Novel GPU-Based Approach for Matrix Factorization using Stochastic Gradient Descent

Mohamed A. Nassar, Layla A. A. El-Sayed, Yousry Taha
Department of Computer and Systems Engineering,
Alexandria University, Alexandria, Egypt

Abstract

Recommender systems are crucial tools used in most of daily-based web applications. Matrix factorization is an advanced and efficient technique for recommender systems. Recently, Stochastic Gradient Descent (SGD) method is considered to be one of the most popular techniques for matrix factorization. SGD is a sequential algorithm, which is difficult to be parallelized for large-scale problems. Nowadays, researches focus on efficiently parallelizing SGD. In this research, we propose a novel GPU approach for parallelizing SGD. The proposed approach is more efficient than recent parallel approaches because it utilizes GPU, reducing non-coalesced access of global memory and achieving load balance of threads. In addition, it does not require any sorting and/or data shuffling as preprocessing phase. Although platform used for implementation of the proposed approach is old, the implementation demonstrates 12.5x speedup over state-of-the-art GPU method, BSGD.

1. Introduction

Recommender systems are crucial tools used nowadays in most of applications including Facebook, Twitter, Email services, News services and Hotel reservation applications [1]. Recommender systems provide users with suggested items (such as movies, friends, news, advertisements, products, etc.). Non-personalized filtering, Content-based filtering (CBF), Collaborative filtering (CF) and Matrix factorization are main categories of techniques used in recommender systems [2, 3, 4].

In non-personalized recommender systems, recommendations are based on the average of ratings given to that item by other users. Although non-personalized recommender systems are simple in implementation, it lacks personalization because each user gets the same recommendation [1]. In CBF, items are recommended similar to those which a user preferred previously. In order to find similarity between users' preferences and items, CBF systems subjectively represent items and users' behaviors as features. The representation of items and users' behaviors are major problems in CBF. On the other hand, CF is a process of recommending items based on the collaboration of users, or the similarity between items [2]. CF overcomes the personalization issue with non-personalized recommender systems. In addition, there are no features required to represent items or users' behaviors as in CBF. CF fails to recommend items if there is no common similarities between users or items. It also becomes time-consuming when data size is large [5, 6].

Matrix factorization technique is a new advanced technique for CF. It is also known as dimensionality reduction technique. Matrix factorization technique compresses representation of user and item by extracting significant latent features [3]. Such technique is considered as one of the most effective techniques for recommender systems [7]. It has many advantages such as: it provides a compact meaningful relation between users and items; recommendation process is simple; the technique also incorporates implicit feedback and temporal effects [3].

Although there are many advantages for such technique over other CF techniques, building/rebuilding the reduced model of users' ratings is complex in terms of computations. Therefore, many researches are directed to design fast and scalable method for CF technique [7, 8, 9, 10].

Matrix factorization is a technique which finds two feature matrices $P \in \mathbb{R}^{kxm}$ and $Q \in \mathbb{R}^{kxn}$ such that $r_{u,v} \simeq p_u^T q_v$, where k is the number of latent features, m and n are numbers of users and items in the system respectively, and $p_u \in \mathbb{R}^k$ and $q_v \in \mathbb{R}^k$ are respectively the u^{th} column of P and the v^{th} column of Q.

The optimization problem is

$$\min_{P,Q} \sum_{(u,v)\in R}((r_{u,v} - p_u^T q_v)^2 + \lambda_P \|p_u\|^2 + \lambda_Q \|q_u\|^2), \quad (1)$$

where $\|.\|$ is the Euclidean norm, $(u,v) \in R$ are the indices for users' ratings, λ_P and λ_Q are the regularization parameters for avoiding of over-fitting. (1) is a difficult optimization problem [7, 9, 10]. Many researches proposed optimization methods to solve (1).

Recently, Stochastic Gradient Descent (SGD) is commonly used for matrix factorization [7, 8, 9, 10]. The time complexity per iteration of SGD is O(|Ω|k), where |Ω| is number of ratings. However, SGD is difficult to parallelize. Parallelized versions for shared memory and distributed systems are found in [7, 11, 12].

Nowadays, all systems are heterogeneous including smart devices. These systems achieve a high performance and energy efficiency by utilizing different kind of processors [13, 14]. FPGAs and GPUs are the most common used multiprocessing devices in nowadays systems. Generally, GPU outperforms FPGA for applications, which have floating-point-based operations like SGD, as GPU has native floating-point processors.

