Locality Sensitive Hash Aggregated Nonlinear Neighborhood Matrix Factorization for Online Sparse Big Data Analysis

ZIXUAN LI, HAO LI, KENLI LI, and FAN WU, College of Computer Science and Electronic Engineering, Hunan University, China
LYDIA CHEN, Department of Electric Engineering, Mathematics and Computer Science, Distributed Systems, Delft University of Technology, Netherlands
KEQIN LI, Department of Computer Science, State University of New York, USA

Matrix factorization (MF) can extract the low-rank features and integrate the information of the data manifold distribution from high-dimensional data, which can consider the nonlinear neighborhood information. Thus, MF has drawn wide attention for low-rank analysis of sparse big data, e.g., Collaborative Filtering (CF) Recommender Systems, Social Networks, and Quality of Service. However, the following two problems exist: (1) huge computational overhead for the construction of the Graph Similarity Matrix (GSM) and (2) huge memory overhead for the intermediate GSM. Therefore, GSM-based MF, e.g., kernel MF, graph regularized MF, and so on, cannot be directly applied to the low-rank analysis of sparse big data on cloud and edge platforms. To solve this intractable problem for sparse big data analysis, we propose Locality Sensitive Hashing (LSH) aggregated MF (LSH-MF), which can solve the following problems: (1) The proposed probabilistic projection strategy of LSH-MF can avoid the construction of the GSM. Furthermore, LSH-MF can satisfy the requirement for the accurate projection of sparse big data. (2) To run LSH-MF for fine-grained parallelization and online learning on GPUs, we also propose CULSH-MF, which works on CUDA parallelization. Experimental results show that CULSH-MF can not only reduce the computational time and memory overhead but also obtain higher accuracy. Compared with deep learning models, CULSH-MF can not only save training time but also achieve the same accuracy performance.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: CUDA parallelization on gpu and multiple GPUs, Graph Similarity Matrix (GSM), Locality Sensitive Hash (LSH), Matrix Factorization (MF), online learning for sparse big data, Top-K nearest neighboors.

The research was partially funded by the National Key R&D Program of China (Grant No. 2020YFB2104000) and the Programs of National Natural Science Foundation of China (Grant Nos. 61860206011, 62172157). This work has been partly funded by the Swiss National Science Foundation NRP75 project (Grant No. 407540_167266) and the China Scholarship Council (CSC) (Grant No. CSC201906130109).
Authors' addresses: Z. Li, H. Li, K. Li (corresponding author), and F. Wu, College of Computer Science and Electronic Engineering, Hunan University, Changsha, China, 410082; emails: {zixuanli, lihao123}@hnu.edu.cn, H.Li-9@tudelft.nl, {lkl, wufan}@hnu.edu.cn; L. Chen, Department of Electric Engineering, Mathematics and Computer Science, Distributed Systems, Delft University of Technology, Delft, Netherlands; email: lydiayChen@ieee.org; K. Li, Department of Computer Science, State University of New York, New Paltz; email: lik@newpaltz.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2022 Association for Computing Machinery.
2577-3224/2022/03-ART37 $15.00
https://doi.org/10.1145/3497749

ACM Reference format:
Zixuan Li, Hao Li, Kenli Li, Fan Wu, Lydia Chen, and Keqin Li. 2022. Locality Sensitive Hash Aggregated Nonlinear Neighborhood Matrix Factorization for Online Sparse Big Data Analysis. *ACM/IMS Trans. Data Sci.* 2, 4, Article 37 (March 2022), 27 pages.
https://doi.org/10.1145/3497749

1 INTRODUCTION

In the era of big data, the data explosion problem has arisen. Thus, a real-time and accurate solution to alleviate information overload on industrial platforms is nontrivial [2]. Big data come from human daily needs, i.e., social relationships, medical data, and recommendation data from e-commerce companies [31]. Moreover, due to the large scale and mutability of spatiotemporal data, sparsity widely exists in big data applications [52]. For accurate big-data processing, representation learning can eliminate redundant information and extract the inherent features of big data, which makes big-data analysis and processing more accurate and efficient [3]. Furthermore, for sparse data from social networks and recommendation systems, low-rank representation learning can extract features as latent variables to represent the node and user properties from the high-dimension space, which can alleviate the information loss owing to missing data [61]. MF is the state-of-the-art unsupervised representation learning model with the same role as **Principal Component Analysis (PCA)** and an autoencoder that can project the high-dimensional space into the low-rank space [9].

Due to its powerful extraction capability for big data, linear and nonlinear dimensionality reduction is widely used as an emerging low-rank representation learning model [4]. As one of the most popular dimensionality reduction models, MF can factorize high-dimensional data into two low-rank factor matrices via the constraints of prior knowledge, i.e., distance metrics and regularization items [19]. Then, MF uses the product of the two low-rank matrices to represent the original high-dimension data, which endows the MF with a strong generalization ability [40]. However, due to the variety of big data, e.g., multiple attributes of images [55], context-aware text information [28], and so on, linear MF is not applicable to an environment with hierarchical information; thus, it should consider the inherent information of big data [1]. Nonlinear MF, e.g., neural MF [60] and the graph for manifold data [34, 46], which relies on the construction of the GSM, can mine deep explicit and implicit information. However, the **Deep Learning (DL)** model for neural MF needs multilayer parameters to extract inherent variables, which can limit the training speed and create huge spatial overhead for constructing a GSM; thus, DL cannot be adopted by industrial big data platforms. Thus, modern industrial platforms are anxious to save parameters in nonlinear MF models [66].

Neighborhood information for nonlinear MF is an emerging topic [23, 67]. The neighborhood model can strengthen the feature representation by capturing the strong relationship points within the data; and this model is popular in Recommendation Systems, Social Networks, and Quality of Service (QoS) [29, 66]. Handling neighborhood information is based on several important neighborhood points that should construct a GSM [51, 66]. However, the use of the GSM should consider the following two problems: (1) the selection and definition of the similarity function should be accurate and (2) the huge time and spatial overhead caused by the construction of the GSM. The first problem can be solved by using DL to select the best similarity [10]. However, the huge computational costs make DL unsuitable for cloud-side platforms. The construction of the GSM takes a huge amount of time and spatial overhead, and its parallelization is difficult. Due to the quadratically increased spatial costs, the second problem is fatal to real applications using

high-dimensional data. In this case, the approximated strategy is considered to replace the calculation of the GSM.

LSH is a statistical estimation technique that is widely used in high-dimensional data for the **Approximate Nearest Neighborhood (ANN)**, and it maps the high-dimensional data to low-dimensional latent space using random projection, which can simplify the approximated search problem into a matching lookup problem [41]. Due to low time complexity, LSH has a fast processing capability for high-dimensional data [65]. Furthermore, LSH has the following drawbacks: (1) the LSH scheme has a slight loss of accuracy; (2) the use of DL can lead to high-precision hashes, but DL is not applicable to cloud-side platforms; (3) online tracking of the hash value for incremental big data; (4) due to information missing, the similarity between sparse data is not very accurate and should be handled by a specific LSH function. Thus, it is nontrivial to achieve a reasonable accuracy in less time with fine-grained parallelization for LSH. Furthermore, with the rapid development of GPU-based cloud-edge computing, increasingly more vendors will tend to use GPU acceleration [37]. There are three challenges to aggregate LSH with nonlinear MF efficiently to extract the deep features of sparse and high-dimensional data: (1) How can a suitable LSH function be defined to reduce the computation time while ensuring reasonable accuracy? (2) How can LSH be accommodated with the nonlinear neighborhood MF to achieve low spatial overhead in an online way? (3) How can a GPU and multiple GPUs be used to achieve a faster calculation speed?

This work is proposed to solve the above problems, and the main contributions are presented as follows:

(1) A novel **Stochastic Gradient Descent (SGD)** algorithm for MF on a GPU (CUSGD++) is proposed. This method can utilize the GPU registers more and disentangle the involved parameters. The experimental results show that it achieves the fastest speed compared to the state-of-the-art algorithms.
(2) simLSH is proposed to replace the GSM and accomplish sparse data encoding. simLSH can greatly reduce the time and memory overheads and improve the overall approximation accuracy. Furthermore, an online method for simLSH is proposed for incremental data.
(3) The proposed CULSH-MF can combine the access optimization on GPU memory of CUSGD++ and the neighborhood information of simLSH for nonlinear MF. Thus, CULSH-MF can complete the training very fast and attain an $8,000\times$ speedup compared to serial algorithms. Furthermore, CULSH-MF can achieve a speedup of $2.0\times$ compared to CUSGD++. Compared with deep learning models, CULSH-MF can achieve the same effect, and CULSH-MF only needs to spend 0.01% of the training time.

In this work, related works and preliminary findings are presented in Sections 2 and 3, respectively. The proposed model for LSH aggregated MF is presented in Section 4. Experiment results are shown in Section 5.

2 RELATED WORK

Owing to the powerful low-rank generalization ability, MF is widely used in various fields of big data processing, i.e., Source Localization [6], Wireless Sensor Networks [58], Network Data Analysis [57], Network Embedding [43], Recommender Systems [24, 25], Hyperspectral Image Classification [64], and Biological Data Analysis [11]. Furthermore, LSH is a powerful hashing tool that can also strengthen the performance of nonlinear dimension reduction, including PCA and MF, for recommendation [16, 39], retrieval [42, 50], and similarity search [68]. Besides, theoretical research in optimization and machine learning communities, e.g., **Maximum Margin Matrix**

Factorization (MMMF) [30, 53], **Nonnegative Matrix Factorization (NMF)** [32, 38], **Probabilistic Matrix Factorization (PMF)** [13, 28, 45], and **Weighted Matrix Factorization (WMF)** [21, 48, 49], also pay considerable attention to MF. The optimization problem for MF is a classic nonconvex problem [8, 17]. An alternative minimization strategy, e.g., **Alternating Least Squares (ALS)** [54], SGD [59, 63], or **Cyclic Coordinate Descent (CCD)** [47], is adopted to solve this nonconvex problem. An efficient big data processing method requires highly efficient hardware and algorithms.

The rapid development and good performance of GPUs also tend to accelerate basic optimization algorithms that consider the global memory access, threads, and thread block synchronization on a GPU. Thus, the parallelization processes of related methods on GPUs have unique specialties. Tan et al. [54] proposed cuALS, which parallelizes ALS on a GPU. Xie et al. [59] proposed cuSGD based on data parallelization. cuSGD [59] achieves the goal of acceleration by adopting data parallelization on a GPU, and it has no load imbalance problem. Nisa et al. [47] optimized the CCD algorithm and proposed the GPU-based CCD++ algorithm. Li et al. [33, 35] proposed CUSNMF based on feature tuple multiplication and summation and CUMSGD based on the elimination of row and column dependencies. These basic algorithms have good performance on a GPU. However, scalability is not considered, which results in significant limitations of model compatibility. Nonlinear MF comprises two components, i.e., a DL model for neural MF [60] and a neighborhood model with GSM for graph MF [14, 29]. He et al. [18] proposed **Neural Collaborative Filtering (NCF)** using the DL model, and this model involves a multilayer neural network that can extract the low-rank feature of MF [60]. The neighborhood model is often integrated into the algorithm and brings better results [14, 29].

The construction of a GSM requires calculating the similarity between high-dimensional points, the choice of similarity functions play a key role in specific environments, and the selection of the Top-K nearest neighbors from the GSM is time consuming [26]. However, designing an effective similarity function is a difficult task. Research on training similarities through DL is emerging [15]. However, high-dimensional data cause the computational complexity of DL to dramatically increase. To further optimize the calculation and save space, pruning strategies and approximation algorithms have been proposed [12]. LSH is such an approximate algorithm based on probability projection [44]. Furthermore, the inverse use of LSH can also achieve the farthest neighbor search [63]. However, most LSH algorithms do not work well in sparse data environments. minLSH is able to calculate the similarity between sets, but does not consider the weights of the elements in the set. Although a considerable amount of work has sought to improve minLSH, this work increases the complexity [56]. simHash [44] showed good performance in similar text detection. LSH can project the feature vectors of similar items to equal hash values with a high probability [20], and this makes LSH widely used for nearest neighbor searches, fast high-dimensional information searches, and similarity connections [36, 62]. Due to the inherent sparsity of big data, using LSH to construct a GSM to aggregate sparse MF on a big data platform is nontrivial work. Furthermore, the accuracy of the low-rank tracking of online learning for incremental big data is a key problem [27]. Chen et al. proposed an online hash for incremental data [7]. However, there is a lack of an online LSH strategy for sparse and online data on parallel and distributed platforms.

3 PRELIMINARIES

In this section, LSH for neighboring points with closer projective hash values is presented in Section 3.1. The basic MF model and nonlinear MF with notations are introduced in Section 3.2, and the related symbols are listed in Table 1.

