Information Sciences 549 (2021) 221–248

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Towards efficient canonical polyadic decomposition on sunway many-core processor

Ming Dun [a], Yunchun Li [a,b], Qingxiao Sun [b], Hailong Yang [b,*], Wei Li [b], Zhongzhi Luan [b], Lin Gan [c], Guangwen Yang [c], Depei Qian [b]

[a] *School of Cyber Science and Technology, Beihang University, Beijing, China*
[b] *School of Computer Science and Engineering, Beihang University, Beijing, China*
[c] *Department of Computer Science and Technology, Tsinghua University, Beijing, China*

ARTICLE INFO

Article history:
Received 29 November 2019
Received in revised form 18 October 2020
Accepted 12 November 2020
Available online 1 December 2020

Keywords:
canonical polyadic decomposition
Sunway architecture
performance optimization

ABSTRACT

Canonical Polyadic Decomposition (CPD) is one of the most popular tensor decomposition methods and plays an important role in big data analysis. For sparse tensor, the major computation procedure in CPD, which is known as matricized tensor times Khatri-Rao product (MTTKRP), exhibits discontinuous memory access and turns to be the performance bottleneck from achieving high performance on emerging processor architectures. In this paper, we propose *swCPD*, an efficient CPD implementation on the many-core Sunway processor. The *swCPD* accelerates the optimization algorithms dominating the performance of MTTKRP, including Alternating Least Squares (ALS), Gradient Descent (GD) and Randomized Block Sampling (RBS), as well as the latest fast Levenberg–Marquardt (fLM++) and Generalized Canonical Polyadic Decomposition with Stochastic Gradient Descent (GCP-SGD). The main idea adopted in *swCPD* is a hierarchical partitioning mechanism. From the computation perspective, the 64 Computation Processing Elements (CPEs) in a Sunway processor are divided into eight *groups*, with each *group* containing seven *workers* and one *controller*. From the data perspective, we partition the sparse tensor into different granularities, which are *blocks*, *bands* and *tiles*. Moreover, we develop a communication mechanism through register communication for cooperation between CPEs. We evaluate the implementation of *swCPD* with both synthesized and real-world datasets. The experiment results show that each optimized algorithm in *swCPD* achieves better performance than corresponding algorithms adopted in cutting-edge CPD implementations.

© 2020 Elsevier Inc. All rights reserved.

1. Introduction

With the evolution of big data analysis, tensors have been applied to various fields such as image processing [1], computer vision [2], and recommendation system [3]. Therefore, tensor-based analytic techniques have gained popularity in HPC community for estimating relationships among huge quantities of multi-dimensional data. Within these techniques, tensor decomposition aims to factorize high-order tensors into the sum of multiple rank-one tensors in order to understand the

* Corresponding author.
E-mail addresses: dunming0301@buaa.edu.cn (M. Dun), lych@buaa.edu.cn (Y. Li), qingxiaosun@buaa.edu.cn (Q. Sun), hailong.yang@buaa.edu.cn (H. Yang), liw@buaa.edu.cn (W. Li), 07680@buaa.edu.cn (Z. Luan), lingan@tsinghua.edu.cn (L. Gan), ygw@tsinghua.edu.cn (G. Yang), depeiq@buaa.edu.cn (D. Qian).

https://doi.org/10.1016/j.ins.2020.11.013
0020-0255/© 2020 Elsevier Inc. All rights reserved.

relationships embedded in the data better. Canonical Polyadic Decomposition (CPD) is one of the most widely-applied schemes of tensor decomposition.

Although there are a large number of research works on how to apply CPD on dense tensors [4,5], they cannot be directly applied to extremely large and sparse tensors. On the one hand, when using the conventional coordinate format (COO) [6], the enormous dataset and corresponding factor matrices take up too large space to be stored in the cache, which increases cache miss and degrades performance. On the other hand, due to different sparse patterns across different datasets, the conventional CPD algorithm for dense tensors faces the challenge of discontinuous memory accesses that causes potential write conflicts, thus can severely deteriorate the algorithm parallelism. Therefore, advanced researches focus on efficient parallel CPD implementations for sparse and large tensors. For instance, Kang *et al.* utilized the MapReduce framework [7] to implement Gigatensor [8] on multicore CPUs, which also adopted Hadamard products for reducing memory footprint. In addition, Li *et al.* proposed *ParTI!* [9] on both multicore CPUs and manycore GPU to support a variety of tensor operations including CPD. However, the cutting-edge CPD libraries are only implemented on CPUs and GPUs.

Meanwhile, the Sunway many-core architecture has become an attractive architecture improve the performance of CPD further. Ranking the first in the TOP500 list in 2015 [10], Sunway TaihuLight supercomputer system reaches peak performance of 125PFLOPS [11]. The Sunway TaihuLight supercomputer has already been applied in a number of research fields, including numerical algorithms [12], climatic simulation [13] and machine learing [14]. This China homegrown supercomputer system contains 40,960 pieces of Sunway SW26010 processors. Each of the Sunway processors consists of 4 core groups (CGs). There are 64 Computation Processing Elements (CPEs) and one Management Processing Element (MPE) within a CG. Each CPE equips with a 64 KB manually-controlled Local Device Memory (LDM). Moreover, DMA and register communication are enabled on CPEs to improve memory access and communication efficiency.

To implement CPD algorithm efficiently on Sunway, it is of great importance to re-design CPD algorithm to adapt to the unique architectural features of Sunway. In this paper, we propose *swCPD*, an efficient CPD implementation on Sunway to improve the performance of tensor decomposition. The *swCPD* contains five optimization algorithms for determining the factor matrices during CPD, including Alternating Least Squares (ALS), Gradient Descent (GD), and Randomized Block Sampling (RBS), as well as the latest fast Levenberg–Marquardt (fLM++) and Generalized Canonical Polyadic Decomposition with Stochastic Gradient Descent (GCP-SGD). These optimization algorithms dominate the performance of Matricized Tensor Times Khatri-Rao Product (MTTKRP), whose performance suffers from discontiguous memory access pattern and synchronization [15]. For the exact MTTKRP in CPD-ALS, CPD-GD and CPD-fLM++, to optimize the memory access, we leverage the blocking mechanism and the CSF tensor format [16] to reduce irregular memory accesses. Specifically, we divide the sparse tensor into eight *blocks* and CPEs into eight *groups*, each *group* computes MTTKRP on one *block*. We further partition each *block* into several *bands*, and each *band* contains non-empty rows of the tensor. Besides, the *bands* are divided into *tiles*, and each *tile* consists of several *fibers*. To optimize the synchronization, we divide the eight CPEs within a *group* into seven *workers* and one *controller*. The *worker* is used for computing MTTKRP, and the *controller* is used for assigning tensor *bands* in the *blocks* to *workers* and storing the updated parts of factor matrices back to main memory. The synchronization between *worker* and *controller* is enforced through register communication with carefully-designed message scheme. Whereas, for the sampled MTTKRP in CPD-RBS and CPD-GCP-SGD, we propose a randomization strategy on top of the exact MTTKRP through shuffling among CPEs with register communication to derive the fully randomized sample.

We implement *swCPD* on the Sunway processor and compare each the performance of each algorithm in *swCPD* with corresponding algorithms adopted in cutting-edge CPD implementations. The experimental results show that our approach can utilize Sunway's architecture features efficiently and thus achieve significant speedup. Specifically, this paper makes the following contributions:

- We leverage the blocking mechanism that partitions the tensors and CPEs into eight *blocks* and eight *groups*, respectively. The *blocks* are further divided into *bands*, which contain *tiles* when computing mode-n MTTKRP.
- We present a multi-role computation scheme for CPD that divides CPEs within a *group* as two roles, which are named as *worker* and *controller*, respectively. This scheme enables efficient synchronization among CPEs within a *group* through register communication.
- We implement *swCPD* that includes CPD-ALS, CPD-GD, CPD-RBS, CPD-fLM++ and CPD-GCP-SGD on Sunway processor, and evaluate its performance by comparing with cutting-edge CPD implementations using both synthesized and real-world datasets.

This paper is organized as follows. We introduce the background of CPD and Sunway architecture in Section 2. Section 3 describes our methodologies of *swCPD*, an efficient implementation of CPD including CPD-ALS, CPD-GD, CPD-RBS, CPD-fLM++ and CPD-GCP-SGD on Sunway processor. Section 4 presents the implementation details of *swCPD*. Section 5 describes the performance auto-tuning approach adopted in *swCPD* to determine the optimal parameters. In Section 6, we present the experimental results through comparison with cutting-edge CPD implementations. Section 7 presents the related works and we conclude our work in Section 8.

2. Background

2.1. Tensor decomposition

2.1.1. Notations and preliminaries

A *tensor* is a multi-dimensional array [4], with each dimension denoted as a *mode* and the number of dimensions is denoted as *order*. An element in an N-th order tensor has N indices, and if a tensor has more than three indices, it is known as a high-order tensor. For the ease of illustration, we use three-dimensional tensor as an example to describe the mathematical details of tensor decomposition.

Before diving into the details, we first explain the notations used throughout this paper. A high-order tensor is denoted with calligraphic letters like $\mathcal{X}$, whose element with coordinate of (i,j,k) is denoted as x_{ijk}. Whereas matrix is denoted with bold upper case letters as $\mathbf{A}$ with its element denoted as a_{ij}. Besides, a vector is denoted with bold lower case letters like $\mathbf{a}$ whose element is denoted as a_i. Moreover, a *fiber* of a tensor $\mathcal{X}$ means changing one of its indices while keeping others fixed, which is denoted as $\mathcal{X}(i,j,:)$ for three-dimensional tensor, while a *row* of a tensor means changing two indices and keeping others fixed. For mathematical operations, the mode-n tensor matricization of tensor $\mathcal{X}$ is denoted as $\mathcal{X}_{(n)}$, which refers to *fiber* re-arrangement. The Frobenius norm of the matrix $\mathbf{A}$ is denoted as $\|\mathbf{A}\|$. Besides, the $\mathbf{A}^T$ denotes the transpose of matrix $\mathbf{A}$. In addition, there are two fundamental operations in tensor decomposition, which are the Khatri-Rao product and Hadamard product. We use $\mathbf{A} \odot \mathbf{B}$ to denote the Khatri-Rao product. Given two column-matching matrices $\mathbf{A} \in \mathbb{R}^{I \times J}$ and $\mathbf{B} \in \mathbb{R}^{M \times J}$, the Khatri-Rao product is defined in Eq. (1), where $\mathbf{a}_i$ and $\mathbf{b}_i$ denote the column vector of the two matrices, respectively. We use $\mathbf{A} * \mathbf{B}$ to denote the Hadamard product, which is defined in Eq. (2). Note that both $\mathbf{A}$ and $\mathbf{B}$ need to be in the same dimension.

$$\mathbf{A} \odot \mathbf{B} = [\mathbf{a}_1 \otimes \mathbf{b}_1, \ldots, \mathbf{a}_j \otimes \mathbf{b}_j] \tag{1}$$

$$c_{ij} = a_{ij} b_{ij} \tag{2}$$

Canonical Polyadic Decomposition (CPD) [17] and Tucker Decomposition [18] are the two most widely applied tensor decomposition methods. We only provide a brief review of CPD here, which is the target algorithm of this paper. The tensor in Canonical Polyadic Decomposition is expressed by the sum of rank-one tensors, as shown in Eq. (3) for three-dimensional tensors $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$. If the vectors $\mathbf{a}_r, \mathbf{b}_r$ and $\mathbf{c}_r$ in Eq. (3) are seen as the column vectors in objective dense factor matrices $\mathbf{A} \in \mathbb{R}^{I \times F}, \mathbf{B} \in \mathbb{R}^{J \times F}$ and $\mathbf{C} \in \mathbb{R}^{K \times F}$ respectively, where F denotes the rank of $\mathcal{X}$, CPD can also be illustrated in Eq. (4). In this paper, we implement five CPD optimization algorithms on Sunway architecture, including CPD-ALS, CPD-GD, CPD-RBS, CPD-GCP-SGD and CPD-fLM++.

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{a}_r \odot \mathbf{b}_r \odot \mathbf{c}_r \tag{3}$$

$$\mathcal{X} = \sum_{r=1}^{R} \mathbf{A}(:,r) \odot \mathbf{B}(:,r) \odot \mathbf{C}(:,r) \tag{4}$$

2.1.2. CPD-ALS algorithm

The Alternating Least Squares (ALS) is the workhorse for CPD [4], whose pseudo-code is shown in Algorithm 1. Its primary idea is to solve a least square problem for one factor matrix while keeping others fixed. Without losing generality, we only discuss the mode-1 operation as an example. Eq. (5) describes the least square problem for factor matrix $\mathbf{A}$ and Eq. (6) presents the optimal solution. The CPD-ALS algorithm is an iterative algorithm and during each iteration, the mode-1 factor matrix $\mathbf{A}$ is updated through Eq. (6), where the symbol $\star$ indicates pseudo-inverse operation.

$$\hat{\mathbf{A}} = \arg\min_{\hat{\mathbf{A}}} \left\| \mathcal{X}_{(1)} - \hat{\mathbf{A}}(\mathbf{C} \odot \mathbf{B})^T \right\| \tag{5}$$

$$\hat{\mathbf{A}} = \mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\star} \tag{6}$$

Algorithm 1. CPD-ALS algorithm

1: Input: $\mathcal{X}, Rank$
2: **while** max iterations unreached **or** not converged **do**
3: $\hat{\mathbf{A}} = \mathcal{X}_{(1)}(\mathbf{B} \odot \mathbf{C})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\star}$
4: $\hat{\mathbf{B}} = \mathcal{X}_{(2)}(\mathbf{A} \odot \mathbf{C})(\mathbf{C}^T\mathbf{C} * \mathbf{A}^T\mathbf{A})^{\star}$
5: $\hat{\mathbf{C}} = \mathcal{X}_{(3)}(\mathbf{A} \odot \mathbf{B})(\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})^{\star}$
6: Store column norms and normalize matrices to 1
7: **end while**

The sparse matricized tensor times Khatri-Rao product (MTTKRP) is identified as the main performance bottleneck of CPD-ALS and CPD-GD [19], which can be expressed as $\hat{\mathbf{A}} = \mathcal{X}_{(1)}(\mathbf{B} \odot \mathbf{C})$, given $\mathcal{X} \in \mathbb{R}^{I\times J\times K}, \mathbf{A} \in \mathbb{R}^{I\times F}, \mathbf{B} \in \mathbb{R}^{J\times F}$ and $\mathbf{C} \in \mathbb{R}^{K\times F}$. We compute the exact MTTKRP through tensor-vector products using Algorithm 2. Meanwhile, from the definition of MTTKRP and Algorithm 2, we can obtain that the computational time complexity of MTTKRP is $\mathcal{O}(NFnnz)$, where nnz is the number of nonzero elements in tensor $\mathcal{X}$.

