Faster and Cheaper: Parallelizing Large-Scale Matrix Factorization on GPUs

Wei Tan
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
wtan@us.ibm.com

Liangliang Cao[*]
Yahoo! Labs
New York City, NY, USA
liangliang@yahoo-inc.com

Liana Fong
IBM T. J. Watson Research Center
Yorktown Heights, NY, USA
llfong@us.ibm.com

ABSTRACT

Matrix factorization (MF) is used by many popular algorithms such as collaborative filtering. GPU with massive cores and high memory bandwidth sheds light on accelerating MF much further when appropriately exploiting its architectural characteristics.

This paper presents cuMF, a CUDA-based matrix factorization library that optimizes alternate least square (ALS) method to solve very large-scale MF. CuMF uses a set of techniques to maximize the performance on single and multiple GPUs. These techniques include smart access of sparse data leveraging GPU memory hierarchy, using data parallelism in conjunction with model parallelism, minimizing the communication overhead among GPUs, and a novel topology-aware parallel reduction scheme.

With only a single machine with four Nvidia GPU cards, cuMF can be 6-10 times as fast, and 33-100 times as cost-efficient, compared with the state-of-art distributed CPU solutions. Moreover, cuMF can solve the largest matrix factorization problem ever reported in current literature, with impressively good performance.

CCS Concepts

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

Keywords

GPU; CUDA; matrix factorization; alternating least square (ALS); parallel algorithms; performance optimization

1. INTRODUCTION

Matrix factorization (MF) factors a sparse rating matrix R (m by n, with N_z non-zero elements) into a m-by-f and a f-by-n matrices, as shown in Figure 1. MF is widely used for collaborative-filtering-based recommendations [1] in e-commerce (e.g., Amazon) and digital content streaming (e.g., Netflix). Very recently, MF is also applied in text mining, deriving hidden features of words [2].

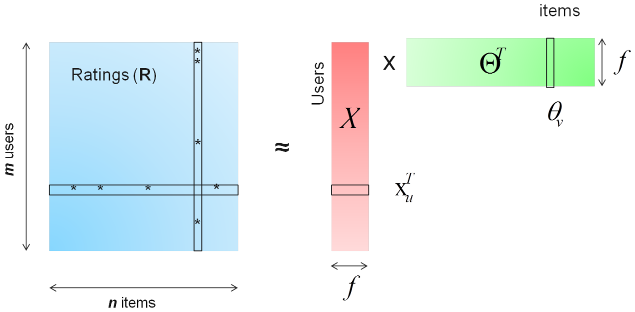


Figure 1: Matrix factorization.

Given the widespread use of MF, a scalable and speedy implementation is very important. In terms of **scale**, many parallel solutions [3–5] aim at medium-sized problems such as the Netflix challenge [6]. However, industry-scale recommendation problems have evolved to two-orders-of-magnitude larger. Figure 2 shows the scale of MF problems, in terms of number of ratings and number of model parameters. As an example, Facebook's MF is with over 100 billion ratings, 1 billion users, and millions of items [7]. No existing system except [7] has tackled problems at this scale. In terms of **speed** to reach acceptable accuracy, recommendations need to evolve promptly in online applications. Current approaches use distributed frameworks, including MPI [5] based, Spark [8] and parameter server [9], to address large-scale MF problems. However, they require costly clusters (e.g., 50-node) and still suffer from long latency.

Recently, the GPU emerges as an accelerator for parallel algorithms [14, 15]. It has big compute power (typically 10x floating-point operations per second–flops vs. a CPU) and memory bandwidth (typically 5x vs. a CPU) [16], but with limited amount of control logic and memory capacity. Particularly, GPU's success in deep learning [17] inspires us to try GPUs for MF. In deep learning, the computation is mainly dense matrix multiplication which is **compute bound**. As a result, GPU can train deep neural network 10x as fast as CPU by saturating its flops. However, unlike

[*] Work done while the author was with IBM.

[1] CCD++ [4], DSGD [10], DSGD++ [11], Facebook [7], Factorbird [9], Flink [12], Hugewiki [3], Netflix [6] SparkALS [8], and YahooMusic [13].

