High-Performance Tucker Factorization on Heterogeneous Platforms

Sejoon Oh, Namyong Park, Jun-Gi Jang, Lee Sael, and U Kang

Abstract—Given large-scale multi-dimensional data (e.g., (user, movie, time; rating) for movie recommendations), how can we extract latent concepts/relations of such data? Tensor factorization has been widely used to solve such problems with multi-dimensional data, which are modeled as tensors. However, most tensor factorization algorithms exhibit limited scalability and speed since they require huge memory and heavy computational costs while updating factor matrices. In this paper, we propose GTA, a general framework for Tucker factorization on heterogeneous platforms. GTA performs alternating least squares with a row-wise update rule in a fully parallel way, which significantly reduces memory requirements for updating factor matrices. Furthermore, GTA provides two algorithms: GTA-PART for partially observable tensors and GTA-FULL for fully observable tensors, both of which accelerate the update process using GPUs and CPUs. Experimental results show that GTA exhibits $5.6 \sim 44.6\times$ speed-up for large-scale tensors compared to the state-of-the-art. In addition, GTA scales near linearly with the number of GPUs and computing nodes used for experiments.

Index Terms—Tensor Analysis, Tucker Factorization, Heterogeneous Computing, GPGPU, OpenCL

1 Introduction

Given large-scale tensors, how can we discover latent concepts/relations of them? How can we design a time-efficient algorithm for analyzing given tensors? Various real-world data can be modeled as tensors or multi-dimensional arrays (e.g., (user, movie, time; rating) for movie recommendations). Tensors are classified into two types: partially observable ones (typically sparse) or fully observable ones (dense). Partially observable tensors consist of observable and missing entries (whose values are unknown); however, there are no missing entries for fully observable ones (consist of nonzero- and zero-value entries). Tensor factorization has been widely used for analyzing tensors [1], [2], [3]. Among tensor factorization methods [4], Tucker factorization has received much interest since it is a generalized form of other factorization methods like CANDECOMP/PARAFAC (CP) decomposition, and it allows us to examine latent factors and hidden relations of a tensor.

While many algorithms have been developed for Tucker factorization [5], [6], [7], [8], most methods exhibit limited scalability since they exploit tensor operations and singular value decomposition (SVD), leading to heavy memory and computational requirements. In particular, tensor operations generate huge intermediate data for large-scale tensors, which is a problem called *intermediate data explosion* [9]. Moreover, they are not suitable for decomposing large-scale tensors due to their limited speed. Although several parallel Tucker methods have been developed to address the performance issue, they do not meet the maximum parallelization efficiency as they are executed in central processing units (CPUs) with a relatively small number of cores. A few Tucker algorithms [10], [11], [12], [13] have been developed to address the above problems, but they fail to solve the scalability and performance issues at the same time. In summary, the major challenges for decomposing large-scale tensors are 1) how to avoid intermediate data explosion and high computational costs caused by tensor operations and SVD, and 2) how to achieve massive parallelism during the factorization process for better performance.

In this paper, we propose GTA, a general framework for Tucker factorization on heterogeneous platforms. GTA performs alternating least squares (ALS) with a row-wise update rule. The row-wise updates considerably reduce the amount of memory required for updating factor matrices, enabling GTA to avoid the *intermediate data explosion* problem. In addition, GTA maximizes parallelization

TABLE 1: Summary of our proposed method GTA and competitors. A check-mark of a method indicates that the algorithm is satisfying a particular aspect. GTA is the only method scalable with all aspects of tensor scale, factorization speed, GPU applicability, and generality; on the other hand, competitors have limited scalability for some aspects.

Method	Scale	Speed	GPU	Type of Tensors
SPLATT	✓			Fully Observed
P-TUCKER	✓			Partially Observed
GTA				
GTA-PART	✓	✓	✓	Partially Observed
GTA-FULL	✓	✓	✓	Fully Observed

- *Sejoon Oh is with the Department of Computer Science and Engineering, Seoul National University, Republic of Korea, E-mail: ohhenrie@snu.ac.kr*
- *Namyong Park is with the Department of Computer Science, Carnegie Mellon University, USA, E-mail: namyongp@cs.cmu.edu*
- *Jun-Gi Jang is with the Department of Computer Science and Engineering, Seoul National University, Republic of Korea, E-mail: elnino4@snu.ac.kr*
- *Lee Sael is with the Department of Computer Science and Engineering, Seoul National University, Republic of Korea, E-mail: saellee@gmail.com*
- *U Kang is with the Department of Computer Science and Engineering, Seoul National University, Republic of Korea, E-mail: ukang@snu.ac.kr (corresponding author)*

efficiency by employing graphics processing units (GPUs) for overcoming a computational bottleneck, while less-computational parts are processed by CPUs. GTA consists of two algorithms: GTA-PART for partially observable tensors and GTA-FULL for fully observable tensors. The main idea of GTA-PART is to compute intermediate data $\delta^{(n)}$ by GPUs, and GTA-FULL is an extension of GTA-PART to fully observable tensors with a factorization technique. Table 1 summarizes a comparison of GTA and competitors regarding various aspects.

Our main contributions are the following:

- **Algorithm.** We propose GTA, a general framework for Tucker factorization on heterogeneous platforms. The key ideas of GTA are 1) a row-wise update rule of factor matrices, 2) nonzero-based parallelization on GPUs for computing intermediate data δ, and 3) a factorization technique for fully observable Tucker factorization.
- **Theory.** We theoretically derive a row-wise update rule of factor matrices and prove the correctness and convergence of it. Moreover, we analyze the time and memory complexities of GTA and other methods, as summarized in Table 4.
- **Performance.** GTA provides the best performance across all aspects: tensor scale, factorization speed, GPU applicability, and generality. Experimental results demonstrate that GTA achieves $5.6 \sim 44.6\times$ speed-up for large-scale tensors, as summarized in Figures 5 to 8.

The source code of GTA and datasets used in this paper are publicly available at **https://github.com/sejoonoh/GTA-Tensor** for reproducibility. The rest of this paper is organized as follows. Section 2 explains preliminaries on a tensor, its operations and factorizations, and heterogeneous computing with OpenCL and SnuCL. Section 3 describes our proposed method GTA. Section 4 presents experimental results of GTA and other methods. After introducing related works in Section 5, we conclude in Section 6.

2 Notations and Preliminaries

In this section, we describe the preliminaries of a tensor in Section 2.1, its operations in Section 2.2, its factorization methods in Section 2.3, and heterogeneous computing with OpenCL and SnuCL in Section 2.4. Notations and definitions are summarized in Table 2.

2.1 Tensor

Tensors, or multi-dimensional arrays, are a generalization of vectors (1-order tensors) and matrices (2-order tensors) to higher orders. As a matrix has rows and columns, an N-order tensor has N modes; their lengths (also called dimensionalities) are denoted by I_1 through I_N, respectively. We denote tensors by boldface Euler script letters (e.g., $\mathcal{X}$), matrices by boldface capitals (e.g., $\mathbf{A}$), and vectors by boldface lowercases (e.g., $\mathbf{a}$). An element of a tensor is denoted by the symbolic name of the tensor with its indices in subscript. For example, $a_{i_1 j_1}$ indicates the (i_1, j_1)th entry of $\mathbf{A}$, and $\mathcal{X}_{(i_1,...,i_N)}$ denotes the $(i_1, ..., i_N)$th entry of $\mathcal{X}$. The i_1th row of $\mathbf{A}$ is denoted by $\mathbf{a}_{i_1:}$, and the i_2th column of $\mathbf{A}$ is denoted by $\mathbf{a}_{:i_2}$.

TABLE 2: Table of symbols.

Symbol	Definition
$\mathcal{X}$	input tensor ($\in \mathbb{R}^{I_1 \times ... \times I_N}$)
$\mathcal{G}$	core tensor ($\in \mathbb{R}^{J_1 \times ... \times J_N}$)
N	order of $\mathcal{X}$
I_n, J_n	dimensionality of the nth mode of $\mathcal{X}$ and $\mathcal{G}$
$\mathbf{A}^{(n)}$	nth factor matrix ($\in \mathbb{R}^{I_n \times J_n}$)
$a^{(n)}_{i_n j_n}$	(i_n, j_n)th entry of $\mathbf{A}^{(n)}$
Ω	set of observable or nonzero-value entries of $\mathcal{X}$
$\Omega^{(n)}_{i_n}$	set of observable entries whose nth mode's index is i_n
$\lvert\Omega\rvert, \lvert\mathcal{G}\rvert$	number of observable entries of $\mathcal{X}$ and $\mathcal{G}$
λ	regularization parameter for factor matrices
$\lVert\mathcal{X}\rVert$	Frobenius norm of tensor $\mathcal{X}$
$\mathcal{M}^{(n)}$	an intermediate table for computing $\mathbf{B}^{(n)}$
T_1	number of CPU cores
T_2	number of GPU cores
α	an element $(i_1, ..., i_N)$ of input tensor $\mathcal{X}$
β	an element $(j_1, ..., j_N)$ of core tensor $\mathcal{G}$

2.2 Tensor Operations

We review some tensor operations used for Tucker factorization. More tensor operations are summarized in [4].

Definition 1 (Frobenius Norm). Given an N-order tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times ... \times I_N}$), the Frobenius norm of $\mathcal{X}$ is denoted by $||\mathcal{X}||$ and defined as follows:

$$||\mathcal{X}|| = \sqrt{\sum_{\forall (i_1,...,i_N) \in \mathcal{X}} \mathcal{X}^2_{(i_1,...,i_N)}}. \tag{1}$$

Definition 2 (Matricization/Unfolding). Matricization transforms a tensor into a matrix. The mode-n matricization of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ is denoted as $\mathbf{X}_{(n)}$. The mapping from an element $(i_1, ..., i_N)$ of $\mathcal{X}$ to an element (i_n, j) of $\mathbf{X}_{(n)}$ is given as follows:

$$j = 1 + \sum_{k=1, k \neq n}^{N} \left[(i_k - 1) \prod_{m=1, m \neq n}^{k-1} I_m \right]. \tag{2}$$

Note that all indices of a tensor and a matrix begin from 1.