Algorithm 2. MTTKRP algorithm

1: Input: $\mathcal{X} \in \mathbb{R}^{I\times J\times K}, Rank$ R
2: $\mathbf{A} \leftarrow 0$
3: **for** $i \leftarrow 0$ *to* I' **do**
4: **for** $j \leftarrow i_pointer[i]$ *to* $i_pointer[i+1]$ **do**
5: $s \leftarrow 0$
6: **for** $k \leftarrow j_pointer[j]$ *to* $j_pointer[j+1]$ **do**
7: **for** $r \leftarrow 0$ *to* R **do**
8: $s[r] += value[k] \cdot \mathbf{C}[k_index[k]][r]$
9: **end for**
10: **end for**
11: **for** $r \leftarrow 0$ *to* R **do**
12: $\mathbf{A}[i_index[i]][r] += s[r] \cdot \mathbf{B}[j_index[j]][r]$
13: **end for**
14: **end for**
15: **end for**

2.1.3. CPD-GD algorithm

The Gradient Descent (GD) [19] is another widely used algorithm for CPD in addition to ALS. Similar to Section 2.1.2, we take mode-1 operation as an example. The GD problem for factor matrix **A** can be expressed in Eq. (7). Next, the gradient of Eq. (7) can be presented as Eq. (8).

$$f = \frac{1}{2}\min\left\|\mathcal{X}_{(1)} - \mathbf{A}(\mathbf{C} \odot \mathbf{B})^T\right\|^2 \tag{7}$$

$$\frac{\partial}{\partial A}f = -\mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) + \mathbf{A}(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B}) \tag{8}$$

In GD, the gradient of **A** can be written as $\triangledown\mathbf{A} = vec(\frac{\partial}{\partial A}f)$, where $vec(.)$ operator denotes the matrix flattening. Then, we can compute the factor matric $\hat{\mathbf{A}} = \mathbf{A} - \alpha \triangledown \mathbf{A}$. The pseudo-code of the general CPD-GD algorithm is shown in Algorithm 3. For GD, the line search can be used to calculate the step size (Line **6**), among which the backtracking line search is a simple but effective method. It starts with a relatively large estimate of the step size for movement, and then iteratively shrinks the step size (i.e., backtracking) until a decrease of the objective function is observed with no more than the expected threshold.

Algorithm 3. CPD-GD algorithm

1: Input: $\mathcal{X}, Rank$
2: **while** max iterations unreached **or** not converged **do**
3: $\triangledown \mathbf{A} = -\mathcal{X}_{(1)}(\mathbf{C} \odot \mathbf{B}) + \mathbf{A}(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})$
4: $\triangledown \mathbf{B} = -\mathcal{X}_{(2)}(\mathbf{A} \odot \mathbf{C}) + \mathbf{B}(\mathbf{A}^T\mathbf{A} * \mathbf{C}^T\mathbf{C})$
5: $\triangledown \mathbf{C} = -\mathcal{X}_{(3)}(\mathbf{B} \odot \mathbf{A}) + \mathbf{C}(\mathbf{B}^T\mathbf{B} * \mathbf{A}^T\mathbf{A})$
6: $\alpha = linesearch(\mathbf{A}, \mathbf{B}, \mathbf{C}, \triangledown \mathbf{A}, \triangledown \mathbf{B}, \triangledown \mathbf{C})$
7: $\hat{\mathbf{A}} = \mathbf{A} - \alpha \triangledown \mathbf{A}$
8: $\hat{\mathbf{B}} = \mathbf{B} - \alpha \triangledown \mathbf{B}$
9: $\hat{\mathbf{C}} = \mathbf{C} - \alpha \triangledown \mathbf{C}$
10: **end while**

As mentioned in Section 2.1.2, the performance bottleneck of CPD-GD is still MTTKRP. The difference is that the line search may require multiple iterations to meet the Armoji condition. During each iteration, the MTTKRP needs to be performed, which brings significant computational overhead. Similar to CPD-ALS in Section 2.1.2, the computational time complexity of MTTKRP in CPD-GD is $\mathcal{O}(NFnnz)$, where nnz is the number of nonzero elements in tensor $\mathcal{X}$.

2.1.4. CPD-RBS algorithm

The Randomized Block Sampling (RBS) [20] is a combination of randomized Block Coordinate Descent (BCD) and Stochastic Gradient Descent (SGD). Same as previous sections, we take a 3rd-order tensor as an example. The high-level overview of the CPD-RBS algorithm is shown in Algorithm 4. A sampled block can be defined by the index sets $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$.

Algorithm 4. CPD-RBS algorithm

1: Input: $\mathcal{X}, Rank$
2: **while** max iterations unreached **or** not converged **do**
3: Randomly generate sample indices
4: **select** $\mathcal{B}_n$ **from** $\mathcal{I}_n = \{1, \ldots, I_n\}$ **where** $n = 1, 2, 3$
5: **Let** $\mathcal{X}_{sub} = \mathcal{X}(\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3)$
6: **Let** $\mathbf{A}_{sub} = \mathbf{A}_k(\mathcal{B}_1, :), \mathbf{B}_{sub} = \mathbf{B}_k(\mathcal{B}_2, :), \mathbf{C}_{sub} = \mathbf{C}_k(\mathcal{B}_3, :)$
7: $\mathbf{A}_{sub} = update(\mathcal{X}_{sub}, \{\mathbf{A}_{sub}\}, \alpha)$
8: $\mathbf{B}_{sub} = update(\mathcal{X}_{sub}, \{\mathbf{B}_{sub}\}, \alpha)$
9: $\mathbf{C}_{sub} = update(\mathcal{X}_{sub}, \{\mathbf{C}_{sub}\}, \alpha)$
10: **Set** $\mathbf{A}_{k+1} = \mathbf{A}_k$ **and** $\mathbf{A}_{k+1}(\mathcal{B}_1, :) = \mathbf{A}_{sub}$
11: **Set** $\mathbf{B}_{k+1} = \mathbf{B}_k$ **and** $\mathbf{B}_{k+1}(\mathcal{B}_1, :) = \mathbf{B}_{sub}$
12: **Set** $\mathbf{C}_{k+1} = \mathbf{C}_k$ **and** $\mathbf{C}_{k+1}(\mathcal{B}_1, :) = \mathbf{C}_{sub}$
13: $k = k + 1$
14: **end while**

As shown in Algorithm 4, there are no dependencies between different sample blocks. Thus, CPD-RBS can decompose the sample blocks in parallel. It is obvious that the block size plays an important role in the algorithm as it influences the convergence speed and the computation time. ALS can be used to compute the *update* process in Algorithm 4. The least square problem has a precise solution, which is not inconsistent with the concept of CPD-RBS. Therefore, CPD-RBS introduces the parameter α to control the step size. The updated factor matrix $\hat{\mathbf{A}}$ can be calculated via Eq. (9). When α is set to 1, the original ALS is used for the update.

$$\hat{\mathbf{A}} = (1 - \alpha)\mathbf{A} + \alpha\mathcal{X}_{(1)}(\mathbf{B} \odot \mathbf{C})(\mathbf{C}^T\mathbf{C} * \mathbf{B}^T\mathbf{B})^{\star} \tag{9}$$

The progress of the CPD-RBS algorithm can be divided into two phases: the search phase with unrestricted step size and the converge phase with restricted step size. In the search phase, the algorithm can converge to a neighborhood of a local optimum. In the converge phase, the accuracy of the solution is improved by a variance reduction strategy. For CPD-RBS, proving convergence to an optimum is still an open problem. However, according to the evaluation of [20], the CPD-RBS algorithm is capable of finding a good approximation. Meanwhile, the primary hotspot in CPD-RBS is the MTTKRP on sampled blocks, and the computational time complexity of the sampled MTTKRP is also $\mathcal{O}(NFnnz)$ due to the fixed sampling ratio, where nnz is the number of nonzero elements in tensor $\mathcal{X}$.

2.1.5. CPD-fLM++

CPD-fLM++ algorithm [21] is the optimized fast Levenberg–Marquardt (LM) algorithm [22] for CPD. Levenberg–Marquardt algorithm, or damped Gauss–Newton algorithm, is developed to alleviate the short-comings of the traditional Gauss–Newton algorithm [23], as the original Guass-Newton algorithm needs the initial point to be near the optimal solution and the Jacobian matrix to be strictly full-ranked. The Levenberg–Marquardt algorithm adds a proximal term in the update rule of Guass-Newton algorithm, as shown in Eq. (10), where $\mathbf{J}_n$ is the Jacobian matrix of the nth iteration point x_n and ϕ is the objective function. Meanwhile, the LM algorithm has super-linear convergence rate and can avoid the "swamp" effect. However, the traditional LM algorithm is inefficient in terms of computation complexity, which is $\mathcal{O}(T^3F^3)$ for CPD, where T is the summation of dimensions in tensor $\mathcal{X}$. To reduce the computation complexity, [21] leveraged the structures of Jacobian Gramian matrix in CPD. As a result, the computation complexity in CPD-fLM++ is reduced to $\mathcal{O}(NF^6)$ as described in Table 1, where N is the number of modes. The pseudo code of CPD-fLM++ is shown in Algorithm 5, where $\mathbf{D}$ is a commutation matrix, $\mathbf{A}_n$ denotes the nth factor matrix, $\otimes$ denotes the Kronecker product and ø denotes the Hadamard division.

$$x_{n+1} = x_n - ({\mathbf{J}_{(n)}}^T\mathbf{J}_n + \lambda_n\mathbf{I})^{-1}\mathbf{J}_n\phi(x_n) \tag{10}$$

Algorithm 5. CPD-fLM++ algorithm

1: Input: $\mathcal{X}$, Rank F
2: $\lambda \leftarrow 1$
3: **while** max iterations unreached **or** not converged **do**
4: $\mathbf{F} \leftarrow (N-1)\mathbf{D}_{F,F}$
5: $v \leftarrow 0$
6: **for** $n = 1, \ldots, N$ **do**
7: $\Gamma_n \leftarrow *_{i=1, i\neq n}^{N}\mathbf{A}_n$
8: $\mathbf{M}_n$ = Diag(vec(Γ_n))
9: $\mathbf{N}_n$ = Diag($\frac{1}{\mathrm{vec}\mathbf{A}_n^T\mathbf{A}_n}$)
10: $\mathbf{H}_{1n} \leftarrow \mathbf{A}_n\Gamma_n - \mathrm{MTTKRP}(\mathcal{X}, \{\mathbf{A}\}, n)(\Gamma_n + \lambda\mathbf{I})^{-1}$
11: $\mathbf{P}_n \leftarrow (\Gamma_n + \lambda\mathbf{I})^{-1} \otimes \mathbf{A}_n^T\mathbf{A}_n - \mathbf{M}_n^{-1}\mathbf{D}_{F,F}\mathbf{N}_n^{-1}$
12: $\mathbf{u}_n \leftarrow \mathbf{P}_n^{-1}\mathrm{vec}(\mathbf{A}_n^T\mathbf{H}_{1n})$
13: $\mathbf{H}_{2n} \leftarrow \mathbf{A}_n\mathrm{Mat}(\mathbf{u}_n)(\Gamma_n + \lambda\mathbf{I})^{-1}$
14: $\mathbf{v} \leftarrow \mathbf{v}+\mathrm{vec}(\mathbf{A}_n^T\mathbf{A}_n)*\mathbf{u}_n$
15: $\mathbf{F} \leftarrow \mathbf{F} + \mathbf{B}_n^{-1}\mathrm{Diag}(\mathbf{A}_n^T\mathbf{A}_n ø \Gamma_n)$
16: **end for**
17: $\mathbf{v} \leftarrow \mathbf{F}^{-1}\mathbf{v}$
18: **for** $n = 1, \ldots, N$ **do**
19: $\mathbf{u}_n \leftarrow \mathbf{P}_n^{-1}(\mathbf{v} ø \mathrm{vec}(\Gamma_n))$
20: $\mathbf{H}_{3n} \leftarrow \mathbf{A}_n\mathrm{Mat}(\mathbf{u}_n)(\Gamma_n + \lambda\mathbf{I})^{-1}$
21: $\tilde{\mathbf{A}_n} \leftarrow \mathbf{A}_n - \mathbf{H}_{1n} + \mathbf{H}_{2n} - \mathbf{H}_{3n}$
22: **end for**
23: **if** objective function increase **then**
24: $\lambda \leftarrow 2\lambda$
25: **else**
26: update the factor matrices
27: $\lambda \leftarrow \frac{\lambda}{2}$
28: **end if**
29: **end while**

Table 1
The time complexity of CPD with different optimization methods.

CPD optimization methods	Time complexity
CPD-ALS	$\mathcal{O}(NFnnz)$
CPD-GD	$\mathcal{O}(NFnnz)$
CPD-RBS	$\mathcal{O}(NFnnz)$
GCP-SGD	$\mathcal{O}(NFnnz)$
CPD-fLM++	$\mathcal{O}(NF^6)$

2.1.6. CPD-GCP-SGD algorithm

Generalized Canonical Polyadic Decomposition algorithm [24] is the generalization of the standard CPD algorithm, which contains more types of loss functions besides the standard square error, including logistic regression, gamma distribution, Rayleigh distribution, Poisson distribution, and negative binomial. The details of the above loss functions can be found in [24]. Given a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, Eq. (11) formulates the GCP algorithm, where $\mathcal{M}$ is the low rank approximation calculated using Eq. (4). Computing the gradient of the objective function is one of the main bottlenecks in GCP [24]. It also needs to compute the MTTKRP routine as shown in Eq. (12), where $\mathcal{Y}$ is the element-wise derivative that is only computed at known elements, and $\mathbf{Z_{BC}}$ is the Khatri-Rao product of factor matrices **B** and **C**. To alleviate GCP's computation overhead, SGD algorithm is adopted to update the factor matrices [25], whose pseudo-code is presented in Algorithm 6. Table 1 shows the time complexity of CPD-GCP-SGD.

$$\min F(\mathcal{M}, \mathcal{X}) = \sum_{i_1}^{I} \sum_{i_2}^{J} \sum_{i_3}^{K} f(x_{i_1 i_2 i_3}, m_{i_1 i_2 i_3}) \tag{11}$$

$$\frac{\partial F}{\partial \mathbf{A}} = \mathbf{Y}_{(k)} \mathbf{Z_{BC}} \tag{12}$$

Algorithm 6. CPD-GCP-SGD algorithm

1: Input: $\mathcal{X}$, Rank F
2: **while** max iterations unreached **or** not converged **do**
3: initialize tensor $\mathcal{Y}$
4: sample the elements
5: **for** each sampled elements **do**
6: update corresponding $\mathcal{Y}$
7: **end for**
8: **for** $n = 1, \ldots, N$ **do**
9: $\tilde{\mathbf{G}}_k \leftarrow$ MTTKRP($\mathcal{Y}, \mathbf{A}_k, k$)
10: **end for**
11: **end while**

2.2. CSF tensor storage format

Compressed Sparse Fiber (CSF) tensor storage format is first introduced by Smith and Karypis [16] and applied in SPLATT [26], which is one of the state-of-the-art parallel CPD libraries. We apply and modify CSF format in our implementation to reduce the data structure size and the amount of data movements during the computation of MTTKRP. We give a brief description of the basic and modified CSF format in the following paragraph. We use the three-dimensional tensor for illustration without losing generality.

