International Journal of Innovative Computing, Information and Control
Volume **15**, Number **2**, April **2019**

ICIC International ©2019 ISSN 1349-4198
pp. **697–711**

A NEW APPROACH OF GPU-ACCELERATED STOCHASTIC GRADIENT DESCENT METHOD FOR MATRIX FACTORIZATION

Feng Li, Yunming Ye, Xutao Li and Jiajie Lu

Shenzhen Key Laboratory of Internet Information Collaboration
Harbin Institute of Technology Shenzhen Graduate School
HIT Campus of University Town of Shenzhen, Shenzhen 518055, P. R. China
lifeng@stu.hit.edu.cn; { yeyunming; lixutao }@hit.edu.cn; ljj316@126.com

Received June 2018; revised October 2018

Abstract. *Matrix-factorization- (MF-) based collaborative filtering (CF) is known to be an effective approach to recommendation, which has been widely used in many recommender systems. Stochastic gradient descent (SGD) is one of the most popular algorithms for solving MF-based CF. However, the large computational burden required by SGD poses a challenge of accelerating the SGD process. In the past few years, the graphics processing unit (GPU) has evolved into a very flexible and powerful computing resource. SGD methods for GPUs exist, and the main job is to find the parallelism in the calculation. However, existing parallel SGD approaches ignore the inherent parallelism of vector computation and so do not use the characteristic that a GPU is suitable for vector and matrix computing. In this paper, we aim to design an approach using the inherent parallelism of vector computation to exploit the large-scale parallelization features of a massively multi-threaded GPU to perform SGD. We make full use of the characteristic that GPU is suitable for vector and matrix computing to design the parallel SGD algorithm. Experimental results demonstrate that the proposed method can be well suited for the massively parallel GPU architecture and can outperform the existing method.*

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

1. Introduction.

Nowadays, the content of the Internet is growing exponentially and the variety of content is making Internet services increasingly more complex. In such an environment, users have difficulty in finding content of interest within the tremendous amount of information. Hence, a great number of Internet service companies have developed recommender systems [1, 2, 3] to help their customers find information more efficiently.

Collaborative filtering (CF) [4, 5], as a key technique in recommender systems, has been successfully utilized in many applications. The basic idea of CF employs the similarity of user behaviors to build a profile of each user to make recommendations. For example, to establish a profile of interests, each user of a CF system rates items they have experienced. Then, the CF system matches the user with people of similar interests or tastes. Ratings from those like-minded people are used to generate recommendations for the given user. The fundamental assumption behind this method is that other users' opinions can be selected and aggregated in such a way to provide a reasonable prediction of the active user's preferences.

Matrix factorization (MF) is used in many data-mining and pattern-recognition problems [6, 7, 8, 9], such as image recognition and text analysis. MF can also be used for CF [10]. The matrix to factorize is an incomplete matrix consisting of the ratings that users

DOI: 10.24507/ijicic.15.02.697

have given to items such as books, songs, and movies. MF-based CF maps both users and items to a joint latent factor space, such that user-item interactions are modeled as inner products in that space. MF characterizes both items and users by the vector factors inferred from item-rating patterns. High inner product results between item and user factors lead to a recommendation.

Different from MF used in image recognition and text analysis, MF for CF must process missing values because the matrix is an incomplete matrix. Therefore, the traditional linear algebra method cannot be used to factorize the matrix. Scholars have proposed many methods for solving the problem of MF with missing values [10, 11, 12], the most popular of which is stochastic gradient descent (SGD), which is widely used in modern machine learning applications [13, 14, 15]. Although promising recommendation performance can be delivered, calculation efficiency is still a problem worth exploring.

In the past few years, the graphics processing unit (GPU) has evolved into a very flexible and powerful computing resource [16]. Originally designed as a specialized device, a GPU has now evolved into a highly parallel, multi-threaded, many-core processor with tremendous computational horsepower and high memory bandwidth. NVIDIA's GPU with CUDA (compute unified device architecture) provides a standard C-like interface that is quite easy to use. Owing to the great contribution of CUDA, a number of complex computational problems have been significantly accelerated. Solving MF on the GPU currently has many achievements [17, 18, 19]. It is also possible to make use of a GPU to improve the computation efficiency of an SGD-based MF method for CF [20, 21]. SGD methods exist for GPUs, but the existing approaches do not use the characteristic that a GPU is suitable for vector and matrix computing [22].

In this paper, we aim to exploit the large-scale parallelization features of a massively multi-threaded GPU to perform SGD. The contributions of the paper are the following.

(1) We analyze the difference between the existing SGD methods for GPUs and other gradient descent methods implemented on the GPU. We imitate the implementation of other gradient descent methods to design the parallel SGD approach. The new parallel SGD approach is designed with the inherent parallelism of vector computation and hence makes full use of the characteristic that a GPU is suitable for vector and matrix computing.

(2) In order to increase parallel computing, we partition the rating matrix to find elements that are not in the same row or column. We present a new method of partitioning the rating matrix that will not create blank blocks.

(3) We design the kernel function to implement the proposed new parallel SGD approach. The kernel function is designed to finish all the work of iteration on the GPU. At the same time, caching use is also optimized.

We introduce a matrix partition-based SGD method on GPUs to address the key issues.

The rest of the paper is organized as follows. In Section 2, we present background knowledge and related work. In Section 3, we introduce the GPU computing model with the existing approach for the SGD method, and in Section 4 we describe the proposed GPU accelerated SGD algorithm. The experimental results are presented in Section 5. Section 6 concludes the paper.

2. Related Work.

We first introduce MF-based CF, and then the SGD solver. Finally, we analyze why it is difficult to implement SGD on GPUs and review two of the most widely used ways of overcoming the problems.

2.1. MF-based CF.

In the context of recommender systems, R is an incomplete matrix and denotes the interaction between users and items, e.g., the rating matrix, where m is the number of users and n is the number of items. We assume that Ω is the observed entries of R.

The entry $(u, v) \in \Omega$ implies that user u has given a rating r_{uv} to item v, where $u = 1, 2, \ldots, m$ and $v = 1, 2, \ldots, n$. Accordingly, each user u is associated with a vector $p_u \in \mathbb{R}^f$, and each item v is associated with a vector $q_v \in \mathbb{R}^f$. For a given user u, the vector p_u indicates their preference, the element of which denotes their interest to a latent group, which can be positive, zero, or negative; for a given item v, each element of q_v represents how likely it belongs to the latent group, which can also be positive, zero, or negative. The resulting inner product, $p_u^T q_v$, captures the interaction between user u and item v over all the latent groups. By using the latent factors, we can approximate the rating interaction between user u and item v as

$$r_{uv} \approx p_u^T q_v \tag{1}$$

For an unknown rating, one can find the inner product by the vector of the corresponding user and item. This will result in recommendations based on the results.

