GPU_MF_SGD: A Novel GPU-Based Stochastic Gradient Descent Method for Matrix Factorization

Mohamed A. Nassar(✉), Layla A. A. El-Sayed, and Yousry Taha

Department of Computer and Systems Engineering, Alexandria University, Alexandria, Egypt
eng.mohamedatif@gmail.com, labohadid@gmail.com, taha@alexu.edu.eg

Abstract. Recommender systems are used in most of nowadays applications. Providing real-time suggestions with high accuracy is considered as one of the most crucial challenges that face them. Matrix factorization (MF) is an effective technique for recommender systems as it improves the accuracy. Stochastic Gradient Descent (SGD) for MF is the most popular approach used to speed up MF. SGD is a sequential algorithm, which is not trivial to be parallelized, especially for large-scale problems. Recently, many researches have proposed parallel methods for parallelizing SGD. In this research, we propose GPU_MF_SGD, a novel GPU-based method for large-scale recommender systems. GPU_MF_SGD utilizes Graphics Processing Unit (GPU) resources by ensuring load balancing and linear scalability, and achieving coalesced access of global memory without preprocessing phase. Our method demonstrates 3.1X–5.4X speedup over the most state-of-the-art GPU method, CuMF_SGD.

Keywords: Collaborative filtering (CF) · Matrix factorization (MF) GPU implementation · Stochastic Gradient Descent (SGD)

1 Introduction

Recently, recommender systems have become a popular tool used in various applications including Facebook, YouTube, Twitter, Email services, News services and Hotel reservation applications [1–5]. In recommender systems, users get a list of suggested items (i.e. movies, friends, news, advertisements, products, etc.). One of the most important challenges that face recommender systems is suggesting accurate recommendations in real-time [6–8].

Recommender systems can be categorized into non-personalized filtering, content-based filtering (CBF), collaborative filtering (CF) and matrix factorization techniques [1, 3, 9–15]. Non-personalized recommender systems suggest items to a user based on average of ratings given to the items by other users. This category of recommender systems is trivial in terms of implementation, but it lacks personalization where recommended items are the same for all users regardless of their profile [1, 4, 13]. In CBF, items are suggested to a user based on items rated previously by the user. CBF represents items and users' behaviors as features to find similarity between users' preferences and items. Defining features that represent items and users' behaviors is a major problem in CBF [1, 14, 16]. CF is a process of suggesting items based on users' collaboration or the similarity between items [1, 3, 9, 10]. CF overcomes the issue of CBF features representation. However, CF cannot suggest items when there are no similarities between users or items. Moreover, CF performs slowly on huge datasets [17–20]. MF, a dimensionality reduction technique, is an advanced technique for recommender systems where the representation of users and items uses the same latent features. Predicting ratings is simply performed by the inner product of user-item feature vector pairs [1, 11, 14]. MF has many advantages over CF for the following reasons: (1) Computations required for predictions are so simple and have negligible time complexity; (2) high accuracy is guaranteed even if there is no similarity between users or items; and (3) MF is scalable for large-scale recommender systems.

In MF, rating matrix R of m $\times$ n is factorized into two low-rank feature matrices P (m $\times$k) and Q (k $\times$n), such that R $\simeq$ P $\times$Q where k is the number of latent features, m and n are numbers of users and items respectively. Figure 1 shows an example of matrix factorization where the following optimization rule has to be applied.

$$\min_{\mathrm{P,Q}} \sum\nolimits_{(u,v)\in R} \left(\left(r_{u,v} - p_u^T q_v \right)^2 + \lambda_P \|p_u\|^2 + \lambda_Q \|q_u\|^2 \right), \tag{1}$$

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 over-fitting. (1) is a difficult optimization problem [11, 16, 17]. To find P and Q, i.e. to build the model, it is required to perform expensive computations [21, 22].

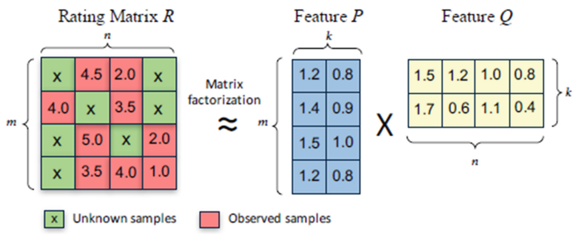


Fig. 1. An example of matrix factorization where m = 4, n = 4, k = 2 [7].

Building/rebuilding the model of users' ratings is complex in terms of computations. Therefore, many researches are directed to design fast and scalable techniques to solve (1) [7, 8, 23–32].

Three main algorithms Coordinate Descent (CD), Alternate Least Square (ALS), and SGD are proposed to solve matrix factorization problem efficiently [7, 33, 40, 41]. CD is shown to be vulnerable to stuck into local optima [22]. Authors in [7, 33] show that SGD is constantly converged faster than ALS.

Moreover, SGD is also more practical in systems where new ratings are progressively entered into the system [7]. Therefore, we focus on improving SGD in this paper.

The basic idea of SGD is to randomly select a rating $r_{u,v}$ from R where u and v are the indices of R. Then, p_u and q_v variables are updated by the following rules:

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

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

Where, $e_{u,v}$ is the difference between the actual rating $r_{u,v}$ and predicted ratings $p_u^T q_v$ 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) [11]. The time complexity per iteration of SGD is $O(|\mu|k)$, where $|\mu|$ is number of ratings. The overall SGD procedure takes hours.

It is worth to mention that there are two main streams to improve the performance of SGD. The first stream focuses on improving statistical properties to reduce the required iterations to converge [34–37]. The second stream works on improving computations per iteration by proposing efficient parallel SGD methods [6–8, 30, 38–42].

In this research, we focus on the second stream where state-of-the-art parallel SGD for MF researches mainly categorized in terms of system type into (1) shared-memory systems; and (2) distributed systems. Shared-memory systems are more efficient than distributed systems as distributed systems depend on network connection bandwidth and SGD requires aggregation of parameters/data at the end of each iteration [7, 43–45]. Nowadays, all shared-memory systems are heterogeneous which includes GPUs and/or Field-programmable Gate Arrays (FPGAs) to accelerate computationally intensive applications [46–50]. Generally, GPU has better performance that FPGA for floating-point-based applications like SGD [46, 48], as GPU comes with native floating-point processors.