There are five arrays in SPLATT [15], which store non-zero values in groups of mode-2 *fibers*. The names of the five arrays are are *i_pointer*, *k_pointer*, *k_index*, *j_index* and *value*. For each row *i* in the ascending order, its *i_pointer* is computed using Eq. (13), where *inumber* denotes the number of non-zero mode-2 *fibers* within the former row.

$$i_pointer[i] = \begin{cases} 0 & i = 0 \\ inumber + i_pointer[i-1] & i \neq 0 \end{cases} \tag{13}$$

The *i_pointer* is indexed by *k_index* and *k_pointer*. The *k_index* stores the mode-3 index of the mode-2 *fiber* (only one mode-1 and one mode-3 index are needed to locate a non-zero mode-2 *fiber*), with the corresponding *k_pointer* computed using Eq. (14), where *knumber* denotes the number of non-zero values within the former mode-2 *fiber*. The *k_pointer* is indexed by *j_index* and *value*. The *j_index* stores the mode-2 index of non-zero elements in the *fiber*, whereas the *value* stores the value of the elements.

$$k_pointer[k] = \begin{cases} 0 & k = 0 \\ knumber + k_pointer[k-1] & k \neq 0 \end{cases} \tag{14}$$

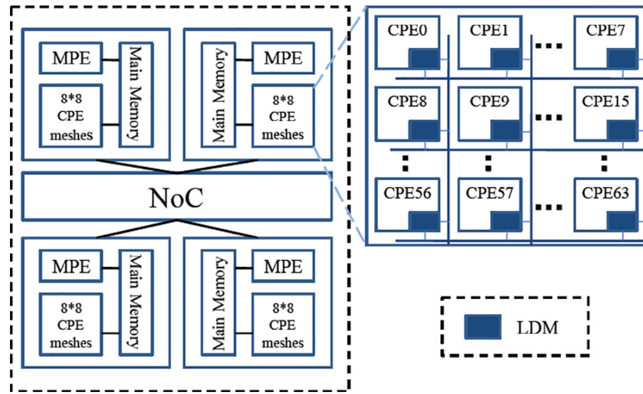

Fig. 1. The architecture of the Sunway SW26010 processor.

Given a three-mode tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ with F non-zero mode-2 *fibers* and *nnz* non-zero elements, the amount of memory occupied in CSF format is $16 + 8 \cdot I + 16 \cdot F + 16 \cdot nnz$ in bytes (indexes in uint64_t and values in double). We modify the CSF to further compress the tensor when there is few non-empty rows. We add an array *i_index* for storing the indexes of non-empty rows in addition to store the non-zero values in group of mode-3 *fibers*. Given the same tensor, if it has I' non-empty rows and F_1 non-zero mode-3 *fibers*, then the amount of memory occupied is $16 + 16 \cdot I' + 16 \cdot F_1 + 16 \cdot nnz$ in bytes (where $I' \ll I$).

2.3. Sunway many-core processor

Fig. 1 shows the architecture of Sunway SW26010 many-core processor. The processor is comprised of 4 core groups (CGs). There is a Management Processing Element (MPE) and 64 Computation Processing Elements (CPEs) within a CG. The MPE and CPEs within a CG share an 8 GB main memory. For each MPE, there is a 32 KB L1 data cache and 256 KB cache. For each CPE, there is a 16 KB L1 instruction cache and a 64 KB programmable Local Device Memory (LDM). The CPE can access main memory through either global load/store instructions (*gld/gst*) or DMA, of which DMA delivers much higher bandwidth when accessing continuous memory space. The Sunway processor also provides register communication between CPEs in the same row or column, which can send 128 bytes at a time and is of lower latency compared to the DMA. With all these unique architectural features, a CG can achieve peak memory bandwidth of 34 GB/s and peak floating-point performance of 756 GFLOPS in double precision [27].

2.4. Roofline model

The roofline model [28] offers insight on how to improve the performance of software and hardware. The roofline model builds up relationships among peak floating-point performance, operational intensity, and memory bandwidth. Therefore, it is quite illustrative to reveal the intrinsic characteristics of the application and provide guidance for performance optimization. According to [28], the upper limit of the operational intensity (I_{max}) can be calculated from Eq. (15):

$$I_{max} = Maximum\ Flops\ per\ Sec\ /\ Maximum\ Bandwidth \tag{15}$$

To better understand how effective our *swCPD* is when adapted to the Sunway architecture, we can build a roofline model of a Sunway CG using the approach similar to [29]. In this paper, we do not need to measure the ideal operational intensity of a Sunway CG, which has already been discussed in [29]. Instead, we only focus on whether *swCPD* exploits the computing potential of a Sunway CG as much as possible.

3. Methodology

In this section, we mainly present our hierarchical partitioning scheme for efficient exact and sampled MTTKRP implementations on Sunway architecture, which are the primary hotspots of CPD. The section describes the design overview, the tensor partitioning, multi-role assignment to CPEs, and the sampling scheme for CPD-RBS. In addition, we present the parallelization strategies for other significant matrix operations in CPD algorithms. Moreover, the auto-tuning mechanism for searching the optimal parameter configuration in CPD implementations is presented.

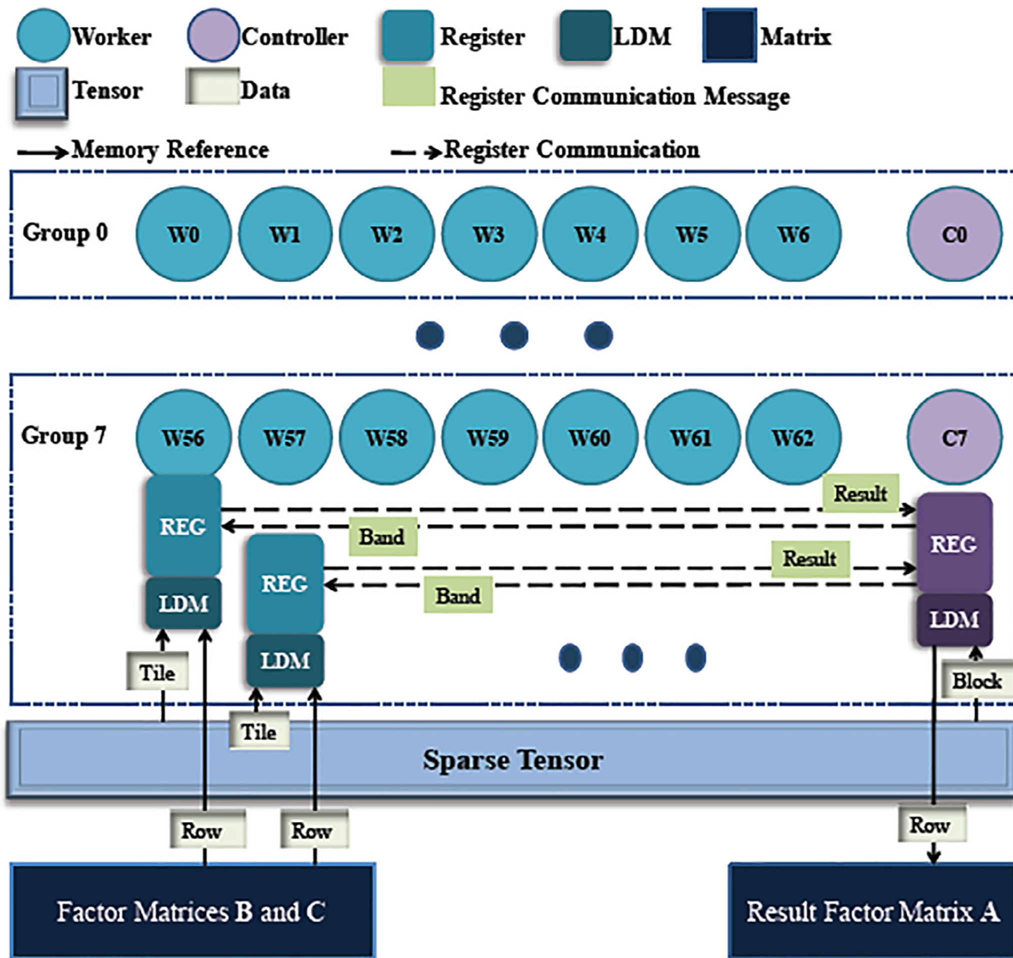

Fig. 2. The design overview of *swCPD*, which adopts hierarchical partitioning scheme for the computation of MTTKRP.

3.1. Exact MTTKRP

For CPD-ALS and CPD-GD, they share the common hotspot of exact MTTKRP. In this paper, the exact MTTKRP means the MTTKRP without randomization and sampling. We describe the optimization strategies for implementing the exact MTTKRP in this section.

3.1.1. Design overview

The main idea of our design is the hierarchical partitioning for both the sparse tensor and the hardware resources to alleviate the performance bottleneck of MTTKRP. As shown in Fig. 2, from the data perspective, the tensor is partitioned into three levels: *blocks*, *bands* and *tiles*. Meanwhile, from the computation perspective, the CPEs are also divided into three levels: *groups*, *controllers* and *workers*. Since the computation for all factor matrices is the same, we take the computation procedure for matrice **A** for illustration. As shown in Fig. 2, the *controllers* load the corresponding *block*, and then assign the *band* to the *workers* using specially designed message. The *controllers* are also in charge of storing the computed rows of **A** back to the main memory. Meanwhile, the *workers* load the *tiles* of the tensor as well as the rows from two factor matrices after receiving the *band* assigned by the *controller*. When the computation is done, they send the results back to the *controller*. The above procedure repeats within each CPE *group* until all the *bands* have been processed.

3.1.2. Partitioning the tensor

Given a sparse tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, we apply a three-level hierarchical blocking technique on it, as shown in Fig. 3. As the empty rows in the tensor can be removed trivially, we assume that there are no empty rows, and denote the number of non-empty rows as I. In the first level, we partition the tensor into eight *blocks*, and each block has $I'/8$ non-empty rows and is assigned to a CPE *group*. We partition the tensor along first dimension to avoid race condition as well as synchronization

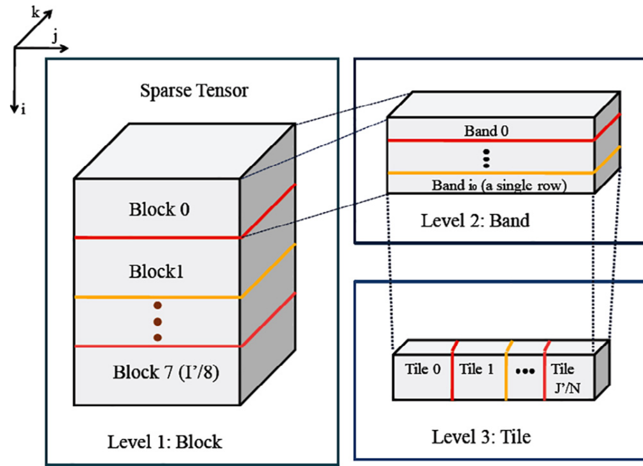

Fig. 3. Three-level tensor partitioning scheme.

between *controller* CPEs (refer to Section 3.1.3). For the second level, we further divide the *block* into *bands*, with each *band* containing a single mode-1 row. Moreover, in the third level, we divide each *band* into several *tiles*, which is small enough to fit in the limited LDM. A *tile* is set to contain at most N mode-3 *Fibers*, thus a *band* with J' mode-3 *Fibers* can be divided into J'/N *tiles*.

3.1.3. Assigning multiple roles to CPEs

Since the memory access from CPEs to main memory has a long latency, if the CPEs write the result back to the main memory whenever a nonzero value in the tensor is processed, the performance of the CPD deteriorates significantly. Thus, it is necessary to aggregate the computational results before writing back to the main memory in order to reduce the latency. Moreover, as described in Section 3.1.2, the *bands* that are computed by CPEs may not contain the same number of nonzero values. Therefore, if the number of *bands* is statically assigned to each CPE per iteration, there could be severe load imbalance among CPEs. Therefore, it is necessary to develop a dynamic allocation scheme for assigning *bands* among CPEs to alleviate the load imbalance. To address the above problems, we divide the 64 CPEs into eight *groups*, each *group* contains the CPEs in the same row. The CPEs within a *group* are further divided into 1 *controller* and 7 *workers* as shown in Fig. 2.

The **CPE *controller*** is in charge of the assigning *bands*, aggregating intermediate results and writing the results of the corresponding part of the factor matrix back to the main memory. There is a reference array in the main memory, through which the *controller* can recognize the number of *bands* in its *block*. Moreover, the *controller* keeps a *loadcounter* variable, which records the number of *bands* assigned to the *worker*.

To assign the *bands* to the *workers*, the *controller* takes advantage of the unique register communication between CPEs on Sunway and sends messages to the *workers* with the format shown in Fig. 4a. Once the *worker* finishes processing current *bands*, it sends an ending message to the *controller*. If the *loadcounter* is less than the number of *bands* in the *block*, the *controller* continues to assign the next *band* to the *worker* and *loadcounter* increases by 1. And if the *loadcounter* equals the number of *bands* in the *block*, the *controller* broadcasts an ending message to all its *workers* with the format shown in Fig. 4b.

i_pos	i_id	i_ptr[i_pos]	i_ptr[i_pos+1]

(a) *Band* assignment message format from *controller* to *worker*.

x	x	x	-1

(b) End processing message format from *controller* to *worker*.

i_pos	r	result1	result2

(c) Result message format from *controller* to *worker*.

i_pos	-1	column_id	i_id

(d) End processing message format from *controller* to *worker*.

Fig. 4. The format of messages.

We utilize register communication to aggregate and write the results back to the main memory. During computation, the *workers* send the results of *band* to *controller* in the message format shown in Fig. 4c, and the *controller* aggregates the intermediate results to a specific message buffer in the LDM. After the entire *band* finishes computation, the *worker* sends the *controller* an ending message in the format shown in Fig. 4d. And then, the *controller* writes the aggregated results that correspond to a row of **A** to the main memory through DMA. The communication and memory access of *controller* is shown in Fig. 2.

The **CPE *worker*** as shown in Fig. 2 conducts the computation of each *band* assigned by the *controller*. Once a *worker* loads the data of the i_0^{th} *band*, it performs the computation according to Algorithm 8. During the computation, a *worker* loads one *tile* of the *band* at a time and uses a buffer to store multiple rows of factor matrices **B** and **C** in order to reuse the data as much as possible. Besides, the *workers* exploit the vectorization units on Sunway to accelerate the computation. When the *worker* finishes the computation of MTTKRP, it sends the results of i_0^{th} row of factor matrix to *controller* through register communication with the message format shown in Fig. 4c. After that, the *controller* broadcasts an ending message to all *workers* with the message format shown in Fig. 4d. When receiving the ending message from the *controller*, the *worker* leases the buffer and exits the MTTKRP routine.

3.2. Sampled MTTKRP

For CPD-RBS, it needs the randomization and sampling strategy to reduce the computation of MTTKRP, which is denoted as sampled MTTKRP in this paper. Here we describe the implementation methodology of the sampled MTTKRP since it is the main challenge for CPD-RBS, which is also a modification of the exact MTTKRP.

As described in Section 2.1.4, one of the main differences between the CPD-RBS and the CPD-ALS is that the CPD-RBS algorithm needs to sample several randomized blocks of a tensor to compute the MTTKRP process to reduce the computation. However, in Sunway architecture, the global shuffling of all indices in a tensor proposed in the original CPD-RBS [20] is inefficient for two reasons. First, the CPEs only have 64 KB local memory, which is too limited to store all the shuffled indices. Second, the CPEs' discrete memory access to the main memory leads to high latency. Therefore, we propose a distributive sampling strategy for CPEs, which leverages the register communication between CPE *groups* and modifies the multiple role scheme of exact MTTKRP as described in Section 3.1. Thus each CPE only needs to store a shorter shuffled array to avoid the risk of LDM overflow and to reduce the potential memory access for acceleration. What's more, since we only deal with sparse tensor in this paper, we only consider sampling the nonzero elements in the tensor to avoid constant fit and divergence.

3.2.1. Group cooperation

As the description of Section 3.1.1, there are eight *groups* in 64 CPEs within a CG which operate *block* in the CSF tensor. To ensure every element is chosen at the same rate, we need to sample a random amount of *bands* in each *block* while the sum of the *bands* does not exceed the limitation denoted as *IBLOCK*. Thus we propose a cooperation scheme which utilizes the efficient register communication between the *groups*.