Table 1. Table of Symbols

Symbol	Definition
I, J	Two variable sets with interaction;
$\mathbf{R}$	Input sparse matrix $\in \mathbb{R}^{M\times N}$;
$\widehat{\mathbf{R}}$	Low-rank approximated matrix $\in \mathbb{R}^{M\times N}$;
$r_{i,j}$	(i, j)th element in matrix $\mathbf{R}$;
Ω	The set (i, j) of non-zero value in matrix $\mathbf{R}$;
Ω_i	The set j of non-zero value in matrix $\mathbf{R}$ for variable I_i;
$\widehat{\Omega}_j$	The set i of non-zero value in matrix $\mathbf{R}$ for variable J_j;
$\mathbf{U}/u_i$	Left low-rank feature matrix $\in \mathbb{R}^{M\times F}$/ ith row;
$\mathbf{V}/v_j$	Right low-rank feature matrix $\in \mathbb{R}^{N\times F}$/ jth row;
μ	The overall relation between variable set I and variable set J;
b_i	The deviation between variable $I_i \in I$ and μ;
$\widehat{b}_j$	The deviation between variable $J_j \in J$ and μ;
$\overline{b}_{i,j}$	Overall baseline rating = $\mu + b_i + \widehat{b}_j$;
n_{j_1,j_2}	The number of entries in variable set I which have relations with Variables $\{j_1, j_2\}$ in variable set J;
ρ_{j_1,j_2}	Pearson similarity of two variables $\{j_1, j_2\} \in J$;
S_{j_1,j_2}	GSM $\stackrel{def}{=} \frac{n_{j_1,j_2}}{n_{j_1,j_2}+\lambda_\rho}\rho_{j_1,j_2}$;
$R(i)$	The set of variables $\in J$ that are explicitly related to the variable $I_i \in I$;
$N(i)$	The set of variables $\in J$ that are implicitly related to the variable $I_i \in I$;
$S^K(j)$	Top-K Nearest Neighbors variables set of the variable $J_j \in J$;
J^K	The Top-K Nearest Neighbors Matrix $\mathbf{J}^K \in \mathbb{R}^{N\times K}$;
$R^K(i; j)$	$= R(i) \bigcap S^K(j)$;
$N^K(i; j)$	$= N(i) \bigcap S^K(j)$;
$\mathbf{W}$	Explicit influence matrix $\in \mathbb{R}^{N\times K}$ to represent the degree of explicit Influence for variable set J;
$\mathbf{C}$	Implicit influence matrix $\in \mathbb{R}^{N\times K}$ to represent the degree of implicit Influence for variable set J;
$w_j/w_{j,k_1}$	jth Explicit influence vector $\in \mathbb{R}^K$ of $\mathbf{W}$ / the k_1th element of w_j;
$c_j/c_{j,k_2}$	jth Implicit influence $\in \mathbb{R}^K$ of $\mathbf{C}$ / the k_2th element of c_j;
$\overline{I}, \overline{J}$	The new variable sets in online learning;
$\widehat{I}, \widehat{J}$	Combination of new variable sets and original variable sets in online learning.

3.1 GSM And LSH

Definition 3.1 (Graph Similarity Matrix GSM). We assume two sets as $I = \{I_1, \ldots, I_i, \ldots, I_M\}$ and $J = \{J_1, \ldots, J_j, \ldots, J_N\}$. Given two variables $\{J_{j_1}, J_{j_2}\} \in J$ and a similarity function $\mathcal{S}(j_1||j_2)$, the goal is to construct a weighted fully directed graph $\mathbf{G}^J$, where each vertex represents a variable in J, and the weight of each edge represents the similarity of the output vertex to the input vertex calculated by $\mathcal{S}(j_1||j_2)$. The construction of GSM $\mathbf{G}^J$ should consider the relation between J and I. The value of $\mathbf{G}^J_{j_1,j_2}$ relies on $\{\{r_{i,j_1}|i \in \widehat{\Omega}_{j_1}\}, \{r_{i,j_2}|i \in \widehat{\Omega}_{j_2}\}\}$.

The neighborhood similarity query for variable set J relies on the GSM $\mathbf{G}^J \in \mathbb{R}^{N\times N}$ [20, 29, 66]. The most important problem in the neighborhood model is to find a set of Top-K similar variables. For this problem, the Top-K nearest neighbors query is emerging.

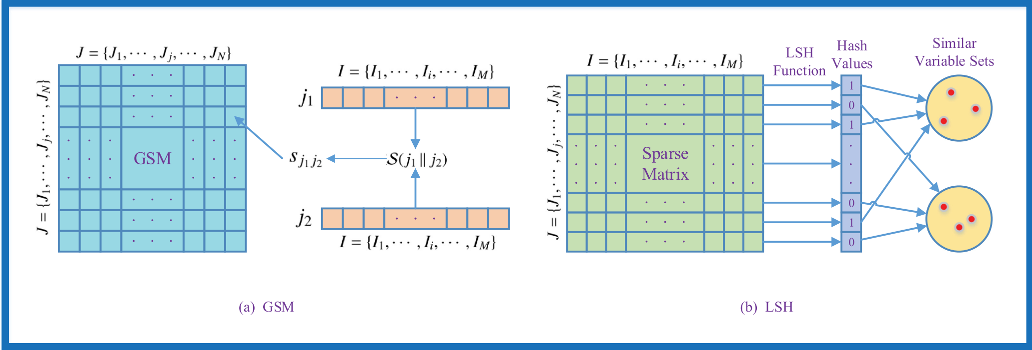

Fig. 1. Comparison of computational complexity and space complexity between GSM and LSH.

Definition 3.2 (Top-K Nearest Neighbors). Given a set of variables **S**, each variable as a vertex constitutes a fully directed graph **G**. The goal is to find a subgraph $\mathbf{S}^K$ where each vertex has **K** and only **K** out edges point to the vertices of its Top-**K** similar variables.

By querying the GSM, the Top-K nearest neighbors can be obtained. However, for a large set of variables, the cost of the GSM is huge. If variable set J has N elements, then the computational complexity is $O(N(N-1))$. Furthermore, the overhead for the Top-K nearest neighbors query of a variable J_j is $O(2NK - K^2 + K)$, and the overhead of Top-K nearest neighbors for the variable set J and the construction of the matrix $\mathbf{J}^K \in \mathbb{R}^{N \times K}$ is $O(2N^2K - NK^2 + NK)$. The overall overhead is $O(N^2(2K+1) + N(K - K^2 - 1))$, and the spatial overhead is $O(NK)$. Thus, the construction of a GSM using high-dimensional sparse big data is not advisable. In the context of high-dimensional sparse big data, the calculation costs of a GSM are squared. In this case, we need to reduce unnecessary calculations or find an alternative method. LSH is a probabilistic projection method that projects two similar variables with a high probability to the same hash value while two dissimilar variables are projected to different hash values with a high probability. We need to judge the similarity between the two variables and find the Top-K nearest neighbors for each variable.

Definition 3.3 (Locality Sensitive Hash LSH). The LSH function is a hash function that satisfies the following two points:

- For any points x and y in $\mathbb{R}^d$ that are close to each other, there is a high probability P_1 that they are mapped to the same hash value $P_H[h(x) = h(y)] \geqslant P_1$ for $||x - y|| \leqslant R_1$; and
- For any points x and y in $\mathbb{R}^d$ that are far apart, there is a low probability $P_2 < P_1$ that they are mapped to the same hash value $P_H[h(x) = h(y)] \leqslant P_2$ for $||x - y|| \geqslant cR_1 = R_2$.

The use of LSH has allowed us to reduce the complexity from $O(N^2)$ to $O(N)$.

As Figure 1 shows, the construction of a GSM requires $O(N^2)$ similarity calculations and consumes $O(N^2)$ space while the calculation and spatial consumption of LSH is $O(N)$.

LSH can alleviate the problem of huge computational overhead. However, there are several problems when the LSH is applied to a system with a neighborhood model: (1) How can a system with a neighborhood model using LSH obtain the same overall accuracy as the original method? (2) How can the computational model for LSH be incorporated in a big data processing system? (3) How can the system with the LSH model accommodate online learning for incremental data?

3.2 Nonlinear Matrix Factorization Model

In big data analysis communities, representation learning can disentangle the explicit and implicit information behind the data, and the low-rank representation problem is presented as follows:

Definition 3.4 (Representation Learning for Sparse Matrix [3]). Assume a sparse matrix $\mathbf{R} \in \mathbb{R}^{M\times N}$ presents the relationship of two variable sets $\{I, J\}$. The value $r_{i,j}$ represents the relation degree of the variables $\{I_i\}$ in I and $\{J_j\}$ in J. Due to missing information, the representation learning task for variable $\{I_i\}$ trains the feature vector u_i relying on nonzero values $\{r_{i,j}|j \in \Omega_i\}$, and the representation learning task for variable $\{J_j\}$ is to train the feature vector v_j relying on nonzero values $\{r_{i,j}|i \in \widehat{\Omega}_j\}$.

Definition 3.5 (Sparse Matrix Low-rank Approximation). Assume a sparse matrix $\mathbf{R} \in \mathbb{R}^{M\times N}$ and a divergence function $\mathcal{D}(\mathbf{R}\|\widehat{\mathbf{R}})$ that evaluates the distance between two matrices. The purpose of the low-rank approximation is to find an optimal low-rank matrix $\widehat{\mathbf{R}}$ and then minimize the divergence.

MF only involves low-rank feature matrices, and the feature vectors are used for cluster and social community detection [9]. A sparse matrix has only a few elements that are valuable, and all other elements are zero. Sparse MF is applied to this problem because it factorizes the sparse matrix into two low-rank feature matrices. In addition, MF model has two limitations: (1) this model is too shallow to capture more affluent features, and (2) this model cannot capture dynamic features.

The approximation value $\widehat{r}_{i,j}$ of the nonlinear matrix factorization model [29] is presented as:

$$\widehat{r}_{i,j} = \underbrace{\overline{b}_{i,j}}_{①} + \underbrace{\left|R^K(i;j)\right|^{-\frac{1}{2}} \sum_{J_{j_1} \in R^K(i;j)} (r_{i,j_1} - \overline{b}_{i,j_1}) w_{j,j_1}}_{②} + \underbrace{\left|N^K(i;j)\right|^{-\frac{1}{2}} \sum_{J_{j_2} \in N^K(i;j)} c_{j,j_2}}_{③} + \underbrace{u_i v_j^T}_{④}. \quad (1)$$

There are four parts in Equation (1), and those parameters can combine the explicit and implicit information of the neighborhood for nonlinear MF, which are introduced as follows [20, 29, 66]:

① $\{\mu, b_i, \widehat{b}_j, \overline{b}_{i,j}\}$: The baseline score is represented as $\overline{b}_{i,j} = \mu + b_i + \widehat{b}_j$ for the relation of variable $I_i \in I$ and variable J_j in set J. Considering that different variables $I_i \in I$ have their own different preferences for the entire variable set J, different variables $J_j \in J$ have their own different preferences for the entire variable set I. To simplify the description, suppose μ is the overall relation between variable set I and variable set J; b_i represents the deviation between variable $I_i \in I$ and μ, which indicates the preference of variable I_i to variable set J; and $\widehat{b}_j$ represents the deviation between variable $J_j \in J$ and μ, which indicates the preference of variable J_j to variable set I. A simple case is presented as: $\mu = \sum_{(i,j)\in\Omega} r_{i,j}/|\Omega|$ (the average relation of the known elements), $b_i = \sum_{j\in\Omega_i} r_{i,j}/|\Omega_i| - \mu$ (the difference between the average relation of the known elements in I_i and μ), and $\widehat{b}_j = \sum_{i\in\widehat{\Omega}_j} r_{i,j}/|\widehat{\Omega}_j| - \mu$ (the difference between the average relation of the known elements in J_j and μ).

$\{n_{j_1,j_2}, S_{j_1,j_2}, S^K(j), R(i), R^K(i;j), w_j\}$: Suppose that J_{j_1} and J_{j_2} are any two variables in J, and $n_{j_1,j_2} = |\widehat{\Omega}_{j_1} \bigcap \widehat{\Omega}_{j_2}|$ is the number of variables $\in I$, both of which are related to variables $\{J_{j_1}, J_{j_2}\}$ $\in J$. ρ_{j_1,j_2} is the Pearson similarity for variables $\{J_{j_1}, J_{j_2}\} \in J$ as a baseline. The (j_1, j_2)th element of GSM is defined as $S_{j_1,j_2} \overset{def}{=} \frac{n_{j_1,j_2}}{n_{j_1,j_2}+\lambda_\rho}\rho_{j_1,j_2}$, where λ_ρ is the regularization parameter that adjusts the importance. By searching for the GSM, the Top-K nearest neighbors variable set $S^K(j)$ of the variable $J_j \in J$ can be obtained. To retain the generalizability, $R(i)$ is denoted as the variable subset of J with explicit relation with variable $I_i \in I$, which contains all the variables for which ratings

by I_i are available. If variable $\{J_{j_1}\} \in R^K(i;j) = R(i) \bigcap S^K(j)$, then variable $I_i \in I$ has more explicit relations with variable J_{j_1}. We parameterize the above explicit relations. Feature vectors $w_j \in \mathbb{R}^K$ are used as the explicit factors for the Top-K nearest neighbors $S^K(j)$ of variable J_j. w_{j,j_1} is used to represent the information gain that variable $J_{j_1} \in R^K(i;j)$ explicitly brings to $J_j \in J$. The closer the basic predicted value $\overline{b}_{i,j}$ is to the true value $r_{i,j}$, the lower the impact received. Therefore, the residual $(r_{i,j_1} - b_{i,j_1})$ is used as the coefficient of w_{j,j_1}. Combining all $(r_{i,j_1} - b_{i,j_1})w_{j,j_1}, J_{j_1} \in R^K(i;j)$ and multiplying the result by a scaling factor $|R^K(i;j)|^{-\frac{1}{2}}$, we obtain $|R^K(i;j)|^{-\frac{1}{2}} \sum_{J_{j_1} \in R^K(i;j)} (r_{i,j_1} - \overline{b}_{i,j_1})w_{j,j_1}$.

③ $\{N(i), N^K(i;j), c_j\}$: To retain the generalizability, $N(i)$ is denoted as the variable subset of J with an implicit relation with the variable $I_i \in I$, and it is not limited to a certain type of implicit data. If $\{J_{j_2}\} \in N^K(i;j) = N(i) \bigcap S^K(j)$, then the variable $I_i \in I$ has more implicit relations with variable J_{j_2}. We parameterize the above implicit relations. Feature vectors $c_j \in \mathbb{R}^K$ are used as the implicit factors for the Top-K nearest neighbors $S^K(j)$ of a variable J_j. c_{j,j_2} is used to represent the information gain that variable $J_{j_2} \in N^K(i;j)$ implicitly brings to variable $J_j \in J$. Combining all $c_{j,j_1}, J_{j_1} \in N^K(i;j)$ and multiplying the result by a scaling factor $|N^K(i;j)|^{-\frac{1}{2}}$, we obtain $|N^K(i;j)|^{-\frac{1}{2}} \sum_{J_{j_1} \in N^K(i;j)} c_{j,j_1}$.

④ $\{u_i, v_j\}$: Original MF model. u_i is the low-rank feature vector for variable $I_i \in I$, and v_j is the low-rank feature vector for variable $J_j \in J$.