Our objective is to propose an efficient parallel SGD method based on GPU, GPU_MF_SGD, which:

- Provides a high scalable SGD implementation i.e. achieves linear scalability when increasing the level of parallelism.
- Utilizes GPU resources, and ensures coalesced access of GPU global memory.
- Achieves load balancing across processing elements.
- Overcomes any preprocessing phase.
- Accesses ratings randomly in parallel.
- Reduces the probability of overwriting occurrence by processing the datasets through a predefined number of steps per iteration.

The remainder of the paper is organized as follows. In Sect. 2, we discuss existing parallelized SGD methods for MF. In Sect. 3, we present GPU_MF_SGD, our proposed efficient parallel method. Experiments and results are discussed in Sect. 4. Finally, conclusion and future work are discussed in Sect. 5.

2 Background

Although SGD is a sequential algorithm, many researches have proposed parallel efficient methods for it. Throughout this section, we discuss the state-of-the-art parallel SGD methods for MF and provide main issues associated with each one.

2.1 Hogwild

It was observed that for randomly selected ratings, the updates of feature matrices P and Q are independent, not sharing same row or column, under the condition of high sparse rating matrix R. Figure 2 shows examples of dependent and independent updates.

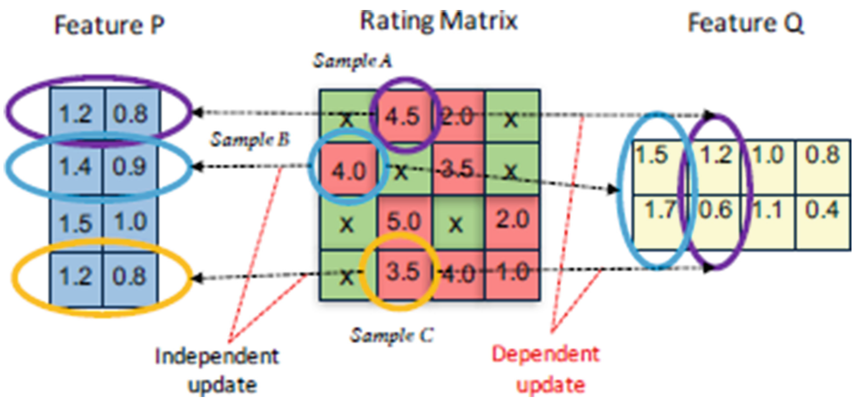


Fig. 2. Examples of independent updates (ratings B and C) and dependent updates (ratings A and C) [38].

In Hogwild [41], concurrent threads randomly select ratings from R and update P and Q. To avoid updating dependent ratings in the same time, synchronization (atomic operation) is required. Figure 3 shows updating sequence of two threads where red dot indicates that two dependent random ratings are accessed and processed simultaneously. Hogwild showed that synchronization is not required when R is very sparse and the number of concurrent threads is small compared to the number of ratings.

Hogwild was proposed for shared memory systems; however, it has many issues as follows:

- Memory discontinuity where random access of shared memory degrades system performance.
- Inapplicability on GPU where random accesses to global memory is so expensive on GPUs.

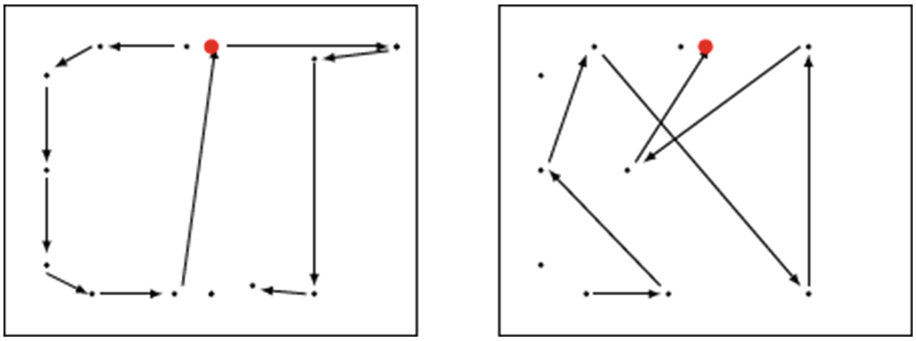


Fig. 3. An example shows updating sequences of two threads in Hogwild [40].

2.2 FSGD

FSGD [22] aimed to overcome overwriting issue and memory discontinuity by introducing the following techniques:

- **Partitioning rating matrix R.** FSGD divides R into blocks and assigns independent blocks to threads.
- **Lock-free scheduling.** Once a thread finishes processing a block, the scheduler assigns a new block, which satisfies the following two criteria. First, it is a free block. Second, the number of past updates is the smallest among all free blocks.
- **Partial random method.** To overcome the issue of memory discontinuity, FSGD simply accesses rating within blocks sequentially, but blocks selection is performed randomly.
- **Random shuffling of R and sorting blocks.** FSGD overcomes the issue of imbalanced distribution of ratings across blocks by random shuffling ratings and sorting partitioned blocks.

Despite the popularity of FSGD, it has a complex preprocessing phase, which includes a complex scheduler, random shuffling of ratings and sorting each block by user identities.

2.3 GPUSGD

GPUSGD [6] proposed SGD method based on matrix blocking using GPU. It divides a rating matrix R into blocks, which are mutually independent, and their corresponding variables are updated in parallel. Independent blocks run simultaneously using thread blocks of GPU. Authors prove that all independent ratings inside each block can be tagged with the same tag number. Therefore, a preprocessing phase which includes tagging and sorting ratings is required to provide coalesced access and independent updates. The experimental results show that GPUSGD performs much better in accelerating the matrix factorization compared with the existing state-of-the-art parallel methods. However, GPUSGD suffers from intensive prepossessing phase (tagging, sorting and partitioning) and load imbalance through GPU blocks and threads.

