CCF Transactions on High Performance Computing (2019) 1:161–176
https://doi.org/10.1007/s42514-019-00017-5

REGULAR PAPER

swTensor: accelerating tensor decomposition on Sunway architecture

Xiaogang Zhong[1,2] · Hailong Yang[1,2] · Zhongzhi Luan[1] · Lin Gan[3] · Guangwen Yang[3] · Depei Qian[1]

Received: 9 May 2019 / Accepted: 6 November 2019 / Published online: 20 November 2019

Abstract
Modern applications are digesting and generating data with rich features that are stored in high dimensional array or tensor. The computation applied to tensor, such as Canonical Polyadic decomposition (CP decomposition) plays an important role in understanding the internal relationships within the data. Using CP decomposition to analyze large tensor with billions of sizes requires tremendous computation power. In the meanwhile, the emerging Sunway many-core processor has demonstrated its computation advantage in powering the first hundred petaFLOPS supercomputer in the world. In this paper, we propose *swTensor* that adapts the CP decomposition to Sunway processor by leveraging the MapReduce framework for automatic parallelization and the unique architecture of Sunway for high performance. Specifically, we divide the major computation of CP decomposition into four sub-procedures and implement each using MapReduce framework with customized design key-value pair. Also, we tile the data during the computation so that it fits into the limited local device memory on Sunway for better performance. Moreover, we propose a performance auto-tuning mechanism to search for the optimal parameter settings in *swTensor*. The experimental results demonstrate *swTensor* achieves better performance than the state-of-the-art *BigTensor* and *CSTF* with the average speedup of 1.36 × and 1.24 ×, respectively. Besides, *swTensor* exhibits better scalability when scaling across multiple Sunway processors.

Keywords Sunway architecture · MapReduce · Tensor decomposition

1 Introduction

The evolution of recommendation system improves user satisfactory with Internet surfing by offering the assistance for searching the desired information. Recommendation system uses tensor to store and compute feature information in order to provide timely response (Sidiropoulos et al. 2017; Nickel et al. 2012). Moreover, tensor also plays an increasingly important role in the fields of computer vision (Shashua and Hazan 2005; Aja-Fernández et al. 2009), image recognition (Lei and Yang 2006; Sonka et al. 2014) and signal processing (Lim and Comon 2010; Cichocki et al. 2015). Using tensor to represent and store the feature information improves the efficiency of application programming and execution. Tensor decomposition is an indispensable method for tensor computation (Tew 2016; Kolda and Bader 2009; Acar et al. 2011; Golub and Van Loan 2012). Tucker decomposition and Canonical Polyadic decomposition (CP decomposition) are two widely used schemes for tensor decomposition. Existing work (Kolda and Bader 2009; Acar et al. 2011; Choi et al. 2018; Kang et al. 2012) perform comprehensive bottleneck analysis and propose solutions from

✉ Hailong Yang
hailong.yang@buaa.edu.cn

Xiaogang Zhong
xiaogang2017@buaa.edu.cn

Zhongzhi Luan
07680@buaa.edu.cn

Lin Gan
lingan@tsinghua.edu.cn

Guangwen Yang
ygw@tsinghua.edu.cn

Depei Qian
depeiq@buaa.edu.cn

[1] Sino-German Joint Software Institute, the School of Computer Science and Engineering, Beihang University, Beijing 100191, China

[2] State Key Laboratory of Software Development Environment, Beihang University, Beijing 100191, China

[3] Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China

[4] Present Address: Beihang University, G823, New Main Building, Beijing 100191, China

different perspectives to improve the performance of tensor decomposition.

In the meanwhile, MapReduce (Dean and Ghemawat 2008) framework from the big data community exhibits its promising merits such as easy to program, automatic parallelism and high scalability. It relieves the burden of programmers from understanding the underlying hardware details when developing large scale parallel application. Using MapReduce, the computation procedure of a big data application is abstracted into two processing stages, *map* and *reduce*, which are automatically parallelized across multiple machines. Due to the above advantages of MapReduce, researchers (Jeon et al. 2016; Tsourakakis 2010; Kang et al. 2012) have been attempting to parallelize the computation of tensor decomposition using MapReduce framework. The combination of tensor decomposition and MapReduce framework improves the efficiency of tensor computation by leveraging massive computing resources.

Moreover, the optimizations to tensor decomposition have been adapted to various hardware architectures and programming frameworks. For instance, Smith et al. (2017) achieve a good speedup of CP decomposition on Intel KNL (Knights Landing) many-core processor. Choi et al. (2018) pinpoint the performance bottleneck of MTTKRP (metricized tensor times Khatri-Rao product) and apply data blocking technique to boost the performance on IBM POWER8 processor. Kang et al. (2012) optimize CP decomposition using MapReduce framework and avoid intermediate data explosion when updating factor matrices iteratively. Specifically, *BigTensor* (Park et al. 2016) and *CSTF* (Blanco et al. 2018) are the most popular open source implementations of tensor decomposition using MapReduce framework. Despite the above research efforts, adapting to the emerging architecture, meanwhile leveraging the parallel processing of MapReduce framework to implement efficient tensor decomposition, is still a challenging research area.

Sunway Taihuligh is the first high-performance supercomputer that exceeds 100 PFLOPS in double precision. Sunway Taihulight is powered by SW26010 processor, which contains four core groups (CGs). Each CG comprises of one Management Processing Element (MPE) and 64 Computing Processing Elements (CPEs). The MPE can fully support interrupt processing, memory management and out-of-order execution. On the contrary, limited functions are supported on CPE. However, the 64 CPEs provide high aggregated computing power. Each CPE has a 64 KB Local Device Memory (LDM) that is manually controlled by programmers. DMA is supported on CPE to achieve high memory bandwidth with accesses in batch. Moreover, CPEs can communicate with each other through register communication. The peak floating point performance of one SW26010 is 3 TFLOPS. Unfortunately, there is no implementation of tensor decomposition available on Sunway processor that, on the one hand takes advantage of MapReduce parallel processing, and on the other hand adapts to the architectural features of Sunway. The missing support hinders applications relying on efficient tensor decomposition to exploit the computation power of Sunway Taihulight.

This work primarily focuses on how to implement efficient sparse CP decomposition with dense factor matrices on Sunway architecture. Leveraging the available MapReduce framework (*swMR*) on Sunway (Zhong et al. 2018), we propose *swTensor* to realize tensor decomposition on Sunway architecture. In *swMR*, the CPEs within a CG are organized into 32 CPE pairs. Within each CPE pair, map and reduce role is assigned to each CPE, respectively. *swTensor* tiles the tensor data into 32 groups to adapt to the design of *swMR* with customized design for key-value pair (Sect. 3.2). Moreover, in *swTensor* the computation of Alternating Least Squares (ALS) algorithm is divided into four sub-procedures in order to avoid the intermediate data explosion (Sect. 3.3). Furthermore, pipeline processing is also applied to deal with the intermediate data, which better utilizes the limited LDM on each CPE.

Specifically, this paper makes the following contributions:

- We propose *swTensor*, an efficient implementation of sparse CP decomposition with dense factor matrices using MapReduce framework on Sunway. The *swTensor* exploits the benefits from both the MapReduce framework for automatic parallel processing and the many-core architecture of Sunway for high performance.
- We propose a data tiling method to adapt to the design of MapReduce framework on Sunway. In addition, we divide the computation of ALS algorithm into four sub-procedures in order to avoid the intermediate data explosion during decomposition.
- We expand *swTensor* to run on large scale by using MPI to coordinate the computation among CGs. With more Sunway processors utilized during decomposition, *swTensor* can support CP decomposition on larger tensor data.
- We identify several performance impacting parameters in *swTensor* and build a performance model for auto-tuning the parameters for optimal performance. This performance auto-tuning method eliminates the manual effort of tuning by hand and the time cost of exhaustive search.
- We evaluate the performance of *swTensor* using datasets at different scales. The experimental results demonstrate that *swTensor* achieves 1.36 × and 1.24 × better performance than *BigTensor* and *CSTF* on average, respectively. In addition, the scalability of *swTensor* is better than *BigTensor* and *CSTF* when scaling beyond a single Sunway processor.

The rest of this paper is organized as follows. Section 2 illustrates the Sunway architecture and the MapReduce framework available on Sunway. In addition, the background on tensor computation is described. We also illustrate the challenges to implement efficient tensor decomposition on Sunway. Section 3 presents the methodology and implementation of *swTensor*. Section 4 presents the performance auto-tuning method to determine the parameter settings for *swTensor*. Section 5 presents the evaluation results of *swTensor* across different scales of tensor data. We also compare the performance of *swTensor* to the-state-of-the-art tensor computation frameworks *BigTensor* and *CSTF* on CPU. Section 6 presents the related works in this field. Section 7 concludes our work.

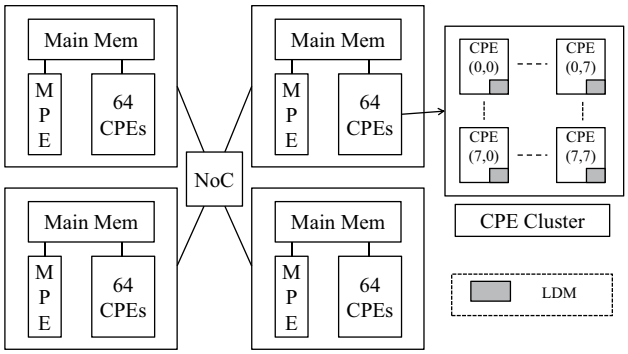


Fig. 1 The architecture of Sunway SW26010 processor. Each processor contains four core groups. Each core group has one MPE and 64 CPEs that are organized as 8×8 mesh. Each CPE has a 64KB LDM

2 Background

2.1 Sunway many-core processor

Sunway Taihulight supercomputer is powered by Sunway SW26010 processor, which contains four core groups (CGs) and each core group has one MPE (Management Processing Element) and 64 CPEs (Computing Processing Elements) that are organized as 8×8 mesh, as shown in Fig. 1. The MPE has 32 KB L1 data cache and 256 KB L2 instruction and data cache, whereas each CPE has 16 KB L1 instruction cache and 64 KB LDM (Local Device Memory). The peak memory bandwidth and double-precision performance of each CG is 34 GB/s and 756 GFLOPS, respectively (Wang et al. 2018). There are two approaches supported for CPE to access main memory, one is the gld/gst (global load/store) for discrete access and the other one is the DMA for batch access. The DMA channel provides much higher bandwidth than discrete access. In addition, CPE can communicate with each other when in the same row or column of the mesh through register communication.

