted along mode-1 indices. There are only two arrays in TB-COO, which are named as d and *entries*, respectively. The d array stores the mode-1 indices for non-zero slices. Meanwhile, the *entries* array stores the tiled non-zero elements linearly, which are packed in entries of length T. There is a tuple containing three elements at the head of each entry, where the first element sp and the second element lp are the range of pointers to d array, whereas the third element bm is a bitmap for computing the pointer to d array for each element in the entry. An additional "1" is added at the head of bm to ensure the variance of mode-1 indices is stored properly. If the i^{th} bit in bm is 0, then the mode-1 index of the $(i-1)^{th}$ element is different from the $(i)^{th}$ element, and vice versa. The rest of a entry stores the mode-2 and mode-3 indices as well as values. The sp and lp are multiplied by -1 to distinguish the indices in the same entry. The pointer to mode-1 indices for non-zero elements can be easily and efficiently retrieved by bit operations as $p_i = sp + i + 1 - gap$ and $gap = \text{popc}(\text{brev}(bm) >> (\text{clz}(bm))) << (63 - i)$, where $brev$ is the function for reversing integers, clz is the function for deriving the position of the first bit with value 1 in an integer, $popc$ is the function for counting number of bit 1 in an integer and i is the position of elements in the entry.

There are several advantages of applying TB-COO in tensor completion on GPUs. Firstly, the TB-COO partitions the non-zero elements evenly in the size of T, which can improve the load balance. Secondly, since the entries in *entries* array are in the same size, it is convenient for threads to compute the position of the entries assigned to them. Moreover, the bitmap in entries can serve as communication signals to enable efficient reduction utilizing warp shuffle mechanism [12] on GPU. Warp shuffle enables threads within the same warp on GPU to read the registers of each other using a single instruction, which is faster than shared memory. Meanwhile, compared to F-COO, the bitmap is more compact (e.g., there are two bitmap arrays in F-COO), and it is much easier for threads to fetch the mode-1 indices through the sp and lp, whereas there are no arrays for mode-1 indices in F-COO, which is inefficient during matrix update process. In addition, compared to tree-based CSF format, TB-COO can better fit GPU architecture since TB-COO does not rely on recursive algorithms for MTTKRP, and its tiles are more friendly for GPU threads. Last but not least, compared to the original COO, the TB-COO consumes less memory. Given a tensor $\mathcal{X}$ with I non-zero slices and nnz non-zero elements, if all indices and values are in the length of eight bytes, the COO format requires $32nnz$ bytes, whereas TB-COO only requires $24(nnz + \frac{nnz}{T}) + 8I$. With $I \ll nnz$ in sparse tensors, TB-COO format can effectively reduce the memory consumption.

3.2 Optimizing ALS

There are three sub-procedures during the update of factor matrices: MTTKRP, computing $\mathbf{H}^T\mathbf{H}$ and solving matrix equations. We generate the $\mathbf{H}^T\mathbf{H}$ and the results of MTTKRP simultaneously by accessing the tensor data only once. The processing logic of computing the MTTKRP and update $\mathbf{H}^T\mathbf{H}$ is shown in Algorithm 1. The

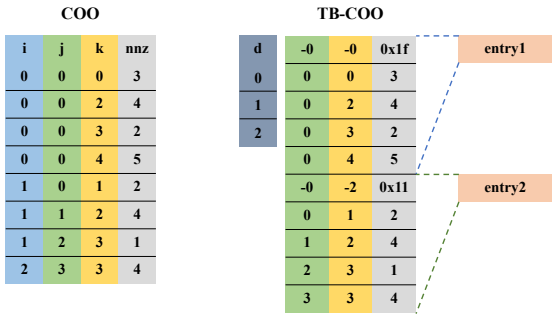


Figure 2: The design of TB-COO sparse tensor storage formats.

warp size W is set to an integer no more than 32 because the warp shuffle mechanism supports user-defined warp size with the upper bound of 32. Each warp is assigned with one tile in $\mathcal{X}$. When the number of non-zero elements nnz of $\mathcal{X}$ is indivisible by T, there will be some inactive threads in the last warp. These inactive threads are omitted using *warpmask* which is generated by function *warpmask_gen* to guarantee correctness (line 6). The *warpmasks* is stored in shared memory to reduce access latency. Each GPU thread first gets the indexing information for its tile to be processed, including sp, lp and bm obtained from function *get_tile_attribute* (line 7). The pointer to d array can be calculated as described in Section 3.1 in function *get_myid* (line 8). Next, it fetches the non-zero elements and conducts the Hadamard product with function *get_mydata* (line 10) and *my_hadamard* (line 12), respectively. The result of Hadamard product is stored in *mylocalmbuf*. After obtaining the results of Hadamard product, the new partial results in $\mathbf{H}^T\mathbf{H}$ are calculated (line 15). Since $\mathbf{H}^T\mathbf{H}$ is a symmetric positive-definite matrix, we only compute the upper triangular matrices to reduce computation. Meanwhile, the partial results of MTTKRP are also generated. Then the threads within the same warp execute the inter-warp reduction with function *warp_reduction* (line 19) before storing the results in global memory, which reduces the need for atomic add operations.

The illustration of inter-warp signal based reduction algorithm for ALS is shown in Figure 3. The assumption of this algorithm is that the mode-1 pointers or indices need to be continuous, which is satisfied in TB-COO format. The iteration time is the longest sequence of consecutive 1 bits in bm without counting the first bit. To simplify analysis for the longest sequence of consecutive 1 bits, the iteration time can be set to the total number of 1 bits, which does not affect the final results. The *SelfBm* denotes the differences of mode-1 indices and can be calculated based on $SelfBm_i = (\text{brev}(bm) >> (\text{clz}(bm)) >> (i+1))\&1$. Besides, the *SelfBm* in lane zero is always set to zero. We utilize *SelfBm* as a signal and a data activator. There are two phases in each iteration. During the first phase, data in lane i is transferred to lane $i-1$ through warp shuffle, and the data will be set to 0 if *SelfBm* of lane i is 0. In the second phase, the original data in lane i will be discarded if *SelfBm* of lane i is 1, in order to avoid repetitive summation. After all iterations, the thread with *SelfBm* = 0 stores the final results into global memory,

which eliminates the expensive atomic operations. If the longest sequence of consecutive 1 bits in the bm is N, the inter-warp reduction algorithm is in complexity of $O(log_2 N)$.

Algorithm 1 Calculating MTTKRP and $\mathbf{H}^T\mathbf{H}$ Update

```
1:  Input: Tensor X, tile sizeT, rank R, warp size W
2:  Output: H^T H for nonzero slices, MTTKRP results
3:  for each thread do
4:      laneid = threadid mod W
5:      warpid = threadid / W
6:      warpmask_gen()
7:      (sp, lp, bm) = get_tile_attribute(warpid)
8:      myfid = get_myid(sp, bm, laneid)
9:      mybm = get_mybm(bm, landid)
10:     (b, c, val) = get_mydata(X, T, blockid, warpid, laneid)
11:     itercounter = get_itercounter(bm)
12:     mylocalmbuf = my_hadamard(b, c, R)
13:     for m = 1 → R do
14:         for j = 1 → m do
15:             newhth = mylocalmbuf[m] * mylocalmbuf[j]
16:             warp_reduction(newhth, mybm, W, mywarpmask)
17:         end for
18:         newm = val * mylocalmbuf[m]
19:         warp_reduction(newm, mybm, W, mywarpmask)
20:     end for
21: end for
```

Figure 3: The illustration of inter-warp reduction for ALS.

We utilize the cuSOLVER [29], a linear algebra library based on cuBLAS and cuSPARSE, to conduct the final process of factor matrix update, including batched Cholesky decomposition and batched matrix equations with patterns of $\mathbf{Ha} = \mathbf{b}$.