HPDC'16, May 31-June 04, 2016, Kyoto, Japan
© 2016 ACM. ISBN 978-1-4503-4314-5/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2907294.2907297

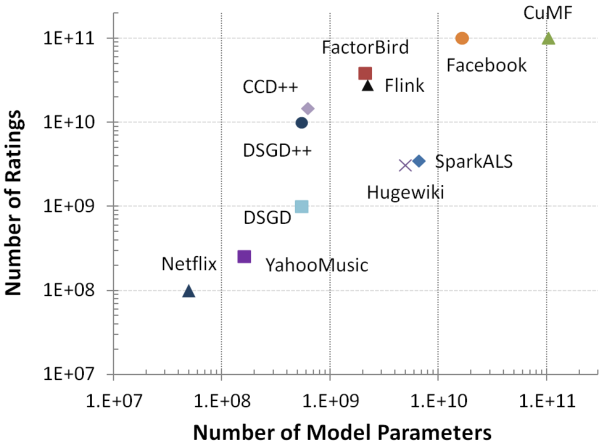


Figure 2: The scale of MF data sets[1]. Y-axis is the N_z of R, and x-axis is $(m + n) \times f$. CuMF can tackle MF problems of greater size, compared with existing systems.

deep learning, a MF problem involves sparse matrix manipulation which is usually **memory bound**. Given this, we want to explore a MF algorithm and a system that can still leverage GPU's compute and memory capability. We identified that, the alternating least square (ALS) algorithm [1] for MF is inherently parallel so as to exploit thousands of GPU cores. Moreover, compared with stochastic gradient descent (SGD), ALS has advantage when R is made up of implicit ratings and therefore no longer sparse [1].

Based on these observations, we design and implement **cuMF** (CUDA Matrix Factorization), a scalable ALS solution on one machine with one or more GPUs. CuMF achieves excellent scalability and performance by innovatively applying the following techniques on GPUs:

(1) On a single GPU, MF deals with sparse matrices, which makes it difficult to utilize GPU's compute power. We optimize memory access in ALS by various techniques including reducing discontiguous memory access, retaining hotspot variables in faster memory, and aggressively using registers. By this means cuMF gets closer to the roofline performance of a single GPU.

(2) On multiple GPUs, we add data parallelism to ALS's inherent model parallelism. Data parallelism needs a faster reduction operation among GPUs, leading to (3).

(3) We also develop an innovative topology-aware, parallel reduction method to fully leverage the bandwidth between GPUs. By this means cuMF ensures that multiple GPUs are efficiently utilized simultaneously.

The resulting CuMF is competitive in both speed and monetary cost. Table 1 shows cuMF's speed and cost compared with three CPU systems, NOMAD (with Hugewiki data) [5], Spark ALS [8], and Factorbird [9]. NOMAD and Spark ALS use Amazon AWS, and we pick an AWS node type similar to what Factorbird uses. CPU and GPU systems' cost is calculated by $(price\ per\ node\ per\ hr) * (\#nodes) * (execution\ time)$, with unit price taken when submitting this paper[2]. CuMF runs on one machine with two Nvidia K80 (four GPUs devices in total) from IBM Softlayer, with an amortized hourly cost of \$2.44. With faster speed and fewer machines, cuMF's overall cost of running these benchmarks is merely 1%-3% of the baseline systems compared. That is, cuMF is 33-100x as cost-efficient.

Table 1: Speed and cost of cuMF on one machine with four GPUs, compared with three CPU systems on cloud.

Baseline	baseline config	#nodes	price /node/hr	cuMF speed	cuMF cost
NOMAD	m3.xlarge	32	\$0.27	**10x**	**3%**
SparkALS	m3.2xlarge	50	\$0.53	**10x**	**1%**
Factorbird	c3.2xlarge	50	\$0.42	**6x**	**2%**

Note: Experiment details are in Section 5. NOMAD [5] uses Hugewiki data and AWS servers; it used m1.xlarge which is now superseded by m3.xlarge by Amazon. Factorbird's node is similar to AWS c3.2xlarge [9]. Speed is measured by wall-clock time when reaching the same accuracy.

Table 2: Notations

Name	**Meaning**	**Range**
R	sparse rating matrix: m by n	
X	low rank matrix: m by f	
Θ	low rank matrix: n by f	
m	vertical dimension of R	10^3 to 10^9
n	horizontal dimension of R	10^3 to 10^9
f	dimension of latent features	5 to 100s
N_z	number of non-zero entries in R	10^8 to 10^{11}
r_{uv}	R's value at position (u, v); $1 \le u \le m, 1 \le v \le n$	
$\mathbf{x}_u^T$	X's uth row; $1 \le u \le m$	
θ_v	Θ^T's vth column; $1 \le v \le n$	
R_{u*}	R's uth row; $1 \le u \le m$	
R_{*v}	R's vth column; $1 \le v \le n$	

Note: usually $N_z \gg m, n$ and $m, n \gg f$.