Before partitioning the tensors to the *groups*, the MPE shuffles an array which stores the indices of the *groups*, through which the *controller* in each *group* can recognize its order in *band* sampling. For the *group* that has the highest priority, its *controller* pickes out a random number betwwen 1 and its number of *bands*. Then it sends a message to the *controller* through the register communication whose group has the second highest priority if there are residual numbers for sampling *bands*. After receiving the messages, the corresponding *controller* will repeat the above process until there are no *bands* that need to be sampled. If the number of sampling *bands* is equivalent to zero, then the *controller* will just abort the MTTKRP routine. This cooperation scheme can assign the number of sampling *bands* to CPE *groups*, since the number of *bands* within the *blocks* are evenly partitioned, this process can ensure that every *bands* can be selected at the same rate.

3.2.2. The shuffling role for CPEs

As presented in Section 3.1.3, there are seven *workers* and a *controller* which are in the same row within a CPE *group*. After the number of **bands** that need to be sampled within the *group* is assigned by the cooperation scheme, except for the different duties of computing the MTTKRP between the *workers* and the *controller*, they need to be responsible for sampling in different dimensions of the tensor.

The **CPE *controller*** is responsible for shuffling the *bands* in its *block* and assigning the sampled *bands* to its *workers* dynamically. After the *controller* obtains information of indices in its *block* from the main memory, it shuffles the indices array which is much shorter than the array of all indices in the first dimension. If there are *rowlength bands* that need to be processed, then it select the first *rowlength* elements in the shuffled indices array and assign them to its *workers*.

The **CPE *worker*** is responsible for shuffling the *tiles* in its *band*. Since a *worker* may need multiple *tiles* to finish the updation for a *band*, the *worker* needs to firstly allocate the number of *fibers* that need to be sampled in a *tile*. Then it shuffles the indices of *fibers* in the loaded *tile*. Furthermore, for each *fiber*, the *worker* also shuffles its indices for the elements and samples *KBLOCK* elements for MTTKRP computation.

3.3. Matrix operations

Besides the MTTKRP routine, there are also matrix operations in the CPD algorithms as shown in Section 2.1.2, Section 2.1.3 and Section 2.1.4. The parallelization of the matrix operations is based on Basic Linear Algebra Subprogram(BLAS) [30], which has already been implemented on Sunway architecture. However, BLAS cannot be directly applied to perform some of the operations such as the Frobenius norm and the normalization. Thus we develop the subprograms for those matrix operations to achieve further acceleration.

3.3.1. Frobenius norm

Since the Frobenius norm of a vector is the sqaure root of the sum of absolute squares of all its elements, it's equivalent to the inner product of the vector itself as shown in Eq. (16). Hence we can leverage the inner product subprogram in the BLAS to help computing the Frobenius norm.

$$\|\mathbf{a}\| = \sqrt{\sum_i a_i} = \sqrt{\mathbf{a}\cdot\mathbf{a}} \tag{16}$$

3.3.2. Normalization

As shown in Algorithm 1, normalization process is needed after the matrices are updated. The normalization is performed after the norms such as Frobenius norm and infinite norm calculated. Since the normalization process needs to normalize all the elements in a vector, we utilize the *dscal* routine in BLAS, which can multiply the normalization factor to a vector in parallel after the factor is calculated.

4. Implementation

In this section, we present the implementation details of CPD on Sunway architecture, which include the processing logics of both *controller* and *worker* within the CPE *group*, as well as the cooperation among CPEs. We focus on elaborating on the implementation details of the exact MTTKRP. Since the sampled MTTKRP can be implemented on top of the exact MTTKRP, we only provide the implementation details of sampling for sampled MTTKRP. Besides, we also present the implementation details of several matrix operations.

4.1. Exact MTTKRP

4.1.1. Processing logic of the ccontroller CPE

The processing logic of the *controller* for MTTKRP is elaborated in Algorithm 7. At the beginning of MTTKRP, the *controller* identifies how many *bands* it needs to process based on the reference array *blockref*. Then the *controller* loads the *i_pointer* and *i_ids* of the corresponding *bands* from the sparse CSF tensor. Next, the *controller* sends messages to the *workers* through register communication. The *controller* records how many *bands* have been assigned through variable *loadcounter*. Once the *worker* generates a new row of factor matrice or finishes the computation of one *band*, it sends messages to notify the *controller*. If it is an ending message, and *loadcounter* does not reach the limit of *loadref*, the *controller* continues to assign a new band to the *worker*, which achieves good load balance among *workers*. Otherwise, if the message contains computation results, the *controller* updates the corresponding values of the factor matrix according to the message. When the buffer to store the factor matrix is full or all the *bands* have been processed, the *controller* writes the updated part of the factor matrix back to the main memory. After finishing the computation of the *block*, the *controller* sends an ending message to all its *workers*.

Algorithm 7. The processing logic of the *Controller*

1: Input: *sparse tensor* $\mathcal{X} \in \mathbb{R}^{I\times J\times K}$ *in CSF*, *Rank R*
2: $bl \leftarrow$ *block size*
3: $lr \leftarrow TOTAL_BUFFER_SIZE/R$
4: **if** $bl > 0$ **then**
5: $tt \leftarrow bl/lr + 1$
6: **if** $bl < lr$ **then**
7: $last \leftarrow bl$
8: **else**
9: $last \leftarrow bl - (tt - 1) * lr$
10: **end if**
11: **for** $m = 0 \rightarrow tt - 1$ **do**
12: $ls \leftarrow bb + m * lr$

13: **if** $m = tt - 1$ **then**
14: $length \leftarrow last$
15: **else**
16: $length \leftarrow lr$
17: **end if**
18: $idbuffer \leftarrow \mathcal{X}.i_id[bb : be]$
19: $ptrbuffer \leftarrow \mathcal{X}.i_ptr[bb : be + 1]$
20: $finishcounter \leftarrow 0$
21: $loadcounter \leftarrow 0$
22: *send computeInfo to workers*
23:/*computeInfo in format shown in Fig. 4a*/
24: **while** $finishcounter < loadlength$ **do**
25: $RegRecvRow(resultInfo)$
26: /*result in format shown in Fig. 4c or Fig. 4d*/
27: **if** *resultInfo is in format shown in* Fig. 4d **then**
28: $finishcounter \leftarrow finishcounter + 1$
29: **if** $loadcounter <= length - 1$ **then**
30: *assign a new band to the worker*
31: $loadcounter \leftarrow loadcounter + 1$
32: **end if**
33: *Write the* $Abuffer(result[0] - ls, :)$ *to main memory*
34: **else**
35: *UpdatetheAbufferelement*
36: **end if**
37: **end while**
38: **end for**
39: **end if**
40: *sendfinishInfotoallworkers*
41: /*finishInfo in format shown in Fig. 4b*/

4.1.2. Processing logic of the worker CPE

The processing logic of the *worker* is shown in Algorithm 8. To improve the data re-use of **B** and **C**, the *worker* preloads the first *BL* rows of **B** and the first *CL* rows of **C** into its LDM. After the *controller* assigns a *band* to the *worker*, the *worker* conducts the computation of MTTKRP as shown in Algorithm 2. Before the computation, it partitions the *bands* into *tiles* and loads a *tile* in the LDM each time. During the computation, it extracts the corresponding sub-arrays from *j_pointer* and *j_ids* in the CSF tensor based on *jl* and *jh*, which are then loaded into *myjptr* and *myjids* respectively resided in LDM. For every *j* in *myj_ids*, the *worker* extracts the the corresponding sub-arrays from the *k_ids* and *values* in the CSF tensor based on (*myjptr:j*), which are then loaded into *mykids* and *myvalues* resided in LDM. Moreover, only when the *j*th row of matrix **B** does not be in the *Bbuffer*, and its following *BL* rows are reloaded into the *Bbuffer*. The similar mechanism is also applied to *Cbuffer*.

Algorithm 8. The processing logic of the *Worker*

1: Input: *sparse tensor* $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$ *in CSF*, *Rank R*
2: *tilesizeN* /*a *tile* contains N Fibers*/
3: $Bbuffers \leftarrow \mathbf{B}(0 : BL - 1, :)$ /*preload factor matrices*/
4: $Cbuffers \leftarrow \mathbf{C}(0 : CL - 1, :)$
5: **while** $recvInfo[3] \geqslant 0$ **do**
6: $RegRecvRow(recvInfo)$
7: /*recvInfo in format shown in Fig. 4a or Fig. 4b*/
8: $i_position \leftarrow recvInfo[0]$
9: $i_id \leftarrow recvInfo[1]$
10: $j_ledge \leftarrow recvInfo[2]$
11: $j_hedge \leftarrow recvInfo[3]$
12: $tn \leftarrow (j_hedge - j_ledge)/N$ /*tn:tile number*/
13: **for** $t = 1 \rightarrow tn$ **do**
14: $jl \leftarrow j_ledge + t * N$

(*continued on next page*)

```
15: jh ← jl + N
16: get myjptr from 𝒳.j_pointer
17: get myjids from𝒳.j_ids
18: for every j in tile do
19: j_id ← myjids[j]
20: get mykids from 𝒳.k_ids
21: get myvalue from 𝒳.value
22: if B(j_id, :) not in Bbuffer then
23: Bbuffers ← B(j_id : j_id + BL − 1, :)
24: end if
25: for every k in the mode-3 Fiber do
26: k_id ← mykids[k]
27: if C(k_id, :) not in Cbuffer then
28: Cbuffers ← C(k_id : k_id + CL − 1, :)
29: end if
30:sum ← 0
31: for r = 0 → R − 1 do
32: tmp1 ← simd_multiply(myvalue[k] * myCbuffer[r])
33: sum[r] ← simd_add(sum[r], tmp1)
34: end for
35: end for
36: for r = 0 → nfactors − 1 do
37: tmp1 ← simd_multiply(sum[r] * myBbuffer[r])
38: Abuffer[r] ← simd_add(sum[r], tmp1)
39: end for
40: end for
41: end for
42: send resultInfo to controller
43: /*resultInfo in format shown in Fig. 4c*/
44: send endingInfo to controller
45: /*endingInfo in format shown in Fig. 4d*/
46: end while
```

4.1.3. Cooperation among CPEs

Meanwhile, to cooperate between *workers* and *controller* within a *group*, we design a data transfer mechanism through the register communication among CPEs. The register communication on Sunway can send a 128-bit message each time. In order to improve the communication efficiency during the computation of MTTKRP, we design four message formats as shown from Fig. 4a–d.

When assigning the *band* from the *controller* to the *workers*, the message format is shown in Fig. 4a, where *i_pos* denotes the index of the *band* in the arrays of *i_pointer* and *i_ids*. The *i_id* denotes the index of the *band* in the original tensor. The last two variables denote the index range of the mode-3 *Fibers* of the *band* in arrays of j_pointer and *i_ids*. When the computation is done, the *controller* sets the last variable in the message to -1, as shown in Fig. 4b, which explicitly notifies the *workers* to exit the MTTKRP routine.

Moreover, the *worker* sends the computation result for the row of the matrix **A** in message format shown in Fig. 4c. The first and second variables denote the buffer's location to store the result in the *controller*, and the last two variables are the generated elements from **A**. When the computation of the *band* is done, the *worker* sets the second variable in the message to -1, as shown in Fig. 4d. Besides, the *i_pos* indicates the row of *Abuffer* to be written back to main memory by the *controller*, whereas the *column_id* indicates the next *band* to be assigned and *i_id* indicates the location where the data is written to in main memory.

4.2. Sampled MTTKRP

4.2.1. Group cooperation

For sampling the *bands* among the CPE *groups*, once a *controller* in the *group* with highest priority selects its number of *bands* to be sampled, it sends a message to inform the *controller* in the *group* with the second highest priority to continue sampling with the message format shown in Fig. 5. The *row_id* is the indice of the sender's row, and the *g_order* is the *group*'s priority in sampling which is from 1 to 7 in one CG. Besides, the *n_res* denotes the number residual *bands* that needs to be sampled. Moreover, the *f_flag* indicates that the sampling process finishes if it is negative, and vice versa.

Fig. 5. The message format of sampling between groups.

4.2.2. Sampling logic of worker CPE

The processing logic of sampling in the *workers* is shown in Algorithm 9. Here we only focus on the sampling strategy in the *workers* since the process of sampling the *bands* is similar to that for the *tiles*. After loading a *tile* in its LDM, if the sampling does not finish as *loadref_r* does not equal to zero, then the *worker* will select a random number between 1 and *loadref_r* to decide how many *fibers* in this *tile* will be sampled. However, if the *tile* is the last tile to be loaded, the *loadref_t* will be set equivalent to *loadref_r*. For the sampled *fiber*, the *worker* will further shuffle the *refarray_k* whose elements are from 0 to the number of nonzero in *j_id*th *fiber*. As we set the sampling size in the third dimension to be *KBLOCK*, the *worker* will sample *KBLOCK* elements for computation.

Algorithm 9. The Sampling Logic of the *Worker* for a *Tile*

1: Input: A Tile T
2: **if** loadref_r $\neq$ 0 **then**
3: **if** loadtime $\neq$ max_trans **then**
4: loadref_t = random(1, loadref_r)
5: **else**
6: loadref_t = loadref_r
7: **end if**
8: loadref_r = loadref_r - loadref_t
9: refarray_j = shuffle(T, length(T))
10: **for** i = 1 $\rightarrow$loadref_t **do**
11: j_id = myjids[refarray_j[i]]
12: refarray_k = shuffle(j_id, length(j_id))
13: **for** k = 1 $\rightarrow$ *KBLOCK* **do**
14: k_id = mykids[refarray_k[k]]
15: **end for**
16: **end for**
17: **end if**

4.3. Matrix operations

4.3.1. Frobenius norm and normalization

The processing logic of computing Frobenius norm and the following normalization for the column vectors in the factor matrices are shown in Algorithm 10, where the *ddot* routine is for the inner product between two vectors in parallel, while the *dscal* is for scalar multiplication.

Algorithm 10. Frobenius norm and normalization

1: Input: A factor matrix $\mathbf{A}$
2: **for** every column vector $\mathbf{a}_i$ in $\mathbf{A}$ **do**
3: lambda[i] = cblas_ddot($\mathbf{a}_i$,$\mathbf{a}_i$)
4: lambda[i] = $\sqrt{\text{lambda[i]}}$
5: cblas_dscal(1/lambda[i], $\mathbf{a}_i$)
6: **end for**

4.3.2. Infinite norm and normalization

The processing logic of computing the infinite norm as well as the following normalization process is described in Algorithm 11. If the normalization factor *lambda* for a vector is not larger than 1, the original value will be replaced by 1.

Algorithm 11. Infinite norm and normalization

1: Input: A factor matrix **A**
2: **for** every column vector $\mathbf{a}_i$ in **A** **do**
3: lambda[i] = cblas_ddot($\mathbf{a}_i$,$\mathbf{a}_i$)
4: lambda[i] = max(1, lambda[i])
5: cblas_dscal(1/lambda[i], $\mathbf{a}_i$)
6: **end for**

5. Performance auto-tuning for *swCPD*

5.1. Identifying the parameters

For *swCPD*, there are several parameters that could directly affect the performance of MTTKRP, including *CALC_CORE_NUMBER*, *BL*, *CL*, *TILE_SIZE*, *NNZ_NUMBER* and *TOTAL_BUFFER_SIZE*. The parameters control the adaptability of MTTKRP to the Sunway architecture, including LDM usage, DMA transfer, CPE parallelism, and register communication. *CALC_CORE_NUMBER* is the number of busy CPEs and controls the tradeoff between delay of register communication and CPE utilization. Since the size of LDM is quite limited, parameters other than *CALC_CORE_NUMBER* are used to control the amount of data fetched to LDM each time. *TOTAL_BUFFER_SIZE* is used to set the LDM memory space reserved for MTTKRP. *BL* is the buffer size of factor matrix **B** and *CL* is the buffer size of factor matrix **C**. *TILE_SIZE* is the buffer size of *Fiber*. *NNZ_NUMBER* is the number of non-zero elements. In addition, these parameters do not affect the convergence of *swCPD* so that they can be applied to all three algorithms in this paper.