2.2. **SGD algorithm.**

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

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

The constant λ controls the extent of regularization and is usually determined by cross-validation.

Instead of calculating the gradient on the whole training set, we can randomly pick up one sample from the training set and update using the gradient on that particular sample:

$$\begin{aligned} p_u &\leftarrow p_u + \gamma \left[\left(r_{uv} - p_u^T q_v\right) q_v - \lambda p_u\right] \\ q_v &\leftarrow q_v + \gamma \left[\left(r_{uv} - p_u^T q_v\right) p_u - \lambda q_v\right] \end{aligned} \tag{3}$$

where γ is the iteration step and can be set manually and adjusted automatically. This approach is called the SGD algorithm.

2.3. **Parallel SGD algorithm.**

We hope to use parallel hardware to solve SGD but solving it in parallel will introduce the problem of losing updates. When the training data r_{uv} are used, p_u and q_v will be updated simultaneously. If we use another processor to process the training data r_{uv}, p_u or q_v will also be updated. This will create conflict and lead to loss of updates. In order to implement the SGD in parallel, we should solve this problem first.

The problem of losing updates is not inevitable. If two elements in the rating matrix are not in the same row or column, loss of updates will not occur.

In order to make the elements that are processed concurrently not in the same row or column, the two following methods are widely used.

2.3.1. *HogWild.*

HogWild [23] assumes that the rating matrix is highly sparse and deduces that, for two randomly sampled ratings, the four serial updates via Equation (3) are likely to be independent. The reason is that the selected ratings to be updated almost never share the same user identity and item identity. That is to say, iterations of SGD, Equation (3), can be executed in parallel in different threads. HogWild drops the synchronization that prevents concurrent variable access via atomic operations, each of which is a series of instructions that cannot be interrupted. Even though potential overwriting may occur, the convergence is proved under some assumptions, such as the rating matrix being very sparse. There are also parallel solutions based on HogWild [21].

2.3.2. *Partitioning rating matrix.* The HogWild algorithm uses overly optimistic assumptions. Different from HogWild, the partitioning rating matrix takes the property that some blocks of the rating matrix are mutually independent, and their corresponding variables can be updated in parallel. The partitioning rating matrix uniformly grids the rating matrix R into many sub-matrices (also called blocks) and applies SGD to some independent blocks simultaneously. In the following discussion, we say two blocks are independent of each other if they share neither any common column nor any common row of the rating matrix.

Based on the main idea of the partitioning rating matrix, scholars have proposed the distributed SGD algorithm [24], fast parallel SGD [25], fast distributed SGD [26], etc.

The existing SGD approach for GPUs is also based on the idea of the partitioning rating matrix, but it uses every thread to process different elements, and updates elements in a vector serially. In the SGD for MF, elements inside a vector have no dependency, so elements in a vector are also updated simultaneously. In this paper, we explore the combination of partitioning rating matrix and the inherent parallelism of vector computing.

3. **GPU Computing.**

A GPU is built around an array of streaming multiprocessors (SMs) [27]. A multi-threaded program is partitioned into blocks of threads that execute independently from each other, so that a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

Figure 1 shows the GPU computing model. Threads are grouped into blocks, and blocks are grouped into a grid. Each thread has a unique local index in its block, and each block has a unique index in the grid. Kernels can use these indices to compute array subscripts, for instance.

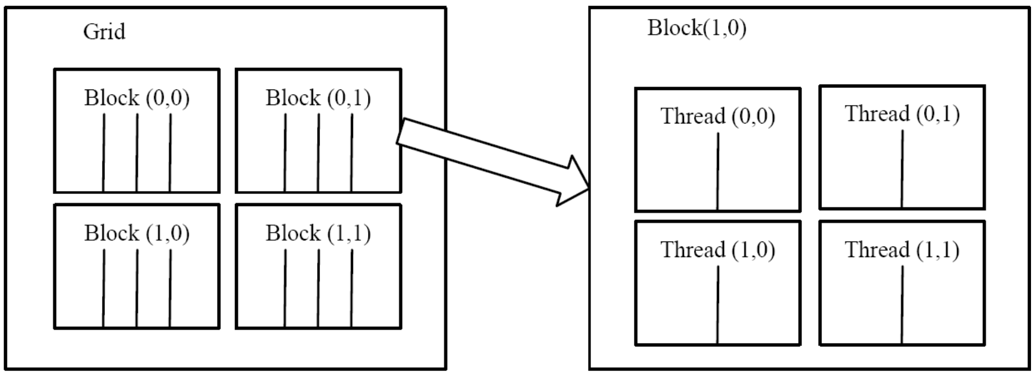


FIGURE 1. GPU computing model

Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block; a warp will always be a subset of threads from a single block. Threads in different blocks may be assigned to different multiprocessors concurrently, to the same multiprocessor concurrently (using multi-threading), or to the same or different multiprocessors at different times, depending on how the blocks are scheduled dynamically.

These abstractions provide fine-grained data parallelism and thread parallelism, nested within coarse-grained data parallelism and task parallelism. They guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel by blocks of threads, and each sub-problem into finer pieces that can be solved cooperatively in parallel by all threads within the block.

Furthermore, threads in a block are partitioned into wraps. NVIDIA GPUs have used similar wrap configurations in which each wrap consists of 32 threads. The execution of wraps is implemented by single instruction multiple data (SIMD) hardware. SIMD hardware executes all threads of a wrap as a bundle. An instruction is run for all threads in the same wrap. This works well when all threads within a wrap follow the same execution path. When threads within a wrap take different control flow paths, e.g., an if-else construct, the SIMD hardware will take multiple passes through these divergent paths. These passes are sequential to each other; thus, they will add to the execution time. When threads in the same wrap follow different paths of control flow, we say that these threads diverge in their execution.

A GPU supports several types of memory that can be used by programmers. Figure 2 shows these GPU memory types. Global memory is allocated to all threads and all threads can access it. Registers are allocated to individual threads and each thread can only access its own registers. Shared memory is allocated to thread blocks and all threads in a block can access variables in the shared memory. Registers, shared memory, and global memory have different latencies. Registers comprise the most efficient storage, while global memory is least efficient. When designing parallel programs, selecting the appropriate storage is required.

