An Efficient Approach of GPU-accelerated Stochastic Gradient Descent Method for Matrix Factorization

Feng Li, Yunming Ye , Xutao Li

Shenzhen Key Laboratory of Internet Information Collaboration, Harbin Institute of Technology, Shenzhen, China
lifeng@stu.hit.edu.cn, yeyunming@hit.edu.cn, lixutao@hit.edu.cn

Abstract

Matrix Factorization (MF) is a very effective tool for Collaborative Filtering (CF) in recommender systems. As a popular solver, Stochastic Gradient Descent (SGD) is widely utilized to find MF solutions for CF. However, SGD solver often suffers from a very slow optimization process, due to its large computation burden. How to speed up it becomes a very important research topic. One of the main techniques to the problem is partitioning the matrix to factorize into blocks and calculate the factorization parallelly with these blocks. In this paper, we would like to use the most modern computation resource Graphics Processing Unit (GPU) to speed up the partition-based computation. Though there are some studies on partition-based SGD with GPUs, due to the sparsity of matrices in real-life scenarios, these methods produce too many blank blocks, which will waste the GPU computing resources. In this paper, we propose a new method, which can avoid the problem and make use of GPUs more efficiently to speed up the SGD based MF solver.

Keywords: Collaborative filtering, Matrix factorization, Stochastic gradient descent, GPU

1 Introduction

Collaborative filtering (CF) [1], as an important tool in recommender system [2-3], has been widely applied to many Internet services. With the support of CF, Internet service providers can effectively improve their service quality. The basic idea of CF is to infer the users' preferences from their historical behavior data, e.g., consuming a product, clicking a webpage, rating a movie, etc. Many methods have been put forward to address the CF problem, among which matrix factorization (MF) is one of the most widely used algorithms [4-5].

In CF based recommender system, R is an incomplete matrix. We assume that Ω is the observed entries of R. The entry $(u,v)\in\Omega$ denotes that user u has given a rating r_{uv} to item v, where $u=1,2,\cdots,m$ and $v=1,2,\cdots,n$. Accordingly, each user u is associated with a vector $p_u\in\mathbb{R}^k$, and each item v is associated with a vector $q_v\in\mathbb{R}^k$. This approach maps both users and items into a k - dimension latent feature space, and each latent factor contains an abstract level of information. The resulting inner product, $p_u^T q_v$, captures the interaction between user u and item v, where a higher inner product score denotes a better recommendation candidate.

To learn the factors (p_u and q_v), MF optimizes the following objective function:

$$\min \sum_{(u,v)\in\Omega} \left(r_{uv}-p_u^T q_v\right)^2+\lambda\left(\|p_u\|^2+\|q_v\|^2\right) \quad (1)$$

where the parameter λ controls the importance of regularization terms. To solve the optimization problem (1), it needs to randomly select a (u,v) and applies the gradient based rules:

$$\begin{aligned} p_u &\leftarrow p_u+\gamma\left[\left(r_{uv}-p_u^T q_v\right)q_v-\lambda p_u\right] \\ q_v &\leftarrow q_v+\gamma\left[\left(r_{uv}-p_u^T q_v\right)p_u-\lambda q_v\right] \end{aligned} \quad (2)$$

where γ is the learning rate. This approach is called Stochastic Gradient Descent (SGD). Its time complexity is $O(|\Omega|kt)$ and $|\Omega|$ denotes the cardinality of set Ω, where t indicates the number of iterations, which can be costly if $|\Omega|$ is too large.

Simply paralleling the SGD procedure can lead to the loss of update problem, i.e., the latent factor of a user u can be simultaneously updated by many pairs (u,v) and some update will be lost because of asynchronization. The iterative process of (2) is inherently sequential. To parallelized SGD under parallel architecture for large-scale data sets, several parallel SGD approaches have been proposed. The core idea of them is to address the loss of update problem.

In the past few years, Graphics Processing Unit (GPU) becomes a very flexible and powerful computing resources [6-8]. Jin et al. develop an SGD

*Corresponding Author: Yunming Ye; E-mail: yeyunming@hit.edu.cn
DOI: 10.3966/160792642019072004009

method on a GPU, which is called GPUSGD [9]. To utilize the computing power of a GPU, the rating matrix is divided into $l \times l$ blocks and every block is of size $z \times z$. Within a block of the rating matrix, the rating element is also divided into several groups, and any two elements in the same group share neither the same row nor the same column. By doing so, every thread of the GPU will update a user vector and an item vector with no update loss.

However, there is one important challenge to solve the problem. The partitioning method expands the original matrix into a square matrix with missing values. It may lead to many blank blocks when the number of users differs greatly from the number of items, which will cause a significant waste of GPU computing resources.

In this paper, we follow the main idea of GPUSGD and propose a new method. In order to solve the problem of waste of computing resources, we design a new computing resource allocation scheme to effectively avoid the effects of blank blocks. The new method will no longer allocate computing resources for blank blocks. The contributions of this paper are summarized as follows:

(1) We analyze the defects of GPUSGD from the perspective of hardware computing.

(2) According to the analysis of the defects, a new scheduling scheme is proposed. The new method avoids blank blocks and can make more efficient use of the GPU.

The rest of the paper is organized as follows: Section 2 presents the related works. Section 3 introduces the GPUSGD. Section 4 describes the improved GPUSGD algorithm. The experimental results are presented in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

In this section, we will review the related works. First, we introduce the related studies of solving MF. Next, we analyze the losing update problem of SGD, and review two existing studies to overcome this problem.

2.1 MF Methods

The MF in CF is different from the conventional factorization method utilized in image or text analysis [10-13]. The key difference is that CF has to deal with an incomplete matrix. Hence, we cannot apply the conventional linear algebra methods, such as QR decomposition, to find its factorization. Many methods have been proposed to the problem, such as alternating least squares (ALS) [14], coordinate descent (CD) [15, 16], and SGD. Among them, SGD is the most widely used for MF based CF.