As described above, it is prohibitive to obtain the optimal parameter settings through exhaustive search, which is impractical in reality. Therefore, we propose an auto-tuning scheme to identify the optimal parameter setting of *swCPD*. Firstly, we need to determine the search space for each parameter. The valid range and constraint of each parameter is shown as follows:
- $CALC_CORE_NUMBER <= 7$, and $CALC_CORE_NUMBER \in N$.
- $TOTAL_BUFFER_SIZE < LDM_{size}$, and $TOTAL_BUFFER_SIZE \in N$.
- $(NNZ_NUMBER + TILE_SIZE) \times 2 + (BL + CL + 2) \times nfactors < TOTAL_BUFFER_SIZE$, and $NNZ_NUMBER, TILE_SIZE, BL, CL \in N$.

5.2. Auto-tuning using genetic search

Based on the above analysis, we use genetic search [31] to determine the optimal settings of the parameters. To the best of our knowledge, *swCPD* is the first CPD library that adopts the auto-tuning approach, such as genetic search, to automatically determine the optimal parameters, whereas the existing cutting-edge libraries [19,26] setup the parameters empirically. We convert the value of each tuning parameter to a binary string named a gene. Multiple genes constitute an individual, which represents a set of parameters. The length of the individual is based on the range requirements (Section 5.1). Fitness is an indicator used to evaluate an individual. Many individuals constitute a population, which can be split into several sub-populations. We use MPI communication to implement the migration among sub-populations. The pipeline of multi-process genetic search is shown in Fig. 6. We choose the single-ring topology for the migration: each subpopulation exchanges individuals with its two neighborhoods.

New individuals in the sub-population are bred through two parents uniform cross-over and mutation. As shown in Fig. 7, the sub-population genetic algorithm consists of three steps: *1)* each parent is selected from its four neighborhoods based on

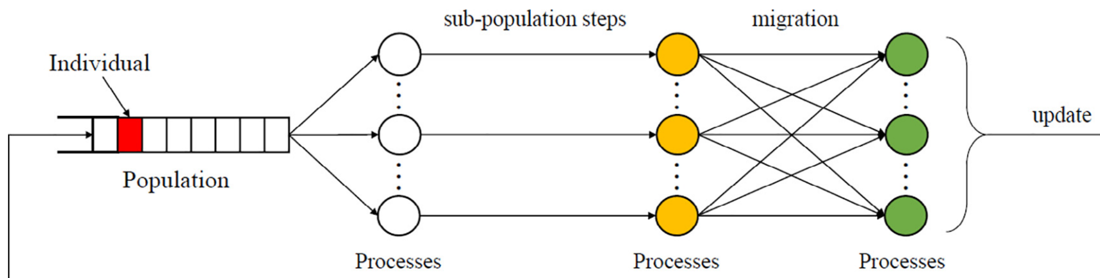

Fig. 6. Multi-process auto-tuning pipeline for *swCPD*.

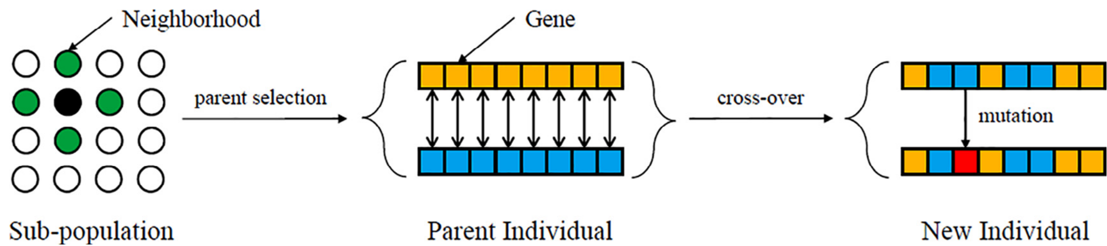

Fig. 7. Sub-population genetic algorithm.

their fitness, and the higher the fitness the higher the selection chance; *2)* each gene in the individual is randomly selected from the parents; *3)* the new individuals are selected based on a low probability to undergo mutation. This probability is called the mutation rate, and it controls the exploration versus exploitation tradeoff [32]. In genetic search, the mutation is used to prevent the individuals from falling into local optimum. The genetic search terminates when *1)* the maximum number of iterations is reached or, *2)* the fitness reaches the threshold. When the fitness converges, the optimal parameter settings are found.

6. Evaluation

6.1. Experimental setup

We conduct our experiments on a CG of Sunway SW26010 processor. To evaluate the performance of our *swCPD*, we use synthesized datasets with tensor data generated similar to [33], in addition to three real-world datasets: MovielensNew, YELP [34] and BookCroosing (BX) [35]. The datasets of Movielens are derived from the data of movie ratings provided by [36]. The detailed specifications of the datasets are shown in Table 2. We report the average execution time of a single iteration. All experiments are conducted in double precision. All the initial values in the factor matrices are randomly generated. Although the initial values may affect the number of iterations for the CPD algorithms, we report the performance in terms of the average execution time of a single iteration, which will not be affected by the number of iterations and the initial values.

6.2. Evaluation criteria

6.2.1. CPD-ALS

We compare the performance of our *swCPD-ALS* algorithm with the CPD-ALS routine in the widely used MATLAB Tensor Toolbox [6] and SPLATT [26], which is one of the cutting-edge parallel tensor decomposition libraries and is based on CPD-ALS. We modify SPLATT so that it can utilize the 64 CPEs within a CG, which is denoted as SW-SPLATT-ALS. Compared to the *swCPD-ALS*, the SW-SPLATT-ALS does not leverage the efficient DMA and register communication, as well as the hierarchical blocking and role assignment strategies. The platform for MATLAB Tensor Toolbox is the Intel Xeon E5620 CPU with 8 GB memory, which is the same as that in a SW26010 CG. We use the performance of Matlab Tensor Toolbox as the baseline. We further test the sensitivity of significant parameters for *swCPD-ALS*.

6.2.2. CPD-GD

We compare the performance of our *swCPD-GD* algorithm with the CP-OPT routine in MATLAB Tensor Toolbox [5] which is also based on gradient descent. CP-OPT is implemented using Poblano Toolbox in MATLAB [37], which is commonly used for gradient-based optimization. The MATLAB Tensor Toolbox stores the tensor in COO format for the sparse tensors, which has much larger memory footprint than CSF. For CP-OPT, the initial factor matrices are generated by the *create_guess* routine.

Table 2
The sparse tensor datasets.

Dataset	I	J	K	nnz
Dataset1	1 K	1 K	1 K	33 K
Dataset2	4.1 K	4.1 K	4.1 K	262 K
Dataset3	4.9 K	4.9 K	4.9 K	343 K
Dataset4	6.4 K	6.4 K	6.4 K	512 K
Dataset5	1 K	1.6 K	4 K	80 K
Dataset6	1 K	1.6 K	1.6 K	40 K
MovielensNew	610	9.7 K	1.8 K	101 K
YELP	70 K	15 K	108	334 K
BookCrossing (BX)	278 K	271 K	102 K	383 K

In addition, we modify the ALS algorithm in SPLATT to the GD algorithm based on [19] in order to run on a CG of Sunway architecture. The modified GD algorithm in SPLATT is named SW-SPLATT-GD. The platform for MATLAB Tensor Toolbox is Intel Xeon E5620 CPU with 8 GB memory. We use the performance of SW-SPLATT-GD as the baseline since the CPD-OPT fails to run when the tensor is extremely sparse or the rank is too large.

6.2.3. CPD-RBS

We compare the performance of our *swCPD-RBS* algorithm with the CPD-RBS algorithm and SW-RBS. The SW-RBS is the RBS implementation in Tensorlab [38] ported to Sunway architecture by utilizing the *gld/gst* for memory access and adopting the global shuffle and sampling strategy described in [20]. We use the performance of SW-RBS as the baseline. The sampling size is set to the same for *swCPD-RBS* and SW-RBS, which varies under different settings of rank.

6.2.4. CPD-fLM++

We compare the performance of our *swCPD-fLM++* algorithm with the SW-fLM++ algorithm. The SW-fLM++ is the fLM++ [21] implementation on Sunway processor that utilizes the *gld/gst* for memory access. We use the performance of *SW-fLM++* as the baseline.

6.2.5. CPD-GCP-SGD

We compare the performance of our *swCPD-GCPSGD* algorithm with the GCP-SGD routine in MATLAB Tensor Toolbox [25] and SW-GCPSGD algorithm. SW-GCPSGD is the CPD-GCP-SGD implementation in MATLAB Tensor Toolbox ported to Sunway processor by utilizing the *gld/gst* for memory access and adopting the loss functions described in [24]. We use the performance of GCP-SGD as the baseline. We choose five loss functions in the comparison, including standard square error (standard CPD), logistic regression, gamma distribution, Rayleigh distribution and Poisson distribution.

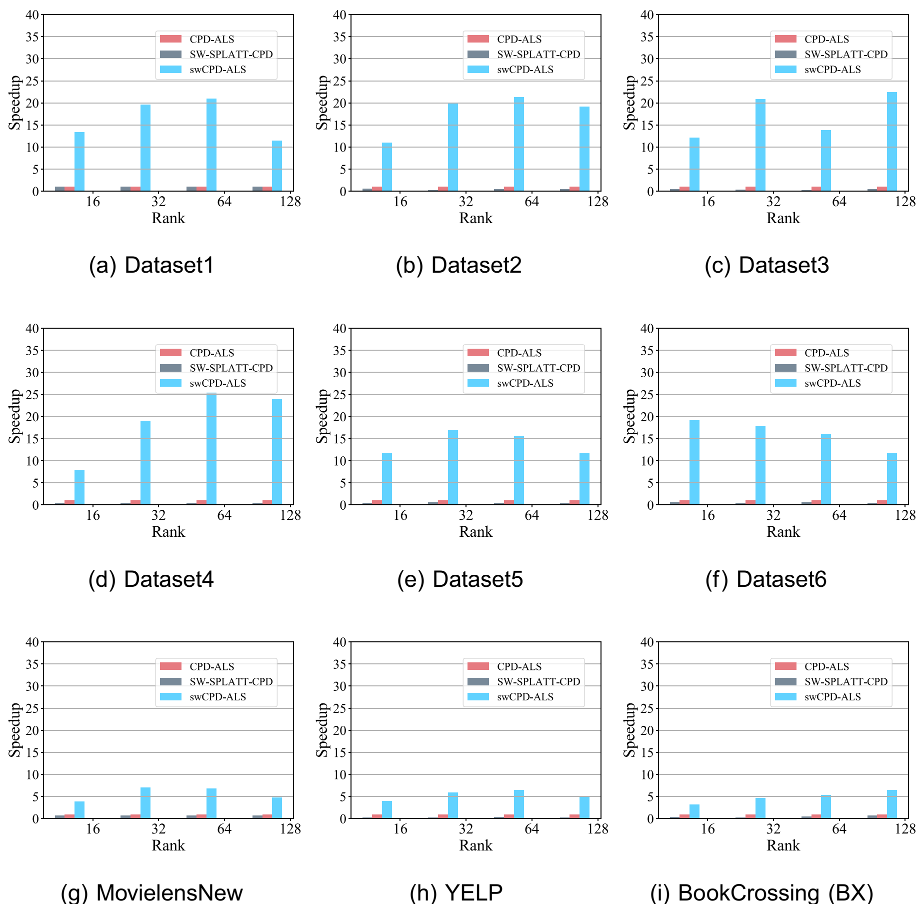

Fig. 8. Performance comparison between CP-ALS in MATLAB Tensor Toolbox, SW-SPLATT-ALS and *swCPD-ALS* on Sunway SW26010 processor with the rank of 16, 32, 64 and 128.

6.3. Performance analysis

6.3.1. CPD-ALS

The performance comparison between MATLAB Tensor Toolbox, SW-SPLATT-ALS and *swCPD-ALS* is shown in Fig. 8. The performance of SW-SPLATT and *swCPD-ALS* is normalized to the baseline. It is clear that *swCPD* achieves the best performance under all datasets across different ranks. Our *swCPD-ALS* improves the performance of CPD by 12.97× on average, with the maximum speedup of 25.5× with Dataset4. Moreover, we observe the performance of SW-SPLATT-ALS is worse than the baseline. The reason is that SW-SPLATT relies on the global load/store (*gld/gst*) instructions on CPEs to access memory, which leads to long memory access latency and thus significantly deteriorates the performance of CPD. We also notice that *swCPD-ALS* achieves less speedup on the two real-world datasets. This is due to the extreme sparsity of these two datasets. Even though the CPEs can buffer contiguous rows of **B** and **C** in LDM, the elements within a *band* are hardly continuous and thus lead to frequent cache misses, which in turn deteriorates the speedup of *swCPD-ALS*.

6.3.2. CPD-GD

Fig. 9 presents the performance comparison results between SW-SPLATT-GD, CP-OPT in MATLAB Tensor Toolbox and *swCPD-GD*. The performance of CP-OPT and *swCPD-GD* is normalized to the baseline SW-SPLATT-GD. If the CP-OPT crashes because of memory overflow or large rank, its performance is denoted as zero. From Fig. 9, it is obvious that the *swCPD-GD* obtains the best performance among those implementations across all the datasets. The highest acceleration rate is 37.21× achieved on Dataset4, with the average acceleration rate of 16.27×. Similar to *swCPD-ALS*, *swCPD-GD* achieves lower acceleration rate in the real-world datasets for its sparsity. However, compared to *swCPD-ALS*, the *swCPD-GD* is more time-consuming since it requires several iterations of the linear search.

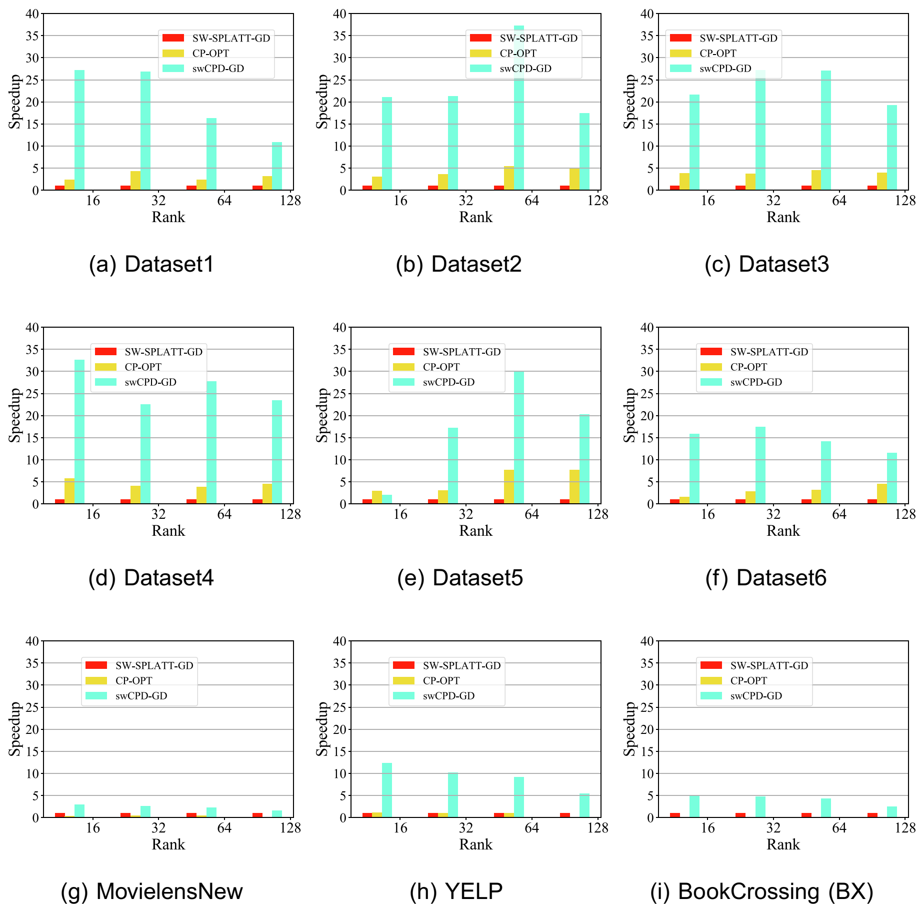

Fig. 9. Performance comparison between CP-OPT in MATLAB Tensor Toolbox, SW-SPLATT-GD and *swCPD-GD* on Sunway SW26010 processor with the rank of 16, 32, 64 and 128. The missing bar indicates it fails to run with CP-OPT in MATLAB Tensor Toolbox.

