MSGD: A Novel Matrix Factorization Approach for Large-scale Collaborative Filtering Recommender Systems on GPUs

Hao Li, Kenli Li, *Senior Member*, *IEEE*, Jiyao An, *Member*, *IEEE*, Keqin Li, *Fellow*, *IEEE*

Abstract— Real-time accurate recommendation of large-scale recommender systems is a challenging task. Matrix factorization (MF), as one of the most accurate and scalable techniques to predict missing ratings, has become popular in the collaborative filtering (CF) community. Currently, stochastic gradient descent (SGD) is one of the most famous approaches for MF. However, it is non-trivial to parallelize SGD for large-scale problems due to the dependence on the user and item pair, which can cause parallelization over-writing. To remove the dependence on the user and item pair, we propose a multi-stream SGD (MSGD) approach, for which the update process is theoretically convergent. On that basis, we propose a CUDA (*Compute Unified Device Architecture*) parallelization MSGD (CUMSGD) approach. CUMSGD can obtain high parallelism and scalability on *Graphic Processing Units* (GPUs). On Tesla K20m and K40c GPUs, the experimental results show that CUMSGD outperforms prior works that accelerated MF on shared memory systems, e.g., DSGD, FPSGD, Hogwild!, and CCD++. For large-scale CF problems, we propose multiple GPUs (multi-GPU) CUMSGD (MCUMSGD). The experimental results show that MCUMSGD can improve MSGD performance further. With a K20m GPU card, CUMSGD can be 5-10 times as fast compared with the state-of-the-art approaches on shared memory platform.

Index Terms—Collaborative filtering (CF), CUDA parallelization algorithm, Matrix factorization (MF), Multi-GPU implementation, Stochastic gradient descent (SGD).

1 INTRODUCTION

In the era of e-commerce, recommender systems have been applied to all aspects of commercial fields. The following topics are two challenges in the development of personalized recommender systems: (1) making recommendations in real-time, i.e., making recommendations when they are needed, and (2) making recommendations that suit users' tastes, i.e., recommendation accuracy. CF, as one of the most popular recommender systems, relies on past user behavior and does not need domain knowledge or extensive and specialized data collection [1]. The main task of CF is to estimate the missing data of user-interested items, e.g., scores, clicks, purchase records, etc. However, data sparsity because of missing data makes it hard to provide accurate prediction [2]. MF, as a dimensionality reduction technique, that maps both users and items into the same latent factor space, updating user/item feature vectors for existing rating and then predicting the unknown ratings relying on the inner products of related user-item feature vector pairs[3], can address the problem of sparsity and has enjoyed surge in research after the Netflix competition [4], [5], [6].

There are several MF techniques that have been applied to CF recommender systems, e.g., Maximum Margin Matrix Factorization (MMMF) [3], [7], Alternating Least Squares (ALS) [8], [9], Cyclic Coordinate Descent (CCD++) [10–12], Non-negative Matrix Factorization (NMF) [13] and Stochastic Gradient Descent (SGD) [4], [5]. Among them, SGD is a popular and efficient approach due to its low time complexity and convenient implementation, which have been drawn wide attention [5]. Many recent works integrate factors and neighbors, and SGD can fuse the factors and neighbors into the update process [4], [5].

Although SGD has been applied to MF recommender systems successfully, due to high scalability and accuracy, it is difficult to parallelize SGD because of the sequential inherence of SGD, which makes SGD infeasible for large-scale CF data sets. With the rapid development of graphics processing hardware, GPU is becoming common; moreover, CUDA programming makes it possible to harness the computation power of GPU efficiently. However, given the sequential essence of SGD, there is a great obstacle preventing SGD from obtaining the efficiency of GPU computation.

In this paper, we find that the main obstacle to parallelize SGD is the dependence on the user and item pair by theoretical analysis. We apply the MSGD to remove the dependence on the user and item pair. On that basis, we propose a CUDA parallelization approach on GPU, namely CUMSGD. CUMSGD divides the task into coarse sub-tasks that are mapped to independent thread blocks, and then be solved by those independent thread blocks. Each sub-task is divided into finer pieces that map to threads within the thread block, then be solved cooperatively by those threads in parallel.

MSGD can split the MF optimization objective into many separable optimization sub-objectives, and these independent optimization sub-objectives are distributed to CUDA

- *Hao Li, Kenli Li, Jiyao An, and Keqin Li are with the College of Computer Science and Electronic Engineering, Hunan University, and National Supercomputing Center in Changsha, Hunan, China, 410082.*
- *Corresponding author: Kenli Li.*
  E-mail: lihao123@hnu.edu.cn, lkl@hnu.edu.cn, jt_anbob@hnu.edu.cn, lik@newpaltz.edu.
- *Keqin Li is also with the Department of Computer Science, State University of New York, New Paltz, New York 12561, USA.*

thread blocks. There is no data dependence between those thread blocks. Thus, CUMSGD does not require thread block synchronization. CUMSGD updates user and item feature matrices alternatively. Thus, in multi-GPU environment, MCUMSGD can update user sub-matrix with emitting and receiving item feature sub-matrix concurrently. The main contributions of this paper are summarized as follows:

- Theoretic convergence and local minimum escaping: MSGD updates the user feature matrix and item feature matrix alternatively, which is under the convergence promise. Stochastic Gradient (SG) based approaches can be used to escape local minimums. Thus, MSGD can obtain reasonable accuracy.
- CUDA parallelization: Considering the fine grained parallelism of MSGD, we propose a CUMSGD approach. The CUMSGD approach can be brought to CUDA-supported GPUs, and can be applied to industrial applications of MF.
- Multi-GPU implementation for large-scale data sets: For web-level CF data sets, we propose the MCUMSGD. We adopt an asynchronous communication strategy to hide data access overhead between the multi-GPU and cyclic update strategy to allocate rating sub-matrices and item feature sub-matrices.

The extensive experimental results show that CUMSGD not only outperforms the state-of-the-art parallelized SGD MF algorithms in a shared memory setting, e.g., fast parallel SGD (FPSGD) [14], [15], distributed SGD (DSGD) [16], [17] and Hogwild! [18], but also outperforms state-of-the-art non-SG algorithm CCD++ [10], [11].

The paper is organized as follows. We introduce related work in Section 2. We describe the MF problem and related notions in Section 3. We derive the update rule of MSGD, and propose parallelization strategy in Section 4. We provide the parallelization strategies of CUMSGD and MCUMSGD in Section 5. We report experimental results of MSGD and the performance comparison with the state-of-the-art parallel approach in Section 6. Finally, in Section 7, conclusions are drawn.

2 RELATED WORK

MF-based techniques have been proposed as collaborative prediction. The idea of MF-based CF has been applied to many related areas, and there are a large number of studies on this topic. Sarwar *et al.* [19] proposed MF-based dimensionality reduction in CF recommender systems, which can solve several limitations of neighbor-based techniques, e.g., sparsity, scalability, and synonymy. Weng *et al.* [20] presented CF semantic video indexing by adopting MF, which takes into account the correlation of concepts and similarity of shots within the score matrix. Zheng *et al.* [21], proposed neighborhood-integrated MF for quality-of-service (QoS) prediction in service computing, which can avoid time-consuming web service invocation. Lian *et al.* [22] proposed point-of-interest (PoI) recommendation by adopting weight MF to cope with the challenges of the extreme sparsity of user-POI matrices.

SGD has been extensively studied, due to its easy implementation and reasonable accuracy. Langford *et al.* [23] proved that the online learning approach (delayed SGD) is convergent. Zinkevich *et al.* [24] proposed parallelized SGD under convergence guarantees. Agarwal and Duchi [25] proposed distributed delayed SGD, in which the master nodes update parameters and the other nodes perform SGD in parallel. Many works have been conducted on parallelized and distributed SGD to accelerate MF on CF recommender systems in the literatures [14–18], [26], [27]. Chin *et al.* [14] focused on a shared memory multi-core environment to parallelize SGD MF. FPSGD cannot solve large-scale CF problem because FPSGD must load CF data into shared memory. Gemulla *et al.* [16] focused on DSGD in a distributed environment. DSGD took the property that some blocks which share no common columns or common rows of the rating matrix. DSGD needs the data communication overhead on nodes. Because of non-even distribution of the rating matrix, the workloads on different nodes may be different. Thus, each node requires synchronization. Yun *et al.* [26] proposed asynchronous and decentralized SGD (NOMAD) in distributed and shared memory multi-core environments. This approach can hide communication time in SGD execution by NOMAD asynchronous via a non-blocking mechanism and can handle different hardware and system loads by balancing the workload dynamically. NOMAD does not need extra time for data exchanges. However, when the number of nodes increases, there is a large extra execution of uniform sampling instructions for each node or core, and the cost of communication will be larger than the cost of computation. Niu *et al.* [18] proposed Hogwild! to parallelize SGD MF. This approach assumes that the rating matrix is highly sparse and deduced that, for two randomly sampled ratings, two serial updates are likely to be independent. Jin *et al.* [27] presented GPUSGD to accelerate MF. This approach must load the whole data into global memory, and it is unsuited to large-scale CF problems.

Owens *et al.* [28] introduced general purpose computing on GPU and listed GPU computing applications, e.g., game physics and computational biophysics. Pratx and Xing [29] reviewed GPU computing applications in medical physics, including three areas: dose calculation, treatment plan optimization, and image processing. There are some works to analyze and predict GPU performance. Guo *et al.* [30] presented a performance modeling and optimization analysis tool that can provide optimal SPMV solutions based on CSR, ELL, COO, and HYB formats. To improve the performance of SPMV, Li *et al.* [31] considered the probability of the row non-zero element probability distribution to use four sparse matrix storage formats efficiently. In CF recommender systems, Gao *et al.* [32] proposed item-based CF recommender systems on GPU. Kato and Hosino [33] proposed singular value decomposition (SVD) based CF on GPU. However, this approach is not suitable for large-scale CF problems.

To solve the problem of large-scale decision variables, known as the *curse of dimensionality*, Cano and Garcia-Martinez [34] proposed distributed GPU computing for large-scale global optimization. To solve the problem of non-even distribution of a sparse matrix, which can lead to load imbalance on GPUs and multi-core CPUs, Yang *et al.* [35] proposed a probability-based partition strategy. A. Stuart and D. Owens [36] proposed MapReduce-based GPU clustering. MapReduce is focused the parallelization rather than the details of communication and resource allocation. Thus, this approach integrates large numbers of map and reduce chunks and uses partial reductions and accumulations to harness the GPU clusters. Chen *et al.* [37] proposed an in-memory heterogeneous CPU-GPU computing architecture to process high data-intensive applications.

3 PRELIMINARY

In this section, we give the problem definition of MF and set up some notions in Section 3.1. We present the state-of-the-art parallelized and distributed SGD and non-SG approach in Section 3.2. CUDA and multi-GPU details are given in Section 3.3.