2.2 MapReduce framework on Sunway

The *swMR* is the most recently proposed MapReduce framework that adapts to the architectural features of Sunway processor (Zhong et al. 2018). In *swMR*, the 64 CPEs in a CG are divided into 32 CPE pairs, and the map/reduce role is assigned to the two CPEs within each CPE pair, respectively. As shown in Fig. 2, when the *map* CPE finishes processing the input data, it generates the intermediate data and sends it to the *reduce* CPE within the same CPE pair through register communication. After the *reduce* CPE finishes processing, it stores the partial results back to main memory. In the meanwhile, the MPE continuously checks whether all CPE pairs are completed. It then combines the partial results from all CPE pairs and generates the final results.

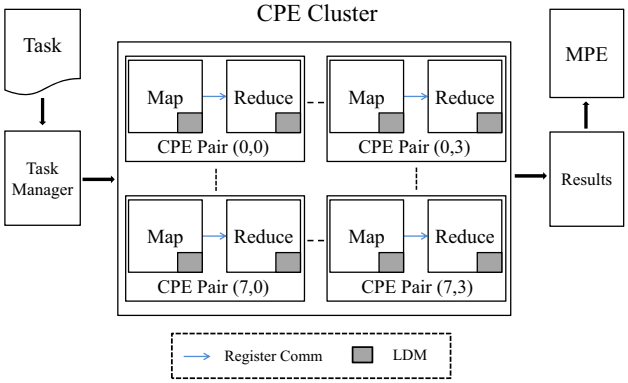


Fig. 2 The *swMR* framework. The map and reduce role is assigned to the CPEs within each CPE pair. The MPE collects the partial results from each CPE pair and combine them into final results

Note that, each CPE pair can process its own input data independently. *swMR* utilizes the LDM on each CPE to cache the intermediate data during the processing in order to avoid accessing main memory frequently. Moreover, *swMR* takes the advantage of double buffering to prefetch the input data from the main memory so that it can overlap the memory access latency with computation. Another unique design of *swMR* is that it can change the processing role (map to reduce or vice versa) within each CPE pair dynamically, which achieves better load balance during runtime.

2.3 Tensor decomposition

2.3.1 Tensor definition and operation

Tensor is the widely used data structure for representing data in high dimension. In general, scalar, vector and matrix can be considered as zero-order tensor, first-order tensor and second-order tensor, respectively. Each dimension is termed as a mode. A tensor with more than three modes is called

high-order tensor (Kolda and Bader 2009). Since it is hard to describe when the tensor has more than three modes, in the following discussion we focus on three-mode tensor without losing the generality. Each element of a three-mode tensor can be located by the indexes i, j and k (Golub and Van Loan 2012). For instance, we can refer to a particular element within a three-mode tensor as $x(i, j, k)$. Accessing elements of tensors with other modes is similar. Using indexes to locate elements within a tensor is efficient especially when the tensor is sparse. There are a large number of mathematical operations that can be applied to tensor. However, due to the page limit, we briefly introduce the important mathematical operations regarding CP decomposition. Readers can refer to Kolda and Bader (2009) for more detailed discussion. In addition, for matrix related mathematical operations, the reader can refer to Golub and Van Loan (2012).

Given matrices $A \in \mathbb{R}^{M\times N}$ and $B \in \mathbb{R}^{P\times Q}$, the Kronecker product between A and B is defined in Eq. (1).

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix} \tag{1}$$

The Khatri-Rao product is column-wise Kronecker product between matrices. For instance, there are two matrices, matrix $A \in \mathbb{R}^{I\times K} = (\mathbf{a_1}, \mathbf{a_2}, \mathbf{a_3}, \ldots, \mathbf{a_k})$ and matrix $B \in \mathbb{R}^{J\times K} = (\mathbf{b_1}, \mathbf{b_2}, \mathbf{b_3}, \ldots, \mathbf{b_k})$. Then the Khatri-Rao product between A and B is defined in Eq. (2).

$$A \odot B = (\mathbf{a_1} \otimes \mathbf{b_1}, \mathbf{a_2} \otimes \mathbf{b_2}, \mathbf{a_3} \otimes \mathbf{b_3}, \ldots, \mathbf{a_k} \otimes \mathbf{b_k}) \tag{2}$$

The Hadamard product is conducted between two matrices with the same size, e.g. matrix $A \in \mathbb{R}^{I\times J}$ and $B \in \mathbb{R}^{I\times J}$. Then the Hadamard product between A and B is defined in Eq. (3).

$$A * B = \begin{bmatrix} a_{11}b_{11} & a_{12}b_{12} & \cdots & a_{1J}b_{1J} \\ a_{21}b_{21} & a_{22}b_{22} & \cdots & a_{2J}b_{2J} \\ \vdots & \vdots & \ddots & \vdots \\ a_{I1}b_{I1} & a_{I2}b_{I2} & \cdots & a_{IJ}b_{IJ} \end{bmatrix} \tag{3}$$

In practice, high-mode tensor is more obscure than vector and matrix, therefore it is difficult to represent and compute high-mode tensor. Hence, an approach to express high-mode tensor with two dimensional matrix is proposed, which is called tensor unfolding or tensor matricization. In general, for a three-mode tensor, there are three ways to unfold it, mode 1, mode 2 and mode 3. For instance, given a tensor $X \in \mathbb{R}^{I\times J\times K}$, mode 1 unfolding generates $X_{(1)} = [X(:,:,1), X(:,:,2)] \in \mathbb{R}^{I\times JK}$, mode 2 unfolding generates $X_{(2)} = [X(:,:,1)^T, X(:,:,2)^T] \in \mathbb{R}^{J\times IK}$, and mode 3 unfolding generates $X_{(3)} = [X(:,1,:)^T, X(:,2,:)^T, X(:,3,:)^T] \in \mathbb{R}^{K\times IJ}$.

2.3.2 Canonical polyadic decomposition

The most widely used tensor decomposition is Canonical Polyadic decomposition (CP decomposition) (Hitchcock 1927) and Tucker Decomposition (Tucker 1963). In this paper, we focus on implememting CP decomposition on Sunway architecture. CP decomposition was initially proposed by Hitchcock in 1927 (Hitchcock 1927). We briefly introduce the mathematical definition of CP decomposition. For a detailed mathematical proof, reader can refer to Kolda and Bader (2009). Given a three-mode tensor $X \in \mathbb{R}^{I\times J\times K}$, the CP decomposition of X is defined Eq. (4), where $A \in \mathbb{R}^{I\times R}$, $B \in \mathbb{R}^{J\times R}$ and $C \in \mathbb{R}^{K\times R}$ are factor matrices. The value of element (i, j, k) located in tensor X can be approximated as $x_{ijk} \approx \sum_{r=1}^{R} a_{ir}b_{jr}c_{kr}$. The computation process of CP decomposition is illustrated in Fig. 3.

$$X \approx \sum_{r=1}^{R} \lambda_r A(:,r) \otimes B(:,r) \otimes C(:,r) \tag{4}$$

As shown in Eq. (4), the difficult part of CP decomposition is to determine the three factor matrices, A, B and C. To address that, Alternating Least Squares (ALS) algorithm is proposed to compute factor matrices. ALS iteratively updates factor matrices until the error is tolerable or maximum iteration is reached. Algorithm 1 shows the code of applying ALS to perform CP decomposition. To compute factor matrix A, we fix matrices B and C, then apply ALS. Equation 5 shows the computation for factor matrix A. A similar approach is applied to compute matrices B and C, respectively. Note that † is the pseudo-inverse operation.

$$\hat{A} = X_{(1)}(C \odot B)(C^T C * B^T B)^{\dagger} \tag{5}$$

Algorithm 1 CP-ALS (Alternating Least Squares)

```
1: Input: Tensor X and maximum number of iterations Count.
2: function CP-ALS(X, Count)
3:     while round not equal Count or error not tolerable do
4:         Â = X_(1)(C ⊙ B)(C^T C ∗ B^T B)^†
5:         B̂ = X_(2)(C ⊙ A)(C^T C ∗ A^T A)^†
6:         Ĉ = X_(3)(B ⊙ A)(B^T B ∗ A^T A)^†
7:     end while
8: end function
```

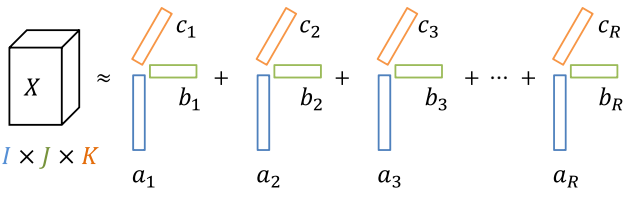


Fig. 3 The illustration of CP decomposition. $X \in \mathbb{R}^{I\times J\times K}$, $A \in \mathbb{R}^{J\times R}$, $C \in \mathbb{R}^{K\times R}$, R is the rank of tensor X

2.3.3 Challenges of implementing CP decomposition on Sunway

In order to provide an efficient implementation of sparse CP decomposition with dense factor matrices on Sunway, we need to adapt the computation characteristics of CP decomposition not only to the architectural features of Sunway for high performance, but also to the MapReduce framework for automatic parallelization. To achieve that, there are several unique challenges we need to address.