With the neighborhood consideration and L_2 norm constraints for the parameters $\{\mathbf{U}, \mathbf{V}, \mu, b_i, \widehat{b}_j, w_j, c_j\}$, the optimization objective is presented as:

$$\arg \min_{\mathbf{U},\mathbf{V},\mu,b_i,\widehat{b}_j,w_j,c_j} \mathcal{D}\left(\mathbf{R} \| \widehat{\mathbf{R}}\right) = \sum_{(i,j) \in \Omega} (r_{i,j} - \widehat{r}_{i,j})^2 + \lambda_b \sum_{i=1}^{M} b_i^2 + \lambda_{\widehat{b}} \sum_{j=1}^{N} \widehat{b}_j^2$$
$$+ \lambda_w \sum_{j=1}^{N} \sum_{J_{j_1} \in R^K(i;j)} w_{j,j_1}^2 + \lambda_c \sum_{j=1}^{N} \sum_{J_{j_1} \in N^K(i;j)} c_{j,j_2}^2 \tag{2}$$
$$+ \lambda_u \sum_{i=1}^{M} ||u_i||^2 + \lambda_v \sum_{j=1}^{N} \left\| v_j \right\|^2,$$

where $\{\lambda_b, \lambda_{\widehat{b}}, \lambda_w, \lambda_c, \lambda_u, and \lambda_v\}$ are the corresponding regularization parameters.

There are two improvements: (1) the neighborhood influences are inherent in some big data applications [1, 22, 67], and (2) the Top-K nearest neighborhood with explicit and implicit information can replace all queries of neighborhood points [20, 29, 66].

4 ONLINE LSH AGGREGATED SPARSE MF ON GPU AND MULTIPLE GPUS

Figure 2 illustrates the structure of this work. First, we consider the interaction value of variable I_i in variable set I and variable J_j in variable set J and generate the interaction matrix $\mathbf{R}$ from this. Second, the original method, which is based on the GSM, can calculate the similarity of every two variables J_{j_1} and J_{j_2} in variable set J to generate a similarity graph $\mathbf{G}^J$; and querying $\mathbf{G}^J$ to obtain the subgraph $\mathbf{S}^K$ can hold the Top-K nearest neighbors of each variable $J_j \in J$. The difference is that the simLSH method we proposed constructs a hash table through p coarse-grained hashings and q fine-grained hashings. Then, we obtain the subgraph $\mathbf{S}^K$ through the hash table. Finally, we train the feature vectors using the updating rule (5).

As Figure 2 shows, this work should consider the following three parts: (1) Interaction matrix $\mathbf{R}$ of two variable sets $\{I, J\}$, which should consider the incremental data and add the coupling

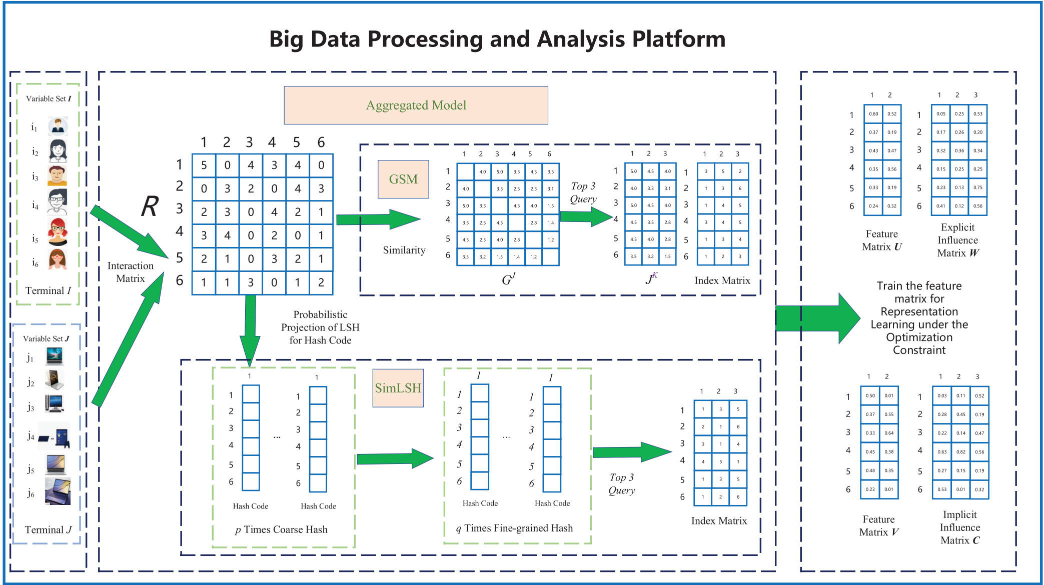

Fig. 2. LSH aggregated sparse MF on big data analysis platform.

ability of the overall system. (2) The construction of a neighborhood relationship should reduce the overall space and computational overhead and maintain the overall accuracy. (3) Training the representation feature vectors in a low computational and high accuracy way. The above objectives guide this section. In this section, LSH for sparse big data and CUDA parallelization are presented in Section 4.1; and then stochastic optimization strategy, CUDA parallelization, and multiple GPUs for sparse big data are presented in Section 4.2, Finally, the online learning solution is presented in Section 4.3.

4.1 LSH And CUDA Parallelization

The Top-K nearest neighbors, which relies on the construction of the GSM, is a key step in the nonlinear neighborhood model. However, the GSM requires a huge amount of calculations, and the time complexity is $O(N^2)$ based on the Pearson similarity. A variety of LSH functions are not friendly to sparse data, because the accuracy of most distance measures will be greatly reduced. This is caused by there being very few positions where the nonzero elements of each vector are the same. The Jaccard similarity is suitable for sparse data, and its representative algorithm is minHash [5]; however, this method only considers the existence of the elements and neglects the real value. To solve this problem, simLSH, which is inspired by simHash applied to text data, is proposed for sparse dig data projection [44]. This method balances the existence of the elements and the value of the elements and maintains low computational complexity. simLSH can effectively combine the number of interactions of variable sets $\{I, J\}$ with the degree of interaction, and simLSH can improve the accuracy while reducing the computational complexity. simLSH is composed of the following two parts:

(1) **Coding for Sparse Big Data:** simLSH randomly generates G-bits $\{0, 1\}$ string H_i for each variable $I_i \in I$, which is equivalent to a simple hash value. The hash value $\overline{H}_j$ for each variable $J_j \in J$ that we need is calculated by H_i and $r_{i,j}$, $i \in \overline{\Omega}_j$. Obviously, the hash value $\overline{H}_j$ should also be

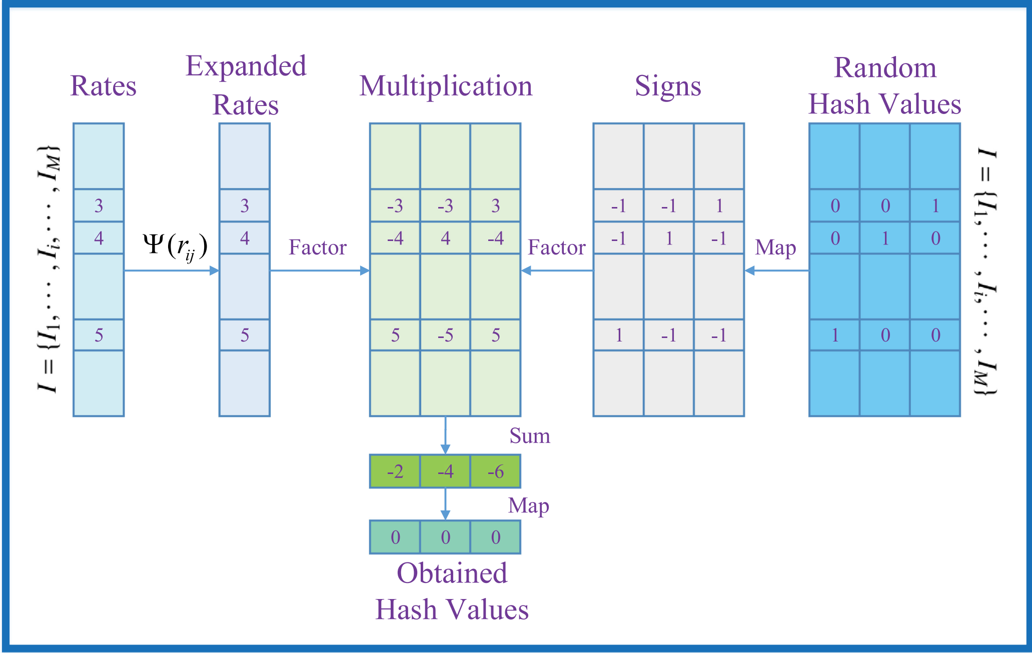

Fig. 3. An example of simLSH.

a G-bits $\{0, 1\}$ string. After the hash value $\overline{H}_j$ for variable $J_j \in J$ is calculated, we obtain $\overline{H}_{j,g} \in \overline{H}_j$ by accumulating $\Phi(H_{i,g}) \cdot \Psi(r_{i,j})$, $i \in \overline{\Omega}_j$. $\Psi(r_{i,j})$ is a function such that there is a suitable interval between different $r_{i,j}$s, and $\Phi(H_{i,g})$ is a function that maps $H_{i,g}$ from $\{0, 1\}$ to $\{-1, +1\}$. Finally, $\Upsilon()$ maps the nonnegative value of $\overline{H}_{j,g}$ to $\{1\}$ and the negative value to $\{0\}$. Then, the G-bit $\{0, 1\}$ string $\overline{H}_j$ is obtained. The entire process of simLSH can be expressed as:

$$\overline{H}_j = \Upsilon\left(\sum_{i \in \widehat{\Omega}_j} \Psi(r_{i,j})\Phi(H_i)\right). \tag{3}$$

As Figure 3 shows, variable J_j has three relation values $r_{i,j}$ $\{3, 4, 5\}$ with $\{i_1, i_2, i_3\} \in \overline{\Omega}_j$. When $G = 3$, $\{H_{i_1}, H_{i_2}, H_{i_3}\}$ are randomly assigned to $\{001, 010, 100\}$, respectively. It takes $\Psi(r_{i,j}) = r_{i,j}$ by calculating $\{(-3-4+5), (-3+4-5), (3-4-5)\}$; and then, the G positions $\{-2, -4, -6\}$ of $\overline{H}_j$ are obtained, respectively. Finally, we obtain the G-bit $\{0, 1\}$ string $\overline{H}_j$ $\{0, 0, 0\}$ by mapping operations.

(2) **Coarse-grained and Fine-grained Hashing:** LSH is an approximation method to estimate the GSM, but it will achieve accuracy losses when applied to sparse big data. In this case, simLSH is proposed to speed up the calculations and improve the accuracy.

Since the maximum probability of two extremely dissimilar variables $\{J_{j_1}, J_{j_2}\}$ with the same hash value is P_2, the mapping of a hash function does not guarantee that the variables $\{J_{j_1}, J_{j_2}\}$ with the same hash value are similar. To alleviate this situation, the multiple random mapping strategy is considered as follows: (1) Coarse-grained Hashing: Similar variables with the same hash values of all mappings are considered. If p random mappings are conducted, where $p \ll N$, then the probability of two dissimilar variables projected as similar pairs is reduced to at most P_2^p. Furthermore, the probability of two similar variables projected as similar pairs is also reduced to at

ALGORITHM 1: CULSH

Input: Sparse matrix $\mathbf{R}$ of variable sets $\{I, J\}$, Random Hash values H_i.
Output: The Top-K Nearest Neighbors Matrix $\mathbf{J}^K \in \mathbb{R}^{N\times K}$. Each row represents the Top-K Nearest Neighbors of a variable $J_j \in J$.

1: **for (Fine-grained Hashing)**: q times Coarse-grained Hashing **do**
2: **for (Coarse-grained Hashing)**: p times simLSH **do**
3: **for (parallel)**: Variables $J_j \in J$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
4: Calculate the hash value $\overline{H}_j$ by Equation (3) for variable $J_j \in J$.
5: **end for**
6: **end for**
7: Count the similar variable pairs with the same hash value in p times simLSH.
8: **end for**
9: Count the similar variable pairs that appear one or more times in q coarse-grained hashings.
10: **for (parallel)**: Variables $J_j \in J$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
11: Search the Top-K nearest neighbors $\{J_{j_1}, \ldots, J_{j_K}\}$ of the variable $J_j \in J$.
12: **end for**

least P_1^p. Under this condition, many similar variable pairs will be missed. (2) Fine-grained Hashing: In this strategy, as long as at least one of the two variables $\{J_{j_1}, J_{j_2}\}$ projected as similar pairs is subjected to coarse-grained hashing, the similar variable pairs $\{J_{j_1}, J_{j_2}\}$ are selected. Suppose that q coarse-grained hashings are conducted. The probability of two similar variables $\{J_{j_1}, J_{j_2}\}$ projected as similar pairs is increased to at least $1 - (1 - P_1^p)^q$. By increasing the values of p and q, the probability that two similar pairs of variables $\{J_{j_1}, J_{j_2}\}$ are projected as similar pairs is increased. This method can improve the probability, and its calculation amount is $p \times q$ times of that of simLSH. We need to adjust the sizes of p and q. Before the model training, we only need to perform multiple simLSHs on N variables to find similar variable pairs, which can reduce the computational complexity to $O(N)$. Even if you use $p \times q$ simLSHs to increase the probability, the computational complexity is only $p \times q \times N$, and $p \times q \times N$ is much smaller than N^2.

Our goal is to find the Top-K nearest neighbors for each variable $J_j \in J$. simLSH does not directly obtain the Top-K nearest neighbors for J_j. It is accomplished by searching for other variables with the same hash value in the hash table. We use the coarse-grained and fine-grained hashing of simLSH and select the K most frequent variables $\{J_1, \ldots, J_K\} \in J$ in the hash table of variable J_j and make a random supplement if the number is less than K. On the CUDA platform, each thread block for simLSH (CULSH) manages a variable J_j. CULSH is described in Algorithm 1 as follows: (1) Lines 1–9: The calculation of simLSH with coarse-grained hashing and fine-grained hashing. In lines 3–5, calculate the hash value $\overline{H}_j$ for variable $J_j \in J$ in parallel and save it, and this only consumes a small amount of memory. (2) Lines 10–12: Search the Top-K nearest neighbors $\{J_{j_1}, \ldots, J_{j_K}\}$ of variable $J_j \in J$ according to hash value $\overline{H}_j$ of variable $J_j \in J$.