In summary, this paper describes a novel implementation of MF on a machine with GPUs and the set of exemplary optimization techniques in leveraging the GPU architectural characteristics. The experimental results demonstrate that with up to four Nvidia GPUs on one machine, cuMF is (1) competitive compared with multi-core methods, on medium-sized problems; (2) much faster than vanilla GPU implementations without memory optimization; (3) 6-10 times as fast, and 33-100 times as cost-efficient as distributed CPU systems, on large-scale problems; (4) more significantly, able to solve the largest matrix factorization problem ever reported.

This paper is organized as follows. Section 2 introduces matrix factorization and explains the two challenges in large-scale ALS, i.e., memory access on one GPU and scalability on multiple GPUs. Section 3 presents the memory-optimized ALS algorithm on a single GPU, to address the challenge in sparse and irregular memory access. Section 4 introduces the scale-up ALS algorithm to parallelize MF on multiple GPUs, to address the challenge in scaling to many GPUs. Section 5 shows the experiment results and Section 6 reviews related work. Section 7 concludes the paper.

2. PROBLEM DEFINITION

2.1 ALS algorithm for matrix factorization

Referring to the notations listed in Table 2, matrix factor-

[2] AWS price: https://aws.amazon.com/ec2/pricing/; GPU machine price: http://www.softlayer.com/gpu

ization is to factor a sparse matrix R with two lower-rank, dense matrices X and Θ, such that $R \approx X \cdot \Theta^T$.

As shown in Figure 1, suppose r_{uv} is the non-zero element of R at position (u, v), we want to minimize the following cost function (1). To avoid overfitting we use weighted-λ-regularization proposed in [6], where n_{x_u} and n_{θ_v} denote the number of total ratings on user u and item v, respectively.

$$J = \sum_{u,v}(r_{uv} - \mathbf{x}_u^T\theta_v)^2 + \lambda(\sum_u n_{x_u}||\mathbf{x}_u||^2 + \sum_v n_{\theta_v}||\theta_v||^2) \quad (1)$$

Many optimization methods, including ALS [6], CGD [4], and SGD [3] have been applied to minimize J. We adopt the ALS approach that would first optimize X while fixing Θ, and then to optimize Θ while fixing X. Consider

$$\frac{\partial J}{\partial \mathbf{x}_u} = 0$$

and

$$\frac{\partial J}{\partial \theta_v} = 0$$

which lead to the following equation:

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

together with:

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

By this means, ALS updates X using eq. (2), and updates Θ using eq. (3), in an alternating manner. Empirically, ALS often converges in 5-20 iterations, with each iteration consisting of both update-X and update-Θ. In the rest of this paper, we explain our method using update-X. The same method is applicable to update-Θ.

The formalism of ALS enables solving in parallel so as to harness the power of GPU. Eqs. (2) and (3) shows that, the updates of each $\mathbf{x}_u$ and θ_v are independent of each other. This independent nature does not hold for SGD, which randomly selects a sample r_{uv}, and updates the parameters by:

$$\begin{aligned}\mathbf{x}_u &= \mathbf{x}_u - \alpha[(\mathbf{x}_u^T\theta_v - r_{uv})\theta_v + \lambda\mathbf{x}_u] \\ \theta_v &= \theta_v - \alpha[(\mathbf{x}_u^T\theta_v - r_{uv})\mathbf{x}_u + \lambda\theta_v] \quad (4)\end{aligned}$$

Suppose there are two random samples r_{uv} and $r_{uv'}$ with the same row index u, their updates to $\mathbf{x}_u$ cannot be treated independently. Previous works on CPUs [3,5,10,11] all partition R into blocks with no overlapping rows and columns. Such a strategy works effectively on tens of CPU cores but is difficult to scale to a GPU with thousands of cores. As a result, we choose ALS instead of SGD for cuMF.