- How to express the sparse CP decomposition with dense factor matrices in MapReduce algorithm so that we can leverage the existing *swMR* framework for automatic parallelization within a CG on Sunway processor. In *swMR*, the CPEs within a CG are divided into 32 CPE pairs, where the map and reduce tasks are performed within each CPE pair. We need to adapt the computation of CP decomposition to the unique design of CPE pairs in *swMR*.
- How to utilize the limited LDM on CPEs to improve the computation efficiency of sparse CP decomposition. Accessing the main memory on Sunway is orders of magnitude slower than LDM. However, the computation of Khatri-Rao product generates massive intermediate data with large tensor. Hence, we need to optimize the computation procedure in order to avoid the intermediate data explosion so that it can fit into the limited LDM on each CPE.
- How to scale the sparse CP decomposition across multiple CGs so that large tensor data can be processed on Sunway. The current *swMR* framework does not support running beyond a single CG. We need to extend our implementation of sparse CP decomposition *swTensor* using MPI, so that computation can be distributed and coordinated on multiple CGs for higher performance.
- How to determine the parameter settings for *swTensor* so that it can achieve better performance when implemented on Sunway processor. Since there are several parameters that could affect the performance of *swTensor*, exhaustively searching is both time consuming and unsustainable. Therefore, we need a performance auto-tuning mechanism to search for the optimal parameter settings for *swTensor*.

3 *swTensor*: methodology and implementation

In this section, we first introduce the design overview of our CP decomposition on Sunway. Then, we describe the implementation details, including the optimizations we use to adapt the CP decomposition to Sunway and avoid the intermediate data explosion.

3.1 Design overview

In order to support CP decomposition on Sunway, we propose *swTensor* that adapts the computation of CP decomposition to the MapReduce framework on Sunway. We notice that the major computation of CP decomposition is to update the factor matrices iteratively using ALS algorithm shown in Algorithm 1. Since the computation of ALS is similar to update factor matrices A, B and C, we take matrix A for illustration as shown in Eq. (5). Note that the Khatri-Rao product is performed in column order, therefore we transpose the factor matrices before the computation. However, for the ease of illustration, we use the original form of factor matrice in our discussion. To further fit in the multi-stage processing on MapReduce framework, we divide the computation procedure of Eq. (5) into four sub-procedures as following:

- Sub-procedure 1: $M = (C \odot B)$.
- Sub-procedure 2: $N = X_{(1)}M$.
- Sub-procedure 3: $Q = (C^TC * B^TB)^{\dagger}$.
- Sub-procedure 4: $\hat{A} = NQ$.

In general, the initial value of the factor matrices A, B and C are randomly generated (Kolda and Bader 2009). The size of tensor X is usually very large (e.g., $10K \times 10K \times 10K$), therefore using the traditional method to directly solve Eq. (5) will generate tremendous intermediate data from Khatri-Rao product. To address the intermediate data explosion problem, we apply data tiling technique (Sect. 3.2) along with the MapReduce processing (Sect. 3.3) on Sunway.

Figure 4 illustrates the computation workflow for calculating factor matrix A with the above four sub-procedures using the MapReduce framework on Sunway. In Sub-procedure 1, the *map* CPEs are responsible for conducting the Khatri-Rao product between factor matrices B and C, which equals to the Kronecker product by columns, to derive M. In Sub-procedure 2, the *map* CPEs perform matrix multiplication between $X_{(1)}$ and M, and then *reduce* CPEs add the JK values to derive N. In Sub-procedure 3, the *map* CPEs perform matrix multiplication for both C^TC and B^TB, and then the *reduce* CPEs perform add operation for both J and K. After that, the *map* CPEs perform Hadamard product between J and K to derive Q. In Sub-procedure 4, the *map* CPEs perform matrix multiplication between N and Q, and then the *reduce* CPEs add the R values to derive A.

Note that in addition to the above four Sub-procedures, the rest matrix operations in CPD include: (1) tensor flatten of X, (2) matrix normalization of A, B and C, and (3) computation of factor matrices B and C. For (1), it is not the performance hotspot of CPD, and thus out of the scope of this

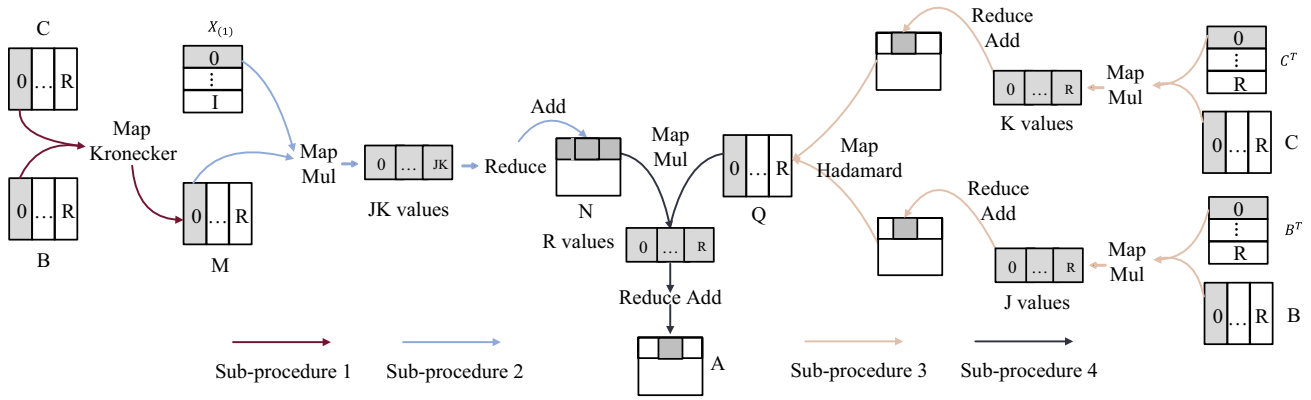


Fig. 4 The computation workflow of *swTensor* for calculating factor matrix A

paper. For (2), we have already included matrix normalization in each Sub-procedure. Since the matrix normalization is never the performance bottleneck of CPD, we execute the matrix normalization on MPE. For (3), since the computation of factor matrices B and C is similar to matrix A, we omit the computation details of matrices B and C for brevity.

3.2 Data tiling

For Sub-procedure 1, $M = (C \odot B) = (\mathbf{c_1} \otimes \mathbf{b_1}, \mathbf{c_2} \otimes \mathbf{b_2}, \ldots, \mathbf{c_k} \otimes \mathbf{b_k})$, the Khatri-Rao product eventually transforms into Kronecker product. For Kronecker product, $\mathbf{c_i} \otimes \mathbf{b_i}$ represents that each element of vector $\mathbf{c_i}$ multiplies all elements of vector $\mathbf{b_i}$. Since the computation on matrices B and C is column-wise, we only need to tile matrix C by column. During each iteration, when the *map* CPE receives its tile and interprets the column index of matrix C, it can automatically locate the corresponding column index of matrix B.

As shown in Fig. 5, we tile the data by columns according to the number of CPE pairs in *swMR*. According to Kronecker product, each CPE pair retrieves its input data by the assigned column index, that is one element c_{ij} from matrix C at a time. The *map* CPE then retrieves the elements $b_{n_1 j \sim n_2 j}$ from the corresponding column of B (denoted as B_{jb}, which are the elements in the *bth* block and the *jth* column of B) based on the row and column indexes contained in the key-value pairs. We use LDM to store the elements from B as many as possible. Then, c_{ij} is multiplied by B_{jb} to derive elements $m_{n_1 j \sim n_2 j}$ in matrix M (denoted as M_{jb}, which are the elements in the *bth* block and the *jth* column of M). For each CPE pair, it stores M_{jb} in LDM each time for the computation in subsequent procedures, which consumes $B_{jb} \times 8$ bytes (generated by the multiplication of B_{jb} and c_{ij}). In our experiments, the M_{jb} consumes 8.4KB on average, which takes up 6.55% of the LDM for each CPE pair.

Then, it proceeds to the Sub-procedure 2, as shown in Fig. 6. We tile $X_{(1)} \in I \times JK$ that is generated from mode 1 unfolding. Multiplying M_{jb} obtained from Sub-procedure 1 and the corresponding vector from $X_{(1)}$ generates the elements for each row of $X_{(1)}$. For instance, $X_{(1)_{ix_1 \sim ix_2}}$ denoted as $X_{(1)ib}$ represents elements from row one and block b in $X_{(1)}$. Note that $x_2 - x_1 = n_2 - n_1$ means that the number of elements within these two blocks equals. After that, we compute $\mathbb{N}_{ijb}$ using Eq. (6). Then, the corresponding elements in each row of $X_{(1)}$ are multiplied by $m_{n_1 j \sim n_2 j}$. When the multiplication is done, the memory space allocated for $m_{n_1 j \sim n_2 j}$ in LDM is released in order to save space for the incoming computation. Then the CPE pair continues to obtain the next element, e.g., $c_{(i+1)j}$, from matrix C to compute the next element of $\mathbb{N}$. Finally, we sum up all the elements in $\mathbb{N}$ that have the same i and j indexes, and derive the element at row i and column j of N, e.g., $N_{0j} = \sum \mathbb{N}_{0jb}$ shown in Fig. 6. By tiling matrices into small data blocks and applying the computation iteratively, we can eliminate the problem of intermediate

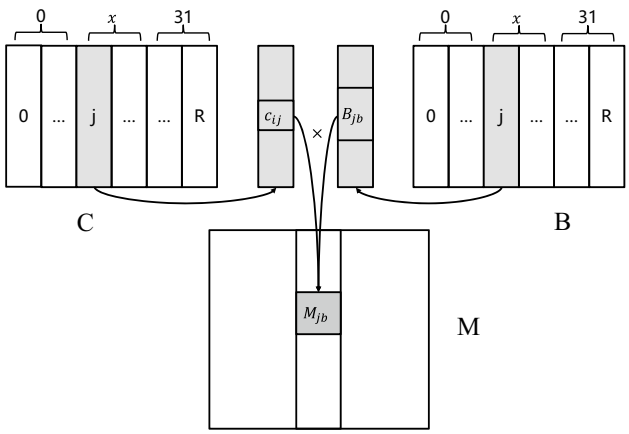


Fig. 5 Data tiling used in Sub-procedure 1. Each CPE pair retrieves an element c_{ij} from matrix C and multiplies the corresponding block of elements B_{jb} from matrix B in order to generate the element M_{jb} in matrix M