GPU is a many-core processor, which has powerful computational capabilities. Although GPU is basically for utilized for graphics processing, GPU has been utilized for parallel intensive numerical computations. A typical GPU consists of hundreds of cores. Compared to the CPU, GPU has better floating-point capability. A GPU is a set of SIMD stream multiprocessors (SMs) with a set of stream processors (SPs) each. Each SM has an efficient shared memory among its SPs. In addition to the shared memory, each SM has a L1 cache managed by hardware. Local registers are available for each SP. GPU Global memory is used for data communication between SMs. A CPU can read/write from/to GPU global memory. At the software level, CUDA [15] is a parallel computing platform which enables software developers to program GPU devices for general-purpose processing (GPGPU). CUDA model is a group of threads running in parallel. In CUDA, we refer to CPU and its memory by term 'host' and GPU and its memory by term 'device'. Host launches kernels -a kernel is a piece of work to be run by GPU threads- to be executed on device. A programmer organizes device as a grid of thread blocks, which consist of a matrix of threads. For a given thread, data to be accessed from memory can be determined by the index of the thread. In order to enhance performance, threads in a single block communicate through the shared memory.

In this paper, the main objective is to propose an innovative parallel approach of SGD based on GPU by efficiently parallelize SGD. Unlike other approaches, this approach aims to overcoming time required by preprocessing phase i.e. no need for any sorting and/or random shuffling of data, and overcoming the complexity of the scheduler to ensure the load balancing across computational resources. Furthermore, our approach eliminates the overhead associated with locking and data locality [7].

The remainder of the paper is organized as follows. In section II, we discuss existing parallelized algorithms. In Section III, we present our efficient parallel proposal. Experimental results are presented in Section IV. Finally, Conclusions and future work are discussed in Section V.

2. Related Work

The basic idea of SGD is to randomly select a $r_{u,v}$ from rating matrix R^{mxn} where u,v are the indices, and m,n are the number of users and items respectively. Then, p_u and q_v variables are updated by the following rules.

$$p_u \leftarrow p_u + \gamma(e_{u,v}q_v - \lambda_P p_u), \quad (2)$$

$$q_v \leftarrow q_v + \gamma(e_{u,v}p_u - \lambda_Q q_v), \quad (3)$$

where $$e_{u,v} = r_{u,v} - p_u^T q_v \quad (4)$$

is the error between the actual and predicted ratings for the $r_{u,v}$. λ_P, λ_Q are the regularization coefficients for avoiding overwriting and γ is the learning rate. Then another random instance $r_{u,v}$ is selected, and p_u and q_v are updated by applying rules (2) and (3) respectively. After finishing all ratings, the previous steps are repeated till reaching accepted Root Mean Square Error (RMSE) [7]. The overall SGD procedure takes hours. Therefore, an efficient parallel SGD is very useful.

Although there is a non-SGD methed like CCD++ which is a prallel coordinate descent method, we present the state-of-the-art methods which are based on SGD methods because generally CCD++ running time and RMSE are still worse than state-of-the-art parallel SGD [7]. Thus parallel SGD remains a powerfull method for matrix factorization [7].

2.1. HogWild [12]

HogWild notices that for sampled ratings selected randomly, the updates (2 - 3) are totally independent provided that rating matrix R is highly sparse. The reason is that the ratings to be processed do not share the same users' identities and items identities. Therefore, iterations of SGD can be run in parallel using different threads. HogWild uses atomic operations when overwriting occurs. [7] proves the convergence under the assumption that rating matrix is very sparse.

2.2. DSGD [16]

DSGD divides a rating matrix R into ratings blocks which are mutually independent and their corresponding variables can be updated in parallel. Independent ratings blocks are running simultaneously.

2.3. FSGD [7]

[16] points out that parallel SGD methods suffer from some issues when they are applied in a shared-memory

environment which cause performance degradation. These issues are as follows:

- Locking problem: The locking problem occurs if a thread is idle because of waiting for other threads.
- Memory discontinuity: HogWild and DSGD randomly pick ratings from R (or from a block of R). The random method generally guarantees good convergence, however it suffers from the memory discontinuity.