2.2 Parallel Stochastic Gradient Descent Algorithm

From Figure 1, we can see that when the training data r_{uv} is used, p_u, q_v, a_u and b_v will be updated simultaneously. If there is another processor to process the training data r_{uv} at the same time, p_u and a_u will also be updated. It will create conflicts and lead to losing updates. In order to implement the SGD in parallel, we should solve this problem first.

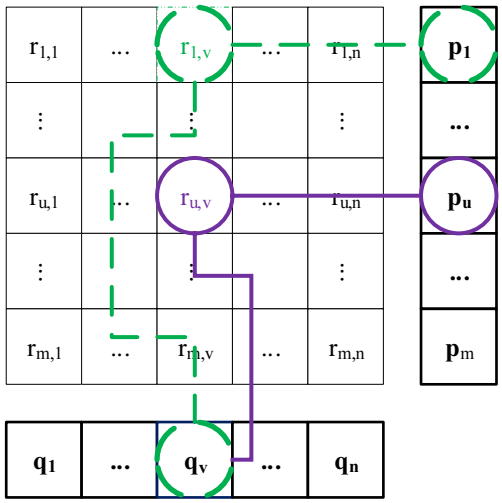


Figure 1. Relationship between training rating data and update results. User associated vector, item associated vector will be updated simultaneously

From Figure 2, we can see that if two elements in the rating matrix share neither the same row nor the same column, losing update will not happen. Hence, the key issue now is to make sure that the elements concurrently processed are not from the same row or the same column. There are two ways which are widely used. Now we will introduce them.

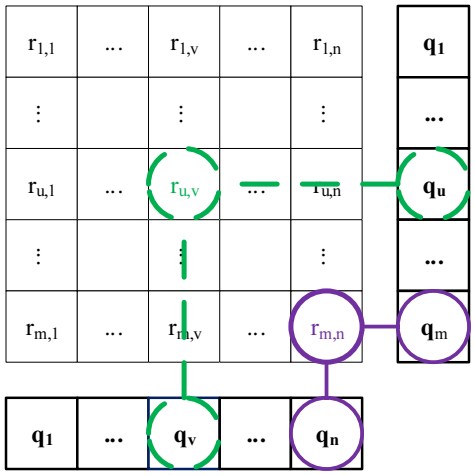


Figure 2. Relationship between training rating data and update results. If two elements in the rating matrix share neither the same row nor the same column, losing update will not occur

2.2.1 HogWild

HogWild [17] is a method that makes use of high sparsity of the incomplete matrix to avoid the losing update problem. If the rating matrix to factorize is highly sparse, we can deduce that two randomly sampled ratings are unlikely from the same row or column. In other words, if the matrix is sparse enough, the losing update problem rarely happens, or can be ignored even it happens. Hence, HogWild drops the synchronization that prevents concurrent variable access via atomic operations, each of which is a series of instructions that cannot be interrupted. Though the losing update problem may still occur, the convergence of HogWild can be shown under some mild condition (e.g., the rating matrix is extremely sparse).

2.2.2 Partition Based Methods

Another strategy to avoid the losing update problem is to partition the rating matrix into blocks, which are named as partition-based methods. As there always exist some blocks that share neither the same rows or columns, we can thus select these blocks to update in parallel. Specifically, partition-based methods uniformly grid the rating matrix R into many blocks and apply the SGD to some independent blocks (here 'independent' means the blocks share no rows or columns). Based on this idea, SGD on different parallel platforms are proposed [9, 18-20].

Fast parallel SGD (FPSGD) is executed on a shared-memory system with several concurrent threads. FPSGD exploits a strategy of keeping all threads busy in running. If the number of concurrent threads is t, R is partitioned into at least $(t+1)\times(t+1)$ blocks uniformly. Once a thread finishes processing a certain block, a new block that share neither same row nor column with the $(t-1)$ block being processing will be assigned to this thread. Every block will be labelled with a tag, which records how many times they were processed. When a new task is assigned to a thread, FPSGD selects the blocks with the least times being processed in all blocks that meet the conditions share neither same row or same column with the block being processed by other threads. If R is partitioned into only $t\times t$ blocks, there is only one block that share neither same row nor column with the $(t-1)$ block being processing, and it is the block just being processed. In this way, other data blocks will never be calculated. FDSGD is an implement of SGD on distribute system, and it is similar as FPSGD.

GPU is different from traditional shared-memory system that all the threads of GPU will execute concurrently. The tasks of all threads must end together and assign new tasks together. All threads will be assigned to a new task until they finish the previous task. Hence, there is no need to assign extra free blocks as FPSGD. Typical method of this category is GPUSGD, which is proposed by Jin et al. [9]. However, GPUSGD will produce blank blocks, which cause a significant waste of GPUs.

3 GPUSGD Algorithm

GPU has many threads, and these threads are organized into thread blocks. Based on the architecture of GPU, GPUSGD has two levels of parallelism. The first level is that the blocks of the rating matrix that share neither the same row nor the same column can be processed by different thread blocks GPU. The second level is that the elements in the same block that share neither the same row nor the same column can be processed by different threads in a thread block.

In the following of this section, we will introduce the two levels of parallelism to implement GPUSGD. The core component to the implementation is labelling data blocks and data items with tags. Data blocks with the same tag can be processed at the same time by different thread blocks and data items in a data block with the same tag can also be processed at the same time by different threads in the same thread block.

3.1 Task Assignment of Thread Blocks

If we use t thread blocks to process, the rating matrix is divided into $t\times t$ blocks with size $z\times z$ first and this produces $t!$ patterns, where t patterns cover all blocks. For example, in Figure 3 the rating matrix divided into 3×3 produces six patterns, and the three patterns in the first row can cover all blocks.