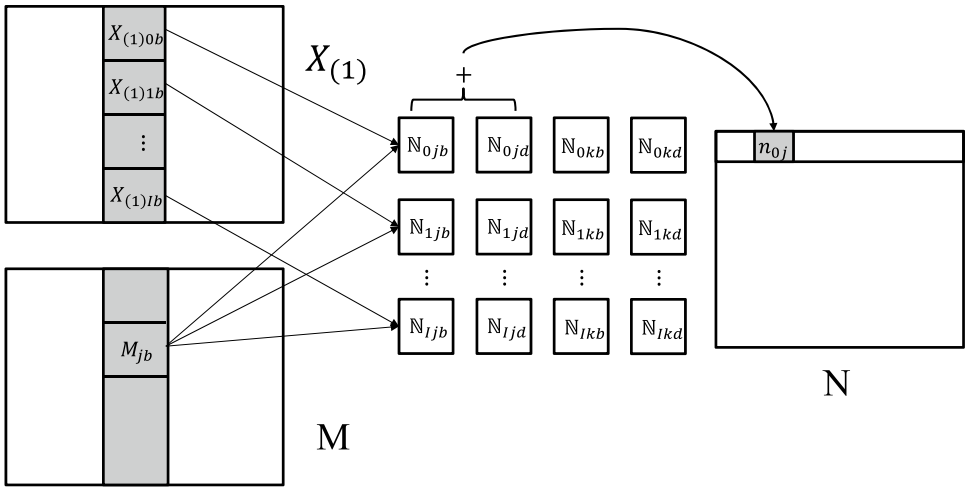


Fig. 6 Data tiling used in Sub-procedure 2. M_{jb} obtained from Sub-procedure 1, is multiplied by the corresponding vector from X_1, which generates $\mathbb{N}_{ijb}$. Then all elements in $\mathbb{N}$ that have the same i and j indexes are sum up to derive the according element for matrix N

data explosion and efficiently utilize the limited LDM on each CPE for better performance. We do not apply data tiling to Sub-procedure 3 and 4, since they generate small amount of intermediate data.

$$\mathbb{N}_{ijb} = \sum_{n=n_1,x=x_1}^{n_2,x_2} m_{nj} X_{(1)ix} \tag{6}$$

3.3 Coordinating computation within MapReduce

In this section, we present the customized design of key-value pair in *swTensor*, which adapts to the MapReduce framework on Sunway to perform the computation of CP decomposition.

For Sub-procedure 1 ($M = C \odot B$), the input key of map function is (*CRow*, *CCol*, *BlockID*), where *CRow* and *CCol* represent the location of element in factor matrix C, e.g. $c_{(CRow,CCol)}$. The *BlockID* represents the block index of the *CCol* column in matrix B. Considering that the span of column width is too large to fin in the limited capacity of LDM, *swMR* tiles the data based on the available space of LDM and then generates a BlockID for each block. The input value is (*BVal*, *Count*), where *BVal* is an array storing the data from the corresponding column of matrix B and *Count* is the number of elements in *BVal*, as shown in Fig. 5. Therefore, we can obtain a particular element $c_{(CRow,CCol)}$ from matrix C using the key. Then, $c_{(CRow,CCol)}$ is multiplied with *BVal* to derive *MVal*. After the computation of Sub-procedure 1, the output key is designed as (*CRow*, *BlockID*), which has the same meaning as input key. The output value is (*MVal*, *Count*), where *Count* represents the number of elements in *MVal*.

Algorithm 2 Retrieving elements of $X_{(1)}$ from main memory

```
1:  function OBTAINING_DATA(X_(1), I, J, BlockAddr)
2:      i = 0
3:      data1 = Data_from_MainMem(X_(1)[i × J + BlockAddr])
4:      i = i + 1
5:      data2 = Data_from_MainMem(X_(1)[i × J + BlockAddr])
6:      i = i + 1
7:      while i! = I and error not tolerable do
8:          while data1 is ready do
9:              Process data1
10:         end while
11:         data1 = Data_from_MainMem(X_(1)[i × J + BlockAddr])
12:         i = i + 1
13:         while data2 is ready do
14:             Process data2
15:         end while
16:         data2 = Data_from_MainMem(X_(1)[i × J + BlockAddr])
17:         i = i + 1
18:     end while
19: end function
```

For Sub-procedure 2 ($N = X_{(1)}M$), its input key-value pair is the output key-value pair from Sub-procedure 1, e.g., key = (*CRow*, *BlockID*) and value = (*MVal*, *Count*). According to Khatri-Rao product and tensor unfolding, the number of columns in matrix $X_{(1)}$ equals the number of rows in matrix M, which is JK. Then, we tile each row of matrix $X_{(1)}$ in the following way. First, we divide each row into K big blocks, where K is the row count of factor matrix C. Second, each big block is further divided into *Count* small blocks, where *Count* is obtained from input key-value pair. To implement the data tiling of matrix $X_{(1)}$, the *map* CPE obtains the corresponding data from $X_{(1)}$, as shown in Algorithm 2. We use double buffering technique to overlap the delay caused by accessing main memory with the computation. For instance, in Algorithm 2, when the *map* CPE finishes processing the current data block (line 9), it can continue to process next data block since it is already prefetched in the double buffer (line 14).

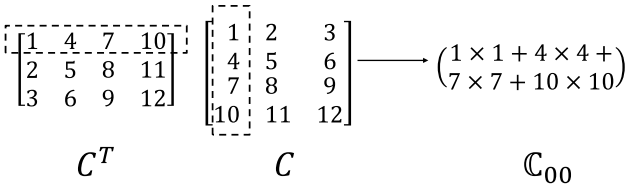


Fig. 7 An example to illustrate the computation of C^TC. Each element (e.g., $\mathbb{C}_{00}$) in $\mathbb{C}$ is derived by each row of C^T multiplied by itself. Therefore, only C^T needs to be stored in memory

For Sub-procedure 3 and 4 ($Q = (C^TC * B^TB)^\dagger$ and $\hat{A} = NQ$), Because C^TC and B^TB are computed in a same way, we take C^TC for illustration. As mentioned in Sect. 3.1, we use transposed factor matrices for better memory efficiency. Therefore we only store C^T and B^T. Fortunately, this does not affect computation of C^TC. As shown in Fig. 7, the computation of C^TC is equivalent to each row of C^T multiplied by itself. The design of key is (*RowID*, *BlockID*), where *RowID* represents the row index and *BlockID* represents the block index of the *CTVal*. And the value is (*CTVal*, *Count*), where *CTVal* is an array with the corresponding data from C^T and *Count* represents the size of *CTVal*. The output key is (*RowID*, *ColID*, *BlockID*), and the value is (*val*) where *val* is the result of C^TC. Because Hadamard product is an element-wise operation, we design the key/value as (*StartIndex*, *EndIndex*) and (*AVal*, *BVal*), respectively. Then *BVal* multiplies *AVal* in an element-wise way. Finally, Sub-procedure 4 ($\hat{A} = NQ$) only requires matrix multiplication to be performed. The design of key-value pair is similar to the computation of $X_{(1)}M$.

3.4 Scaling beyond core group

One limitation of the current MapReduce framework on Sunway is that it can only utilize one CG, and thus cannot scale across multiple Sunway processors. Since our CP decomposition *swTensor* is based on the above MapReduce framework, it hinders *swTensor* from handling much larger tensor data. Therefore, in order to overcome the scalability limitation, we extend *swTensor* using MPI so that it can run CP decomposition at large scale across multiple Sunway processors.

The fundamental idea for scaling the computation of CP decomposition is to use the MPE as master, which then divides factor matrices by columns and distributes the column partitions to other CGs for parallel processing with original design of *swTensor* unchanged. When the computation on each CG is completed, the master MPE gathers the partial results from all CGs and generates the final results. Note that we store the transposed factor matrices as described in Sect. 3.1, hence the factor matrices are actually divided by rows. For the ease of illustration, we describe our MPI extension based on the original matrix format.

For Sub-procedure 1, considering Kronecker products are column-wise operations, if we divide the factor matrices by rows, the elements of M are distributed across different CGs. In that case, the row size of $X_{(1)}$ does not match the row size of M in Sub-procedure 2. In order to fulfill the computation in Sub-procedure 2, it requires large amount of data transfers among CGs to gather the distributed elements of M and thus deteriorates the computation performance. Therefore, we partition the factor matrices by columns to ensure the matching of subsequent computation. Whereas for Sub-procedure 3, the C^TC is computed by rows, therefore similar idea is also applied to partition the matrix across multiple CGs for vector multiplication. As for Hadamard products, since they are element-wise operations, the same data partition method also applies. For Sub-procedure 4, which is the matrix multiplication, we apply the same idea to partition the matrices by columns.

4 Performance auto-tuning

4.1 Identifying the parameters

There are several parameters (e.g., px_1 to px_6) that could affect the performance of *swTensor*, which control the data tiling, LDM usage, DMA transfer and CPE parallelism in order to adapt to the Sunway architecture. For Sub-procedure 1, it tiles the factor matrices B and C based on the number of CPE pairs by column, shown as Fig. 5. After data tiling, the CPE fetches the corresponding data blocks from B and C, and processes the data by column. For instance, When a CPE pair processes the data of column j, we need to obtain the data of c_{ij} and a data vector $b_{n1j\sim n2j}$ of matrix B from main memory. Since the size of LDM is quite limited and we cannot load all data to LDM at once, parameter px_1 is used to control the amount of data fetched to LDM each time. For Sub-procedure 2, to compute N_{0jb}, the corresponding data block of $X_{(1)0b}$ needs to be loaded in LDM from main memory, as shown in Fig. 6. Since the LDM is also used to store the partial results of M_{jb} computed by Sub-procedure 2, there is not enough space to store the entire data block $X_{(1)0b}$. Therefore, we tile $X_{(1)0b}$ and use parameter px_2 to control the amount of data fetched from main memory for each tile.

In Sub-procedure 3 and 4, as shown in Fig. 7, the CPE loads the corresponding data of C^T and B^T from main memory to LDM. These two Sub-procedures suffer a similar problem of limited LDM and we apply a similar solution to tile the data fetched from main memory to LDM.

Hence, we use parameter px_3 and px_4 to control the amount of data fetched from matrices N and Q in main memory, respectively. In addition, the *swMR* framework also has two important parameters px_5 and px_6 that could impact the performance of *swTensor*. The parameter px_5 is an unbalanced threshold which judges whether load imbalance occurs between two CPEs within a CPE pair. When the performance difference between the two CPEs is more than px_5, *swMR* realizes load imbalance occurs and adjusts the load for the next round of computation within a CPE pair for better performance. Whereas parameter px_6 is the dynamic balancing ratio that controls the amount of computing load to be re-assigned from the heavy-loaded CPE to the light-loaded CPE within a CPE pair.