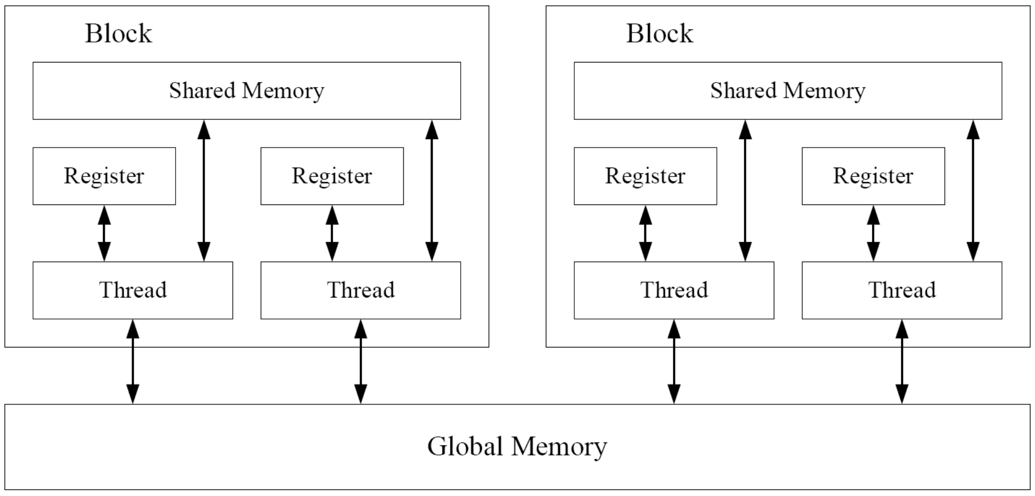


FIGURE 2. GPU memory model

Previous work developed SGD on a GPU, which is called GPUSGD [20]. To utilize the computing power of a GPU, the rating matrix is divided into $l \times l$ blocks, where l is the number of thread blocks. Within a block of the rating matrix, the rating element is also divided into several groups, and the elements in the same group share neither the same row nor column. After this data processing, every thread of the GPU will update a user vector and an item vector with no update loss and also avoid thread divergence. In the process of updating, all elements in the user vector and the item vector will be updated serially. In this work, every thread is used to update a user vector and item vector associated with a rating in the rating matrix. The core of this work is to ensure that a large number of threads execute in parallel and do not interfere with each other.

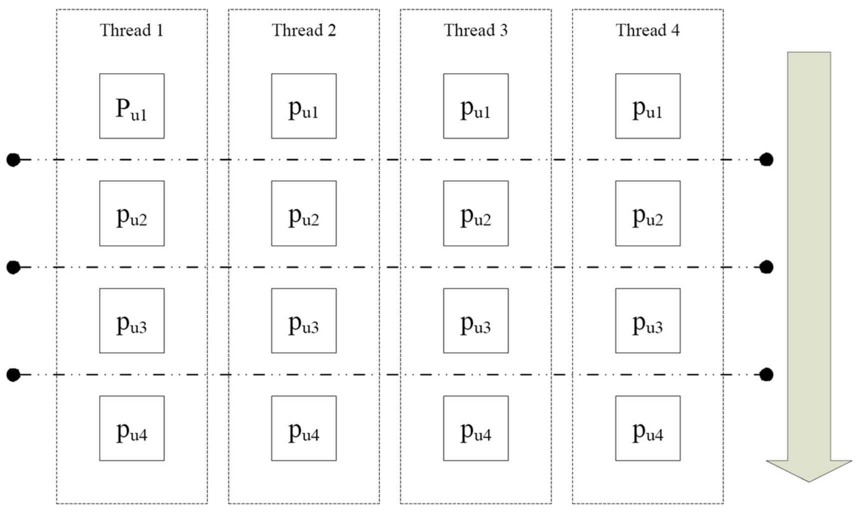
GPU parallelism is well suited for vector or matrix calculations. Matrix calculations can be done very quickly on a GPU and many matrix computing libraries have already been developed. SGD for MF is full of matrix computing. However, the past work did not study the parallel optimization of SGD on a GPU from the vector perspective.

In this paper, we develop a new method that can implement the SGD on a GPU from the vector perspective.

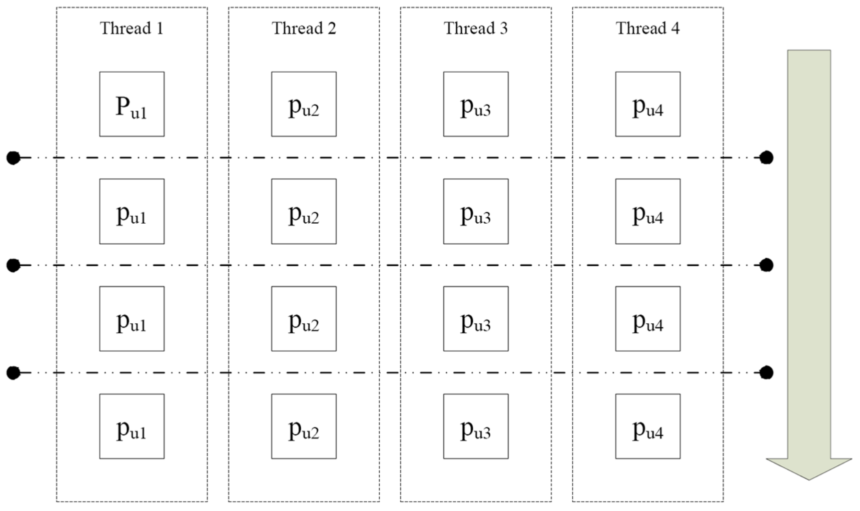
4. **NewGPUSGD Method.** From the above description of a GPU, we can conclude that it is suitable for computing with many different data executing the same instruction. Matrix computing is suitable for a GPU. Other optimization methods, when solving, generally use the GPU to calculate all the elements in the vector at the same time. The SGD is also used to update the vector. Thus, we design SGD for MF-based CF for GPU computing along this idea. This is called NewGPUSGD. The thread workflows of GPUSGD and NewGPUSGD are shown in Figure 3. GPUSGD updates every element in a vector serially in a thread, while NewGPUSGD updates every element in a vector in parallel.

However, there is a significant difference between SGD for MF and for other optimization methods. The solution vector of other optimization methods is always of thousands, so it can make full use of a GPU if every thread computes an element.

SGD for MF computes many vectors as the goal of optimization. The vectors solved by SGD for MF are only tens or hundreds, far less than the number of threads that the



(a) The workflow of GPUSGD



(b) The workflow of NewGPUSGD

FIGURE 3. Thread workflows of GPUSGD and NewGPUSGD; dotted lines across all threads represent data synchronization

GPU can run. Therefore, we must calculate multiple solution vectors at the same time. This is needed to avoid the problem of losing updates, and data partitioning is used.

If we want to use several threads to update all the elements in a vector, another problem arises, namely the result of $p_u^T q_v$ is necessary. Therefore, we need to obtain $p_u^T q_v$ before updating the vectors. The difficulty of designing a kernel function of an updating vector is to calculate $p_u^T q_v$ through multi-threads.

We use different thread blocks to update different vectors, and threads within a thread block update different elements in a vector.

Next, we introduce how to partition the rating matrix and how to design the kernel function for a GPU.

