Trillion-Tensor: Trillion-Scale CP Tensor Decomposition

Zeliang Zhang[1] , **Xiao-Yang Liu**[2] , **Pan Zhou**[1]
[1]Huazhong Uniersity of Science and Technology
[2]Department of Electrical Engineering, Columbia University, USA
u201813780@hust.edu.cn, xl2427@columbia.edu, panzhou@hust.edu.cn

Abstract

Due to the storage limitation, tensors with billions or trillions of nonzero elements cannot be loaded into the main memory. Existing tensor decomposition methods only support billion-scale tensors. In this paper, we implement a compression-based algorithm *trillion-tensor* for trillion-scale (number of elements) CP tensor decomposition by trading computation for storage. We make full use of the parallelism of tensor operations to accelerate the proposed trillion-tensor scheme on CPUs and GPUs, respectively. In our experiments, we test tensors ranging from million-scale to trillion-scale and obtain a relatively low mean squared error. Comparing to the baseline method PARACOMP, trillion-tensor supports $8,000$ larger tensors and a speedup up to $6.95\times$.

1 Introduction

Real-world big data are naturally modeled as a multi-dimensional array, called a tensor. Examples include a time-evolving social network, knowledge base and web data. Tensor decomposition is the basis of many machine learning applications including graphic analysis, image classification, data mining, etc. There are two major tensor decomposition models, CP tensor decomposition and Tucker tensor decomposition.

In recent years, the size of tensors becomes increasingly large, approaching millions to trillions of nonzero elements. Due to the limitation of the main memory, both CP and Tucker tensor decomposition algorithms are impractical for large-scale tensors. To address this problem, there have been proposed many adaptive methods for large-scale tensor decomposition, including CDTF [1] and 2PCP [2] for CP tensor decomposition, and Haten2 [3] and GigaTensor [4] for Tucker tensor decomposition. These methods may handle tensors with each mode up to millions. However, the input tensors are required to be extremely sparse, where the total number of nonzero elements is up to millions. On the other hand, the widely used tensor toolboxes Tensor Toolbox [5] and TensorLy [6] operate in the main memory, but cannot support large dense tensors as shown in Table 1. Compression-based algorithm is a powerful technique to handle large scale data with a limited memory in many research areas [7]. The PARACOMP (parallel randomly compressed cubes) algorithm [8] was a promising compression-based tensor decomposition algorithm, but the reported size was $500 \times 500 \times 500$.

Work	Tensor Size
TensorLy [6]	$800 \times 800 \times 800$
Tensor Toolbox [5]	$1,200 \times 1,200 \times 1,200$
PARACOMP [8]	$500 \times 500 \times 500$
Trillion-Tensor (**Ours**)	$10,000 \times 10,000 \times 10,000$

Table 1: Comparison of supported tensor sizes. Our proposed Trillion-Tensor supports $580\times$ larger tensors.

There are three primary challenges with trillion-scale tensor decomposition. 1) Designing a compression method to process an "out-of-memory" tensor; 2) Exploring an appropriate trick to recover the permutations and scaling of the full mode factorization matrices; and 3) implementing efficient parallel schemes both on CPUs and GPUs, respectively.

In this paper, we explore the potential of a compression-based algorithm and implement *trillion-tensor* to support trillion-scale (number of elements) CP tensor decomposition by trading computation for storage. We make full use of the parallelism of tensor operations and design two efficient parallel schemes on CPUs and GPUs, respectively. In our experiments, we test tensors ranging from million-scale to trillion-scale and obtain a relatively low mean squared error. Evaluation results show that *trillion-tensor* supports $8,000$ larger tensors and a speedup up to $6.95\times$, compared with the baseline method PARACOMP [8].

The remainder of this paper is organized as follows. In Section 2, we describe the CP tensor decomposition algorithm. Section 3 describes the proposed *trillion-tensor* algorithm and the parallel schemes for CPUs and GPUs, respectively. In Section 4, we present performance evaluations in terms of reconstruction error and running time. In Section 5, we conclude this paper.

2 Large-scale CP Tensor Decomposition

Notations: we use uppercase calligraphic letters to denote third-order tensors, e.g., $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$, uppercase boldface letters to denote matrices, e.g., $\boldsymbol{A} \in \mathbb{R}^{I\times J}$. $\boldsymbol{X}_{(n)}$ denotes the n-mode matricization of a tensor $\mathcal{X}$. We use $\odot$ to denote Khatri-Rao product.