When scaling to multiple GPUs, we propose the tensor blocking strategy as shown in Figure 4. We develop a two-level blocking mechanism for ALS algorithm to partition the tensor along with all modes. Firstly, the sparse tensor is partitioned evenly into blocks, and each block contains several non-zero slices with approximately equal number of non-zero elements. Then the *blocks* are converted to TB-COO formats, and each GPU will update factor matrices with the tiles in its assigned *blocks*. The CPU stores the ranges of slices in each *blocks*, which are used to guide the matrix exchanging between GPUs and CPU, as well as between GPUs. After calculating the tiles, GPUs need to exchange the updated tiles with each other. We use the NVIDIA Collective Communications Library (NCCL) for communication between GPUs. The synchronization of factor matrices are conducted by using the *AllReduce* primitive in NCCL.

To improve the convergence of ALS algorithm, we utilize the randomization strategy proposed in [37]. Instead of updating the factor matrices cyclicly, the processing order of each mode is reshuffled at the beginning of each epoch.

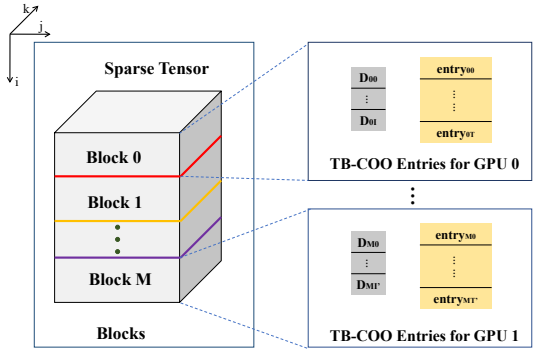


Figure 4: The illustration of tensor blocking strategy on multiple GPUs for ALS and CCD+ algorithms.

3.3 Optimizing CCD+

To accelerate CCD+, similar tensor blocking and communication strategies used in ALS are applied, where all modes are partitioned. We adopt a two-level partitioning mechanism in CCD+ as illustrated in Figure 4. Besides, we adopt a multi-tensor strategy to further optimize the performance, which is also adopted in previous work for tensor decomposition [38] and tensor completion [37]. There are N tensors and the n^{th} tensor is for updating the n^{th} factor matrix. Meanwhile, there are two sub-processes when updating the columns in CCD+ as illustrated in Section 2.4. One sub-process computes the numerators and denominators, and the other one conducts the division to generate new elements for factor matrices. During the first sub-process, it performs calculation using the signal based inter-warp reduction algorithm shown in Algorithm 1, where each warp is assigned with a *tile* in TB-COO format and the threads within the same warp aggregate their partial results after calculating their numerators and denominators. During the second sub-process, each thread performs the division for a row in the factor matrix.

Moreover, to further improve the convergence of CCD+ algorithm, we adopt the randomization mechanism in [37]. At each iteration, we conduct a reshuffle to the modes to decide their execution order.

3.4 Optimizing SGD

For SGD algorithm, we optimize the ASGD [8] algorithm, where all non-zero elements are processed based on equations described in Section 2.3, with weights added. Unlike the ALS and CCD+ algorithm, SGD algorithm adopts the single tensor strategy, which means that only one tensor needs to be generated during the update procedure along all modes. Moreover, unlike ALS and CCD+, SGD cannot be paralleled among slices as shown in Section 2.3, thus we develop a different blocking strategy for it when scaling to multiple GPUs. Since the non-zero elements are used to update three rows in factor matrices simultaneously, the blocking strategy we adopted to optimize SGD algorithm across multiple GPUs is illustrated in Figure 5. After the sparse tensor is converted to TB-COO format,

the entries in the formats are evenly partitioned among M GPUs, where each entry partition is named as a *band*. When each GPU finishes the update of its own *band*, they exchange the updated matrices with each other before the next iteration. We design special buffers on each GPU to store the partial results of factor matrices received from other GPUs. The data synchronization between GPUs is performed using AllReduce primitive in NCCL.

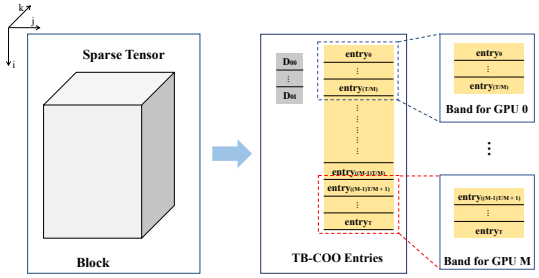


Figure 5: The illustration of tensor blocking strategy on multiple GPUs for SGD algorithm.

3.5 Auto-tuning

For implementing our *cuTC* library that includes the ALS, CCD+ and SGD optimization algorithms using TB-COO format, there are several parameters that can directly influence the convergence and performance of tensor completion. Based on our empirical study, the parameters affecting convergence include regularization parameter λ ($0.01 < \lambda < 500$, and $\lambda \in R$) for all three algorithms as well as learning rate η ($0 < \eta < 1$, and $\eta \in R$) and weights w for SGD ($0 < w < 1$, and $w \in R$). Whereas, the parameters affecting performance include the GPU block size *GPU_BLOCK_SIZE* ($GPU\_BLOCK\_SIZE = 2^M$, and $M \in [1, 9]$) for all three algorithms and the tile length l ($0 < l < 64$, and $l \in N$) in TB-COO for SGD. Note that the tile length of TB-COO is fixed for ALS and CCD+ due to the inter-warp reduction adopted for optimizing these two algorithms. Therefore, the only tunable parameter is *GPU_BLOCK_SIZE* during performance auto-tuning for ALS and CCD+. Based on our empirical study, the performance-related parameters do not affect the convergence of tensor completion.

Generally, determining the optimal settings for the above parameters with the exhaustive search is prohibitive. Therefore, we propose an auto-tuning scheme using the genetic search [9] to identify the optimal parameter settings automatically. In the genetic search, multiple genes constitute an individual, where each gene is a tuning parameter. Many individuals constitute a population. Each process handles the operations of a sub-population. The migration among the sub-populations is achieved through MPI. For migration, each sub-population exchanges individuals with its two neighborhoods. The new individual in the sub-population is bred through its two parents. Breeding involves two steps: *1)* each gene in the individual is randomly selected from the parents; *2)* genes in the individual mutate with a certain probability. The mutation is used to avoid individuals falling into local optimum. The optimal parameter settings are determined when the genetic algorithm converges to constant.

In our study, we perform two rounds of auto-tuning to determine the parameters for optimizing convergence and performance. Specifically, the auto-tuning is first applied to optimize the set of parameters affecting the convergence to ensure the algorithm reaches acceptable RMSE. With the settings of convergence parameters fixed, the auto-tuning is then applied to optimize the set of parameters affecting the performance.

4 EVALUATION

4.1 Experiment Setup

Tensor Completion Libraries. To the best of our knowledge, there is no public tensor completion library on GPU available. The cuTensor [30] library from Nvidia only contains basic tensor operations, including tensor contraction, reduction and element-wise tensor operations, which cannot solve the tensor completion problem directly. Therefore, to provide a fair comparison, we use two cutting-edge tensor completion libraries on CPU including SPLATT [37] and Cyclops [52]. The tensor completion routine in SPLATT transforms input tensor to CSF format and adopts the Intel Math Kernel Library (MKL) to accelerate the matrix operations. Whereas the tensor completion routine in Cyclops transforms input tensor to COO-like format and executes on CPU using Cyclops library [39]. Moreover, to provide a comparison on GPU, we have implemented COO-TC on GPU, which is a tensor completion library using COO format. To improve the performance of COO-TC, we also utilize the blocking strategy with shared memory, where each thread is assigned with equal number of non-zero elements. The source code of *cuTC* is available online at https://github.com/abovekumo/cuTC.
Tensor Datasets. We use real-world datasets for evaluation. We adopt five real-world datasets, including YELP [49], MovieLens [33], MIT-DAPRA [14], Nell-1 and Nell-2 [2]. The details of datasets are listed in Table 2, where I, J, K denotes the length of each dimension for each dataset, nnz denotes the number of non-zero elements and *Sparsity* is calculated as $\frac{nnz}{I \times J \times K}$. Each dataset is split with 80% for training, 10% for validation and 10% for test.