4.2 Building analytic performance model

To derive the optimal settings for the parameters in *swTensor* through exhaustive search would take either a prohibitively long time, or too much human efforts, which is impractical in real-world. Therefore, we propose an auto-tuning scheme to identify the optimal parameter settings of *swTensor*. In order to use the auto-tuning scheme, we need to build an analytic performance model $T(px_1, px_2, px_3, px_4, px_5, px_6)$, which can precisely measure the performance of *swTensor*.

The valid range and constraint of the parameters in the performance model is as follows:

- $0 < px_1 < LDM_{size}$, $0 < px_2 < LDM_{size}$ and $0 < px_1 + px_2 < LDM_{size}$, besides $px_1, px_2 \in N$.
- $0 < px_3 < LDM_{size}, 0 < px_4 < LDM_{size}$ and $px_3, px_4 \in N$.
- $0 \leqslant px_5 \leqslant 5000$ and $px_5 \in N$.
- $0 \leqslant px_6 \leqslant 1$ and $px_6 \in R$.

The whole procedure of *swTensor* is divided into 4 Sub-procedures. Hence, the performance model T can be built as Eq. (7). Within each Sub-procedure, there are two types of operations: *1)* computation including Khatri-Rao product, Hadamard product and matrix operations; *2)* accessing main memory through DMA. Since the computation result of Sub-procedure 1 is the input for Sub-procedure 2, we use $T_{sub1,2}$ to represent the combined performance of Sub-procedure 1 and 2, as shown in Eq. (8). Equation 8 can be further expanded as Eq. (9). Note that in each Sub-procedure, the DMA data transfer is overlapped with the computation, therefore we consider the maximum execution time of the overlapped operations in Eq. (9).

$$T = T_{sub1} + T_{sub2} + T_{sub3} + T_{sub4} \tag{7}$$

$$T = T_{sub1,2} + T_{sub3} + T_{sub4} \tag{8}$$

$$\begin{aligned} T = 3&\left(T_{DMAinit} + \sum^{n-2} max(T_{DMA}, T_{sub1,2}) + T_{sub1,2last} \right) \\ + 6&\left(T_{DMAinit} + \sum^{n-2} max(T_{DMA}, T_{sub3}) + T_{sub3last} \right) \\ + 3&\left(T_{DMAinit} + \sum^{n-2} max(T_{DMA}, T_{sub4}) + T_{sub4last} \right) \end{aligned} \tag{9}$$

The constant value 3 in Eq. (9) means that there are three factor matrices to compute, whereas the constant value 6 means that when we compute these three factor matrices, there are two types of computation to perform including matrix product and Hadamard product. T_{DMA} means the time for DMA transfer, where Eq. (10) show the details on how to calculate the DMA transfer time. Since the calculation of T_{DMA} is similar for each Sub-procedure, we omit the T_{DMA} equations for other Sub-procedures.

$$T_{DMA} = \begin{cases} \frac{(1-px_6\frac{T_0-T_1}{T_{0(1)}})(px_1+px_2)}{V_{DMA}}, & |T_0 - T_1| \geqslant px_5 \\ \frac{px_1+px_2}{V_{DMA}}, & |T_0 - T_1| < px_5, init\_round \end{cases} \tag{10}$$

The calculation of $T_{sub1,2}$, T_{sub3} and T_{sub4} is shown in Eqs. (11–13), respectively. T_0 and T_1 represent the computation time of CPE 0 and CPE 1 within a CPE pair, respectively. V_{DMA} represents the DMA bandwidth, and V_{flop} represents the peak floating point performance of CPE. Note that the initial value of T_0 and T_1 is 1, where $T_0 - T_1 = 0$ means that the load on CPE 0 and CPE 1 is balanced. After the first round of computation, our implementation evaluates whether load imbalance occurs between two CPEs within a CPE pair. If load imbalance is detected, T_0 and T_1 will be updated to $(1 - px_6)T_{subpre}$ and $px_6 T_{subpre}$, respectively, where T_{subpre} represents the execution time of previous Sub-procedure.

$$T_{sub1,2} = \begin{cases} \frac{\left(1-px_6\frac{T_0-T_1}{T_{0(1)}}\right)(px_1+px_2)+px_1}{V_{flop}}, & |T_0 - T_1| \geqslant px_5 \\ \frac{2px_1+px_2}{V_{flop}}, & |T_0 - T_1| < px_5, last\_round \end{cases} \tag{11}$$

$$T_{sub3} = \begin{cases} \frac{\left(1-px_6\frac{T_0-T_1}{T_{0(1)}}\right)px_3}{V_{flop}}, & |T_0 - T_1| \geqslant px_5 \\ \frac{px_3}{V_{flop}}, & |T_0 - T_1| < px_5, last\_round \end{cases} \tag{12}$$

$$T_{sub3} = \begin{cases} \frac{\left(1-px_6\frac{T_0-T_1}{T_{0(1)}}\right)px_4}{V_{flop}}, & |T_0 - T_1| \geqslant px_5 \\ \frac{px_4}{V_{flop}}, & |T_0 - T_1| < px_5, last\_round \end{cases} \tag{13}$$

4.3 Auto-tuning using simulated annealing algorithm

Based on the above analytic performance model, we use simulated annealing algorithm (Bertsimas and Tsitsiklis 1993; Aarts et al. 2007) to determine the optimal parameter settings. The simulated annealing algorithm tries to find the global optimal solution by accepting, with probability, a worse solution to step out local optimal solution.The auto-tuning procedure for *swTensor* is shown in Fig. 8. We first initialize the parameters randomly, within their value range. T means the temperature that we use in an auto-tuning algorithm, α means the ratio when decreasing the temperature, *solution* represents a set of parameter settings, and f represents the analytic performance model. For each iteration of the algorithm, a new solution named as neighbor solution, is generated and then the algorithm computes the increment dE of the analytic performance model f. If dE is less than zero then algorithm accepts the neighbor solution as a current solution, otherwise the neighbor solution is accepted with probability $p = exp(-dE/T)$. If the iteration meets the termination condition of the algorithm, the current solution will be the final optimal solution, otherwise the algorithm decreases the temperature to $T = \alpha \times T$, and generates a new neighbor solution to start over the iteration. In our case, the simulated annealing algorithm takes 24 minutes to derive the global optimal parameter settings, which is more efficient than exhaustive search.

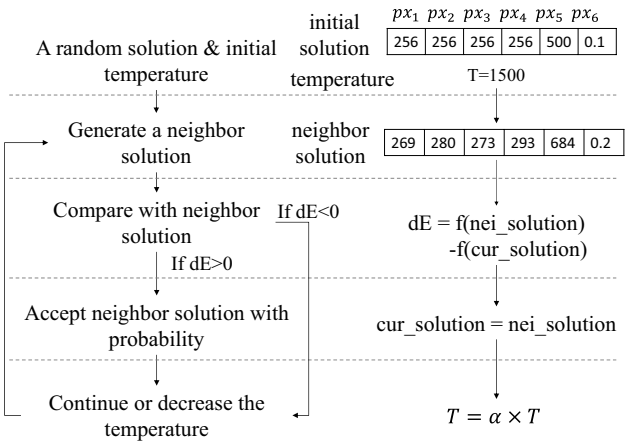


Fig. 8 The performance auto-tuning mechanism using simulated annealing

5 Evaluation

In this section, we first compare the performance of *swTensor* with existing work running on a CPU cluster with equivalent computation capability to a Sunway CG. Then, we build the roofline model to show how effectively our *swTensor* adapted to the Sunway architecture. Moreover, we present the performance results when scaling *swTensor* beyond a single CG. Finally, we use several real-world datasets to further demonstrate the performance advantage of *swTensor*. We provide performance comparison with existing works that are also based on MapReduce framework such as *BigTensor* and *CSTF*.

5.1 Experimental setup

We conduct the experiments on a Sunway SW26010 processor, which contains four CGs. Each CG consists of one MPE and 64 CPEs. Each Sunway processor has 32GB DDR3 memory. The detailed specifications of SW26010 are listed in Table 1. We compare the performance of *swTensor* with *BigTensor* (Park et al. 2016) and *CSTF* (Blanco et al. 2018), which are state-of-the-art tenor decomposition frameworks running on CPU. To provide a fair comparison, we run *BigTensor* on a CPU cluster of 10 nodes, which delivers equivalent peak floating point performance to a Sunway CG. The detailed specifications of each CPU node are also listed in Table 1. Moreover, we also port *BigTensor* to Sunway (denoted as *swBT*) without the optimizations proposed in *swTensor* to better understand the effectiveness of our approach. In order to evaluate the performance of *swTensor* under different sizes of tensor, we synthetically generate random tensor data (Kolda and Bader 2009) of size $I \times J \times K$, ranging from 2×10^3 to 9×10^3 and the number of nonzeros ranging from 2×10^5 to 6×10^5. The datasets evaluated are shown in Table 2.