2.4 CuMF_SGD

CuMF_SGD [7] is the most recent parallelized SGD method based on GPU. Two equivalent schemes in terms of accuracy and performance (Batch-Hogwild and Wavefront-update) were proposed. CuMF_SGD overcomes complexity and consumed time to schedule blocks when the number of thread blocks becomes large. CuMF_SGD utilizes GPU resources using half-precision (2 bytes), which does not affect the accuracy and improves memory bandwidth. In addition, it accesses global memory in coalesced manner.

CuMF_SGD exploits the spatial data locality using L1 cache. Preprocessing phase is necessary to shuffle ratings and partitioning data into batches. Wavefront-update reduces the existing complex scheduling schemes [21, 22], which maintains two-dimensional lookup table to find the coordinate (row and column) to update. Wavefront-update uses only one-dimensional lookup which only maintains columns. Figure 4 shows an example of four concurrent thread blocks (workers) working on R, which is partitioned, into 4 × 8 blocks. At first iteration (waive), workers are assigned to independent blocks and update the status of columns in the lockup. After processing the independent blocks, workers need to check the status of the columns before processing other blocks. Wavefront-update requires preprocessing phase of partitioning and maintains a scheduler. In addition, the scheduler cannot maintain the same number of updates per block if ratings are not uniformly distributed across blocks [22].

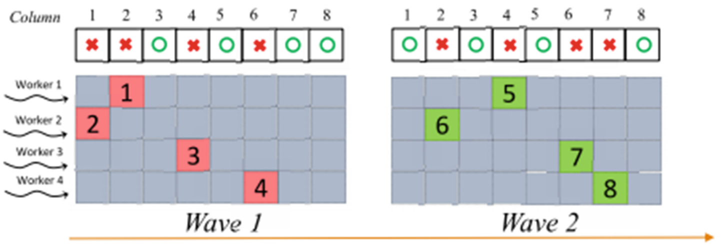


Fig. 4. Wavefront-update example where each parallel worker is assigned to a row and a randomized column update sequence [7].

Throughout this section, we introduced the recent existing parallel methods to enhance SGD for MF. In addition, we highlighted the main issues associated with each method. In the following section, we discuss our novel GPU-based method, which aims to overcome the main issues.

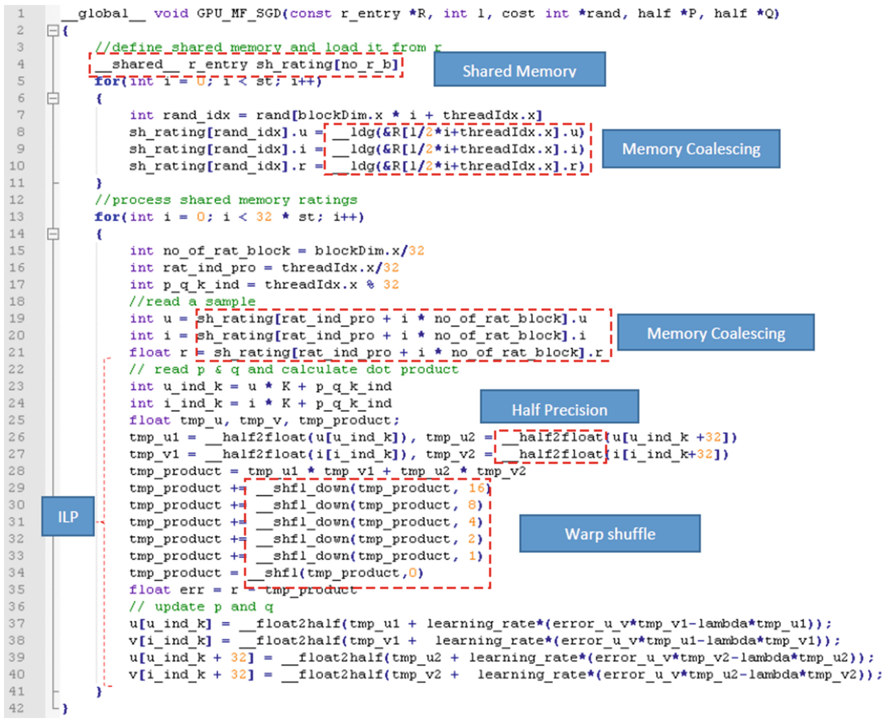
3 A Novel GPU-Based SGD Method for MF

According to the most recent researches and existing recommender systems, we can summarize the following observations.

- **Observation 1.** SGD for MF is memory bound i.e. the number of floating point operations is lower than the number of memory accesses by 37% in SGD [7]. Consequently, memory access utilization directly affects the performance of the SGD. In addition, proposing efficient GPU method can improve the performance of SGD as GPU has higher memory bandwidth and inter-device connection speed compared with CPU.
- **Observation 2.** All proposed methods neglect a considerable execution time of the preprocessing phase. Recent proposed methods [6–8, 30] consist of preprocessing phase and processing phase. Preprocessing phase includes one or more of the following procedures: shuffling dataset, partitioning the dataset into blocks, sorting partitioned dataset and/or scheduling execution of dataset blocks using a complex scheduler. For the most recent method CuMF_SGD [7], we found that around 40% of the overall execution time (preprocessing time + processing time) is spent in preprocessing phase for 20 M MovieLens dataset when k = 32 and running on NVIDIA Tesla K80 [51]. Therefore, overcome preprocessing phase enhances the overall performance of recommender systems.
- **Observation 3. Rating matrices for recommender systems are highly sparse.** For 20 M Movielens, Netflix, Yahoo and Hugewiki datasets, matrix densities are 0.11%, 1.17%, 0.29% and 0.15% respectively.
- **Observation 4. Random parallel processing of the rating matrix R does not affect the accuracy when R is very sparse and the number of threads is lower than the number of ratings.** Hogwild [40] proved that overwriting issue, which may occur because of random parallel processing of the rating matrix R, does not require atomic operations and does not affect the accuracy.
- **Observation 5. Existing methods suffer scalability issues [7].** Due to the complex scheduler and/or required synchronization between processing elements, existing methods scale only to a limited number of processing elements/threads.
- **Observation 6. Load Imbalance is the reason of imbalance in ratings distribution across dataset blocks.** Recommender systems are highly dynamic systems [52] where the number of ratings, number of users and number of items are changing over time. Partitioning rating matrix into blocks and assigning them uniformly across processing elements lead to load imbalance and therefore non-utilized resources.