Definition 3 (n-Mode Product). n-mode product enables multiplications between a tensor and a matrix. The n-mode product of a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$ with a matrix $\mathbf{U} \in \mathbb{R}^{J_n \times I_n}$ is denoted by $\mathcal{X} \times_n \mathbf{U}$ ($\in \mathbb{R}^{I_1 \times \cdots \times I_{n-1} \times J_n \times I_{n+1} \times \cdots \times I_N}$). Element-wise, we have

$$(\mathcal{X} \times_n \mathbf{U})_{i_1 \cdots i_{n-1} j_n i_{n+1} \cdots i_N} = \sum_{i_n=1}^{I_n} (\mathcal{X}_{(i_1 i_2 \cdots i_N)} u_{j_n i_n}). \tag{3}$$

2.3 Tensor Factorization Methods

Our proposed method GTA is based on Tucker factorization, one of the most popular decomposition methods. More details about other factorization algorithms are summarized in Section 5.

Definition 4 (Tucker Factorization). Given an N-order tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times ... \times I_N}$), Tucker factorization approximates $\mathcal{X}$ by a core tensor $\mathcal{G}$ ($\in \mathbb{R}^{J_1 \times ... \times J_N}$) and factor matrices

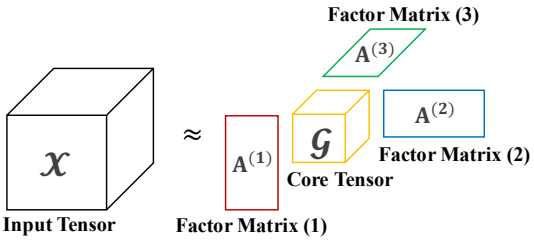

Fig. 1: Tucker factorization for a 3-way tensor.

$\{\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n} | n = 1...N\}$. Figure 1 illustrates a Tucker factorization result for a 3-way tensor. Core tensor $\mathcal{G}$ is assumed to be smaller and denser than the input tensor $\mathcal{X}$, and factor matrices $\mathbf{A}^{(n)}$ to be normally orthogonal. Regarding interpretations of factorization results, each factor matrix $\mathbf{A}^{(n)}$ represents the latent features of the object related to the nth mode of $\mathcal{X}$, and each element of a core tensor $\mathcal{G}$ indicates the weights of the relations composed of columns of factor matrices. Tucker factorization with tensor operations is presented as follows:

$$\min_{\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}} ||\mathcal{X} - \mathcal{G} \times_1 \mathbf{A}^{(1)} \cdots \times_N \mathbf{A}^{(N)}||. \quad (4)$$

An element-wise expression is given as follows:

$$\mathcal{X}_{(i_1,...,i_N)} \approx \sum_{\forall (j_1,...,j_N) \in \mathcal{G}} \mathcal{G}_{(j_1,...,j_N)} \prod_{n=1}^{N} a^{(n)}_{i_n j_n}. \quad (5)$$

Definition 5 (Partially Observable Tucker Factorization). Given a tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times ... \times I_N}$) with observable entries Ω, the goal of partially observable Tucker factorization (POTF) of $\mathcal{X}$ is to find factor matrices $\mathbf{A}^{(n)}$ ($\in \mathbb{R}^{I_n \times J_n}, n = 1, \cdots, N$) and a core tensor $\mathcal{G}$ ($\in \mathbb{R}^{J_1 \times ... \times J_N}$), which minimize (6).

$$L(\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{\forall \alpha \in \Omega} \left(\mathcal{X}_\alpha - \sum_{\forall \beta \in \mathcal{G}} \mathcal{G}_\beta \prod_{n=1}^{N} a^{(n)}_{i_n j_n} \right)^2 + \lambda \sum_{n=1}^{N} \|\mathbf{A}^{(n)}\|^2 \quad (6)$$

Note that the loss function (6) only depends on observable entries of $\mathcal{X}$, and L_2 regularization is used in (6) to prevent overfitting, which has been generally utilized in machine learning problems. We note that Frobenius norm for such regularization is broadly utilized [14], [15], [16]. α and β indicate entries of tensor $\mathcal{X}$ and $\mathcal{G}$, respectively.

Definition 6 (Fully Observable Tucker Factorization). Given a tensor $\mathcal{X}$ ($\in \mathbb{R}^{I_1 \times ... \times I_N}$) with nonzero-value entries Ω, the goal of fully observable Tucker factorization (FOTF) of $\mathcal{X}$ is to find factor matrices $\mathbf{A}^{(n)}$ ($\in \mathbb{R}^{I_n \times J_n}, n = 1, \cdots, N$) and a core tensor $\mathcal{G}$ ($\in \mathbb{R}^{J_1 \times ... \times J_N}$), which minimize (7).

$$L(\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \sum_{\forall \alpha \in \Omega} \left(\mathcal{X}_\alpha - \sum_{\forall \beta \in \mathcal{G}} \mathcal{G}_\beta \prod_{n=1}^{N} a^{(n)}_{i_n j_n} \right)^2 + \sum_{\forall \alpha \notin \Omega} \left(\sum_{\forall \beta \in \mathcal{G}} \mathcal{G}_\beta \prod_{n=1}^{N} a^{(n)}_{i_n j_n} \right)^2 + \lambda \sum_{n=1}^{N} \|\mathbf{A}^{(n)}\|^2 \quad (7)$$

Note that there is an additional term (second term in (7)) for zero-value entries in the loss function compared to (6), which contributes to increasing the time complexity of FOTF.

Definition 7 (Alternating Least Squares). To minimize the loss functions (6) and (7), an alternating least squares (ALS) technique is widely used [4], which updates a factor matrix or a core tensor while keeping all others fixed.

Algorithm 1: Tucker-ALS

Input : Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, and core tensor dimensionality $J_1, ..., J_N$.
Output: Updated factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ $(n = 1, ..., N)$, and updated core tensor $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$.

1 initialize all factor matrices $\mathbf{A}^{(n)}$
2 **repeat**
3 **for** $n = 1...N$ **do**
4 $\mathcal{Y} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)\mathbf{T}} \cdots \times_{n-1} \mathbf{A}^{(n-1)\mathbf{T}} \times_{n+1} \mathbf{A}^{(n+1)\mathbf{T}} \cdots \times_N \mathbf{A}^{(N)\mathbf{T}}$
5 $\mathbf{A}^{(n)} \leftarrow J_n$ leading left singular vectors of $\mathcal{Y}_{(n)}$
6 **until** *the max. iteration or reconstruction error converges*;
7 $\mathcal{G} \leftarrow \mathcal{X} \times_1 \mathbf{A}^{(1)\mathsf{T}} \cdots \times_N \mathbf{A}^{(N)\mathsf{T}}$

Algorithm 1 describes a conventional Tucker factorization based on the ALS, which is called the *higher-order orthogonal iteration* (HOOI) (see [4] for details). The computational and memory bottleneck of Algorithm 1 is updating factor matrices $\mathbf{A}^{(n)}$ (lines 4-5), which requires tensor operations and SVD. Specifically, Algorithm 1 requires storing a full-dense matrix $\mathcal{Y}_{(n)}$, and the amount of memory needed for storing $\mathcal{Y}_{(n)}$ is $O(I_n \prod_{m \neq n} J_m)$. The required memory grows rapidly when the order, the dimensionality, or the rank of a tensor increase, and ultimately causes *intermediate data explosion* [9]. Moreover, Algorithm 1 computes SVD for a given $\mathcal{Y}_{(n)}$, where the complexity of exact SVD is $O(\min(I_n \prod_{m \neq n} J_m^2, I_n^2 \prod_{m \neq n} J_m))$. The computational costs for SVD increase rapidly as well for a large-scale tensor. Notice that Algorithm 1 assumes missing entries of $\mathcal{X}$ as zeros during the update process (lines 4-5), and core tensor $\mathcal{G}$ (line 7) is uniquely determined and relatively easy to be computed by an input tensor and factor matrices.

In summary, applying the naive Tucker-ALS algorithm on sparse tensors generates severe accuracy and scalability issues. Therefore, Algorithm 1 needs to be revised to focus only on observed entries and scale for large-scale tensors at the same time. In that case, an alternative ALS approach is applicable to Algorithm 1, which is utilized for partially observable matrices and CP factorizations. The alternative ALS approach is discussed in Section 3.

Definition 8 (Intermediate Data). We define intermediate data as memory requirements for updating $\mathbf{A}^{(n)}$ (lines 4-5 in Algorithm 1), excluding memory space for storing $\mathcal{X}$, $\mathcal{G}$, and $\mathbf{A}^{(n)}$. The size of intermediate data plays a critical role in determining which Tucker factorization algorithms are space-efficient, as we will discuss in Section 3.5.2.

2.4 Heterogeneous Computing

Heterogeneous computing refers to systems that use more than one kind of processor or cores. These systems gain performance efficiency by adding dissimilar co-processors [17]. A well-known example of heterogeneous platforms is a machine with CPUs and GPUs [18]. According to [19], even a single GPU-CPU framework provides advantages which multiple CPUs on their own do not offer due to the specialization in each chip.

OpenCL [20]. How can we write programs executed well across those heterogeneous platforms? The answer is using OpenCL, which is a parallel programming standard for heterogeneous platforms. The key advantages of OpenCL are 1) its generality that can be applied to various processors (such as GPUs, DSPs, or FPGAs), and 2) easy-to-use abstractions and a broad set of programming APIs for manipulating accelerators. The specification document[1] provides detailed information about OpenCL.

SnuCL [21]. OpenCL operates on only a single node. In order to run OpenCL applications for distributed-GPU environment, an additional abstraction layer that communicates with OpenCL is needed. SnuCL is a communication library written in MPI that provides an illusion that all computing nodes are aggregated to a single node with multiple processors. SnuCL allows us to run OpenCL applications implemented for a single node on distributed-GPU environments without additional implementations.

3 Proposed Methods

In this section, we describe GTA, our generalized Tucker factorization algorithm on heterogeneous platforms. As described in Definition 7, the computational and memory bottleneck of the standard Tucker-ALS algorithm occurs while updating factor matrices. Therefore, it is imperative to update them efficiently in order to maximize speed and scalability. However, there are several challenges in designing an optimized algorithm for updating factor matrices.