5.2 Performance

We compare the performance of *swTensor* running on a CG of Sunway processor to *BigTensor* and *CSTF* running on a CPU cluster of 10 nodes due to the equivalent peak performance as shown in Table 1. This experimental setup intends to provide a fair comparison on processors with similar computation capacity. The performance results are shown in Fig. 9. It is clear that *swTensor* achieves better performance than *BigTensor* and *CSTF* across all tensor sizes with the average speedup of 1.36 × and 1.24 ×, respectively. Especially, when the tensor size is small, *swTensor* achieves more speedup over *BigTensor* and *CSTF* on the same dataset. The largest speedup achieved by *swTensor* is 1.62× and 1.56× running on dataset1 compared to *BigTensor* and *CSTF*, respectively. However, as the tensor size increases, the execution time of *swTensor*, *BigTensor* and *CSTF* increases accordingly. The performance advantage of *swTensor* becomes less significant compared to *BigTensor* and *CSTF* when the tensor size is large. When running on dataset5, the performance of *swTensor* is only 1.12 × and

Table 1 The hardware and software specifications of the Sunway processor and CPU cluster

Specifications		Sunway SW26010	Xeon E5620
Hardware	Core	1 MPE@1.45GHz and 64 CPEs@1.45GHz per CG	4 cores@2.4GHz
	Node	4 CGs	2 Xeon E5620
	Memory	32GB (8GB per CG)	12GB
	GFLOPS	750 (per CG)	768 (10 nodes)
Software	OS	customized	CentOS v6.5
	MapReduce	swMR (Zhong et al. 2018)	Hadoop v2.9.0
	JDK	–	OpenJDK v1.8.0
	MPI	MVAPICH2 v2.2a	–
	Compiler	sw5cc v5.421	–

1.03 × better than *BigTensor* and *CSTF*, respectively. The reason for the diminishing speedup of *swTensor* when the tensor size scales is due to the limited capacity of LDM on each CG. As the tensor size becomes too large to be cached in LDM of all CPEs, more data accesses are inevitable to read and write from main memory, which degrades the performance of *swTensor*. This observation justifies our contribution to scale *swTensor* to multiple CGs in order to handle larger tensor size. We also notice that *swBT* achieves the worst performance among all the CPD implementations. The results are as expected because *swBT* does not apply all the optimizations proposed in *swTensor* such as data tiling and coordinated computation, which leads to inefficient adaptation to the Sunway architecture. Since the performance of *swBT* is far below the rest of CPD implementations, we do not include *swBT* in the following evaluation of scalability and case study.

5.3 Roofline model analysis

To better understand how effective our *swTensor* is adapted to the Sunway architecture, we build a roofline model of a Sunway CG using similar approach in Xu et al. (2017). Note that the roofline model of Sunway has already been validated in Xu et al. (2017) with thorough experiments. Since the

Table 2 The tensor datasets evaluated in the experiments

Data	I	J	K	Nonzeros
Dataset1	2×10^3	2×10^3	2×10^3	2×10^5
Dataset2	4×10^3	4×10^3	4×10^3	3×10^5
Dataset3	6×10^3	6×10^3	6×10^3	4×10^5
Dataset4	7×10^3	7×10^3	7×10^3	5×10^5
Dataset5	9×10^3	9×10^3	9×10^3	6×10^5

I, J, K indicates the dimension of the tensor data

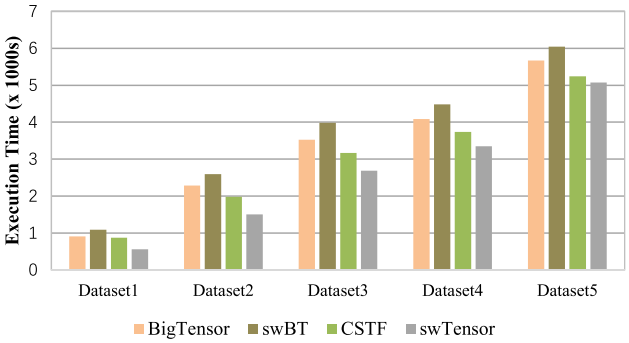


Fig. 9 The performance comparison of *swTensor*, *BigTensor*, *swBT* and *CSTF* on synthetic datasets

performance counters are quite limited on Sunway CPEs, it is difficult to measure the operational intensity of *swTensor* during runtime. Therefore, we calculate the operational intensity through algorithm analysis, which is also adopted in Williams et al. (2009). Note that the attainable performance of *swTensor* is still measured for the experiment run.

The advantage of roofline model is that it builds up relationships among peak floating point performance, operational intensity and memory bandwidth, which is quite illustrative to reveal the intrinsic characteristics of the application and provide guidance for performance optimization. To derive the operational intensity of CP decomposition, we analyze the computation procedure and realize the major floating point computation happens during step 2 as described in Sect. 3.1. The amount of floating point operations is $JK \times 2 \times I \times R$, where R is the rank of the tensor. In each iteration, the computation procedure is the same. Therefore, we only need to calculate the operational intensity for one iteration. In each iteration, there are three factor matrices accessed including A, B and C. Take the industry YELP dataset (Yelp dataset challenge 2019) for example, the calculation of operational intensity for CP decomposition is shown in Eq. (14).

$$\begin{aligned} OperationalIntensity &= Flops/Bytes \\ &= 2268GFlop * 3/728.5 * 10^9 Bytes \\ &= 9.34 \end{aligned} \tag{14}$$

The roofline model of a Sunway CG is shown in Fig. 10. The ridge point of the roofline model is 33.19, which means the application with operational intensity larger than the ridge point is no longer memory bounded. It is obvious that our *swTensor* reaches the operational intensity of 9.34, which is still under the slope of the roofline model and thus mostly bounded by the memory bandwidth. In the case of dataset (Yelp dataset challenge 2019), our *swTensor* achieves the performance of 523.39GFLOPS on a Sunway CG, which

reaches 69.78% efficiency of the theoretical peak performance of a Sunway CG. Indicated by the roofline model, there is still 30.22% performance space for further optimization. Therefore, to further exploit the computation power of a Sunway CG, we need to improve the operational intensity through vectorization and instruction re-ordering (Xu et al. 2017). However, we leave it for future work.

5.4 Scalability

To evaluate the scalability, we run *swTensor* across multiple Sunway nodes (each Sunway node contains one Sunway processor that consists of four CGs) ranging from 1 to 10. We also run *BigTensor* and *CSTF* on CPU cluster for comparison. The performance of a single node is used as the baseline. As shown in Fig. 11, *swTensor* exhibits much better scalability than *BigTensor* and *CSTF* on different datasets as the number of nodes scales. For small (Dataset1), medium (Dataset3) and large (Dataset5) size of tensor data, *swTensor* achieves 3.11 ×, 2.91 × and 2.76 × speedup, respectively when scaling to 10 nodes. Whereas for *BigTensor*, the speedup is only 2.32 ×, 2.12 × and 1.92 ×, respectively. For *CSTF* the speedup is only 1.21 ×, 1.30 × and 1.36 ×, respectively. The scalability of *CSTF* is worse than both *BigTensor* and *swTensor* due to the overhead of massive intermediate data generated during the computation. We also notice that the scalability of *swTensor* is far from linear, one reason for the sub-optimal scalability of *swTensor* could be attributed to our inefficient MPI implementation, which requires further optimization. Another reason could be, as the number of CGs increases while the size of tensor data stays constant, the computation assigned to each CG becomes insufficient to amortize the overhead of MPI communication. However, the concrete reasons still require further investigation in our future work.

5.5 Case study

In addition to the synthetic datasets, we evaluate *swTensor* with several real-world datasets as listed in Table 3. The *YELP* dataset (Yelp dataset challenge 2019) contains hundreds of thousands of reviews across 10,100 businesses, whereas the *ML-20M* dataset (Harper and Konstan 2016) contains 10,000,054 ratings and 95,580 tags applied to 10,681 movies from *MovieLens*. The *nell1* and *delliciously3d* datasets are from FROSTT (Smith et al. 2017), where *nell1* represents tensor with *noun-verb-noun* triplets, and *delliciously3d* is a *user-item-tag* tensor crawled from tagging systems. As shown in Fig. 12a, *swTensor* achieves better performance than *BigTensor* and *CSTF* on *YELP* and *ML-20M* datasets. For the *YELP* dataset, *swTensor* achieves 1.31 × and 1.18 × speedup compared to *BigTensor* and *CSTF*, respectively, whereas for the *ML-20M* dataset *swTensor* achieves 1.33 × and 1.22 × speedup, respectively. In Fig. 12b, on *nell1* and *delliciously3d* datasets, *swTensor* and *CSTF* achieve similar performance, both of which are better than *BigTensor*. Note that the reason why the performance of *CSTF* compared to *BigTensor* is different from the results reported in Blanco et al. (2018) can be understood from two aspects: (1) The hardware settings of the experiment platforms are not exactly the same. Except CPU, the performance specifications of memory and disk subsystems may also be different. (2) The *CSTF* is implemented on top of Spark, which is more sensitive to CPU performance compared to *BigTensor* that is implemented on top of Hadoop. In our experiment platform, the CPU is less powerful compared to Blanco et al. (2018) in terms of both core number as well as CPU frequency, which constrains the performance advantage of *CSTF* compared to *BigTensor*. Nevertheless, the results with the real-world datasets also demonstrate the effectiveness of our *swTensor* for supporting tensor decomposition on Sunway architecture.

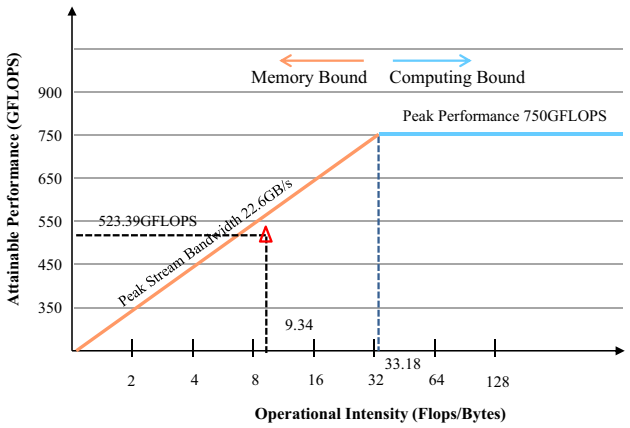


Fig. 10 The roofline model of a Sunway CG when running *swTensor* with the YELP dataset

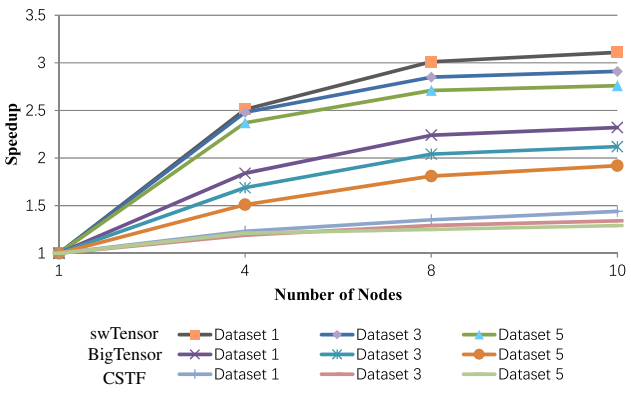


Fig. 11 The scalability of *swTensor*, *BigTensor* and *CSTF* running on different datasets across multiple nodes