4.2 Stochastic Optimization Strategy and CUDA Parallelization on GPUs and Multiple GPUs

The basic optimization objective (2) involves six tangled parameters $\{\mathbf{U}, \mathbf{V}, b_i, \widehat{b}_j, w_j, c_j\}$. The state-of-the-art parallel strategy of SGD in References [59, 63] cannot disentangle the involved parameters. Due to the entanglement of the parameters, the optimization objective (2) is nonconvex and alternative minimization is adopted [8, 17, 47, 54], which can disentangle the involved parameters

as follows:

$$\begin{cases} \arg\min_{u_i} \sum_{j\in\Omega_i} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_u \sum_{i=1}^{M} ||u_i||^2; \\ \arg\min_{v_j} \sum_{i\in\widehat{\Omega}_j} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_v \sum_{j=1}^{N} \left\|v_j\right\|^2; \\ \arg\min_{b_i} \sum_{j\in\Omega_i} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_b \sum_{i=1}^{M} b_i^2; \\ \arg\min_{\widehat{b}_j} \sum_{i\in\widehat{\Omega}_j} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_{\widehat{b}} \sum_{j=1}^{N} \widehat{b}_j^2; \\ \arg\min_{w_{j,j_1}} \sum_{i\in\widehat{\Omega}_j} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_w \sum_{J_{j_1}\in R^K(i;j)} w_{j,j_1}^2; \\ \arg\min_{c_{j,j_2}} \sum_{i\in\widehat{\Omega}_j} \left(r_{i,j} - \widehat{r}_{i,j}\right)^2 + \lambda_c \sum_{J_{j_2}\in N^K(i;j)} c_{j,j_2}^2. \end{cases} \tag{4}$$

SGD is a powerful optimization strategy for large-scale optimization problems [17, 54]. Using SGD to solve the optimization problem (4) is presented as:

$$\begin{cases} b_i \leftarrow b_i + \gamma_{b_i}\left(e_{i,j} - \lambda_b b_i\right); \\ \widehat{b}_j \leftarrow \widehat{b}_j + \gamma_{\widehat{b}_j}\left(e_{i,j} - \lambda_{\widehat{b}}\widehat{b}_j\right); \\ u_i \leftarrow u_i + \gamma_u\left(e_{i,j}v_j - \lambda_u u_i\right); \\ v_j \leftarrow v_j + \gamma_v\left(e_{i,j}u_i - \lambda_v v_j\right); \\ w_{j,j_1} \leftarrow w_{j,j_1} + \gamma_w\left(\left|R^K(i;j)\right|^{-\frac{1}{2}} e_{i,j}(r_{i,j_1} - \overline{b}_{i,j_1}) - \lambda_w w_{j,j_1}\right); \\ c_{j,j_2} \leftarrow c_{j,j_2} + \gamma_c\left(\left|N^K(i;j)\right|^{-\frac{1}{2}} e_{i,j} - \lambda_c c_{j,j_2}\right), \end{cases} \tag{5}$$

where the parameters $\{\gamma_{b_i}, \gamma_{\widehat{b}_j}, \gamma_u, \gamma_v, \gamma_w, \gamma_c\}$ are the corresponding learning rates and $e_{i,j} = r_{i,j} - \widehat{r}_{i,j}$. The update rule (5) has parallel inherence. Then, the proposed CULSH-MF is composed of the following three steps:

(1) **Basic Optimization Structure (CUSGD++):** CUSGD++ only considers the basic two parameters $\{\mathbf{U}, \mathbf{V}\}$. Compared with cuSGD, CUSGD++ has the following two advantages: (1) Due to the higher usage of GPU registers in **Stream Multiprocessors (SMs)**, u_i or v_j can be updated in the registers, avoiding the time overhead caused by a large number of memory accesses. The memory access model is illustrated in Figure 4. SM $\{1, 2\}$ update $\{u_1, u_2\}$ in the registers, respectively; and $\{\{v_1, v_3, v_4, v_7, v_8, v_{11}, v_{13}\}, \{v_1, v_4, v_6, v_7, v_9, v_{10}, v_{12}\}\}$ are returned to global memory after each update step. (2) Due to the disentanglement of the parameters in the update rule (5), the data access conflict is reduced, which ensures a high access speed. From the update rule (5), the update processes of $\{\mathbf{U}, \mathbf{V}\}$ are symmetric. Algorithm 2 only describes the update process of $\{\mathbf{U}\}$ in the registers as follows: **(1)** Lines 2–3: Given TB thread blocks, feature vectors $\{u_i | i \in \{1, \ldots, M\}\}$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$. Each thread block TB_{tb_idx}

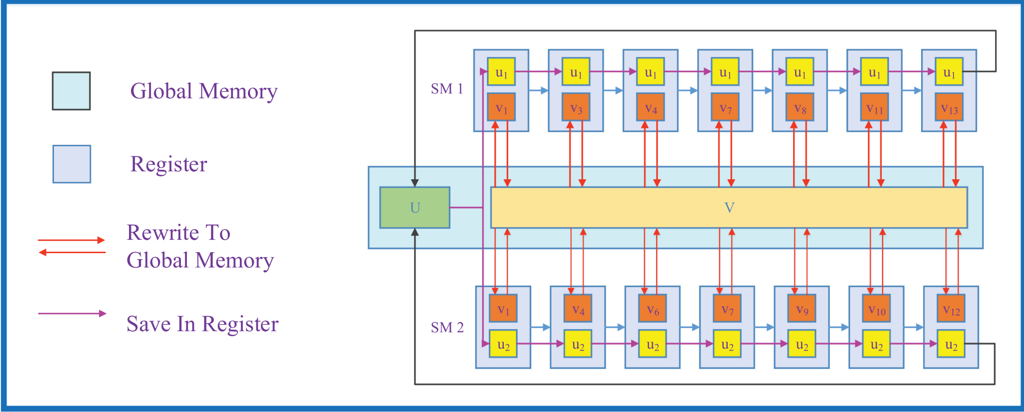

Fig. 4. Memory access model of CUSGD++.

reads its own feature vector u_i from the global memory into the registers. **(2)** Line 4: The feature vector u_i with all nonzero values $\{r_{i,j}|j \in \Omega_i\}$ in the thread block TB_{tb_idx} is updated. **(3)** Lines 5–7: Use the *warp shuffle instructions* [34] to accelerate the dot product $u_i v_j^T$ of two vectors $\{u_i, v_j\}$ and broadcast the result. This technology with additional hardware support uses registers that are faster than shared memory and does not involve thread synchronization. Furthermore, this technology aligns and merges memory to reduce the access time. The number of threads in a *thread warp* under the CUDA platform is 32, and elements $\{u_{i,f}, v_{j,f}|f \in \{1,\ldots,F\}\}$ in feature vectors $\{u_i, v_j\}$ are evenly assigned to thread blocks $\{T_{t_idx}|t_idx \in \{1,\ldots,32\}\}$. A thread T_{t_idx} in each thread block TB_{tb_idx} sequentially reads the corresponding elements $\{u_{i,f}, v_{j,f}|f\%32 = t_idx, f \in \{1,\ldots,F\}\}$ in feature vectors $\{u_i, v_j\}$, and the thread T_{t_idx} calculates the corresponding products $\{u_{i,f} v_{j,f}|f\%32 = t_idx, f \in \{1,\ldots,F\}\}$. Then, the *warp shuffle* in the thread T_{t_idx} to obtain the dot product $u_i v_j^T = \sum_{t_idx} \sum_{f\%32=t_idx} u_{i,f} v_{j,f}$. **(4)** Lines 8–10: Feature vectors u_i are updated in the registers to avoid rereading from global memory for the next update, and feature vectors v_j are updated directly in global memory. **(5)** Line 11: After all nonzero values $\{r_{i,j}|j \in \Omega_i\}$ have been updated, the latest u_i are written to global memory because it will no longer be used.

(2) **Aggregated Model (CULSH-MF):** The updating process of $\{\mathbf{W}, \mathbf{C}\}$ for each thread T_{t_idx} is imbalanced. This imbalance does not affect the serial model. However, it obviously affects the running speed of the parallel model. The most significant impacts are the following two points: (1) discontinuous memory access and (2) imbalanced load on each thread T. To solve the above problems, an adjustment for the parameters $\{\mathbf{W}, \mathbf{C}\}$ is proposed in this section. In CULSH-MF, the adjustment takes the set $R(i)$ as a complement of the set $N(i)$. Therefore, $S^K(j) = R^K(i;j) \bigcup N^K(i;j)$,$R^K(i;j) \bigcap N^K(i;j) = \varnothing$. Thus, the number of the involved elements for $\{\mathbf{W}, \mathbf{C}\}$ are equal and each variable J_j involves $2K$ parameters $\{\{w_{j,k}|k \in \{1,\ldots,K\}\}, \{c_{j,k}|k \in \{1,\ldots,K\}\}\}$. For the convenience of the expression, we use k_1 and k_2 to represent the indexes of j_1 and j_2 in these K parameters, respectively, which means that w_{j,j_1} and c_{j,j_2} are represented as w_{j,k_1} and c_{j,k_2}, respectively. The computational process of $\sum_{J_{j_1} \in R^K(i;j)} (r_{i,j_1} - \overline{b}_{i,j_1}) w_{j,k_1}$ and $\sum_{J_{j_2} \in N^K(i;j)} c_{j,k_2}$ involves the dot product and summation operations. Thus, the *warp shuffle instructions*, which can align and merge memory to reduce the overhead for GPU memory access, are used.

CULSH-MF also takes advantage of the register to reduce the memory access overhead and then increase the overall speed. Due to the limited space, we only introduce the update rule of

ALGORITHM 2: CUSGD++

$\mathcal{G}\{parameter\}$: parameter in global memory
$\mathcal{R}\{parameter\}$: parameter in register memory
Input: Initialization of low-rank feature matrices $\{\mathbf{U}, \mathbf{V}\}$, interaction matrix $\mathbf{R}$, learning rate $\{\gamma_u, \gamma_v\}$, regularization parameter $\{\lambda_u, \lambda_v\}$, and training epoches *epo*.
Output: $\mathbf{U}$.

```
1:  for : loop from 1 to epo do
2:      for (parallel): {TB_{tb_idx} | tb_idx ∈ {1, ..., TB}} manages its own feature vectors {u_i | i ∈ {1, ..., M}}
        do
3:          R{u_i} ← G{u_i}
4:          for : all {r_{i,j} | j ∈ Ω_i} do
5:              Calculate r̂_{i,j} = u_i v_j^T.
6:              Calculate e_{i,j} = r_{i,j} − r̂_{i,j}.
7:              Update u_i, v_j by update rule (5).
8:              R{u_i} ← u_i
9:              G{v_j} ← v_j
10:         end for
11:         G{u_i} ← R{u_i}
12:     end for
13: end for
```

$\{\mathbf{V}, \widehat{b}_j, \mathbf{W}, \mathbf{C}\}$ in the registers. In Algorithm 3, the update process is presented in detail as follows: **(1)** Line 1: Average value $\mu = \sum_{(i,j)\in\Omega} r_{i,j}/|\Omega|$ as the basis value. **(2)** Lines 3–7: Given TB thread blocks, parameters $\{v_j, \widehat{b}_j, w_j, c_j | j \in \{1, \ldots, N\}\}$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$. Each thread block TB_{tb_idx} reads its own parameters $\{v_j, \widehat{b}_j, w_j, c_j\}$ from the global memory into the registers. In addition, the reading of memory is also aligned and merged. **(3)** Line 8: The parameters $\{v_j, \widehat{b}_j, w_j, c_j\}$ with all nonzero values $\{r_{i,j} | i \in \widehat{\Omega}_j\}$ in thread block TB_{tb_idx} are updated. **(3)** Lines 9–11: Use the *warp shuffle instructions* [34] to accelerate the dot product $u_i v_j^T$ and summation $\{\sum_{j_1 \in R^K(i;j)} (r_{i,j_1} - b_{i,j_1}) w_{j,k_1}, \sum_{j_2 \in N^K(i;j)} c_{k,k_2}\}$. Elements $\{u_{i,f}, v_{j,f}, w_{j,k_1}, c_{j,k_2} | f \in \{1, \ldots, F\}, k_1, k_2 \in \{1, \ldots, K\}\}$ in parameters $\{u_i, v_j, w_j, c_j | j \in \{1, \ldots, N\}\}$ are evenly assigned to thread blocks $\{T_{t_idx} | t_idx \in \{1, \ldots, 32\}\}$. A thread T_{t_idx} in each thread block TB_{tb_idx} sequentially reads the corresponding elements $\{u_{i,f}, v_{j,f}, w_{j,k_1}, c_{j,k_2} | f\%32 = k_1\%32 = k_2\%32 = t_idx, f \in \{1, \ldots, F\}, k_1, k_2 \in \{1, \ldots, K\}\}$ in parameters $\{u_i, v_j, w_j, c_j\}$, and the thread T_{t_idx} calculates the corresponding calculations $\{u_{i,f} v_{j,f}, (r_{i,j_1} - b_{i,j_1}) w_{j,k_1}, c_{k,k_2} | f\%32 = k_1\%32 = k_2\%32 = t_idx, f \in \{1, \ldots, F\}, k_1, k_2 \in \{1, \ldots, K\}\}$. Please note that since $S^K(j) = R^K(i;j) \bigcup N^K(i;j)$ and $R^K(i;j) \bigcap N^K(i;j) = \emptyset$, the thread only calculates one of $(r_{i,j_1} - b_{i,j_1}) w_{j,k_1}$ and c_{k,k_2}. This makes the load of each thread T_{t_idx} relatively balanced during the update process. Then, the *warp shuffle* in thread T_{t_idx} to obtain the $\widehat{r}_{i,j} = \mu + b_i + \widehat{b}_j + \sum_{t_idx} (\sum_{f\%32=t_idx} u_{i,f} v_{j,f} + \sum_{\substack{k_1\%32=t_idx \\ j_1 \in R^K(i;j)}} (r_{i,j_1} - b_{i,j_1}) w_{j,k_1} + \sum_{\substack{k_2\%32=t_idx \\ j_2 \in N^K(i;j)}} c_{k,k_2})$. **(4)** Lines 12–18: Parameters $\{v_j, \widehat{b}_j, w_j, c_j\}$ are updated in the registers to avoid rereading from global memory for the next update, and parameters $\{u_i, b_i\}$ are updated directly in global memory. **(5)** Lines 19–22: After all nonzero values $\{r_{i,j} | i \in \widehat{\Omega}_j\}$ have been updated, the latest $\{v_j, \widehat{b}_j, w_j, c_j\}$ are written to global memory because they will no longer be used. These operations are similar to CUSGD++.

