Matrix Factorization on GPUs with Memory Optimization and Approximate Computing

Wei Tan*
Citadel
Chicago, IL 60603
weitan@ieee.org

Shiyu Chang
IBM Research
Yorktown Heights, NY 10598
shiyu.chang@ibm.com

Liana Fong
IBM Research
Yorktown Heights, NY 10598
llfong@us.ibm.com

Cheng Li*
University of Illinois at Urbana-Champaign
Urbana, IL 60801
cli99@illinois.edu

Zijun Wang
IBM Research
Yorktown Heights, NY 10598
zijun.wang@ibm.com

LiangLiang Cao*
HelloVera.AI
New York, NY 10013
llc@hellovera.ai

ABSTRACT

Matrix factorization (MF) discovers latent features from observations, which has shown great promises in the fields of collaborative filtering, data compression, feature extraction, word embedding, *etc.* While many problem-specific optimization techniques have been proposed, alternating least square (ALS) remains popular due to its general applicability (*e.g.* easy to handle positive-unlabeled inputs), fast convergence and parallelization capability. Current MF implementations are either optimized for a single machine or with a need of a large computer cluster but still are insuffcent. This is because a single machine provides limited compute power for large-scale data while multiple machines suffer from the network communication bottleneck.

To address the aforementioned challenge, accelerating ALS on garphics processing units (GPUs) is a promising direction. We propose a novel approach in this paper. We analyze the procedure of MF and focus on enhancing the efficiency via both **memory optimization** and **approximate computing**. The former exploits GPU memory hierarchy to increase data reuse, while the later reduces unnecessary computing without hurting the convergence of the learning algorithm. Extensive experiments on large-scale datasets show that our system not only outperforms all competing CPU solutions by a large margin but also has a **2x-4x** performance gain compared to the state-of-the-art GPU solution. Our implementations are open-sourced and publicly available.

CCS CONCEPTS

•**Computing methodologies** → **Factor analysis;**
•**Computer systems organization** → **Heterogeneous (hybrid) systems;**
•**Theory of computation** → *Massively parallel algorithms;*

*Work done while at IBM Research.

ICPP 2018, Eugene, OR, USA
© 2018 ACM. 978-1-4503-6510-9/18/08...$15.00
DOI: 10.1145/3225058.3225096

KEYWORDS

Matrix factorization, machine learning, GPU, CUDA, Parallel Computing.

1 INTRODUCTION

Matrix factorization (MF) is one of the most important data mining techniques due to its implementation simplicity and broad applicability. For instance, MF is the core of modern recommender systems [13, 15, 33]. MF has also been widely used in compressing large models (*e.g.* deep neural networks) for mobile usage [1], calculating word embedding [15, 28], *etc.* However, the big data processing, with massive data is generated at an unprecedented rate, demands further acceleration of MF. For example, the number of active users of Facebook exceed 1.860 billion in the fourth quarter of 2016[1]. Solving MF efficiently under such a large scale challenges many existing solutions.

Although many studies [3, 9, 16, 22, 27, 30, 36, 37, 39] have been conducted to accelerate MF, they are still insufficient to process large scale data set. These methods either use multiple threads on one machine or multiple processes on distributed systems. The former one uses shared memory which is efficient but hard to handle big data in real-world settings. On the other hand, the communication cost becomes the major bottleneck in distributed systems, which significantly reduces its efficiency in terms of aggregated floating point operations per second (FLOPS). Nevertheless, with recent successes of deep learning [4] using graphics processing units (GPUs), there comes a new venue for expediting other data mining algorithms [2, 31]. GPU has superior compute power and memory bandwidth compared to CPU [10]. Moreover, GPUs on one server can leverage interconnections such as NVLink [25] (40 GB/s per link with four links per GPU) which is much faster than any existing network. Therefore, we consider to solve the problem of MF using the alternating least square (ALS) method on GPUs.

In this paper, we propose a novel approach in solving MF on GPUs, termed cuMFals , with major contributions in two-fold:

- To fully utilize the GPU memory hierarchy, we identify hotspot variables to retain data as close to compute as possible. Afterward, the memory loading process is accelerated through an innovative

[1]https://ibm.biz/BdstmU

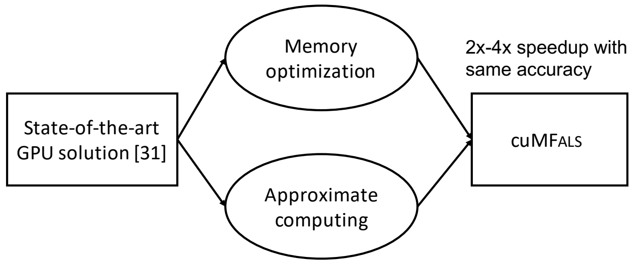
Memory optimization
2x-4x speedup with same accuracy
State-of-the-art GPU solution [31]
cuMFALS
Approximate computing

Figure 1: We optimize the state-of-the-art GPU implementation via two directions: memory optimization and approximate computing. Combining the two, we achieve 2x - 4x speedup with the same accuracy.

and non-conventional scheme by exploiting GPU architectural features such as occupancy and cache.

- We develop an iterative conjugate gradient (CG) solver on GPUs. This approximate solver reduced the compute complexity from $O(f^3)$ to $O(f^2)$ (f is the dimension of latent features) without hurting convergence. This optimization brings a speedup of **4x** compared to calculating in exact. Moreover, CG works naturally with Nvidia's newly developed half precision feature, which further doubles the speed.

A graphical illustration of our proposed approach is shown in Figure 1. By jointly optimizing the memory loading scheme and the approximate compute strategy, we are able to not only outperform all distributed CPU solutions by a large margin, but also make **2x-4x** improvements over the state-of-the-art GPU implementation. Such performance gains have been validated through extensive experiments on various GPU architectures. Our implementation is open-sourced on GitHub[2], available as a library to accelerate many applications.

2 PRELIMINARIES

Matrix factorization (MF) factorizes a matrix $R \in \mathbb{R}^{m\times n}$ (with N_z non-zero elements) into two low-rank matrices $X \in \mathbb{R}^{m\times f}$ and $\Theta \in \mathbb{R}^{n\times f}$, such that $R \approx X \cdot \Theta^T$. For any u and v, such tat $1 \le u \le m$ and $1 \le v \le n$, r_{uv} is the (i,j) entry of R. Thus, $r_{uv} \approx \boldsymbol{x}_u^T \cdot \boldsymbol{\theta}_v$, where $\boldsymbol{x}_u, \boldsymbol{\theta}_v \in \mathbb{R}^f$ are the u^{th} column of X^T and the v^{th} column of Θ^T, respectively. Then the optimization problem of MF is given as:

$$\min_{X,\Theta} \sum_{r_{uv}\neq 0} (r_{uv} - \boldsymbol{x}_u^T \boldsymbol{\theta}_v)^2 + \lambda(\sum_u n_{x_u} ||\boldsymbol{x}_u||^2 + \sum_v n_{\theta_v} ||\boldsymbol{\theta}_v||^2), \quad (1)$$

where n_{x_u} and n_{θ_v} are the number of non-zero elements of $\boldsymbol{x}_u$ and $\boldsymbol{\theta}_v$, respectively; λ is the regularization parameter. Two important approaches ALS and SGD both minimize equation (1), yet using different approaches that we will discuss the next.

ALS: ALS is an iterative method that first optimizes X while fixing Θ, and then solves Θ while fixing X. In every iteration, all observations ($r_{uv} \neq 0$) are used to update the current variable. Moreover, both subproblems are convex and the update procedures for them are given below.