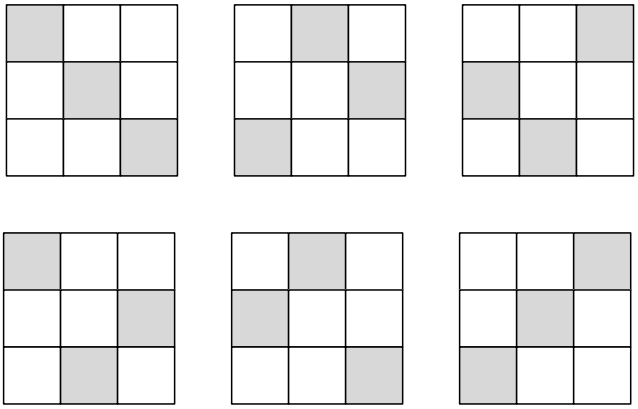


Figure 3. Six rules of a matrix divided into 3×3

To exploit this feature, we could label every block with a tag. Every block has a 2-dimension ID, denoted by its index (i,j), where $1\le i\le t$ and $1\le j\le t$. Hence, we can label block (i,j) by $(j-i+t)\%t$. Thus, t tags from 0 to $t-1$ will label all the blocks. It can be proved that blocks with the same tag share neither the same rows or columns. In every round of iteration, the task of each thread block can be divided into t stages. Each stage handles data blocks with one tag. After a

data block is processed, all thread blocks need to synchronize.

3.2 Task Assignment of Threads

In every stage, the threads in a thread block should process all the data in a block. In order to maximize the use of multi-threaded computing power, each piece of data in the data block also needs to be tagged with a tag.

A rating (u,v) can be labelled with $(v\%z-u\%z+z)\%z$. Thus, z tags from 0 to $z-1$ will label all the blocks. It can be proved that the ratings with the same tag are from different rows or columns. They can be processed by different threads in a thread block.

In GPU computing, continuous threads can read continuously stored data at once, which is called coalesced memory access. In order to speed up data accessing, GPUSGD also organizes the rating matrix and stores data with the same tag continuously.

4 New Strategy of Data Partitioning

In Section 3, we introduce the basic idea of GPUSGD and illustrate the algorithm with a few simple examples. However, in the actual recommendation system, there is a big challenge, that is, the number of users and the number of items are often very different. Sometimes, the number of users is thousands of times as the number of items, or vice versa. In other words, the rating matrix is not a square matrix.

4.1 Data Partitioning and Problem

GPUSGD divides R into $t\times t$ blocks, and it needs every block to be a square matrix. The missing value is used to expand R into a square matrix and then partition it. For an $m\times n$ matrix R, we should first select a relatively large one in m and n, and denoted as k. Then we expand k to the smallest integer that can be divisible by t, and denoted as K. Now, we can expand R into a $K\times K$ matrix using missing value.

For example, a 7×3 rating matrix is shown in Figure 4. The shaded elements in the figure represent observed entries, and the unshaded elements represent missing values. If we want to divide it into 3×3 blocks, the matrix should be expanded to a 9×9 square matrix as shown in Figure 5.

When we divide the expanded rating matrix into 3×3, it will generate too many blank matrix. It will cause many threads to be idle and wait after distributing data to the threads of GPU based on this strategy.

To illustrate the problem, we use the matrix in Figure 6 as an example. In this example, there are not as many filled elements as in Figure 5. If we use GPU to process this matrix as in GPUSGD, there is a thread block that handles empty matrix blocks in every round as shown in Figure 7.

1	2	3
4	5	6
7	8	9
10	11	12
13	14	15
16	17	18
19	20	21

Figure 4. 7×3 rating matrix that is not a square matrix

1	2	3						
4	5	6						
7	8	9						
10	11	12						
13	14	15						
16	17	18						
19	20	21						

Figure 5. Expanding the 7×3 rating matrix to square matrix with missing values

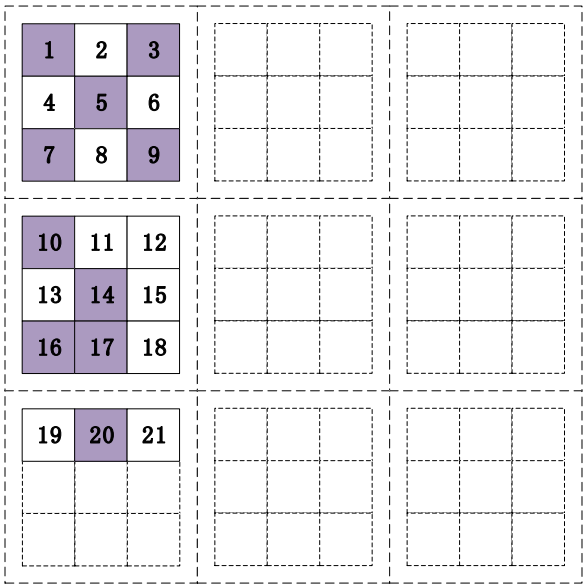


Figure 6. The expanded matrix is divided into 3×3

(1, 1) 0	(1, 2) 1	(1, 3) 2	(1, 4) 3
(2, 1) 3	(2, 2) 0	(2, 3) 1	(2, 4) 2
(3, 1) 2	(3, 2) 3	(3, 3) 0	(3, 4) 1
(4, 1) 1	(4, 2) 2	(4, 3) 3	(4, 4) 0

Figure 7. An expanded rating matrix divided into 4×4. Dotted lines represent expanded blocks

Once a task is launched, the GPU generates the corresponding grid of threads. As we discussed in previous section, these threads are assigned to execution resources on a block-by-block basis. In the current generation of GPU, the execution resources are organized into streaming multiprocessors (SMs). Multiple thread blocks can be assigned to each SM.

With a limited number of SMs and a limited number of blocks that can be assigned to each SM, there is a limit on the number of blocks that can be actively executing in GPU. Most grids contain many blocks than this number. The runtime system maintains a list of blocks that need to execute and assigns new blocks to SMs as they complete executing the blocks previously assigned to them.

Dealing with the blank blocks consumes SMs, which causes a waste. In order to further improve the computational efficiency, we must think of ways to skip these blank blocks during the calculation process.