Table 2: The tensor datasets.

Dataset	I	J	K	nnz	Density
YELP	71K	16K	108	334K	2.72E-6
MovieLens	72K	11K	157	10M	8.04E-5
MIT-DARPA	22K	22K	24M	28M	2.41E-9
Nell-1	3M	2M	25M	144M	9.6E-13
Nell-2	12K	9K	29K	77M	2.46E-5
Yahoo!	205K	133	101	257M	9.33E-2
Amazon-T	4.8M	1.8M	1.8M	850M	6.66E-8
Patents-T	46	239K	239K	1.5B	5.71E-4

Experiment Platform. To compare with other tensor completion libraries, our evaluation is conducted on a CPU-GPU server, which contains two Intel Xeon E5-2680v4 CPUs, each with 14 cores. Moreover, there are two NVIDIA Tesla V100 GPUs in the server. The GPUs are connected to the CPU sockets through the PCIe bus. On this server, *cuTC* exchanges data between GPUs through PCIe bus instead of utilizing NCCL. The important hardware and software configurations of the server are listed in Table 3. We parallelize linear algebra routines with MKL on CPUs and cuSOLVER on GPUs,

respectively. Besides, Cyclops v1.5.5 and HPTT are used to build the tensor completion routine in Cyclops. Whereas the tensor completion routine in SPLATT is installed with OpenMP and MKL. *cuTC* utilizes OpenMP to control the parallel execution among multiple GPUs on a single node. To further evaluate the scalability of our *cuTC*, we provide more experiments on another server with eight V100 GPUs connected through NVLink. The configuration details are provided in Section 4.3.

Table 3: Experimental platform configuration.

	Intel Xeon E5-2680v4	NIVDIA Tesla V100
Microarchitecture	Broadwell	Volta
Frequency	2.4GHz	1.23GHz
Memory Size	189 GB	32 GB
Peak DP Performance	537.6 GFLOPS	7.066 TFLOPS
Compiler	Intel Compiler v19.0.4	CUDA NVCC v10.2

Evaluation Criteria. Firstly, we compare the performance of *cuTC* on a single GPU as well as on multiple GPUs, with COO-TC on a single GPU, SPLATT-COMPLETION and Cyclops-COMPLETION on multicore CPUs. Then, we compare the storage overhead of TB-COO and other popular sparse tensor formats. In addition, we compare with MM-CSF [27] by implementing MTTKRP algorithm with TB-COO on GPU. Moreover, we evaluate the scalability of *cuTC* with increasing rank size on multiple GPUs. The effectiveness of our proposed optimizations is further analyzed using the roofline model [47]. Eventually, we present the experiment results with auto-tuning on convergence and performance. In all experiments, the execution time represents the average execution time of an epoch, and the convergence is evaluated with RMSE in validation datasets named RMSE-vl, which is formulated as RMSE $= \sqrt{\frac{\sum_{\mathcal{X}_{i,j,k}} \mathcal{L}^2_{i,j,k}}{nnz(\mathcal{X})}}$. The RMSE-vl is calculated after the last epoch. For GPU implementations, the data transfer time between CPU and GPU within an epoch is included. All the computation is done in double precision. The symbol $\emptyset$ denotes an execution instance fails due to out-of-memory error, and the symbol ∞ denotes an execution instance fails to converge with RMSE metric.

4.2 Performance Analysis

4.2.1 Performance Comparison. The performance results of *cuTC* with ALS, CCD+ and SGD algorithm compared to SPLATT-COMPLETION and Cyclops-COMPLETION at rank $R = 16$ are shown in Table 4, Table 5 and Table 6, respectively. SPLATT-COMPLETION and Cyclops-COMPLETION represent the tensor completion routine in SPLATT and Cyclops, respectively. Since the Cyclops-COMPLETION does not support calculating the RMSE for result validation, we omit the RMSE-vl results for Cyclops-COMPLETION (with – symbol). For the ease of discussion, we choose the execution time of SPLATT-COMPLETION as the baseline. For the datasets where SPLATT-COMPLETION fails to converge, we omit the speedup results (with – symbol). Meanwhile, the highest speedup achieved by *cuTC* on these optimization algorithms is bolded in Table 4-6.

First of all, the performance of *cuTC* on multiple GPUs exceeds SPLATT-COMPLETION and Cyclops-COMPLETION, at all datasets with all three optimization algorithms. In addition, the results from *cuTC* achieve similar or even better RMSE at most datasets compared to other implementations. For *cuTC* with ALS algorithm, the highest speedup of 3.21× is achieved on YELP dataset, whereas the average speedup is 1.86×. For CCD+ algorithm, *cuTC* achieves the highest speedup of 829.58× on NELL-2 dataset, whereas the average speedup is 205.63×. For SGD algorithm, the highest speedup of 7.86 × is achieved on MovieLens dataset, whereas the average speedup is 4.73×. Meanwhile, when comparing *cuTC* with COO-TC on a single GPU, it is obvious that *cuTC* performs better than COO-TC, especially with ALS and CCD+. The reason is that with TB-COO format, there are bitmaps to indicate the difference of the indices for non-zero elements, which allows efficient reduction and eliminates expensive atomic operations.

We notice that *cuTC* achieves worse RMSE than SPLATT-COMPLETION on MovieLens dataset. We suspect the reason is due to the initial value of the factor matrices randomly generated in *cuTC*, which affects the RMSE for particular datasets. However, the exact reason requires further investigation. We also notice that the average speedup of *cuTC* with ALS is far below with CCD+ and SGD algorithms. This is because ALS algorithm generates additional $\mathbf{H}^T\mathbf{H}$, which requires more memory access than CCD+ and SGD. In general, *cuTC* achieves the best performance with CCD+ algorithm since it requires less memory access and adopts the inter-warp reduction algorithm for eliminating the expensive atomic operations.

Moreover, *cuTC* is able to conduct the tensor completion for all five datasets, whereas the other three libraries fail on several datasets. Particularly, Cyclops-COMPLETION fails on the three large datasets due to memory overflow with all three optimization algorithms, whereas SPLATT-COMPLETION fails to converge on NELL-1 and NELL-2 datasets with SGD algorithm (generating invalid RMSE-vl results). COO-TC fails to converge on Nell-1 dataset with CCD+ and SGD because there is no auto-tuning scheme to obtain the optimal parameter configurations, and it also causes memory overflow with ALS since it needs much larger memory buffer for $\mathbf{H}^T\mathbf{H}$ than *cuTC*.

cuTC on multiple GPUs achieves better performance than on a single GPU, especially on large datasets. This is because the performance improvement for updating factor matrices outweighs the overhead of data transfer between CPU and GPU when processing large datasets. Whereas on small datasets, the computation of updating factor matrices is quite limited, which is not enough for compensating the data transfer overhead, and thus leads to deteriorated performance speedup. In addition, due to the memory constraint of a single GPU, *cuTC* also suffers from memory overflow, especially when running ALS algorithm on large datasets.