Update X: The optimal solution of the u^{th} column of X^T is obtained by solving the following linear system:

$$\sum_{r_{uv}\neq 0} (\boldsymbol{\theta}_v \boldsymbol{\theta}_v^T + \lambda I) \cdot \boldsymbol{x}_u = \Theta^T \cdot R_{u*}^T. \quad (2)$$

Update Θ: Similarly, the optimal solution of v^{th} column of Θ^T is obtained by solving:

$$\sum_{r_{uv}\neq 0} (\boldsymbol{x}_u \boldsymbol{x}_u^T + \lambda I) \cdot \boldsymbol{\theta}_v = X^T \cdot R_{*v}. \quad (3)$$

Here, R_{u*} and R_{*v} are the u^{th} row and v^{th} column of R, respectively. It is worth mentioning that the updates of each $\boldsymbol{x}_u$ and $\boldsymbol{\theta}_v$ are independent. In other words, every row of matrix X can be updated in parallel while keeping Θ fixed. The same procedure is applicable to update Θ as well. To ease repeating, throughout the rest of this paper, we will only focus on solving X.

The solution of equation (2) has a closed form as

$$\boldsymbol{x}_u = (\sum_{r_{uv}\neq 0} \boldsymbol{\theta}_v \boldsymbol{\theta}_v^T + \lambda I)^{-1} \cdot \Theta^T \cdot R_{u*}^T, \quad (4)$$

which involves calculating a matrix inverse. Nevertheless, matrix inverse is compute intensive and unnecessary in solving the linear system in (2). Instead, many literatures [18, 29] solve the problem in a two-step fashion:

(i) compute intermediate results of $A_u = \sum_{r_{uv}\neq 0} (\boldsymbol{\theta}_v \boldsymbol{\theta}_v^T + \lambda I)$ and $b_u = \Theta^T \cdot R_{u*}^T$, which are called `get_hermitian` and `get_bias`, respectively;

(ii) solve the linear system, which will be referred as `solve`.

Our method also follows the two-step solving scheme. However, for each step, we propose a novel technique to better utilize both compute and memory resources of GPUs. Comparing `get_hermitian` and `get_bias`, we note that the compute complexity is dominated by the former one. Thus, we firstly focus on the optimization of `get_hermitian`, not `get_bias` in ALS in this paper.

SGD: SGD is also an iterative algorithm. However, differ to ALS, under each iteration, SGD only work with a small subset of observations denoted as Ω^k (*a.k.a.* mini-batch), where k is the number of iterations. Usually, samples in Ω^k are randomly selected from all observations. Then, the updating equations for both X and Θ for the k^{th} iteration are given as:

$$\begin{aligned} \boldsymbol{x}_u^k &= \boldsymbol{x}_u - \alpha^k \sum_{v:r_{u,v}\in\Omega^k} (\boldsymbol{x}_u^T \boldsymbol{\theta}_v - r_{uv})\boldsymbol{\theta}_v + \lambda \boldsymbol{x}_u, \text{ and} \\ \boldsymbol{\theta}_v^k &= \boldsymbol{\theta}_v - \alpha^k \sum_{u:r_{u,v}\in\Omega^k} (\boldsymbol{x}_u^T \boldsymbol{\theta}_v - r_{uv})\boldsymbol{x}_u + \lambda \boldsymbol{\theta}_v, \end{aligned} \quad (5)$$

where α^k is known as the learning rate. The vanilla SGD algorithm requires passing over randomly sampled data multiple times (till converge). When multiple updates run in parallel, for example, two samples r_{uv} and $r_{uv'}$ are updating at the same time, their updates to $\boldsymbol{x}_u$ may overwrite each other. To address this issue, previous studies either partition R into blocks with no overlapping rows and columns [9, 32, 37, 39], or let multiple workers independently update ignoring conflicts [22].

[2]https://github.com/cuMF/cumf_als

Table 1: Compute and memory complexity per epoch: ALS vs. SGD. ALS is compute intensive and SGD is memory intensive, so they need different optimizations on GPUs.

		Compute (C)	Memory (M)	C/M
ALS	get_hermitian	$O(N_z f^2)$	$O(N_z f + (m+n)f^2)$	f
	solve	$O((m+n)f^3)$	$O((m+n)f^2)$	f
SGD		$O(N_z f)$	$O(N_z f)$	1

Complexity: Following the roofline model [34], we calculate the computation and memory complexity for both ALS and SGD and summarize them in Table 1. Comparatively, ALS has a higher compute-to-memory ratio than SGD, which means ALS is compute intensive while SGD is memory intensive. Although the compute complexity of ALS is heavier than SGD per iteration, the number of iterations to converge is significant fewer [15]. Moreover, parallelization of ALS is easier since no sophisticated locking scheme is needed [15, 35]. At last, ALS is more suitable for the case of MF with implicit inputs, which makes it more broadly applicable.

Based on this complexity analysis, we focus our study in accelerating ALS in this paper, while SGD would be used as comparing topic.

Approximate computing: This term applied to computation that returns approximated result as the trade-off between accurracy and cost/performance. [7] explores some of the applications and hardware designs for approximate computing. Their work showed acceleration gain of 1.9X to 2.1X with only 2.5% quality loss. [19] provides a detailed survey on both software techniques and hardware design for a large variety of applications to leverage the power of approximate computing for cost or performance gain. We exploit approximate computing in two aspects. Firstly, our iterative process in linear equation solver would stop within some tolerable converaging values. Secondly, we use reduced precison hardware feature to maximize the utilization of memory bandwidth and memory capacity.

3 MEMORY OPTIMIZATION FOR HIGH FLOPS

As mentioned in section 2, an ALS update includes two steps, *i.e.* `get_hermitian` and `solve`. This section describes the memory optimization on `get_hermitian` and the next section introduces the approximate computing techniques on `solve`. As seen from Table 1, `get_hermitian` has a compute complexity of $O(N_z f^2)$. This is big in large-scale problems where N_z can be tens of billions, which leads to our first observation.

Observation 1. `get_hermitian` is compute intensive.

For a compute intensive function to achieve high FLOPS, it needs to retain data as close as possible to compute units [10, 34]. In other words, it needs effective caching to reduce read from external memory, *a.k.a.* DRAM, as DRAM cannot sustain high FLOPS of GPUs. Specifically to cuMFals , we need to identify the frequently-used variables, exploit the GPU memory hierarchy and place hotter variables in faster memory. This leads to the following solution.

Solution 1. Utiilize register and shared memory.

To decide what to cache and to where, we analyze the memory usage in calculating A_u:

- A_u is read and written once when adding each $\theta_v \theta_v^T$. Therefore, A_u is read and written by n_{x_u} times, that is N_z/m on the average.
- Each θ_v needs to be read f times when calculating $\theta_v \theta_v^T$.

We can now allocate variables into different places in GPU memory hierarchy based on their reuse. Usually $N_z/m \gg f$, and as a result A_u is more frequently accessed than θ_v. Therefore, A_u deserves the fastest cache, *i.e.* register, and θ_v is put into shared memory, the second fastest cache. Figure 2 illustrates the memory optimization to `get_hermitian`. For a given x_u, its required features, *i.e.* θ_vs such that $r_{uv} \neq 0$, are staged from Θ^T in global memory (the matrix at the top) into a shared memory space of size $BIN \times f$ (the thinner matrix in the middle), in batches. For each staged feature θ_v, we calculate $\theta_v \theta_v^T$ in tiles of size T and add to the corresponding sub-blocks of A_u in registers (the symmetric matrix at the bottom). Each sub-block in A_u aggregates the outer product of two tiles in θ_v. Consider the symmetricity of A_u, we only need to calculate the bottom half of it. A_u stored in registers is flushed to global memory when all required $\theta_v \theta_v^T$s are added in.