The algorithm has the following advantages: **(1)** It stores a large number of parameters in registers, avoiding frequent access to global memory and decreasing the time consumption; and **(2)** The parameter distribution is regular such that each thread T_{t_idx} is balanced, which can avoid

ALGORITHM 3: CULSH-MF

$\mathcal{G}\{parameter\}$: parameter in global memory
$\mathcal{R}\{parameter\}$: parameter in register memory
Input: Initialization for $\{\mathbf{U}, \mathbf{V}, \mu, b_i, \widehat{b}_j, \mathbf{W}, \mathbf{C}\}$, sparse matrix $\mathbf{R}$, learning rate parameters $\{\gamma_b, \gamma_{\widehat{b}}, \gamma_u, \gamma_v, \gamma_w, \gamma_c\}$, regularization parameters $\{\lambda_b, \lambda_{\widehat{b}}, \lambda_u, \lambda_v, \lambda_w, \lambda_c\}$, and training epoches *epo*.
Output: $\{\mathbf{U}, \mathbf{V}, \mu, b_i, \widehat{b}_j, \mathbf{W}, \mathbf{C}\}$.

1: $u \leftarrow$ Average value of rating matrix $\mathbf{R}$.
2: **for** *loop* from 1 to *epo* **do**
3: **for (parallel)**: $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ manages its own parameters $\{u_i, v_j, w_j, c_j | j \in \{1, \ldots, N\}\}$ **do**
4: $\mathcal{R}\{\widehat{b}_j\} \leftarrow \mathcal{G}\{\widehat{b}_j\}$;
5: $\mathcal{R}\{v_i\} \leftarrow \mathcal{G}\{v_j\}$
6: $\mathcal{R}\{w_j\} \leftarrow \mathcal{G}\{w_j\}$
7: $\mathcal{R}\{c_j\} \leftarrow \mathcal{G}\{c_j\}$
8: **for** all $\{r_{i,j} | i \in \widehat{\Omega}_j\}$ **do**
9: Calculate $\widehat{r}_{i,j}$ by Equation (1).
10: Calculate $e_{i,j} = r_{i,j} - \widehat{r}_{i,j}$.
11: Update $\{b_i, \widehat{b}_j, u_i, v_j, w_j, c_j\}$ by update rule (5).
12: $\mathcal{R}\{\widehat{b}_j\} \leftarrow \widehat{b}_j$
13: $\mathcal{R}\{v_j\} \leftarrow v_j$
14: $\mathcal{R}\{w_j\} \leftarrow w_j$
15: $\mathcal{R}\{c_j\} \leftarrow c_j$
16: $\mathcal{G}\{b_i\} \leftarrow b_i$
17: $\mathcal{G}\{u_i\} \leftarrow u_i$
18: **end for**
19: $\mathcal{G}\{\widehat{b}_j\} \leftarrow \mathcal{R}\{\widehat{b}_j\}$
20: $\mathcal{G}\{v_j\} \leftarrow \mathcal{R}\{v_j\}$
21: $\mathcal{G}\{w_j\} \leftarrow \mathcal{R}\{w_j\}$
22: $\mathcal{G}\{c_j\} \leftarrow \mathcal{R}\{c_j\}$
23: **end for**
24: **end for**

idle threads and can improve the active rate of threads. Compared with CUSGD++, CULSH-MF can assemble more tangled parameters of the nonlinear MF model. The parameters $\{v_j, \widehat{b}_j, w_j, c_j\}$ are taken as a whole, and the memory is merged and aligned. Then, the use of *warp shuffle* can further optimize the memory access by allowing the computational overhead to be further reduced. The spatial overhead is $O(|\Omega| + MF + NF + 3NK)$ for interaction sparse matrix $\mathbf{R}$, low-rank factor matrices $\{\mathbf{U}, \mathbf{V}\}$, influence matrices $\{\mathbf{W}, \mathbf{C}\}$, and the Top-$K$ GSM matrix $\mathbf{J}^K$.

(3) **Multi-GPU Model:** With big data, a single GPU still cannot meet our requirements. Therefore, the method must be extended to multiple GPUs (MCUSGD++/MCULSH-MF). We use data parallelism to allow multiple GPUs to run our algorithms at the same time. To avoid data conflicts, each GPU-updated block cannot be on the same I_i or on the same J_j. After the update is completed, the updated parameters are not sent back to the CPU because another GPU needs these data directly. Transferring data directly in the GPUs avoids the extra time overhead of uploading to the CPU and then allocates them to other GPUs. Each GPU is assigned some specific parameters, which are not needed by other GPUs. After all updates are completed, each GPU passes the parameters that are stored at that time back to the CPU.

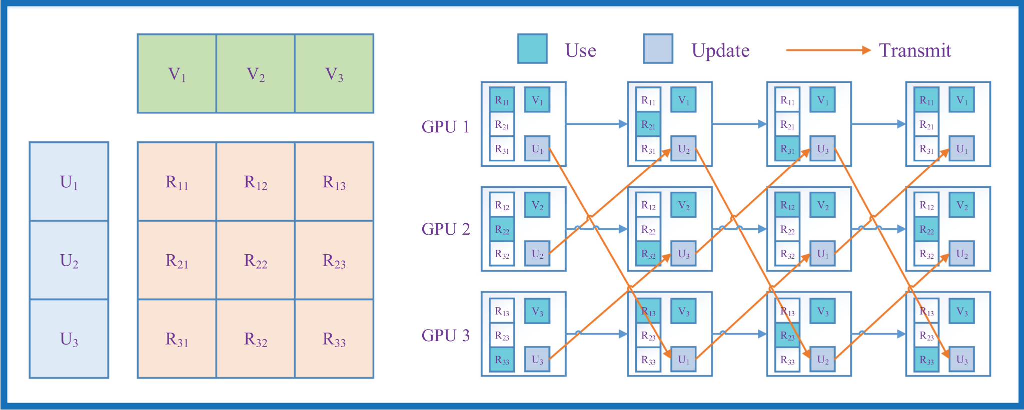

Fig. 5. Multi-GPU solution.

Assume that we have D GPUs. The sparse matrix $\mathbf{R}$ is divided into $D \times D$ parts $\{\mathbf{R}_{d_1,d_2}|d_1, d_2 \in \{1,\dots,D\}\}$. Low-rank feature matrices $\{\mathbf{U}, \mathbf{V}\}$ are divided into $\{\{\mathbf{U}_{d_1}|d_1 \in \{1,\dots,D\}\}, \{\mathbf{V}_{d_2}|d_2 \in \{1,\dots,D\}\}\}$, respectively. Influence matrices $\{\mathbf{W}, \mathbf{C}\}$ are divided into $\{\{\mathbf{W}_{d_2}|d_2 \in \{1,\dots,D\}\}, \{\mathbf{C}_{d_2}|d_2 \in \{1,\dots,D\}\}\}$, respectively. The parameters $\{\mathbf{R}_{d_1,d_2}, \mathbf{V}_{d_2}, \mathbf{W}_{d_2}, \mathbf{C}_{d_2}|d_1 \in \{1,\dots,D\}\}$ are allocated to the d_2th GPU and do not require transmission. Parameter $\mathbf{V}_{d_1}$ is allocated to the d_1th GPU at initialization and then transferred to another GPU after each update step. Figure 5 depicts MCUSGD++ on three GPUs. MCULSH-MF is similar and is given in parentheses below. The sparse matrix $\mathbf{R}$ is divided into 3×3 blocks. The training process of all the parameters is divided into three parts: (1): GPUs $\{1, 2, 3\}$ update $\{\{\mathbf{U}_1, \mathbf{V}_1, (\mathbf{W}_1, \mathbf{C}_1)\}, \{\mathbf{U}_2, \mathbf{V}_2, (\mathbf{W}_2, \mathbf{C}_2)\}, \{\mathbf{U}_3, \mathbf{V}_3, (\mathbf{W}_3, \mathbf{C}_3)\}\}$ and then transmit $\{\mathbf{U}_1, \mathbf{U}_2, \mathbf{U}_3\}$ to GPUs $\{3, 1, 2\}$, respectively; (2): GPUs $\{1, 2, 3\}$ update $\{\{\mathbf{U}_2, \mathbf{V}_1, (\mathbf{W}_1, \mathbf{C}_1)\}, \{\mathbf{U}_3, \mathbf{V}_2, (\mathbf{W}_2, \mathbf{C}_2)\}, \{\mathbf{U}_1, \mathbf{V}_3, (\mathbf{W}_3, \mathbf{C}_3)\}\}$ and transmit $\{\mathbf{U}_2, \mathbf{U}_3, \mathbf{U}_1\}$ to GPUs $\{3, 1, 2\}$, respectively; and (3): GPUs $\{1, 2, 3\}$ update $\{\{\mathbf{U}_3, \mathbf{V}_1, (\mathbf{W}_1, \mathbf{C}_1)\}, \{\mathbf{U}_1, \mathbf{V}_2, (\mathbf{W}_2, \mathbf{C}_2)\}, \{\mathbf{U}_2, \mathbf{V}_3, (\mathbf{W}_3, \mathbf{C}_3)\}\}$ and transmit $\{\mathbf{U}_3, \mathbf{U}_1, \mathbf{U}_2\}$ to GPUs $\{3, 1, 2\}$, respectively.

4.3 Online Learning

Big data analysis should consider the incremental data, and the corresponding model can be compatible with the incremental data. The amount of incremental data is much smaller than the amount of original data. Thus, the time overhead for retraining the overall data is not worthwhile. It is nontrivial to design an online model for incremental data. The variable sets $\{I, \overline{I}, \widehat{I}\}$ and $\{J, \overline{J}, \widehat{J}\}$ are denoted as the original variable set, new variable set, and overall variable set, respectively. In this work, we consider that the new variable sets $\overline{I}$ and $\overline{J}$ enter the system and interact with variable sets J and I, respectively. Please note that this allows variable set $\overline{I}$ to interact with variable set $\overline{J}$.

For the original variable $J_j \in J$, the Top-K nearest neighbors $\{J_{j_1}, \dots, J_{j_K}\} \in J$ are kept. For the new variable $\overline{J}_{\overline{j}} \in \overline{J}$, we search its Top-$K$ nearest neighbors $\{\widehat{J}_{\widehat{j}_1}, \dots, \widehat{J}_{\widehat{j}_K}\} \in \widehat{J}$. The hash value of variable set J depends on I, and the hash value of variable set $\overline{J}$ depends on $\widehat{I}$. To keep them consistent, we update the hash value of variable $J_j \in J$; then, we save the intermediate variables $\sum_{i\in\widehat{\Omega}_j} \Psi(r_{i,j})(2 \cdot H_i - 1)$ of simLSH and update $\overline{H}_j = \Upsilon(\sum_{i\in\widehat{\Omega}_j} \Psi(r_{i,j})\Phi(H_i) + \sum_{\overline{i}\in\widehat{\Omega}_j} \Psi(r_{\overline{i},j})\Phi(H_{\overline{i}}))$. Furthermore, we obtain $\overline{H}_{\overline{j}} = \Upsilon\left(\sum_{\widehat{i}\in\widehat{\Omega}_{\overline{j}}} \Psi(r_{\widehat{i},\overline{j}})\ \Phi(H_{\widehat{i}}) \right)$. The online learning solution is described in Algorithm 4 as follows: (1) Lines 1–3: Update the hash value $\overline{H}_j$ for variable $J_j \in J$. Saving the

ALGORITHM 4: Online Learning

Input: $\{b_i, u_i, \widehat{b}_j, v_j, w_j, c_j\}$, new variable sets $\overline{I}$ and $\overline{J}$, random Hash values $H_{\overline{i}}$.
Output: $\{b_{\overline{i}}, u_{\overline{i}}\}$,$\{\widehat{b}_{\overline{j}}, v_{\overline{j}}, w_{\overline{j}}, c_{\overline{j}}\}$.
1: **for** *loop* from 1 to *epo* **do**
2: **for (parallel)**: Variables $J_j \in J$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
3: Update the hash value $\overline{H}_j$ for variable $J_j \in J$.
4: **end for**
5: **for (parallel)**: Variables $\overline{J}_{\overline{j}} \in \overline{J}$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
6: Calculate the hash value $\overline{H}_{\overline{j}}$ for variable $\overline{J}_{\overline{j}} \in \overline{J}$.
7: **end for**
8: **for (parallel)**: Variables $\overline{J}_{\overline{j}} \in \overline{J}$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
9: Search the Top-K nearest neighbors $\{\widehat{J}_{\widehat{j}_1}, \ldots, \widehat{J}_{\widehat{j}_K}\}$ of the variable $\overline{J}_{\overline{j}} \in \overline{J}$.
10: **end for**
11: **for (parallel)**: Variables $J_j \in J$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
12: Update $\{b_{\overline{i}}, u_{\overline{i}}\}$ for variable $\overline{I}_{\overline{i}} \in \overline{I}$.
13: **end for**
14: **for (parallel)**: Variables $\overline{J}_{\overline{j}} \in \overline{J}$ are evenly assigned to thread blocks $\{TB_{tb_idx} | tb_idx \in \{1, \ldots, TB\}\}$ **do**
15: Update $\{\widehat{b}_{\overline{j}}, v_{\overline{j}}, w_{\overline{j}}, c_{\overline{j}}\}$ for variable $\overline{J}_{\overline{j}} \in \overline{J}$.
16: **end for**
17: **end for**

intermediate variables makes the process only require a small amount of calculation. (2) Lines 4–6: Calculate hash value $\overline{H}_{\overline{j}}$ for variable $\overline{J}_{\overline{j}} \in \overline{J}$. Both the hash value of variable set J and the hash value of variable set $\overline{J}$ depend on $\widehat{I}$. (3) Lines 7–9: Search the Top-K nearest neighbors $\{\widehat{J}_{\widehat{j}_1}, \ldots, \widehat{J}_{\widehat{j}_K}\}$ of variable $\overline{J}_{\overline{j}} \in \overline{J}$. The Top-$K$ nearest neighbors in the overall variable set $\widehat{J}$ can provide more information. (4) Lines 10–12: Update $\{b_{\overline{i}}, u_{\overline{i}}\}$ for variable $\overline{I}_{\overline{i}} \in \overline{I}$. $\{r_{\overline{i},j} | \overline{I}_{\overline{i}} \in \overline{I}, J_j \in J\}$ is used and $\{\widehat{b}_j, v_j, w_j, c_j\}$ remains unchanged, but they can still be stored in registers to reduce memory access. (5) Lines 13–15: Updating $\{\widehat{b}_{\overline{j}}, v_{\overline{j}}, w_{\overline{j}}, c_{\overline{j}}\}$ for variable $\overline{J}_{\overline{j}} \in \overline{J}$, $\{r_{\widehat{i},\overline{j}} | \widehat{I}_{\widehat{i}} \in \widehat{I}, \overline{J}_{\overline{j}} \in \overline{J}\}$ is used, and $\{\widehat{b}_j, v_j, w_j, c_j\}$ remains unchanged.