Furthermore, we measure the absolute performance (in GFLOPS) and iteration number of SPLATT-COMPLETION, COO-TC and *cuTC*. We do not compare with Cyclops-COMPLETION because its performance falls behind other implementations by order of magnitude (shown in Table 4∼Table 6). The absolute performance is measured using Performance Application Programming Interface (PAPI) [25] on CPU and NVPROF [1] on GPU, respectively. We present the average GFLOPS per epoch. The iteration number denotes the number of iterations that the algorithm needs to reach convergence. The absolute performance in Figure 6 demonstrates

Table 4: The performance comparison of *cuTC*, SPLATT-COMPLETION and Cyclops-COMPLETION with ALS Algorithm.

Dataset	SPLATT-COMPLETION			Cyclops-COMPLETION			COO-TC			*cuTC*-single GPU			*cuTC*-multiple GPUs		
	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl
YELP	0.045s	1	3.98	15.64s	0.0029	–	0.018s	0.25	2.84	0.013s	3.46	4.05	0.014s	**3.21**	3.98
MovieLens	0.14s	1	0.84	38.10s	0.0037	–	0.25s	0.56	1.57	0.14s	1	3.67	0.075s	1.87	3.67
MIT-DARPA	3.18s	1	34.30	∅	–	–	∅	–	∅	∅	–	∅	2.01s	1.58	1.12
Nell-1	4.81s	1	23.61	∅	–	–	∅	–	∅	∅	–	∅	2.48s	1.58	23.60
Nell-2	0.51s	1	189.52	∅	–	–	5.61s	0.091	1204.03	∅	–	∅	0.49s	1.04	82.50

Table 5: The performance comparison of *cuTC*, SPLATT-COMPLETION and Cyclops-COMPLETION with CCD+ Algorithm.

Dataset	SPLATT-COMPLETION			Cyclops-COMPLETION			COO-TC			*cuTC*-single GPU			*cuTC*-multiple GPUs		
	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl
YELP	0.023s	1	3.98	3.65s	0.0063	–	0.0032s	7.19	3.98	0.0019s	12.11	3.98	0.0034s	6.76	3.98
MovieLens	0.29s	1	0.84	46.50s	0.0063	–	0.032s	9.06	3.67	0.0032s	90.63	3.67	0.0037s	78.38	3.67
MIT-DARPA	4.28s	1	1.07	∅	–	–	∞	–	∞	0.53s	8.08	1.13	0.44s	9.73	1.13
Nell-1	53.92s	1	30.97	∅	–	–	∞	–	∞	0.68s	79.29	23.60	0.52s	103.69	23.60
Nell-2	5.89s	1	190.72	∅	–	–	∞	–	∞	0.0094s	626.60	82.50	0.0071s	**829.58**	82.50

Table 6: The performance comparison of *cuTC*, SPLATT-COMPLETION and Cyclops-COMPLETION with SGD Algorithm.

Dataset	SPLATT-COMPLETION			Cyclops-COMPLETION			COO-TC			*cuTC*-single GPU			*cuTC*-multiple GPUs		
	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl	Time	Speedup	RMSE-vl
YELP	0.015s	1	4.45	0.24s	0.063	–	0.0043s	3.49	1.47	0.0035s	4.29	4.06	0.0044s	3.41	5.23
MovieLens	0.11s	1	0.88	2.03s	0.054	–	0.030s	3.67	1.23	0.022s	5	3.75	0.0140s	**7.86**	3.71
MIT-DARPA	2.68s	1	1.12	∅	–	–	∞	–	∞	0.81s	3.31	1.34	0.92s	2.91	1.26
Nell-1	∞	–	∞	∅	–	–	∞	–	∞	1.55s	–	23.61	1.35s	–	24.38
Nell-2	∞	–	∞	∅	–	–	∞	–	∞	0.24s	–	82.50	0.14s	–	82.67

that *cuTC* exceeds SPLATT-COMPLETION and COO-TC on all datasets. We observe that the *cuTC* on a single GPU may achieve better performance than on multiple GPUs when running on small datasets. The reason is that the performance speedup of parallelization across multiple GPUs cannot compensate the synchronization overhead on small datasets. From the iteration number shown in Table 7, it is obvious that SGD algorithm usually requires more epochs than ALS and CCD+ algorithm to reach convergence. In most cases, *cuTC* requires less epochs to reach convergence than SPLATT-COMPLETION due to the optimal parameter settings determined by auto-tuning scheme.

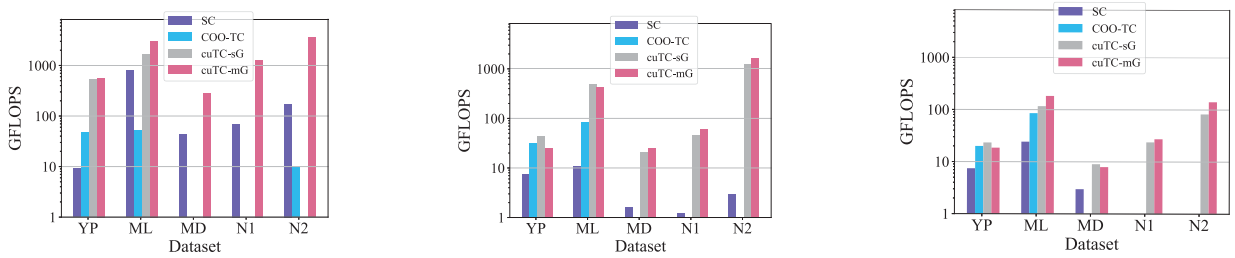
Table 7: The iteration comparison of SPLATT-COMPLETION, COO-TC and *cuTC* on both a single GPU and multiple GPUs. SC, *cuTC*-sG and *cuTC*-mG denote SPLATT-COMPLETION, *cuTC* on a single GPU and multiple GPUs, respectively.

Dataset	SC			COO-TC			*cuTC*-sG			*cuTC*-mG		
	ALS	CCD+	SGD	ALS	CCD+	SGD	ALS	CCD+	SGD	ALS	CCD+	SGD
YELP	21	21	24	20	3	118	20	4	3	5	4	3
MovieLens	23	131	85	21	3	99	20	4	134	7	4	96
MIT-DARPA	21	124	25	∅	∞	∞	∅	4	39	4	4	33
Nell-1	20	20	∞	∅	∞	∞	∅	4	3	4	4	3
Nell-2	20	20	∞	20	∞	∞	∅	4	3	7	4	3

Table 8: The performance of *cuTC* on Amazon-T and Patents-T datasets running on eight GPUs.

	ALS	CCD+	SGD
Amazon-T	0.5739s	0.1342s	0.3781s
Patents-T	0.7387s	0.02439s	1.8232s

4.2.2 Storage Comparison. To understand the storage overhead of TB-COO, we compare TB-COO with COO, HiCOO and MM-CSF on the evaluation datasets. The analytical results are shown in Table 9, where "*-ONE" means that the algorithm uses one tensor representation to update d modes, and "*-ALL" means that the algorithm uses d tensor representations to update d modes. We use open source implementations of CSF-ONE and CSF-ALL from SPLATT-COMPLETION [37], COO and HiCOO from ParTi! [21], and MM-CSF from [27]. For HiCOO, we set the block size B to 128 according to [21]. For consistency, we use data type of *uint*64_*t* and *double* to store indices and values, respectively. We present the results of both TB-COO-ALL (used in ALS and CCD+) and TB-COO-ONE (used in SGD). From Table 9, we can see that TB-COO-ONE can reduce the memory footprint compared to COO, and TB-COO-ALL has the similar memory footprint as CSF-ALL, which is due to the compression in TB-COO through bitmap. Although TB-COO-ALL uses more memory compared to COO/HiCOO/MM-CSF, it can achieve better performance than COO and MM-CSF as shown in Section 4.2.1 and Section 4.2.3.