In order to overcome previous issues, [16] introduces the following implementation techniques:

- Lock-free scheduling: Once a thread finished processing a block, the scheduler assigns a new block which satisfies the following two criteria: (1) it is a free block. (2) Its number of past updates is the smallest among all free blocks.
- Partial Random Method: In order to overcome the issue of memory discontinuity, FSGD simply accesses rating within blocks sequentially, but block selection is in a random manner.

Unfortunately, the scheduler is unable to keep the number of updates for all blocks balanced in a case where most available ratings are in certain blocks. [7] overcomes the imbalance issue by random shuffling of R, gridding R into blocks and sorting each block by user identities.

2.4. BSGD [11]

[11] introduces batch SGD (BSGD) for GPU, where the gradient of a batch of ratings are taken at the same time. [11] proves that for a selected small batch size, BSGD works like SGD. BSGD mainly grids the GPU into s thread blocks where s is the batch size. Each thread block has k threads where k is number of latent features. Before starting the algorithm, shuffling the dataset is always done. Then, batches are sent to the GPU in streaming manner. Each rating from a batch is assigned to a corresponding thread block.

2.5. GPUSGD [20]

GPUSGD divides a rating matrix R into ratings blocks which are mutually independent and their corresponding variables can be updated in parallel. Independent ratings blocks are running simultaneously using thread blocks of GPU. Authors prove that all independent ratings (i.e. ratings are not sharing the same rows and columns) inside each block can be tagged with same tag number. Therefore a preprocessing phase, tagging and sorting ratings inside each block are performed to provide coalesced access and independent updates by GPU block threads. The experimental results show that GPUSGD performs much better in accelerating the matrix factorization compared with the existing state-of-the-art parallel methods.

3. Proposed Approach

Before discussing the new proposed approach, we highlight the issues with the parallel methods discussed in previous section. Although FSGD overcomes two main issues of previous implementations which are locking problem and memory discontinuity, FSGD suffers from the following issues:

- Ineffective complex scheduler: In order to overcome load imbalance in DSGD, a complex logic scheduler has been proposed. However the scheduler is unable to keep the number of updates for all blocks balanced in a case where most available ratings are in certain blocks. Many systems including Netflix and MovieLens suffer from such issue [17]. Therefore, complex scheduler is considered impractical without applying random shuffling, gridding of all ratings into blocks and sorting each block by user identities before running DSGD [7].
- Complexity of random shuffling of all ratings and sorting blocks: Although random shuffling, gridding R into blocks and sorting each block by user identities is excellent solution for overcoming imbalanced updates for blocks, time complexity required for random shuffling are O(m+n) as random shuffling is performed for row indexes and column indexes separately. In addition, time complexity for sorting users/items identities in each block is O(mlogm)/O(nlogn) respectively as each block has space of O(mn). [7] excludes these times from calculations of time required for FSGD. These times cannot be excluded from overall processing time, as rating matrices for dynamic systems have to be updated frequently. Therefore, random shuffling and sorting blocks are necessary each time running FSGD.

For BSGD, the following issues have to be considered:

- Complexity of random shuffling of rating matrix: BSGD requires random shuffling of rating matrix at the beginning in order to split data into batches. As mentioned in previous point, time complexity for random shuffling cannot be excluded.
- Non-utilized GPU: BSGD assigns each thread block with only one rating from R.
- Non-scalability: BSGD implementation was in a limited size rating matrix. In case of increasing the number of ratings, the processing time increases proportionally.

For GPUSGD, the following issues have to be considered:

- Complexity of preprocessing phase: GPUSGD requires partitioning of rating matrix, tagging blocks entries, and sorting each block by tags.

- Non-utilized GPU resources: GPUSGD assumes that rating matrix is not sparse which is not the case in recommender systems.

In order to overcome previous issues, we propose Efficient SGD (ESGD) for GPU. First, we represent the ratings as one-dimensional array R of length L. Each entry of R has the following format (u_id, i_id, r) where u_id is user identity, i_id is item identity and r is rating provided by user_id to i_id. In ESGD, ratings and feature matrices (P and Q) are stored in global memory of GPU device. We grid the GPU into two-dimension thread blocks and each thread block has one-dimension threads of size th_size such that th_size is multiple of k (number of features). Threads inside each thread block load shared memory with sh_size ratings in st steps, where sh_size = th_size x st. The loading process is performed in coalesced manner to overcome the issue of random access of global memory (memory discontinuity)[7]. Figure 1 shows an example of loading R to shared memory of two thread blocks in two steps where L = 8, th_size = 2 and sh_size = 4.