Because we choose to excessively use registers, they become the constrained resources. Consequently, the occupancy of `get_hermitian`, *i.e* number of A_us that can be calculated concurrently is low.

Observation 2. Aggressive use of registers leads to low occupancy, which makes read from global memory latency-bound instead of bandwidth-bound.

Current Nvidia GPUs have 65536 float registers in each stream-multiprocessor (SM). When $f = 100$, each thread of `get_hermitian` needs 168 registers and each block needs 64 threads. As a result, an SM can hold $65536/(168 \times 64) \approx 6$ thread-blocks, i.e. an SM can update 6 rows concurrently. Compared with the SM capacity to hold 32 thread-blocks, this is a low occupancy. This indicates that there are relatively few concurrent threads loading from global memory, which leads to the next solution.

Solution 2. Use non-coalesced and cache-assisted read, which is non-conventional but proven faster.

In GPU programming, memory coalescing is considered as a best practice to achieve good performance [14]. Memory coalescing means that adjacent threads should access adjacent global memory addresses. It can consolidate memory load requests and avoid wasting the bandwidth. When using coalescing, read from global memory can bypass L1 cache, because loaded data are all used.

Given the low occupancy of `get_hermitian`, coalesced read, despite its efficiency, cannot saturate the memory bandwidth. On the other hand, when the occupancy is low, the working data can almost fit into L1 cache. For example, when $f = 100$ and $BIN = 32$, the θ_vs being actively loaded per SM is $100 \times 32 \times 6$ (thread-block)$\times 4$ (bytes per float)= 75 KB. This number is between Nvidia Maxwell's

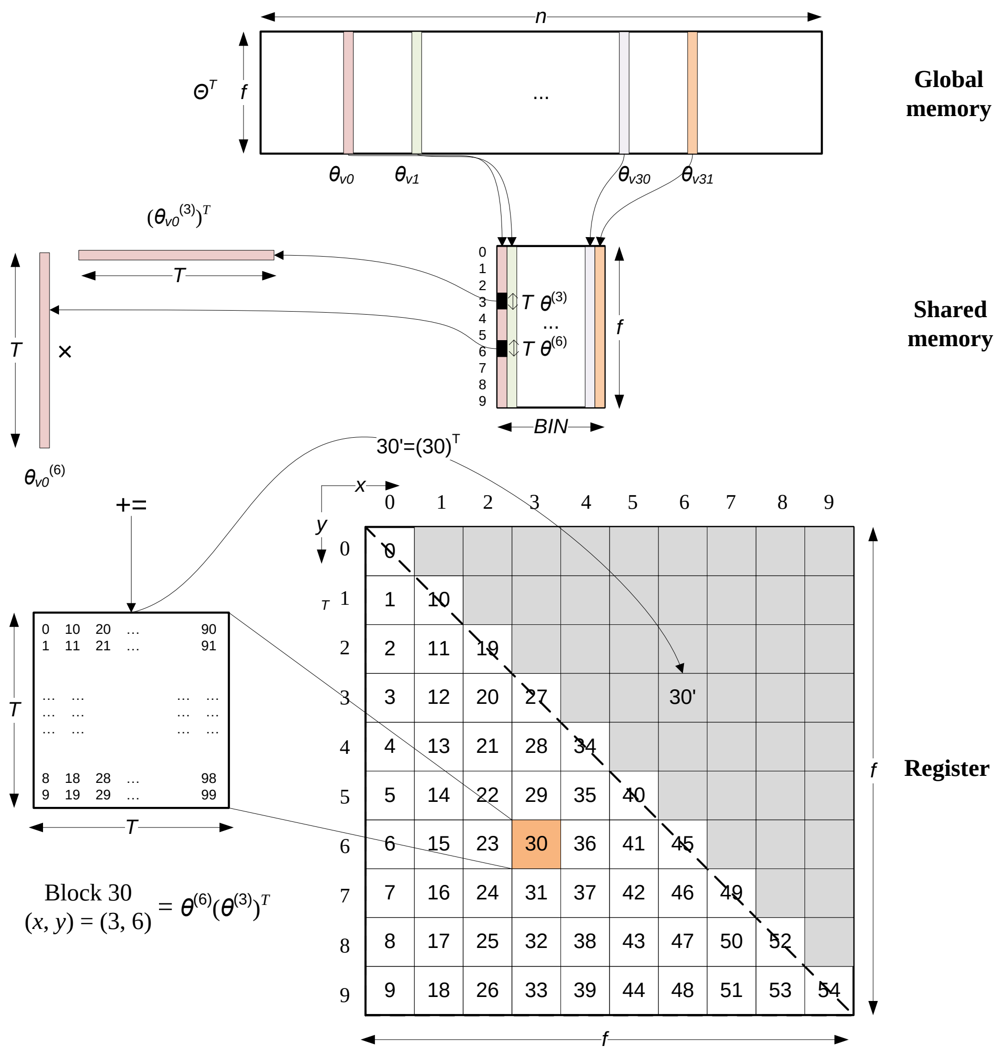

Figure 2: The memory optimization to `get_hermitian`. For a given x_u, its required θ_vs such that $r_{uv} \neq 0$, are staged from Θ^T from global memory to a shared memory buffer of size $BIN * f$, in batches. For each θ_v in shared memory, we calculate $\theta_v \theta_v^T$ in tiles of size T, and add to sub-blocks of A_u in registers. Each sub-block in A_u adds the outer product of two tiles in θ_v. Consider the symmetricity, we only calculate tiles with coordinates of $x \leq y$. For example, for θ_{v0}, tiles 3 and 6 need to do outer product and add to sub-block 30 in A_u, i.e., $block_{30} \mathrel{+}= \theta_{v0}^{(6)} (\theta_{v0}^{(3)})^T$, and $block_{30'} = (block_{30})^T$. A_u in registers is flushed to global memory after all required $\theta_v \theta_v^T$s are added into it.

L1 cache of 48 KB and L2 cache of 128 KB (3 MB shared by 24 SMs). Inspired by this observation, we use a parallel but non-coalesced read scheme as illustrated in Figure 3 (b). Without losing generality, we load 32 features $\boldsymbol{\theta}_{v0}, \boldsymbol{\theta}_{v1}, \boldsymbol{\theta}_{v30}, \ldots, \boldsymbol{\theta}_{v31}$ using 32 threads. With the coalesced scheme in Figure 3 (a), 32 threads together read one $\boldsymbol{\theta}_v$ column before moving to the next one. Alternatively, in the non-coalesced scheme in Figure 3 (b), 32 threads read 32 columns concurrently, with each thread reading one column. Because of the small working data set size, L1 and L2 cache can efficiently serve as the coalescing cache. That is, the non-coalesced load requests issued from $t_0, t_1, \ldots, t_{30}, t_{31}$ are going to hit L1 and L2, which makes it even more efficient than coalesced read.