(a) ALS (b) CCD+ (c) SGD

Figure 6: The performance comparison (in GFLOPS) of SPLATT-COMPLETION (SC), COO-TC and *cuTC* on both a single GPU and multiple GPUs. YP, ML, MD, N1 and N2 on the x-axis are abbreviations of YELP, MovieLens, MIT-DARPA, Nell-1 and Nell-2 datasets. SC, *cuTC*-sG and *cuTC*-mG denote SPLATT-COMPLETION, *cuTC* on a single GPU and multiple GPUs, respectively. The blank space indicates the implementation fails to converge or runs out of memory.

Table 9: The storage comparison (in MB) of TB-COO-ONE, TB-COO-ALL, COO, HiCOO, MM-CSF, CSF-ONE and CSF-ALL.

Dataset	TB-COO-ONE	TB-COO-ALL	COO	HiCOO	CSF-ONE	CSF-ALL	MM-CSF
YELP	6.87	19.65	8.18	4.00	7.53	25.38	5.25
MovieLens	189.33	566.99	244.14	85.74	152.96	448.53	71.82
MIT-DARPA	537.11	1792.55	694.24	410.53	349.91	1719.72	689.05
Nell-1	2733.69	8367.64	3505.85	2479.46	2767.20	9736.63	2951.03
Nell-2	1451.81	4355.52	1876.94	683.04	1005.39	3497.28	772.50

4.2.3 Comparison on MTTKRP. To provide a fair comparison with the state-of-the-art sparse tensor format on GPU, we evaluate the performance of MTTKRP implemented using TB-COO and MM-CSF [27] (since MM-CSF only provides MTTKRP implementation). The results are shown in Figure 7. The MM-CSF only supports MTTKRP execution on a single GPU, whereas *cuTC* can run on multiple GPUs. It is clear that *cuTC* outperforms MM-CSF on most datasets. Compared to MM-CSF, *cuTC* achieves 1.82× speedup on average on a single GPU. Moreover, the performance of *cuTC* further outperforms MM-CSF when scaling the MTTKRP execution on multiple GPUs, with an average speedup of 2.55×. We notice that the only case with MovieLens dataset, where *cuTC* performs worse than MM-CSF on a single GPU. The reason is that the MovieLens dataset is relatively dense on one dimension while sparse on other two dimensions. The sparsity in MovieLens generates more fibers in an entry of TB-COO, and thus causes more atomic operations during inter-warp reduction. The excessive atomic operations deteriorate the performance of TB-COO on MovieLens dataset.

4.2.4 Rank Scalability. The results for rank scalability of *cuTC* with ALS, CCD+ and SGD algorithms on two GPUs are shown in Figure 8. First of all, the ALS algorithm has more chance to end up with memory overflow due to the requirement for extra memory space to store $\mathbf{H}^T\mathbf{H}$, especially on the memory constrained GPU architecture. In addition, the slope of the execution time curve for ALS algorithm is much higher than CCD+ and SGD, as the rank

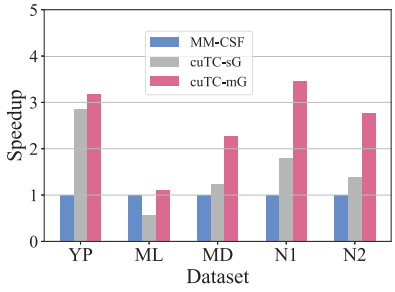
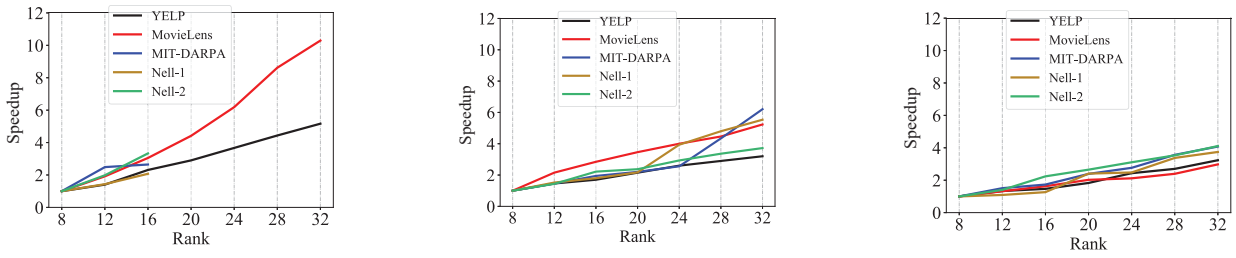


Figure 7: Performance comparison of *cuTC* and MM-CSF (baseline) on MTTKRP. YP, ML, MD, N1 and N2 on the x-axis are abbreviations of YELP, MovieLens, MIT-DARPA, Nell-1 and Nell-2 datasets. *cuTC*-sG and *cuTC*-mG denote *cuTC* on a single GPU and multiple GPUs, respectively.

size increases. This is because the amount of computation required for updating $\mathbf{H}^T\mathbf{H}$ in ALS exhibits a square relationship with rank size. In general, when the rank size is relatively small (below 12), the execution time of all three algorithms is almost the same across all datasets. However, as the rank size increases, the SGD algorithm exhibits better scalability across all datasets.

4.3 Scaling to more GPUs

To demonstrate the scalability of *cuTC* on more GPUs, we measure the performance of *cuTC* on another server with eight NVIDIA V100 GPUs. This server has one Intel Xeon Gold 6240 CPU and eight NVIDIA V100 GPUs, connected through NVLink. On this server, we use NCCL to exchange data among GPUs. To evaluate the scalability, we use a larger dataset, Yahoo! [6] dataset. We choose the performance of *cuTC* on a single GPU as baseline. As shown in Figure 9, ALS and SGD achieve good scalability on Yahoo! dataset. Meanwhile, as the number of GPUs scales, the cost of data transfer and synchronization between GPUs increases, which slows down the performance speedup. For CCD+, since its computation



(a) ALS (b) CCD+ (c) SGD

Figure 8: Rank scalability of *cuTC* with ALS, CCD+ and SGD algorithms. All results are normalized to the execution time in rank 8 on each dataset (baseline).

complexity is much lower than ALS and SGD (e.g., avoid computing $\mathbf{H}^T\mathbf{H}$ matrices), when scaling to more GPUs, the computation on each GPU becomes too small that offsets the benefit of parallelization. Therefore, its performance becomes stagnant when the number of GPUs is beyond two. Moreover, we evaluate *cuTC* on other two large datasets Amazon [24] and Patents [35]. Since the datasets are too large to be fitted with even eight GPUs, we truncate the datasets as large as they can be fitted with eight GPUs by randomly partitioning at fixed ratio and dimensions. We truncate the datasets as Amazon-T and Patents-T by randomly partitioning at fixed ratio and dimensions. The details of the truncated datasets are shown in Table 2. We truncate the datasets as long as they can be fitted with eight GPUs. The performance results are shown in Table 8, where the CCD+ algorithm achieves the best performance on these two datasets.

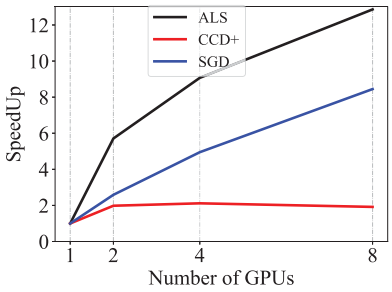


Figure 9: Scalability of *cuTC* with ALS, CCD+ and SGD algorithms on Yahoo! dataset. All results are normalized to the execution time on a single GPU (baseline).