4.1. **Data partitioning and sorting.**

The dimension of a user vector and item vector is small. To make full use of GPU computing power, we must update more than one vector at the same time. According to the analysis in Section 2, the vectors updated at the same time should share neither the same user nor the same item.

The common idea is to partition R into several blocks as in FPSGD and GPUSGD. If we plan to assign l thread blocks, then the matrix needs to be divided into $l \times l$ blocks. We can label every block with a tag and require that the block shares neither the same row index nor column index having the same tag. We label the block in a diagonal direction as shown in Figure 4. Up to l tags can identify all the blocks.

Blocks with the same tag can be processed at the same time. In each round of iterations, all blocks are processed in tag order, so that all elements are calculated.

If the blocks that identify different tags are processed at the same time, loss of updates will also occur. For example, as shown in Figure 4, if a thread block has already processed the block $(1, 1)$ and starts processing $(1, 2)$ immediately and another thread block is still processing the block $(2, 2)$, the situation in which two elements share the same row or column in R may occur. Thus, each time blocks identified by a tag are processed, the calculation needs to be synchronized once. The matrix in Figure 4 is being processed in a GPU as shown in Figure 5.

(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 4. Data partitions and labelling with tags

In GPUSGD, R is partitioned into $l \times l$ blocks of size $z \times z$. The rating matrix is not a square matrix, so it sets $z = \frac{m}{l}$ if $m \geq n$ and $z = \frac{n}{l}$ if $m < n$. This fills the original matrix to a square matrix. This is because GPUSGD also needs to select elements that can be processed simultaneously in a block, and a square block is convenient. However, our method will not process the elements in the same block simultaneously, so, in the proposed NewGPUSGD, R is partitioned into $l \times l$ blocks of size $t \times s$, where $t = \frac{m}{l}$ and $s = \frac{n}{l}$.

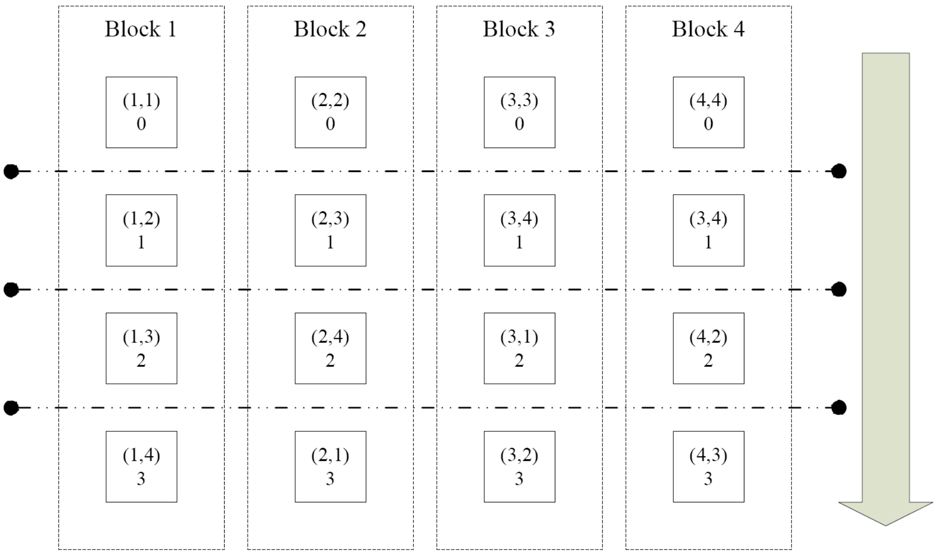


FIGURE 5. Block workflow of NewGPUSGD; dotted lines across all thread blocks represent data synchronization

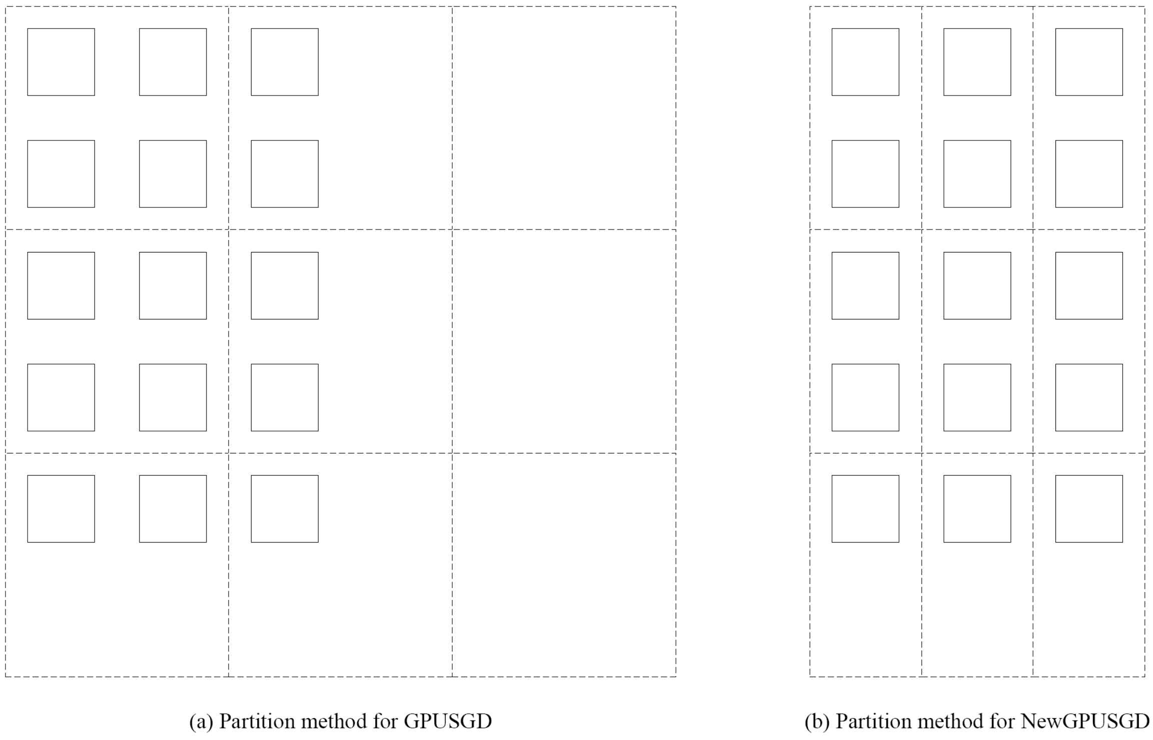

(a) Partition method for GPUSGD
(b) Partition method for NewGPUSGD

FIGURE 6. Data partition methods of GPUSGD and NewGPUSGD

The difference between the two partitioning methods is shown in Figure 6. In actual data, the number of users is usually much greater than the number of items, or the number of items is much greater than the number of users. Therefore, the original method of division into square matrices produces too many blank blocks. It can be seen that our partitioning method is more efficient.