1) **Maximize scalability.** The aforementioned Tucker-ALS algorithm suffers from *intermediate data explosion* and high computational costs while updating factor matrices. How can we formulate efficient algorithms for updating factor matrices in terms of time and memory?
2) **Exploiting GPUs during factorizations.** In order to fully exploit the performance of GPUs, a target task ought to be parallelized and computationally intensive. How can we find and accelerate those GPU-suitable tasks during updating factor matrices?
3) **Avoiding heavy computational costs of fully observable tensor factorization (FOTF).** FOTF requires more computations than those of partially observable one due to numerous zero-value entries in a loss function. How can we devise an efficient algorithm for FOTF?

To overcome the above challenges, we suggest the following main ideas, which we describe in later subsections.

1) **A row-wise update rule** is used for updating factor matrices. The update rule minimizes memory requirement for intermediate data (Figure 3 and Section 3.2).

1. https://www.khronos.org/registry/OpenCL/specs/2.2/pdf/OpenCL_API.pdf

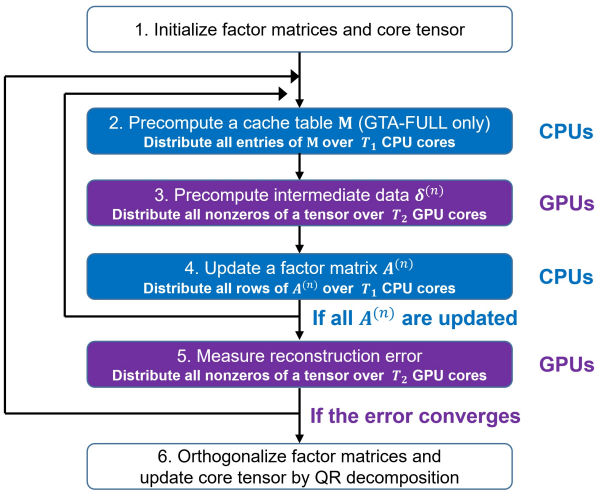

Fig. 2: An overview of GTA. After initialization, GTA iteratively updates factor matrices using CPUs and GPUs. When reconstruction error converges, GTA makes factor matrices orthogonal and updates a core tensor by QR factorization.

2) **GTA-PART** utilizes GPUs for calculating intermediate data δ by nonzero-based parallelization, while the rest parts are computed by CPUs (Section 3.3).
3) **GTA-FULL** also employs GPUs for computing δ as GTA-PART does and significantly reduces the number of computations required for FOTF using a factorization technique with a table $\mathcal{M}^{(n)}$ (Section 3.4).

We first suggest an overview of how GTA factorizes given tensors using Tucker method in Section 3.1. After that, we describe details of our main ideas in Sections 3.2∼3.4, and we offer a theoretical analysis of GTA in Section 3.5.

3.1 Overview

Our proposed method GTA consists of GTA-PART and GTA-FULL which provide a fast and scalable Tucker factorization for partially and fully observable tensors, respectively. GTA-PART extends upon the default version of P-Tucker [13] to run on heterogeneous platforms. While P-Tucker uses only row-wise parallelization (Section 3.3) on CPUs, GTA-PART utilizes both row-wise (CPUs) and nonzero-wise (GPUs) parallelization. GTA-FULL further extends GTA-PART to efficiently run on fully observable tensors by reducing heavy computational costs based on careful reformulation of the optimization function (Section 3.4).

Figure 2 and Algorithm 2 describe the main process of GTA (includes both GTA-PART and GTA-FULL). First, GTA initializes all $\mathbf{A}^{(n)}$ and $\mathcal{G}$ with random real values between 0 and 1 (step 1 and line 1). After that, GTA iteratively updates factor matrices (steps 2-4 and lines 3-18; see Section 3.2) When all factor matrices are updated, GTA measures reconstruction error using (5) on GPUs (step 5 and line 19; more details are given in the Supplementary Material [22]). GTA stops iterations if the error converges or the maximum iteration is reached (line 20). Finally, GTA performs QR decomposition on all $\mathbf{A}^{(n)}$ to make them orthogonal and updates $\mathcal{G}$ (step 6 and lines 21-24). Specifically, QR decomposition [23] on each $\mathbf{A}^{(n)}$ is defined as follows:

Algorithm 2: GTA Algorithm

Input : Tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times \cdots \times I_N}$, factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ $(n = 1, ..., N)$, and core tensor $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$.

Output: Updated factor matrices $\mathbf{A}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ $(n = 1, ..., N)$ and core tensor $\mathcal{G} \in \mathbb{R}^{J_1 \times J_2 \times \cdots \times J_N}$.

1 initialize $\mathbf{A}^{(n)}$ and $\mathcal{G}$ with random values between 0 and 1
2 **repeat**
3 **for** n = 1...N **do**
4 **if** GTA-FULL **then**
 ▷ **Compute an intermediate table $\mathcal{M}^{(n)}$**
5 **for** $\beta = \forall(j_1, ..., j_N) \in \mathcal{G}$ **do** ▷ **parallel, CPU**
6 **for** $\gamma = \forall(j_1, ..., j_N) \in \mathcal{G}$ **do**
7 $\mathcal{M}^{(n)}[\beta][\gamma] \leftarrow \mathcal{G}_\beta \mathcal{G}_\gamma \left(\sum_{\forall \alpha \in \mathcal{X}_{i_n}^{(n)}} \prod_{k \neq n} (a^{(k)}_{\alpha_k \beta_k} a^{(k)}_{\alpha_k \gamma_k}) \right)$
8 calculate $\mathbf{B}^{(n)}$ using (12)
 ▷ **Precompute $\delta^{(n)}$ for GTA-PART & GTA-FULL**
9 **for** $\alpha = \forall(i_1, ..., i_N) \in \Omega$ **do** ▷ **parallel, GPU**
10 **for** $\beta = \forall(j_1, ..., j_N) \in \mathcal{G}$ **do**
11 $\delta^{(n)}_\alpha(j_n) \leftarrow \delta^{(n)}_\alpha(j_n) + \mathcal{G}_\beta \prod_{k \neq n} a^{(k)}_{i_k j_k}$
12 **for** i_n = 1...I_n **do** ▷ **parallel, CPU**
13 **for** $\alpha = \forall(i_1, ..., i_N) \in \Omega^{(n)}_{i_n}$ **do**
14 **if** GTA-PART **then**
15 calculate $\mathbf{B}^{(n)}_{i_n}$ using (11)
16 calculate $\mathbf{c}^{(n)}_{i_n:}$ using (13)
17 find an inverse of $[\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]$ or $[\mathbf{B}^{(n)} + \lambda \mathbf{I}_{J_n}]$
18 update $[a^{(n)}_{i_n 1}, \cdots, a^{(n)}_{i_n J_n}]$ using (10)
19 calculate reconstruction error on GPUs using (5)
20 **until** *the maximum iteration or* $\|\mathcal{X} - \mathcal{X}'\|$ *converges;*
21 **for** n = 1...N **do**
22 $\mathbf{A}^{(n)} \rightarrow \mathbf{Q}^{(n)}\mathbf{R}^{(n)}$ ▷ **QR decomposition**
23 $\mathbf{A}^{(n)} \leftarrow \mathbf{Q}^{(n)}$ ▷ **Orthogonalize $\mathbf{A}^{(n)}$**
24 $\mathcal{G} \leftarrow \mathcal{G} \times_n \mathbf{R}^{(n)}$ ▷ **Update core tensor $\mathcal{G}$**

$$\mathbf{A}^{(n)} = \mathbf{Q}^{(n)}\mathbf{R}^{(n)},\ n = 1...N \tag{8}$$

where $\mathbf{Q}^{(n)} \in \mathbb{R}^{I_n \times J_n}$ is *column-wise orthonormal* and $\mathbf{R}^{(n)} \in \mathbb{R}^{J_n \times J_n}$ is *upper-triangular*. Therefore, by substituting $\mathbf{Q}^{(n)}$ for $\mathbf{A}^{(n)}$, GTA succeeds in making factor matrices orthogonal. Core tensor $\mathcal{G}$ must be updated accordingly in order to maintain the same reconstruction error. According to [24], the update rule of core tensor $\mathcal{G}$ is given as follows:

$$\mathcal{G} \leftarrow \mathcal{G} \times_1 \mathbf{R}^{(1)} \cdots \times_N \mathbf{R}^{(N)}. \tag{9}$$

3.2 A Row-wise Update Rule of Factor Matrices

GTA updates factor matrices in a row-wise manner based on ALS. From a high-level point of view, as most ALS methods do, GTA updates a factor matrix at a time while maintaining all others fixed. However, when all other matrices are fixed, there are several approaches [14] for updating a single factor matrix. Among them, GTA selects a row-wise update method; a key benefit of the row-wise update is that all rows of a factor matrix are independent of each other in terms of minimizing the loss functions (Equations (6) and (7)). This property enables applying multi-core parallelism on updating factor matrices. Given a row of a factor matrix, an update rule is derived by computing a gradient with respect to the given row and setting it to zero, which minimizes the loss functions (6) and (7). The update rule for the i_nth row of $\mathbf{A}^{(n)}$ (see Figure 3) is given as follows; proofs of Equation (10) are provided in Theorems 1 and 2.

$$\begin{aligned} [a^{(n)}_{i_n 1}, ..., a^{(n)}_{i_n J_n}] &\leftarrow \underset{[a^{(n)}_{i_n 1}, ..., a^{(n)}_{i_n J_n}]}{\arg\min} L(\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) \\ &= \mathbf{c}^{(n)}_{i_n:} \times [\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]^{-1} \ \text{(GTA-PART)} \\ \text{or } &\ \mathbf{c}^{(n)}_{i_n:} \times [\mathbf{B}^{(n)} + \lambda \mathbf{I}_{J_n}]^{-1} \ \text{(GTA-FULL)} \end{aligned} \tag{10}$$

where $\mathbf{B}^{(n)}_{i_n}$ is a $J_n \times J_n$ matrix whose (j_1, j_2)th entry is

$$\sum_{\forall \alpha \in \Omega^{(n)}_{i_n}} \delta^{(n)}_\alpha(j_1) \delta^{(n)}_\alpha(j_2), \tag{11}$$