To showcase the effectiveness of solution 2, we measure the performance of coalesced and non-coalesced read in `get_hermitian`. We use the Netflix dataset (see Section 5.1 for more details on datasets) and measure the time of three phases in `get_hermitian`: load from global memory to shared memory (**load**), compute A_u (**compute**), and write A_u to global memory (**write**). Figure 4 shows the performance of both update-X and update-Θ procedures, in three different settings: `coal` means coalesced read, the setting illustrated in Figure 3 (a); `nonCoal-L1` means non-coalesced read,

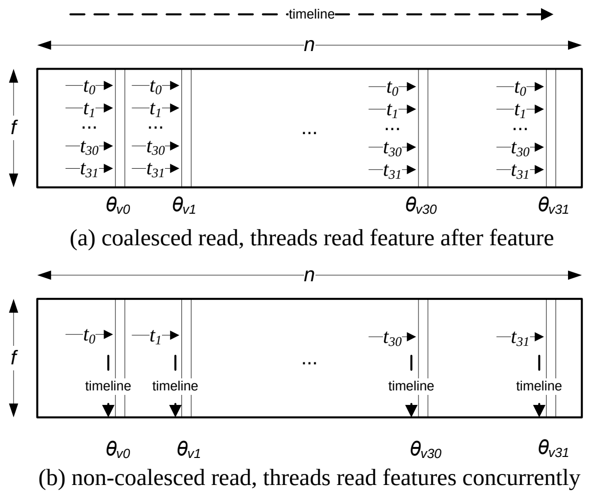
timeline
n
f
t_0 t_1 … t_{30} t_{31}
θ_{v0} θ_{v1} θ_{v30} θ_{v31}
(a) coalesced read, threads read feature after feature
timeline
(b) non-coalesced read, threads read features concurrently

Figure 3: Load from global to shared memory in get_hermitian. (a) coalesced read: all threads read one column before moving to the next. This issues fewer memory instructions but is lack of parallelism. (b) Non-coalesced read: multiple threads read multiple columns concurrently. In low occupancy, the columns are cached and subsequent non-coalesced reads will hit cache.

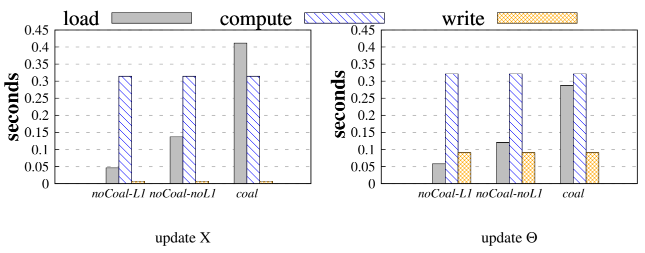

Figure 4: The performance of coalesced and non-coalesced read from global to shared memory in get_hermitian, using the Netflix dataset. Bar load shows the memory load time; non-coalesced read with L1 cache (nonCoal-L1) is the fastest.

the setting in Figure 3 (b) and with L1 cache; nonCoal-noL1 means non-coalesced read with L1 cache bypassed. Result shows:

- For shared memory load, non-coalesced with L1 performs best; non-coalesced without L1 is worse and coalesced read is worst.
- The compute time is almost constant in all settings. This is because they all need $N_z \times f^2$ fused multiple-add (FMA) operations.
- Update-X and update-Θ need to write $m \times f^2$ and $n \times f^2$ floats to global memory, respectively. Since $m < n$ in Netflix data, update-Θ takes longer in write.

4 APPROXIMATE COMPUTING IN SOLVER

Section 3 describes how to efficiently obtain A_u. After that, we need to solve m equations $A_u \boldsymbol{x}_u = \boldsymbol{b}_u$ as illustrated in equation (2).

Algorithm 1 The CG solver for $A\boldsymbol{x} = \boldsymbol{b}$.

```
1: procedure CGSolve($A, \boldsymbol{x}, \boldsymbol{b}, f_s, \epsilon$)
2:     $\boldsymbol{r} = \boldsymbol{b} - A \cdot \boldsymbol{x}$;   $\boldsymbol{p} = \boldsymbol{r}$;   $rs_{old} = \boldsymbol{r}^T \cdot \boldsymbol{r}$
3:     for $j = 1 : f_s$ do
4:         $\boldsymbol{a_p} = A \cdot \boldsymbol{p}$;   $\alpha = rs_{old}/(\boldsymbol{p}^T \cdot \boldsymbol{a_p})$
5:         $\boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p}$;   $\boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{p}$
6:         $rs_{new} = \boldsymbol{r}^T \cdot \boldsymbol{r}$
7:         if $\sqrt{rs_{new}} < \epsilon$ then
8:             break
9:         end if
10:        $\boldsymbol{p} = \boldsymbol{r} + (rs_{new}/rs_{old})\boldsymbol{p}$
11:        $rs_{old} = rs_{new}$
12:    end for
13:    return $\boldsymbol{x}$
14: end procedure
```

4.1 Approximate solver with CG

The direct solver, *e.g.*, the batch LU solver in cuBLAS [23], gives an exact solution to $A\boldsymbol{x} = \boldsymbol{b}$ with compute complexity of $O(f^3)$, or $O(m \times f^3)$ for m rows. This cubic complexity leads to long solve time, especially when m is big. As shown in Table 1, when m becomes big, R's rows become sparse, and $m \times f^3$ gets closer to $N_z \times f^2$. To demonstrate this, we measure the solver time of 10 ALS iterations on Netflix data. Column LU_FP32 in Figure 5 shows that, the time taken by the LU solver is almost twice as much as that by get_hermitian. This clearly indicates that after applying optimization on get_hermitian, solve executing time now becomes dominant. This leads to the following observation.

Observation 3. Solve is compute intensive and dominant.

This observation inspires us to seek an alternative to the direct solver. We notice that, as an iterative approach, ALS updates X and Θ based on estimations from the previous iteration. As errors exist in estimations, the solution of each step is inherently inaccurate. Therefore, solution accuracy may be sacrificed in exchange for compute speed, leading to our attempt for an approximate solver.

Solution 3. An approximate conjugate gradient solver.

The iterative CG solver is introduced in [11]. With f iterations each of complexity $O(f^2)$, it yields the exact solution with complexity $O(f^3)$. Based on this, we seek to further reduce computation while maintaining convergence quality. The pseudo code of our approximate CG solver is summarized in Algorithm 1, where f_s is given to control the number of iterations (see Line 3), and ϵ for tolerance control. Empirically this approximation does not impact ALS's convergence, while effectively reducing the solver's complexity from $O(f^3)$ to $O(f^2)$ when $f_s \ll f$.

4.2 Use reduced precision

Replacing LU solver with an approximate CG solver, the solver's compute-to-memory ratio (recall Table 1) now drops from $O(f)$

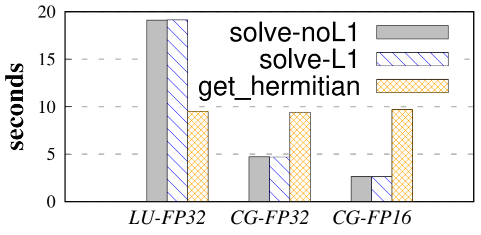

Figure 5: The solver time of 10 ALS iterations using Netflix data on Nvidia Maxwell Titan X. $f = 100$ and we use $f_s = 6$ (the smallest number that does not hurt convergence) for CG. CG-FP32 is $1/4$ of the LU-FP32 time; CG-FP16 takes $1/2$ of the CG-FP32 time. Using L1 (`solve-L1`) takes the same time as without it (`solve-noL1`).

to $O(1)$, converting the original compute intensive problem into a memory intensive one.

Observation 4. CG solver is memory intensive.

As seen in Algorithm 1, CG solver is dominated by dense matrix-vector multiply $A \cdot \boldsymbol{p}$ (Line 4), which is in turn dominated by reading A that is of memory complexity $O(f^2)$. This insight inspires us that further acceleration is possible by reducing the size of A.

Solution 4. Use reduced precision in CG solver to double the effective memory bandwidth.