4.2. **Kernel function.** Here, we design the kernel function to update the vector. We want every thread to update an element. Therefore, when calling a kernel function, the number of threads in each thread block needs to be equal to the latent dimension.

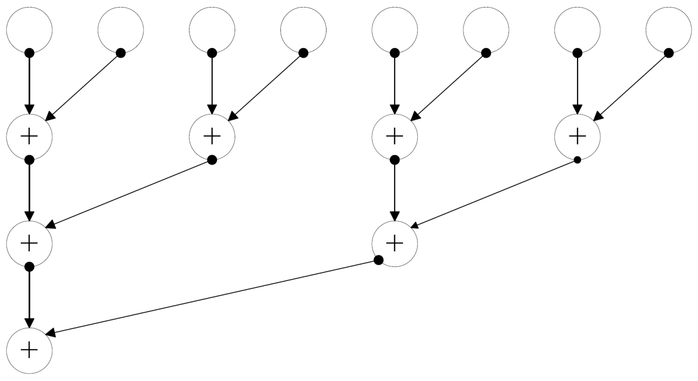


Figure 7. Schematic of summation by multi-thread

Before updating the vector, we should calculate $r_{uv} - p_u^T q_v$ first. Every thread will use this result. If all the threads calculate the result once, it is a waste of time and storage.

When calculating $p_u^T q_v$, the first step is to calculate every multiplication of the corresponding elements. This step has no result dependencies and can therefore be directly assigned to each thread. The next step is to calculate the sum of f elements by multi-threads. As shown in Figure 7, after l threads calculate l multiplications, two items are summed with half the number of threads in each round, and this process is performed recursively until the final result is obtained.

In the simple GPU program, every thread will only access the data of itself. However, in this step, every thread must access the result obtained by other threads. Thus, it cannot be stored in the register that can be only accessed by a thread. To meet this requirement, we store the results obtained by f multiplications in shared memory, whose data belong to the thread block and which is more efficient than the global memory, so that each thread in the thread block can access it efficiently. The GPU kernel can be seen in Algorithm 1.

Algorithm 1 NewGPUSGD (GPU Kernel)

```
1:  Allocating an array $preScore$ in the shared memory
2:  $tid \leftarrow$ thread ID
3:  for every block assigned to this thread block do
4:      for every element in this block do
5:          Determine the corresponding user and vector
6:          $preScore[tid] \leftarrow userVector[idx] \times itemVector[idx]$
7:          $sign \leftarrow 1$
8:          while $sign \times 2 < f$ do
9:              if $tid\%2 = 0$ then
10:                 $preScore[tid] \leftarrow preScore[tid] + preScore[tid + sign]$
11:             Synchronize
12:             $sign \leftarrow sign \times 2$
13:         $userVector[idx] \leftarrow userVector[idx] + \gamma \times \{(rating - preScore[0])$
            $\times itemVector[idx] - \lambda \times userVector[idx]\}$
14:         $itemVector[idx] \leftarrow itemVector[idx] + \gamma \times \{(rating - preScore[0])$
            $\times itemVector[idx] - \lambda \times itemVector[idx]\}$
15:         Synchronize
16:     Synchronize
```

4.3. **Analysis of NewGPUSGD.** Compared to the GPUSGD method, the proposed method has the following advantages.

(1) GPUSGD must select data in the same block that can be processed simultaneously and update every element of a vector serially. The proposed NewGPUSGD uses all threads in a thread block to process a vector, and it is very easy to avoid the divergence of threads. GPUSGD must rearrange data to achieve this goal. NewGPUSGD avoids many operations in data pre-processing.

(2) The new blocking strategy of NewGPUSGD reduces many blank blocks. This can take full advantage of the GPU's computing resources and avoid waste of resources.

(3) Compared to the existing SGD approach for GPUs, the new kernel function will make less idle threads. For example, in a block shown in Figure 8, if the dimension of the hidden feature is 3, NewGPUSGD needs to allocate three threads in the thread block, and it takes eight units of time to complete the computation of the block. In the case of GPUSGD, also in the case of allocating three threads, four phases are required to process this block. The first phase processes elements 1, 6, 11, and 16. Since there are only three threads, elements 1, 6, and 11 must be processed in parallel, and then one thread is used to process element 16, at the same time the other two threads are in a wait state. This computation of the first phase is completed in six units of time. It takes a total of 15 units of time for GPUSGD to process this block. As can be seen from the analysis of this example, NewGPUSGD is more efficient than GPUSGD.

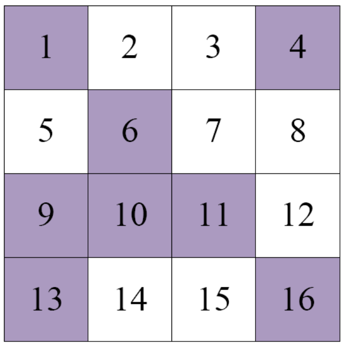


FIGURE 8. Example block. Elements with the shadow represent that the value is known; elements without shadow represent that the value is missing.

5. **Experiment.** The performance of the proposed NewGPUSGD method is evaluated using four datasets: MovieLens10M, M100k, Netflix, and BDMovie. These datasets are all publicly available on the Internet.

GPU results were achieved using 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.

The main target of this paper is to accelerate the computation of the GPU, so we chose a suitable set of parameters. We fixed 100 as the maximum number of iterations to observe the iteration time. γ was set to 0.005 and λ to 0.03. Step is not particularly large, so it will make the iteration swing. The regularization parameters are small and do not have an excessive effect on the result. Setting of the parameter affects the convergence of the iteration but does not affect the acceleration. The observed experimental results are from two precisions, single and double.

5.1. **Proposed SGD method.** The proposed method mainly has two variables: latent dimension and number of blocks. In the GPU implementation, the number of blocks is equal to the number of thread blocks, and the latent dimension is equal to the number of threads in every thread block.

First, we observe the convergence of the iterations when the number of blocks is different. The trend of the iteration is shown according to the root-mean-square error (RMSE):

$$\mathrm{RMSE} = \sqrt{\frac{1}{\|V\|}\sum_{(u,v)\in\Omega}\left(\hat{r_{uv}} - r_{uv}\right)^2} \tag{4}$$

where V is the validation set and $\|V\|$ the cardinality of set V. RMSE can be used to evaluate the precision of a recommendation. We block the rating matrix into 64×64 and 1024×1024 matrices and fix the latent dimension at 16. The result is shown in Figure 9.