6.3.3. CPD-RBS

The comparison results between SW-RBS and *swCPD-RBS* are shown in Fig. 10. We can draw the conclusion that *swCPD-RBS*'s performance exceeds that of SW-RBS across all the datasets. From Dataset4, the highest acceleration rate emerges, which is at 37.44×. Moreover, the average acceleration rate is 7.81×. When compared with the former two optimized algorithms in *swCPD*, it can be concluded that the *swCPD-RBS* achieves the least acceleration among most of the datasets since it has the least requirements for computation.

6.3.4. CPD-fLM++

The performance comparison between *swCPD-fLM++* and SW-fLM++ is shown in Fig. 11. Note that when the rank grows to 128 or the tensor is relatively large, the algorithms encounter memory overflow since computing CPD-fLM++ requires eleven intermediate matrices in $F^2 \times F^2$ as shown in Algorithm 5. For example, when the rank is 128, the size of the intermediate matrices will be 2 GB each, whose total memory footprint exceeds the memory capacity of Sunway processor. It is obvious that *swCPD-fLM++* achieves better performance than SW-fLM++ across all datasets, with the highest speedup of 39.57× on Dataset4 under rank 64. Moreover, the average speedup of *swCPD-fLM++* is 22.77×. The speedup is due to that *swCPD-fLM++* can significantly reduce the computation overhead of CPD-fLM++, and thus obtain better performance.

6.3.5. CPD-GCP-SGD

The results of the performance comparison among *swCPD-GCPSGD*, GCP-SGD and SW-GCPSGD with different loss functions are shown from Figs. 12–16. It is clear that *swCPD-GCPSGD* achieves the best performance among most of the datasets, with the highest speedup of 29.92× on Dataset4 under loss function following Poisson distribution. In addition, the speedup of *swCPD-GCPSGD* on the same dataset under different loss functions is similar because the computation cost of different loss functions on each element of the dataset is nearly the same.

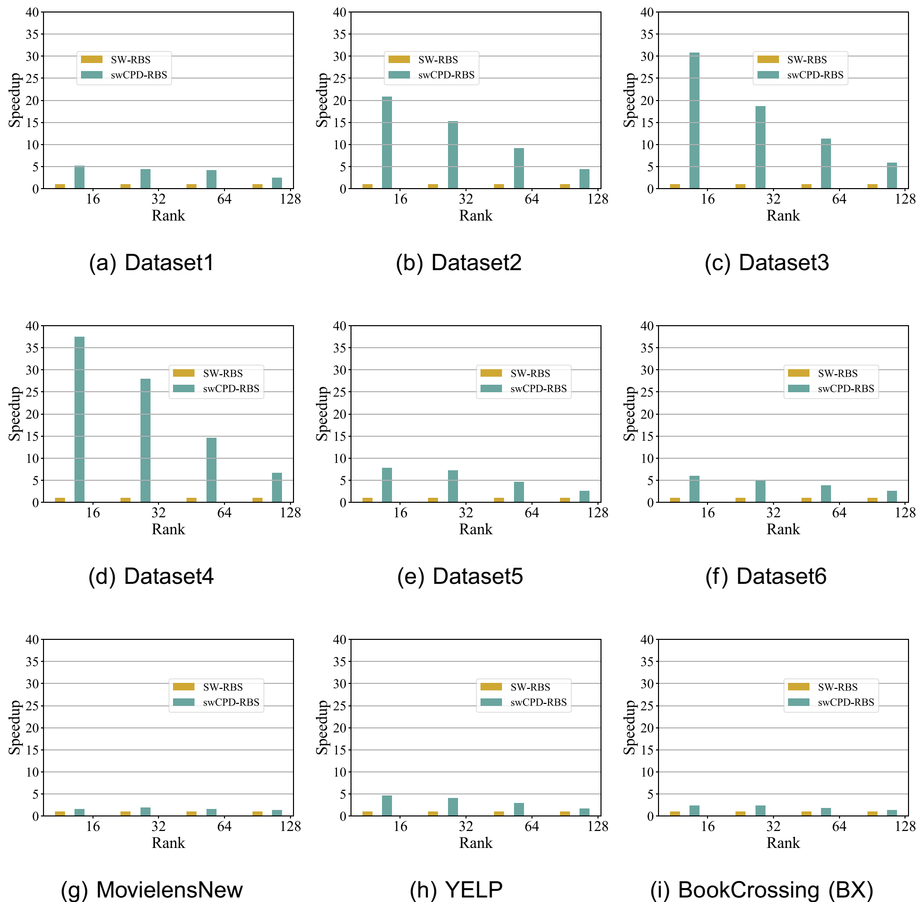

Fig. 10. Performance comparison between SW-RBS and *swCPD-RBS* on Sunway SW26010 processor with the rank of 16, 32, 64 and 128.

6.4. Roofline analysis

To further analyze our implementation's efficiency, we apply the roofline model [28] to *swCPD* on a CG of Sunway processor. We focus on the theoretical analysis for exact MTTKRP in the roofline analysis. Given a tensor $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3}$ of rank R with I, F and nnz denoting the number of non-zero rows, the number of non-zero *Fibers* and the number of non-zero values, respectively. The tensor $\mathcal{X}$ is stored in CSF format. Let Q, W, and I represent the amount of data accessed from memory, the number of floating-point operations, and the arithmetic intensity respectively. The calculation of Q, W and I is shown in Eq. (17), Eq. (18) and Eq. (19), respectively.

$$Q = (2 + R)(F + nnz) \tag{17}$$

$$W = 2R(F + nnz) \tag{18}$$

$$I = \frac{W}{Q \cdot 8bytes} = \frac{R}{8 + 4R} \tag{19}$$

From Eq. (19), it is clear that the arithmetic intensity of CPD is less than 1. Based on the roofline analysis in [29], the ridge point of the Sunway processor is 8.46. Therefore, the CPD is severely memory-bound. As a result, the optimizations adopted in our *swCPD* are effective in addressing the above bottleneck. Take Dataset4 for example, this dataset contains 262,144 non-zero elements and *fibers*. When the rank is set to 128, without our proposed optimizations, the performance of CPD is 0.012 GFLOPS, and the arithmetic intensity is 0.246 FLOPS/BYTE. After applying the hierarchical partitioning scheme, the arithmetic intensity of CPD improves to 0.46 FLOPS/BYTE, which indicates our approach is quite effective in alleviating the performance bottleneck of memory access.

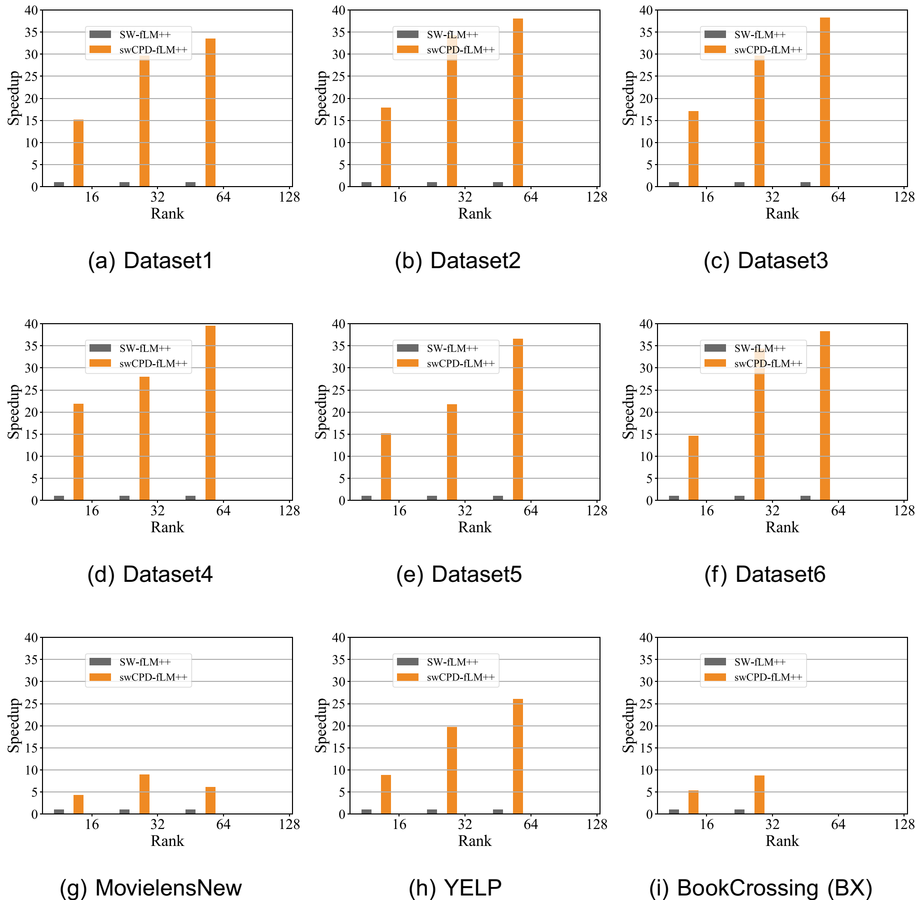

Fig. 11. Performance comparison between *swCPD-fLM++* and SW-fLM++ on Sunway processor with the rank of 16, 32, 64 and 128.

6.5. Parameter sensitivity analysis

To understand the performance impact of the parameter settings, we study three parameters in the exact *MTTKRP* that has a direct impact on the performance of CPD-ALS, including tile size *N*, buffer size *BL* of factor matrice **B** and buffer size *CL* of factor matrice **C** in LDM. Regarding *BL* and *CL*, when they are set too large, the DMA transaction time increases. However, when they are set too small, the buffer hit rate decreases significantly and thus causes more DMA transactions. Due to the compound performance impact of *BL* and *CL* on CPD is more complex, we measure the performance of *swCPD* under different settings of *BL* and *CL*. Fig. 17 presents the performance results under all datasets with $rank = 16$. It is clear that when the setting of *BL* and *CL* is too large, the performance deteriorates. We also notice that when the setting of *BL* and *CL* is moderate, the performance is more sensitive to *BL* than *CL*. In our implementation, we set *BL* to 4 and *CL* to 4, which delivers the best performance across all datasets.

6.6. Performance auto-tuning

In addition to Section 6.5, we expand the number of tuning parameters to 6 (Section 5.2). Here we take *swCPD-ALS* for an example and set the mutation rate to 0.01. Other algorithm implementations of *swCPD* exhibit a similar tendency when using the performance auto-tuning scheme. As shown in Fig. 18, three datasets (Dateset3, Dataset4, and YELP) are selected to evaluate the efficiency of genetic search. There are 64 individuals in a sub-population, and sub-population number of 2, 4, 8, and 16 are evaluated. The results are recorded every four generations.

In Fig. 18, as the generation increases, the performance of *swCPD-ALS* gradually converges to the optimal, which means the genetic search has found the optimal setting of the parameters. Furthermore, the larger the total number of individuals is, the faster the convergence speed is. Besides, when the population size is too small, the genetic search may fall into a local optimum (e.g., 2-Processes in Dataset4). The highly irregular structure of YELP constrains the parameter search space on Sunway architecture, which is faster to converge than Dataset3 and Dataset4.

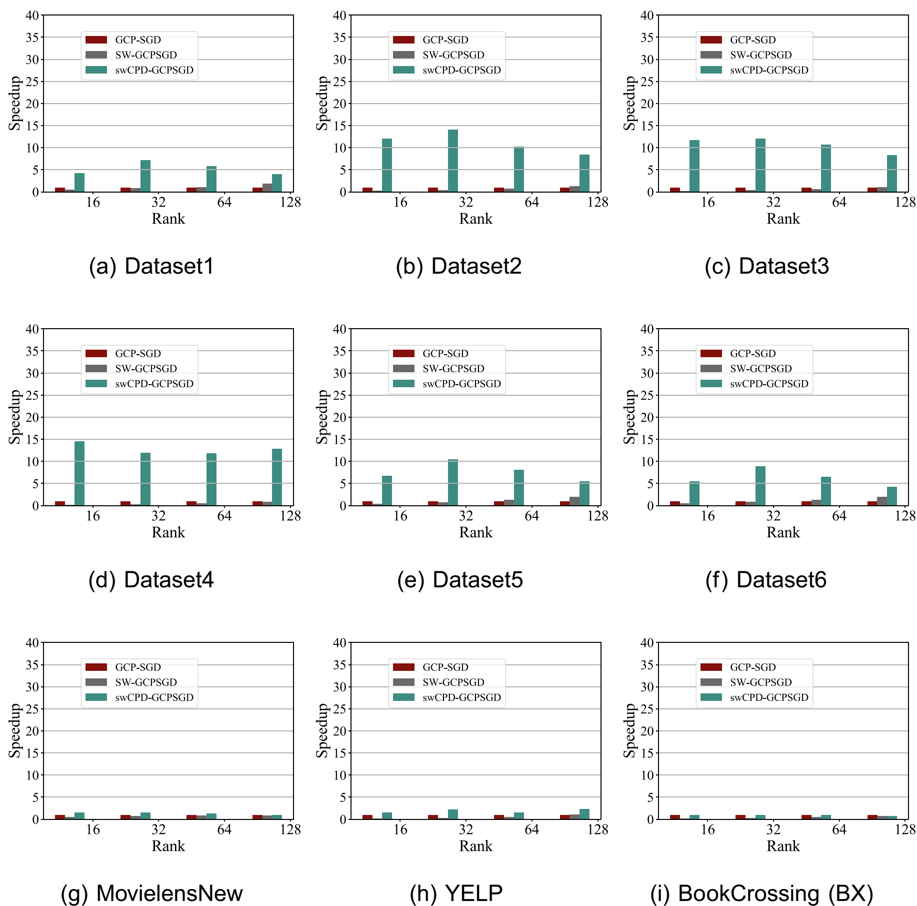

Fig. 12. Performance comparison among *swCPD-GCPSGD*, CPD-GCP-SGD and SW-GCPSGD on Sunway processor with the rank of 16, 32, 64 and 128. The loss function is standard square error (standard CPD).