2.1 CP Tensor Decomposition

The CP tensor decomposition factorizes a tensor into a sum of component rank-one tensors, which can be written in matricized forms $\boldsymbol{X_{(1)}} \approx \boldsymbol{A}(\boldsymbol{C} \odot \boldsymbol{B})^T$, $\boldsymbol{X_{(2)}} \approx \boldsymbol{B}(\boldsymbol{C} \odot \boldsymbol{A})^T$, and $\boldsymbol{X_{(3)}} \approx \boldsymbol{C}(\boldsymbol{B} \odot \boldsymbol{A})^T$. CP tensor decomposition uses the Alternative Least Squares (ALS) method to obtain the mode matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$.

2.2 Compression-based CP Tensor Decomposition

PARACOMP (parallel randomly compressed cubes) [8] is a compression-based algorithm with a potential to support large tensor decomposition. It first uses a set of random matrices to compress the data tensor $\mathcal{X}$ into reduced-size replicas $\{\mathcal{Y}_p\}_{p=1,2,...,P}$, then each replica $\mathcal{Y}_p$ is factored into mode matrices $\{(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p)\}_{p=1,2,...,P}$. After proper permutation and scaling, solving a master linear squares problem per mode will result in the full mode loading matrices $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*)$.

3 The Proposed Trillion-Tensor Scheme

3.1 Overview of Our Trillion-Tensor

The *trillion-tensor* algorithm mainly consists of three stages, namely compression stage, factorization stage, and a scaling and permutation stage, as given in Alg.1 and in Fig.1. First, a tensor $\mathcal{X} \in \mathbb{R}^{300\times 300\times 300}$, consisting of 27 tensor blocks $\mathcal{B}_n \in \mathbb{R}^{100\times 100\times 100}$, $n = 1, 2, ..., 27$, is first compressed into 8 replicas $\mathcal{Y}_p \in \mathbb{R}^{50\times 50\times 50}$, $p = 1, 2, ..., 8$. Second, each replica is factored to mode matrices independently. Thirdly, with a proper permutation and normalization, solving a least squares problem for each mode will result in the full mode matrices $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*)$. Using the factorization of the first block $\mathcal{B}_1$, we get the permutation matrix $\boldsymbol{\Pi}$ and the scaling matrix $\boldsymbol{\Sigma}$. Then, we use $\boldsymbol{\Pi}$ and $\boldsymbol{\Sigma}$ to obtain the full mode matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$.

3.2 Compression Stage

To compress $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ into $\mathcal{Y} \in \mathbb{R}^{L\times M\times N}$, the (i, j, k)-th element of $\mathcal{Y}$ takes the scalar form:

$$\mathcal{Y}_{lmn} = \sum_{i=1}^{I}\sum_{j=1}^{J}\sum_{k=1}^{K} \boldsymbol{U}_{il}\boldsymbol{V}_{jm}\boldsymbol{W}_{kn}\mathcal{X}_{ijk}, \qquad (1)$$

where $\boldsymbol{U} \in \mathbb{R}^{I\times L}, \boldsymbol{V} \in \mathbb{R}^{J\times M}, \boldsymbol{W} \in \mathbb{R}^{K\times N}$ are random matrices from the normal distribution.

In *trillion-tensor*, $\mathcal{X}$ will be compressed into P replicas $\mathcal{Y}_p$, $p = 1, 2, ..., P$, $P \geq \max(\frac{I-2}{L-2}, \frac{J}{M}, \frac{K}{N})$. To process the "out-of-memory" tensor, we adopt a spliting strategy