Table 3 The real-world tensor datasets

Data	I	J	K	Nonzeros
YELP (Yelp dataset challenge 2019)	70×10^3	15×10^3	108	334×10^3
ML-20M (Harper and Konstan 2016)	71×10^3	10×10^3	157	10×10^6
nell1 (Smith et al. 2017)	2×10^6	2×10^6	25×10^6	144×10^6
dellicious3d (Smith et al. 2017)	532×10^3	17×10^6	2×10^6	140×10^6

I, J, K indicates the dimension of the tensor data

6 Related work

There are many research works that have been devoting significant efforts to optimize the algorithm of tensor decomposition. The computation of MTTKRP is identified to be the major performance bottleneck by carefully analyzing and instrumenting the computation of MTTKRP (Choi et al. 2018). Moreover, the authors in Choi et al. (2018) combine two data blocking methods and apply it to the MTTKRP, which achieves considerable speedup. *eALS* (element-wise Alternating Least Squares) (He et al. 2016), based on the traditional ASL algorithm, offers a new way to optimize tensor decomposition for online recommendation system. The dynamically changing features in real-world data are taken into consideration by eALS. A highly scalable tensor decomposition scheme, which is effective for speedup relation extraction, has been proposed by Chang et al. (2014). DFacTo (Choi and Vishwanathan 2014) presents a distributed algorithm that exploits the properties of the Khatri-Rao product to accelerate Alternating Least Squares (ALS) and Gradient Descent (GD) algorithms used in tensor factorization.

Besides, tensor decomposition has been adapted to various hardware architectures for better performance. On the Intel many-core processor KNL (Knights Landing), the computation of CP decomposition is balanced among the processing units, which leads to 1.8 × performance speedup (Smith et al. 2017). Li et al. (2016); Ma et al. (2019) propose an optimized design of sparse tensor-times-dense matrix multiply on GPU that exploits fine thread granularity, coalesced memory access, rank blocking and fast shared memory. F-COO (Liu et al. 2017) proposes a unified tensor format along with GPU-specific optimizations that leverages the similar computation patterns between tensor operations. The implementation using F-COO achieves better performance than the implementation using vendor libraries on GPU. Most recently, balanced-CSF (Nisa e al. 2019) (balanced compressed sparse fiber) format has been proposed on GPU that extends the CSF format to better utilize the massive parallelism on GPU for accelerating the irregular computation of sparse MTTKRP with load balance. GTA (Oh et al. 2019) provides a general framework for Tucker factorization on both CPU and GPU that implements alternating least squares with a row-wise update in parallel, which effectively accelerates the factor matrice update process with less memory consumption. Phipps and Kolda (2019) propose a portable approach to determine the level of parallelism for MTTKRP on different architectures, which includes fine-grained parallelism, portable thread-local arrays and atomic-write contention avoidance.

Moreover, there are research works adapting CP decomposition to MapReduce framework to accelerate tensor decomposition with better parallelism and automatic scalability. *FlexiFaCT* (Beutel et al. 2014), *HaTen2* (Jeon et al. 2015), *Gigatensor* (Kang et al. 2012) and *BigTensor* (Park et al. 2016) all take advantage of MapReduce framework to accelerate tensor decomposition on cluster. *FlexiFaCT*

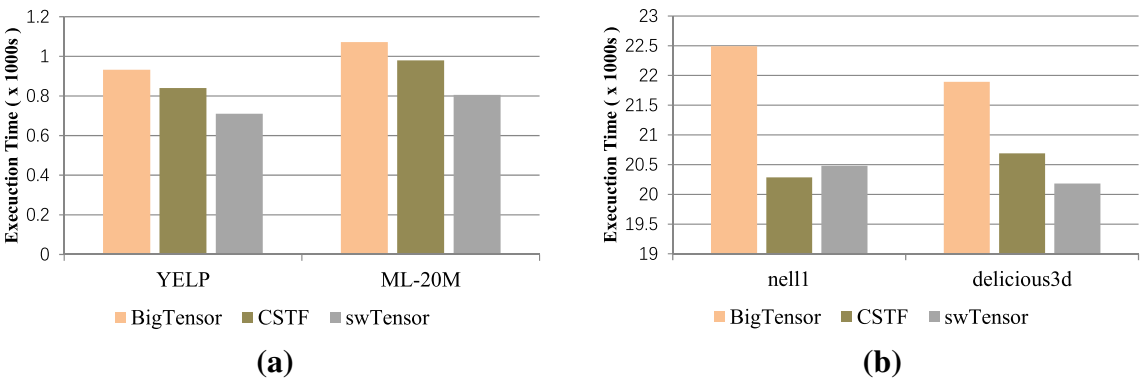


Fig. 12 The performance comparison of *swTensor*, *BigTensor* and *CSTF* on real-world datasets

supports decomposition on both matrix and tensor by using stochastic gradient descent on different objective functions. *HaTen2* reduces the amount of intermediate data and the number of computing jobs when the data size is tremendous. *GigaTensor* uses a novel computing algorithm to avoid intermediate data explosion as well as reduce the number of floating point operations, which demonstrates its effectiveness by evaluating with real-world datasets. *CSTF* (Blanco et al. 2018) proposes a novel queuing strategy to exploit the data reuse between the computation procedures in CP decomposition that reduces the communication cost significantly.

In the meanwhile, there are surging research works (Zhong et al. 2018; Liu et al. 2018; Hu et al. 2019; Liu et al. 2019; Han et al. 2019; Chen et al. 2018; Li et al. 2018a, b; Duan et al. 2018) based on Sunway architecture in the past few years, which provide valuable experience to our work. The achievable performance by leveraging the architecture features of Sunway such as memory architecture, CPEs and register communication, is quantitatively measured by Xu et al. (2017) with both memory-bound and computing-bound benchmarks. The observations described in Xu et al. (2017) provide useful insights for performance optimization on Sunway architecture. *swMR* (Zhong et al. 2018), a MapReduce programming framework based on Sunway architecture, leverages the computing resources of Sunway processor to automatically parallelize the map/reduce processing and optimize the performance using the unique architectural features such as CPEs and register communication. A sparse matrix vector multiplication algorithm optimized for Sunway architecture, is proposed by Liu et al. (2018). The proposed technique optimizes the sparse matrix vector multiplication by tiling resource and data into three levels, and then leverage register communication and local device memory to implement effective data transfer and better usage of CPEs. Li et al. (2018b) re-designs the computation of sparse triangular solver (SpTRSV) by assigning different roles to CPEs within a CG on Sunway processor. This multi-role design provides an efficient implementation of SpTRSV on Sunway by carefully manipulating the local device memory and register communication for synchronization.

7 Conclusion and future work

In this paper, we present an efficient implementation of tensor decomposition *swTensor* by adapting the architecture features of Sunway many-core architecture. Specifically, we leverage the Sunway MapReduce framework *swMR* for auto-parallelization of tensor decomposition, and tile the tensor data based on the 32 CPE pairs in *swMR* for load balance. In addition, we design customized key-value pairs to utilize the parallel processing of *swMR* and divide the original CP decomposition into four sub-procedures in order to avoid intermediate data explosion with the limited LDM on Sunway. Moreover, we propose a performance auto-tuning mechanism to search for the optimal parameter settings for *swTensor*. The experiment results show that *swTensor* performs better than the state-of-the-art *BigTensor* and *CSTF* in both performance and scalability. For the future work, we would like to extend *swTensor* to support more types of tensor decomposition such as Tucker decomposition (Tucker 1963), which are widely used in various application fields. Moreover, we would like to further optimize the MPI implementation of *swTensor* so that it can process tensor data with massive size by utilizing the Sunway processors at large scale.

Acknowledgements The authors would like to thank all anonymous reviewers for their insightful comments and suggestions. This work is supported by National Key R&D Program of China (Grant No. 2016YFB1000503 and 2016YFA0602200), National Natural Science Foundation of China (Grant No. 61502019 and 61732002), State Key Laboratory of Software Development Environment (Grant No. SKLSDE-2018ZX-19), Center for High Performance Computing and System Simulation, Pilot National Laboratory for Marine Science and Technology (Qingdao). Hailong Yang is the corresponding author.

Compliance with ethical standards

Conflict of interest On behalf of all authors, the corresponding author states that there is no conflict of interest.

References

Aarts, E., Korst, J., Michiels, W.: Simulated annealing. Local Search Combin. Optim. **36, partii**(3), 187–210 (2007)

Acar, E., Dunlavy, D.M., Kolda, T.G., Mørup, M.: Scalable tensor factorizations for incomplete data. Chemometr. Intell. Lab. Syst. **106**(1), 41–56 (2011)

Aja-Fernández, S., de Luis Garcia, R., Tao, D., Li, X.: Tensors in Image Processing and Computer Vision. Springer, Berlin (2009)

Bertsimas, D., Tsitsiklis, J.: Simulated annealing. Stat. Sci. **8**(1), 10–15 (1993)

Beutel, A., Talukdar, P.P., Kumar, A., Faloutsos, C., Papalexakis, E.E., Xing, E.P.: Flexifact: scalable flexible factorization of coupled tensors on hadoop. In: Proceedings of the 2014 SIAM International Conference on Data Mining, pp. 109–117. SIAM (2014)

Blanco, Z., Liu, B., Dehnavi, M.M.: Cstf: Large-scale sparse tensor factorizations on distributed platforms. In: Proceedings of the 47th International Conference on Parallel Processing, p. 21. ACM (2018)

Chang, K.W., Yih, S.W.t., Yang, B., Meek, C.: Typed tensor decomposition of knowledge bases for relation extraction (2014)

Chen, B., Fu, H., Wei, Y., He, C., Zhang, W., Li, Y., Wan, W., Zhang, W., Gan, L., Zhang, W., et al.: Simulating the Wenchuan earthquake with accurate surface topography on Sunway taihulight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 40. IEEE Press (2018)

Choi, J., Liu, X., Smith, S., Simon, T.: Blocking optimization techniques for sparse tensor computation. In: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp. 568–577. IEEE (2018)

Choi, J.H., Vishwanathan, S.V.N.: Dfacto: distributed factorization of tensors. Adv. Neural Inf. Process. Syst. **2**, 1296–1304 (2014)