3.1 Problem and Notation

We denote matrices by uppercase letters and vectors by bold-faced lowercase letters. Let $A \in \mathbb{R}^{m\times n}$ be the rating matrix in a recommender system, where m and n are the numbers of users and items, respectively. $a_{i,j}$ will denote the (i,j) entry of A. We denote $U \in \mathbb{R}^{m\times r}$ and $V \in \mathbb{R}^{n\times r}$ as the user feature matrix and item feature matrix, respectively. Because of the sparsity of CF problems, we use Ω to denote sparse matrix indices. We use Ω_i and $\overline{\Omega}_j$ to denote the column indices and row indices in the ith row and jth column, respectively. We also denote the kth column of U by $\overline{\mathbf{u}}_k$, and the kth column of V by $\overline{\mathbf{v}}_k$. The kth elements of $\mathbf{u}_i$ and $\mathbf{v}_j$ are represented by $u_{i,k}$ and $v_{j,k}$, respectively. The ith elements of $\overline{\mathbf{u}}_k$ is $\overline{u}_{k,i}$, and the jth element of $\overline{\mathbf{v}}_k$ is $\overline{v}_{k,j}$. In CF MF recommenders, most of algorithms in related literatures take UV^T as the low-rank approximation matrix to predict the non-rated or zero entries of the sparse rating matrix A, and the approximation process is accomplished by minimizing the Euclidean distance function to measure the approximation degree [3–5], [7]. The MF problem for recommender systems is defined as following [3–5], [7]:

$$\arg\min_{U,V} \; \underset{A\leftrightarrow U,V}{d} = \sum_{(i,j)\in\Omega} \Big((a_{i,j} - \widehat{a}_{i,j})^2 + \lambda_U \|\mathbf{u}_i\|^2 + \lambda_V \|\mathbf{v}_j\|^2 \Big), \tag{1}$$

where $\widehat{a}_{i,j} = \mathbf{u}_i \mathbf{v}_j^T$, $\|\cdot\|$ is the Euclidean norms, and λ_u and λ_v are regularization parameters to avoid over-fitting. We observe that problem (1) involves two parameters, U and V. Problem (1) is non-convex. Thus, the distance parameter d may fall into local minima. The SG-based approach can escape the local minima by randomly selecting a (i,j) entry. Once $a_{i,j}$ is chosen, the objective function in problem (1) is reduced to

$$d_{i,j} = (a_{i,j} - \widehat{a}_{i,j})^2 + \lambda_U \|\mathbf{u}_i\|^2 + \lambda_V \|\mathbf{v}_j\|^2. \tag{2}$$

After calculating the sub-gradient over $\mathbf{u}_i$ and $\mathbf{v}_j$, and selecting the regularization parameters λ_U and λ_V, the update rule is formulated as:

$$\begin{cases} \mathbf{v}_j \leftarrow \mathbf{v}_j + \gamma(e_{i,j}\mathbf{u}_i - \lambda_V \mathbf{v}_j); \\ \mathbf{u}_i \leftarrow \mathbf{u}_i + \gamma(e_{i,j}\mathbf{v}_j - \lambda_U \mathbf{u}_i), \end{cases} \tag{3}$$

where

$$e_{i,j} = a_{i,j} - \widehat{a}_{i,j}, \tag{4}$$

is the error between the existing rating and predicted rating of the (i,j) entry, and γ is the learning rate.

The update rule (3) is inherently sequential. We randomly select three entry indices $\{(i_0,j_0),(i_0,j_1),(i_0,j_2)\}$, where $\{j_0,j_1,j_2\} \in \Omega_{i_0}$. We select three threads $\{T_0,T_1,T_2\}$ to update the specific feature vector of $\{a_{i_0,j_0}, a_{i_0,j_1}, a_{i_0,j_2}\}$, respectively. The update process is as follows:

$$T_0: \begin{cases} \mathbf{v}_{j_0} \leftarrow \mathbf{v}_{j_0} + \gamma(e_{i_0,j_0}\mathbf{u}_{i_0} - \lambda_V \mathbf{v}_{j_0}); \\ \mathbf{u}_{i_0} \leftarrow \mathbf{u}_{i_0} + \gamma(e_{i_0,j_0}\mathbf{v}_{j_0} - \lambda_U \mathbf{u}_{i_0}), \end{cases}$$
$$T_1: \begin{cases} \mathbf{v}_{j_1} \leftarrow \mathbf{v}_{j_1} + \gamma(e_{i_0,j_1}\mathbf{u}_{i_0} - \lambda_V \mathbf{v}_{j_1}); \\ \mathbf{u}_{i_0} \leftarrow \mathbf{u}_{i_0} + \gamma(e_{i_0,j_1}\mathbf{v}_{j_1} - \lambda_U \mathbf{u}_{i_0}), \end{cases}$$
$$T_2: \begin{cases} \mathbf{v}_{j_2} \leftarrow \mathbf{v}_{j_2} + \gamma(e_{i_0,j_2}\mathbf{u}_{i_0} - \lambda_V \mathbf{v}_{j_2}); \\ \mathbf{u}_{i_0} \leftarrow \mathbf{u}_{i_0} + \gamma(e_{i_0,j_2}\mathbf{v}_{j_2} - \lambda_U \mathbf{u}_{i_0}). \end{cases}$$

We observe that three threads $\{T_0,T_1,T_2\}$ update $\mathbf{u}_{i_0}$ simultaneously, which is the parallelization over-writing problem. The update process cannot be separated into independent parts because of the dependence on the user and item pair.

3.2 The State-of-the-Art Parallelization Approaches

3.2.1 *SG Based*

Limited by the over-writing problem, i.e., the dependence on the user-item pair, prior works focus on splitting the rating matrix into several independent blocks that do not share the same row and column and applying SGD on those independent blocks.

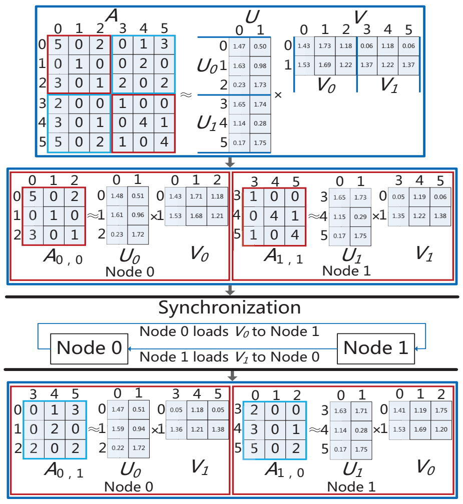


Fig. 1. An illustration of DSGD synchronous cyclic update.

DSGD In the following discussion, the independent blocks share neither common column nor common row of the rating matrix. Given a 2-by-2 divided rating matrix and 2 nodes, cyclic update approach of DSGD is illustrated in Fig. 1. The first training iteration assigns 2 diagonal matrix blocks to 2 nodes $\{Node\ 0, Node\ 1\}$; $Node\ 0$ updates $\{U_0, V_0\}$, $Node\ 1$ updates $\{U_1, V_1\}$. In the last iteration, $Node\ 0$ and $Node\ 1$ update $\{U_0, V_1\}$ and $\{U_1, V_0\}$, respectively. In distributed systems, data communication, synchronization, and data imbalance prevent DSGD from obtaining scalable speedup.

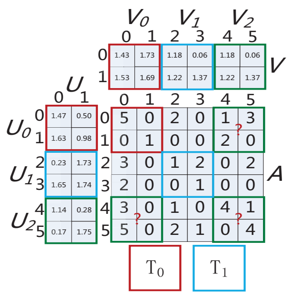


Fig. 2. An illustration of FPSGD.

FPSGD Fig. 2 illustrates FPSGD. FPSGD splits the rating matrix into several independent blocks and implants SGD in each independent block. The first iteration assigns 2 diagonal matrix blocks to 2 threads $\{T_0, T_1\}$; thread T_0 updates $\{U_0, V_0\}$, and thread T_1 updates $\{U_1, V_1\}$. We assume that thread T_0 accomplishes the update at first, and then thread T_0 has three choices, e.g., $\{U_2, V_0\}$, $\{U_0, V_2\}$, $\{U_2, V_2\}$ to the next iteration.

Hogwild! The intuition of Hogwild! is that the rating matrix is very sparse. Thus, the occurrence probability of the over-writing problem is low.

3.2.2 *Non-SG Based*

CCD++ Based on the idea of coordinate descent, CCD++ updates ($\overline{\mathbf{u}}_1$, $\overline{\mathbf{v}}_1$) until ($\overline{\mathbf{u}}_r$, $\overline{\mathbf{v}}_r$) cyclically. Updating $\overline{\mathbf{u}}_k$ can be converted into updating $\overline{u}_{k,i}$ in parallel, and updating $\overline{\mathbf{v}}_k$ can be converted into updating $\overline{v}_{k,j}$ in parallel. The one-variable subproblem of updating $\overline{\mathbf{u}}_k$ is reformulated into

$$\overline{u}_{k,i} \leftarrow \arg\min_{z} \sum_{j\in\Omega_i} \{ (a_{i,j} - (\widehat{a}_{i,j} - \overline{u}_{k,i}\overline{v}_{k,j}) - z\overline{v}_{k,j})^2 + \lambda_U z^2 \}, \tag{5}$$

and the one-variable subproblem of updating $\overline{\mathbf{v}}_k$ is reformulated into

$$\overline{v}_{k,j} \leftarrow \arg\min_{w} \sum_{i\in\overline{\Omega}_j} \{ (a_{i,j} - (\widehat{a}_{i,j} - \overline{u}_{k,i}\overline{v}_{k,j}) - z\overline{u}_{k,j})^2 + \lambda_V w^2 \}. \tag{6}$$

The solution of (5) is

$$z \leftarrow \frac{\sum_{j\in\Omega_i}(a_{i,j} - (\widehat{a}_{i,j} - \overline{u}_{k,i}\overline{v}_{k,j}))\overline{v}_{k,j}}{\lambda_U + \sum_{j\in\Omega_i}\overline{v}_{k,j}^2}, \overline{u}_{k,i} \quad \leftarrow z, \tag{7}$$

and the solution of (6) is

$$w \leftarrow \frac{\sum_{j\in\Omega_i}(a_{i,j} - (\widehat{a}_{i,j} - \overline{v}_{k,j}\overline{u}_{k,i}))\overline{u}_{k,i}}{\lambda_U + \sum_{i\in\overline{\Omega}_j}\overline{u}_{k,i}^2}, \overline{v}_{k,j} \quad \leftarrow w. \tag{8}$$

3.3 GPU-Based Computing

The GPU resides on a *device*, and a GPU consists of many *stream multiprocessors* (SMs). Each SM contains a number of *stream processors* (SPs). CUDA is a programming interface that can enable GPU to be compatible with various programming languages and applications. In CUDA, *kernels* are functions that are executed on GPU. A kernel function is executed by a batch of threads. The batch of threads is organized as a grid of thread blocks. The thread blocks map to SMs. As shown in Fig. 3, the greater the number of SMs a GPU has, the higher the parallelism degree the GPU has. Thus, a GPU with more SMs will execute a CUDA program in less time than a GPU with fewer SMs. Threads in a blocks are organized into small groups of 32 called *warps* for execution on the processors, and warps are implicitly synchronous; however, threads in different blocks are asynchronous. CUDA assumes that CUDA kernel, i.e., CUDA program, is executed on a GPU (*device*) and the rest of the C program is executed on the CPU (*host*). CUDA threads access data from multiple memory hierarchies. Each thread has private local memory, and each thread block has shared memory that is visible to all threads within the thread block. All threads can access global memory.