Threads access ratings randomly from shared memory using pre-calculated random array (Rand). Rand is an array containing a pre-calculated random numbers from 0 to sh_size – 1. It is worth to mention that the process of generating Rand is offline and one-time. Figure 2 shows an example of accessing ratings randomly from shared memory of a thread block in two steps when sh_size = 4 and th_size = 2. Shaded entries of shared memory show the accessed entries at a certain moment.

In order to assure a high probability of independent updates, we define the following two probabilities:

- The probability of getting l random ratings from rating matrix R^{mxn} (P_u). Then

$$P_u = \frac{\text{The number of combinations to get l independent ratings } (U)}{The\ total\ number\ of\ combinations(A)}$$

Where $U = \prod_{k=0}^{l-1} J_k / l!$

given that $J_k = \left(J_{k-1} - 2\sqrt{J_{k-1}} + 1\right)$ and $J_0 = nm$, and $A = \binom{nm}{l}$

Proof

It is required to have l random selections for independent ratings. Let J_k is the number of possible independent ratings at $(k+1)^{th}$ selection where $0 \le k < l$. For 1st selection (k = 0), J_0=nm as we can select any rating from matrix ratings. For 2nd selection (k=1), $J_1 = nm - (2\sqrt{nm} - 1)$ where $(2\sqrt{nm} - 1)$ are the ratings which are in the same row or column of 1st selection.

At K step, $J_k = (J_{k-1} - 2\sqrt{J_{k-1}} + 1)$. Therefore, total number of permutations will be $\prod_{k=0}^{l-1} J_k$. In order to remove repetitions, we divide the $\prod_{k=0}^{l} J_k$ by $l!$.

- The probability of overwriting happens ($\bar{P}_u$) when $l > m$ or $l > n$. Then, $\bar{P}_u = 1$.

Proof

After 1st random selection of independent rating, the ratings which are sharing the same row or column removed, so the matrix dimensions are reduced to be (m-1) x (n-1). For $(m-1)^{th}$ selection, the matrix dimension become 1 x (n-m-1). Consequently, any additional selection means overwriting occurs.

Using our proposed approach where l > m, the probability of overwriting is (x/st) where x is the sparsity of the rating matrix and st is the number of steps to process rating matrix. For large practical systems, sparsity is lower than 0.005. Therefore using st =2, the probability of overwriting becomes 0.0025. Therefore assure high probability of independent updates of rows of P and Q by random access of shared memory in predefined steps (st). In case of updating the same rows of P and Q, we use atomic operations.

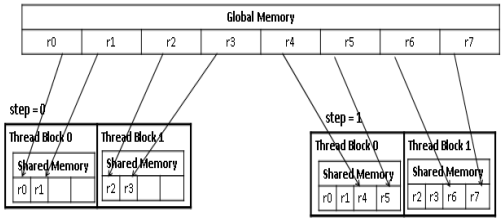

Figure 1. Process of loading rating array into shared memory of thread blocks

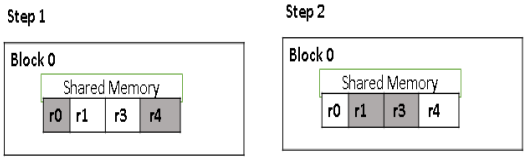

Figure 2. An example of accessing shared memory randomly

We assure a high probability of independent updates of rows of P and Q by random access of shared memory in predefined steps (st). Suppose that x% of R are suffering from the overwriting issue (sharing same user identity or/and item identity) using our approach the percentage will be (x/st)%. In a case of updating the same rows of P and Q, we use atomic operations.

As threads access ratings in shared memory randomly, therefore P and Q rows are accessed randomly, which degrades performance drastically. Therefore, we assign each consecutive k threads the task of accessing P and Q rows of a certain rating. Consequently, the access

of P and Q becomes coalesced which enhances the performance dramatically. Figure 3 and Figure 4 shows two examples of non-coalesced and coalesced access of P rows where th_size = 2 and k = 2. We assume the user_id of two ratings are 0 and 200 consequently.