Cichocki, A., Mandic, D., De Lathauwer, L., Zhou, G., Zhao, Q., Caiafa, C., Phan, H.A.: Tensor decompositions for signal processing applications: from two-way to multiway component analysis. IEEE Signal Process. Mag. **32**(2), 145–163 (2015)

Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107–113 (2008)

Duan, X., Gao, P., Zhang, T., Zhang, M., Liu, W., Zhang, W., Xue, W., Fu, H., Gan, L., Chen, D., et al.: Redesigning lammps for peta-scale and hundred-billion-atom simulation on sunway taihulight. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, p. 12. IEEE Press (2018)

Golub, G.H., Van Loan, C.F.: Matrix Computations, vol. 3. JHU Press, Baltimore (2012)

Han, Q., Yang, H., Luan, Z., Qian, D.: Accelerating tile low-rank gemm on Sunway architecture: Poster. In: Proceedings of the 16th ACM International Conference on Computing Frontiers, pp. 295–297. ACM (2019)

Harper, F.M., Konstan, J.A.: The movielens datasets: history and context. ACM Trans. Interact. Intell. Syst. (tiis) **5**(4), 19 (2016)

He, X., Zhang, H., Kan, M.Y., Chua, T.S.: Fast matrix factorization for online recommendation with implicit feedback. In: Proceedings of the 39th International ACM SIGIR conference on Research and Development in Information Retrieval, pp. 549–558. ACM (2016)

Hitchcock, F.L.: The expression of a tensor or a polyadic as a sum of products. J. Math. Phys. **6**(1–4), 164–189 (1927)

Hu, Y., Yang, H., Luan, Z., Qian, D.: Massively scaling seismic processing on sunway taihulight supercomputer. arXiv preprint arXiv :1907.11678 (2019)

Jeon, I., Papalexakis, E.E., Faloutsos, C., Sael, L., Kang, U.: Mining billion-scale tensors: algorithms and discoveries. Int. J. Very Large Data Bases **25**(4), 519–544 (2016)

Jeon, I., Papalexakis, E.E., Kang, U., Faloutsos, C.: Haten2: Billion-scale tensor decompositions. In: Data Engineering (ICDE), 2015 IEEE 31st International Conference on, pp. 1047–1058. IEEE (2015)

Kang, U., Papalexakis, E., Harpale, A., Faloutsos, C.: Gigatensor: scaling tensor analysis up by 100 times-algorithms and discoveries. In: Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 316–324. ACM (2012)

Kolda, T.G., Bader, B.W.: Tensor decompositions and applications. SIAM Rev. **51**(3), 455–500 (2009)

Lei, C., Yang, Y.H.: Tri-focal tensor-based multiple video synchronization with subframe optimization. IEEE Trans. Image Process. **15**(9), 2473–2480 (2006)

Li, J., Ma, Y., Yan, C., Vuduc, R.: Optimizing sparse tensor times matrix on multi-core and many-core architectures. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, pp. 26–33. IEEE Press (2016)

Li, L., Yu, T., Zhao, W., Fu, H., Wang, C., Tan, L., Yang, G., Thomson, J.: Large-scale hierarchical k-means for heterogeneous many-core supercomputers. In: SC18: International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 160–170. IEEE (2018a)

Li, M., Liu, Y., Yang, H., Luan, Z., Qian, D.: Multi-role sptrsv on sunway many-core architecture. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), pp. 594–601. IEEE (2018b)

Lim, L.H., Comon, P.: Multiarray signal processing: Tensor decomposition meets compressed sensing. arXiv preprint arXiv:1002.4935 (2010)

Liu, B., Wen, C., Sarwate, A.D., Dehnavi, M.M.: A unified optimization approach for sparse tensor operations on gpus. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER), pp. 47–57. IEEE (2017)

Liu, C., Xie, B., Liu, X., Xue, W., Yang, H., Liu, X.: Towards efficient spmv on sunway manycore architectures. In: Proceedings of the 2018 International Conference on Supercomputing, pp. 363–373. ACM (2018)

Liu, C., Yang, H., Sun, R., Luan, Z., Qian, D.: swtvm: Exploring the automated compilation for deep learning on sunway architecture. arXiv preprint arXiv:1904.07404 (2019)

Ma, Y., Li, J., Wu, X., Yan, C., Sun, J., Vuduc, R.: Optimizing sparse tensor times matrix on gpus. J. Parallel Distrib. Comput. **129**, 99–109 (2019)

Nickel, M., Tresp, V., Kriegel, H.P.: Factorizing yago: scalable machine learning for linked data. In: Proceedings of the 21st international conference on World Wide Web, pp. 271–280. ACM (2012)

Nisa, I., Li, J., Sukumaran-Rajam, A., Vuduc, R., Sadayappan, P.: Load-balanced sparse mttkrp on gpus. arXiv preprint arXiv :1904.03329 (2019)

Oh, S., Park, N., Jang, J.G., Sael, L., Kang, U.: High-performance tucker factorization on heterogeneous platforms. IEEE Trans. Parallel Distrib. Syst. (2019)

Park, N., Jeon, B., Lee, J., Kang, U.: Bigtensor: Mining billion-scale tensor made easy. In: Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, pp. 2457–2460. ACM (2016)

Phipps, E.T., Kolda, T.G.: Software for sparse tensor decomposition on emerging computing architectures. SIAM J. Sci. Comput. **41**(3), C269–C290 (2019)

Shashua, A., Hazan, T.: Non-negative tensor factorization with applications to statistics and computer vision. In: Proceedings of the 22nd international conference on Machine learning, pp. 792–799. ACM (2005)

Sidiropoulos, N.D., De Lathauwer, L., Fu, X., Huang, K., Papalexakis, E.E., Faloutsos, C.: Tensor decomposition for signal processing and machine learning. IEEE Trans. Signal Process. **65**(13), 3551–3582 (2017)

Smith, S., Choi, J.W., Li, J., Vuduc, R., Park, J., Liu, X., Karypis, G.: Frostt: The formidable repository of open sparse tensors and tools (2017)

Smith, S., Park, J., Karypis, G.: Sparse tensor factorization on many-core processors with high-bandwidth memory. In: Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International, pp. 1058–1067. IEEE (2017)

Sonka, M., Hlavac, V., Boyle, R.: Image processing, analysis, and machine vision. Cengage Learning (2014)

Tew, P.A.: An investigation of sparse tensor formats for tensor libraries. Ph.D. thesis, Massachusetts Institute of Technology (2016)

Tsourakakis, C.E.: Data mining with mapreduce: Graph and tensor algorithms with applications. Diss. Master's thesis, Carnegie Mellon University (2010)

Tucker, L.R.: Implications of factor analysis of three-way matrices for measurement of change. Probl. Meas. Change **15**, 122–137 (1963)

Wang, X., Xue, W., Liu, W., Wu, L.: swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures. In: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 338–353. ACM (2018)

Williams, S., Waterman, A., Patterson, D.: Roofline: An insightful visual performance model for floating-point programs and multicore architectures. Tech. rep., Lawrence Berkeley National Lab. (LBNL), Berkeley (2009)

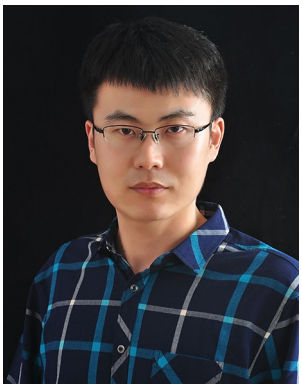
Xu, Z., Lin, J., Matsuoka, S.: Benchmarking sw26010 many-core processor. In: Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International, pp. 743–752. IEEE (2017)

Yelp dataset challenge. (2019) https://www.yelp.com/dataset/challenge
Zhong, X., Li, M., Yang, H., Liu, Y., Qian, D.: swmr: A framework for accelerating mapreduce applications on sunway taihulight. IEEE Trans. Emerg. Topics Comput. (2018)

Xiaogang Zhong is a master student in School of Computer Science and Engineering, Beihang University. He is currently working on big data processing optimization. His research interests include parallel programming, computational graph optimization and computer architecture.

Hailong Yang is an assistant professor in School of Computer Science and Engineering, Beihang University. He received the Ph.D degree in the School of Computer Science and Engineering, Beihang University in 2014. He has been involved in several scientific projects such as performance analysis for big data systems and performance optimization for large scale applications. His research interests include parallel and distributed computing, HPC, performance optimization and energy efficiency. He is a member of China Computer Federation (CCF).

Zhongzhi Luan received the Ph.D. in the School of Computer Science of Xi'an Jiaotong University. He is an Associate Professor of Computer Science and Engineering, and Assistant Director of the Sino-German Joint Software Institute (JSI) Laboratory at Beihang University, China. Since 2003, His research interests including distributed computing, parallel computing, grid computing, HPC and the new generation of network technology.



Lin Gan is an assistant researcher in the Department of Computer Science and Technology at Tsinghua University, and the assistant director of the National Supercomputing Center in Wuxi. His research interests include high performance computing solutions based on hybrid platforms such as GPUs, FPGAs, and Sunway CPUs. Gan received a PhD in computer science from Tsinghua University. He is the recipient of the 2016 ACM Gordon Bell Prize, the 2017 ACM Gordon Bell Prize Finalist, the 2018 IEEE-CS TCHPC Early Career Researchers Award for Excellence in HPC, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc. He is a member of IEEE

Guangwen Yang is a professor in the Department of Computer Science and Technology at Tsinghua University, and the director of the National Supercomputing Center in Wuxi. His research interests include parallel algorithms, cloud computing, and the earth system model. Yang received a PhD in computer science from Tsinghua University. He has received the ACM Gordon Bell Prize in the year of 2016 and 2017, and the Most Significant Paper Award in 25 Years awarded by FPL 2015, etc. He is a member of IEEE.

Depei Qian is a professor at the Department of Computer Science and Engineering, Beihang University, China. He received his master degree from University of North Texas in 1984. He is currently serving as the chief scientist of China National High Technology Program (863 Program) on high productivity computer and service environment. He is also a fellow of China Computer Federation (CCF). His research interests include innovative technologies in distributed computing, high performance computing and computer architecture.