We choose the newly introduced 16-bit floating point format (FP16, compared with the default 32-bit floating point format FP32) in Nvidia GPUs to store A. This optimization saves 50% memory bandwidth to load A and consequently doubles the loading speed. To validate, we run 10 iterations of ALS using Netflix data on a Maxwell GPU. Figure 5 shows the total solver time. The time of CG solver with FP32 (`CG-FP32`) is only 1/4 of that of LU solver with FP 32 (`LU-FP32`). When CG uses FP16, `CG-FP16` takes 1/2 of the time compared with `CG-FP32`. In total, CG-FP16 can reduce the run-time to 1/8 compared with LU-FP32.

Does L1 cache benefit the CG solver?

Figure 5 also illustrates that, loading A_u with L1 cache does not yield any performance benefit. This is coherent with the analysis in section 3: L1 cache is only useful to coalesce the non-coalesced memory access when occupancy is low. With batch CG's high occupancy and coalesced read, L1 cache is not useful at all. This also explains why L1 cache is disabled by default in Nvidia GPUs.

5 EXPERIMENTS

In this section, we show the advantages of the proposed cuMFals framework compared to a set of state-of-the-art implementations for both CPU and GPU. Our experiments are designed to answer the following questions:

- How fast cuMFals is compared to competing implementations?
- How efficiently cuMFals utilizes compute resource (in terms of FLOPS) and memory bandwidth of GPU, as argued in sections 3 and 4?
- How does cuMFals compare to SGD?
- Can cuMFals extent to the setting of MF with implicit feedback (*a.k.a* one-class or positive-unlabeled inputs)?

Table 2: Benchmark datasets and parameters.

Dataset	m	n	N_z	f	λ	*RSME*
Netflix	480,189	17,770	99M	100	0.05	0.92
YahooMusic	1,000,990	624,961	252.8M	100	1.4	22
Hugewiki	50,082,603	39,780	3.1B	100	0.05	0.52

5.1 Datasets

We utilize three publicly available datasets as follows:

- ***Netflix*** [38]: The Netflix dataset consists ratings on movies. Each rating is in the scale of one to five.
- ***YahooMusic*** [6]: Similar to the Netflix, this dataset contains 250 million ratings in the range of 1 to 100 for music collected by the Yahoo! Music Radio service.
- ***Hugewiki*** [39]: Hugewiki contains a snapshot of Wikipedia. The observation matrix R describes the frequency of English terms appeared in different documents.

The detailed statistics are summarized in Table 2. We choose 0.92, 22 and 0.52 as the convergence value for Netflix, YahooMusic and Hugewiki, respectively, because these values are used by many papers and considered well-accepted.

5.2 Experiment setting

For the purpose of comprehensive evaluations, experiments are conducted on three different generations of Nvidia GPUs: Kepler, Maxwell and Pascal. Table 3 illustrates the configurations of the three servers we use. CPU-only experiments are conducted on the most powerful Pascal server unless otherwise mentioned.

For quantitative comparison, we follow the standard experiment setting [37, 39] by reporting how fast the root mean square error (RMSE) on the testing test reduces. The stopping criteria for all algorithms is when the RSME on testing set reaches an "acceptable level". Specifically, the acceptable RSME is 0.92, 22.0 and 0.52 for Netflix, YahooMusic and Hugewiki, respectively. Furthermore, we make use of the original training and testing files from the providers of Netflix and YahooMusic datasets while randomly extract 10% of the data as the testing set for Hugewiki. It is worth mentioning that, we focus on system-level efficiency in terms of running time instead of the recommendation accuracy. To achieve the goal, we use the same set of parameters (f and λ) as reported from earlier works [31, 37, 39], which is also shown in Table 2.

5.3 Convergence speed: is cuMFals fast?

There are many studies and systems on accelerating MF [3, 9, 16, 20, 22, 27, 30, 35–37, 39]. Among them, we compare with the representative works below because **they are with state-of-art performance (i.e., convergence speed) or scalability**.

- **LIBMF** [3, 39]: The state-of-the-art CPU-based multi-thread solution using a single machine.
- **NOMAD** [37]: NOMAD is a CPU-based solution using SGD. Different from LIBMF, it runs on multiple machines using message passing interface (MPI) to communicate.

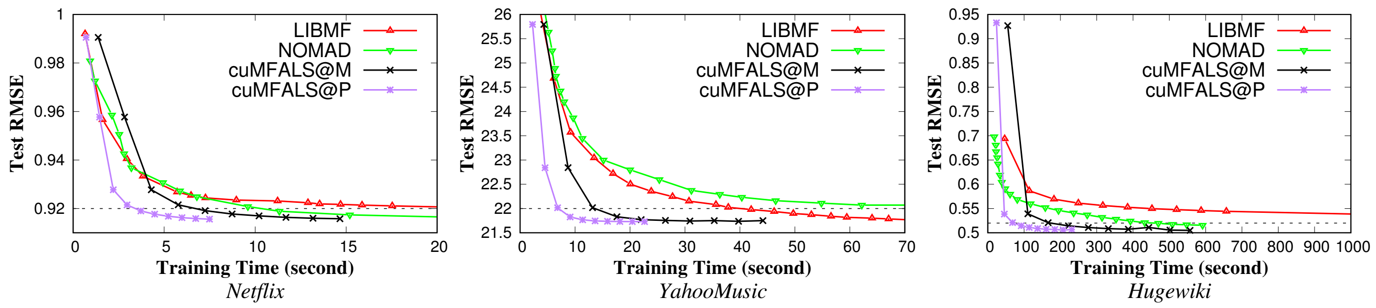

Figure 6: cuMFals vs. CPU solutions w.r.t. convergence time. LIBMF uses 40 cores on one machine; NOMAD uses 32 machines for Netflix and YahooMusic, and 64 machines for Hugewiki. cuMFals uses one GPU for Netflix and YahooMusic, and four GPUs for Hugewiki. cuMFals on Maxwell (@M) and Pascal (@P) converges significantly faster than all other approaches.

Table 3: Config of Kepler, Maxwell and Pascal servers.

Kepler	
CPU	Two 8-core Intel Xeon E5-2667, 256 GB RAM
GPU	Two Kepler K40, each: 4 TFLOPS, 12 GB RAM, 288 GB/s
Maxwell	
CPU	Two 12-core Intel Xeon E5-2670, 512 GB RAM
GPU	Four Titan X, each: 7 TFLOPS, 12 GB RAM, 340 GB/s
Pascal	
CPU	Two 10-core IBM Power8 with SMT 8, 512 GB RAM
GPU	Four Tesla P100, each: 11 TFLOPS, 16 GB, 740 GB/s

- **BIDMach** [2]: BIDMach is a single GPU library that contains a set of matrix functions on top of which machine learning algorithms can be built. It also implements ALS based on a general purpose sparse matrix function.
- **HPC-ALS** [8]: It implements ALS on single GPU by exploiting registers and shared memory. However, it has no non-coalesced read, approximate solver or reduced precision.
- **GPU-ALS** [31]: The state-of-art ALS implementation on GPUs but without our memory optimization and approximation presented in Section3 and Section4, respectively.
- **GPU-SGD** [35]: Section 2 discussed the difference between ALS and SGD solvers for MF. We also compared with a CUDA-based SGD solution that solves MF problems with one or multiple GPUs, using matrix blocking and Hogwild!-style algorithms [22] to parallelize the SGD updates. For individual SGD updates, it leverages GPU architectural features such as cache, warp-shuffle instructions, and half-precision floats. In Section 5.5 we will discuss and compare these two methods in detail.