Algorithm 1 Our Trillion-Tensor Scheme

```
Input: tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ with rank F; dimensions of
    compression replicas, $L, M, N$; the first tensor block
    $\mathcal{B}_1 \in \mathbb{R}^{d_1\times d_2 d_3}$; the number of replicas $P$; the number
    of common columns $S$.
Output: mode matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$;
 1: Randomly generate matrices $\boldsymbol{U}_p \in \mathbb{R}^{I\times L}, \boldsymbol{V}_p \in$
    $\mathbb{R}^{J\times M}, \boldsymbol{W}_p \in \mathbb{R}^{K\times N}$, $p = 1, 2, .., P$, and set the first
    $S$ columns the same, respectively,
 2: $\mathcal{Y}_p \leftarrow$ compress $\mathcal{X}$ using $(\boldsymbol{U}_p, \boldsymbol{V}_p, \boldsymbol{W}_p)$ according to
    (1), for $p = 1, .., P$,
 3: for $p = 1$ to $P$ do
 4:     $(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p) \leftarrow$ rank-F CP decomposition of $\mathcal{Y}_p$,
 5:     $(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p) \leftarrow$ divide each column of
        $(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p)$ by the maximum of the first $S$
        rows, respectively,
 6:     $\boldsymbol{\Pi}_p \leftarrow \arg\max_{\boldsymbol{\Pi}} \mathrm{Tr}(\boldsymbol{A}_1(1:S,:)^T\boldsymbol{A}_p(1:S,:)\boldsymbol{\Pi})$,
 7:     $(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p) \leftarrow (\boldsymbol{A}_p\boldsymbol{\Pi}, \boldsymbol{B}_p\boldsymbol{\Pi}, \boldsymbol{C}_p\boldsymbol{\Pi})$,
 8: end for
 9: $(\boldsymbol{A}^*, \boldsymbol{B}^*, \boldsymbol{C}^*) \leftarrow$ solve the master linear least squares
    problem (2) per mode,
10: $(\boldsymbol{A}', \boldsymbol{B}', \boldsymbol{C}') \leftarrow$ rank-F CP decomposition of $\mathcal{B}_1$,
11: $\boldsymbol{\Pi}, \boldsymbol{\Sigma} \leftarrow$ match $\boldsymbol{A}'$ with the first $d_1$ rows of $\boldsymbol{A}^*$ using
    the Hungarian algorithm.
12: $\boldsymbol{A} \leftarrow \boldsymbol{A}^*\boldsymbol{\Pi}\boldsymbol{\Sigma}$,
13: Repeat line 11-12 for $\boldsymbol{B}$ and $\boldsymbol{C}$, respectively,
14: return $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$.
```

that we first split $\mathcal{X}$ into multiple blocks $\mathcal{B}_n \in \mathbb{R}^{d_1\times d_2\times d_3}$ $n = 1, 2, ..., \frac{IJK}{d_1 d_2 d_3}$ as shown for example in Fig.1. Then, due to the block matrix multiplication, we exploit the related components of $(\boldsymbol{U}_p, \boldsymbol{V}_p, \boldsymbol{W}_p)$ to compress each block $\mathcal{B}_n$ to replicas $\mathcal{Y}_p^n$, $p = 1, 2, ...P$. Finally, accumulate them to the final replicas $\mathcal{Y}_p$ respectively, i.e. $\mathcal{Y}_p += \mathcal{Y}_p^n$, $p = 1, 2, .., P$.

3.3 Factorization Stage

On the factorization stage, each replica $\mathcal{Y}_p$ is factored into $(\boldsymbol{A}_p, \boldsymbol{B}_p, \boldsymbol{C}_p)$, $p = 1, 2, .., P$. Due to a property of the Kronecker product [9], we have $\boldsymbol{A}_p = \boldsymbol{U}_p^T\boldsymbol{A}\boldsymbol{\Pi}_p\boldsymbol{\Sigma}_p$, $p = 1, 2, ..., P$. Here, we apply Hungarian algorithm [10] to the trace maximization problem to get rid of $\boldsymbol{\Pi}_p$ and divide each column by its maximum of the first S rows to get rid of $\boldsymbol{\Sigma}_p$, where S is the number of shared columns in the random matrices $\boldsymbol{U}_p, p = 1, 2, ..., P$. Finally, solve a master least squares problem

$$\begin{bmatrix} \boldsymbol{A}_1 \\ \vdots \\ \boldsymbol{A}_P \end{bmatrix} = \begin{bmatrix} \boldsymbol{U}_1^T \\ \vdots \\ \boldsymbol{U}_P^T \end{bmatrix} \boldsymbol{A}\boldsymbol{\Pi}\boldsymbol{\Sigma} \qquad (2)$$

to obtain $\boldsymbol{A}^* = \boldsymbol{A}\boldsymbol{\Pi}\boldsymbol{\Sigma}$, and similarly for $\boldsymbol{B}^*$ and $\boldsymbol{C}^*$.