2.2 Challenges of speedy and scalable ALS

Table 3 lists the compute cost and memory footprint of solving X with eq. (2), using single precision. The calculation is divided into two phases, i.e.,

get_hermitian_x to obtain the left-hand Hermitian matrix $A_u = \sum_{r_{uv}\neq 0}(\theta_v\theta_v^T + \lambda I)$ and the right-hand $B_u = \Theta^T \cdot R_{u*}^T$, and

batch_solve to solve many equations $A_u\mathbf{x}_u = B_u$.

In line 3 of Table 3: *one item* in phase get_hermitian_x, to solve one row $\mathbf{x}_u$, obtaining A_u needs to calculate N_z/m times[3] of $\theta_v\theta_v^T$s, each of which needs $f(f+1)/2$ multiplications. The cost of obtaining B_u is $(N_z + N_z f)/m + 2f$ [18]. In terms of memory, A_u uses f^2 floats, B_u uses f, Θ^T uses nf, and a row of R in Compressed Sparse Row (CSR) format uses $(2N_z + m + 1)/m$. In phase batch_solve, solving the linear equation $A_u\mathbf{x}_u = B_u$ does not need additional memory storage by using in-place solvers, but has an f^3 computation cost.

Challenge 1. On a single GPU, how to optimize sparse, irregular and intensive memory access.

Table 3 shows that, computation is bounded in both phases **get_hermitian_x** ($\mathcal{O}(N_z f^2)$) and **batch_solve** ($\mathcal{O}(mf^3)$). CUDA library cuBLAS [19] already provides dense solvers for phase batch_solve, so we focus on the get_hermitian_x phase. This phase is very costly, especially when $N_z \gg m$ and therefore $N_z f^2 > mf^3$. What is more troublesome is the *sparse*, *irregular* and *intensive* memory access in this phase. Details are as follows:

1. Access many columns θ_v subject to $r_{uv} \neq 0$ for every u. This access is *irregular* w.r.t. Θ^T, due to the sparseness of R. In each iteration to solve one $\mathbf{x}_u$ we need to access n_{x_u} columns (N_z/m, on average) spread **sparsely** and **discontiguously** across the n columns of Θ^T. For example, in the Netflix data set [6], one user rates around 200 items on average, leading to a discontiguous access of 200 columns from the total 17,770 in Θ^T.

2. Aggregate many $\theta_v\theta_v^T$s and $\mathbf{x}_u\mathbf{x}_u^T$s, is memory *intensive* due to the large number of θ_vs and $\mathbf{x}_u$s to aggregate. According to eq. (2), obtaining A_u needs to calculate many $\theta_v\theta_v^T$s and aggregate them. Therefore, each element in column vector θ_v is accessed frequently, and the partial aggregation result is updated frequently. To calculate $\theta_v\theta_v^T$ we need to read each element of θ_v f times; after obtaining a $\theta_v\theta_v^T$, to add it to $\sum_{r_{uv}\neq 0}(\theta_v\theta_v^T + \lambda I)$ we need to write $f(f+1)/2$, or f^2 elements if the downstream solver does not appreciate symmetricity.

Section 3 presents how cuMF tackles Challenge 1, with experiment results shown in Sections 5.2 and 5.3.

Challenge 2. On multiple GPUs, how to scale and minimize communication overhead.

When m, n, N_z and f get larger, ALS is bounded by the memory capacity of a single GPU. For example, the update-X iteration is to be bounded by memory footprint of m A_us (mf^2 without considering symmetricity), X^T (mf), Θ^T (nf) and R ($2N_z + m + 1$). The current Nvidia Maxwell and Kepler GPUs have 12 GB memory per device. Each device would only be able to load 3 billion (3×10^9) single precision floats. However, the smallest data set, i.e., Netflix, in Figure 2, has $m = 480\text{K}$. When $f = 100$, m Hermitian matrices are with size $mf^2 = 480\text{K} \times 100^2 = 4.8$ billion floats > 3 billion.

Previous CPU solutions already encountered and partially addressed this memory capacity issue. PALS [6] partitions X and R by rows, solving each partition in parallel by replicating Θ^T. However, this **model parallelism** is only feasible when Θ^T is small. SparkALS [8], the ALS implementation in Spark MLlib [20], also partitions X and R by rows, and then solve each partition X_i in parallel. Its improvement

[3] N_z/m is the average number of non-zero entries per row.