5 EXPERIMENTS

CULSH-MF is composed of two parts: (1) Basic parallel optimization model depends on CUSGD++, which can utilize the GPU registers more and disentangle the involved parameters. CUSGD++ achieves the fastest speed compared to the state-of-the-art algorithms. (2) The Top-K nearest neighborhood query relies on the proposed simLSH, which can reduce the time and memory overheads. Furthermore, it can improve the overall approximation accuracy. To demonstrate the effectiveness of the proposed model, we present the experimental settings in Section 5.1. The speedup performance of CUSGD++ compared with the state-of-the-art algorithms is shown in Section 5.2. The accuracy, robustness, online learning, and multiple GPUs of CULSH-MF are presented in Section 5.3.

Table 2. Datasets

Parameter	Netflix	MovieLens	Yahoo! Music
M	480,189	69,878	586,250
N	17,770	10,677	12,658
$\|\Omega\|$	99,072,112	9,900,054	91,970,212
$\|\Gamma\|$	1,408,395	100,000	1,000,000
Max Value	5	5	100
Min Value	1	0.5	0.5

CULSH-MF is a nonlinear neighborhood model for low-rank representation learning, and we compare CULSH-MF with the DL model in Section 5.4 to demonstrate the effectiveness of CULSH-MF.

5.1 Experimental Setting

The experiments were run on an NVIDIA Tesla P100 GPU with CUDA version 10.0. The same software and hardware conditions can better reflect the superiority of the proposed algorithm. The experiments are conducted on 3 public datasets: Netflix,[1] MovieLens,[2] and Yahoo! Music.[3] For MovieLens and Yahoo! Music, data cleaning is conducted, and 0 values are changed from 0 to 0.5. This will make cuALS work properly, which is one of the shortcomings of cuALS. The specific situations of the datasets are shown in Table 2. The ratings in the Yahoo! Music dataset are relatively large, which affects the training process. In the actual training process, we divided all the ratings in the Yahoo! Music dataset by 20, and then we multiply by 20 when verifying the results. In this way, the ratings of the three datasets are in the same interval, which facilitates the parameter selection. The accuracy is measured by the *RMSE* as:

$$RMSE = \sqrt{\left(\sum_{(i,j)\in\Gamma} (v_{i,j} - \widetilde{v}_{i,j})^2 \right) \Big/ |\Gamma|}, \tag{6}$$

where Γ denotes the test sets.

The number of threads in a *thread warp* under the CUDA system is 32. Therefore, we set the number of threads in the thread block to a multiple of 32. This is done to maximize the utilization of the warp. Then, to align access, we set the parameters $\{F, K\}$ as multiples of 32.

5.2 CUSGD++

CUSGD++ is used to compare cuALS [54] and cuSGD [59] on the three datasets. The parameters of cuALS and cuSGD were set as described in their papers and optimized according to the hardware environment, and CUSGD++ uses the dynamic learning rate in Reference [63] as

$$\gamma_t = \frac{\alpha}{1 + \beta \cdot t^{1.5}}, \tag{7}$$

where the parameters $\{\alpha, \beta, t, \gamma_t\}$ represent the initial learning rate, adjusting parameter of the learning rate, the number of current iterations, and the learning rate at t iterations, respectively. The learning rate and other parameters in CUSGD++ are listed in Table 3.

The GPU experiments are conducted on three datasets. To ensure running fairness, we ensure that the GPU executes these algorithms independently, and there is no other work. Figure 6 shows

[1] https://www.netflixprize.com/.
[2] https://grouplens.org/datasets/movielens/.
[3] https://webscope.sandbox.yahoo.com/.

Table 3. CUSGD++ Parameters

Parameter	Netflix	MovieLens	Yahoo! Music
α	0.04	0.04	0.01
β	0.3	0.3	0.1
λ_u	0.035	0.035	0.02
λ_v	0.035	0.035	0.02

Table 4. Speedup Comparison on the Baseline cuALS

Algorithm	Netflix	MovieLens	Yahoo! Music
cuALS	15.00	1.30	15.60
cuSGD	5.05 (3.0×)	0.31 (4.2×)	1.92 (8.1×)
CUSGD++	1.49 (10.1×)	0.15 (8.7×)	0.69 (22.6×)

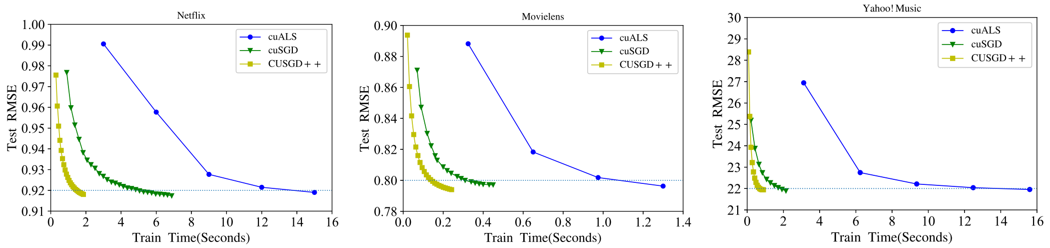

Fig. 6. RMSE vs. time: The experimental results demonstrate that CUSGD++ converges faster than other approaches.

the relationship between the *RMSE* and training time. In Table 4, the times it takes to achieve an acceptable *RMSE* (0.92, 0.80, and 22.0 for Netflix, MovieLens and Yahoo! Music, respectively) are presented. cuALS has an extremely fast descent speed, but the time of each iteration is very long, because the matrix inversion calculation is performed twice for each iteration. Furthermore, because the number of $\{r_{i,j}|j \in \Omega_i\}$ for each I_i is very different and the number of $\{r_{i,j}|i \in \widehat{\Omega}_j\}$ for each J_j is the same, the thread load imbalance further increases the time overhead. cuSGD has a slower descent speed but less time overhead per iteration due to using data parallelism without load-balancing issues.

cuSGD has an obvious flaw in that it does not take full advantage of the hardware resources of the GPU. cuSGD stores data in global memory, which makes it take too much time to read and write data. Our proposed CUSGD++ is significantly faster than the state-of-the-art algorithms on the GPU. CUSGD++ and cuSGD have the same number of iterations to obtain an acceptable *RMSE*, and the speed of a single iteration is 2–3 times faster than cuSGD. With the same gradient descent algorithm, the proposed CUSGD++ and cuSGD algorithms are basically the same in terms of descent speed. CUSGD++ makes full use of the GPU hardware. Therefore, the time overhead of each iteration is only approximately 1/3 that of cuSGD. It is inevitable that CUSGD++ results in a thread load imbalance problem, and our further work is to solve this problem. Simultaneously, we simply sort the index of the row or column for $I_i \in I$ according to the number of $\{r_{i,j}|j \in \Omega_i\}$. Therefore, I_i containing more nonzero elements $\{r_{i,j}|j \in \Omega_i\}$ is updated first. This approach can reduce the time overhead on a single iteration and achieve speedups of $\{1.02X, 1.03X, 1.06X\}$ on the Netflix, MovieLens, and Yahoo! Music datasets, respectively.

Table 5. The Initial Learning Speed and Regularization Parameters of CULSH-MF for All Three Datasets

Parameter	Netflix	MovieLens	Yahoo! Music
α_i	0.02	0.035	0.02
$\widehat{\alpha}_j$	0.02	0.035	0.02
α_u	0.02	0.035	0.02
α_v	0.02	0.035	0.02
α_w	0.001	0.002	0.001
α_c	0.001	0.002	0.001
λ_{b_i}	0.01	0.02	0.02
$\lambda_{\widehat{b}_j}$	0.01	0.02	0.02
λ_u	0.01	0.02	0.02
λ_v	0.01	0.02	0.02
λ_w	0.05	0.002	0.05
λ_c	0.05	0.002	0.05

Table 6. Running Time (Seconds)

Algorithm	Platform	F	K	Time
Serial	Intel Xeon E5-2620 CPU	32	32	782.64
LSH-MF	Intel Xeon E5-2620 CPU	32	32	17.66
CULSH-MF	Nvidia Tesla P100 GPU	32	32	0.09

5.3 CULSH-MF

Before introducing the experiment, we will introduce the selection of the relevant parameters. CULSH-MF still uses the dynamic learning rate in Equation (7). The initial learning rate and regularization parameters are shown in Table 5, and β for all three datasets is 0.3.

To clarify the superiority of CULSH-MF, the experimental presentation is split into the following five parts: (1) The overall performance comparison, (2) the performance comparison for the various methods of Top-K nearest neighborhood query, (3) the performance comparison of neighborhood nonlinear MF with naive MF methods, (4) the performance comparison on a GPU and multiple GPUs, and (5) the robustness of CULSH-MF.

We first compare the serial algorithms, i.e., LSH-MF and GSM-based Top-K nearest neighborhood MF [29]. To ensure the fairness of the comparison, the parameters used are the same [29]. The serial algorithms are conducted on an Intel Xeon E5-2620 CPU, and the CUDA parallelization algorithms are conducted on an NVIDIA Tesla P100 GPU. Parameters $\{F, K\}$ are set as $\{32, 32\}$, respectively. Table 6 presents the time overhead of the three algorithms on the MovieLens dataset (baseline RMSE 0.80). The experimental results show that the LSH-MF can achieve a 44.3× speedup compared to the GSM-based Top-K nearest neighborhood MF. CULSH-MF can achieve a 196.22× speedup compared to the LSH-MF serial algorithm. These results demonstrate that the proposed algorithms are efficient.

The comparison baselines of the GSM and simLSH are set under the same experimental conditions. To make the experiment more rigorous, a randomized control group was added, and it randomly selects K variables for each variable rather than the Top-K nearest neighbors query.

Furthermore, we compared two other LSH algorithms, **random projection (RP_cos) based on cosine distance** and minHash based on Jaccard similarity. On sparse data, compared to the

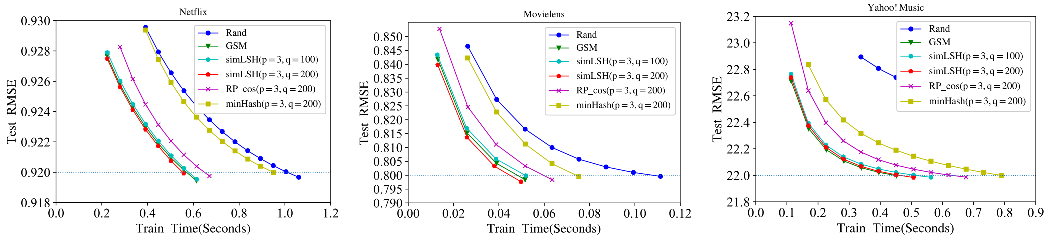

Fig. 7. *RMSE* vs. time: The comparison between GSM, simLSH (various p and q values), and other LSH algorithms.

Euclidean distance, the LSH algorithms based on the cosine distance have less accuracy loss. In addition, minHash can approximately calculate the Jaccard similarity between sets or vectors. The above two LSH functions are simple and have low computational complexity, Furthermore, the more complex LSH functions are not suitable for high-dimensional sparse data.

The baseline *RMSE*s are $\{0.92, 0.80, 22.0\}$ for Netflix, MovieLens, and Yahoo! Music, respectively. For the MovieLens and Netflix datasets, $\Psi(r_{i,j}) = r_{i,j}^2$ is set to expand the gap between interaction values, and the Yahoo! Music dataset has more dense interaction values. Thus, $\Psi(r_{i,j}) = r_{i,j}^4$. We use a byte as a hash value ($G = 8$) and set λ_ρ as the commonly used 100. Figure 7 shows that the random selection method performs worse than the GSM-based method, simLSH, and other LSH algorithms on the three datasets. When the parameters $\{p, q\}$ are set as $\{p = 3, q = 100\}$, simLSH is almost the same as that of the GSM.

When the parameters $\{p, q\}$ are set as $\{p = 3, q = 100\}$, simLSH surpasses the GSM, and the performances of RJ_cos and minHash are far from that of simLSH. The reason is that the datasets are very sparse, and the descent speed brought by minHash is not very impressive.