4.4 Roofline Model Analysis

Given a sparse tensor $\mathcal{X}$ with I', J', K' non-zero slices in three modes respectively, nnz non-zero elements and rank R, we utilize the roofline model to better understand the effectiveness of our optimizations applied in *cuTC* for ALS, CCD+ and SGD algorithms on GPU. All indices and values are stored with eight bytes. The roofline ceiling on NVIDIA V100 GPU is 7.83 TFLOPS and the peak memory bandwidth is 900 GB/s, therefore the ridge point I is 8.7 FLOPS/Byte. We take Nell-2 dataset with $R = 16$, $nnz = 77M$ and $TL = 32$ for quantitative analysis. Here, R is the rank, nnz is the number of non-zeros and TL is the length of TB-COO tiles. Other datasets exhibit similar tendency with roofline model analysis.

4.4.1 ALS. For ALS algorithm, we only analyze the update of MTTKRP and $\mathbf{H}^T\mathbf{H}$ since we utilize cuSOLVER for solving matrix equations. Based on the equations in Section 2.2, we can calculate the amount of data Q accessed from memory as $\frac{(4+\frac{1}{TL}+(2+\alpha)R+\frac{\alpha R^2}{2})nnz}{N}$, the number of floating-point operations W as $\frac{(2R+\frac{R^2}{2})nnz}{N}$ and the arithmetic intensity I as Equation 4 for ALS in mode-1 tensor completion, where N is the number of GPUs and α is the ratio of performing atomic operations. Our inter-warp reduction optimization can reduce the ratio of atomic operations (α) down to 3.14% for dataset Nell-2, and thus increase I from 0.11 FLOP/Bytes to 0.49 FLOP/Bytes, which indicates our optimization can effectively improve the computation intensity, and in turn mitigate the memory-bound constraint.

$$I = \frac{(2R + \frac{R^2}{2})}{(4 + \frac{1}{TL} + (2+\alpha)R + \frac{\alpha R^2}{2}) * 8bytes} \tag{4}$$

4.4.2 CCD+. For CCD+ algorithm, we only analyze the sub-procedure of updating the numerators and denominators, where the inter-warp reduction algorithm is applied. Based on the equations in Section 2.4, the arithmetic intensity I can be calculated from Equation 5 for CCD+ in mode-1 tensor completion. It is obvious that when $\alpha = 1$ (not optimized) there is $I < \frac{4}{3}$, which indicates CCD+ is memory-bound on GPU. Similarly to ALS, for Nell-2 dataset, our inter-warp reduction optimization is effective to mitigate the memory-bound constraint, by increasing I from 0.095 FLOP/Bytes to 0.15 FLOP/Bytes.

$$I = \frac{4R}{(4 + \frac{1}{TL} + 3R + 2\alpha R) * 8bytes} \tag{5}$$

4.4.3 SGD. For SGD algorithm, based on the equations in Section 2.3, the arithmetic intensity I can be calculated from Equation 6 for SGD in mode-1 tensor completion. It is clear that SGD is memory-bound on GPU since I is far less than the ridge point. Currently, our tiling scheme for SGD does not optimize its memory

access. It would be a potential optimization direction in our future work to alleviate the memory-bound constraint for SGD algorithm.

$$I = \frac{4R}{(4 + \frac{1}{TL} + 4R) * 8bytes} \quad (6)$$

4.5 Effectiveness of Auto-tuning

We apply auto-tuning to determine the optimal parameter settings for both algorithm convergence and performance. However, due to the limited search space for performance tuning of ALS and CCD+, we only apply performance auto-tuning for SGD algorithm. For the genetic search adopted in our auto-tuning, we set the number of sub-populations to 16, where each sub-population contains 64 individuals. In addition, the cross-over rate and the mutation rate are set to 0.8 and 0.005, respectively. The tuning parameters considered are described in Section 3.5. For the convergence, we set the search space parameters of λ, η, and w to 2048, 4, and 2, respectively. Based on these parameters, the search spaces of the ALS, CCD+ and SGD algorithms can be calculated. The genetic search has been performed for 20 generations each time.

4.5.1 Convergence. We chose RMSE-vl as the tuning target for algorithm convergence. As shown in Table 6, inappropriate parameter settings may lead to not convergent or positive infinite RMSE-vl, which invalidates the computation results. Therefore, it is necessary to determine the appropriate parameters through auto-tuning. As shown in Figure 10 (a), (b) and (c), we evaluate the effectiveness of auto-tuning for all three algorithms. The results demonstrate that for all datasets, the genetic search can quickly determine the optimal parameter settings in a few generations. Note that the convergence curve varies across different datasets, which indicates the sensitivity of the dataset to different parameter settings.

4.5.2 Performance. As shown in Figure 10 (d), we apply the genetic search to auto-tuning the performance of SGD algorithm. With the increasing generations, the genetic search gradually identifies the optimal parameter settings. Different from the convergence curves, we notice that the trends of the performance curves are similar across different datasets. The above observation further demonstrates that the necessity of auto-tuning method for achieving better performance of SGD algorithm through determining the optimal parameter settings regardless of the datasets.

4.5.3 Time Cost. As shown in Figure 11, we compare the time cost required to find the optimal parameter settings under the same search space for the genetic search and the random search. To evaluate the time cost, the termination condition is to find the optimal parameter settings (determined by exhaustive search) within the specified search space. We measure the time cost of the random search as half of the exhaustive search (assuming the search space is large enough, the probability for finding the optimal with random search is 50%). The evaluation results show that the genetic search can find the optimal solution in a shorter time in most cases (except Nell-1 in Performance-SGD). In addition, as the search space becomes larger, the advantage of the genetic search over the random search becomes more significant. For example, the difference of time cost in Figure 11 (c) is larger than that in (a) and (b). Note that the time cost of the genetic algorithm itself is negligible compared to evaluating the objective function (execution time of ALS/CCD+/SGD) to be optimized. The above results prove the effectiveness of the genetic search for performance auto-tuning.

5 RELATED WORK

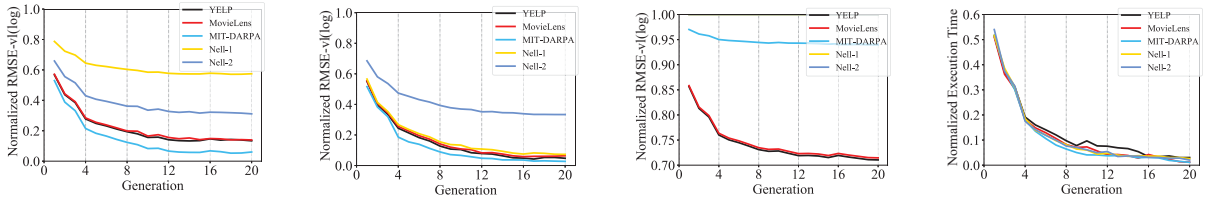
5.1 Optimizing Canonical Polyadic Decomposition

Recent researches have exploited tensor decomposition for its wide application in recommendation system [34], health record analysis [11] and machine learning [45]. Optimizing the performance of CPD algorithm and its main hotspot matricized tensor times Khatri-Rao product (MTTKRP) are the primary research targets. Kang *et al.* [15] utilized the MapReduce framework [5] to develop the Gigatensor that can handle tensors scaled in terabytes and minimize the intermediate data size. Besides, DFacTo [4] contains both ALS and Gradient Descent (GD) algorithm for CPD. It accelerates the MTTKRP process of CPD-ALS algorithm by utilizing Sparse matrix-vector multiplication (SpMV). In addition, Smith *et al.* [38] implemented SPLATT that adopts cache-friendly reordering and tiling mechanisms for high-performance CPD-ALS algorithm. Moreover, Choi *et al.* [3] utilized fine-grained blocking techniques to further optimize the SPLATT. Phipps *et al.* [31] developed portable and efficient CPD-ALS algorithm based on Kokkos [7] as well as improved the MTTKRP process. Larsen *et al.* [20] used leverage scores to sample the rows to accelerate the CPD-ALS algorithm without sacrificing accuracy. All above works have promoted the advance of tensor completion.