3.4 Scaling and Permutations

To recover the scaling and permutations, we use the factors $(\boldsymbol{A}', \boldsymbol{B}', \boldsymbol{C}')$ of the first block $\mathcal{B}_1$. Because the CP tensor decomposition is unique, we assume that

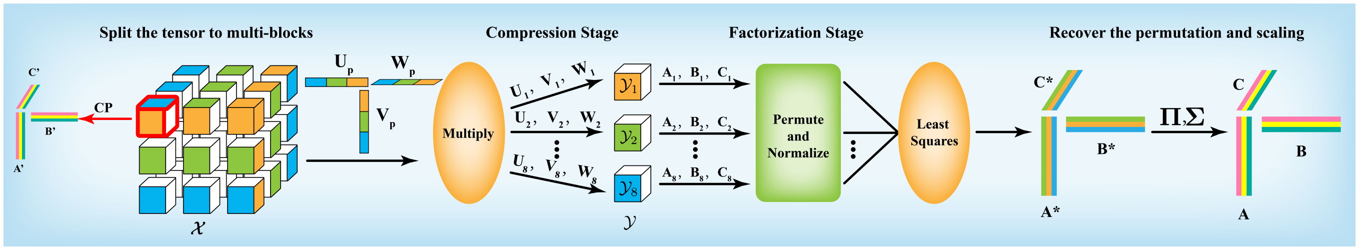

Figure 1: Overview of the proposed trillion-tensor decomposition, where the tensor $\mathcal{X} \in \mathbb{R}^{300\times300\times300}$, the amount P of the compression replicas $\mathcal{Y}$ is $8 \geq \max(\frac{300-2}{50-2}, \frac{300}{50}, \frac{300}{50})$.

$(\boldsymbol{A}^{'}, \boldsymbol{B}^{'}, \boldsymbol{C}^{'})$ are the first d_1 rows of full mode matrices $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$, i.e., $\boldsymbol{A}^{'} = \boldsymbol{A}^*(1 : d_1, :)\boldsymbol{\Pi\Sigma}$. Here, we make the diagonal entries of the scaling matrix $\boldsymbol{\Sigma}$ composed of the maximums of each column in $\boldsymbol{A}^{'}$, i.e. $\boldsymbol{\Sigma} = \text{diag}(\max(\boldsymbol{A'}(:,1)), ..., \max(\boldsymbol{A'}(:,F)))$. To obtain the full mode matrix $\boldsymbol{A}$, first, we divide each column of $\boldsymbol{A'}$ by its maximum and divide each column of $\boldsymbol{A}^*$ by its maximum of the first d_1 rows. Second, similar to the factorization stage, we apply the Hungarian algorithm [10] to maximize $\text{Tr}(\boldsymbol{A}^{'T}\boldsymbol{A}^*(1 : d_1, :)\boldsymbol{\Pi})$ and obtain the permutation matrix $\boldsymbol{\Pi}$. Thus, we get the full mode matrix $\boldsymbol{A} = \boldsymbol{A}^*\boldsymbol{\Pi\Sigma}$, similarly for $\boldsymbol{B}$ and $\boldsymbol{C}$.

In practice, we usually make the first block on this stage, $\mathcal{B}_1 \in \mathbb{R}^{d_1\times d_2\times d_3}$, to be different from that on the compression stage in the size. And in our experiment of the large-scale tensor decomposition, it is proved to be effective if we set $d_1 \neq d_2 \neq d_3$ and make the size of the tensor block on current stage smaller than that on the compression stage when the rank F of tensor $\mathcal{X}$ is relative smaller than the size of tensor.

3.5 Optimizations

The computational complexity of the compression stage is $O(PIJK \times \min(L, M, N))$ versus $O(\frac{IJ}{F})$ of the factorization stage [8]. Huge time consumption of the compression stage is the bottleneck of this algorithm with the size of tensor increasing.

On the compression stage, operations of all tensor blocks $\mathcal{B}_n, n = 1, 2, ..., \frac{IJK}{d_1d_2d_3}$, are independent of each other, and the respective compression result $\mathcal{Y}_p^n$ will be accumulated to the common replica $\mathcal{Y}_p$, $p = 1, 2, ..., P$. There is $\frac{IJK}{d_1d_2d_3}$ parallelism in this process. Thus, we design two parallelization schemes for efficient compression on CPUs and GPUs, respectively.