4.2 Improved Strategy

The thread block's tasks of GPUSGD is shown in Figure 8. We can reassemble the tasks, as shown in Figure 9. By doing this, we can effectively reduce the number of thread blocks.

Now we introduce the new partitioning and assigning strategy. Given a block number t, the number of users m and the number of items n. If $m > n$, the user can be divided into t groups and the items can be divided into $s = \left\lceil \frac{n \times t}{m} \right\rceil$ groups. Based on this strategy, the rating matrix can be divided into $t \times s$, and every block is $z \times z$, where $z = \left\lceil \frac{t}{m} \right\rceil$.

Every block has a 2-dimension ID, represented by (i, j), where $1 \le i \le t$ and $1 \le j \le s$. Every block should be labelled by $(j - i + t)\%t$. We can assign s thread blocks to process this rating matrix, and every thread block will process t blocks.

With this method, the new partitioning strategy will not produce blank block. As a result, the method can avoid completely waiting thread blocks. We note that, within a thread block, the data should be labelled with a tag as GPSUGD. We named the improved method as efficient GPUSGD (eGPUSGD). The overall workflow of eGPUSGD can be seen in Figure 10 which is composed three key steps, namely, data partitioning, data organization and task assignment. The shaded elements in the figure represent observed values as Figure 4.