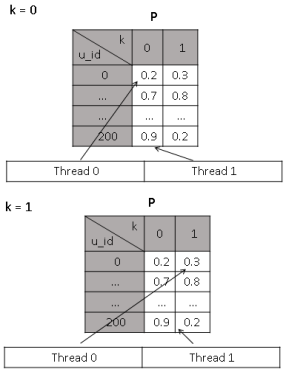

Figure 3. Example of non-coalesced access for P

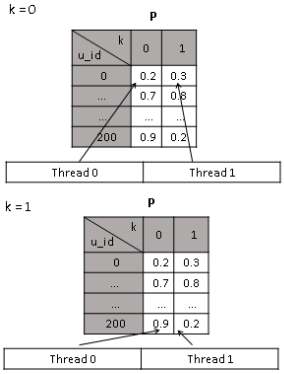

Figure 4. Example of coalesced access for P

Algorithm 1 shows the overall procedure of ESGD. First, we grid the GPU into two-dimensional thread blocks with size of $\sqrt{L/sh\_size}\ x\ \sqrt{L/sh\_size}$, and organize each thread block to have th_size of threads where th_size x st = sh_size (steps 1 and 2). Then, host copies R, feature matrices (P and Q) and Rand array to global memory of the GPU (step 3). Then, calling for kernel execution is performed (step 4). Finally, device copies P and Q to host for calculation of RMSE (steps 5 and 6). Steps 3, 4, 5 and 6 are repeated until reaching accepted RMSE. Algorithm 2 shows the GPU kernel algorithm. The modular arithmetic operation (step 21) is expensive on GPU, so that it is important to select k to be radix 2. Therefore, performing bitwise operation (x mod k = x & (k – 1)) improves the performance. For simplicity and better presentation of Algorithm 1 and 2, we assumed that results of division and root operations are integer numbers.

Algorithm 1. The overall procedure of ESGD

Require R, P, Q, st, Rand, sh_size, L, th_size, k
1. grid device in two-dimension thread blocks of size ($\sqrt{L/sh\_size}, \sqrt{L/sh\_size}$)
2. set # of threads in each thread block to be th_size
3. copy R, P, Q, Rand to device (d_R, d_P, d_Q, d_Rand respectively)
4. call device kernel ESGD_kernel(d_R, d_P, d_Q, d_Rand)
5. copy d_P and d_Q to host P and Q
6. calculate RMSE
7. Repeat steps 3, 4, 5 and 6 until reaching accepted RMSE value

Algorithm 2. The ESGD kernel

Require d_R, d_P, d_Q, d_Rand
1. col = blockIdx.x * blockDim.x + threadIdx.x
2. row= threadIdx.y
3. abs_threadIdx = row * blockDim.y + col
4. abs_blockIdx = bockIdx.y * gridDim.x + blockidx.x
5. thIdx = threadIdx.x
6. define shared memory array sh_rat of sh_size length
7. define shared memory array sh_pred of th_size/k length
8. **for** i = {0,…,st-1} **do** //loading data into shared memory
9. sh_rat[blockDim.x*i+thIdx] = d_R[blockDim.x*i+thIdx+sh_size*abs_blockIdx]
10. synchronize threads
11. **end for**
12. **for** i = {0,…,st-1} **do**
13. randIdx = Rand[thIdx + st * th_size]
14. **for** j = {0,…, k-1} **do**
15. synchronize threads
16. newThIdx = (thIdx + j * th_size)/k
17. u = sh_rat[newThIdx + j * th_size].u_id
18. v = sh_rat[newThIdx + j * th_size].i_id
19. r = sh_rat[newThIdx + j * th_size].r
20. p_q_idx = thIdx%k
21. p = P[u_id,p_q_idx]
22. q = Q[i_id,p_q_idx]
23. atomicAdd(sh_pred[thIdx/k], p*q)
24. synchronize threads
25. $e_{u,v}$ = r - sh_pred[thIdx/k]
26. d_u = $\gamma(e_{u,v}q - \lambda_P p)$
27. d_v = $\gamma(e_{u,v}p - \lambda_Q q)$
28. atomicadd(P[u_id,p_q_idx], d_u)
29. atomicadd(Q[i_id,p_q_idx], d_v)
30. **end for**
31. **end for**

4. Experiments and Results

We evaluated our proposed GPU-based ESGD on real free datasets available from MovieLens. For our experiments, we used 100K, 1M and 10M MovieLens datasets. The initial values of P and Q are randomly generated with a uniform distribution. Table 1 shows statistics and parameters used for each dataset. We evaluated the accuracy of ESGD using RMSE metric. As the objective of the research is to enhance the performance rather than RMSE, so we apply fixed learning rate instead of an adaptive learning rate.