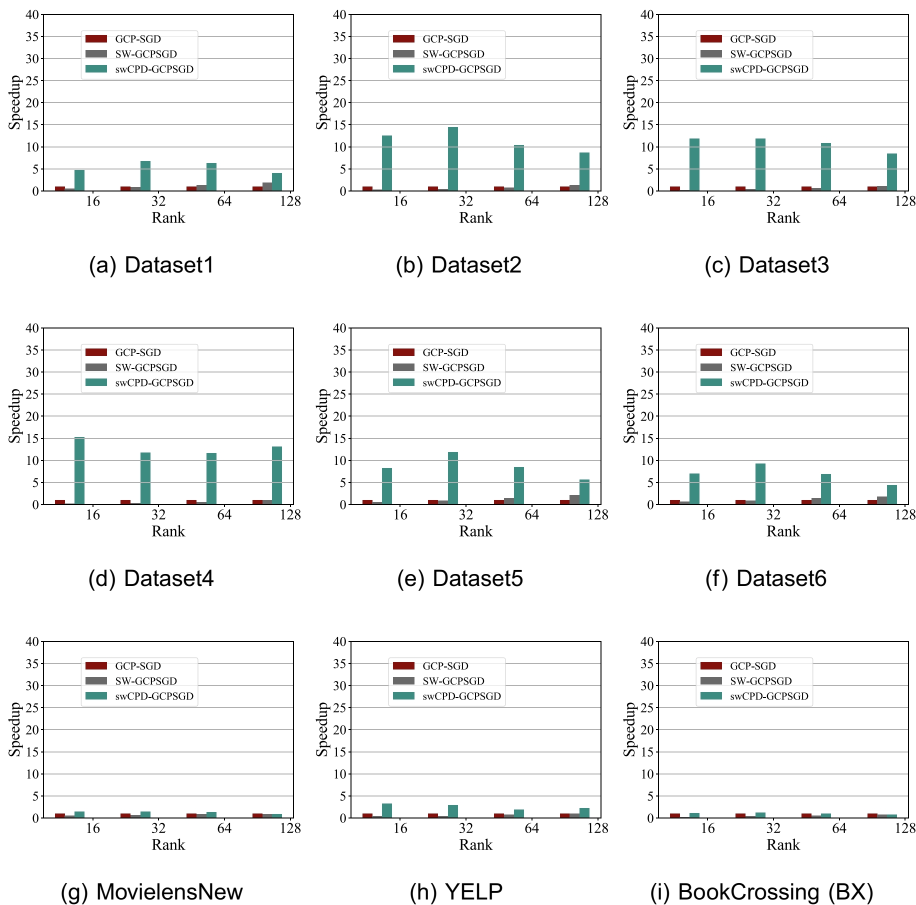

Fig. 13. Performance comparison among *swCPD-GCPSGD*, GCP-SGD and SW-GCPSGD on Sunway processor with the rank of 16, 32, 64 and 128. The loss function is logistic regression.

7. Related work

7.1. Tensor decomposition

There are plenty of researches about sparse-and-large tensor decomposition on architectures such as CPU and GPU. *Gigatensor* [8] developed the CPD algorithm on CPU based on MapReduce [7] paradigm. Moreover, *Gigatensor* applied Hadamard product in MTTKRP to avoid intermediate data construction. Smith *et al.* [26] proposed *SPLATT* on CPUs. This algorithm computes CPD-ALS efficiently through data tilting and CSR-like tensor reordering. *ParTI!* [9] supported essential sparse tensor operations and CPD on multicore CPU and GPU.

Furthermore, as MTTKRP is the primary hotspot in CPD algorithm [15], there are many efforts towards accelerating MTTKRP. Choi and Vishwanathan [19] introduced an efficient MTTKRP algorithm named *DFacto* that reformulates MTTKRP to a sequence of sparse matrix–vector multiplications (SpMVs). Based on SPLATT [26], Smith, Choi *et al.* [15] utilized fine-grained blocking techniques named rank blocking and multi-dimensional blocking for tensors and matrixes to obtain a better acceleration rate of MTTKRP than SPLATT. Besides, HyperTensor [39] also applied a fine-grained partition scheme that divides the nonzero elements separately. Cheng et al. [40] implemented a novel CPD algorithm based on orthogonality structure in factor matrices to improve the robustness and accuracy.

Randomized or sampling tools have been widely adopted in numerical algorithms. However, the randomized numerical linear algebra used to focus on low-rank matrices. The incorporation of such tools into tensor operations was fairly recent [41,20,42,43,25]. Wang *et al.* [41] proposed randomized methods for tensor contractions via FFTs. Such tensor contractions are encountered in CPD methods (e.g., ALS). Vervliet *et al.* [20] proposed a randomized block sampling CPD method that combined randomization and stochastic optimization. Cheng *et al.* [42] proposed *SPALS*, which speeded up sparse ALS via leverage scores sampling. Battaglino *et al.* [43] extended randomized ALS to tensors and reduced the workload of CPD-ALS by checking the stopping condition via sampling-based techniques. Kolda *et al.* [25] proposed using stochastic gradients formed

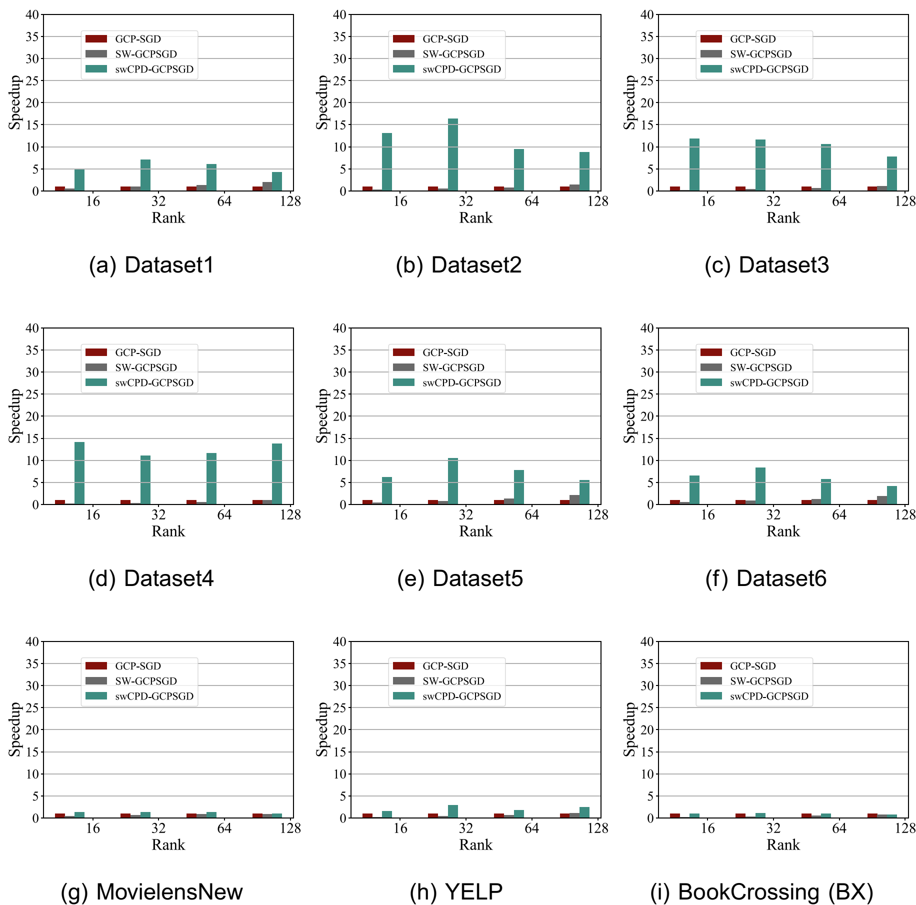

Fig. 14. Performance comparison among *swCPD-GCPSGD*, GCP-SGD and SW-GCPSGD on Sunway processor with the rank of 16, 32, 64 and 128. The loss function is gamma distribution.

from randomly sampled elements for generalized canonical polyadic (GCP) tensor decomposition. The reason why GCP uses SGD is that after generalizing the objective function, the amount of calculation will increase exponentially.

In e-commerce, matrix factorization is used to provide personalized recommendations for products that suit users taste [44]. Similar ideas can be applied to tensor completion [45–47]. Tensor completion is used to recover a tensor by filling in missing values (i.e., the zeros). Gandy *et al.*[45] proposed an algorithm based on the Douglas-Rachford splitting technique to solve the low-rank tensor recovery problem numerically. Liu *et al.* [46] proposed *HaLRTC* that applies the alternating direction method of multipliers (ADMM) to estimate missing values in tensors. Smith *et al.*[47] studied three algorithms (i.e., ALS, SGD and CCD++) to explore opportunities for parallelism accomplished in tensor completion. Besides, Zhang *et al.* [48] utilized CPD algorithm to compress the parameters of the large weight tensors in a large deep computation model.

7.2. Performance optimization on sunway architecture

Many researchers have paid attention to adapt and optimize algorithms and applications on Sunway system. In terms of linear algebra algorithm, Wang et al. [27] implemented a sparse triangular solver (SpTRSV) to Sunway architecture by proposing a brand new Sparse Level Tile layout and utilizing Producer–Consumer pairing methodology. What's more, Liu *et al.* [12] introduced an efficient SpMV algorithm which partitions CPEs into different roles and takes advantage of fast communication between CPEs and this partition scheme was also adopted by Li *et al.* [49] for SpTRSV. Li *et al.* [50] proposed *swCholesky*, which highly optimized sparse Cholesky factorization on Sunway processor. Zhong *et al.* [51] proposed *swTensor* that adapted the CP decomposition to Sunway processor by leveraging the MapReduce framework for automatic parallelization. Xiao *et al.* [52] developed CASpMV on Sunway architecture, which contains a four-way auto-tuning partition scheme based on a statistical model that describes the characteristics of the sparse matrix structure. Meanwhile, ahSpMV [53] is an auto-tuning hybrid computing scheme for Sunway processor, which adopts the Hybrid (HYB) sparse matrix format.

Many scientific computing and deep learning parallel applications have been adapted to Sunway architecture. Fu *et al.* [13] adapted the widely-used climatic simulation application CAM to Sunway architecture. Zhong *et al.* [54] proposed *swMR*,

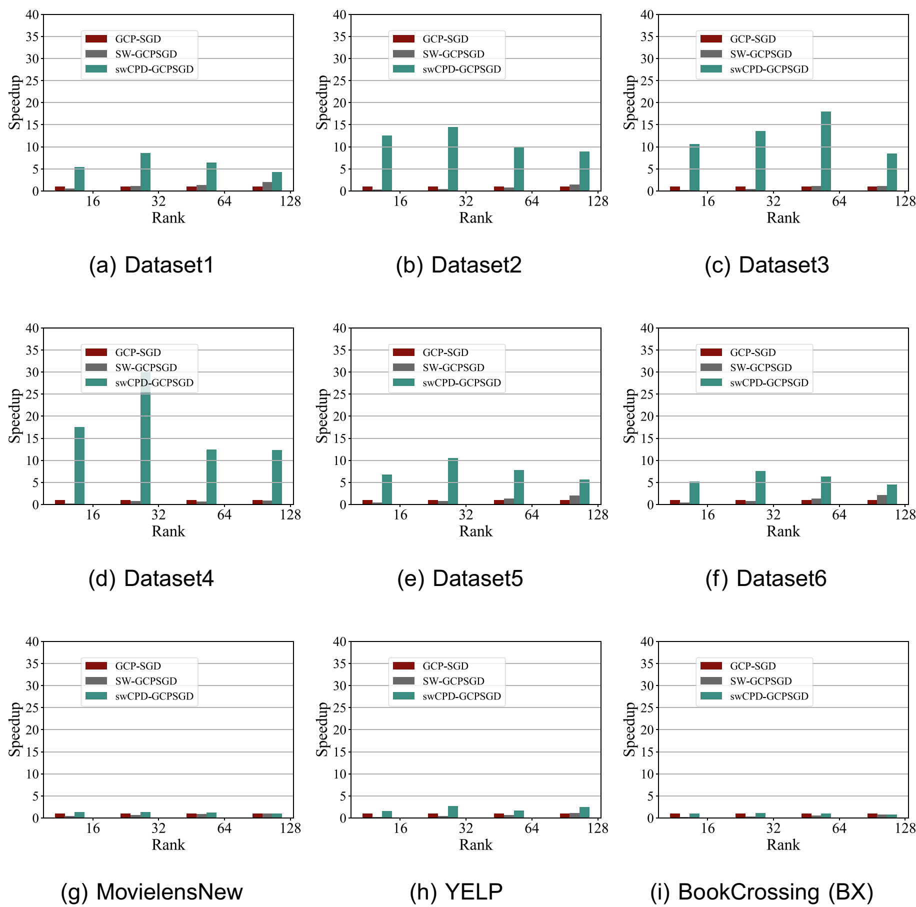

Fig. 15. Performance comparison among *swCPD-GCPSGD*, GCP-SGD and SW-GCPSGD on Sunway processor with the rank of 16, 32, 64 and 128. The loss function is Rayleigh distribution.

which implemented MapReduce on Sunway many-core processor. Hu *et al.* [55] implemented large-scale seismic processing and improved the signal-to-noise ratio. Fang *et al.* [56] proposed *swDNN* that mapped the convolutional neural networks (CNNs) onto the four CGs within the chip. Li *et al.* [14] transferred the deep learning framework Caffe to Sunway TaihuLight. Those researches inspired us on how to improve the performance of the CPD algorithm on Sunway architecture.

8. Conclusion

In this paper, we have provided efficient CPD implementations with optimization algorithms including Alternating Least Squares, Gradient Descent, Randomized Block Sampling, fast Levenberg–Marquardt, and Generalized Canonical Polyadic Decomposition with Stochastic Gradient Descent on the Sunway processor. We have proposed a hierarchical partitioning scheme that partitions both CPEs and sparse tensors into three levels. The partitioning scheme improves the data locality and communication efficiency by utilizing DMA, LDM, and register communication. Moreover, for the randomized MTTKRP process, we have proposed a cooperative and efficient sampling scheme among the CPEs. Our implementation has achieved better performance than the widely adopted Matlab Tensor Toolbox, Matlab Tensorlab, and SPLATT across both synthesized and real-world datasets.

CRediT authorship contribution statement

Ming Dun: Conceptualization, Methodology, Software. **Yunchun Li:** Writing - original draft. **Qingxiao Sun:** Data curation, Visualization, Investigation. **Hailong Yang:** Supervision, Writing - original draft, Resources, Funding acquisition. **Wei Li:** Validation. **Zhongzhi Luan:** Writing - review & editing. **Lin Gan:** Writing - review & editing. **Guangwen Yang:** Writing - review & editing. **Depei Qian:** Writing - review & editing.