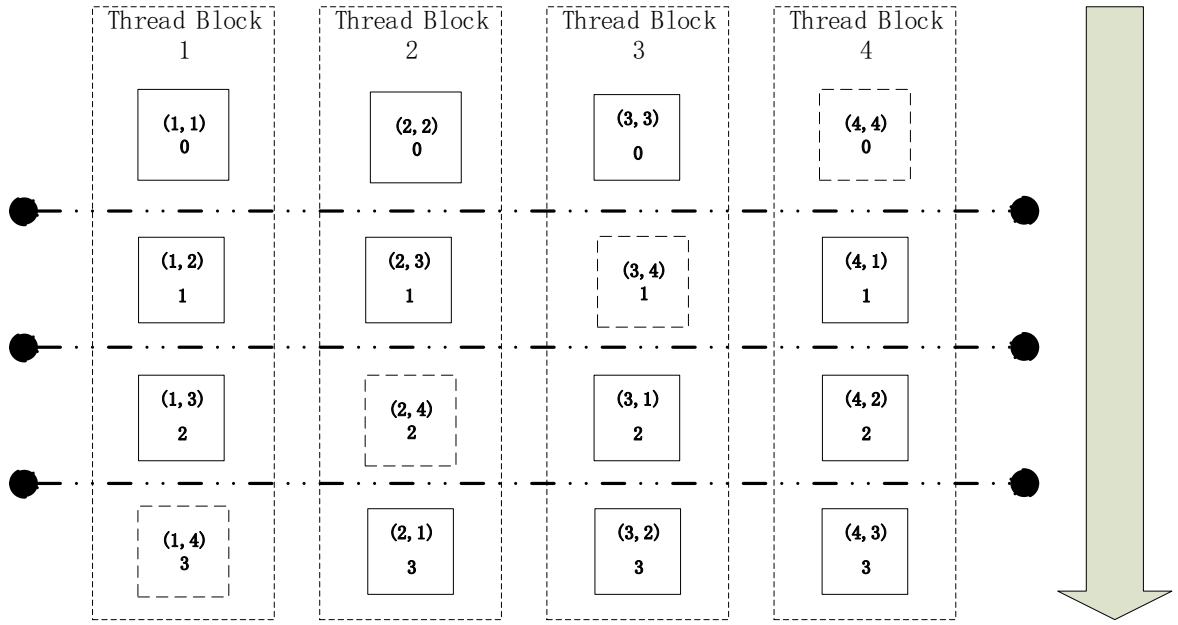


Figure 8. The task of every thread block when using GPUSGD

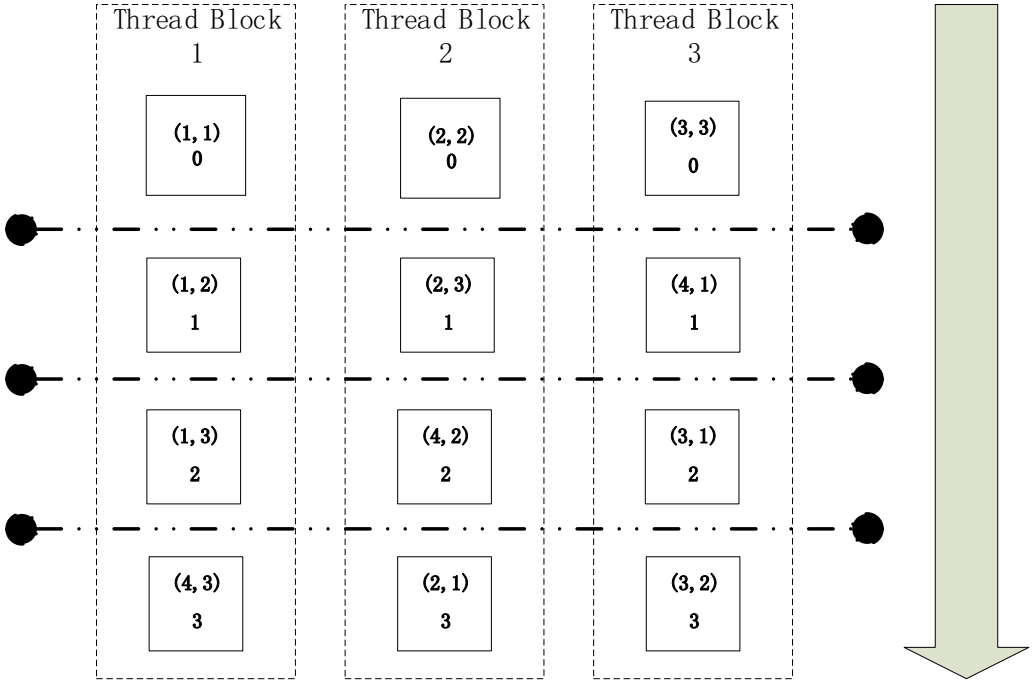


Figure 9. The reassembled task of every thread block

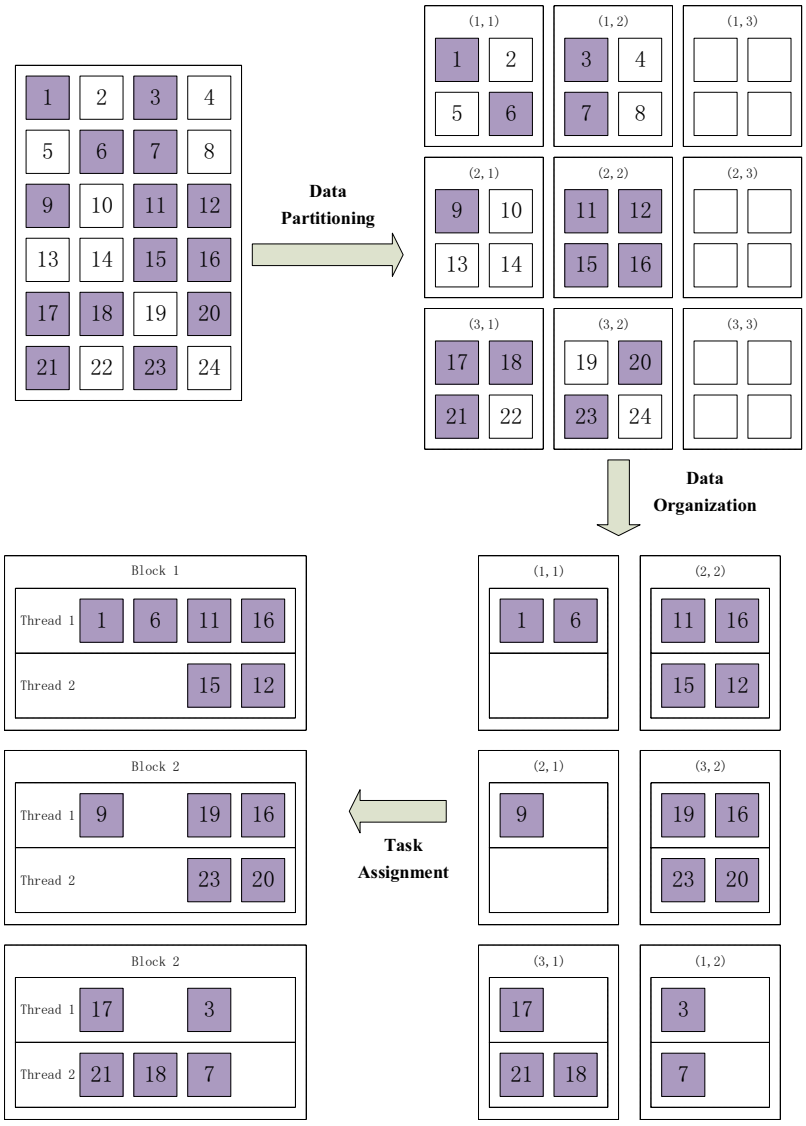


Figure 10. The overall workflow of eGPUSGD

5 Experiment

This section tests the proposed method on real dataset and analyze the experimental results. The performance of the proposed GPUSGD method is evaluated on four datasets: MovieLens10M(M10M), M100k, Netflix, and BDMovie. These datasets are all publicly available on the Internet. The statistics of these datasets are summarized in Table 1.

Table 1. The statistics of datasets

	M10M	M100k	Netflix	BDMovie
#users	71567	943	475749	9721
#items	65133	1682	17770	7889
#rating	9301274	80000	9301274	1199604

The experiments are performed on a computer with an NVIDIA GX1080TI and CUDA-SDK 9.0. In the following, the performance of the proposed SGD on GTX1080Ti is first presented, and the computation time is then compared to GPUSGD on the same platform.

We fix 100 as the maximum number of iterations, γ is set to 0.005 and λ is set to 0.03. The observed experimental results are from two precisions, single and double.

First, we test the convergence of the iterations when the number of blocks is different. For evaluation, we adopt the widely utilized root-mean-square error (RMSE):

$$RMSE = \sqrt{\frac{1}{|V|}\sum_{(u,v)\in V}\left(\hat{r}_{uv} - r_{uv}\right)^2} \tag{3}$$

where V is the validation set. RMSE is reported as the number of iterations is increased in Figure 10. We fix the latent dimensions at 16 and block the rating matrix into 32×32. We observe the experimental results from both single-precision and double-precision perspectives. From Figure 11, we can see that the convergence of the proposed eGPUSGD is convergent, which is the same as the existing GPUSGD method.