$\mathbf{B}^{(n)}$ is a $J_n \times J_n$ matrix whose (j_1, j_2)th entry is

$$\sum_{\forall \alpha \in \mathcal{X}^{(n)}_{i_n}} \delta^{(n)}_\alpha(j_1) \delta^{(n)}_\alpha(j_2), \tag{12}$$

$\mathbf{c}^{(n)}_{i_n:}$ is a length J_n vector whose jth entry is

$$\sum_{\forall \alpha \in \Omega^{(n)}_{i_n}} \mathcal{X}_\alpha \delta^{(n)}_\alpha(j), \tag{13}$$

$\delta^{(n)}_\alpha$ is a length J_n vector whose jth entry is

$$\sum_{\forall \beta \in \mathcal{G}, \beta_n = j} \mathcal{G}_\beta \prod_{k \neq n} a^{(k)}_{i_k j_k}, \tag{14}$$

α and β indicate entries of tensor $\mathcal{X}$ and $\mathcal{G}$, respectively. $\Omega^{(n)}_{i_n}$ or $\mathcal{X}^{(n)}_{i_n}$ indicates the subset of Ω or $\mathcal{X}$ whose nth mode's index is i_n, respectively. λ is a regularization parameter, and $\mathbf{I}_{J_n}$ is a $J_n \times J_n$ identity matrix. As shown in the update rule (10), GTA-PART and GTA-FULL share two intermediate data $\mathbf{c}^{(n)}_{i_n:}$ and $\delta^{(n)}_\alpha$, while $\mathbf{B}^{(n)}_{i_n}$ and $\mathbf{B}^{(n)}$ are exclusively used for GTA-PART and GTA-FULL, respectively. The important property is that $\mathbf{B}^{(n)}_{i_n}$, $\mathbf{c}^{(n)}_{i_n:}$, and $\delta^{(n)}_\alpha$ are computed only by the subset of observable entries $\Omega^{(n)}_{i_n}$. Hence, GTA-PART fully exploits the sparsity of given tensors. In a case of GTA-FULL, naive calculation of $\mathbf{B}^{(n)}$ requires huge computational costs as Equation (12) is calculated by all entries of a tensor. Thus, GTA-FULL utilizes a factorization technique to minimize the costs (see Section 3.4 for details).

Lines 3-18 of Algorithm 2 describe how GTA updates factor matrices. In the case of GTA-FULL, it computes a table $\mathcal{M}^{(n)}$ used for computing $\mathbf{B}^{(n)}$ (lines 4-8; see Section 3.4). GTA precomputes intermediate data $\delta^{(n)}$ by nonzero-based parallelization on GPUs (lines 9-11; see Section 3.3). Then, GTA chooses a row $\mathbf{a}^{(n)}_{i_n:}$ of a factor matrix $\mathbf{A}^{(n)}$ to update (line 12). After that, GTA-PART computes $\mathbf{B}^{(n)}_{i_n}$ and $\mathbf{c}^{(n)}_{i_n:}$ required for updating a row $\mathbf{a}^{(n)}_{i_n:}$, while GTA-FULL only calculates $\mathbf{c}^{(n)}_{i_n:}$ (lines 13-16). GTA-PART and GTA-FULL perform a matrix inverse operation on $[\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]$ and $[\mathbf{B}^{(n)} + \lambda \mathbf{I}_{J_n}]$, respectively (line 17). Finally, GTA-PART and GTA-FULL update a row $\mathbf{a}^{(n)}_{i_n:}$ by Equation (10) (line 18).

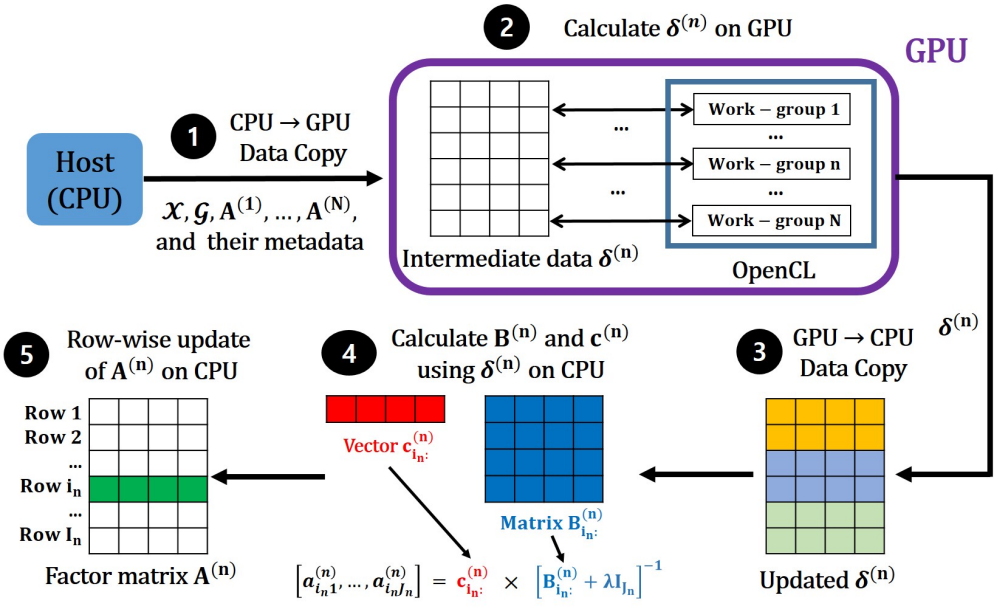

Fig. 3: An overview of GTA-PART for updating a factor matrix $\mathbf{A}^{(n)}$. GTA-PART first sends required data for computing intermediate data $\delta^{(n)}$ such as $\mathcal{X}$ or $\mathcal{G}$ from CPU to GPU(s). After that, GTA-PART calculates $\delta^{(n)}$ using nonzero-based parallelization on GPU(s) ($\delta^{(n)}$ is computed by all nonzeros of an input tensor $\mathcal{X}$). GTA-PART receives updated $\delta^{(n)}$ from GPU(s) and calculates other intermediate data $\mathbf{B}^{(n)}$ and $\mathbf{c}^{(n)}$ using precomputed $\delta^{(n)}$ on CPUs. Finally, GTA-PART updates a factor matrix $\mathbf{A}^{(n)}$ in a row-wise manner with multiple CPUs. λ is a regularization parameter, and $\mathbf{I}_{J_n}$ is a $J_n \times J_n$ identity matrix.

GTA utilizes GPUs to accelerate computational bottlenecks; meanwhile, less computational tasks (e.g., updating a row using (10)) and tasks that are hard to be parallelized (e.g., computing a matrix inverse) are processed by CPUs. There are two sections where CPU parallelization is applied in Algorithm 2. The first section (lines 5-7) is for computing a table $\mathcal{M}^{(n)}$, and the second section (lines 12-18) is for updating factor matrices in a row-wise manner. For each section, GTA carefully distributes tasks to CPUs while maintaining the independence between them. Furthermore, GTA utilizes a dynamic scheduling method offered by OpenMP library [25] to assure that CPU workload is balanced. The details of how GTA parallelizes each section are given as follows.

- **Section 1: Computing a table $\mathcal{M}^{(n)}$.** All entries of $\mathcal{M}^{(n)}$ are independent of each other during their updates. Therefore, GTA distributes all entries of $\mathcal{M}^{(n)}$ uniformly to each CPU, and updates them in parallel.
- **Section 2: Updating factor matrices.** All rows of $\mathbf{A}^{(n)}$ are independent of each other regarding minimizing the loss functions (6) and (7). Therefore, GTA distributes all rows uniformly to each CPU, and updates them in parallel. Since $|\Omega^{(n)}_{i_n}|$ differs for each row, the workload of each CPU may vary considerably. Thus, GTA employs dynamic scheduling in this part.

3.3 GTA-PART for Partially Observable Tensors

GTA-PART is a GPU-accelerated Tucker factorization method for partially observable tensors. GTA-PART utilizes GPUs for calculating intermediate data $\delta^{(n)}$ in order to accelerate the update process. For the other parts that are less computational (e.g., updating a row $a^{(n)}_{i_n:}$) or tricky to be parallelized (e.g., computing a matrix inverse), GTA-PART allows CPUs to handle them.

A key idea of GTA-PART is computing intermediate data δ (computational bottleneck; refer to Proof 4 for detailed analysis) by massive parallelism with GPUs. As shown in Equation (14), a vector $\delta^{(n)}_\alpha$ corresponds to a single nonzero [2] (or α) of $\mathcal{X}$. Thus, if we accumulate all $\delta^{(n)}_\alpha$, a matrix or table $\delta^{(n)}$ corresponds to all nonzeros of $\mathcal{X}$. Since all nonzeros of an input tensor $\mathcal{X}$ are independent of each other in terms of computing $\delta^{(n)}_\alpha$, GTA-PART calculates $\delta^{(n)}$ using nonzero-based parallelization before updating $\mathbf{A}^{(n)}$. The following process describes how GTA-PART utilizes GPUs for computing intermediate data $\delta^{(n)}$.

- 1. Initializing GPU environments: GTA-PART first compiles GPU kernel codes and initializes OpenCL variables (e.g., devices, queues, ...) and memory objects.
- 2. CPU → GPU Data Copy: GTA-PART sends tensor $\mathcal{X}$, factor matrices $\mathbf{A}^{(n)}$, core tensor $\mathcal{G}$, and their metadata to GPUs for computing $\delta^{(n)}$ as a form of OpenCL memory objects. For multi- and distributed-GPU environments, GTA-PART equally distributes nonzeros of a tensor to each GPU.
- 3. Executing GPU Kernel: Each work-group of GPUs fills out values of $\delta^{(n)}_\alpha$ using Equation (14) in parallel for their allocated nonzeros.
- 4. GPU → CPU Data Copy: For a single-GPU environment, GTA-PART receives a fully-computed $\delta^{(n)}$ from the GPU. However, for multi- and distributed-GPU environments, GTA-PART accumulates all partial $\delta^{(n)}$ from each GPU and builds a complete $\delta^{(n)}$.