Table 3: Compute cost and memory footprint of ALS: the update-X step

		compute cost		**memory footprint**	
		A_u in (2)	B_u in (2)	A_u in (2)	B_u in (2)
get_hermitian_x	one item	$N_z f(f+1)/2m$	$(N_z + N_z f)/m + 2f$	f^2	$nf+f+(2N_z+m+1)/m$
	m_b items	$m_b N_z f(f+1)/2m$	$m_b(N_z + N_z f)/m + 2m_b f$	$m_b f^2$	$nf+m_b f+m_b(2N_z+m+1)/m$
	all m items	$N_z f(f+1)/2$	$N_z + N_z f + 2mf$	mf^2	$nf+mf+(2N_z+m+1)$
batch_solve	one item	f^3			
	m_b items	$m_b f^3$			
	all m items	mf^3			

Note: here we omit some minor computations and auxiliary data structures needed in eq. (2).

to PALS is that, instead of replicating Θ^T, it splits Θ^T into overlapping partitions $\{\Theta_i^T\}$, where Θ_i^T contains only the necessary θ_v columns for all $\mathbf{x}_u$s in X_i. This improvement still has several deficiencies:

1. Generating Θ_i^T from X_i is actually a graph partitioning task and time consuming.
2. Transferring each Θ_i^T to X_i involves much network traffic, especially when $N_z \gg m$.
3. Θ_i^T may still be too big to fit into a single GPU device, especially when $N_z \gg m$.

Section 4 presents how cuMF tackles Challenge 2, with experiment results shown in Sections 5.4 and 5.5.

3. MEMORY-OPTIMIZED ALS ON ONE GPU

3.1 The GPU memory hierarchy

To address **Challenge 1** "On a single GPU, how to optimize sparse, irregular and intensive memory access", we need direct control on GPU's memory hierarchy. We choose Nvidia GPUs because they provides a rich set of *programmable memory* of different characteristics, shown in Table 4. [4]

Table 4: Programmable GPU memory

Memory type	**Size**	**Latency**	**Scope**
global	large	high	application
texture	medium	medium	application, read-only
shared	small	low	thread block
register	small	lowest	thread; not indexable

Although the principles of memory optimization are generally known, the specific implementation of ALS on GPU is not trivial due to the following reasons:

1. GPU has a lower clock frequency than CPU (typically < 1 GHz vs. 2-3 GHz). If the massive parallelism in GPU is not fully utilized, cuMF is likely to be slower than the highly-optimized CPU implementations.
2. Compared with CPU, GPU's global memory is smaller, e.g., 12 GB. In contrast, GPU has a much larger register file, e.g., 4 MB, which is largely ignored nowadays.
3. The control of register, shared, texture and global memory is complex. The global memory is large but slow, texture memory is read-only, and register and shared memory are not visible across GPU kernels (i.e., device functions). Moreover, registers are not *dynamically indexable*, which prevents them from being used for large arrays.

Due to these difficulties, without insight on both GPU hardware and algorithm specifics, an implementation can easily be bounded by memory capacity, latency or bandwidth, preventing us from harnessing the full power of GPU.

3.2 The base ALS algorithm

The base ALS algorithm 1 shows how to update X with eq. (2). The algorithm to update Θ is similar with all variables symmetrically exchanged. Algorithm 1 consists of two procedures: Get_Hermitian_X() and Batch_Solve().

Algorithm 1 Base ALS: Update X
Input $R_{m\times n}$
Input $\Theta^T : [\theta_1, \theta_2, ..., \theta_n]_{f\times n}$
Output $X : [\mathbf{x}_1^T; \mathbf{x}_2^T; ...; \mathbf{x}_m^T]_{m\times f}$

```
procedure Get_Hermitian_X($R, \Theta^T$)
    for $u \leftarrow 1, m$ do
        $\Theta_u^T \leftarrow$ sub-matrix of $\Theta^T$ with cols $\theta_v$ s.t. $r_{uv} \neq 0$
        $A_u \leftarrow 0$
        for all columns $\theta_v$ in $\Theta_u^T$ do
            $A_u \leftarrow A_u + \theta_v \theta_v^T + \lambda I$
        end for
        $B_u \leftarrow \Theta^T \cdot R_{u*}^T$
    end for
    return $([A_1, A_2, ...A_m], [B_1, B_2, ..., B_m])$
end procedure

procedure Batch_Solve($[A_1, A_2, ...A_m], [B_1, B_2, ..., B_m]$)
    for $u \leftarrow 1, m$ do
        $\mathbf{x}_u \leftarrow$ solve $A_u \cdot \mathbf{x}_u = B_u$
    end for
    return $[\mathbf{x}_1, \mathbf{x}_2, ...\mathbf{x}_m]^T$
end procedure

$(A, B) \leftarrow$ Get_Hermitian_X($R, \Theta^T$)
$X \leftarrow$ Batch_Solve($A, B$)
```

[4] There are *non-programmable memory* such as L1 and L2 cache. They also accelerate memory access but are not directly controllable by programmers. Therefore in cuMF we focus on the optimization by using the programmable memory.