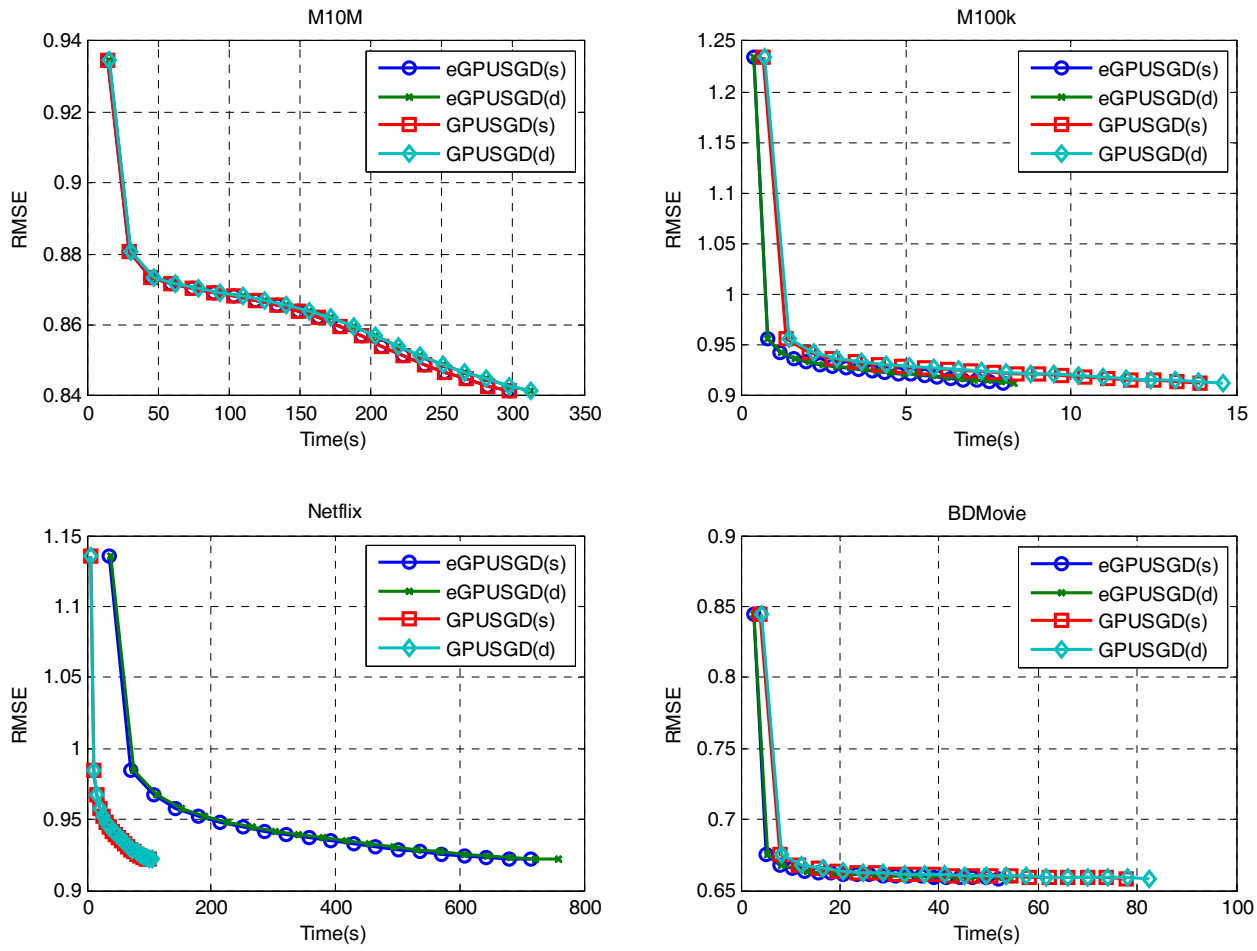


Figure 11. The RMSE trend

Next, we evaluate the efficiency of the proposed method. We fix the latent dimensions at 16 and 32. The rating matrix is block into 32×32, 64×64, 128×128 and 256×256 separately. Figure 12 reports the time consumptions when different sizes of blocks are utilized.