Figure 3 summarizes how GTA-PART updates a factor

2. We use a term 'nonzero' instead of 'observable entry' for simplicity.

matrix $\mathbf{A^{(n)}}$ while fixing all the parameters. GTA-PART first computes intermediate data $\delta^{(n)}$ by the above procedure. Once $\delta^{(n)}$ is computed, GTA-PART updates $\mathbf{A^{(n)}}$ using row-wise parallelization on CPUs. Specifically, given a row $a^{(n)}_{i_n:}$, GTA-PART calculates $\mathbf{B}^{(n)}_{i_n}$ and $\mathbf{c}^{(n)}_{i_n:}$ using precomputed $\delta^{(n)}$ and updates the given row by Equation (10). After updating all rows of $\mathbf{A^{(n)}}$, GTA-PART iterates the same procedure for other factor matrices.

In summary, GTA-PART is a heterogeneous computing algorithm which accelerates the computational bottleneck by GPUs and handles the rest parts on CPUs.

3.4 GTA-FULL: An Extension of GTA-PART to FOTF

GTA-FULL is a GPU-accelerated algorithm for fully observable Tucker factorization (FOTF). As GTA-FULL is an extension of GTA-PART to FOTF, overall process of GTA-FULL is similar to that of GTA-PART (e.g., exploiting GPUs for computing δ). The main difference between GTA-FULL and GTA-PART is that GTA-FULL uses a factorization technique to reduce heavy computational costs from many zero-value entries. Before explaining the technique, we need to consider the following question: why cannot we just apply GTA-PART directly to FOTF? The fundamental problem of applying GTA-PART to FOTF is that computing intermediate data $\mathbf{B}^{(n)}_{i_n}$ becomes a gigantic computational bottleneck even with GPUs. Update rules of GTA-FULL for $a^{(n)}_{i_n:}$, $\mathbf{c}^{(n)}_{i_n:}$, and $\delta^{(n)}_{\alpha}$ are derived the same with those of GTA-PART when we compute a gradient of the loss function (7). However, an update rule of GTA-FULL for $\mathbf{B}^{(n)}_{i_n}$ becomes different from that of GTA-PART, and the equation is given by the following: $\mathbf{B}^{(n)}$ is a $J_n \times J_n$ matrix whose (j_1, j_2)th entry is

$$\sum_{\forall \alpha \in \mathbf{x}^{(n)}_{i_n}} \left(\sum_{\forall \beta \in \mathcal{G}^{(n)}_{j_1}} \mathcal{G}_\beta \prod_{k \neq n} a^{(k)}_{i_k \beta_k} \right) \left(\sum_{\forall \gamma \in \mathcal{G}^{(n)}_{j_2}} \mathcal{G}_\gamma \prod_{k \neq n} a^{(k)}_{i_k \gamma_k} \right) \tag{15}$$

Note that $\mathcal{G}^{(n)}_j$ indicates a set of core tensor entries whose nth mode's index is j. As shown in Equation (15), there are two differences between GTA-FULL and GTA-PART regarding $\mathbf{B}^{(n)}$. First, $\mathbf{B}^{(n)}$ of GTA-FULL is shared for all rows of $\mathbf{A^{(n)}}$. When a row i_n varies in Equation (15), the nth mode's index of $\mathbf{x}^{(n)}_{i_n}$ differs while the other indices remain the same. Since the index i_n has no effect on Equation (15) due to the $k \neq n$ term, $\mathbf{B}^{(n)}$ is the same for all rows of $\mathbf{A^{(n)}}$. Second, a computational cost of computing $\mathbf{B}^{(n)}$ tremendously increases as $\mathbf{B}^{(n)}$ is calculated by both nonzero- and zero-value entries of $\mathcal{X}$ while $\mathbf{B}^{(n)}_{i_n}$ is only computed by nonzeros of $\mathcal{X}$ (e.g., GTA-FULL requires about 3.7×10^{11} floating point operations for computing $\mathbf{B}^{(n)}$ of the MovieLens tensor, which is $20000\times$ larger than that of GTA-PART).

Then, how can we minimize the cost of calculating $\mathbf{B}^{(n)}$? The answer is using a factorization technique. To understand the main idea of a factorization technique, we suggest the following toy problem. As shown in Figure 4, a naive solution of the toy problem is aggregating multiplication results for all possible index combinations, which has exponential time complexity $O(NI^N)$. However, a factorization technique expressed as multiplications of each array's summation ($\prod_k \sum_{i=1}^{i=I} A^{(k)}_i$) remarkably reduces time complexity to $O(NI)$. Unfortunately, it is not straightforward to apply the factorization technique to Equation (15) as core tensor values ($\mathcal{G}_\beta$ and $\mathcal{G}_\gamma$) are weighted to multiplication results. Hence, we transform (15) into the following Equation (16).

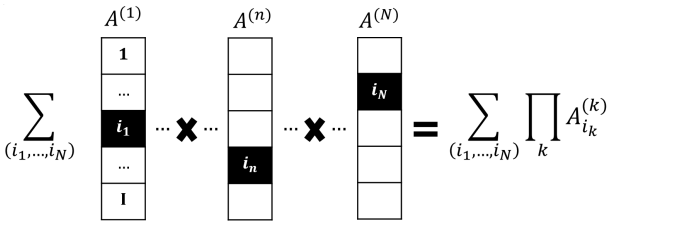

Fig. 4: A toy problem for computing $\mathbf{B}^{(n)}$ efficiently. The problem is computing summations of multiplications, which is a simplified version of Equation (15). The solution is using a factorization ($\prod_k \sum_{i=1}^{i=I} A^{(k)}_i$) with time complexity $O(NI)$.

$$\sum_{\forall \beta \in \mathcal{G}^{(n)}_{j_1}} \sum_{\forall \gamma \in \mathcal{G}^{(n)}_{j_2}} \mathcal{G}_\beta \mathcal{G}_\gamma \left(\sum_{\forall \alpha \in \mathbf{x}^{(n)}_{i_n}} \prod_{k \neq n} (a^{(k)}_{i_k \beta_k} a^{(k)}_{i_k \gamma_k}) \right) = \sum_{\forall \beta \in \mathcal{G}^{(n)}_{j_1}} \sum_{\forall \gamma \in \mathcal{G}^{(n)}_{j_2}} \mathcal{M}^{(n)}[\beta][\gamma], \tag{16}$$

$$\text{where } \mathcal{M}^{(n)}[\beta][\gamma] = \mathcal{G}_\beta \mathcal{G}_\gamma \left(\sum_{\forall \alpha \in \mathbf{x}^{(n)}_{i_n}} \prod_{k \neq n} (a^{(k)}_{i_k \beta_k} a^{(k)}_{i_k \gamma_k}) \right).$$

After deciding core tensor entries β and γ in (16), a factorization technique is applicable to computing an element of a table $\mathcal{M}^{(n)} \in \mathcal{R}^{|\mathcal{G}| \times |\mathcal{G}|}$. Correspondences between calculating $\mathcal{M}^{(n)}[\beta][\gamma]$ and the toy problem are given as follows (Table 3).

TABLE 3: Correspondences between calculating $\mathcal{M}^{(n)}[\beta][\gamma]$ and the toy problem. The factorization technique from the toy problem is used for computing $\mathcal{M}^{(n)}[\beta][\gamma]$ efficiently.

	Toy problem	**Calculating $\mathcal{M}^{(n)}[\beta][\gamma]$**
Number of arrays	N	$N-1$
Array size	I	$I_1, ..., I_{n-1}, I_{n+1}, ..., I_N$
Selected value from the kth array	$A^{(k)}_{i_k}$	$a^{(k)}_{i_k \beta_k} a^{(k)}_{i_k \gamma_k}$

Without the factorization technique, computing $\mathcal{M}^{(n)}[\beta][\gamma]$ takes $O(NI^{N-1})$ (assuming all $I_1 = ... = I_N = I$ for simplicity). However, it is reduced to $O(NI)$ by factorization. After filling out all entries of the table $\mathcal{M}^{(n)}$ using factorization, GTA-FULL finally updates $\mathbf{B}^{(n)}$ by the following equation.

$$\mathbf{B}^{(n)}[j_1][j_2] = \sum_{\forall \beta \in \mathcal{G}^{(n)}_{j_1}} \sum_{\forall \gamma \in \mathcal{G}^{(n)}_{j_2}} \mathcal{M}^{(n)}[\beta][\gamma] \tag{17}$$

Regarding the other data such as $a^{(n)}_{i_n:}$, $\mathbf{c}^{(n)}_{i_n:}$, and $\delta^{(n)}_{\alpha}$, GTA-FULL uses the same procedure with that of GTA-PART to update them. In summary, GTA-FULL is based on GTA-PART and tackles the computational bottleneck for computing $\mathbf{B}^{(n)}$ by a factorization technique.

TABLE 4: Complexity analysis of GTA and other methods with respect to time and memory. The optimal complexities are in bold. Note that memory complexity indicates the space requirement for intermediate data. $|\Omega|$ is the number of observable entries in a given tensor $\mathcal{X}$, and T_1 and T_2 are the numbers of CPU and GPU cores, respectively ($T_1 << T_2$).

Algorithm	Time Complexity (per iteration)	Memory Complexity
GTA-PART	$\mathbf{O(NIJ^3/T_1 + N^2\|\Omega\|J^N/T_2)}$	$O(J\|\Omega\|)$
GTA-FULL	$O(N^2IJ^{2N}/T_1 + N^2\|\Omega\|J^N/T_2)$	$O(J\|\Omega\| + J^{2N})$
P-TUCKER	$O(NIJ^3/T_1 + N^2\|\Omega\|J^N/T_1)$	$\mathbf{O(J^2T_1)}$
SPLATT	$O(NJ^{N-1}(\|\Omega\| + J^{2(N-1)})/T_1)$	$O(IJ^{N-1})$

3.5 Theoretical Analysis

3.5.1 Convergence Analysis

In this section, we theoretically prove the correctness and the convergence of GTA-PART and GTA-FULL.