It is worth mentioning that, the performance of CPU-based algorithms not necessarily improves as the number of threads/machines increases due to two reasons: 1) synchronization on shared data structures and 2) communication overhead. Enlarging the number of compute resource may even hurt their performance [8]. Therefore, we use 40 threads for LIBMF, which achieves the best performance. For NOMAD, we use the best settings as reported in [37], which are 32 machines for Netflix and Yahoo, and 64 machines for Hugewiki. Moreover, BIDMach and HPC-ALS can only use one GPU, while GPU-ALS and cuMFals can adopt multiple GPU settings. We test all GPU-based algorithms using one GPU on both Netflix and YahooMusic. Furthermore, to show how well both GPU-ALS and our framework scale with the number of GPUs, we use four GPUs for both algorithms on the Hugewiki dataset.

Figure 6 shows the relation between test RMSE and training time while table 4 summarizes the time when RMSE reaches an acceptable level. Clearly, cuMFals outperforms all CPU solutions with a large margin. Specifically, on both Netflix and YahooMusic datasets, cuMFals with single Pascal GPU (cuMFals@P) achieves 5.6x-7x performance gain compared to LIBMF. As for Hugewiki, cuMFals with four Pascal GPUs only takes 68 seconds to converge, which is significantly faster compared to 459 seconds for NOMAD (6.7x) and 3021 seconds for LIBMF (44.4x). The reason BIDMach is not included in the table is that it does not converge to the acceptance level. Regardless the convergence, we can observe the ALS kernel of BIDMach runs at 40 GFLOPS, which is similar to the reported measurement in their original paper [2]. However, 40 GFLOPS is much lower than cuMFals (see section 5.4 for our performance in terms of FLOPS). On the other hand, since HPC-ALS is not open-source, we only compare our performance of per iteration time on Netflix, which has been reported in their paper. Results show that cuMFals runs twice as fast as HPC-ALS on the same hardware (Kepler K40). Furthermore, compared with GPU-ALS, cuMFals has a significant performance advantage thanks to our memory optimization and approximate computing techniques. On Netflix with Maxwell GPU, cuMFals only needs 6.5 seconds to converge while GPU-ALS needs 28 seconds, *i.e.* a 4x speedup. As a summary, cuMFals also outperforms all state-of-art GPU solutions.

5.4 Has cuMFals fully exploited GPU?

In this section, we validate whether the proposed cuMFals framework has fully exploited the potential of GPU hardware. The analytics are done by examining if the compute-intensive kernel `get_hermitian` has achieved high FLOPS, and if the memory-intensive CG solver has achieved high memory bandwidth.

Has `get_hermitian` achieved high FLOPS?

To obtain `get_hermitian`: $A_u = \sum_{r_{uv}\neq 0} (\theta_v \theta_v^T + \lambda I)$ for $1 \leq u \leq m$, one needs to read the sparse matrix R and perform m matrix

Table 4: Training time in seconds when converging to acceptable RMSE. @M: Maxwell GPU, @P: Pascal GPU.

Studies	*Netflix*	*YahooMusic*	*Hugewiki*
LIBMF [3]	23	38	3021
NOMAD [37]	9.6	109	459
GPU-ALS@M [31]	28	42	400
cuMFals@M	6.5	13.2	166
cuMFals@P	3.3	6.8	68
cuMFals@P /LIBMF	7x	5.6x	44.4x

multiplications. The of size of each multiplication is $\mathbb{R}^{f \times n_{x_u}} \times \mathbb{R}^{n_{x_u} \times f}$, where $n_{x_1} + n_{x_2} + \ldots + n_{x_m} = N_z$. To our best knowledge, no existing GPU library including cuBLAS has implemented the `get_hermitian` function for us to compare. The closest baseline is the batched-matrix-multiplication `gemmBatched` [24] in cuBLAS, which calculates m matrix multiplications of the same dimension: $\mathbb{R}^{a \times b} \times \mathbb{R}^{b \times c}$. Although `gemmBatched` cannot parallelize matrix multiplications with different sizes, to compare, we set the dimension of each computation in our `get_hermitian` to be the same. Under this setting, two algorithms can be fairly compared and we measure the FLOPS achieved by both with Kepler, Maxwell and Pascal on Netflix.

Experimental results, as illustrated in Figure 7(a), shows that cuMFals achieves higher FLOPS in all three generations of GPUs. This is impressive because `get_hermitian` compared to `gemmBatched` in cuBLAS needs to perform extra work. Specifically, `get_hermitian` needs to read sparse R to get references to Θ, and batch-multiply matrices with variable sizes. However, cuBLAS only needs to read dense input and batch-multiply matrices with the same size, which has no time cost on finding references. Moreover, regarding FLOPS efficiency (*i.e.*, the achieved-FLOPS divided by the device's peak-FLOPS) Figure 7(a) indicates that cuMFals achieves better performance in Nvidia newly developed architectures. This can be explained by the fact that performance of `get_hermitian` is generally limited by the number of registers. Comparing Kepler, Maxwell and Pascal, the number of registers per core increases as technology evolves. At the same time, it reveals that our design discipline matches the development trend of GPU.

Has the CG solver achieved high memory bandwidth?

We measure the memory transfer rate (GB/s) between GPU SMs and its DRAM, and compare it to the bandwidth achieved by CUDA function `cudaMemcpy`. Since `cudaMemcpy` only copies memory and deals with no computation, the comparison with it can indicate how well the CG solver can saturate the device memory bandwidth. As seen from Figure 7(b), cuMFals achieves a higher bandwidth than `cudaMemcpy` on all three types of GPUs. This demonstrates that our proposed CG solver utilizes the memory bandwidth efficiently.

5.5 ALS vs. SGD on GPUs

As discussed in early Section 2, ALS and SGD have their own attributes in solving the problem of MF. SGD runs faster per iteration but requires more iterations. When the rating matrix gets denser, ALS has more advantage because SGD's complexity grows [15] and also becomes harder to parallelize [35].

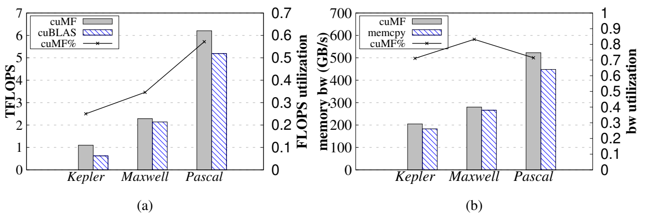

Figure 7: (a) The FLOPS and efficiency of `get_hermitian` on GPUs of three generations. cuMFals achieves higher FLOPS than the batch cuBLAS routine for fixed size and higher FLOPS efficiency on newer GPUs. (b) The memory bandwidth achieved by CG solver is shown to be higher than the bandwidth of `cudaMemcpy`.

We implemented both ALS and SGD[3] and compare their performance on GPUs. We follow the same setting as before, and report the results in Figure 8. Come as no surprise, ALS runs slower in each iteration, but requires fewer iterations to coverage. On one GPU, ALS converges slightly faster than SGD on Netflix, but slightly slower than SGD on YahooMusic and Hugewiki. However, with four GPUs (als@4), ALS converges faster than SGD on Hugewiki data.

Moreover, as seen in Table 1, SGD's computation complexity ($O(N_z f)$) grows linearly with N_z. This makes it inefficient when the rating matrix R becomes more dense. This issue becomes severe when dealing with implicit inputs, where the rating matrix is considered fully dense, i.e., $N_z = m * n$ [12]. ALS can easily adapt to the setting of MF with implicit inputs, which we will discuss separately in section 5.6.