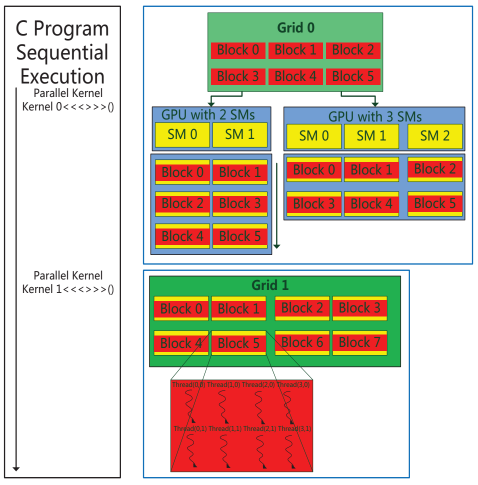

C Program Sequential Execution
Parallel Kernel Kernel 0<<<>>>()
Parallel Kernel Kernel 1<<<>>>()
Grid 0
Block 0 Block 1 Block 2
Block 3 Block 4 Block 5
GPU with 2 SMs
SM 0 SM 1
GPU with 3 SMs
SM 0 SM 1 SM 2
Grid 1
Block 0 Block 1 Block 2 Block 3
Block 4 Block 5 Block 6 Block 7

Fig. 3. CUDA kernel and thread batching.

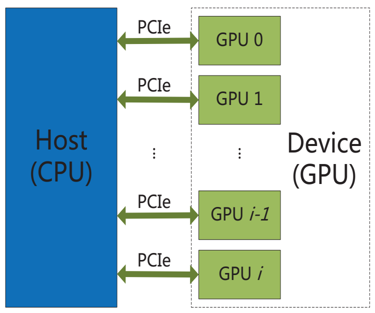


Fig. 4. Multi-GPU parallel computing model.

Multiple GPUs are connected to the *host* by *peripheral communication interconnect express* (PCIe) in Fig. 4. CUDA can perform computation on the device and data transfers between the *device* and *host* concurrently. More state-of-the-art GPUs (K20m, K40c) have two copy engines, which can operate data transfers to and from a device, and among devices concurrently. CUDA provides synchronization instructions, which can ensure the correct execution on multi-GPU, and each GPU of the multi-GPU has a consumer/producer relationship. Memory copies between two different devices can be performed after the instruction of CUDA peer-to-peer memory access has been enabled.

4 A MSGD MF APPROACH

In Section 4.1, we propose the MSGD MF approach, which can remove the dependence on the user and item pair. We present the parallelization design of MSGD in Section 4.2.

4.1 Proposed MSGD Approach

The essence of SGD parallelization over-writing problem is that the variables that are selected by multiple threads may share the same row or column identity. In a training epoch, multiple threads select $a_{i,j}$ belonging to the same row of A, and SGD only updates V. Correspondingly, multiple threads selected $a_{i,j}$ belonging to the same column of A, and SGD only updates U. Then, the parallelization over-writing problem can be solved. However, how can the convergence and accuracy be guaranteed? In the following section, we will give the derivation of MSGD update rules.

MSGD splits the distance function d of problem (1) into

$$d = \sum_{i=1}^{m} |\Omega_i| d_i, \tag{9}$$

and

$$d = \sum_{j=1}^{n} |\overline{\Omega}_j| \overline{d}_j, \tag{10}$$

where

$$\begin{aligned} d_i &= \frac{1}{|\Omega_i|} \sum_{j\in\Omega_i} \left(e_{i,j}^2 + \lambda_U \|\mathbf{u}_i\|^2 + \lambda_V \|\mathbf{v}_j\|^2\right) \\ &= \frac{1}{|\Omega_i|} \sum_{j\in\Omega_i} \psi_j(\mathbf{u}_i), \end{aligned} \tag{11}$$

and

$$\begin{aligned} \overline{d}_j &= \frac{1}{|\overline{\Omega}_j|} \sum_{i\in\overline{\Omega}_j} \left(e_{i,j}^2 + \lambda_U \|\mathbf{u}_i\|^2 + \lambda_V \|\mathbf{v}_j\|^2\right) \\ &= \frac{1}{|\overline{\Omega}_j|} \sum_{i\in\overline{\Omega}_j} \overline{\psi}_i(\mathbf{v}_j). \end{aligned} \tag{12}$$

Thus, minimizing d is equivalent to minimize $\sum\limits_{j\in\Omega_i} |\Omega_i| d_i$ and $\sum\limits_{i\in\overline{\Omega}_j} |\overline{\Omega}_j| \overline{d}_j$ alternatively. Furthermore, minimizing $|\Omega_i| d_i$ is equivalent to minimize d_i, and minimizing $|\overline{\Omega}_j| \overline{d}_j$ is equivalent to minimize $\overline{d}_j$. If we fix $\mathbf{v}_j, j \in \Omega_i$ for d_i, then the $d_i, i \in \{1, 2, \cdots, m\}$ are independent of each other. Therefore, applying gradient decent with learning rate γ to search for the optimal solution to minimize d_i and $\overline{d}_j$ is derived as follows:

$$\begin{aligned} \arg\min_{\mathbf{u}_i} d_i \Rightarrow \mathbf{u}_i &\leftarrow \mathbf{u}_i - \gamma \nabla d_i(\mathbf{u}_i) \\ &= \mathbf{u}_i - \frac{2\gamma}{|\Omega_i|} \left(\sum_{j\in\Omega_i} \nabla \psi_j(\mathbf{u}_i) \right) \\ &= \mathbf{u}_i - \frac{2\gamma}{|\Omega_i|} \left(\sum_{j\in\Omega_i} (-e_{i,j}\mathbf{v}_j + \lambda_U \mathbf{u}_i) \right), \end{aligned} \tag{13}$$

and

$$\begin{aligned} \arg\min_{\mathbf{v}_j} \overline{d}_j \Rightarrow \mathbf{v}_j &\leftarrow \mathbf{v}_j - \gamma \nabla \overline{d}_j(\mathbf{v}_j) \\ &= \mathbf{v}_j - \frac{2\gamma}{|\overline{\Omega}_j|} \left(\sum_{i\in\overline{\Omega}_j} \nabla \overline{\psi}_i(\mathbf{v}_j) \right) \\ &= \mathbf{v}_j - \frac{2\gamma}{|\overline{\Omega}_j|} \left(\sum_{i\in\overline{\Omega}_j} (-e_{i,j}\mathbf{u}_i + \lambda_V \mathbf{v}_j) \right). \end{aligned} \tag{14}$$

However, at each step, gradient descent requires the evaluation of $|\Omega_i|$ and $|\overline{\Omega}_j|$ to update $\mathbf{u}_i$ and $\mathbf{v}_j$ respectively, which is expensive. SGD is a popular modification for the gradient descent update rule (13) [38–43], where at each training epoch $t = 1, 2, \ldots$, we randomly draw $j_t, j_t \in \Omega_i$, and the modification for the gradient descent update rule (13) to update U is as follows:

$$\begin{aligned} \mathbf{u}_i^t &= \mathbf{u}_i^{t-1} - \gamma \nabla \psi_{j_t}(\mathbf{u}_i^{t-1}) \\ &= \mathbf{u}_i^{t-1} + \gamma(e_{i,j}\mathbf{v}_{j_t}^{t-1} - \lambda_U \mathbf{u}_i^{t-1}). \end{aligned} \tag{15}$$

The same modification for the gradient descent update rule (14) to update V is as follows:

$$\begin{aligned} \mathbf{v}_j^t &= \mathbf{v}_j^{t-1} - \gamma \nabla \overline{\psi}_{i_t}(\mathbf{v}_j^{t-1}) \\ &= \mathbf{v}_j^{t-1} + \gamma(e_{i,j}\mathbf{u}_{i_t}^{t-1} - \lambda_V \mathbf{v}_j^{t-1}), \end{aligned} \tag{16}$$

where at each epoch $t = 1, 2, \ldots$, we randomly draw $i_t, i_t \in \overline{\Omega}_j$. The expectation $\mathbb{E}[\mathbf{u}_i^t | \mathbf{u}_i^{t-1}]$ is identical to (13), and the expectation $\mathbb{E}[\mathbf{v}_j^t | \mathbf{v}_j^{t-1}]$ is identical to (14) [38–42]. The algorithm of MSGD is described in Algorithm 1.

Algorithm 1 MSGD

Input: Initial U and V, rating matrix A, learning rate γ, regularization parameters λ_V and λ_U, training epoch epo.
Output: U, V.

```
1:  for $loop$ from $epo$ to 0 do
2:      for $j$ from 0 to $n-1$ do
3:          for $i$ from the first to the last element of $\overline{\Omega}_j$ do
4:              Update $\mathbf{v}_j$ by update rule (16).
5:          end for
6:      end for
7:      for $i$ from 0 to $m-1$ do
8:          for $j$ from the first to the last element of $\Omega_i$ do
9:              Update $\mathbf{u}_i$ by update rule (15).
10:         end for
11:     end for
12: end for
13: return $(U, V)$.
```

Since problems (11) and (12) are symmetric, we only consider the example of problem (11). In general, under two constraint conditions where ψ_j is smooth and Lipschitz-continuous with constant L and d_i is strongly-convex with constant μ, the MSGD is convergent, and as long as we pick a small constant step size $\gamma < 1/L$, gradient descent of d_i obtains linear convergence rate $O(1/t)$ [39], [44]. We present the convergence analysis of MSGD in the supplementary material. Furthermore, MSGD has the same time complexity as SGD, i.e., $O(r|\Omega|)$.

4.2 Parallelization Design

We report two parallelization approaches of MSGD, e.g., a decentralized approach and a centralized approach.