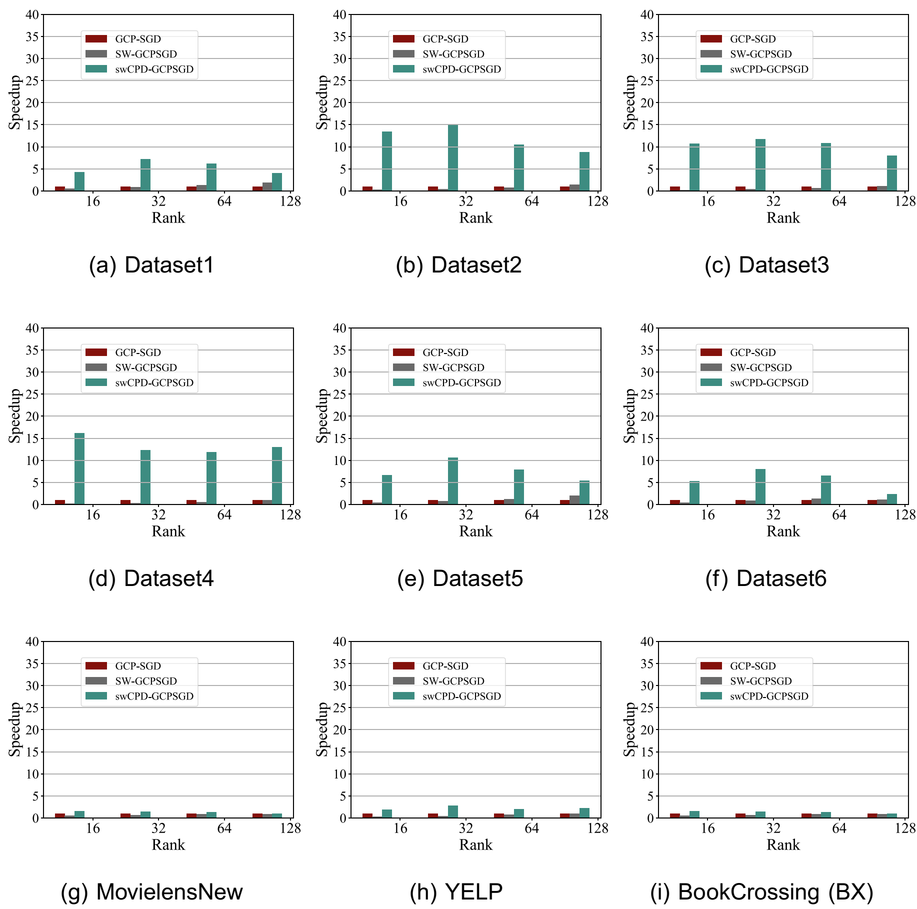

Fig. 16. Performance comparison among *swCPD-GCPSGD*, GCP-SGD and SW-GCPSGD on Sunway processor with the rank of 16, 32, 64 and 128. The loss function is Poisson distribution.

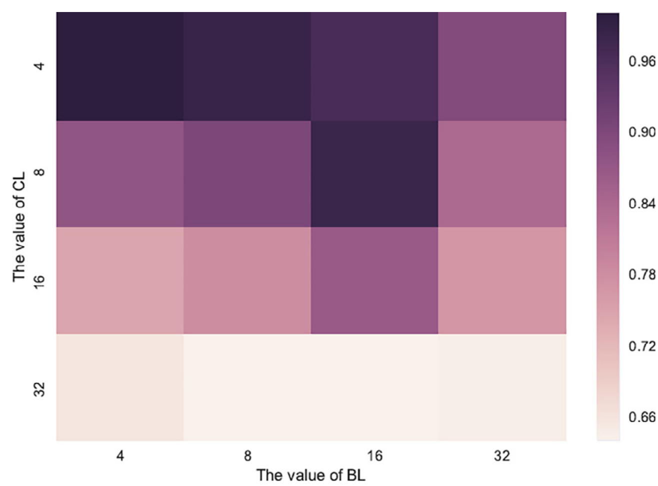

Fig. 17. Parameter sensitivity analysis of CPD on the buffer size of factor matrice B (BL) and C (CL).

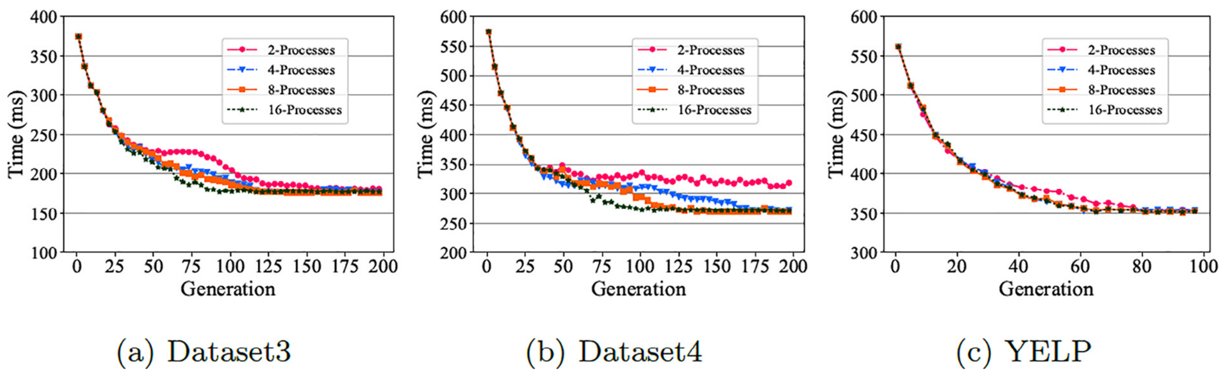

Fig. 18. Convergence of genetic search with 2, 4, 8 and 16 processes. The y-axis is the average execution time of a single iteration in *swCPD-ALS*.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The authors would like to thank all anonymous reviewers for their insightful comments and suggestions. This work is supported by National Key Research and Development Program of China (Grant No. 2020YFB1506703), National Natural Science Foundation of China (Grant No. 62072018 and 61502019), Center for High Performance Computing and System Simulation, Pilot National Laboratory for Marine Science and Technology (Qingdao). Hailong Yang is the corresponding author.

References

[1] M. Sonka, V. Hlavac, R. Boyle, Image Processing, Analysis, and Machine Vision, Cengage Learning, 2014.
[2] S. Aja-Fernández, R. de Luis Garcia, D. Tao, X. Li, Tensors in Image Processing and Computer Vision, Springer Science & Business Media, 2009.
[3] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, N. Oliver, Tfmap: optimizing map for top-n context-aware recommendation, in: Proceedings of the 35th International ACM SIGIR Conference on Research and Development in Information Retrieval, ACM, 2012, pp. 155–164.
[4] T.G. Kolda, B.W. Bader, Tensor decompositions and applications, SIAM Rev. 51 (3) (2009) 455–500.
[5] E. Acar, D.M. Dunlavy, T.G. Kolda, A scalable optimization approach for fitting canonical tensor decompositions, J. Chemom. 25 (2) (2011) 67–86.
[6] B.W. Bader, T.G. Kolda, Efficient matlab computations with sparse and factored tensors, SIAM J. Sci. Comput. 30 (1) (2007) 205–231.
[7] J. Dean, S. Ghemawat, Mapreduce: simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113.
[8] U. Kang, E. Papalexakis, A. Harpale, C. Faloutsos, 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, ACM, 2012, pp. 316–324.
[9] J. Li, Y. Ma, R. Vuduc, ParTI!: a parallel tensor infrastructure for multicore cpus and gpus (Oct 2018). URL:https://github.com/hpcgarage/ParTI..
[10] J. Dongarra, Sunway taihulight supercomputer makes its appearance, Nat. Sci. Rev. 3 (3) (2016) 265–266.
[11] H. Fu, J. Liao, J. Yang, L. Wang, Z. Song, X. Huang, C. Yang, W. Xue, F. Liu, F. Qiao, et al, The sunway taihulight supercomputer: system and applications, Sci. China Inf. Sci. 59 (7) (2016) 072001.
[12] C. Liu, B. Xie, X. Liu, W. Xue, H. Yang, X. Liu, Towards efficient spmv on sunway manycore architectures, in: Proceedings of the 2018 International Conference on Supercomputing, ACM, 2018, pp. 363–373..
[13] H. Fu, J. Liao, W. Xue, L. Wang, D. Chen, L. Gu, J. Xu, N. Ding, X. Wang, C. He, et al., Refactoring and optimizing the community atmosphere model (cam) on the sunway taihulight supercomputer, in: SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2016, pp. 969–980..
[14] L. Li, J. Fang, H. Fu, J. Jiang, W. Zhao, C. He, X. You, G. Yang, swcaffe: a parallel framework for accelerating deep learning applications on sunway taihulight, in: 2018 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2018, pp. 413–422..
[15] J. Choi, X. Liu, S. Smith, T. Simon, Blocking optimization techniques for sparse tensor computation, in: 2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2018, pp. 568–577..
[16] S. Smith, G. Karypis, Tensor-matrix products with a compressed sparse tensor, in: Proceedings of the 5th Workshop on Irregular Applications: Architectures and Algorithms, ACM, 2015, p. 5.
[17] F.L. Hitchcock, The expression of a tensor or a polyadic as a sum of products, J. Math. Phys. 6 (1–4) (1927) 164–189.
[18] L.R. Tucker, Implications of factor analysis of three-way matrices for measurement of change, Probl. Measur. Change 15 (1963) 122–137.
[19] J.H. Choi, S. Vishwanathan, Dfacto: distributed factorization of tensors, Advances in Neural Information Processing Systems (2014) 1296–1304.
[20] N. Vervliet, L. De Lathauwer, A randomized block sampling approach to canonical polyadic decomposition of large-scale tensors, IEEE J. Select. Top. Signal Process. 10 (2) (2015) 284–295.
[21] K. Huang, X. Fu, Low-complexity levenberg-marquardt algorithm for tensor canonical polyadic decomposition, in: ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), IEEE, 2020, pp. 3922–3926.
[22] J.J. Moré, The levenberg-marquardt algorithm: implementation and theory, in: Numerical Analysis, Springer, 1978, pp. 105–116..
[23] A. Ruhe, Accelerated gauss-newton algorithms for nonlinear least squares problems, BIT Numer. Math. 19 (3) (1979) 356–367.
[24] D. Hong, T.G. Kolda, J.A. Duersch, Generalized canonical polyadic tensor decomposition, SIAM Rev. 62 (1) (2020) 133–163.
[25] T.G. Kolda, D. Hong, Stochastic gradients for large-scale tensor decomposition, arXiv preprint arXiv:1906.01687..
[26] S. Smith, N. Ravindran, N.D. Sidiropoulos, G. Karypis, Splatt: Efficient and parallel sparse tensor-matrix multiplication, in: 2015 IEEE International Parallel and Distributed Processing Symposium, IEEE, 2015, pp. 61–70..
[27] X. Wang, W. Liu, W. Xue, L. Wu, swsptrsv: a fast sparse triangular solve with sparse level tile layout on sunway architectures, in: ACM SIGPLAN Notices, vol. 53, ACM, 2018, pp. 338–353..

[28] S. Williams, A. Waterman, D. Patterson, Roofline: an insightful visual performance model for floating-point programs and multicore architectures, Tech. rep., Lawrence Berkeley National Lab. (LBNL), Berkeley, CA (United States) (2009)..
[29] Z. Xu, J. Lin, S. Matsuoka, Benchmarking sw26010 many-core processor, in: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), IEEE, 2017, pp. 743–752..
[30] J.J. Dongarra, J. Du Croz, S. Hammarling, R.J. Hanson, An extended set of fortran basic linear algebra subprograms, ACM Trans. Math. Software (TOMS) 14 (1) (1988) 1–17.
[31] D.E. Goldberg, J.H. Holland, Genetic algorithms and machine learning, Mach. Learn. 3 (2) (1988) 95–99.
[32] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W.S. Moses, S. Verdoolaege, A. Adams, A. Cohen, Tensor comprehensions: framework-agnostic high-performance machine learning abstractions, arXiv preprint arXiv:1802.04730..
[33] A.R. Benson, G. Ballard, A framework for practical parallel fast matrix multiplication, in: ACM SIGPLAN Notices, vol. 50, ACM, 2015, pp. 42–53..
[34] Yelp open dataset (2019). URL:http://www.yelp.com/dataset..
[35] C.-N. Ziegler, S.M. McNee, J.A. Konstan, G. Lausen, Improving recommendation lists through topic diversification, in: Proceedings of the 14th international conference on World Wide Web, ACM, 2005, pp. 22–32.
[36] F.M. Harper, J.A. Konstan, The movielens datasets: history and context, ACM Trans. Interact. Intell. Syst. (TIIS) 5 (4) (2016) 19..
[37] D.M. Dunlavy, T.G. Kolda, E. Acar, Poblano v1. 0: a matlab toolbox for gradient-based optimization, Sandia National Laboratories, Tech. Rep. SAND2010-1422..
[38] N. Vervliet, O. Debals, L. De Lathauwer, Tensorlab 3.0-numerical optimization strategies for large-scale constrained and coupled matrix/tensor factorization, in: 2016 50th Asilomar Conference on Signals, Systems and Computers, IEEE, 2016, pp. 1733–1738.
[39] O. Kaya, B. Uçar, Scalable sparse tensor decompositions in distributed memory systems, in: SC'15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 2015, pp. 1–11..
[40] L. Cheng, Y.-C. Wu, H.V. Poor, Probabilistic tensor canonical polyadic decomposition with orthogonal factors, IEEE Trans. Signal Process. 65 (3) (2016) 663–676.
[41] Y. Wang, H.-Y. Tung, A.J. Smola, A. Anandkumar, Fast and guaranteed tensor decomposition via sketching, in, Adv. Neural Inf. Process. Syst. (2015) 991–999.
[42] D. Cheng, R. Peng, Y. Liu, I. Perros, Spals: fast alternating least squares via implicit leverage scores sampling, in, Adv. Neural Inf. Process. Syst. (2016) 721–729.
[43] C. Battaglino, G. Ballard, T.G. Kolda, A practical randomized cp tensor decomposition, SIAM J. Matrix Anal. Appl. 39 (2) (2018) 876–901.
[44] Y. Koren, R. Bell, C. Volinsky, Matrix factorization techniques for recommender systems, Computer 42 (8) (2009) 30–37.
[45] S. Gandy, B. Recht, I. Yamada, Tensor completion and low-n-rank tensor recovery via convex optimization, Inverse Prob. 27 (2) (2011) 025010.
[46] J. Liu, P. Musialski, P. Wonka, J. Ye, Tensor completion for estimating missing values in visual data, IEEE Trans. Pattern Anal. Mach. Intell. 35 (1) (2012) 208–220.
[47] S. Smith, J. Park, G. Karypis, Hpc formulations of optimization algorithms for tensor completion, Parallel Comput. 74 (2018) 99–117.
[48] Q. Zhang, L.T. Yang, Z. Chen, P. Li, An improved deep computation model based on canonical polyadic decomposition, IEEE Trans. Syst. Man Cybern. Syst. 48 (10) (2017) 1657–1666.
[49] M. Li, Y. Liu, H. Yang, Z. Luan, D. Qian, 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 (HPCCCity/DSS), IEEE, 2018, pp. 594–601.
[50] M. Li, Y. Liu, H. Yang, Z. Luan, L. Gan, G. Yang, D. Qian, Accelerating sparse cholesky factorization on sunway manycore architecture, IEEE Trans. Parallel Distrib. Syst. (2019) 1, https://doi.org/10.1109/TPDS.2019.2953852.
[51] X. Zhong, H. Yang, Z. Luan, L. Gan, G. Yang, D. Qian, swtensor: accelerating tensor decomposition on sunway architecture, CCF Trans. High Perform. Comput. (2019) 1–16.
[52] G. Xiao, K. Li, Y. Chen, W. He, A. Zomaya, T. Li, Caspmv: a customized and accelerative spmv framework for the sunway taihulight, IEEE Trans. Parallel Distrib. Syst..
[53] G. Xiao, Y. Chen, C. Liu, X. Zhou, ahspmv: an autotuning hybrid computing scheme for spmv on the sunway architecture, IEEE Internet Things J. 7 (3) (2019) 1736–1744.
[54] X. Zhong, M. Li, H. Yang, Y. Liu, D. Qian, swmr: a framework for accelerating mapreduce applications on sunway taihulight, IEEE Trans. Emerg. Top. Comput..
[55] Y. Hu, H. Yang, Z. Luan, D. Qian, Massively scaling seismic processing on sunway taihulight supercomputer (2019). arXiv:1907.11678..
[56] J. Fang, H. Fu, W. Zhao, B. Chen, W. Zheng, G. Yang, swdnn: a library for accelerating deep learning applications on sunway taihulight, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2017, pp. 615–624..