3.3 The memory-optimized ALS algorithm MO-ALS

Table 3 indicates that, GET_HERMITIAN_X() in Algorithm 1 is memory intensive. We observed that Lines 3-7, i.e., computing A_u, takes much of the overall execution time. To optimize the performance, we enhance Algorithm 1 by leveraging different types of GPU memory. We call this memory-optimized ALS algorithm **MO-ALS**, as described in Algorithm 2. The following lines in Algorithm 1 are enhanced in MO-ALS:

1. Reading from Θ^T in Line 3. Θ^T with dimension $f \times n$ is stored in global memory. When collecting the sub-matrix Θ_u^T from Θ^T, we use **texture memory** as the cache because: (1) this collecting process enjoys spatial locality, (2) Θ^T is read-only when updating X, and (3) different Θ_u^Ts can potentially re-use the same θ_vs cached in texture memory. As a result, this caching step reduces discontiguous memory access. This optimization is shown in Line 3 in Algorithm 2.
2. Storage of Θ_u^T in Line 3. We use one thread block with f threads to calculate each A_u, and use the per-block **shared memory** to store Θ_u^T, so as to speed up the subsequent read in Line 6. However, for each A_u, we are not able to copy the whole Θ_u^T into its shared memory space. This is because Θ_u^T is of size $f \times n_{x_u}$ and too big compared to the 48 or 96 KB per-SM[5] shared memory. If a single thread block consumes too much shared memory, other blocks are prohibited from launching, resulting in low parallelism. In order to achieve higher parallelism, we select a bin size *bin*, and for each $\mathbf{x}_u$ only allocate a share memory space $\Theta_u^T[bin]$ of size $f \times bin$. In practice we choose *bin* between 10 and 30, while n_{x_u} can be hundreds to thousands. We iteratively move a subset of Θ_u^T into $\Theta_u^T[bin]$ to be processed in the following step. This optimization is shown in Lines 5-10 in Algorithm 2.
3. Update of A_u in Line 6. Here we need to read a θ_v from $\Theta_u^T[bin]$, calculate the $f \times f$ elements of $\theta_v \cdot \theta_v^T$, and add them to global variable A_u. Obviously A_u is a memory hotspot. In order to speedup the aggregation in A_u, we choose **register memory** to hold $\sum\limits_{\theta_v \in \Theta_u^T[bin]} \theta_v \theta_v^T$, and only update global memory A_u after we iterate over all columns in Θ_u^T. This reduces global memory access by a factor of n_{x_u}. This optimization is shown in Line 8 in Algorithm 2. More details are discussed in the following Section 3.4.

Figure 3 illustrates the memory usage of MO-ALS.

Algorithm 2 MO-ALS: Memory-Optimized ALS; update X on one GPU.
$\mathcal{G}\{var\}$: *var* in global memory
$\mathcal{T}\{var\}$: *var* in texture memory
$\mathcal{S}\{var\}$: *var* in shared memory
$\mathcal{R}\{var\}$: *var* in register memory
Input $R_{m \times n}$
Input $\Theta^T : [\theta_1, \theta_2, ..., \theta_n]_{f \times n}$
Output $X : [\mathbf{x}_1^T; \mathbf{x}_2^T; ...; \mathbf{x}_m^T]_{m \times f}$

```
 1: procedure GET_HERMITIAN_X_MO($R, \Theta^T$)
 2:     for $u \leftarrow 1, m$ do
 3:         $\mathcal{T}\{\Theta_u^T\} \leftarrow$ sub-matrix of $\mathcal{G}\{\Theta^T\}$ with cols $\theta_v$ s.t. $r_{uv} \neq 0$
 4:         $\mathcal{R}\{A_u\} \leftarrow 0$
 5:         while $\mathcal{T}\{\Theta_u^T\}$ has more cols not processed do
 6:             $\mathcal{S}\{\Theta_u^T[bin]\} \leftarrow$ next $bin$ cols from $\mathcal{T}\{\Theta_u^T\}$
 7:             for all cols $\theta_v$ in $\mathcal{S}\{\Theta_u^T[bin]\}$ do
 8:                 $\mathcal{R}\{A_u\} \leftarrow \mathcal{R}\{A_u\} + \mathcal{S}\{\theta_v\}\mathcal{S}\{\theta_v^T\} + \lambda I$
 9:             end for
10:         end while
11:         $\mathcal{G}\{A_u\} \leftarrow \mathcal{R}\{A_u\}$
12:         $\mathcal{G}\{B_u\} \leftarrow \mathcal{G}\{\Theta^T\} \cdot \mathcal{G}\{R_{u*}^T\}$
13:     end for
14:     return $\mathcal{G}([A_1, A_2, ...A_m], [B_1, B_2, ..., B_m])$

15:     $(A, B) \leftarrow$ GET_HERMITIAN_X_MO($R, \Theta^T$)
16:     $X \leftarrow$ BATCH_SOLVE($A, B$)
17: end procedure
```