Table 7 shows the optimal *RMSE* and the corresponding time overhead. Table 7 (top) demonstrates that simLSH can achieve a better *RMSE* than using the GSM and simLSH is better than the GSM and other LSH algorithms not only in descent speed but also in accuracy. Table 7 (middle) shows the time overhead of GSM, simLSH, and other LSH algorithms on the three datasets, and simLSH takes much less time than the GSM. The calculation time required for RP_cos is slightly larger than that of simLSH, and minHash requires considerable calculation time due to the high dimensionality of the datasets. Table 7 (bottom) shows the spatial overhead of GSM, simLSH, and other LSH algorithms on the three datasets, and simLSH takes much less space than the GSM. Furthermore, simLSH can surpass the GSM, since it can adjust the parameters to achieve a balance between accuracy and time, and it can set appropriate parameters according to actual needs. Figure 8 shows the influence of various values of $\{p, q\}$ on the three datasets. The increase in p will reduce the probability of two dissimilar variables projecting to the same hash value to P_2^p, but the probability $1 - (1 - P_1^p)^q$ of two similar variables projected to the same hash value will decrease. Choosing a suitable p will achieve higher accuracy.

We should select the best parameters and ensure which parameters play a greater role. To ensure that the threads are fully utilized, the parameters $\{F, K\}$ are all set as $\{32, 64, 96, 128\}$. Figure 9 illustrates the influences of $\{F, K\}$ on CULSH-MF. As Figure 9 shows, under the same F, CULSH-MF with the neighborhood model obtains higher accuracy than CUSGD++ without the neighborhood model in terms of the *RMSE*. Then, CULSH-MF is compared with CUSGD++ to demonstrate to what degree the neighborhood model can improve the accuracy. Figure 10 shows that CULSH-MF with the parameters $\{F = 128, K = 32\}$ achieves a much faster descent speed than CUSGD++ with $F = 128$. The neighborhood model with a low K can greatly improve the descent speed, and

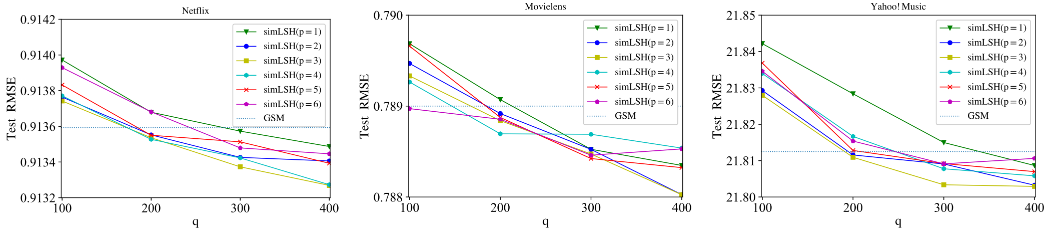

Fig. 8. *RMSE* vs. influence of various value of $\{p, q\}$.

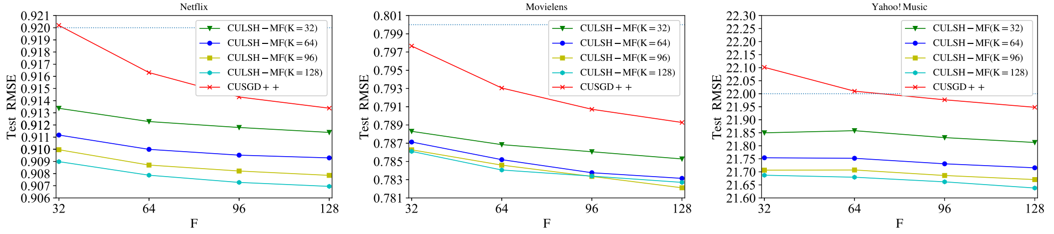

Fig. 9. *RMSE* vs. influence of various value of $\{F, K\}$. Compared with F, increasing K can reduce RMSE more.

Table 7. The Optimal *RMSE* of Various Top-K Methods (Up), the Time Overhead of Various Top-K Methods (Seconds) (Middle), and the Space Overhead of Various Top-K Methods (MB) (Down)

Indicator	Method	Netflix	MovieLens	Yahoo! Music
RMSE	Rand	0.9157	0.7947	21.99
	GSM	0.9136	0.7890	21.81
	simLSH (p = 3, q = 100)	0.9137	0.7893	21.83
	simLSH (p = 3, q = 200)	0.9135	0.7888	21.81
	RP_cos (p = 3, q = 200)	0.9139	0.7896	21.87
	minHash (p = 3, q = 200)	0.9138	0.7892	21.82
Time Overhead (Seconds)	Rand	0.0	0.0	0.0
	GSM	422.996	27.150	295.417
	simLSH (p = 3, q = 100)	15.414	2.777	25.994
	simLSH (p = 3, q = 200)	31.017	5.602	52.012
	RP_cos (p = 3, q = 200)	47.262	8.184	78.953
	minHash (p = 3, q = 200)	270.003	38.224	319.831
Space Overhead (MB)	Rand	0.0	0.0	0.0
	GSM	1,204.578	434.869	611.209
	simLSH (p = 3, q = 100)	20.336	12.219	14.486
	simLSH (p = 3, q = 200)	40.672	24.438	28.972
	RP_cos (p = 3, q = 200)	40.672	24.438	28.972
	minHash (p = 3, q = 200)	40.672	24.438	28.972

it can reach the target *RMSE* with only a few iterations. CUSGD++ has a shorter training time per iteration, but it requires more training periods. Thus, CULSH-MF can outperform CUSGD++ owing to the overall training time with the optimal *RMSE*. Another noteworthy result is that CULSH-MF runs faster than CUSGD++ as the value of F increases. CULSH-MF with parameter

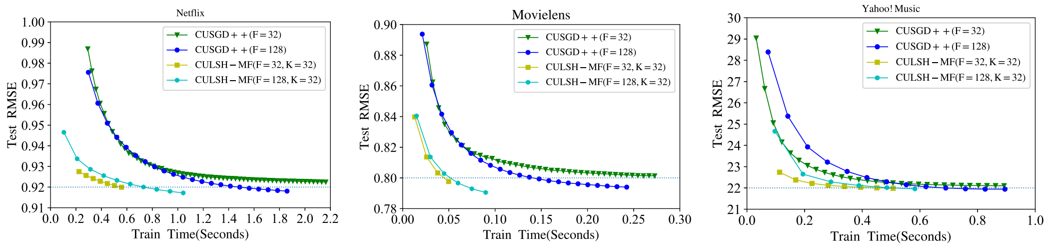

Fig. 10. *RMSE* vs. time: CULSH-MF outperforms CUSGD++ on all three datasets.

Table 8. *RMSE* Deviation of the Noisy Data and the Clean Data

Noise Rate	Algorithm	Netflix	MovieLens	Yahoo! Music
1%	CUSGD++(F = 128)	0.00116	0.00157	0.13840
	CULSH-MF(F = 32, K = 32)	0.00096	0.00166	0.09770
0.5%	CUSGD++(F = 128)	0.00055	0.00092	0.06012
	CULSH-MF(F = 32, K = 32)	0.00045	0.00076	0.04792
0.1%	CUSGD++(F = 128)	0.00032	0.00040	0.01404
	CULSH-MF(F = 32, K = 32)	0.00011	0.00006	0.00954
0.05%	CUSGD++(F = 128)	0.00018	0.00028	0.00814
	CULSH-MF(F = 32, K = 32)	0.00002	0.00004	0.00424
0.01%	CUSGD++(F = 128)	0.00011	0.00016	0.00412
	CULSH-MF(F = 32, K = 32)	0.00001	0.00002	0.00194

Table 9. Online Datasets

Parameter	Netflix	MovieLens	Yahoo! Music
M	475,388	69,180	580,388
N	17,593	10,571	12,532
$\|\Omega\|$	98,339,095	9,789,247	90,752,595
$\overline{M}$	4,801	698	5,862
$\overline{N}$	177	106	126
$\|\overline{\Omega}\|$	733,017	110,807	1,217,617

$K = 32$ can achieve $\{2.67X, 2.97X, 1.36X\}$ speedups compared to CUSGD++ when $F = \{32, 64, 128\}$, respectively.

Finally, we present the experimental results of the robustness of CULSH-MF and CUSGD++, the online learning and multiple GPU solutions of CULSH-MF. First, data inevitably have noise, and a robust model should suppress noise interference. The experiment is conducted on all datasets with noise rates of $\{1\%, 0.5\%, 0.1\%, 0.05\%, 0.01\%\}$. The experimental results in Table 8 show that CULSH-MF has more robustness than CUSGD++, which means that the neighborhood nonlinear model performs more robustly than the naive model. Second, we divide the training datasets of Netflix, MovieLens and Yahoo! Music into original set Ω and new set $\overline{\Omega}$, and $|\Omega| \ll |\overline{\Omega}|$. The specific conditions of the dataset are shown in Table 9. In the online experiments, the *RMSE* of our online CULSH-MF on the Netflix, MovieLens, and Yahoo! Music datasets only increased by $\{0.00015, 0.00040, 0.00936\}$, respectively, which means that online CULSH-MF avoids the retraining process. Third, multiple GPUs can accommodate a larger data, and CULSH-MF is extended

Table 10. Time Comparison (Seconds) to Obtain Basic HR of Various Nonlinear MF Methods

Algorithm	MovieLens1m (HR 0.65)	Pinterest (HR 0.85)
GMF	219.6	335.1
MLP	940.4	1289.9
NeuMF	308.5	402.3
CULSH-MF	0.0343	0.0452

to MCULSH-MF. Due to the communication overhead between each GPU, MCULSH-MF cannot reach the linear speeds, and properly distributing communications can shorten the computation time. CULSH-MF can obtain $\{1.6X, 2.4X, 3.2X\}$ speedups on $\{2, 3, 4\}$ GPUs, respectively, compared to CULSH-MF on a GPU.

Our model also applies to recommendations for implicit feedback and has a very obvious time advantage. NCF works well but takes too much time, and CULSH-MF can achieve similar results with a lower time overhead. We change the loss function of CULSH-MF to the cross-entropy loss function, and the update formula will also follow the corresponding change. This derivation is too simple and will not be repeated here. Because the time overheads to train the deep learning models on large-scale datasets are unacceptable, three deep learning models, e.g., **Generalized Matrix Factorization (GMF), the Multilayer Perceptron (MLP)**, and **Neural Matrix Factorization (NeuMF)**, of Reference [18] are just tested on two small datasets, e.g., MovieLens1m and Pinterest. (1) GMF is a deep learning model based on matrix factorization that extends classic matrix factorization. It first performs one-hot encoding on the indexes in the sets $\{I, J\}$ of the input layer, and the obtained embedding vectors are used as the latent factor vectors. Then, through the neural matrix decomposition layer, it calculated the matrix Hadamard product of factor vector I and factor vector J. Finally, a weight vector and the obtained vector are projected to the output layer by the dot product. (2) The MLP is used to learn the interaction between latent factor vector I and latent factor vector J, which can give the model greater flexibility and nonlinearity. With the same conditions as GMF, the MLP uses the embedded vector of the one-hot encoding of indices I and J as the latent factor vector of I and J. The difference is that MLP concatenates latent factor vector I with latent factor vector J. The model uses the standard MLP; and each layer contains a weight matrix, a deviation vector, and an activation function. (3) GMF uses linear kernels to model the interaction of potential factors, while MLP uses nonlinear kernels to learn the interaction functions from data. To consider the above two factors at the same time, NeuMF integrates GMF and the MLP, embeds GMF and the MLP separately, and combines these two models by connecting their last hidden layers in series. This allows the fusion model to have greater flexibility. The **Hit Ratio (HR)** is used to measure the accuracy of the nonlinear models. We use the same datasets and the same metrics. For the same baseline HR, we compare the time overheads of CULSH-MF and the three nonlinear models, i.e., GMF, the MLP, and NeuMF. The experimental results are shown in Table 10. Table 10 shows that the time overhead of the CULSH-MF is only 0.01% that of the three nonlinear models, i.e., GMF, the MLP, and NeuMF. Furthermore, the parameters of the CULSH-MF are much smaller than those of the three nonlinear models, i.e., GMF, the MLP, and NeuMF.

REFERENCES

[1] Ting Bai, Ji-Rong Wen, Jun Zhang, and Wayne Xin Zhao. 2017. A neural collaborative filtering model with interaction-based neighborhood. In *Proceedings of the ACM Conference on Information and Knowledge Management.* 1979–1982.

[2] Gema Bello-Orgaz, Jason J. Jung, and David Camacho. 2016. Social big data: Recent achievements and new challenges. *Inf. Fus.* 28 (2016), 45–59.