Theorem 1 (Correctness of* GTA-PART*). The proposed row-wise update rule (18) minimizes the loss function (6) regarding the updated parameters.

$$\underset{[a^{(n)}_{i_n1},...,a^{(n)}_{i_nJ_n}]}{\arg\min} \quad L(\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \mathbf{c}^{(n)}_{i_n:} \times [\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]^{-1} \tag{18}$$

Proof 1.

$$\frac{\partial L}{\partial a^{(n)}_{i_nj_n}} = 0, \forall j_n, 1 \le j_n \le J_n$$

$$\Leftrightarrow \sum_{\forall \alpha \in \Omega^{(n)}_{i_n}} \left(\left(\mathcal{X}_\alpha - \sum_{\forall \beta \in \mathcal{G}} \mathcal{G}_\beta \prod_{n=1}^{N} a^{(n)}_{i_nj_n} \right) \times \left(-\delta^{(n)}_\alpha(j_n) \right) \right) + \lambda a^{(n)}_{i_nj_n} = 0$$

$$\Leftrightarrow [a^{(n)}_{i_n1}, ..., a^{(n)}_{i_nJ_n}] \left(\sum_{\forall \alpha \in \Omega^{(n)}_{i_n}} \left(\delta^{(n)\mathbf{T}}_\alpha \delta^{(n)}_\alpha \right) + \lambda \mathbf{I}_{J_n} \right) = \sum_{\forall \alpha \in \Omega^{(n)}_{i_n}} \left(\mathcal{X}_\alpha \delta^{(n)}_\alpha \right)$$

$$\Leftrightarrow [a^{(n)}_{i_n1}, ..., a^{(n)}_{i_nJ_n}] = \mathbf{c}^{(n)}_{i_n:} \times [\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]^{-1}$$

■

Note that the full proof of Theorem 1 is in the Supplementary Material of GTA [22].

Theorem 2 (Correctness of* GTA-FULL*). The proposed row-wise update rule (19) minimizes the loss function (7) regarding the updated parameters.

$$\underset{[a^{(n)}_{i_n1},...,a^{(n)}_{i_nJ_n}]}{\arg\min} \quad L(\mathcal{G}, \mathbf{A}^{(1)}, ..., \mathbf{A}^{(N)}) = \mathbf{c}^{(n)}_{i_n:} \times [\mathbf{B}^{(n)} + \lambda \mathbf{I}_{J_n}]^{-1} \tag{19}$$

Proof 2. Please refer to the Supplementary Material [22]. ■

Theorem 3 (Convergence of* GTA-PART *and* GTA-FULL*). GTA-PART and GTA-FULL converge to local minima or saddle points.

Proof 3. The loss functions (6) and (7) are non-increasing since the update rule (10) with *argmin* function makes loss functions unchanged or smaller for every update. ■

We note that the ALS method also exhibits a local linear convergence for CP decomposition [14], [26].

3.5.2 Complexity Analysis

In this section, we analyze time and memory complexities of GTA-PART and GTA-FULL. For simplicity, we assume $I_1 = ... = I_N = I$ and $J_1 = ... = J_N = J$. Table 4 summarizes the time and memory complexities of GTA and other methods. Note that we calculate time complexities per iteration (lines 3-19 in Algorithm 2), and we focus on memory complexities of intermediate data, not of all variables.

Theorem 4 (Time complexity of* GTA-PART*). The time complexity of GTA-PART is $O(NIJ^3/T_1 + N^2|\Omega|J^N/T_2)$.

Proof 4. Please refer to the Supplementary Material [22]. ■

Theorem 5 (Memory complexity of* GTA-PART*). The memory complexity of GTA-PART is $O(J|\Omega|)$.

Proof 5. The intermediate data of GTA-PART consist of a vector $\mathbf{c}^{(n)}_{i_n:}$ ($\in \mathbb{R}^J$) , and three matrices $\delta^{(n)}$ ($\in \mathbb{R}^{J \times |\Omega|}$), $\mathbf{B}^{(n)}_{i_n}$, and $[\mathbf{B}^{(n)}_{i_n} + \lambda \mathbf{I}_{J_n}]^{-1}$ ($\in \mathbb{R}^{J \times J}$). As all memory complexities of other intermediate data are negligible compared to that of $\delta^{(n)}$, The total memory complexity of GTA-PART is equivalent to the memory complexity of $\delta^{(n)}$, which is $O(J|\Omega|)$. ■

Please refer to the Supplementary Material [22] for proofs of GTA-FULL's time and memory complexities.

4 EXPERIMENTS

In this section, we present experimental results of GTA and other methods. We focus on answering the following questions.

1) **Effectiveness of GPUs (Section 4.2).** How much do GPUs accelerate the factorization speed of GTA-PART compared to the state-of-the-art method P-TUCKER?
2) **Effectiveness of the factorization technique (Section 4.3).** How much does the factorization technique accelerate the update process of GTA-FULL compared to the state-of-the-art method SPLATT?
3) **GPU scalability (Section 4.4).** How well do GTA-PART and GTA-FULL scale with respect to the number of GPUs used for parallelization?
4) **Correctness verification (Section 4.5).** How accurately GTA-PART and GTA-FULL factorize given tensors compared to P-TUCKER and SPLATT?

We describe the datasets and experimental settings in Section 4.1, and answer the questions in Sections 4.2 to 4.4.

4.1 Experimental Settings

4.1.1 Datasets

We use both real-world and synthetic tensors to evaluate GTA and competitors. Table 5 summarizes the tensors we used in experiments, which are available at **https://github.com/sejoonoh/GTA-Tensor**. For real-world tensors, we use Netflix, MovieLens, DBLP, and Facebook datasets. Netflix is movie rating data which consist of (user, movie, year-month, rating). MovieLens is movie rating data which consist of (user, movie, year, hour, rating). DBLP is publication data which consist of (author, conference, year, count). Facebook is social network data which consist of (user 1, user 2,

TABLE 5: Summary of real-world and synthetic tensors used for experiments. M: million, K: thousand.

Name	Order	Dimensionality	$\|\Omega\|$	Rank
Netflix	3	(480K, 18K, 74)	100M	10
MovieLens	4	(138K, 27K, 21, 24)	20M	10
DBLP	3	(418K, 4K, 50)	1.3M	10
Facebook	3	(64K, 64K, 870)	1.5M	10
Synthetic	4	(10K, 10K, 10K, 10K)	∼10M	10

date, friendship). Notice that Netflix and MovieLens are partially observable tensors, while DBLP and Facebook are fully observable tensors. For synthetic tensors, we generate random tensors of size $I_1 = I_2 = ... = I_N$ with real-valued entries between 0 and 1. We also assume that the core tensor $\mathcal{G}$ is of size $J_1 = J_2 = ... = J_N$.

4.1.2 Competitors

We compare GTA-PART and GTA-FULL with two state-of-the-art Tucker factorization (TF) methods. Descriptions of all methods are given as follows:

- **P-TUCKER [13]** the scalable TF method for partially observable tensors which minimizes memory complexities of intermediate data by employing a row-wise update rule of factor matrices.
- **SPLATT [12]:** the speed-focused TF algorithm for fully observable tensors which accelerates a tensor-times-matrix chain (TTMc) by a compressed sparse fiber (CSF) structure.

Notice that other Tucker methods (e.g., [10], [11], [27]) are excluded since they present limited scalability or speed compared to that of the competitors mentioned above.

4.1.3 Environment

GTA is implemented in C with OPENMP and OPENCL libraries utilized for CPU and GPU parallelization. For competitors, we use codes provided by the authors (P-TUCKER[3] and SPLATT[4]). We run experiments on a heterogeneous CPU/GPU cluster called "Chundoong[5]". In our experiments, we use up to 8 computing nodes, where each node consists of 16 CPUs (Intel Xeon E5-2650 2.0 GHz) and 4 GPUs (NVIDIA GTX 1080). We note that 'Distributed' environment, which is labeled "Distr.Mult." in figures, indicates multiple nodes (N) with multiple GPUs (M) are used for experiments (in total, NM GPUs). A parameter λ of GTA and P-TUCKER is set to 0.001; for SPLATT, we set the number of CSF allocations to 1 and choose a LAPACK SVD routine for stable convergence. We set the maximum running time per iteration to 2 hours and the maximum number of iterations to 10. In reporting running times, we use average elapsed time per iteration instead of total running time in order to confirm the theoretical complexities (see Table 4), which are analyzed by per-iteration.

4.2 Effectiveness of GPUs

We evaluate how much GPUs accelerate the factorization speed of GTA-PART compared to P-TUCKER using both synthetic and real-world tensors.

3. https://github.com/sejoonoh/P-Tucker
4. https://github.com/ShadenSmith/splatt
5. http://aces.snu.ac.kr/chundoong/

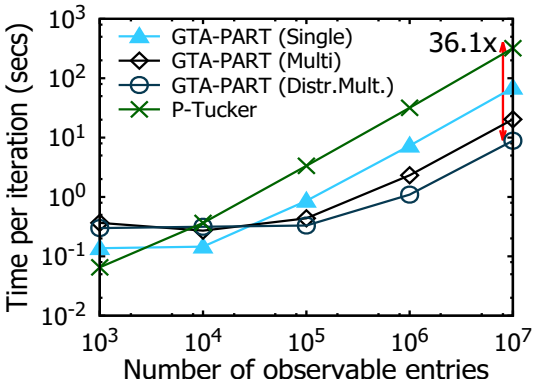

Fig. 5: Performance of GTA-PART and P-TUCKER with respect to the number of observable entries of a partially observable synthetic tensor. Distributed-GPU version of GTA-PART runs up to 36.1× faster than P-TUCKER when $|\Omega| = 10^7$.