Based on the mentioned observations, we introduce an efficient SGD method for MF based on GPU, GPU_MF_SGD. Before discussing the proposed method, it is worth to mention that the representation of the rating matrix R is Coordinate list (COO) [53] i.e. R is represented as one-dimensional array of length l where l is number of ratings in R. Each entry of R has structure r_entry (u, i, r), where u is user identity, i is item identity and r is the rating of user u to item i. Figure 5 shows the code of the GPU_MF_SGD kernel. We describe the algorithm throughout the following main optimization techniques.

- **Shared memory utilization.** Instead of accessing rating matrix R randomly from global memory (memory discontinuity) [40], we utilize shared memory which is two orders of magnitude faster than global memory access to shuffle R and improves system performance as follows [54–56]. We configure each thread block



```
1  __global__ void GPU_MF_SGD(const r_entry *R, int l, cost int *rand, half *P, half *Q)
2  {
3      //define shared memory and load it from r
4      __shared__ r_entry sh_rating[no_r_b]
5      for(int i = 0; i < st; i++)
6      {
7          int rand_idx = rand[blockDim.x * i + threadIdx.x]
8          sh_rating[rand_idx].u = __ldg(&R[l/2*i+threadIdx.x].u)
9          sh_rating[rand_idx].i = __ldg(&R[l/2*i+threadIdx.x].i)
10         sh_rating[rand_idx].r = __ldg(&R[l/2*i+threadIdx.x].r)
11     }
12     //process shared memory ratings
13     for(int i = 0; i < 32 * st; i++)
14     {
15         int no_of_rat_block = blockDim.x/32
16         int rat_ind_pro = threadIdx.x/32
17         int p_q_k_ind = threadIdx.x % 32
18         //read a sample
19         int u = sh_rating[rat_ind_pro + i * no_of_rat_block].u
20         int i = sh_rating[rat_ind_pro + i * no_of_rat_block].i
21         float r = sh_rating[rat_ind_pro + i * no_of_rat_block].r
22         // read p & q and calculate dot product
23         int u_ind_k = u * K + p_q_k_ind
24         int i_ind_k = i * K + p_q_k_ind
25         float tmp_u, tmp_v, tmp_product;
26         tmp_u1 = __half2float(u[u_ind_k]), tmp_u2 = __half2float(u[u_ind_k +32])
27         tmp_v1 = __half2float(i[i_ind_k]), tmp_v2 = __half2float(i[i_ind_k+32])
28         tmp_product = tmp_u1 * tmp_v1 + tmp_u2 * tmp_v2
29         tmp_product += __shfl_down(tmp_product, 16)
30         tmp_product += __shfl_down(tmp_product, 8)
31         tmp_product += __shfl_down(tmp_product, 4)
32         tmp_product += __shfl_down(tmp_product, 2)
33         tmp_product += __shfl_down(tmp_product, 1)
34         tmp_product = __shfl(tmp_product,0)
35         float err = r - tmp_product
36         // update p and q
37         u[u_ind_k] = __float2half(tmp_u1 + learning_rate*(error_u_v*tmp_v1-lambda*tmp_u1));
38         v[i_ind_k] = __float2half(tmp_v1 +  learning_rate*(error_u_v*tmp_u1-lambda*tmp_v1));
39         u[u_ind_k + 32] = __float2half(tmp_u2 + learning_rate*(error_u_v*tmp_v2-lambda*tmp_u2));
40         v[i_ind_k + 32] = __float2half(tmp_v2 +  learning_rate*(error_u_v*tmp_u2-lambda*tmp_v2));
41     }
42 }
```

Fig. 5. The exemplify code of GPU_MF_SGD Kernel with highlighted optimization techniques where K = 64.

to have *th_size* threads and shared memory array (*sh_rating*) with predefined length (*no_r_b*) where *no_r_b/th_size* = *st* (Step 4). st is the number of iterations required for each thread block to process *sh_rating*. There are two main reasons behind the idea of processing sh_rating in *st* steps as follows: (1) to utilize resources as for each thread block, the available shared memory size is multiple of the available threads; and (2) to reduce matrix density by (100/st)%, thus reducing the probability of dependent updates. Shuffling R is guaranteed by coalesced access to global memory and random accesses to *sh_rating* using offline calculated array of random numbers (rand) (Steps 5, 7, 8, 9). Figure 6 shows an example of loading R to the shared memory of two thread blocks where *st* = 2, *l* = 8 and *th_size* = 2. It can be shown that two levels of shuffling are performed with negligible time complexity, as (1) each thread loads two ratings into shared memory; first rating is from the first half of R and the second rating is from the second half of R; and (2) each thread accesses shared memory randomly.

- **Coalesced Access of P and Q.** Although threads access ratings from shared memory in a coalesced manner (Steps 19, 20, 21), readings and writings for rows of P and Q are performed randomly which degrade performance drastically. To overcome this issue, we configure each consecutive 32 threads to complete

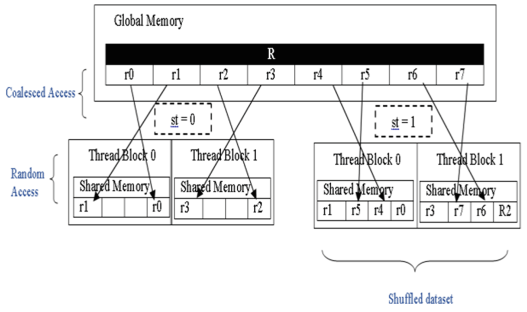


Fig. 6. An example of the process of loading rating array R into shared memory of thread blocks.