3.4 Enhanced utilization of registers

We exploit the GPU register file which is larger and has higher bandwidth compared to its shared memory [21]. For example, in the latest Nvidia Maxwell generation GPUs, each SM has a 256 KB register file and only 96 KB shared memory. However, while there is much focus on using shared memory [22], the use of registers is surprisingly ignored. This **under-utilization of registers** is mainly due to the fact that, register variables cannot be dynamically indexed. That is to say, you cannot declare and refer to an array in register file[6]. In Algorithm 2, A_u is with size f^2 and to put it in register and access it, we have to declare f^2 variables instead of a single array. This makes the CUDA code hard to write. We use macro expansion in C to generate such a verbose paragraph of code. The snippet in Listing 1 demonstrates how the expanded code looks like when $f = 10$.

```
get_Au_kernel()
{ ...
  //declare Au in registers
  float temp0 = 0, temp1 = 0, temp2 = 0,
  temp3 = 0, temp4 = 0, temp5 = 0,
  temp6=0, temp7=0, temp8=0, temp9=0;
  ...
  float temp90 = 0, temp91 = 0, temp92 = 0,
  temp93 = 0, temp94 = 0, temp95 = 0,
  temp96=0, temp97=0, temp98=0, temp99=0;
  //aggregate Au in register
  for(k){
    temp0 += theta[k*f]*theta[k*f];
    temp1 += theta[k*f]*theta[k*f+1];
    ...
    temp98 += theta[k*f+9]*theta[k*f+8];
    temp99 += theta[k*f+9]*theta[k*f+9];
  }
  //copy register to global memory
  Au[0] = temp0;
  Au[1] = temp1;
  ...
  Au[98] = temp98;
  Au[99] = temp99;
```

[5] SM or SMX: stream multiprocessor. A GPU device usually consists of 10 to 15 SMs.

[6] An exception is that, the CUDA compiler may put very small (≤ 5) arrays on registers in loop unfolding.