We select three entry indices $\{(i_0, j_0), (i_0, j_1), (i_0, j_2)\}$, where $\{j_0, j_1, j_2\} \in \Omega_{i_0}$. We select three threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{v}_{j_0}, \mathbf{v}_{j_1}, \mathbf{v}_{j_2}\}$, respectively. As decentralized approach, threads $\{T_0, T_1, T_2\}$ update $\{\mathbf{v}_{j_0}, \mathbf{v}_{j_1}, \mathbf{v}_{j_2}\}$ in parallel as follows:

$$\begin{bmatrix} \mathbf{v}_{j_0}^t \\ \mathbf{v}_{j_1}^t \\ \mathbf{v}_{j_2}^t \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{j_0}^{t-1} + \gamma(e_{i_0,j_0}\mathbf{u}_{i_0}^{t-1} - \lambda_V \mathbf{v}_{j_0}^{t-1}) \\ \mathbf{v}_{j_1}^{t-1} + \gamma(e_{i_0,j_1}\mathbf{u}_{i_0}^{t-1} - \lambda_V \mathbf{v}_{j_1}^{t-1}) \\ \mathbf{v}_{j_2}^{t-1} + \gamma(e_{i_0,j_2}\mathbf{u}_{i_0}^{t-1} - \lambda_V \mathbf{v}_{j_2}^{t-1}) \end{bmatrix}. \tag{17}$$

In the $(i_0 + 1)$th iteration, threads $\{T_0, T_1, T_2\}$ select three different column indices from Ω_{i_0}, and update $\{\mathbf{v}_{j_0}, \mathbf{v}_{j_1}, \mathbf{v}_{j_2}\}$ in parallel.

We select three entry indices $\{(i_0, j_0), (i_1, j_1), (i_1, j_2)\}$, where $i_0 \in \overline{\Omega}_{j_0}$, $i_1 \in \overline{\Omega}_{j_1}$, $i_2 \in \overline{\Omega}_{j_2}$. We select three threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{v}_{j_0}, \mathbf{v}_{j_1}, \mathbf{v}_{j_2}\}$, respectively. As centralized approach, threads $\{T_0, T_1, T_2\}$ update $\{\mathbf{v}_{j_0}, \mathbf{v}_{j_1}, \mathbf{v}_{j_2}\}$ in parallel as follows:

$$\begin{bmatrix} \mathbf{v}_{j_0}^t \\ \mathbf{v}_{j_1}^t \\ \mathbf{v}_{j_2}^t \end{bmatrix} = \begin{bmatrix} \mathbf{v}_{j_0}^{t-1} + \gamma(e_{i_0,j_0}\mathbf{u}_{i_0}^{t-1} - \lambda_V \mathbf{v}_{j_0}^{t-1}) \\ \mathbf{v}_{j_1}^{t-1} + \gamma(e_{i_1,j_1}\mathbf{u}_{i_1}^{t-1} - \lambda_V \mathbf{v}_{j_1}^{t-1}) \\ \mathbf{v}_{j_2}^{t-1} + \gamma(e_{i_2,j_2}\mathbf{u}_{i_2}^{t-1} - \lambda_V \mathbf{v}_{j_2}^{t-1}) \end{bmatrix}. \tag{18}$$

Furthermore, we report two approaches to update U in parallel.

We select three entry indices $\{(i_0, j_0), (i_1, j_0), (i_2, j_0)\}$, where $\{i_0, i_1, i_2\} \in \overline{\Omega}_{j_0}$. We select three threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{u}_{i_0}, \mathbf{u}_{i_1}, \mathbf{u}_{i_2}\}$, respectively. The decentralized approach of threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{u}_{i_0}, \mathbf{u}_{i_1}, \mathbf{u}_{i_2}\}$ is as follows:

$$\begin{bmatrix} \mathbf{u}_{i_0}^t \\ \mathbf{u}_{i_1}^t \\ \mathbf{u}_{i_2}^t \end{bmatrix} = \begin{bmatrix} \mathbf{u}_{i_0}^{t-1} + \gamma(e_{i_0,j_0}\mathbf{v}_{j_0}^{t-1} - \lambda_U \mathbf{u}_{i_0}^{t-1}) \\ \mathbf{u}_{i_1}^{t-1} + \gamma(e_{i_1,j_0}\mathbf{v}_{j_0}^{t-1} - \lambda_U \mathbf{u}_{i_1}^{t-1}) \\ \mathbf{u}_{i_2}^{t-1} + \gamma(e_{i_2,j_0}\mathbf{v}_{j_0}^{t-1} - \lambda_U \mathbf{u}_{i_2}^{t-1}) \end{bmatrix}. \tag{19}$$

We select three entry indices $\{(i_0, j_0), (i_1, j_1), (i_1, j_2)\}$, where $j_0 \in \Omega_{i_0}$, $j_1 \in \Omega_{i_1}$, $j_2 \in \Omega_{i_2}$. We select three threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{u}_{i_0}, \mathbf{u}_{i_1}, \mathbf{u}_{i_2}\}$, respectively. The centralized approach of threads $\{T_0, T_1, T_2\}$ to update $\{\mathbf{u}_{i_0}, \mathbf{u}_{i_1}, \mathbf{u}_{i_2}\}$ is as follows:

$$\begin{bmatrix} \mathbf{u}_{i_0}^t \\ \mathbf{u}_{i_1}^t \\ \mathbf{u}_{i_2}^t \end{bmatrix} = \begin{bmatrix} \mathbf{u}_{i_0}^{t-1} + \gamma(e_{i_0,j_0}\mathbf{v}_{j_0}^{t-1} - \lambda_U \mathbf{u}_{i_0}^{t-1}) \\ \mathbf{u}_{i_1}^{t-1} + \gamma(e_{i_1,j_1}\mathbf{v}_{j_1}^{t-1} - \lambda_U \mathbf{u}_{i_1}^{t-1}) \\ \mathbf{u}_{i_2}^{t-1} + \gamma(e_{i_2,j_2}\mathbf{v}_{j_2}^{t-1} - \lambda_U \mathbf{u}_{i_2}^{t-1}) \end{bmatrix}. \tag{20}$$

There are two parallelization approaches to update both U and V. We only discuss MSGD updating V in the following sections due to limited space.

Based on the idea of MSGD parallelization designs, we consider some CUDA programming problems as follows:

- Which is the highest efficiency CUDA parallelization approach?
- How can GPU memory accommodate large-scale recommender systems data sets?

5 CUDA PARALLELIZATION DESIGN

In this section, we report two scheduling strategies of CUDA thread in Section 5.1. We introduce CUMSGD coalesced memory access in Section 5.2. We present the details of multi-GPU implementation in Section 5.3.

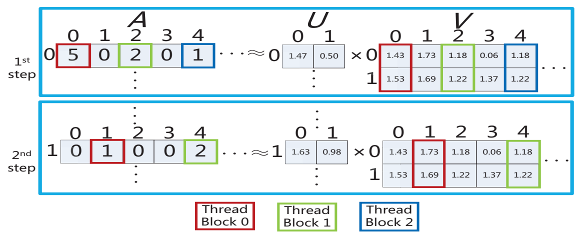


Fig. 5. CUDA decentralized updating V.

5.1 Two Parallelization Approaches

In this section, we show the scheduling strategies of thread block. The *scheduling strategy* is how thread blocks select entries in the rating matrix A. It is necessary to give attention to the difference between the two approaches, e.g., the decentralized and centralized approach. In the decentralized approach, thread blocks select entries in a row of A. In the centralized approach, first, a thread block selects a column index j; then, the thread block selects an entry index (i, j), where $i \in \overline{\Omega}_j$.

Algorithm 2 CUDA decentralized updating V

Input: Initial U and V, rating matrix A, learning rate γ, regularization parameter λ_V, training epoches epo, and total number of thread blocks C.
Output: V.

```
 1: $T \leftarrow$ Thread block $id$.
 2: for $loop$ from $epo$ to 0 do
 3:   for $i = 0, 2, \ldots, m-1$ do
 4:     parallel: % Each thread block $T$ updates its own $\mathbf{v}_j$
        by for loop independently
 5:     Select column index $j \in \Omega_i$.
 6:     Update $\mathbf{v}_j$ by update rule (16).
 7:     end parallel
 8:   end for
 9: end for
10: return $V$.
```

Fig. 5 illustrates a toy example of the decentralized updating V. As shown in Fig. 5, a training epoch is separated into 6 steps. Taking the 1st and 2nd steps as an example, after thread block 0 updates $\mathbf{v}_0$, $0 \in \Omega_0$, thread block 0 selects $\mathbf{v}_1$, $1 \in \Omega_1$, which is not updated in the 2nd step. We observed that a rating matrix has a large number of zero entries, whereas only non-zero rating entries are typically stored on GPU global memory with their indices. The distribution of non-zero entries is random in a rating matrix. CUMSGD cannot guarantee that a thread block selects the same column index on two neighbouring update steps. Meanwhile, the shared memory of a thread block only occupies a small part. Thus, after a thread block updating $\mathbf{v}_j$, the thread block loads $\mathbf{v}_j$ back to GPU global memory for the next update step. Algorithm 2 describes the thread block scheduling strategy of the decentralized approach.

Fig. 6 illustrates a toy example of the centralized updating V. The three different colored boxes on three columns $\{0, 2, 4\}$ in rating matrix A represent that three thread blocks update the corresponding $\{\mathbf{v}_0, \mathbf{v}_2, \mathbf{v}_4\}$ by $\{\{a_{i,0}, \mathbf{u}_i, \mathbf{v}_0, i \in$

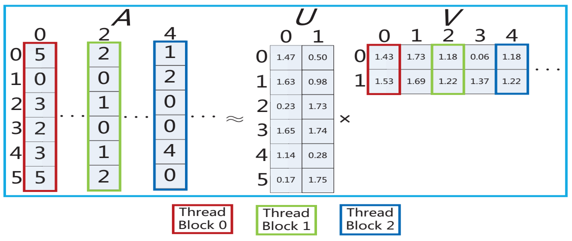


Fig. 6. CUDA centralized updating V.

Algorithm 3 CUDA centralized updating V

```
Input: Initial $U$ and $V$, rating matrix $A$, learning rate $\gamma$, regularization parameter $\lambda_V$, training epoches $epo$, and total number of thread blocks $C$.
Output: $V$.
 1: $T \leftarrow$ Thread block $id$.
 2: for $loop$ from $epo$ to 0 do
 3:   for $j$ from $T$ to $n-1$ on the interval of $C$ do
 4:     parallel: % Each thread block $T$ updates its own $\mathbf{v}_j$ by for loop independently
 5:     Select $i \in \overline{\Omega}_j$.
 6:     Update $\mathbf{v}_j$ by update rule (16).
 7:     end parallel
 8:   end for
 9: end for
10: return $V$.
```

$\overline{\Omega}_0\}, \{a_{i,2}, \mathbf{u}_i, \mathbf{v}_2, i \in \overline{\Omega}_2\}, \{a_{i,4}, \mathbf{u}_i, \mathbf{v}_4, i \in \overline{\Omega}_4\}\}$, respectively. After one training iteration, a thread block will select a $\mathbf{v}_j$ which isn't updated in current training epoch. Algorithm 3 shows thread block scheduling strategy of the centralized update approach.