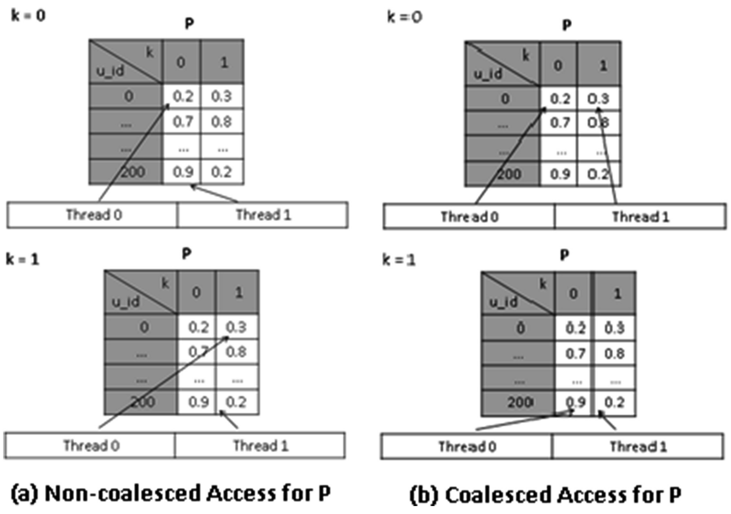


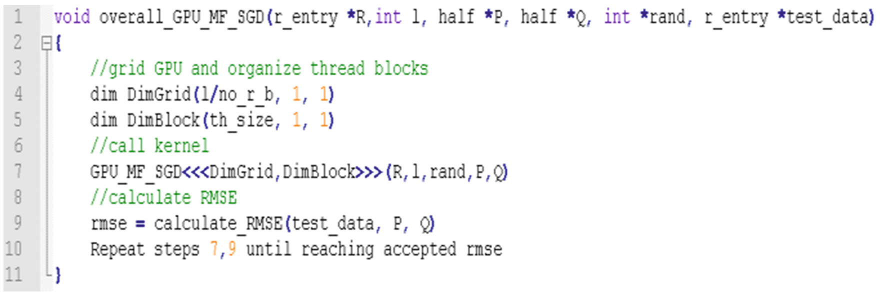
Fig. 7. Examples of non-coalesced and coalesced access for P.

computations of a rating (Steps 23, 24, 26, 27). Figure 7 shows two examples of non-coalesced and coalesced access of P rows where *th_size* = 2 and k = 2.

- **Warp shuffle [6, 57].** Instead of using shared memory to synchronize and broadcast dot product results of p and q, we use warp shuffle to broadcast the result of the dot product results (Steps 29 to 34). The warp shuffle has better performance than shared memory as (1) it uses extra hardware support; (2) register operations are faster than shared memory operations; and (3) there is no need to synchronize between threads [7].
- **Instruction level parallelism (ILP) [7].** For K > 32, each thread is responsible for K/32. Instructions order is considered to maximize the ILP (Steps from 23 to 40).

- **Half-precision.** As SGD for MF is a memory-bounded algorithm, any enhancement for memory access will improve the performance. New GPU architecture offers storage of half-precision (2 bytes) which is fast to be transformed to float and does not affect the accuracy [7] (Steps 26, 27).
- **Warp divergence avoidance [58].** GPU has performance penalties with conditional statements as different paths of executions are generated. GPU_MF_SGD does not contain conditional branches, which improves overall performance.
- **No preprocessing phases.** All existing methods have a computational intensive preprocessing phase, which includes sorting, partitioning, random shuffling, scheduling, etc. to ensure independent updates, random access, and load balance. GPU_MF_SGD does not include any preprocessing phase as we guarantee a high probability of independent updates for P and Q by random access of ratings and processing R in predefined steps.
- **Linear Scalability.** If we increase the number of threads, GPU_MF_SGD theoretically achieve linear scalability. Unlike existing methods, they lack scalability due to required synchronization and/or scheduling [6, 7, 22, 40].

Figure 8 shows a code of GPU_MF_SGD overall procedure. First, we grid the GPU into one-dimensional thread blocks with a size of l/no_r_b, and organize each thread block into 1D threads of size th_size (Steps 4, 5). Then, calling for kernel execution is performed (Step 7). Finally, calculation of RMSE is performed (Step 9). Steps 7, 9 are repeated until reaching accepted RMSE.



```
1   void overall_GPU_MF_SGD(r_entry *R,int l, half *P, half *Q, int *rand, r_entry *test_data)
2   {
3       //grid GPU and organize thread blocks
4       dim DimGrid(l/no_r_b, 1, 1)
5       dim DimBlock(th_size, 1, 1)
6       //call kernel
7       GPU_MF_SGD<<<DimGrid,DimBlock>>>(R,l,rand,P,Q)
8       //calculate RMSE
9       rmse = calculate_RMSE(test_data, P, Q)
10      Repeat steps 7,9 until reaching accepted rmse
11  }
```

Fig. 8. Exampify code of GPU_MF_SGD overall procedure.

4 Experiments and Results

We implemented GPU_MF_SGD using Compute Unified Device Architecture (CUDA). Different types of public datasets are used to evaluate performance and accuracy. It is worth to mention that we compared our results with state-of-the-art GPU method, CuMF_SGD [7] for the following reasons:

- CuMF_SGD outperforms all existing shared memory methods by 3.1X – 28.2X.
- CuMF_SGD source code is publicly available, unlike other existing GPU methods.
- CuMF_SGD has less computational complexity for preprocessing phase compared with other implementation.
- CuMF_SGD has consistent results and graphs, unlike other existing GPU methods.

4.1 Experimental Setup

We executed both implementations (CuMF_SGD and GPU_MF_SGD) on high-performance computing service (HPC) provided by the Bibliotheca Alexandria [66]. Table 1 shows specifications of the used platform.

Table 1. Specifications of the used platform

RAM size	128 GB	Operating system	CentOS 6.8
Number of CPUs	2	Scheduler	Slurm [67]
GPU used	NVIDIA Tesla K80	Number of GPU devices	2

We used common public datasets: MovieLens [59], Netflix [60, 61] and Yahoo! Music [62, 63]. Table 2 shows details of datasets used in experiments. We extracted 10% test random sample from different datasets using GraphLab [64, 65].