5.2 Parallelizing Tensor Completion

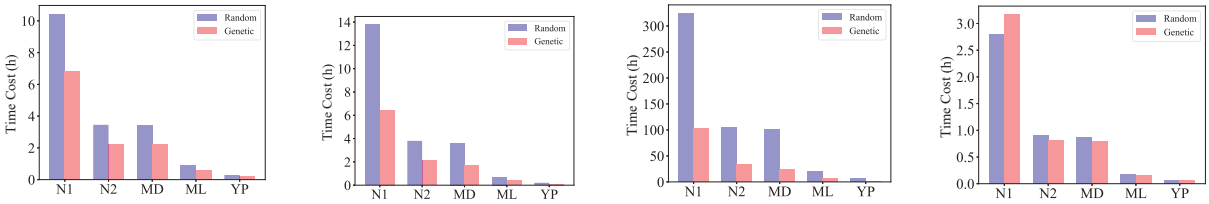
Tensor completion is generalization of matrix completion and the algorithms applied in tensor completion have already been widely adopted by matrix completion (such as ALS [13], SGD [18], CCD+ [50]). Moreover, researchers have already been making efforts to design parallel matrix completion algorithms [50].

For parallel tensor completion, most of the recent advances focus on decomposition based tensor completion [41]. Smith *et al.* [37] developed scalable parallel ALS, SGD and CCD+ algorithms for tensor completion on both shared and distributed memory systems. In addition, stratification and randomization have been adopted to further optimize the performance and convergence, respectively. Meanwhile, Karlsson *et al.* [16] implemented parallel ALS and CCD+ algorithm through novel data distribution paradigm on distributed memory systems. While efficient, those parallel algorithms are only optimized targeting multicore CPUs, which cannot be adopted to GPUs. Furthermore, Zhang *et al.* [52] developed ALS, SGD and CCD+ algorithm based on Cyclops tensor algebra library [39], which is a C++ library leveraging multiple parallel paradigms and libraries such MPI/OpenMP, HPTT [42], CUDA and ScaLAPACK. To improve the algorithm efficiency, they proposed a novel multi-tensor routine named TTTP that can outperform ALS. However, their implementations do not target at optimizing tensor completion on GPU neither. In addition, the Cyclops library transforms the tensor to matrices as well as uses dense tensor to store the output of sparse tensor-times-dense matrix multiply (spTTM) operation, which degrades the efficiency [22]. Zhang *et al.* [51] utilized the



(a) Convergence-ALS (b) Convergence-CCD+ (c) Covergence-SGD (d) Performance-SGD

Figure 10: Auto-tuning for *cuTC* with ALS, CCD+ and SGD algorithms. For SGD, both performance and convergence auto-tuning is included, whereas for ALS and CCD+ only convergence auto-tuning is included due to the limited search space in performance tuning. The y-axis is the execution time or log(RMSE-vl) normalized to the worst case results.



(a) Convergence-ALS (b) Convergence-CCD+ (c) Covergence-SGD (d) Performance-SGD

Figure 11: Comparison of the time cost using genetic search and random search. N1, N2, MD, ML and YP on the x-axis are abbreviations of Nell-1, Nell-2, MIT-DARPA, MovieLens and YELP datasets.

tubal sampling pattern in tensors from Internet of Things (IoT) and big data applications to develop the tensor completion algorithm on GPUs. However, they only optimized the least square algorithm for tensor completion. To the best of our knowledge, this is the first work to develop an efficient tensor completion library targeting multiple GPUs.

5.3 Sparse Tensor Storage Format

For sparse tensor computation, the storage format has a profound impact on its performance. Therefore, researchers have been paying attention to design novel sparse tensor storage formats for exploiting parallelism, reducing memory consumption and optimizing the performance. The coordinate format (COO) [17] that stores the indices and the values of non-zero elements directly, benefits from the insensitivity of the irregularities in sparse tensor structures. However, the original COO format can cause massive atomic operations on GPU, when the same indices are shared by multiple GPU threads. The F-COO [22] only stores the product modes and complements flag arrays to indicate the change in index mode to optimize tensor decomposition on GPU. However, F-COO increases the difficulty in searching for the index mode, which plays a significant role in tensor completion. Moreover, Li *et al.* proposed HiCOO [21] that compresses the indices in sparse tensor blocks for improving data locality and reducing memory consumption. Meanwhile, Smith *et al.* proposed *compressed data fiber* (CSF) format [36], which is the generalization of CSR format in matrices to optimize CPD on multi-core CPUs and reduce memory consumption. Nevertheless, the recursive algorithms for implementing CPD with CSF format do not fit on GPU [22]. To improve the load balance and further exploit parallelism on GPU, Nisa *et al.* [28] modified CSF to HB-CSF by splitting the dense slices. Besides, to improve the memory bandwidth, Srivastava *et al.* [43] proposed a customized Compressed Interleaved Sparse Slice (CISS) format for the tensor factorization accelerator *Tensaurus*. However, this format only works for *Tensaurus* accelerator, and has idle elements in its entries, which wastes memory capacity. Nisa *et al.* [27] further optimized the CSF format by proposing a mixed-mode CSF, which addresses the multi-mode tensor storage problem. In this paper, we propose a new storage format TB-COO along with an optimized tensor completion library that scales well to multiple GPUs.

6 CONCLUSION

In this paper, we develop the first tensor completion library on GPUs for three optimization algorithms, including ALS, CCD+ and SGD. We propose a novel sparse tensor storage format TB-COO, that extends COO format with tiles and bitmap for improving load balance. Based on TB-COO, we present an inter-warp reduction algorithm that leverages the warp shuffle mechanism on GPU to eliminate expensive atomic operations. We also develop different partitioning schemes for different algorithms when scaling to multiple GPUs. Moreover, we adopt an auto-tuning method to further improve the convergence and performance of the algorithms. Our implementation *cuTC* exceeds the state-of-the-art tensor completion libraries in performance on real-world datasets, while achieving similar or even better accuracy.

ACKNOWLEDGMENTS

This work is supported by National Key Research and Development Program of China (Grant No. 2020YFB1506703), and National Natural Science Foundation of China (Grant No. 62072018 and 61732002). Hailong Yang is the corresponding author.