5.2 Coalesced Memory Access

We set the parameter r as an integral multiple of 32 (warpsize), for *warp* synchronization execution and coalesced global memory access.

Fig. 7 illustrates coalesced access of CUMSGD to U and V in global memory. As shown in Fig. 7, a thread block can access $\mathbf{v}_j, j \in \Omega_i$ continuously due to the coalesced access on GPU global memory via 32-, 64- or 128-byte memory transactions [45]. A *warp* (warpsize=32) accesses the successive 128-bytes in global memory at the same time if the memory access is aligned. For a 128-byte memory transaction, a warp fetches aligned 16 double-type elements from global memory to local memory two times. We set the number of threads per thread block as $r \in \{32, 64, 128, 256, 512\}$, to synchronize *warp* execution.

5.3 Multi-GPU Approach

We extend CUMSGD to MCUMSGD when the size of a data set is larger than the memory capacity of a single GPU. We present multi-GPU data allocation and scheduling strategies, and provide space and time complexity analysis of MCUMSGD on p GPUs.

To improve speedup performance on multi-GPU, we consider how to reduce the task dependence. It is that the rating matrix A is divided into p^2 sub-matrix blocks, then the cyclic update strategy is adopted. It is very different from DSGD cyclic update strategy in Fig. 1, which needs data communication cost between the p nodes. Each GPU updates the item feature sub-matrix, then updates the user feature sub-matrix with emitting item feature sub-matrix to other GPU concurrently, which can hide the communication overhead of loading the item feature sub-matrix. Thus, each GPU needs cache area to store the received item feature sub-matrix. MCUMSGD can operate multi-GPU synchronization and data access instructions by PCIe. Thus, it can extend the

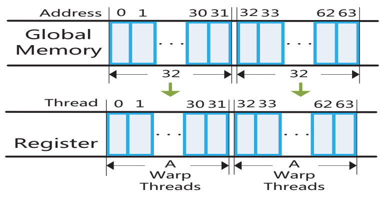


Fig. 7. Coalesced access on V and U in global memory.

Algorithm 4 The basic MCUMSGD approach

```
Input: Initial $U$ and $V$, rating matrix $A$, learning rate $\gamma$, column index matrix $Q$, the number of GPU $p$, training epoches $epo$, and regularization parameters $\lambda_U$, $\lambda_V$.
Output: $U$,$V$.
 1: for $loop$ from $epo$ to 0 do
 2:   for $q \in \{0, 1, \cdots, p-1\}$ do
 3:     Load $\{A^q, U^q, V^q\}$ to GPU $q$.%GPUs Initial workload
 4:   end for
 5:   parallel: % GPU $q$, $q \in \{0, \cdots, p-1\}$
 6:   for $l \in \{0, 1, \cdots, p-1\}$ do
 7:     $j \leftarrow Q_{l,q}$.
 8:     Update $V^j$ by $\{A^{q,j}, U^q, V^j\}$.
 9:     Synchronization on all GPUs.
10:     Update $U^q$ by $\{A^{q,j}, U^q, V^j\}$.
11:     Load $V^j$ to GPU $((q-1+p) \bmod p)$.
12:   end for
13:   end parallel
14: end for
15: return $(U, V)$.
```

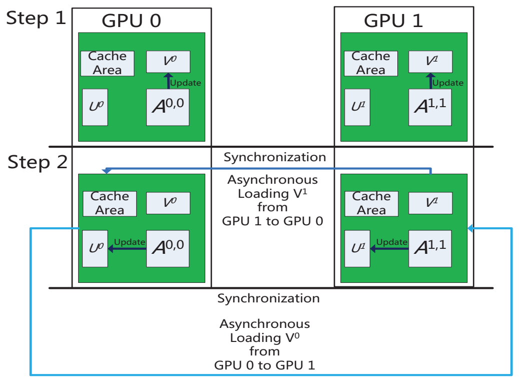

Step 1
GPU 0
GPU 1
Cache Area
Update
Step 2
Synchronization
Asynchronous Loading V¹ from GPU 1 to GPU 0
Synchronization
Asynchronous Loading V⁰ from GPU 0 to GPU 1

Fig. 8. Two GPUs asynchronous cyclic update.

data splitting strategy of DSGD and asynchronous communication mechanism to MCUSGD on multi-GPU.

Given p GPUs, we divide A into p parts $\{A^0, \cdots, A^{p-1}\}$ by row, and the qth part is divided into p sub-parts, $\{A^{q,0}, \cdots, A^{q,p-1}\}$. We define $S = \{S^0, S^1, \ldots, S^{p-1}\}$ as a partition of row indices of U, and we define $G = \{G^0, G^1, \ldots, G^{p-1}\}$ as a partition of column indices of V. We divide U into p parts $\{U^0, U^1, \ldots, U^{p-1}\}$, where U^q is a vector set, and the row indices of U^q correspond to S_q. We divide V into p parts $\{V^0, V^1, \ldots, V^{p-1}\}$, where V^q is a vector set, and the column indices of V^q correspond to G_q. Then, we load $\{A^q, U^q, V^q\}$ to GPU q at initial step.

We define a column index matrix $Q \in \mathbb{R}^{p \times p}$, where

$$Q = \begin{bmatrix} 0 & 1 & \ldots & p\text{-}2 & p\text{-}1 \\ 1 & 2 & \ldots & \text{p-1} & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ p\text{-}2 & p\text{-}1 & \ldots & p\text{-}4 & p\text{-}3 \\ p\text{-}1 & 0 & \ldots & p\text{-}3 & p\text{-}2 \end{bmatrix}.$$

For example, in a training epoch, $(l+1)$th iteration, $l \in \{0, 1, \ldots, p-1\}$, GPU q updates (U^q,V^j), $j = Q_{l,q}$. We now formally define the MCUMSGD (see Algorithm 4). GPU q updates j, $j \in Q_{l,q}$ in its own processing queue (line 7), and updates V^j (line 8). Then, GPU q loads V^j to GPU $\big((q-1+p) \bmod p\big)$, and updates U^q asynchronously (line 10-11). The computation time cost of each GPU is different due to those p GPUs owning different numbers of rating non-zero entries, and the asynchrony problem can be handled by synchronization (line 9). We assume two GPUs $\{GPU\ 0, GPU\ 1\}$. Fig. 8 illustrates that the two GPUs update $\{V^0, V^1\}$ cyclically, and the illustration is as follows:

Step 1. GPU 0 updates V^0 by $\{A^{0,0}, U^0, V^0\}$. GPU 1 updates V^1 by $\{A^{1,1}, U^1, V^1\}$.

Step 2. GPU 0 updates U^0 by $\{A^{0,0}, U^0, V^0\}$ with GPU 0 emitting V^0 to GPU 1 asynchronously. GPU 1 updates U^1 by $\{A^{1,1}, U^1, V^1\}$ with GPU 1 emitting V^1 to GPU 0 asynchronously.

We consider the complexity analysis in the following cases:

Case 1. The rating matrix and two feature matrices $\{A, U, V\}$ are distributed among p GPUs. Each GPU also store a $1/p$ fraction of non-zero entries of rating matrix A. Each GPU has to store a $1/p$ fraction of the user feature matrix U and the item feature matrix V. Because storing a row of U or V requires r space, cache area on each GPU needs $O(nk/p)$ space. The maximum space complexity per GPU is $O\big((mk+2nk+2|\Omega|)/p\big)$. The cost of communication time complexity is $O(nk/p)$. It is difficult to define the time complexity of CUDA kernels due to the execution time of a CUDA kernel depending on many elements, e.g., thread blocks, space complexity, and the two update approaches, as mentioned in Section 4.2.

Case 2. The number of GPUs, p, increases. We distribute $\{A, U, V\}$ across p GPUs. As expected, when p increases, the nk/p decreases more slowly than $|\Omega|/p^2$. Thus, the cost of communication will overwhelm the cost of updating V^j, which will lead to slowdown.

Case 3. MCUMSGD integrates the two approaches mentioned in Section 4.2 to save space cost. We consider two sparse matrix storage formats, e.g., CSR and CSC. CUMSGD can use the user-oriented CSR format for decentralized updating V and centralized updating U, and use item-oriented CSC format for decentralized updating U and centralized updating V. Thus, the minimum space complexity of the hybrid approach is $O\big((mk+2nk+|\Omega|)/p\big)$.

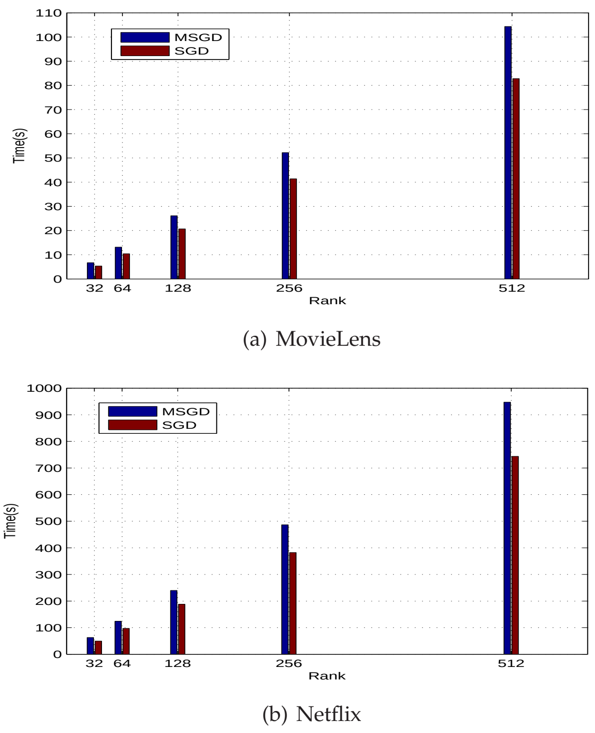


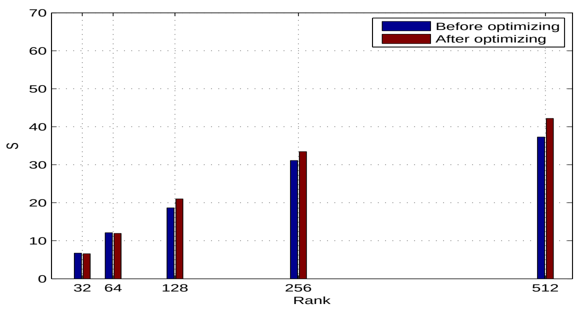
Fig. 9. The time of sequential MSGD versus sequential SGD on CPU.