Parallel implementation on CPUs: This scheme advocates exploiting the property of multi-core processors on CPUs. In this scheme, we compress the tensor block $\mathcal{B}_n$, $n = 1, 2, ..., \frac{IJK}{d_1d_2d_3}$, in parallel on multi processes. On each sub-process, one of tensor blocks, $\mathcal{B}_n$, will be loaded in memory and independently compressed into sub-replicas $\mathcal{Y}_p^n, p = 1, 2, ..., P$, then the sub-replicas will be accumulated to corresponding common replicas $\mathcal{Y}_p$ shared on the main process, $p = 1, 2, ..., P$.

Parallel implementation on GPUs: This scheme advocates exploiting GPUs' high performance in matrix multiplication. Consider a given situation that for a trillion-scale tensor $\mathcal{X} \in \mathbb{R}^{10,000\times10,000\times10,000}$ composed of tensor blocks $\mathcal{B}_n \in \mathbb{R}^{500\times500\times500}$, $n = 1, 2, ..., 8000$, there are $3 \times 8,000$ times of matrix multiplication need to be done. The huge time consumption of extra large-scale matrix multiplication on CPUs is always unbearable. However, compared to CPU, GPU is more suitable for large-scale matrix multiplication for its much greater number of relatively unexceptional processing cores.

Thus, compared to the parallel implementation on CPUs, this scheme differs only in the matrix computation step, in which it transfer the block $\mathcal{B}_n$ of each sub-process from CPUs to GPUs and the following compression steps except the accumulation will work on GPUs. The accumulation step will first transfer $\mathcal{Y}_p^n$, $p = 1, 2, ..., P$ to CPUs and then accumulate them to corresponding replicas.

4 Performance Evaluations

In this section, we will provide numerical experiment results to show the performance of our work on various scales of tensors from one hundred million to one trillion.

We make our experiments on a server which has two Intel(R) Xeon(R) Gold 5118 CPUs. Each of CPUs has 12 cores @2.30GHz supporting 24 hardware threads. There is a Tesla V100 GPU which consists of 16 GB device memory. There are 128 GB DDR4 memories on the server. We show the mean squared error (MSE) to measure the accuracy and speedups of baseline versus optimization schemes. The speedups is defined as $(\textit{baseline running time})/(\textit{optimized running time})$.

For each case on scales ranging from one hundred million to one trillion, we first generate the mode matrices $A \in \mathbb{R}^{I\times F}$,$B \in \mathbb{R}^{J\times F}$, $C \in \mathbb{R}^{K\times F}$ from an independent normal distribution to generate the huge tensor $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ where we set $I = J = K$ ranging from 1000 to 10000 and the rank F is set to be a constant small value 5. On the compression stage, we set the size of compressed tensor cubes $\mathcal{Y}_p \in \mathbb{R}^{L\times M\times N} = 50$ as same value $L = M = N$. The size of one tensor block $B \in \mathbb{R}^{d_1\times d_2\times d_3}$ on the compression stage is set as $d_1 = d_2 = d_3 = 500$ while it is set as $d_1 = 50, d_2 = 100$ and $d_3 = 150$ on the scaling and permutation stage. We set P as $\max(\frac{I-2}{L-2}, \frac{J}{M}, \frac{K}{N})$+10 to avoid the situation that

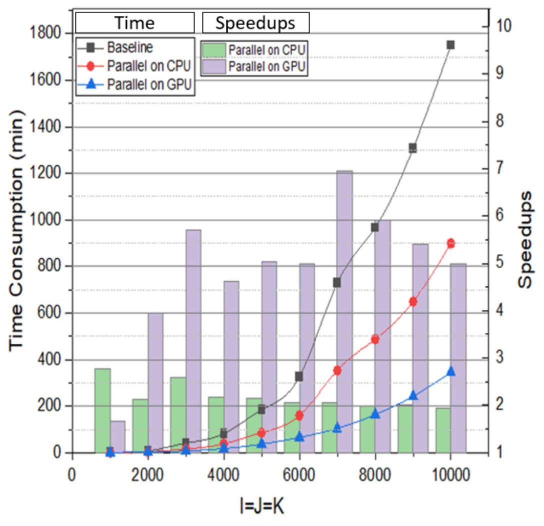

Figure 2: Comparison between the baseline and optimized implementation.

if the CP tensor decomposition of one or a little more compression replicas can't converge on the factorization stage, drop it (them) in time.