5.6 Implicit matrix factorization

MF with implicit inputs has been widely used in real-life applications [12, 33] where explicit ratings are replaced by implicit ones such as purchase or number of clicks. To show ALS is able to handle implicit inputs, we follow the same setting as shown in [12], by considering a binary matrix $P \in \mathbb{R}^{m \times n}$. $p_{uv} = 1$ if the implicit observations $r_{uv} > 0$, and $p_{uv} = 0$ otherwise. The original paper also adds confidence measures c_{uv} to predictions, which leads to the following cost function:

$$\min_{X,\Theta} \sum_{u,v} c_{uv}(p_{uv} - \boldsymbol{x}_u^T \boldsymbol{\theta}_v)^2,$$

where $c_{uv} = 1 + \alpha r_{uv}$ and α is a given scaling constant. In other words, any $r_{uv} = 0$ is no longer treated as a missing rating, but as a zero-rating with low confidence c_{uv}. Under this assumption [12], P is not sparse and therefore SGD will be costly. In such a way, SGD loses its competitiveness. Therefore, we compare cuMFals with two open-source libraries for implicit MF: *implicit*[4] and *QMF*[5]. Experiments demonstrate that cuMFals converges under

[3]https://github.com/cuMF/

[4]http://github.com/benfred/implicit

[5]http://github.com/quora/qmf

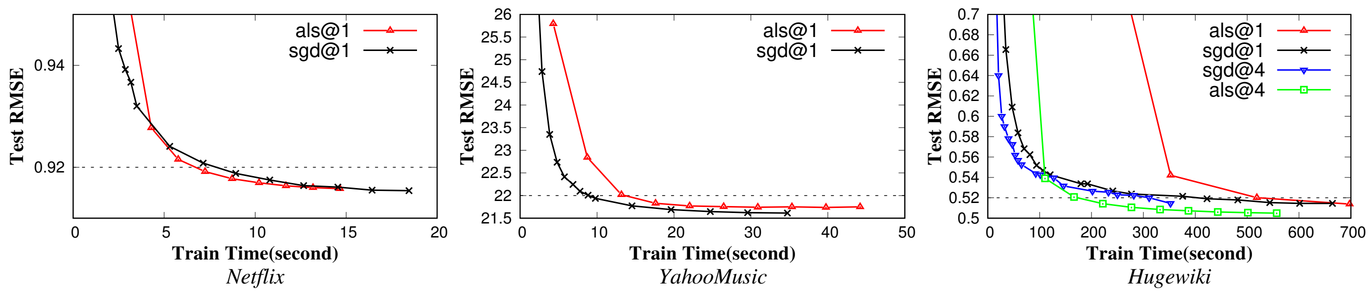

Figure 8: ALS vs. an SGD solution [35] on one (@1) and four (@4) GPUs.

the implicit setting and the per iteration time of cuMFals , *implicit* and *QMF* are 2.2, 90, and 360 seconds, respectively.

6 RELATED WORK

This section reviews related work on parallel matrix factorization with SGD, ALS and cyclic coordinate descent (CCD) algorithms. Table 5 is a summary and details are in the following subsections.

6.1 Parallel SGD

SGD is inherently serial where each time one sample is selected to update. To accelerate this process, two samples can update in parallel if they are neither in same row nor same column. This observation has led to two ways to parallel SGD for MF: lock-free Hogwild! [22] and blocking [9, 27, 36, 39]. *Hogwild!* observes that when R is very sparse and the number of parallel workers is much less than the dimension of R, they can independently update samples with a low probability of conflict. *Blocking* divides R into several sub-blocks, and sub-blocks that do not share rows or columns can update in parallel.

CPU approaches. SGD has been parallelized in multi-core [27, 39], multi-node MPI [32, 37], MapReduce [9] and parameter-server [5, 30] systems. These methods partition R into blocks with no overlapping rows or columns, and work on these blocks in parallel. They further optimize the algorithm with asynchronous communication, overlapping communication and computation, and shared memory. For example, LIBMF [39] is very efficient on multi-cores. However, it stops scaling when using few dozens cores [21, 35], because of the locking in a shared data structure. Moreover, LIBMF is a single-machine solution and therefore cannot deal with large-scale problems. NOMAD [37] extends the idea of block partitioning, and alleviate the issue of global locking. It performs similarly to LIBMF on a single machine and can scale to a 64-node HPC cluster. Parameter server [5] can be used to implement distributed SGD. For example, Petuum [5] can scale MF to hundreds of cores in a cluster, and Factorbird [30] is a parameter server specifically implemented for matrix factorization.

GPU approaches. Both Hogwild and blocking schemes are implemented in [35]. It has efficient kernels for SGD update, leveraging cache, warp-shuffle instructions, and half-precision.

6.2 Parallel ALS and CCD

CPU approaches for ALS. PALS [38] and SparkALS [18] parallelize ALS by feature full replication and partial replication, respectively. These approaches are not feasible when feature matrices get extremely large. Facebook [13] tackles this issue by partitioning the feature matrix and rotate its parts among multiple nodes. GraphLab [17] distributes the feature matrix among multiple machines. When updating in a machine, needed features are fetched on-demand from other machines.

GPU approaches for ALS. BIDMach [2] provides generic matrix kernels for many machine learning algorithms including MF. However, its sparse kernel is not specifically optimized for ALS and slower than cuMFals . HPC-ALS [8] optimizes the `get_hermitian` kernel similar to us. However, they used neither non-coalesced read, nor approximate solver nor reduced precision.

Parallel CCD. CCD++ [36] performs sequential updates on one row of the decomposed matrix while fixing other variables. CCD++ has lower time complexity but makes less progress per iteration, compared with ALS. [20] further accelerates CCD++ on GPUs using loop fusion and tiling. The resulting algorithm is shown to be faster than CCD++ on CPUs [36] as well as GPU-ALS [31] that is without memory optimization and approximate computing.

7 CONCLUSION

Due to the importance of MF in the field of data mining, in this paper, we accelerate ALS, one of the most important MF solving algorithms with GPUs. Specifically, we identify challenges that make ALS running slow such as constraints in the utilization of capacity and bandwidth of memory, and computation intensiveness. To alleviate these problems, we propose a novel framework named cuMFals. Our algorithm exploits the GPU memory architectures and shortens the time of reading data via an innovative scheme. At the same time, the proposed algorithm also eliminates unnecessary computing in solving MF without hurting convergence. We conduct extensive experiments under various settings. Our proposed method achieves the state-of-the-art performance, suggesting that cuMFals can significantly advance the task of MF. We also integrated cuMFals into Spark MLlib, accelerating its ALS algorithm[6].

In future work, we would like to further analyze different GPU-accelerated MF algorithms and investigate algorithm selection based on dataset characteristics such as dimensions and sparsity,

[6]https://github.com/IBMSparkGPU/CUDA-MLlib

Table 5: Parallel MF solutions using SGD, ALS and CCD, on CPUs and GPUs.

	CPU	GPU
SGD	**lock-free**: workers independently sample & update single-node: HogWild! [22]; multi-nodes: FactorBird [30], Petuum [5] **blocking**: workers pick non-overlapping blocks blockDim=#workers: DSGD [9] blockDim>#workers: LIBMF [39], NOMAD [37], DSGD++ [32] nested blocking: dcMF [21], MLGF-MF [27]	**single and multiple GPUs**: GPU-SGD – SGD with lock-free and blocking [35]
ALS	**replicate all features**: PALS [38], DALS [32] **partial replicate features**: SparkALS [18], GraphLab [17], Sparkler [16] **rotate features**: Facebook [13] **approximate ALS**: [29]	**single GPU**: BIDMach [2], HPC-ALS [8] **single and multiple GPUs**: GPU-ALS [31] and cuMFals
CCD	**multi-core and multi node**: CCD++ [36]	**single GPU**: parallel CCD++ [20]

and hardware resource constraints such as number of GPUs. We also plan to investigate a hybrid solution that combines SGD and ALS. A scenario could be using ALS for the initial batch training and SGD for incremental updates of the model. Last but not the least, we would like to exploit the new Nvidia Tensor Cores [26] hardware that natively supports half-precision arithmetic, to further speed up cuMFals .