Table 1. The statistics and parameters for each dataset

Dataset	100K	1M	10 M
m	943	6,040	71,567
n	1,683	3,952	65,133
# Training	90,000	810,082	9,000,000
# Test	10,000	89,915	1,000,000
k	16	16	16
λ	0.05	0.05	0.05
γ	0.01	0.01	0.05
sh_size	2000	2000	2000
th_size	1000	1000	1000
St	2	2	2

We first normalize data of rating matrix before applying ESGD by subtracting baseline predictor (b_{u_id,i_id}) prior to computation model [18]. Baseline predictor form used is

$$b_{u\_id,i\_id} = \mu + b_{u\_id} + b_{i\_id}$$

where μ is the average overall ratings, b_{u_id} and b_{i_id} are user and item baseline predictors, respectively. They are calculated as per the following equations:

$$b_{u\_id} = \frac{1}{|I_{u\_id}|} \sum_{i \in I_{u\_id}} (r_{u\_id,i\_id} - \mu)$$

$$b_{i\_id} = \frac{1}{|U_{i\_id}|} \sum_{u \in U_{i\_id}} (r_{u\_id,i\_id} - b_{u\_id} - \mu)$$

In prediction steps, we added the b_{u_id,i_id} to the ESGD prediction results in order to get a correct values of RMSE.

In order to calculate total time required per iteration for ESGD (t), we used the following equation:

$$t = t_{hd} + t_{GPU} + t_{dh}$$

where t_{GPU} is the GPU processing time, t_{hd} and t_{dh} are the times required to transfer data from host to GPU and vice versa, respectively.

Regarding platform, we used a Laptop Sony VPCEG28FA with Intel i5 2.4 GHz processor and 4GB RAM. The GPU is NVIDIA GeForce 410M with 512 MB dedicated RAM. After running ESGD for 100K, 1M and 10M datasets, we found that times required per iteration are 0.02, 0.20 and 2.29 sec(s) respectively. It is worthwhile to mention that time required per iteration is constant for a given dataset as the behavior of ESGD per each iteration does not change. Table 2 shows minimum RMSE and the number of iterations to get it for each dataset.

Table 2. The Minimum RMSE and the Number of Iterations to get it for each dataset

Dataset	RMSE	# of iterations
100K	0.9378	8
1M	0.8991	11
10M	0.8709	15

Compared with previous methods mentioned in the previous section, our method is the fastest one, as it does not require any time for shuffling dataset and/or sorting portions of the dataset before processing it. BSGD processed ratings from first 50,000 users on the first 2,000 movies of Netflix data [19, 20]. We used the same dataset to compare the performance of ESGD with BSGD. Although processing time of BSGD does not include preprocessing time, processing time per iteration of ESGD is 0.1 sec and the processing time for BSGD requires on best case 1.255 sec per iteration. Accordingly, ESGD demonstrates 12.5x speedup over BSGD. Figure 5 compares time required versus number of iterations between BSGD and ESGD.

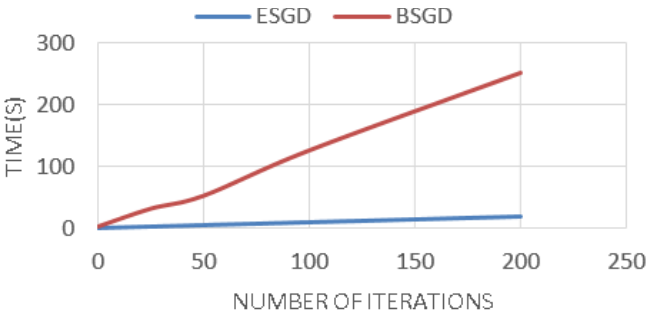

Figure 5. A comparison between BSGD and ESGD

5. Conclusions and Future work

In this research, we proposed ESGD which is GPU-based innovative parallel method for SGD. Unlike previous methods, ESGD does not require any sorting and/or random shuffling of data as preprocessing phase. In addition, ESGD does not require any complex scheduler for load balancing of data across computational resources. In ESGD, utilization of computational resources and load balancing of data are accomplished. Although the platform used for experiments is not up-to-date, our results show that ESGD is the fastest method over any previous methods. The time required per iteration is 2.29 secs for 10M dataset. Despite of that performance is the objective of ESGD, The RMSE of ESGD reaches 0.8709 in 15 iterations.