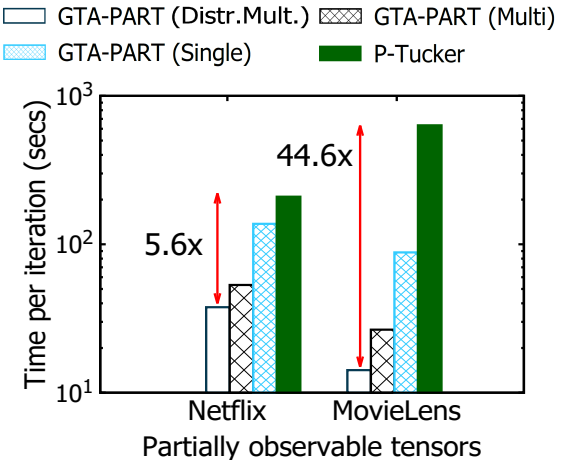

Fig. 6: Performance of GTA-PART and P-TUCKER for real-world partially observable tensors. Distributed-GPU version of GTA-PART shows the fastest factorization speed, which is up to 44.6× faster than P-TUCKER.

4.2.1 Synthetic Data

We increase the number of observable entries ($|\Omega|$) from 10^3 to 10^7, while fixing $N = 4$, $I_n = 10^4$, and $J_n = 10$. As shown in Figure 5, GTA-PART presents the fastest factorization speed when $|\Omega| \geq 10^4$ and runs up to 36.1× faster than P-TUCKER on the largest tensor with $|\Omega| = 10^7$. When $|\Omega|$ is small, P-TUCKER runs faster than GTA-PART as data copying between CPU and GPU becomes a computational bottleneck of GTA-PART, while P-TUCKER does not suffer from that cost. Besides, when $|\Omega|$ is small, distributed-GPU version of GTA-PART runs slower than single-node versions due to communication costs between computing nodes (if $|\Omega|$ is large, communication costs are negligible compared to computational costs). Detailed results of Figure 5 and analysis of (copy + communication) time of GTA-PART are summarized in the Supplementary Material [22].

4.2.2 Real-world Data

We measure the average running time per iteration of GTA-PART and P-TUCKER on the real-world partially observable tensors introduced in Section 4.1.1. Like the results for synthetic tensors, distributed-GPU version of GTA-PART also

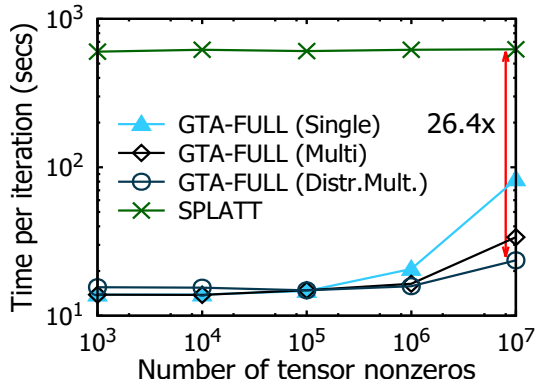

Fig. 7: Performance of GTA-FULL and SPLATT with respect to the number of nonzeros of a fully observable synthetic tensor. Distributed-GPU version of GTA-FULL runs up to 26.4× faster than SPLATT when $|\Omega| = 10^7$.

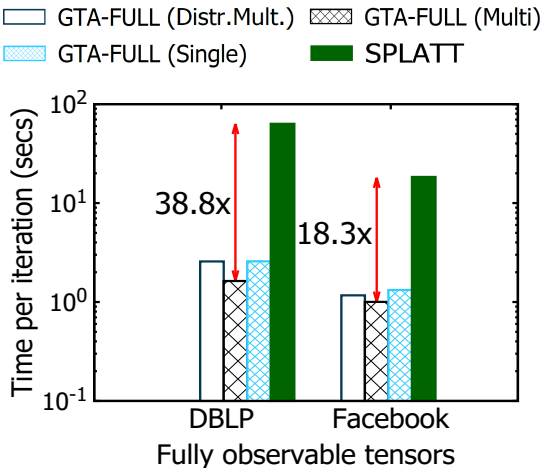

Fig. 8: Performance of GTA-FULL and SPLATT for real-world fully observable tensors. Multi-GPU version of GTA-FULL shows the fastest factorization speed (up to 38.8× compared to SPLATT).

exhibits the fastest speed among all methods (see Figure 6). Note that GTA-PART succeeds in decomposing large-scale real-world tensors and runs $5.6 - 44.6\times$ faster than the competitor.

4.3 Effectiveness of the Factorization Technique

We evaluate how much the factorization technique accelerates the update process of GTA-FULL compared to SPLATT using both synthetic and real-world tensors.

4.3.1 *Synthetic Data*

We increase the number of nonzeros of a tensor ($|\Omega|$) from 10^3 to 10^7, while fixing $N = 4$, $I_n = 10^4$, and $J_n = 10$. As shown in Figure 7, GTA-FULL has the fastest running time for all cases, and distributed-GPU version of GTA-FULL runs up to 26.4× faster than SPLATT when $|\Omega|$ is 10^7. Detailed results of Figure 7 and analysis of (copy + communication) time of GTA-FULL are summarized in the Supplementary Material [22].

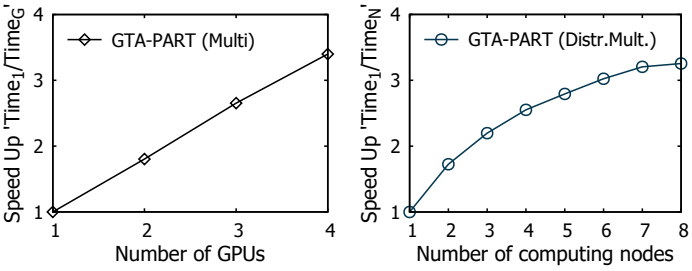

Fig. 9: The GPU scalability of GTA-PART with respect to the number of GPUs (left) and computing nodes (right) used for experiments. GTA-PART presents near-linear scalability regarding the number of GPUs and computing nodes.

4.3.2 *Real-world Data*

We evaluate the performance of GTA-FULL and SPLATT on the real-world fully observable tensors: DBLP and Facebook. Unlike GTA-PART, a distributed-GPU version of GTA-FULL runs slower than a single-node one since the number of tensor nonzeros is not large enough to ignore communication costs (see Figure 8). Notice that GTA-FULL factorizes the real-world tensors $18.3 - 38.8\times$ faster than SPLATT.

4.4 GPU Scalability

We measure the speed-ups of GTA-PART by increasing the number of GPUs and computing nodes from 1 to 8, while fixing $N = 4$, $I_n = 10^4$, $J_n = 10$, and $|\Omega| = 10^7$. $Time_1/Time_{GorN}$ indicates the speed-up, where $Time_{GorN}$ is the running time using G GPUs or N computing nodes, respectively. Figure 9 shows near-linear scalability of GTA-PART with respect to the number of GPUs and computing nodes. GTA-PART scales linearly with a higher coefficient for a single node (left) compared to distributed environments (right) due to the communication costs. We omit GPU scalability results of GTA-FULL since they are similar to those of GTA-PART.

4.5 Correctness Verification

We verify whether our method is correct or not using two accuracy metrics: reconstruction error and test root mean square error (RMSE); the former describes how precisely a method factorizes a given tensor, and the latter indicates how accurately a method estimates values of test data. GTA shows comparable reconstruction error and test RMSE as the state-of-the-art methods, P-TUCKER and SPLATT. Moreover, GTA offers a stable quality of factor matrices regardless of random initialization. The detailed experimental results of GTA for accuracy and robustness are offered in the Supplementary Material [22].

5 Related Work

In this section, we review related works on CP and Tucker factorizations, tensor factorization methods on heterogeneous platforms, and applications of Tucker factorization.

CP Decomposition (CPD). Many algorithms have been developed for scalable CPD. GigaTensor [9] is the first distributed CP method running on the MapReduce framework.

Park et al. [28] propose a distributed algorithm, DBTF, for fast and scalable Boolean CPD. In [29], Papalexakis et al. present a sampling-based, parallelizable method for sparse CPD. AdaTM [30] is an adaptive tensor memoization algorithm for CPD of sparse tensors, which automatically tunes algorithm parameters. Kaya and Uçar [31] propose distributed memory CPD methods based on hypergraph partitioning of sparse tensors. Those algorithms are based on the ALS similarly to the conventional Tucker-ALS.

Since the above CP methods are based on fully observable tensors, scalable CPD methods for partially observable tensors have gained increasing attention in recent years. Karlsson et al. [32] discuss parallel formulations of ALS and CCD++ for tensor completion in the CP format. Smith et al. [33] explore three optimization algorithms for high performance, parallel tensor completion: alternating least squares (ALS), stochastic gradient descent (SGD), and coordinate descent (CCD++). For distributed platforms, Shin et al. [14] propose CDTF and SALS, which are ALS-based CPD methods for partially observable tensors. Note that [14] and [33] offer a row-wise parallelization for CPD as GTA does for Tucker decomposition.

Tucker Factorization (TF). De Lathauwer et al. [6] propose Tucker-ALS, described in Algorithm 1. As the size of real-world tensors increases rapidly, there has been a growing need for scalable TF methods. One major challenge is the "intermediate data explosion" problem [9]. MET (Memory Efficient Tucker) [7] tackles this challenge by adaptively ordering computations and performing them in a piecemeal manner. HaTen2 [8] reduces intermediate data by reordering computations and exploiting the sparsity of real-world tensors in MapReduce. However, both MET and HaTen2 suffer from a limitation called M-bottleneck [10] that arises from explicit materialization of intermediate data. S-HOT [10] avoids M-bottleneck by employing on-the-fly computation. Kaya and Uçar [11] discuss a shared and distributed memory parallelization of an ALS-based TF for sparse tensors. [34] optimizes tensor-times-dense matrix operation for sparse and semi-sparse tensors, which is applied to TF to improve its performance. [35] proposes optimizations of HOOI for dense tensors on distributed systems. The above methods assume fully observable tensors and depend on SVD for updating factor matrices.

For partially observable tensors, only few Tucker methods have been developed including [13]. Liu et al. [36] define the trace norm of a tensor, and present three convex optimization algorithms for low-rank tensor completion. Liu et al. [37] propose a core tensor Schatten 1-norm minimization method with a rank-increasing scheme for tensor factorization and completion.