Table 2. Details about the datasets used in experiments

Dataset	MovieLens	Netflix	Yahoo!Music
M	138493	480189	1823178
N	27278	17770	136735
K	32	64	128
# Training set	18000236	90432454	646084797
# Test set	2000027	10048051	71787199

The setup parameters for GPU_MF_SGD are as follows. We set *th_size* to be 1024, which is the maximum number of threads available per thread block for GPU. In addition, we chose *st* to be 2 to shuffle dataset and achieve a high level of sparsity with reasonable complexity performance. Regarding SGD parameters for both CuMF_SGD and GPU_MF_SGD, we used common parameters used by previous work. For learning rate, we used the same learning rate scheduling technique used by [30], where s_t, the learning rate at iteration t, is reduced using the following formula:

$$s_t = \frac{\alpha}{1 + \beta \,.\, t^{1.5}} \tag{5}$$

α is the initial learning rate and β is a constant parameter. The parameters are shown in Table 3.

Table 3. The parameters used per dataset

Dataset	λ	α	ß
MovieLens	0.05	0.08	0.3
Netflix	0.05	0.08	0.3
Yahoo!Music	0.05	0.08	0.2

4.2 Scalability Study

To study the scalability of both methods, we used the number of updates per second as the performance metric [7]:

$$\#\ update/s = \frac{\#Iterations \times \#Samples}{Elapsed\ Time} \tag{6}$$

where # Iterations, # Samples and Elapsed Time indicate the number of iterations, number of ratings in R, and execution time in seconds, respectively.

Scalability study of CuMF_SGD and GPU_MF_SGD for the MovieLens dataset is shown in Fig. 9. We have two curves for CuMF_SGD (CuMF_SGD_Pro and CuMF_SGD_Pre) where in CuMF_SGD_Pro, the elapsed time is only GPU execution time, and in CuMF_SGD_Pre, the elapsed time is execution time plus preprocessing time. GPU_MF_SGD implementation shows linear scalability while CuMF_SGD has limitations in terms of scalability.

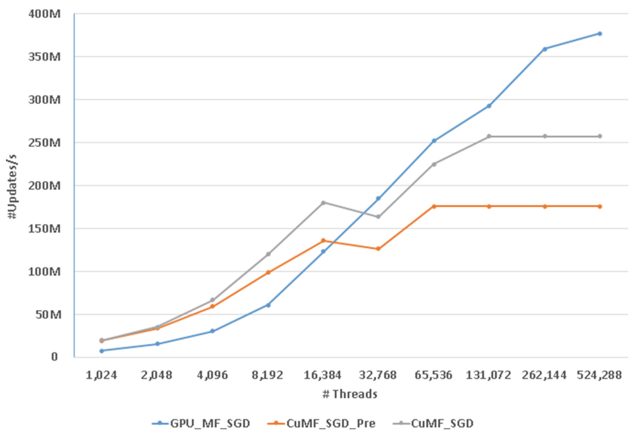


Fig. 9. # updates/s for different methods versus #threads for MovieLens.

Table 4. Training times and GPU_MF_SGD speedup

Dataset	CuMF_SGD_Pro	CuMF_SGD_Pre	GPU_MF_SGD	GPU_MF_SGD speedup	
				CuMF_SGD_Pro	CuMF_SGD_Pre
MovieLens	0.56 s	0.98 s	0.18 s	3.1X	5.4X
Netflix	4.34 s	6.70 s	1.26 s	3.4X	5.3X
Yahoo! Music	49.08 s	66.1 s	15 s	3.3X	4.4X

4.3 Training Time Speedup

We measured training time until convergence to an accepted RMSE [68, 69] (0.93, 0.92, 1.23) for MovieLens, Netflix, and Yahoo!Music respectively). Table 4 shows training times and GPU_MF_SGD speedup over both CuMF_SGD_Pro and CuMF_SGD_Pre.

Results show that GPU_MF_SGD is 3.1X – 5.4X, 3.4X – 5.3X and 3.3X – 4.4X faster than CuMF_SGD for MovieLens, Netflix, and Yahoo!Music respectively. Generally, GPU_MF_SGD outperforms CuMF_SGD by 3.1X to 5.4X for all datasets.

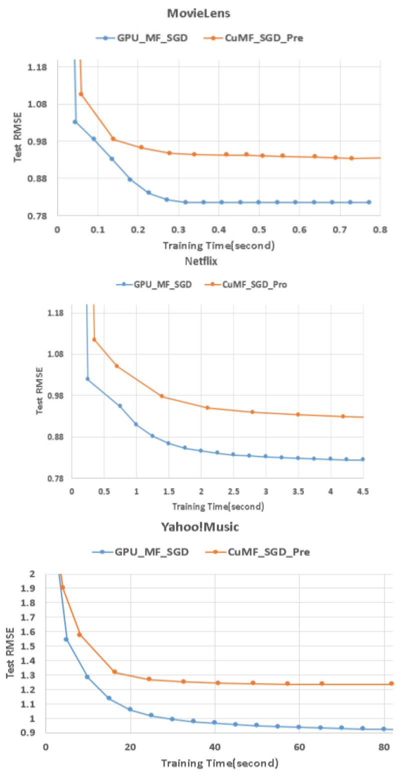


Fig. 10. Convergence speed for different datasets.

4.4 Convergence Analysis

Figure 10 shows the RMSE on test set with respect to the training time. It is obvious that our method converges faster than CuMF_SGD and achieves better RMSE for all datasets.

Therefore, GPU_MF_SGD is considered as the fastest SGD method for MF because it can do more updates per second, as shown in Fig. 9. Unlike all previous methods including CuMF_SGD, GPU_MF_SGD utilizes shared memory and does not require any preprocessing phase.

5 Conclusions and Future Work

In this research, we proposed GPU_MF_SGD, which is GPU-based innovative parallel SGD method for MF. Unlike previous methods, GPU_MF_SGD does not require any preprocessing phase like sorting and/or random shuffling of the dataset. In addition, GPU_MF_SGD does not require any complex scheduler for load balancing of datasets across computational resources. In GPU_MF_SGD, utilization of computational resources, high scalability, and load balance are achieved. Our empirical study shows that GPU_MF_SGD provides the highest number of updates per sec and considered as the fastest method. Evaluations on common public datasets show that GPU_MF_SGD runs 3.1X – 5.4X faster than CuMF_SGD.