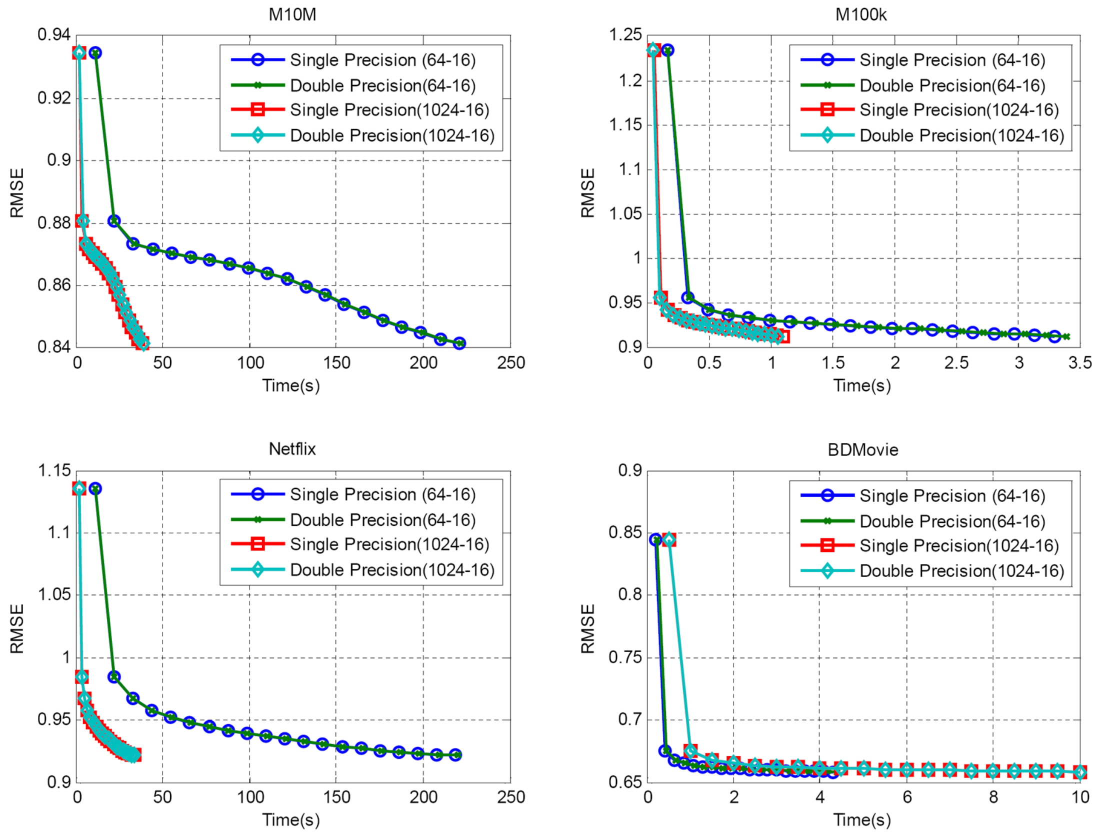


Figure 9. RMSE trend

It can be seen that, although the number of blocks is different, each round of iterations calculates all the samples for one round, and while the order will bring minor changes, the overall trend is similar. Therefore, we can see that NewGPUSGD is stable and will not change the final iteration result with the change of matrix partition.

We analyzed the effect of these two variables through experiments as follows. We fixed the latent dimensions at 16 and 32. The effect of the number of different blocks is shown in Figure 10. It can be seen from the figure that the more blocks are, the greater the efficiency is.

When the number of blocks is small, there will be fewer thread blocks allocated on the GPU. Therefore, many GPU computing resource resources are not utilized. As the number of blocks increases, the computing resources of the GPU are more fully utilized and the computational efficiency is increased. At the same time, if the block number increases, the distribution of the elements will be more balanced, so thread blocks in the waiting state will be reduced.