It would be beneficial to investigate the best parameters to use i.e. th_size and st. It is clear that for small values of st, number of iterations becomes smaller, however probability of updating same rows of P and Q increases which might degrade the performance. In addition, it would be extremely interesting to study the possibilities to overcome the limitation of GPU global memory i.e. ratings size is bigger than global memory size. Therefore, it is necessary to study the following two main approaches to accomplish better performance for big data. First, pipelining the data transfer from/to GPU and kernel execution. Second, using GPU clustering to parallelize the execution [20].

6. References

[1] Mohammad Aamir and Mamta Bhusry, "Recommendation System: State of the Art Approach," International Journal of Computer Applications, Volume 120, June 2015.

[2] Zheng Wen, "Recommendation System Based on Collaborative Filtering," CS229 Lecture Notes, Stanford University, December 2008.

[3] Y. Koren, R. Bell and C. Volinsky, "Matrix factorization techniques for recommender systems," Computer, vol. 42 Issue 8, pp. 30-37, August 2009.

[4] Rashid Kaleem et al., "Stochastic gradient descent on GPUs," Proceedings of the 8th Workshop on General Purpose Processing using GPUs, pp 81-89, 2015.

[5] M. Gates, et al., "Accelerating collaborative filtering using concepts from high performance computing," IEEE International Conference in Big Data (Big Data), 2015.

[6] Zhongya Wang et al., "A CUDA-enabled Parallel Implementation of Collaborative Filtering," Procedia Computer Science, Volume 30, pp 66-74, 2014.

[7] Wei-Sheng Chin, et al., "A Fast Parallel Stochastic Gradient Method for Matrix Factorization in Shared Memory Systems," ACM Transactions on Intelligent Systems and Technology (TIST), Volume 6 Issue 1, April 2015.

[8] Kimikazu Kato and Tikara Hosino, "Singular Value Decomposition for Collaborative Filtering on a GPU," IOP Conf. Series: Materials Science and Engineering, Volume 10 Issue 1, 2010.

[9] Blake Foster, et al., "A gpu-based approximate svd algorithm," In Proceedings of the 9th international conference on Parallel Processing and Applied Mathematics, Springer-Verlag, Volume 1, pp. 569–578, Berlin, Heidelberg, 2012.

[10] Hsiang-Fu Yu, et al., "Parallel matrix factorization for recommender systems," Knowledge and Information Systems(KAIS), Volume 41 Issue 3, Pages 793-819, December 2014.

[11] Christopher Robert Cullinan, et al., "Computing Performance Benchmarks among CPU, GPU, and FPGA," MathWorks, April 2012.

[12] Kevin Tang, "Collaborative Filtering with Batch Stochastic Gradient Descent," http://www.its.caltech.edu/~ktang/CS179/index.html, July 2015.

[13] Feng Niu et al., "HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent," In Advances in Neural Information Processing Systems", pp 693-701, June 2011.

[14] K. C. Nunna, Farhad Mehdipour, et al., "A Survey on Big Data Processing Infrastructure: Evolving Role of FPGA," The International Journal of Big Data Intelligence, Volume 2 Issue 3, 2015.

[15] Karydi et al., "Parallel and Distributed Collaborative Filtering: A Survey," Journal of ACM Computing Surveys (CSUR) Surveys Homepage, Volume 49 Issue 2, August 2016.

[16] GPU Memory Types – Performance Comparison, https://www.microway.com/hpc-tech-tips/gpu-memory-types, Last Visited 5 Septemper 2015.

[17] Rainer Gemulla, et al., "Large-scale matrix factorization with distributed stochastic gradient descent," In Proceedings of the 17th ACM SIGKDD.

[18] Neal Lathia, "Evaluating Collaborative Filtering Over Time," PHD Thesis, 2010.

[19] S. Gower, "Netflix Prize and SVD," http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-gower-netflix-SVD.pdf, pp. 1–10, 2014.

[20] J. Jin, et al., "GPUSGD: A GPU-accelerated stochastic gradient descent algorithm for matrix factorization," Concurrency and Computation: Practice and Experience, Volume 27, December 2015.