In GPU_MF_SGD method, we did not spend much effort in parameters tuning. We suggest studying different optimization techniques for parameter selection as a future work. Furthermore, it would be extremely interesting to study the possibilities to overcome the limitation of GPU global memory i.e. the size of ratings is bigger than global memory size. Therefore, for future work, scaling up our proposed method to run on multiple GPUs [6, 7, 70–72] is an interesting research point.

References

1. Ricci, F., et al.: Recommender Systems Handbook. Springer, New York (2011)
2. Ekstrand, M.D., et al.: Collaborative filtering recommender systems. Found. Trends Hum. Comput. Interact. **4**(2), 81–173 (2011)
3. Poriya, A., et al.: Non-personalized recommender systems and user-based collaborative recommender systems. Int. J. Appl. Inf. Syst. **6**(9), 22–27 (2014)
4. Aamir, M., Bhusry, M.: Recommendation system: state of the art approach. Int. J. Comput. Appl. **120**, 25–32 (2015)
5. Recommender System. https://en.wikipedia.org/wiki/Recommender_system. Accessed 11 July 2017
6. Jin, J., et al.: GPUSGD: a GPU-accelerated stochastic gradient descent algorithm for matrix factorization. Concurr. Comput. Pract. Exp. **28**, 3844–3865 (2016)
7. Xie, X., et al.: CuMF_SGD: parallelized stochastic gradient descent for matrix factorization on GPUs. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. ACM (2017)

8. Li, H., et al.: MSGD: a novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs. IEEE Trans. Parallel Distrib. Syst. **29**(7), 1530–1544 (2018)
9. Nassar, M.A., El-Sayed, L.A.A., Taha, Y.: Efficient parallel stochastic gradient descent for matrix factorization using GPU. In: 2016 11th International Conference for Internet Technology and Secured Transactions (ICITST). IEEE (2016)
10. Wen, Z.: Recommendation system based on collaborative filtering. In: CS229 Lecture Notes, Stanford University, December 2008
11. Leskovec, J., et al.: Mining of Massive Datasets, Chap. 9, pp. 307–340. Cambridge University Press, Cambridge (2014)
12. Koren, Y., Bell, R., Volinsky, C.: Matrix factorization techniques for recommender systems. Computer **42**(8), 30–37 (2009)
13. Kaleem, R., et al.: Stochastic gradient descent on GPUs. In: Proceedings of the 8th Workshop on General Purpose Processing Using GPUs, pp. 81–89 (2015)
14. Konstan, J.A., Riedl, J.: Recommender systems: from algorithms to user experience. User Model. User Adap. Inter. **22**(1), 101–123 (2012)
15. Anastasiu, D.C., et al.: Big Data and Recommender Systems (2016)
16. Melville, P., Sindhwani, V.: Recommender systems. In: Sammut, C., Webb, G.I. (eds.) Encyclopedia of Machine Learning, pp. 829–838. Springer, New York (2011)
17. Kant, V., Bharadwaj, K.K.: Enhancing recommendation quality of content-based filtering through collaborative predictions and fuzzy similarity measures. J. Proc. Eng. **38**, 939–944 (2012)
18. Ma, A., et al.: A FPGA-based accelerator for neighborhood-based collaborative filtering recommendation algorithms. In: Proceedings of IEEE International Conference on Cluster Computing, pp. 494–495, September 2015
19. Anthony, V., Ayala, A., et al.: Speeding up collaborative filtering with parametrized preprocessing. In: Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, Australia, August 2015
20. Gates, M., et al.: Accelerating collaborative filtering using concepts from high performance computing. In: IEEE International Conference in Big Data (Big Data) (2015)
21. Wang, Z., et al.: A CUDA-enabled parallel implementation of collaborative filtering. Proc. Comput. Sci. **30**, 66–74 (2014)
22. Gemulla, R., Nijkamp, E., Haas, P.J., Sismanis, Y.: Large-scale matrix factorization with distributed stochastic gradient descent. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM (2011)
23. Chin, W.-S., et al.: A fast parallel stochastic gradient method for matrix factorization in shared memory systems. ACM Trans. Intell. Syst. Technol. **6**(1), 2 (2015)
24. Zastrau, D., Edelkamp, S.: Stochastic gradient descent with GPGPU. In: Proceedings of the 35th Annual German Conference on Advances in Artificial Intelligence (KI'12), pp. 193–204 (2012)
25. Shah, A., Majumdar, A.: Accelerating low-rank matrix completion on GPUs. In: Proceedings of International Conference on Advances in Computing, Communications and Informatics, December 2014
26. Kato, K., Hosino, T.: Singular value decomposition for collaborative filtering on a GPU. IOP Conf. Ser. Mater. Sci. Eng. **10**(1), 012017 (2010)
27. Foster, B., et al.: A GPU-based approximate SVD algorithm. In: Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics, vol. 1, pp. 569–578. Springer, Berlin (2012)
28. Yu, H.-F., et al.: Parallel matrix factorization for recommender systems. Knowl. Inf. Syst. **41** (3), 793–819 (2014)