REFERENCES

[1] Jimmy Ba and Rich Caruana. 2014. Do Deep Nets Really Need to be Deep?. In *NIPS*. 2654–2662. http://papers.nips.cc/paper/5484-do-deep-nets-really-need-to-be-deep.pdf

[2] John Canny, Huasha Zhao, Bobby Jaros, Ye Chen, and Jiangchang Mao. 2015. Machine learning at the limit. In *IEEE BigData, Big Data 2015*. 233–242.

[3] Wei-Sheng Chin, Yong Zhuang, Yu-Chin Juan, and Chih-Jen Lin. 2015. A learning-rate schedule for stochastic gradient methods to matrix factorization. In *PAKDD*. Springer.

[4] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. 2013. Deep learning with COTS HPC systems. In *ICML*. 1337–1345.

[5] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R. Ganger, Phillip B. Gibbons, Garth A. Gibson, and Eric P. Xing. 2014. Exploiting Bounded Staleness to Speed Up Big Data Analytics. In *USENIX ATC*. 37–48.

[6] Gideon Dror, Noam Koenigstein, Yehuda Koren, and Markus Weimer. 2012. The Yahoo! Music Dataset and KDD-Cup '11. In *KDD Cup 2011 competition*.

[7] Hadi Esmaeilzadeh, Adrian Sampson, Luis Ceze, and Doug Burger. 2012. Neural Acceleration for General-Purpose Approximate Programs. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. 449–460.

[8] Mark Gates, Hartwig Anzt, Jakub Kurzak, and Jack Dongarra. 2015. Accelerating collaborative filtering using concepts from high performance computing. In *IEEE BigData*. 667–676.

[9] Rainer Gemulla, Erik Nijkamp, Peter J. Haas, and Yannis Sismanis. 2011. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*. 69–77.

[10] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach.* Elsevier.

[11] Magnus Rudolph Hestenes and Eduard Stiefel. 1952. *Methods of conjugate gradients for solving linear systems.* Vol. 49. NBS.

[12] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *ICDM*. IEEE, 263–272.

[13] Maja Kabiljo and Aleksandar Ilic. 2015. Recommending items to more than a billion people. https://code.facebook.com/posts/861999383875667. (2015). [Online; accessed 17-Aug-2015].

[14] David B Kirk and W Hwu Wen-mei. 2012. *Programming massively parallel processors: a hands-on approach.* Morgan Kaufmann.

[15] Yehuda Koren, Robert M. Bell, and Chris Volinsky. 2009. Matrix Factorization Techniques for Recommender Systems. *Computer* 42, 8 (2009), 30–37.

[16] Boduo Li, Sandeep Tata, and Yannis Sismanis. 2013. Sparkler: Supporting Large-scale Matrix Factorization. In *EDBT*. 625–636.

[17] Yucheng Low, Danny Bickson, Joseph Gonzalez, Carlos Guestrin, Aapo Kyrola, and Joseph M Hellerstein. 2012. Distributed GraphLab: a framework for machine learning and data mining in the cloud. In *VLDB*. 716–727.

[18] Xiangrui Meng, Joseph K. Bradley, Burak Yavuz, Evan R. Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, D. B. Tsai, Manish Amde, Sean Owen, Doris Xin, Reynold Xin, Michael J. Franklin, Reza Zadeh, Matei Zaharia, and Ameet Talwalkar. 2016. MLlib: Machine Learning in Apache Spark. *Journal of Machine Learning Research* 17 (2016), 34:1–34:7. http://jmlr.org/papers/v17/15-237.html

[19] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* (March 2016), 62:1–62:33. DOI: http://dx.doi.org/10.1145/2893356

[20] Israt Nisa, Aravind Sukumaran-Rajam, Rakshith Kunchum, and P. Sadayappan. 2017. Parallel CCD++ on GPU for Matrix Factorization. In *GPGPU@PPoPP*. 73–83.

[21] Yusuke Nishioka and Kenjiro Taura. 2015. Scalable Task-Parallel SGD on Matrix Factorization in Multicore Architectures. In *ParLearning*.

[22] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. 2011. HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 693–701.

[23] Nvidia. 2015. cuBLAS. http://docs.nvidia.com/cuda/cublas/. (2015). [Online; accessed 17-Aug-2015].

[24] Nvidia. 2016. cuBLAS. http://docs.nvidia.com/cuda/cublas/#cublas-lt-t-gt-gemmbatched. (2016). [Online; accessed 7-Nov-2016].

[25] Nvidia. 2016. NVIDIA NVLink. http://www.nvidia.com/object/nvlink.html. (2016). [Online; accessed 26-Nov-2016].

[26] Nvidia. 2017. Programming Tensor Cores in CUDA 9. https://devblogs.nvidia.com/programming-tensor-cores-cuda-9/. (2017). [Online; accessed 21-Jan-2018].

[27] Jinoh Oh, Wook-Shin Han, Hwanjo Yu, and Xiaoqian Jiang. 2015. Fast and Robust Parallel SGD Matrix Factorization. In *KDD*. 865–874.

[28] Jeffrey Pennington, Richard Socher, and Christopher D Manning. 2014. Glove: Global vectors for word representation. In *EMNLP*. 1532–1543.

[29] Istvan Pillaszy, David Zibriczky, and Domonkos Tikk. 2010. Fast ALS-based matrix factorization for explicit and implicit feedback datasets. In *RecSys*. 71–78.

[30] Sebastian Schelter, Venu Satuluri, and Reza Bosagh Zadeh. 2014. Factorbird-a Parameter Server Approach to Distributed Matrix Factorization. In *NIPS Workshop on Distributed Matrix Computations*.

[31] Wei Tan, Liangliang Cao, and Liana Fong. 2016. Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs. In *HPDC*. 219–230.

[32] Christina Teflioudi, Faraz Makari, and Rainer Gemulla. 2012. Distributed Matrix Completion. In *ICDM*. 655–664.

[33] Aaron van den Oord, Sander Dieleman, and Benjamin Schrauwen. 2013. Deep content-based music recommendation. In *NIPS*. 2643–2651.

[34] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (2009), 65–76.

[35] Xiaolong Xie, Wei Tan, Liana L Fong, and Yun Liang. 2017. CuMF_SGD: Fast and Scalable Matrix Factorization. In *HPDC*.

[36] Hsiang-Fu Yu, Cho-Jui Hsieh, Si Si, and Inderjit S. Dhillon. 2012. Scalable Coordinate Descent Approaches to Parallel Matrix Factorization for Recommender Systems. In *ICDM*. 765–774.

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

[38] Yunhong Zhou, Dennis M. Wilkinson, Robert Schreiber, and Rong Pan. 2008. Large-Scale Parallel Collaborative Filtering for the Netflix Prize. In *AAIM*. 337–348.

[39] Yong Zhuang, Wei-Sheng Chin, Yu-Chin Juan, and Chih-Jen Lin. 2013. A fast parallel SGD for matrix factorization in shared memory systems. In *RecSys*. 249–256.