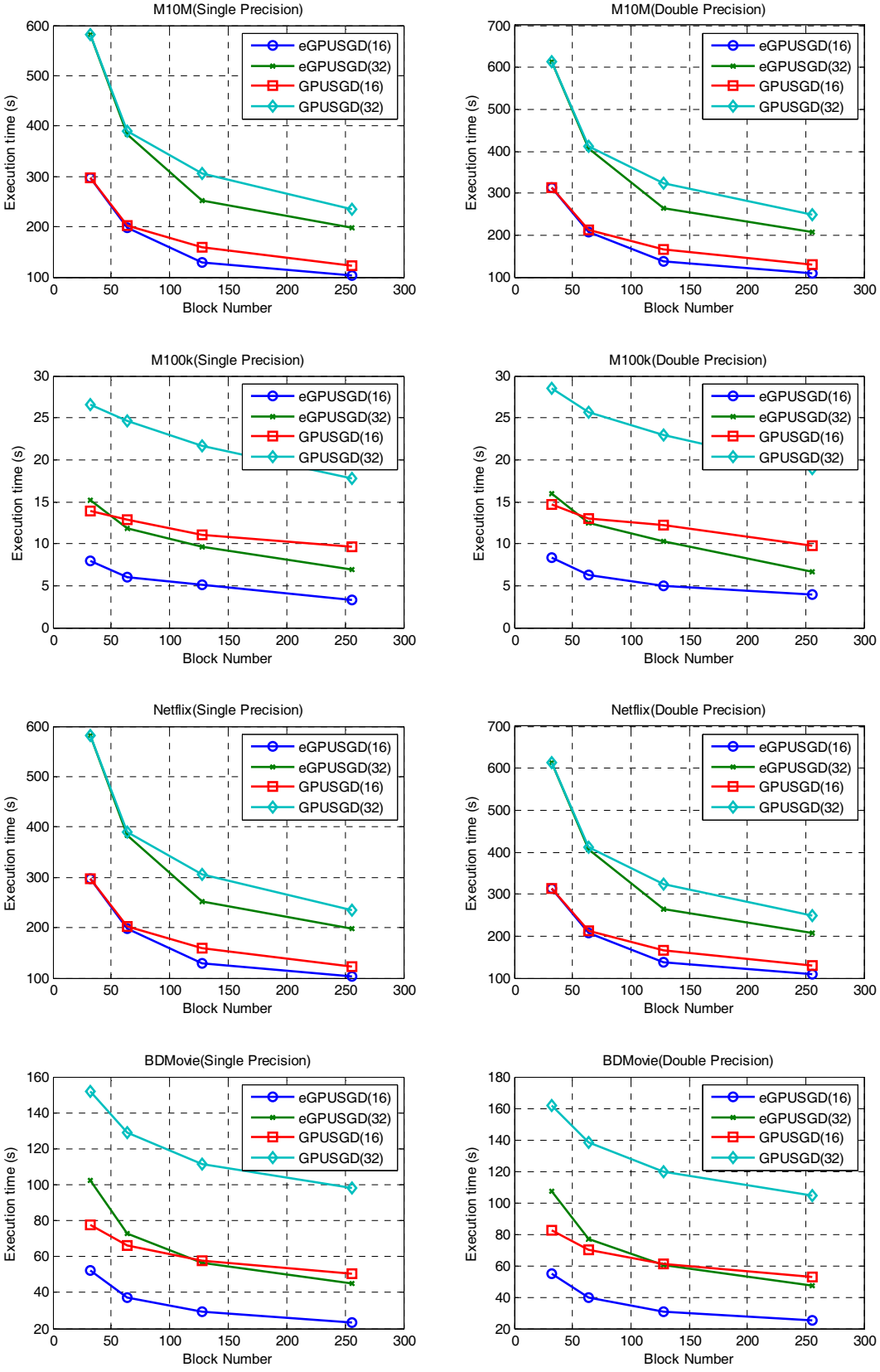


Figure 12. Time consumption of different sizes of blocks

From Figure 12, we can see that the efficiency of the new method is better than the old method in almost all cases. When the M10M is blocked into 32×32, there is no blank blocks, so the method has the same time consumption as the old method. For the dataset Netflix, the blank blocks is more so the new method is much better than the old method.

Finally, we block the matrix into 32×32 and 256×256 and test how the number of latent dimensions affects the time consumption. The number of latent dimensions ranges in 16,32,64 and 128. Figure 13 depicts the results.

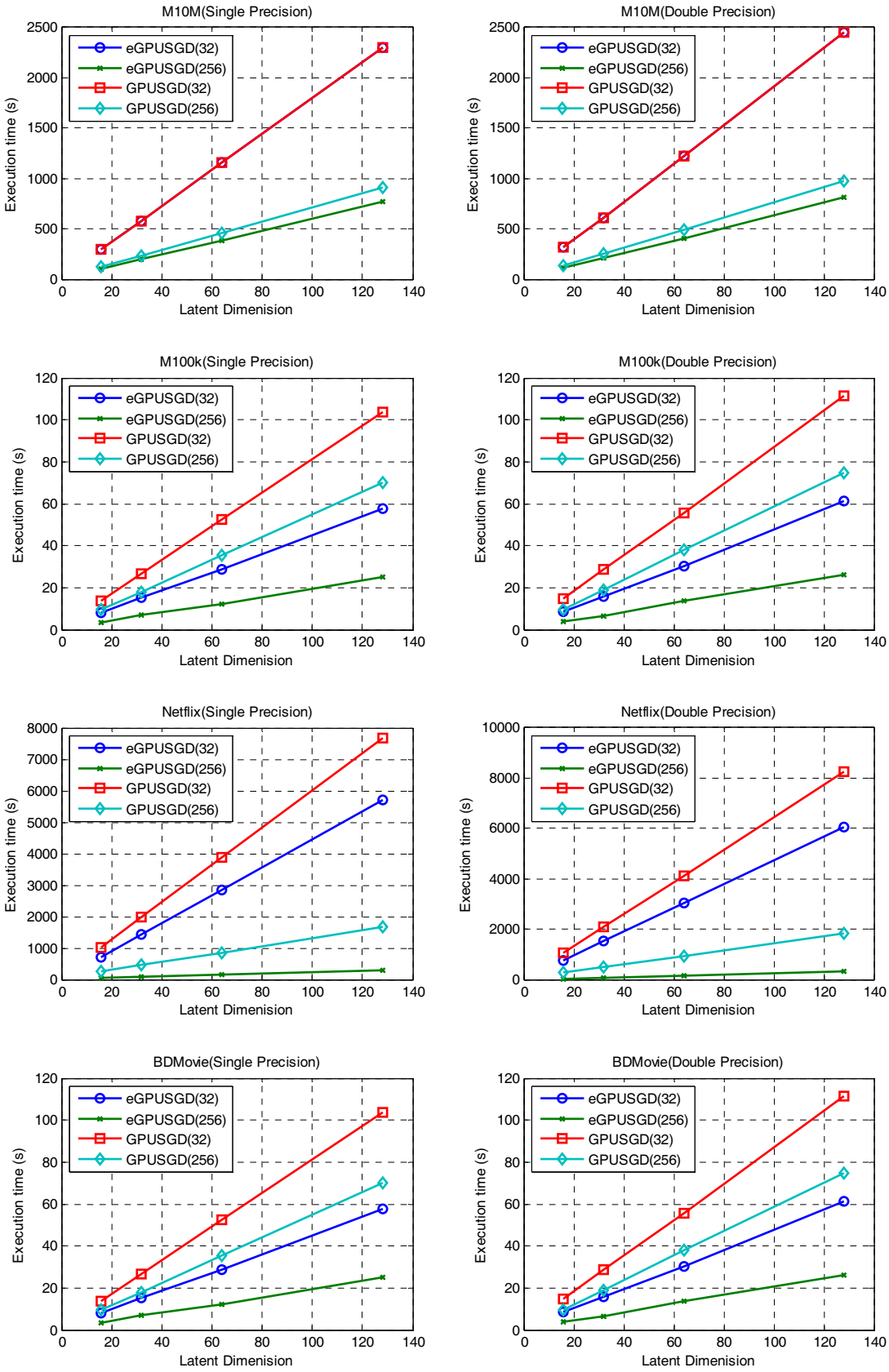


Figure 13. Time consumption of different latent dimensions

From Figure 13, we can see that the efficiency of eGPUSGD is better than GPUSGD in the case of the same parameters.

6 Conclusion

Matrix Factorization based Collaborative Filtering has been widely used in many recommender systems.

SGD is one of the most popular algorithms for solving matrix factorization with missing value. However, the large computational burden required by SGD poses a significant efficiency challenge. In this paper, we improve the existing GPUSGD method to further improve its efficiency. The results on four real-world data sets show that the proposed algorithm eGPUSGD can make a very good use of the massively parallel GPU architecture and improve the efficiency of GPUSGD significantly.

Acknowledgments

This research was supported in part by NSFC under Grant No. 61572158 and 61602132, Shenzhen Science and Technology Program under Grant No. JCYJ20160 330163900579 and and JCYJ20170811160212033.