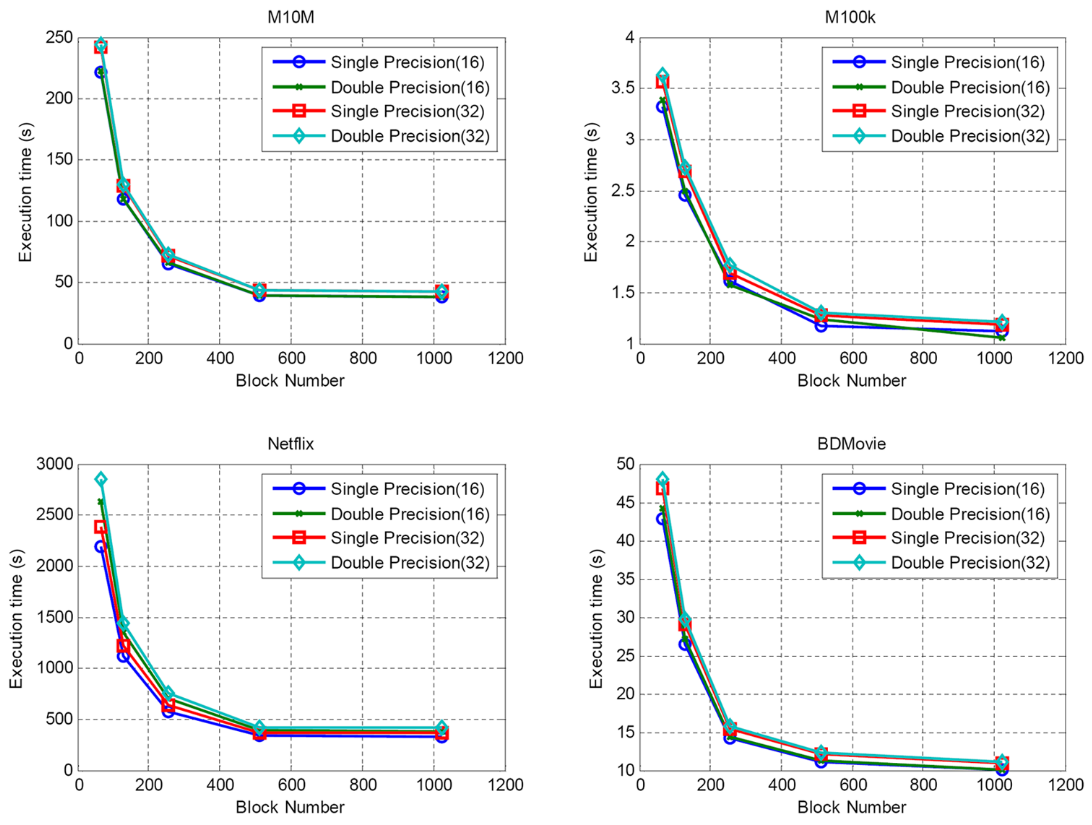


FIGURE 10. Time consumption of different block numbers

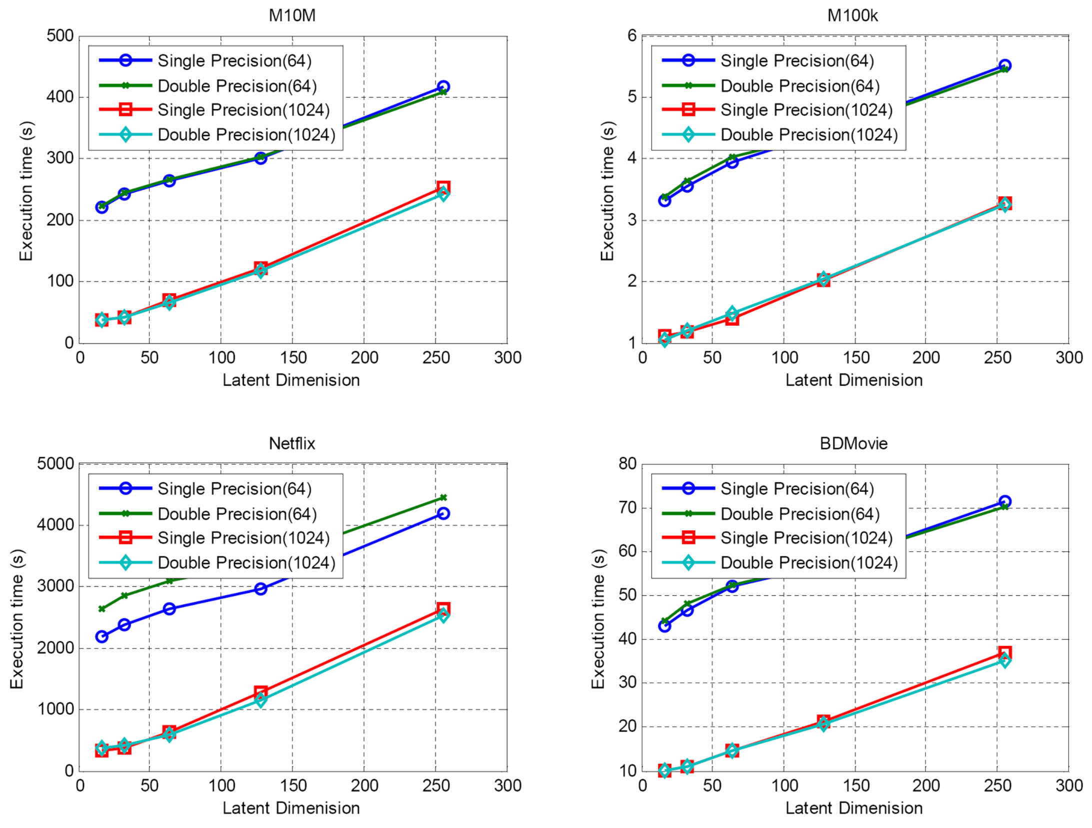


FIGURE 11. Time consumption of latent dimension

We then fixed the number of blocks at 64 and 1024, and observed the effect of changing latent dimension. The result is shown in Figure 11. When the dimension of hidden features increases, the number of threads needed increases, as does the duration of the calculation.

As the dimension of the latent dimension increases, the amount of computation increases in an approximately linear manner. Owing to the fact that the computing resources of the GPU are fixed, the calculation time also increases in an approximately linear manner.

5.2. **Comparison with GPUSGD.** The paper introducing the previous GPUSGD was the first on implementing the SGD algorithm on a GPU. Therefore, we compared our experimental results with the results presented in that paper.

We blocked the rating matrix into a 64 × 64 matrix and fixed the latent dimension at 16. We then blocked the rating matrix into a 1024 × 1024 matrix and set the latent dimensions at 16 and 256. When using GPUSGD, the number of threads per thread block does not need to equal the latent dimension. To be fair, we used the same number of thread blocks and threads to calculate the same problem. We set the number of threads per thread block to equal the latent dimension. The result is shown in Figure 12.

When the number of blocks is 64 × 64, the two methods allocate 16 thread blocks on the GPU, so neither method can fully utilize the computing resources of the GPU. At this time, both methods have their own outcomes. On Netflix, GPUSGD is faster. On the other three datasets, NewGPUSGD is faster.

When the number of blocks becomes 1024 × 1024, both methods can make full use of the computing resources of the GPU. At this time, due to the fact that NewGPUSGD does not generate blank blocks and the task allocation of threads is more balanced, the advantages of NewGPUSGD compared to GPUSGD are apparent.

When the number of blocks is 1024 × 1024, the advantage of NewGPUSGD is further highlighted by increasing the dimension of the latent feature to 256. The computation

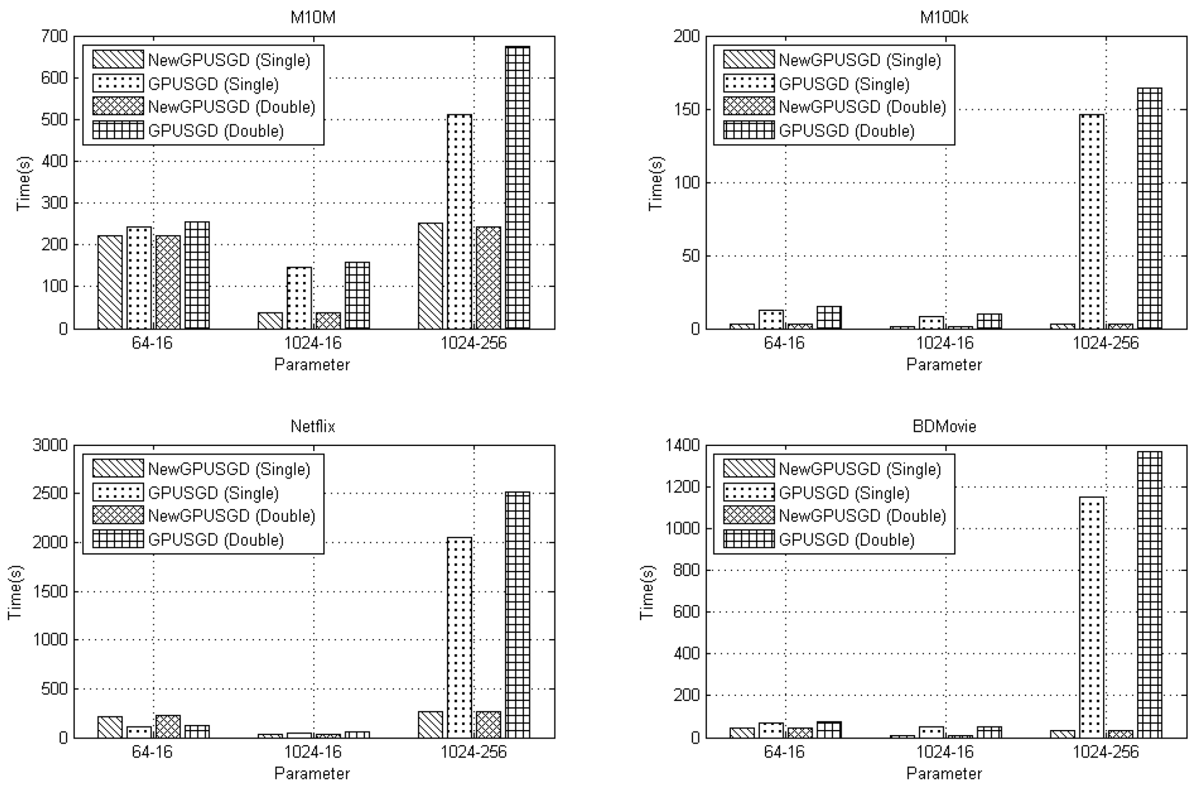


FIGURE 12. Time consumption of GPUSGD and NewGPUSGD

time of both NewGPUSGD and GPUSGD grows in an approximately linear fashion. Thus, NewGPUSGD has an advantage after the dimension of latent features increases.