6 EXPERIMENTS

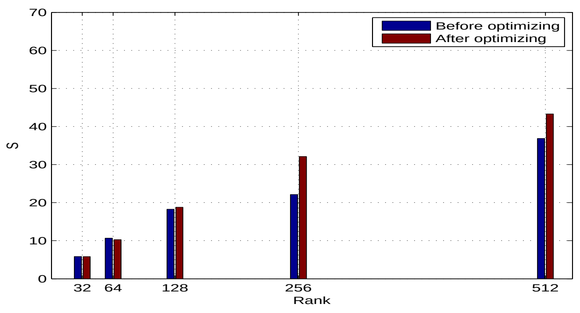
In this section, the experimental results of MSGD and the comparison with state-of-the-art approaches are presented. We provide the details of experimental settings in supplementary material. We compare MSGD and SGD in Section 6.1. We present the speedup performance of CUMSGD in Section 6.2. We compare CUMSGD with state-of-the-art approaches in Section 6.3. Finally, in Section 6.4 and Section 6.5, we answer the two questions mentioned in Section 4.2.

6.1 MSGD versus SGD

We compare the sequential approach of MSGD and SGD on CPU, and the comparison is illustrated in Fig. 9. As shown in Fig. 9, the time of the sequential part increases linearly as the r increases, and MSGD takes longer time than SGD because of the increased fetching times of U and V. Thus, we can conclude that the SGD outperforms the MSGD on one CPU due to less fetching times of U and V.

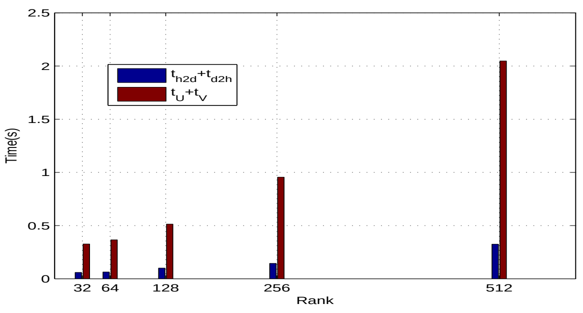


(a) MovieLens

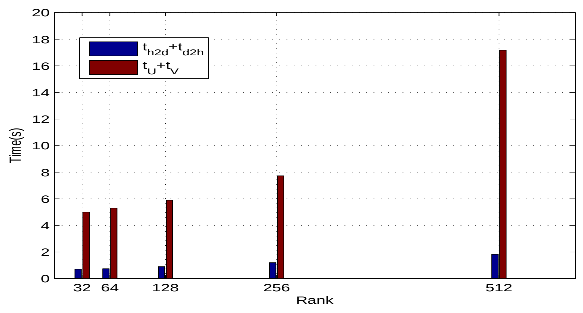


(b) Netflix

Fig. 10. A comparison between the maximum synchronization instruction method (Before optimizing) and the reduced synchronization instruction method (After optimizing).



(a) MovieLens



(b) Netflix

Fig. 11. $t_U + t_V$ versus $t_{h2d} + t_{d2h}$ of CUMSGD.

However, this advantage cannot guarantee the higher performance in the parallel case. Because of SGD parallelization over-writing problem, SGD suffers from low parallelism. Furthermore, we present the accuracy comparison of MSGD and SGD in supplementary material to demonstrate that MSGD can achieve the comparable accuracy than SGD.

6.2 CUMSGD Speedup

In this section, we report the performance of GPU speedup, and give some ideas of how to improve the speedup by optimizing reduction, tuning parameters, e.g., r, the number of thread blocks, and GPU hardware parameters.

6.2.1 *Optimizing Reduction*

Increasing the rank (r) of the feature matrix can improve the accuracy of CF MF to a certain degree [4], [5], in addition to three parameters, e.g., $\{\lambda_U, \lambda_V, \gamma\}$. The time complexity of the sequential multiplication of two vectors is $O(r)$. The two-vector multiplication on CUMSGD only needs $O(\log_2(r))$. Thus, a larger r can improve GPU speedup performance further. Meanwhile, improving the efficiency of vector multiplication and reduction can improve the efficiency of CUMSGD significantly. From CUDA update process described in Algorithms 2 and 3, a thread block that has r threads can update a $\mathbf{v}_j$ only once by $\{e_{i,j}, \mathbf{u}_i, \mathbf{v}_j\}$. The value of $e_{i,j}$ needs the pre-computation of $\mathbf{u}_i\mathbf{v}_j^T$, and vector dot multiplication needs the cooperation of the r threads within the thread block. Thus, the $\mathbf{u}_i$ and $\mathbf{v}_j$ are stored in shared memory of the thread block. Thread synchronization instruction can be saved from $\log_2(r)$ to $\log_2(r) - 5$ by the warp synchronization mechanism presented in [45]. Fig. 10 illustrates the comparison between the maximum synchronization instruction method (before optimizing) and the reduced synchronization instruction method (after optimizing). As shown in Fig. 10, the saved synchronization instruction method can improve speedup after r larger than 128 because the saved cost of warp synchronization is larger than the cost of warp scheduling.

TABLE 1
The *occupancy* of CUMSGD on K20m and K40c GPU.

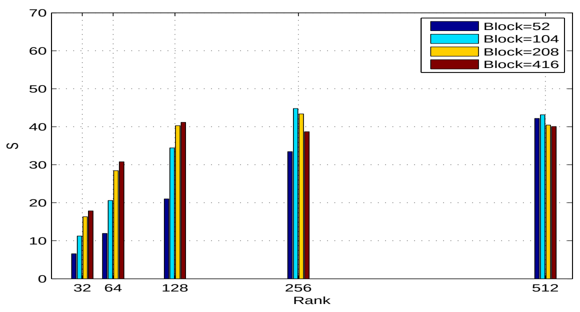
Rank(Threads Per Block)	32	64	128	256	512
Register Per thread	21	21	21	21	21
Shared memory Per block(bytes)	256	512	1024	2048	4096
Active Thread Per SM	512	1024	2048	2048	2048
Active Warps Per SM	16	32	64	64	64
Active Thread Blocks Per SM	16	16	16	8	4
Occupancy of per SM	25%	50%	100%	100%	100%

6.2.2 *Speedup on K20m GPU*

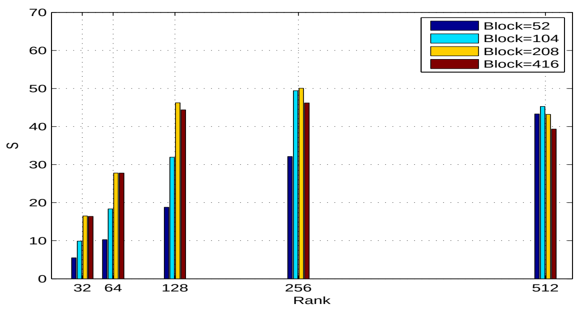
For the comparability of speedup on various state-of-the-art MF parallelization methods, we compare the CUMSGD on a single K20m GPU and the MSGD on a single CPU core.

Fig. 11 illustrates the comparison on $t_U + t_V$ versus $t_{h2d} + t_{d2h}$. As shown in Fig. 11, $t_U + t_V$ is larger than $t_{h2d} + t_{d2h}$. Thus, we can draw the conclusion that the time cost of updating U and V is much longer than the cost of data copies between CPU (*host*) and GPU (*device*). CUMSGD needs more than one training epoch to obtain a reasonable

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2718515, IEEE Transactions on Parallel and Distributed Systems

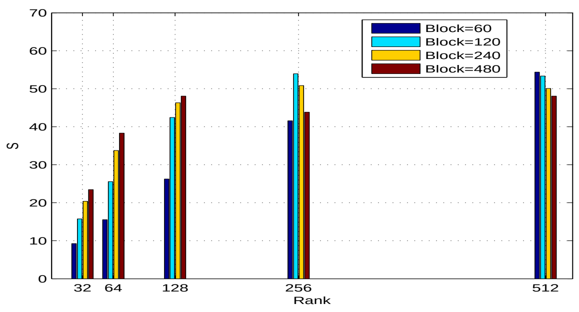


(a) MovieLens

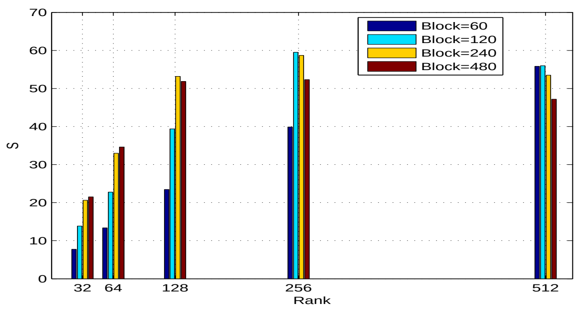


(b) Netflix

Fig. 12. Speedup on K20m GPU. K20m GPU has 13 SMs. We test S on five r, e.g., $r \in \{32, 64, 128, 256, 512\}$. We fix r, and test S on four different number of thread blocks, e.g., $\{52, 104, 208, 416\}$.



(a) MovieLens



(b) Netflix

Fig. 13. Speedup on K40c GPU. K40c GPU has 15 SMs. We test S on five r, e.g., $r \in \{32, 64, 128, 256, 512\}$. We fix r, and test S on four various number of thread blocks, e.g., $\{60, 120, 240, 480\}$.

RMSE, and we suppose that CUMSGD needs *epo* training epoches. Thus, the speedup (S) is approximated by

$$S = \frac{epo \cdot T_1}{epo(t_U + t_V)} = \frac{T_1}{t_U + t_V}. \tag{21}$$

The choice of the appropriate r of feature matrices can guarantee the reasonable accuracy, and spend less training time in CF MF problem [4], [5], [46]. In our work, because we focus on the speedup performance on various r of feature matrices rather than the choice of appropriate r, we test five sets of experiments on various r, e.g., $r \in \{32, 64, 128, 256, 512\}$. GPU *occupancy* is the ratio of the number of active threads to the total number of threads, and high *occupancy* means that the GPU is working in high efficiency. The GPU *occupancy* is calculated by *CUDA Occupancy Calculator* [45]. In CUMSGD, a thread block has r threads, and the number of thread blocks is tunable, which can control the *occupancy*. Table 1 lists *occupancy* under various conditions, e.g., $r \in \{32, 64, 128, 256, 512\}$.

As shown in Fig. 12, in the case of 100% *occupancy*, $r \in \{128, 256, 512\}$ can obtain optimal speedup performance. In the case of 52 thread blocks, the speedup (S) increases as r increases, because the *occupancy* is not full until $r = 512$. In fact, when $r = 512$, the case with 52 thread blocks can obtain a 40x speedup, which is almost the best one in this condition. Table 1 and Fig. 12 demonstrate that the optimal setting of thread blocks to achieve the optimal speedup performance on GPU is *the total number of SMs · active thread blocks per SM*.