[3] Yoshua Bengio, Aaron Courville, and Pascal Vincent. 2013. Representation learning: A review and new perspectives. *IEEE Trans. Pattern Anal. Mach. Intell.* 35, 8 (2013), 1798–1828.
[4] Christian Borgs, Jennifer Chayes, Christina E. Lee, and Devavrat Shah. 2017. Thy friend is my friend: Iterative collaborative filtering for sparse matrix estimation. In *Advances in Neural Information Processing Systems*. 4715–4726.
[5] A. Z. Broder, M. Charikar, A. M. Frieze, and M. Mitzenmacher. 2000. Min-Wise independent permutations. *J. Comput. Syst. Sci.* 60, 3 (2000), 630–659.
[6] Junting Chen and Urbashi Mitra. 2019. Unimodality-constrained matrix factorization for non-parametric source localization. *IEEE Trans. Sig. Process.* 67, 9 (2019), 2371–2386.
[7] Xixian Chen, Haiqin Yang, Shenglin Zhao, Michael R. Lyu, and Irwin King. 2019. Making online sketching hashing even faster. *IEEE Trans. Knowl. Data Eng.* 33, 3 (2019), 1089–1101.
[8] Yuejie Chi, Yue M. Lu, and Yuxin Chen. 2019. Nonconvex optimization meets low-rank matrix factorization: An overview. *IEEE Trans. Sig. Process.* 67, 20 (2019), 5239–5269.
[9] John P. Cunningham and Zoubin Ghahramani. 2015. Linear dimensionality reduction: Survey, insights, and generalizations. *J. Mach. Learn. Res.* 16, 1 (2015), 2859–2900.
[10] Alexey Dosovitskiy and Thomas Brox. 2016. Generating images with perceptual similarity metrics based on deep networks. In *Advances in Neural Information Processing Systems*. 658–666.
[11] Ghislain Durif, Laurent Modolo, Jeff E. Mold, Sophie Lambert-Lacroix, and Franck Picard. 2019. Probabilistic count matrix factorization for single cell expression data analysis. *Bioinformatics* 35, 20 (2019), 4011–4019.
[12] Raul Castro Fernandez, Jisoo Min, Demitri Nava, and Samuel Madden. 2019. Lazo: A cardinality-based method for coupled estimation of Jaccard similarity and containment. In *IEEE 35th International Conference on Data Engineering (ICDE)*. IEEE, 1190–1201.
[13] Nicolo Fusi, Rishit Sheth, and Melih Elibol. 2018. Probabilistic matrix factorization for automated machine learning. In *Advances in Neural Information Processing Systems*. 3348–3357.
[14] Kelum Gajamannage, Randy Paffenroth, and Erik M. Bollt. 2019. A nonlinear dimensionality reduction framework using smooth geodesics. *Pattern Recog.* 87 (2019), 226–236.
[15] Fei Gao, Yi Wang, Panpeng Li, Min Tan, Jun Yu, and Yani Zhu. 2017. DeepSim: Deep similarity for image quality assessment. *Neurocomputing* 257 (2017), 104–114.
[16] Guibing Guo, Enneng Yang, Li Shen, Xiaochun Yang, and Xiaodong He. 2019. Discrete trust-aware matrix factorization for fast recommendation. In *28th International Joint Conference on Artificial Intelligence*. AAAI Press, 1380–1386.
[17] Benjamin David Haeffele and René Vidal. 2019. Structured low-rank matrix factorization: Global optimality, algorithms, and applications. *IEEE Trans. Pattern Anal. Mach. Intell.* 42, 6 (2019), 1468–1482.
[18] Xiangnan He, Lizi Liao, Hanwang Zhang, Liqiang Nie, Xia Hu, and Tat-Seng Chua. 2017. Neural collaborative filtering. In *26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 173–182.
[19] Xiangnan He, Hanwang Zhang, Min-Yen Kan, and Tat-Seng Chua. 2016. Fast matrix factorization for online recommendation with implicit feedback. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR'16)*. ACM, 549–558.
[20] Gang Hu, Jie Shao, Dongxiang Zhang, Yang Yang, and Heng Tao Shen. 2017. Preserving-ignoring transformation based index for approximate k nearest neighbor search. In *IEEE 33rd International Conference on Data Engineering (ICDE)*. IEEE, 91–94.
[21] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *8th IEEE International Conference on Data Mining*. IEEE, 263–272.
[22] Qiang Huang, Jianlin Feng, Qiong Fang, and Wilfred Ng. 2017. Two efficient hashing schemes for high-dimensional furthest neighbor search. *IEEE Trans. Knowl. Data Eng.* 29, 12 (2017), 2772–2785.
[23] Susmit Jha, Sunny Raj, Steven Fernandes, Sumit K. Jha, Somesh Jha, Brian Jalaian, Gunjan Verma, and Ananthram Swami. 2019. Attribution-based confidence metric for deep neural networks. In *Advances in Neural Information Processing Systems*. 11826–11837.
[24] Wenjun Jiang, Guojun Wang, Md Zakirul Alam Bhuiyan, and Jie Wu. 2016. Understanding graph-based trust evaluation in online social networks: Methodologies and challenges. *ACM Comput. Surv.* 49, 1 (2016), 1–35.
[25] Wenjun Jiang, Jie Wu, Feng Li, Guojun Wang, and Huanyang Zheng. 2015. Trust evaluation in online social networks using generalized network flow. *IEEE Trans. Comput.* 65, 3 (2015), 952–963.
[26] Yaron Kanza, Elad Kravi, Eliyahu Safra, and Yehoshua Sagiv. 2017. Location-based distance measures for geosocial similarity. *ACM Trans. Web* 11, 3 (2017), 1–32.
[27] Jaya Kawale, Hung H. Bui, Branislav Kveton, Long Tran-Thanh, and Sanjay Chawla. 2015. Efficient Thompson sampling for online matrix-factorization recommendation. In *Advances in Neural Information Processing Systems*. 1297–1305.

[28] Donghyun Kim, Chanyoung Park, Jinoh Oh, Sungyoung Lee, and Hwanjo Yu. 2016. Convolutional matrix factorization for document context-aware recommendation. In *10th ACM Conference on Recommender Systems*. ACM, 233–240.

[29] Yehuda Koren. 2008. Factorization meets the neighborhood: A multifaceted collaborative filtering model. In *14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 426–434.

[30] Vikas Kumar, Arun K. Pujari, Sandeep Kumar Sahu, Venkateswara Rao Kagita, and Vineet Padmanabhan. 2017. Collaborative filtering using multiple binary maximum margin matrix factorizations. *Inf. Sci.* 380 (2017), 1–11.

[31] Alexandros Labrinidis and Hosagrahar V. Jagadish. 2012. Challenges and opportunities with big data. *Proc. VLDB Endow.* 5, 12 (2012), 2032–2033.

[32] Daniel D. Lee and H. Sebastian Seung. 2001. Algorithms for non-negative matrix factorization. In *Advances in Neural Information Processing Systems*. 556–562.

[33] Hao Li, Kenli Li, Jiyao An, and Keqin Li. 2017. MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs. *IEEE Trans. Parallel Distrib. Syst.* 29, 7 (2017), 1530–1544.

[34] Hao Li, Keqin Li, Jiyao An, Weihua Zheng, and Kenli Li. 2019. An efficient manifold regularized sparse non-negative matrix factorization model for large-scale recommender systems on GPUs. *Inf. Sci.* 496 (2019), 464–484.

[35] Hao Li, Zixuan Li, Kenli Li, Jan S. Rellermeyer, Lydia Chen, and Keqin Li. 2021. SGD_Tucker: A novel stochastic optimization strategy for parallel sparse Tucker decomposition. *IEEE Trans. Parallel Distrib. Syst.* 32, 7 (2021), 1828–1841. DOI: https://doi.org/10.1109/TPDS.2020.3047460

[36] Hangyu Li, Sarana Nutanong, Hong Xu, Foryu Ha, et al. 2018. C2Net: A network-efficient approach to collision counting LSH similarity join. *IEEE Trans. Knowl. Data Eng.* 31, 3 (2018), 423–436.

[37] H. Li, K. Ota, M. Dong, A. Vasilakos, and K. Nagano. 2020. Multimedia processing pricing strategy in GPU-accelerated cloud computing. *IEEE Trans. Cloud Comput.* 8, 4 (2020), 1264–1273.

[38] Xuelong Li, Guosheng Cui, and Yongsheng Dong. 2016. Graph regularized non-negative low-rank matrix factorization for image clustering. *IEEE Trans. Cyber.* 47, 11 (2016), 3840–3853.

[39] Defu Lian, Xing Xie, and Enhong Chen. 2019. Discrete matrix factorization and extension for fast item recommendation. *IEEE Trans. Knowl. Data Eng.* 1 (2019), 1–1.

[40] Dawen Liang, Jaan Altosaar, Laurent Charlin, and David M. Blei. 2016. Factorization meets the item embedding: Regularizing matrix factorization with item co-occurrence. In *Proceedings of the 10th ACM Conference on Recommender Systems*. ACM, 59–66.

[41] Kevin Lin, Huei-Fang Yang, Jen-Hao Hsiao, and Chu-Song Chen. 2015. Deep learning of binary hash codes for fast image retrieval. In *IEEE Conference on Computer vision and Pattern Recognition Workshops*. 27–35.

[42] Xin Liu, Zhikai Hu, Haibin Ling, and Yiu-ming Cheung. 2019. MTFH: A matrix tri-factorization hashing framework for efficient cross-modal retrieval. *IEEE Trans. Pattern Anal. Mach. Intell.* (2019).

[43] Xin Liu, Tsuyoshi Murata, Kyoung-Sook Kim, Chatchawan Kotarasu, and Chenyi Zhuang. 2019. A general view for network embedding as matrix factorization. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining (Melbourne VIC, Australia) (WSDM'19)*. Association for Computing Machinery, New York, NY, USA, 375–383. https://doi.org/10.1145/3289600.3291029

[44] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *16th International Conference on World Wide Web (WWW'07)*. ACM, 141–150.

[45] Andriy Mnih and Ruslan R. Salakhutdinov. 2008. Probabilistic matrix factorization. In *Advances in Neural Information Processing Systems*. 1257–1264.

[46] Federico Monti, Michael Bronstein, and Xavier Bresson. 2017. Geometric matrix completion with recurrent multi-graph neural networks. In *Advances in Neural Information Processing Systems*. 3697–3707.

[47] Israt Nisa, Aravind Sukumaran-Rajam, Rakshith Kunchum, and P. Sadayappan. 2017. Parallel CCD++ on GPU for matrix factorization. In *General Purpose GPUs*. ACM, 73–83.

[48] Rong Pan, Yunhong Zhou, Bin Cao, Nathan N. Liu, Rajan Lukose, Martin Scholz, and Qiang Yang. 2008. One-class collaborative filtering. In *8th IEEE International Conference on Data Mining*. IEEE, 502–511.

[49] Weixiang Shao, Lifang He, and S. Yu Philip. 2015. Multiple incomplete views clustering via weighted nonnegative matrix factorization with $L_{2,1}$ regularization. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*. Springer, 318–334.

[50] Ling Shen, Richang Hong, Haoran Zhang, Xinmei Tian, and Meng Wang. 2019. Video retrieval with similarity-preserving deep temporal hashing. *ACM Trans. Multimedia Comput., Commun. Applic.* 15, 4 (2019), 1–16.

[51] D. F. Silva, C. M. Yeh, Y. Zhu, G. E. A. P. A. Batista, and E. Keogh. 2019. Fast similarity matrix profile for music analysis and exploration. *IEEE Trans. Multimedia* 21, 1 (2019), 29–38. DOI: https://doi.org/10.1109/TMM.2018.2849563

[52] Konstantinos Slavakis, Georgios B. Giannakis, and Gonzalo Mateos. 2014. Modeling and optimization for big data analytics: (Statistical) learning tools for our era of data deluge. *IEEE Sig. Process. Mag.* 31, 5 (2014), 18–31.

[53] Nathan Srebro, Jason Rennie, and Tommi S. Jaakkola. 2005. Maximum-margin matrix factorization. In *Advances in Neural Information Processing Systems*. 1329–1336.

[54] Wei Tan, Liangliang Cao, and Liana Fong. 2016. Faster and cheaper: Parallelizing large-scale matrix factorization on GPUs. In *25th ACM International Symposium on High-performance Parallel and Distributed Computing*. ACM, 219–230.

[55] George Trigeorgis, Konstantinos Bousmalis, Stefanos Zafeiriou, and Björn W. Schuller. 2016. A deep matrix factorization method for learning attribute representations. *IEEE Trans. Pattern Anal. Mach. Intell.* 39, 3 (2016), 417–429.

[56] Wei Wu, Bin Li, Ling Chen, and Chengqi Zhang. 2017. Consistent weighted sampling made more practical. In *26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 1035–1043.

[57] Kun Xie, Xueping Ning, Xin Wang, Dongliang Xie, Jiannong Cao, Gaogang Xie, and Jigang Wen. 2016. Recover corrupted data in sensor networks: A matrix completion solution. *IEEE Trans. Mob. Comput.* 16, 5 (2016), 1434–1448.

[58] Kun Xie, Lele Wang, Xin Wang, Gaogang Xie, and Jigang Wen. 2017. Low cost and high accuracy data gathering in WSNs with matrix completion. *IEEE Trans. Mob. Comput.* 17, 7 (2017), 1595–1608.

[59] Xiaolong Xie, Wei Tan, Liana L. Fong, and Yun Liang. 2017. CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs. In *26th International Symposium on High-performance Parallel and Distributed Computing*. ACM, 79–92.

[60] Hong-Jian Xue, Xinyu Dai, Jianbing Zhang, Shujian Huang, and Jiajun Chen. 2017. Deep matrix factorization models for recommender systems. In *International Joint Conference on Artificial Intelligence*. 3203–3209.

[61] Shuicheng Yan and Huan Wang. 2009. Semi-supervised learning by sparse representation. In *SIAM International Conference on Data Mining*. SIAM, 792–801.

[62] Chenyun Yu, Sarana Nutanong, Hangyu Li, Cong Wang, and Xingliang Yuan. 2016. A generic method for accelerating LSH-based similarity join processing. *IEEE Trans. Knowl. Data Eng.* 29, 4 (2016), 712–726.

[63] Hyokun Yun, Hsiang-Fu Yu, Cho-Jui Hsieh, S. V. N. Vishwanathan, and Inderjit Dhillon. 2014. NOMAD: Non-locking, stOchastic Multi-machine algorithm for Asynchronous and Decentralized matrix completion. *Proc. VLDB Endow.* 7, 11 (2014), 975–986.

[64] Lefei Zhang, Liangpei Zhang, Bo Du, Jane You, and Dacheng Tao. 2019. Hyperspectral image unsupervised classification by robust manifold matrix factorization. *Inf. Sci.* 485 (2019), 154–169.

[65] Xuyun Zhang, Christopher Leckie, Wanchun Dou, Jinjun Chen, Ramamohanarao Kotagiri, and Zoran Salcic. 2016. Scalable local-recoding anonymization using locality sensitive hashing for big data privacy preservation. In *25th ACM International on Conference on Information and Knowledge Management*. ACM, 1793–1802.

[66] Yiwen Zhang, Kaibin Wang, Qiang He, Feifei Chen, Shuiguang Deng, Zibin Zheng, and Yun Yang. 2019. Covering-based web service quality prediction via neighborhood-aware matrix factorization. *IEEE Trans. Serv. Comput.* (2019), 1–12.

[67] Haitao Zhao and Zhihui Lai. 2019. Neighborhood preserving neural network for fault detection. *Neural Netw.* 109 (2019), 6–18.

[68] Xiaofeng Zhu, Xuelong Li, Shichao Zhang, Zongben Xu, Litao Yu, and Can Wang. 2017. Graph PCA hashing for similarity search. *IEEE Trans. Multimedia* 19, 9 (2017), 2033–2044.

Received February 2020; revised July 2021; accepted November 2021