29. Yu, H.F., Hsieh, C.J., et al.: Scalable coordinate descent approaches to parallel matrix factorization for recommender systems. In: Proceedings of the IEEE 12th International Conference on Data Mining, pp. 765–774 (2012)
30. Yun, H., Yu, H.-F., Hsieh, C.-J., Vishwanathan, S.V.N., Dhillon, I.: NOMAD: non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion. Proc. VLDB Endow. **7**(11), 975–986 (2014)
31. Yang, X., et al.: High performance coordinate descent matrix factorization for recommender systems. In: Proceedings of the Computing Frontiers Conference. ACM (2017)
32. Zadeh, R., et al.: Matrix completion via alternating least square (ALS). In: CME 323 Lecture Notes, Stanford University, Spring (2016)
33. Tan, W., Cao, L., Fong, L.: Faster and cheaper: parallelizing large-scale matrix factorization on GPUs. In: Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016 (2016)
34. Aberger, C.R.: Recommender: An Analysis of Collaborative Filtering Techniques (2016)
35. Papamakarios, G.: Comparison of Modern Stochastic Optimization Algorithms (2014)
36. Toulis, P., Airoldi, E., Rennie, J.: Statistical analysis of stochastic gradient methods for generalized linear models. In: International Conference on Machine Learning, pp. 667–675 (2014)
37. Toulis, P., Tran, D., Airoldi, E.: Towards stability and optimality in stochastic gradient descent. In: Artificial Intelligence and Statistics, pp. 1290–1298 (2016)
38. Zhou, Y., Wilkinson, D., et al.: Large-scale parallel collaborative filtering for the Netflix prize. In: Proceedings of International Conference on Algorithmic Aspects in Information and Management (2008)
39. Xie, X., Tan, W., Fong, L.L., Liang, Y.: Cumf_sgd: fast and scalable matrix factorization (2016). arXiv preprint arXiv:1610.05838. https://github.com/cuMF/cumf_sgd
40. Tang, K.: Collaborative filtering with batch stochastic gradient descent, July 2015. http://www.its.caltech.edu/~ktang/CS179/index.html
41. Niu, F., et al.: HOGWILD!: a lock-free approach to parallelizing stochastic gradient descent. In: Advances in Neural Information Processing Systems, pp. 693–701, June 2011
42. Gemulla, R., et al.: Large-scale matrix factorization with distributed stochastic gradient descent. In: Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 69–77 (2011)
43. Zhang, H., Hsieh, C.-J., Akella, V.: Hogwild++: a new mechanism for decentralized asynchronous stochastic gradient descent. In: 2016 IEEE 16th International Conference on Data Mining (ICDM), pp. 629–638. IEEE (2016)
44. Zhang, C., Ré, C.: Dimmwitted: a study of main-memory statistical analytics. Proc. VLDB Endow. **7**(12), 1283–1294 (2014)
45. Udell, M., et al.: Generalized low rank models. Found. Trends Mach. Learn. **9**(1), 1–118 (2016)
46. CUDA C Programming Guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4FH9nydq8. Accessed 5 Sept 2016
47. Nunna, K.C., et al.: A survey on big data processing infrastructure: evolving role of FPGA. Int. J. Big Data Intell. **2**(3), 145–156 (2015)
48. Nassar, M.A., El-Sayed, L.A.A.: Radix-4 modified interleaved modular multiplier based on sign detection. In: International Conference on Computer Science and Information Technology, pp. 413–423. Springer, Berlin (2012)
49. Nassar, M.A., El-Sayed, L.A.A.: Efficient interleaved modular multiplication based on sign detection. In: IEEE/ACS 12th International Conference of Computer Systems and Applications (AICCSA), November 2015

50. Karydi, E., et al.: Parallel and distributed collaborative filtering: a survey. J. ACM Comput. Surv. **49**(2), 37 (2016)
51. Ma, X., Wang, C., Yu, Q., Li, X., Zhou, X.: A FPGA-based accelerator for neighborhood-based collaborative filtering recommendation algorithms. In: 2015 IEEE International Conference on Cluster Computing (CLUSTER), pp. 494–495. IEEE (2015)
52. http://www.nvidia.com/object/tesla-k80.html. Accessed 22 July 2017
53. Lathia, N.: Evaluating collaborative filtering over time. Ph.D. thesis (2010)
54. Sparse Matrix. https://en.wikipedia.org/wiki/Sparse_matrix#Storing_a_sparse_matrix. Accessed 12 Feb 2017
55. http://supercomputingblog.com/cuda/cudamemoryandcachearchitecture/. Accessed 26 June 2017
56. GPU memory types – performance comparison. https://www.microway.com/hpc-tech-tips/gpu-memory-types. Accessed 5 Sept 2015
57. Pankratius, V., et al.: Fundamentals of Multicore Software Development. CRC Press, Boca Raton (2011)
58. del Mundo, C., Feng, W.: Enabling efficient intra-warp communication for fourier transforms in a many-core architecture. In: Proceedings of the 2013 ACM/IEEE International Conference on Supercomputing (2013)
59. Han, T.D., Abdelrahman, T.S.: Reducing branch divergence in GPU programs. In: Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units, p. 3. ACM (2011)
60. Harper, F.M., Konstan, J.A.: The MovieLens datasets: history and context. ACM Trans. Interact. Intell. Syst. **5**(4), 19 (2016)
61. Gower, S.: Netflix prize and SVD, pp. 1–10. http://buzzard.ups.edu/courses/2014spring/420projects/math420-UPS-spring-2014-gower-netflix-SVD.pdf (2014)
62. Bennett, J., Lanning, S.: The Netflix prize. In: Proceedings of KDD Cup and Workshop, p. 35 (2007)
63. Dror, G., Koenigstein, N., Koren, Y., Weimer, M.: The Yahoo! music dataset and KDD-Cup'11. In: Proceedings of KDD Cup 2011, pp. 3–18 (2012)
64. Zheng, L.: Performance evaluation of latent factor models for rating prediction. Ph.D. dissertation, University of Victoria (2015)
65. Low, Y., et al.: GraphLab: a new parallel framework for machine learning. In: Proceedings of the Twenty-Sixth Annual Conference on Uncertainty in Artificial Intelligence, UAI-10, pp. 340–349, July 2010
66. Chin, W.-S., et al.: A learning-rate schedule for stochastic gradient methods to matrix factorization. In: PAKDD, pp. 442–455 (2015)
67. https://hpc.bibalex.org/. Accessed July 2017
68. https://slurm.schedmd.com/. Accessed July 2017
69. Shani, G., Gunawardana, A.: Evaluating recommendation systems. In: Ricci, F., Rokach, L., Shapira, B., Kantor, P. (eds.) Recommender Systems Handbook, pp. 257–297. Springer, Boston (2011)
70. Ginger, T., Bochkov, Y.: Predicting business ratings on yelp report (2015). http://cs229.stanford.edu/proj2015/013_report.pdf
71. Hwu, W.: Efficient host-device data transfer. In: Lecture Notes, University of Illinois at Urbana-Champaign, December 2014
72. Bhatnagar, A.: Accelerating a movie recommender system using VirtualCL on a heterogeneous GPU cluster. Master thesis, July 2015