6.2.3 *Speedup on K40c GPU*

Scalability is an evaluation indicator in industrial applications [47], [48]. It is difficult to implant prior parallel approach onto GPU because block partitioning prevents the GPU from allocating the optimal number of thread blocks dynamically, which makes the GPU unable to obtain the optimal computing power. CUMSGD removes the dependence on the user-item pair, which enables SGD to acquire high scalability and high parallelism. Fig. 13 illustrates the speedup performance of CUMSGD on K40c GPU. Because of the higher bandwidth, increased memory clock rate and more number of SMs compared with K20m GPU, K40c GPU obtains better speedup performance than K20m GPU. Thus, we firmly confirm that CUMSGD has high scalability and applicability to industrial applications.

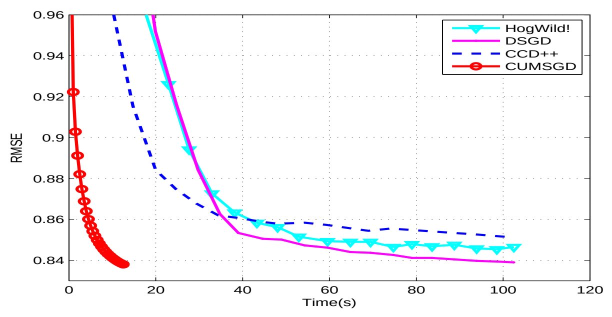
6.3 Comparisons with State-of-the-Art Methods

We compared CUMSGD with various parallel matrix factorization methods on $r = 128$, including two SG-based methods, e.g., DSGD, Hogwild!, and a non-SG-based method, CCD++, in Fig. 14. Fig. 14 illustrates the test RMSE versus training time. Among the three parallel stochastic gradient methods, CUMSGD is faster than DSGD, and Hogwild!. We believe that this result is because MSGD has high parallelism and GPU has high computing power.

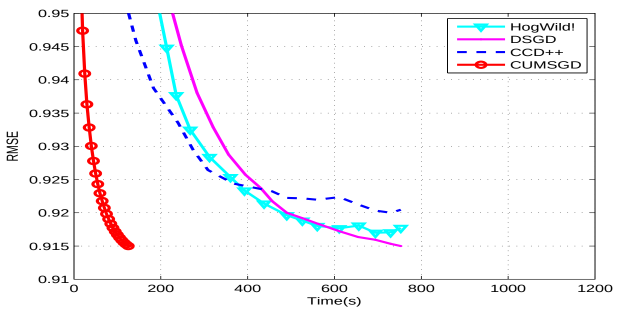
The reasons that we compare RMSE versus computation time on various MF parallelization are as follows:

- In the Hogwild!, each thread selects ratings randomly. It is difficult to identify a full training epoch.
- FPSGD may not have a full training epoch because each thread selects an unprocessed block randomly.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2017.2718515, IEEE Transactions on Parallel and Distributed Systems

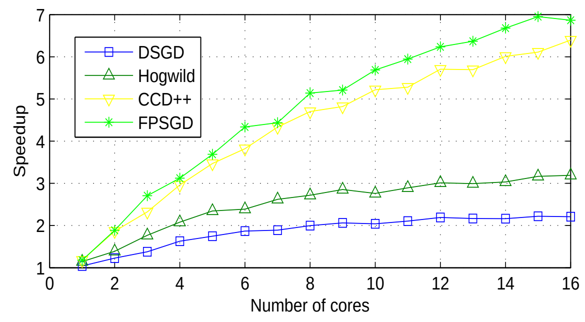


(a) MovieLens

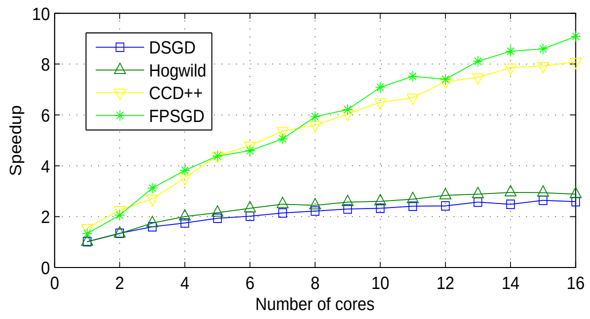


(b) Netflix

Fig. 14. RMSE versus computation time on a 16-core system for Hogwild!, DSGD, and CCD++, and a K20m GPU for CUMSGD(Time in seconds) using double-precision floating point arithmetic.



(a) MovieLens



(b) Netflix

Fig. 15. Speedup of various MF methods on a 16-cores system.

As shown in Fig. 14, CCD++ can obtain the same speed in the beginning and then becomes more stable than DSGD and CUMSGD. We suspect that CCD++ may converge to some local minimum. Fig. 14 shows that Hogwild! perform relatively poorer than DSGD and CUMSGD. We suspect that this result is because Hogwild! randomly select ratings and blocks, and the over-writing problem might decrease the convergence of HogWilg!. SG-based methods can escape local minima, and obtain slightly better test RMSE than the non-SG method, CCD++. The speedup performance on shared memory platform of prior methods, e.g., DSGD, FPSGD, Hogwild! and CCD++, are presented in Fig. 15. We observe that the speedup of the four methods grow slower as the number of cores increase linearly. We present the comparison of CUMSGD and FPSGD in supplementary material.

6.4 User-oriented or Item-oriented Setting?

MSGD updates U and V alternatively. We consider that which one of user-oriented ordering and item-oriented ordering can obtain higher performance on GPU. Recall that, in Sections 4 and 5, the both orderings can be adopted by CUMSGD. We observe that there are four update combinations in an update epoch as follows:

- Updating U on user-oriented ordering (centralized approach) and updating V on user-oriented ordering (decentralized approach).
- Updating U on item-oriented ordering (decentralized approach) and updating V on item-oriented ordering (centralized approach).
- Updating U on item-oriented ordering (decentralized approach) and updating V on user-oriented ordering (decentralized approach).
- Updating U on user-oriented ordering (centralized approach) and updating V on item-oriented ordering (centralized approach).

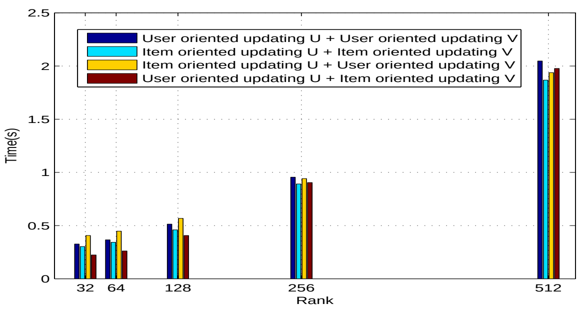
Fig. 16 illustrates the GPU running time of a training epoch for the four combinations. We test the GPU running time on the optimal number of thread blocks mentioned in Section 6.2. We suspect that the reasons for the gaps between the GPU running times of the four approaches are as follows:

- As illustrated in Section 4, decentralized updating V needs more data access times than centralized updating V from GPU global memory.
- The data load of decentralized updating V is more balanced than centralized updating V.

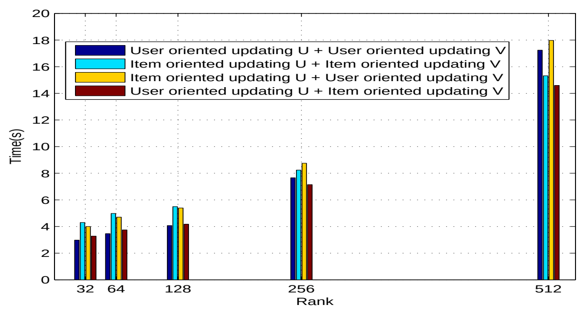
From the experimental results, the combination of user-oriented and item-oriented setting can obtain higher speedup performance than a single user-oriented or item-oriented setting; however, a single setting can save space cost.

6.5 Multi-GPU Implementation

Multi-GPU can solve the problem that large-scale data sets cannot be loaded into a single GPU. The dependence on $\{(A^{i,j}, U^i, V^j)|i, j \in 0, 1, \cdots, p-1\}$ may be the bottleneck of improving the performance of multi-GPU speedup due to huge cost of synchronization and communication within multi-GPU. As shown in Fig. 8, MCUMSGD hides the data access time into updating U, but the synchronization cost

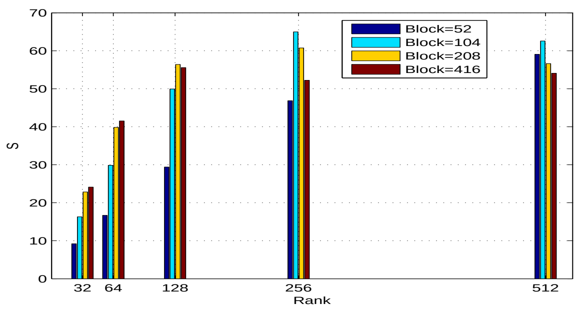


(a) MovieLens

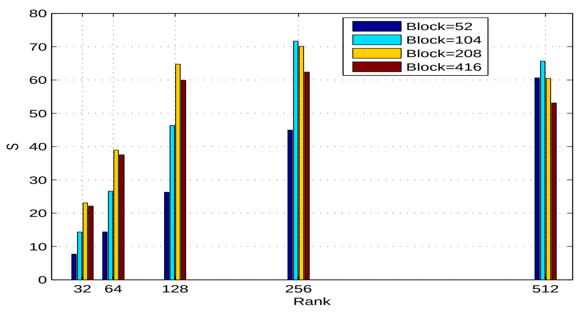


(b) Netflix

Fig. 16. User-oriented update and item-oriented update. User-oriented and item-oriented settings can be combined in four approaches.



(a) MovieLens



(b) Netflix

Fig. 17. Multi-GPU: two GPUs cyclic update.

is still existing. Compared with Fig. 12, the speedup performance on a single K20m GPU, Fig. 17 illustrates the speedup performance on two K20m GPUs. We observe that the two K20m GPUs can obtain a higher speedup performance than on a single K20m GPU, whereas the improvement of speedup performance doesn't increase linearly with the increase of the number of GPUs. We recall the cyclic update process in Section 5.3. We observe that the parallelism is reduced when two GPUs update $\{V^0, V^1\}$ cyclically. Furthermore, the reduced parallelism and unbalanced load of each SMs owing to the irregular distribution of non-zero entries in the rating matrix A, may make some SMs be idle in the update process, which may lead to nonlinear speedup.