Figure 2 shows the comparison of time performance between the baseline and the optimized versions. Compared to the baseline, the optimized version on CPUs achieves an average of 2.18× speedups with up to 2.77× speedups, and the optimized version on GPUs achieves an average of 4.92× speedups with up to 6.95× speedups. For trillion-scale tensor of size $10,000 \times 10,000 \times 10,000$, we reduce the baseline time of nearly half a day to no more than 6 hours.

Figure 3 shows the comparison of the mean squared error (MSE) between the baseline and the optimized versions. It can be seen that the all the MSE are increasing as the scale increases, but all of them control the error under the magnitude of 10^{-7}. Comparing to the baseline, the parallel scheme on CPUs is almost the same with more than 2× speedups. For the parallel scheme on GPUs, it trades just a little more loss for the significant acceleration.

5 Conclusions

In this paper, we have proposed a trillion-tensor scheme for CP tensor decomposition that can support trillion-scale tensors. We provide acceleration techniques for CPU and GPU implementations, respectively. We tested various tensor sizes ranging from millions to trillions and obtain a low reconstruction error. Comparing with the baseline method PARACOMP [8], the proposed trillion-tensor scheme achieves a maximum of 6.95× speedups over the baseline implementation.

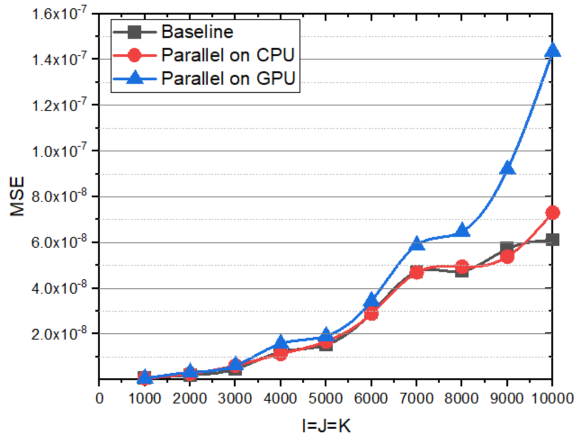

Figure 3: Comparison of reconstruction error between the baseline and optimized implementations.

References

[1] Sael L. Shin, K. and U. Kang, "Fully scalable methods for distributed tensor factorization," *IEEE Transactions on Knowledge and Data Engineering*, pp. 100–113, 2016.

[2] Shengyu Huang K. Selçuk Candan Li, Xinsheng and Maria Luisa Sapino, "2PCP: Two-phase cp decomposition for billion-scale dense tensors," *IEEE 32nd International Conference on Data Engineering (ICDE)*, pp. 835–846, 2016.

[3] Papalexakis E.E. Kang U. Jeon, I. and C. Faloutsos, "Haten2: Billion-scale tensor decompositions," *IEEE 31st International Conference on Data Engineering*, pp. 1047–1058, 2015.

[4] Abhay Harpale Christos Faloutsos U Kang, Evangelos Papalexakis, "GigaTensor: Scaling tensor analysis up by 100 times algorithms and discoveries," *kdd*, 2014.

[5] Brett W. Bader, Tamara G. Kolda, et al., "Matlab tensor toolbox version 3.1," June 2019.

[6] Anima Anandkumar Maja Pantic Jean Kossaifi, Yannis Panagakis, "TensorLy: Tensor learning in python," *Journal of Machine Learning Research*, 2019.

[7] Xiao-Yang Liu, Yanmin Zhu, Linghe Kong, Cong Liu, Yu Gu, Athanasios V Vasilakos, and Min-You Wu, "Cdc: Compressive data collection for wireless sensor networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 8, pp. 2188–2197, 2014.

[8] Evangelos E. Papalexakis Nicholas D. Sidiropoulos and Christos Faloutsos, "Parallel randomly compressed cubes," *IEEE Signal Processing Magazine*, pp. 57–70, 2014.

[9] J. Brewer, "Kronecker products and matrix calculus in system theory," *IEEE Trans. Circuits Syst.*, vol. 19, pp. 772–781, 1978.

[10] Harold W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, pp. 83–97, 1955.