GPU-based Tensor Factorization Although GPUs are gaining popularity as innovative tools for accelerating tensor factorization (TF), there are only few GPU-based TF methods [38], [39], [27], [34]. [38] provides a GPU-accelerated nonnegative factorization for a 3-way tensor. [39] accelerates tensor operations for CPD by flagged-coordinate structures and 2-dimensional parallelization on GPUs. To the best of our knowledge, [27] is the first GPU-accelerated Tucker method which performs block-wise n-mode product on GPUs. However, it is not scalable as its memory complexity is equivalent to the tensor size $O(I^N)$. Recently, optimizations for TF on CPU and GPU platforms have been presented in [34].

Applications of Tucker Factorization. Sun et al. [40] apply a 3-way TF to a tensor consisting of (users, queries, Web pages) to personalize Web search. Sun et al. [41] propose a framework for content-based network analysis and visualization. TF is also used for analyzing (cancer, gene sets, genes) relations in multi-platform cancer data [42].

6 Conclusion

In this paper, we propose GTA, a general framework for Tucker factorization on heterogeneous platforms. By using ALS with a row-wise update rule, accelerating a computational bottleneck with GPUs, and devising a factorization technique for FOTF, GTA successfully offers time-optimized algorithms with theoretical proof and analysis. GTA runs $5.6-44.6\times$ faster than the state-of-the-art and exhibits near-linear scalability with respect to the number of GPUs and computing nodes. Future works include developing approximation algorithms of GTA that use sparse core tensors for faster factorization speed, or applying sampling techniques on observable entries to accelerate decompositions, while sacrificing little accuracy.

Acknowledgment

This work was supported by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT, and Future Planning (NRF-2016M3C4A7952587, PF Class Heterogeneous High-Performance Computer Development). The ICT at Seoul National University provides research facilities for this study. The Institute of Engineering Research at Seoul National University provided research facilities for this work.

References

[1] J. Zhang, Y. Han, and J. Jiang, "Tucker decomposition-based tensor learning for human action recognition," *Multimedia Systems*, vol. 22, no. 3, pp. 343–353, 2016.
[2] X. Zhang, G. Wen, and W. Dai, "A tensor decomposition-based anomaly detection algorithm for hyperspectral image," *TGRS*, vol. 54, no. 10, pp. 5801–5820, 2016.
[3] N. Zheng, Q. Li, S. Liao, and L. Zhang, "Flickr group recommendation based on tensor decomposition," in *SIGIR*, pp. 737–738, 2010.
[4] T. G. Kolda and B. W. Bader, "Tensor decompositions and applications," *SIAM Review*, vol. 51, no. 3, pp. 455–500, 2009.
[5] L. R. Tucker, "Some mathematical notes on three-mode factor analysis," *Psychometrika*, vol. 31, no. 3, pp. 279–311, 1966.
[6] L. D. Lathauwer, B. D. Moor, and J. Vandewalle, "On the best rank-1 and rank-(R_1 , R_2, ... , R_N) approximation of higher-order tensors," *SIMAX*, vol. 21, no. 4, pp. 1324–1342, 2000.
[7] T. G. Kolda and J. Sun, "Scalable tensor decompositions for multi-aspect data mining," in *ICDM*, pp. 363–372, 2008.
[8] I. Jeon, E. E. Papalexakis, C. Faloutsos, L. Sael, and U. Kang, "Mining billion-scale tensors: algorithms and discoveries," *VLDB J.*, vol. 25, no. 4, pp. 519–544, 2016.
[9] U. Kang, E. E. Papalexakis, A. Harpale, and C. Faloutsos, "Gigatensor: scaling tensor analysis up by 100 times - algorithms and discoveries," in *KDD*, pp. 316–324, 2012.
[10] J. Oh, K. Shin, E. E. Papalexakis, C. Faloutsos, and H. Yu, "S-hot: Scalable high-order tucker decomposition," in *WSDM*, 2017.
[11] O. Kaya and B. Uar, "High performance parallel algorithms for the tucker decomposition of sparse tensors," in *ICPP*, pp. 103–112, 2016.

[12] S. Smith and G. Karypis, "Accelerating the tucker decomposition with compressed sparse tensors," in *Europar*, 2017.
[13] S. Oh, N. Park, L. Sael, and U. Kang, "Scalable tucker factorization for sparse tensors - algorithms and discoveries," in *ICDE*, 2018.
[14] K. Shin, L. Sael, and U. Kang, "Fully scalable methods for distributed tensor factorization," *TKDE*, vol. 29, no. 1, pp. 100–113, 2017.
[15] G. I. Allen, "Regularized tensor factorizations and higher-order principal components analysis," *arXiv preprint arXiv:1202.2476*, 2012.
[16] J. Kim, Y. He, and H. Park, "Algorithms for nonnegative matrix and tensor factorizations: A unified view based on block coordinate descent framework," *J. of Global Optimization*, vol. 58, pp. 285–319, Feb. 2014.
[17] A. Shan, "Heterogeneous processing: a strategy for augmenting moore's law," Jan 2006.
[18] S. Mittal and J. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," vol. 47, 07 2015.
[19] S. Mittal and J. S. Vetter, "A survey of cpu-gpu heterogeneous computing techniques," *ACM Comput. Surv.*, vol. 47, pp. 69:1–69:35, July 2015.
[20] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, pp. 66–73, May 2010.
[21] J. Kim, S. Seo, J. Lee, J. Nah, G. Jo, and J. Lee, "Snucl: An opencl framework for heterogeneous cpu/gpu clusters," ICS '12, pp. 341–352, ACM, 2012.
[22] S. Oh, N. Park, J.-G. Jang, S. Lee, and U. Kang, "Supplementary material for gta." https://datalab.snu.ac.kr/GTA/supple.pdf, 2018.
[23] L. N. Trefethen and D. Bau, *Numerical Linear Algebra*. SIAM, 1997.
[24] T. G. Kolda, "Multilinear operators for higher-order decompositions," tech. rep., Sandia National Laboratories, 2006.
[25] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.
[26] A. Uschmajew, "Local convergence of the alternating least squares algorithm for canonical tensor approximation," *SIAM Journal on Matrix Analysis and Applications*, vol. 33, no. 2, pp. 639–652, 2012.
[27] B. Zou, M. Lan, C. Li, L. Tan, and H. Chen, "Context-aware recommendation using gpu based parallel tensor decomposition," in *Advanced Data Mining and Applications*, pp. 213–226, 2014.
[28] N. Park, S. Oh, and U. Kang, "Fast and scalable distributed boolean tensor factorization," in *ICDE*, 2017.
[29] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos, "Parcube: Sparse parallelizable tensor decompositions," in *ECML PKDD*, pp. 521–536, 2012.
[30] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc, "Model-driven sparse cp decomposition for higher-order tensors," in *IPDPS*, pp. 1048–1057, 2017.
[31] O. Kaya and B. Uçar, "Scalable sparse tensor decompositions in distributed memory systems," in *SC*, pp. 1–11, 2015.
[32] L. Karlsson, D. Kressner, and A. Uschmajew, "Parallel algorithms for tensor completion in the cp format," *Parallel Computing*, vol. 57, pp. 222 – 234, 2016.
[33] S. Smith, J. Park, and G. Karypis, "An exploration of optimization algorithms for high performance tensor completion," *SC*, 2016.
[34] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc, "Optimizing sparse tensor times matrix on gpus," *Journal of Parallel and Distributed Computing*, 2018.
[35] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar, "On optimizing distributed tucker decomposition for dense tensors," *CoRR*, vol. abs/1707.05594, 2017.
[36] J. Liu, P. Musialski, P. Wonka, and J. Ye, "Tensor completion for estimating missing values in visual data," *Pattern Anal. Mach. Intell.*, vol. 35, pp. 208–220, 2013.
[37] Y. Liu, F. Shang, W. Fan, J. Cheng, and H. Cheng, "Generalized higher-order orthogonal iteration for tensor decomposition and completion," in *NIPS*, pp. 1763–1771, 2014.
[38] J. Antikainen, J. Havel, R. Josth, A. Herout, P. Zemcik, and M. Hauta-Kasari, "Nonnegative tensor factorization accelerated using gpgpu," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, pp. 1135–1141, July 2011.
[39] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi, "A unified optimization approach for sparse tensor operations on gpus," in *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 47–57, Sept 2017.
[40] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen, "Cubesvd: A novel approach to personalized web search," in *WWW*, pp. 382–390, 2005.
[41] J. Sun, S. Papadimitriou, C.-Y. Lin, N. Cao, S. Liu, and W. Qian, "Multivis: Content-based social network exploration through multi-way visual analysis," in *SDM*, pp. 1064–1075, 2009.
[42] J. Lee, S. Oh, and L. Sael, "Gift: Guided and interpretable factorization for tensors with an application to large-scale multi-platform cancer analysis," *Bioinformatics*, p. bty490, 2018.

Sejoon Oh is a B.S. student in the Department of Computer Science and Engineering of Seoul National University. His research interests include tensor analysis, high-performance computing, and data mining.

Namyong Park is a Ph.D. student in the Computer Science Department of Carnegie Mellon University. He received his B.S. and M.S. in Computer Science and Engineering from Seoul National University. His research interests include data mining and machine learning.

Jun-Gi Jang is a M.S./Ph.D. student in Computer Science and Engineering of Seoul National University. He received his B.S. in Mechanical and Aerospace Engineering from Seoul National University. His research interests include tensor and time series analysis.

Lee Sael is a BK Associate Professor in the Deptartment of Computer Science and Engineering of Seoul National University. She received her Ph.D. in Computer Science from Purdue University in 2010, and her B.S. in Computer Science from Korea University in 2005. She has published in numerous journals and proceedings in the areas of Bioinformatics, Data Mining, and Machine Learning.

U Kang is an associate professor in the Department of Computer Science and Engineering of Seoul National University. He received his Ph.D. in Computer Science at Carnegie Mellon University, and his B.S. in Computer Science and Engineering at Seoul National University. He won 2013 SIGKDD Doctoral Dissertation Award, and three best paper awards including 2018 ICDM 10-year best paper award. His research interests include data mining.