REFERENCES

[1] Thomas Bradley. 2012. GPU performance analysis and optimisation. *NVIDIA Corporation* (2012).
[2] Andrew Carlson, Justin Betteridge, Bryan Kisiel, Burr Settles, Estevam R. Hruschka Jr., and Tom M. Mitchell. 2010. Toward an Architecture for Never-Ending Language Learning.. In *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*. AAAI Press, Atlanta, Georgia, 1306–1313.
[3] Jee Choi, Xing Liu, Shaden Smith, and Tyler Simon. 2018. Blocking optimization techniques for sparse tensor computation. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, IEEE, Vancouver, BC, Canada, 568–577.
[4] Joon Hee Choi and S Vishwanathan. 2014. DFacTo: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., Montreal, Canada, 1296–1304.
[5] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified data processing on large clusters. *Commun. ACM* 51, 1 (2004), 107–113.
[6] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2011. The yahoo! music dataset and kdd-cup'11. In *Proceedings of the 2011 International Conference on KDD Cup 2011-Volume 18*. JMLR.org, San Diego, USA, 3–18.
[7] H Carter Edwards, Christian R Trott, and Daniel Sunderland. 2014. Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3202–3216.
[8] Rainer Gemulla, Erik Nijkamp, Peter J Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, San Diego, California, USA, 69–77.
[9] David E Goldberg and John H Holland. 1988. Genetic algorithms and machine learning. *Machine learning* 3, 2 (1988), 95–99.
[10] Mark Harris. 2017. Nvidia dgx-1: The fastest deep learning system. (2017).
[11] Joyce C Ho, Joydeep Ghosh, and Jimeng Sun. 2014. Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, New York, NY, USA, 115–124.
[12] ThienLuan Ho, Seung-Rohk Oh, and HyunJin Kim. 2017. A parallel approximate string matching under Levenshtein distance on graphics processing units using warp-shuffle operations. *PloS one* 12, 10 (2017), 1–15.
[13] Prateek Jain, Praneeth Netrapalli, and Sujay Sanghavi. 2013. Low-rank matrix completion using alternating minimization. In *Proceedings of the forty-fifth annual ACM symposium on Theory of computing*. Association for Computing Machinery, New York, NY, USA, 665–674.
[14] Inah Jeon, Evangelos E. Papalexakis, U Kang, and Christos Faloutsos. 2015. HaTen2: Billion-scale Tensor Decompositions. In *IEEE International Conference on Data Engineering (ICDE)*. IEEE, Seoul, South Korea, 1047–1058.
[15] U Kang, Evangelos Papalexakis, Abhay Harpale, and Christos Faloutsos. 2012. Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*. Association for Computing Machinery, New York, NY, USA, 316–324.
[16] Lars Karlsson, Daniel Kressner, and André Uschmajew. 2016. Parallel algorithms for tensor completion in the CP format. *Parallel Comput.* 57 (2016), 222–234.
[17] Oguz Kaya and Bora Uçar. 2015. Scalable sparse tensor decompositions in distributed memory systems. In *SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–11.
[18] Raghunandan H Keshavan, Andrea Montanari, and Sewoong Oh. 2010. Matrix completion from noisy entries. *Journal of Machine Learning Research* 11, Jul (2010), 2057–2078.
[19] Tamara G Kolda and Brett W Bader. 2009. Tensor decompositions and applications. *SIAM review* 51, 3 (2009), 455–500.
[20] Brett W Larsen and Tamara G Kolda. 2020. Practical leverage-based sampling for low-rank tensor decomposition. *arXiv preprint arXiv:2006.16438* (2020).
[21] Jiajia Li, Jimeng Sun, and Richard Vuduc. 2018. HiCOO: hierarchical storage of sparse tensors. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 238–252.
[22] Bangtian Liu, Chengyao Wen, Anand D Sarwate, and Maryam Mehri Dehnavi. 2017. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE international conference on cluster computing (CLUSTER)*. IEEE, Hawaii, USA, 47–57.
[23] Daniel Maneval, Benoît Ozell, and Philippe Després. 2019. pGPUMCD: an efficient GPU-based Monte Carlo code for accurate proton dose calculations. *Physics in Medicine & Biology* 64, 8 (2019), 085018.
[24] Julian McAuley and Jure Leskovec. 2013. Hidden factors and hidden topics: understanding rating dimensions with review text. In *Proceedings of the 7th ACM conference on Recommender systems*. ACM, New York, NY, USA, 165–172.
[25] Philip J Mucci, Shirley Browne, Christine Deane, and George Ho. 1999. PAPI: A portable interface to hardware performance counters. In *Proceedings of the department of defense HPCMP users group conference*, Vol. 710. Citeseer.
[26] Maxim Naumov. 2011. Incomplete-LU and Cholesky preconditioned iterative methods using CUSPARSE and CUBLAS. (2011).
[27] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. 2019. An efficient mixed-mode representation of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. Association for Computing Machinery, New York, NY, USA, 1–25.
[28] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P Sadayappan. 2019. Load-balanced sparse mttkrp on gpus. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, Rio de Janeiro, Brazil, 123–133.
[29] NVIDIA. 2019. The CUDA solver library (cuSOLVER). (2019). https://docs.nvidia.com/cuda/cusolver/index.html
[30] Nvidia. 2020. cuTENSOR: A High-Performance CUDA Library For Tensor Primitives. (2020). https://docs.nvidia.com/cuda/cutensor/index.html
[31] Eric T Phipps and Tamara G Kolda. 2019. Software for sparse tensor decomposition on emerging computing architectures. *SIAM Journal on Scientific Computing* 41, 3 (2019), C269–C290.
[32] Benjamin Recht, Christopher Re, Stephen Wright, and Feng Niu. 2011. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*. Curran Associates, Inc., Granada, Spain, 693–701.
[33] GroupLens Research. 2019. MovieLens. (2019). http://grouplens.org/datasets/movielens/
[34] Yue Shi, Alexandros Karatzoglou, Linas Baltrunas, Martha Larson, Alan Hanjalic, and Nuria Oliver. 2012. TFMAP: optimizing MAP for top-n context-aware recommendation. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*. Association for Computing Machinery, New York, NY, USA, 155–164.
[35] Shaden Smith, Jee W. Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. 2017. FROSTT: The Formidable Repository of Open Sparse Tensors and Tools. (2017). http://frostt.io/
[36] Shaden Smith and George Karypis. 2015. Tensor-matrix products with a compressed sparse tensor. In *Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms*. Association for Computing Machinery, New York, NY, USA, 1–7.
[37] Shaden Smith, Jongsoo Park, and George Karypis. 2018. HPC formulations of optimization algorithms for tensor completion. *Parallel Comput.* 74 (2018), 99–117.
[38] Shaden Smith, Niranjay Ravindran, Nicholas D Sidiropoulos, and George Karypis. 2015. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *2015 IEEE International Parallel and Distributed Processing Symposium*. IEEE, Hyderabad, India, 61–70.
[39] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. 2014. A massively parallel tensor contraction framework for coupled-cluster computations. *J. Parallel and Distrib. Comput.* 74, 12 (2014), 3176–3190.
[40] Mingcong Song, Yang Hu, Huixiang Chen, and Tao Li. 2017. Towards pervasive and user satisfactory CNN across GPU microarchitectures. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, Austin, USA, 1–12.
[41] Qingquan Song, Hancheng Ge, James Caverlee, and Xia Hu. 2019. Tensor completion algorithms in big data analytics. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 13, 1 (2019), 1–48.
[42] Paul Springer, Tong Su, and Paolo Bientinesi. 2017. HPTT: a high-performance tensor transposition C++ library. In *Proceedings of the 4th ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*. Association for Computing Machinery, New York, NY, USA, 56–62.
[43] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. 2020. Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, San Diego, USA, 689–702.
[44] Huachun Tan, Bin Cheng, Wuhong Wang, Yu-Jin Zhang, and Bin Ran. 2014. Tensor completion via a multi-linear low-n-rank factorization model. *Neurocomputing* 133 (2014), 161–169.
[45] M Alex O Vasilescu. 2011. Multilinear projection for face recognition via canonical decomposition. In *Face and Gesture 2011*. IEEE, Santa Barbara, USA, 476–483.
[46] Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C Denny, Abel Kho, You Chen, Bradley A Malin, and Jimeng Sun. 2015. Rubik: Knowledge guided tensor

factorization and completion for health data analytics. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, New York, NY, USA, 1265–1274.

[47] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[48] Shengen Yan, Guoping Long, and Yunquan Zhang. 2013. StreamScan: fast scan algorithms for GPUs without global barrier synchronization. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Association for Computing Machinery, New York, NY, USA, 229–238.

[49] YELP. 2015. YELP. (2015). http://www.yelp.com/dataset_challenge/

[50] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit Dhillon. 2012. Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In *2012 IEEE 12th International Conference on Data Mining*. IEEE, Brussels, Belgium, 765–774.

[51] Tao Zhang, Xiao-Yang Liu, and Xiaodong Wang. 2020. High Performance GPU Tensor Completion With Tubal-Sampling Pattern. *IEEE Transactions on Parallel and Distributed Systems* 31, 7 (2020), 1724–1739.

[52] Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. 2019. Enabling Distributed-Memory Tensor Completion in Python using New Sparse Tensor Kernels. *arXiv preprint arXiv:1910.02371* (2019).