In general, NewGPUSGD has obvious advantages under the same allocation of computing resources.

6. Conclusion.

MF-based CF has been widely used in many recommender systems. SGD is one of the most popular algorithms for solving MF problems with missing values. However, the large computational burden required by SGD poses a challenge of accelerating the SGD process. In the past several years, the GPU has evolved into a very flexible and powerful many-core processor. We implemented SGD on GPU and solved two problems, the update-loss problem and the thread task design problem. The results on various types of data show that the proposed algorithm can be well suited for the massively parallel GPU architecture.

Acknowledgment. This research was supported in part by NSFC under Grant No. 6157 2158 and 61602132, Shenzhen Science and Technology Program under Grant No. JCYJ2 0160330163900579 and JCYJ20170811160212033.

REFERENCES

[1] P. Resnick and H. R. Varian, Recommender systems, *Communications of the ACM*, vol.40, no.3, pp.56-58, 1997.

[2] J. B. Schafer, J. Konstan and J. Riedl, Recommender systems in e-commerce, *ACM Conference on Electronic Commerce*, pp.158-166, 1999.

[3] R. Burke, Hybrid recommender systems: Survey and experiments, *User Modeling and User-Adapted Interaction*, vol.12, no.4, pp.331-370, 2002.

[4] F. Cacheda, V. Carneiro, D. Fernández et al., Comparison of collaborative filtering algorithms: Limitations of current techniques and proposals for scalable, high-performance recommender systems, *ACM Trans. the Web*, vol.5, no.1, pp.1-33, 2011.

[5] J. L. Herlocker, J. A. Konstan, L. G. Terveen et al., Evaluating collaborative filtering recommender systems, *ACM Trans. Information Systems*, vol.22, no.1, pp.5-53, 2004.

[6] V. P. Pauca, J. Piper and R. J. Plemmons, Nonnegative matrix factorization for spectral data analysis, *Linear Algebra and Its Applications*, vol.416, no.1, pp.29-47, 2006.

[7] F. Shahnaz, M. W. Berry, V. P. Pauca et al., Document clustering using nonnegative matrix factorization, *Information Processing and Management*, vol.42, no.2, pp.373-386, 2006.

[8] S. A. Vavasis, On the complexity of nonnegative matrix factorization, *SIAM Journal on Optimization*, vol.20, no.3, pp.1364-1377, 2009.

[9] Z. Huang, Y. Ye, X. Li et al., Joint weighted nonnegative matrix factorization for mining attributed graphs, *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp.368-380, 2017.

[10] Y. Koren, R. Bell and C. Volinsky, Matrix factorization techniques for recommender systems, *Computer*, vol.42, no.8, pp.42-49, 2009.

[11] D. Tikk, Fast ALS-based matrix factorization for explicit and implicit feedback datasets, *ACM Conference on Recommender Systems*, pp.71-78, 2010.

[12] Y. Zhou, D. Wilkinson, R. Schreiber et al., Large-scale parallel collaborative filtering for the Netflix prize, *International Conference on Algorithmic Aspects in Information and Management*, pp.337-348, 2008.

[13] T. Zhang, Solving large scale linear prediction problems using stochastic gradient descent algorithms, *International Conference on Machine Learning*, pp.919-926, 2004.

[14] M. Hardt, B. Recht and Y. Singer, Train faster, generalize better: Stability of stochastic gradient descent, *arXiv preprint arXiv:1509.01240*, 2015.

[15] S. Zhang, C. Zhang, Z. You et al., Asynchronous stochastic gradient descent for DNN training, *IEEE International Conference on Acoustics, Speech and Signal Processing*, pp.6660-6663, 2013.

[16] D. Steinkraus, I. Buck and P. Y. Simard, Using GPUs for machine learning algorithms, *Proc. of the 8th International Conference on Document Analysis and Recognition*, Seoul, Korea, pp.1115-1120, 2005.

[17] I. Nisa, A. Sukumaran-Rajam, R. Kunchum et al., Parallel CCD++ on GPU for matrix factorization, *Proc. of the General Purpose GPUs*, pp.73-83, 2017.

[18] W. Tan, L. Cao and L. Fong, Faster and cheaper: Parallelizing large-scale matrix factorization on GPUs, *Proc. of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp.219-230, 2016.

[19] H. Yu, C. Hsieh, S. Si et al., Parallel matrix factorization for recommender systems, *Knowledge and Information Systems*, vol.41, no.3, pp.793-819, 2014.

[20] J. Jin, S. Lai, S. Hu et al., GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization, *Concurrency and Computation: Practice and Experience*, vol.28, no.14, pp.3844-3865, 2016.

[21] X. Xie, W. Tan, L. L. Fong et al., CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUs, *Proc. of the 26th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp.79-92, 2017.

[22] F. Li, Y. Ye, Z. Tian and X. Zhang, CPU versus GPU: Which can perform matrix computation faster – Performance comparison for basic linear algebra subprograms, *Neural Computing and Applications*, DOI: 10.1007/s00521-018-3354-z, 2018.

[23] B. Recht, C. Re, S. Wright et al., Hogwild: A lock-free approach to parallelizing stochastic gradient descent, *Advances in Neural Information Processing Systems*, pp.693-701, 2011.

[24] R. Gemulla, E. Nijkamp, P. J. Haas et al., Large-scale matrix factorization with distributed stochastic gradient descent, *Proc. of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Diego, CA, USA, pp.69-77, 2011.

[25] Y. Zhuang, W. S. Chin, Y. C. Juan et al., A fast parallel SGD for matrix factorization in shared memory systems, *Proc. of the 7th ACM Conference on Recommender Systems*, Hong Kong, pp.249-256, 2013.

[26] F. Li, B. Wu, L. Xu et al., A fast distributed stochastic gradient descent algorithm for matrix factorization, *Proc. of the 3rd International Conference on Big Data, Streams and Heterogeneous Source Mining: Algorithms, Systems, Programming Models and Applications*, pp.77-87, 2014.

[27] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi et al., Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp.73-82, 2008.