7 CONCLUSION AND FUTURE WORKS

In this paper, we propose a CUDA parallelization approach of SGD, which is named MSGD, to accelerate the MF. MSGD removes the dependence on the user-item pair and splits the optimization objective of MF into multiple independent sub-objectives. For large-scale data sets, we propose a multi-GPU strategy to solve large-scale MF problems. From the experimental results, we observe that CUMSGD is faster than FPSGD, DSGD, Hogwild!, and CCD++. CUMSGD can obtain 50x speedup on a K20m GPU, 60x speedup on a K40c GPU, and 70x speedup on two K20m GPUs.

The multi-GPU approach cannot obtain linear speedup. We explain the reason as reduced parallelism and load unbalance. We attempt to solve this problem by the combination of centralized and decentralized update approaches. The centralized approach of CUMSGD is based on evenly dividing rows and columns of sparse rating matrix, and this approach is not necessarily load balance among SMs. Thus, we are interested in using intelligent approach to make an even distribution among SMs. We would like to improve the convergence rate of MSGD, and integrate MSGD with state-of-the-art learning model of CF MF. Online learning or incremental learning is a widely used approach to real-time system [49]. Thus, we would like to expand MSGD to online learning model of CF MF problem.

To solve the large-scale MF problem, we can extend MCUMSGD to heterogeneous CPU-GPU. MSGD has inherent parallelism, so we plan to extend MSGD to parallel and distributed platforms, e.g., OPENMP, MapReduce, and Hadoop.

ACKNOWLEGEMENT

The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant No. 61432005), the National Outstanding Youth Science Program of National Natural Science Foundation of China (Grant No. 61625202), the National Natural Science Foundation of China (Grant Nos. 61370095, 61370097, 61472124, 61572175, and 61672224). The International Science & Technology Cooperation Program of China (Grant Nos. 2015DFA11240, 2014DFB30010), and the National High-tech R&D Program of China (Grant No. 2015AA015305).

REFERENCES

[1] Y. Cai, H.-f. Leung, Q. Li, H. Min, J. Tang, and J. Li, "Typicality-based collaborative filtering recommendation," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 3, pp. 766–779, 2014.
[2] G. Adomavicius and A. Tuzhilin, "Toward the next generation of recommender systems: A survey of the

state-of-the-art and possible extensions," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 6, pp. 734–749, 2005.
[3] N. Srebro, J. Rennie, and T. S. Jaakkola, "Maximum-margin matrix factorization," in *Advances in neural information processing systems*, 2004, pp. 1329–1336.
[4] Y. Koren, "Factor in the neighbors: Scalable and accurate collaborative filtering," *ACM Transactions on Knowledge Discovery from Data (TKDD)*, vol. 4, no. 1, p. 1, 2010.
[5] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, no. 8, pp. 30–37, 2009.
[6] J. Bennett and S. Lanning, "The netflix prize," in *Proceedings of KDD cup and workshop*, vol. 2007, 2007, p. 35.
[7] N. Srebro, "Learning with matrix factorizations," Ph.D. dissertation, Citeseer, 2004.
[8] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management*. Springer, 2008, pp. 337–348.
[9] W. Tan, L. Cao, and L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on gpus," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2016, pp. 219–230.
[10] H.-F. Yu, C.-J. Hsieh, S. Si, and I. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*. IEEE, 2012, pp. 765–774.
[11] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Parallel matrix factorization for recommender systems," *Knowledge and Information Systems*, vol. 41, no. 3, pp. 793–819, 2014.
[12] C.-J. Hsieh and I. S. Dhillon, "Fast coordinate descent methods with variable selection for non-negative matrix factorization," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 1064–1072.
[13] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, 2001, pp. 556–562.
[14] W.-S. Chin, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "A fast parallel stochastic gradient method for matrix factorization in shared memory systems," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 6, no. 1, p. 2, 2015.
[15] ——, "A learning-rate schedule for stochastic gradient methods to matrix factorization," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, 2015, pp. 442–455.
[16] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2011, pp. 69–77.
[17] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *2012 IEEE 12th International Conference on Data Mining (ICDM)*. IEEE, 2012, pp. 655–664.
[18] F. Niu, B. Recht, C. Re, and S. Wright, "Hogwild!: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, 2011, pp. 693–701.
[19] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Application of dimensionality reduction in recommender system-a case study," DTIC Document, Tech. Rep., 2000.
[20] M.-F. Weng and Y.-Y. Chuang, "Collaborative video reindexing via matrix factorization," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 8, no. 2, p. 23, 2012.
[21] Z. Zheng, H. Ma, M. R. Lyu, and I. King, "Collaborative web service qos prediction via neighborhood integrated matrix factorization," *IEEE Transactions on Services Computing*, vol. 6, no. 3, pp. 289–299, 2013.
[22] D. Lian, C. Zhao, X. Xie, G. Sun, E. Chen, and Y. Rui, "Geomf: joint geographical modeling and matrix factorization for point-of-interest recommendation," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2014, pp. 831–840.
[23] J. Langford, M. Zinkevich, and A. J. Smola, "Slow learners are fast," in *Advances in Neural Information Processing Systems*, 2009, pp. 2331–2339.
[24] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *Advances in neural information processing systems*, 2010, pp. 2595–2603.
[25] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems*, 2011, pp. 873–881.
[26] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. Dhillon, "Nomad: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion," *Proceedings of the VLDB Endowment*, vol. 7, no. 11, pp. 975–986, 2014.
[27] J. Jin, S. Lai, S. Hu, J. Lin, and X. Lin, "Gpusgd: A gpu-accelerated stochastic gradient descent algorithm for matrix factorization," *Concurrency and Computation: Practice and Experience*, 2015.
[28] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, 2008.
[29] G. Pratx and L. Xing, "Gpu computing in medical physics: A review," *Medical physics*, vol. 38, no. 5, pp. 2685–2697, 2011.
[30] P. Guo, L. Wang, and P. Chen, "A performance modeling and optimization analysis tool for sparse matrix-vector multiplication on gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1112–1123, 2014.
[31] K. Li, W. Yang, and K. Li, "Performance analysis and optimization for spmv on gpu using probabilistic modeling," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 1, pp. 196–205, 2015.
[32] Z. Gao, Y. Liang, and Y. Jiang, "Implement of item-based recommendation on gpu," in *2012 IEEE 2nd International Conference on Cloud Computing and Intelligent Systems (CCIS)*, vol. 2. IEEE, 2012, pp. 587–590.
[33] K. Kato and T. Hosino, "Singular value decomposition

for collaborative filtering on a gpu," in *IOP Conference Series: Materials Science and Engineering*, vol. 10, no. 1. IOP Publishing, 2010, pp. 012–017.

[34] A. Cano and C. Garcia-Martinez, "100 million dimensions large-scale global optimization using distributed gpu computing," in *2016 IEEE Congress on Evolutionary Computation (CEC)*. IEEE, 2016, pp. 3566–3573.

[35] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, 2015.

[36] J. A. Stuart and J. D. Owens, "Multi-gpu mapreduce on gpu clusters," in *2011 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2011, pp. 1068–1079.

[37] C. Chen, K. Li, A. Ouyang, and K. Li, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," in *2016 IEEE 45th International Conference on Parallel Processing (ICPP)*. IEEE, 2016, pp. 542–551.

[38] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proceedings of COMPSTAT'2010*. Springer, 2010, pp. 177–186.

[39] R. Johnson and T. Zhang, "Accelerating stochastic gradient descent using predictive variance reduction," in *Advances in Neural Information Processing Systems*, 2013, pp. 315–323.

[40] T. Zhang, "Solving large scale linear prediction problems using stochastic gradient descent algorithms," in *Proceedings of the twenty-first international conference on Machine learning*. ACM, 2004, p. 116.

[41] N. L. Roux, M. Schmidt, and F. R. Bach, "A stochastic gradient method with an exponential convergence _rate for finite training sets," in *Advances in Neural Information Processing Systems*, 2012, pp. 2663–2671.

[42] S. J. Reddi, A. Hefny, S. Sra, B. Poczos, and A. Smola, "Stochastic variance reduction for nonconvex optimization," in *Proceedings of The 33rd International Conference on Machine Learning*, 2016, pp. 314–323.

[43] A. Nemirovski, A. Juditsky, G. Lan, and A. Shapiro, "Robust stochastic approximation approach to stochastic programming," *SIAM Journal on optimization*, vol. 19, no. 4, pp. 1574–1609, 2009.

[44] Y. Nesterov, *Introductory lectures on convex optimization: A basic course*. Springer Science & Business Media, 2013, vol. 87.

[45] C. Nvidia, "Nvidia cuda c programming guide," *NVIDIA Corporation*, vol. 120, p. 18, 2011.

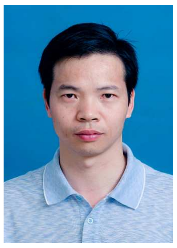
[46] S. D. Babacan, M. Luessi, R. Molina, and A. K. Katsaggelos, "Sparse bayesian methods for low-rank matrix estimation," *IEEE Transactions on Signal Processing*, vol. 60, no. 8, pp. 3964–3977, 2012.

[47] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng, "Stochastic gradient boosted distributed decision trees," in *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 2009, pp. 2061–2064.

[48] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[49] E. Hazan *et al.*, "Introduction to online convex optimization," *Foundations and Trends® in Optimization*, vol. 2, no. 3-4, pp. 157–325, 2016.

Hao Li is currently working toward the Ph.D. degree at Hunan University, China. His research interests are mainly in large-scale sparse matrix and tensor factorization, recommender systems, social network, data mining, machine learning, and GPU and multi-GPU computing.



Kenli Li received the Ph.D. degree in computer science from Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at University of Illinois at Urbana-Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University and deputy director of National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, grid and cloud computing. He has published more than 130 research papers in international conferences and journals such as IEEE-TC, IEEE-TPDS, IEEE-TSP, JPDC, ICPP, CCGrid. He is an outstanding member of CCF. He is a senior member of the IEEE and serves on the editorial board of *IEEE Transactions on Computers*.

Jiyao An received the Ph.D. degree in Mechanical Engineering from Hunan University, China, in 2012. He was a Visiting Scholar with the Department of Applied Mathematics, University of Waterloo, Ontario, Canada. He is currently a full Professor in the College of Computer Science and Electronic Engineering in Hunan University, Changsha, China. His research interests include Cyber-Physical Systems (CPS), Takagi-Sugeno fuzzy systems, Parallel and Distributed Computing, and Computational Intelligence. He has publish more than 50 papers in international and domestic journals and refereed conference papers. He is a member of the IEEE and ACM, and a senior member of CCF. He is an active reviewer of international journals.

Keqin Li is a SUNY Distinguished Professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems. He has published over 480 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Services Computing*, and *IEEE Transactions on Sustainable Computing*. He is an IEEE Fellow.