References

[1] F. Cacheda, V. Carneiro, D. Fernández, V. Formoso, Comparison of Collaborative Filtering Algorithms: Limitations of Current Techniques and Proposals for Scalable, High-performance Recommender Systems, *ACM Transactions on the Web*, Vol. 5, No. 1, pp. 1-33, February, 2011.

[2] P. Resnick, H. R. Varian, Recommender Systems, *Communications of the ACM*, Vol. 40, No. 3, pp. 56-58, March, 1997.

[3] T. Gopalakrishnan, P. Sengottuvelan, A. Bharathi, R. Lokeshkumar, An Approach To Webpage Prediction Method Using Variable Order Markov Model In Recommendation Systems, *Journal of Internet Technology*, Vol. 19, No. 2, pp. 415-424, March, 2018.

[4] Y. Koren, R. Bell, C. Volinsky, Matrix Factorization Techniques for Recommender Systems, *Computer*, Vol. 42, No. 8, pp. 42-49, August, 2009.

[5] Z. Zhang, Y. Liu. A List-wise Matrix Factorization Based POI Recommendation by Fusing Multi-Tag, Social and Geographical Influences, *Journal of Internet Technology*, Vol. 19, No. 1, pp. 127-136, January, 2018.

[6] D. Steinkraus, I. Buck, P. Y. Simard, Using GPUs for Machine Learning Algorithms, *Proceedings of 8th* International *Conference on Document Analysis and Recognition, Seoul*, Korea, 2005, pp. 1115-1120.

[7] W. B. Jiang, B. Luo, H. Jin, A. L. Yuille, J. S. Xiao, A Novel Parallelized Feature Extraction in Grouped Scale Space Based on Graphic Processing Units, *Journal of Internet Technology*, Vol. 17, No. 5, pp. 1061-1069, September, 2016.

[8] F. Li, Y. Ye, Z. Tian, X. Zhang. CPU versus GPU: Which Can Perform Matrix Computation Faster-Performance Comparison for Basic Linear Algebra Subprograms, *Neural Computing and Applications*, https://doi.org/10.1007/s00521-018-3354-z, January, 2018.

[9] J. Jin, S. Lai, S. Hu, J. Lin, X. Lin, GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization, *Concurrency and Computation: Practice and Experience*, Vol. 28, No. 14, pp. 3844-3865, December, 2016.

[10] V. P. Pauca, J. Piper, R. J. Plemmons, Nonnegative Matrix Factorization for Spectral Data Analysis, *Linear Algebra and Its Applications*, Vol. 416, No. 1, pp. 29-47, July, 2006.

[11] F. Shahnaz, M. W. Berry, V. P. Pauca, R. J. Plemmons, Document Clustering Using Nonnegative Matrix Factorization, *Information Processing & Management*, Vol. 42, No. 2, pp. 373-386, March, 2006.

[12] S. A. Vavasis, On the Complexity of Nonnegative Matrix Factorization, *SIAM Journal on Optimization*, Vol. 20, No. 3, pp. 1364-1377, August, 2009.

[13] L. Eldén, *Matrix Methods in Data Mining and Pattern Recognition*, Society for Industrial and Applied Mathematics, 2007.

[14] I. Pilászy, D. Zibriczky, D. Tikk, Fast Als-based Matrix Factorization for Explicit and Implicit Feedback Datasets, *Proceedings of the 4th ACM conference on Recommender systems*, Barcelona, Spain, 2010, pp. 71-78.

[15] H. F. Yu, C. J. Hsieh, S. Si, I. S. Dhillon, Parallel Matrix Factorization for Recommender Systems, *Knowledge and Information Systems*, Vol. 41, No. 3, pp. 793-819, December, 2014.

[16] H. F. Yu, C. J. Hsieh, S. Si, I. S. Dhillon, Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems, *Proceedings of the 12th International Conference on Data Mining*, Brussels, Belgium, 2012, pp. 765-774.

[17] B. Recht, C. Re, S. Wright, F. Niu, Hogwild: A Lock-free Approach to Parallelizing Stochastic Gradient Descent, *Proceedings of the 25th Annual Conference on Neural Information Processing Systems (NIPS)*, Granada, Spain, 2011, pp. 693-701.

[18] R. Gemulla, E. Nijkamp, P. J. Haas, Y. Sismanis, Large-scale Matrix Factorization with Distributed Stochastic Gradient Descent, *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, 2011, pp. 69-77.

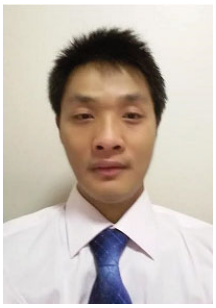
[19] Y. Zhuang, W. S. Chin, Y. C. Juan, C. J. Lin, A Fast Parallel SGD for Matrix Factorization in Shared Memory Systems, *Proceedings of the 7th ACM Conference on Recommender Systems*, Hong Kong, China, 2013, pp. 249-256.

[20] F. Li, B. Wu, L. Xu, C. Shi, J. Shi, A Fast Distributed Stochastic Gradient Descent Algorithm for Matrix Factorization, *Proceedings of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, New York, NY, 2014, pp. 77-87.

Biographies

Feng Li is now a Ph.D. student in the Harbin Institute of Technology Shenzhen Graduate School. He received the Master degree in Computer Science and the Bachelor degree in Mathematics from Harbin Institute of Technology in 2013 and 2011. His research interests include high performance computing and randomized algorithms.

Yunming Ye is a Professor in the Harbin Institute of Technology Shenzhen Graduate School. He received the Ph.D. degree in Computer Science from Shanghai Jiao Tong University. His research interests include data mining, text mining, and ensemble learning algorithms.



Xutao Li is now an associate professor in the Shenzhen Graduate School, Harbin Institute of Technology. He received the PhD degrees in Computer Science from Harbin Institute of Technology in 2013. His research interests include machine learning and graph mining, especially tensor based learning and mining algorithms.