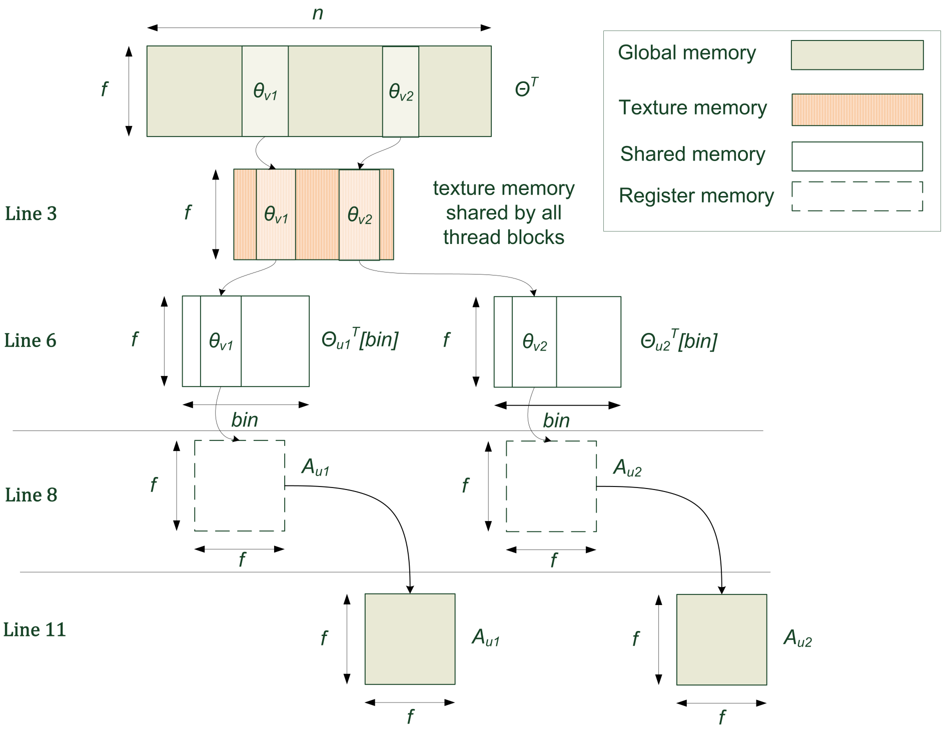


Figure 3: Illustration of memory usage in MO-ALS. Line numbers correspond to those in Algorithm 2. For simplicity, we solve two rows of X, i.e., $\mathbf{x}_{u1}$ and $\mathbf{x}_{u2}$, in parallel. In reality we solve as many rows of X as possible in parallel.

```
}
```

Listing 1: CUDA kernel code to use registers when f is 10

Limitation of MO-ALS. Algorithm 2 is able to deal with big X with one GPU, as long as Θ can fit into it. When X is big and Θ is small, we first load the whole Θ to the GPU, then load R and solve X in batches. However, this batch-based approach does not work when Θ cannot fit into a single GPU. This motivates us to scale to multiple GPUs on a single machine, as presented in Section 4.

4. SCALE-UP ALS ON MULTIPLE GPUS

Section 3 addresses **Challenge 1** regarding memory optimization on a single GPU. As problem size gets bigger, we need to address **Challenge 2**: "On multiple GPUs, how to scale and minimize communication overhead." This section presents a scale-up algorithm called **SU-ALS** which adds **data-parallelism** and **parallel-reduction** on top of MO-ALS.

4.1 The SU-ALS algorithm

In distributed machine learning, **model parallelism** and **data parallelism** are two common schemes [23]. Model parallelism partitions **parameters** among multiple learners with each one learns a subset of parameters. Data parallelism partitions the training **data** among multiple learners with each one learns all parameters from its partial observation. These two schemes can be combined when both model parameters and training data are large.

ALS is inherently suitable for model parallelism, as the updates of each $\mathbf{x}_u$ and θ_v are independent. As discussed in Section 2.2, both PALS and SparkALS employ only model parallelism without considering data parallelism. To solve X in parallel, PALS and SparkALS partition X among multiple nodes. PALS broadcasts the whole Θ^T while SparkALS transfers a subset of it to each X partition. As pointed out by [4], both approaches are inefficient and may cause out-of-memory failure, when Θ^T is big and ratings are skewed.

To tackle large-scale problems, on top of the existing model parallelism, we design a data-parallel approach. A limitation of model parallelism is that, it requires all A_us in one partition $X^{(j)}$ $(1 \leq j \leq q)$ to be computed on the same GPU. Consequently, a subset of Θ^T has to be transferred into that GPU. In contrast, our data-parallel approach distributes the computation of any single Hermitian matrix A_u to multiple GPUs. Instead of transferring all θ_vs to one GPU, it calculates a local A_u on each GPU with only the local θ_vs, and reduce (aka., aggregate) many local A_us later. Assume that there are p GPUs to parallelize on, we re-write eq. (2) to its data-parallelism form as:

$$A_u = \sum_{r_{uv} \neq 0} (\theta_v \theta_v^T + \lambda I) = \sum_{i=1}^{p} \sum_{r_{uv} \neq 0}^{GPU_i} (\theta_v \theta_v^T + \lambda I) \qquad (5)$$

This approach is described in Algorithm 3 and illustrated in Figure 4.

Lines 2-4: partitions the input data. Θ^T is evenly split by columns into p partitions, X is evenly split by rows into q partitions, and R is split by rows and columns following the partition schemes of X and Θ^T.

Lines 5-7: copies $\Theta^{T(i)}$ to GPU$_i$ $(1 \leq i \leq p)$, in parallel.

Lines 8-20: loop over $\{X^{(1)}, X^{(2)}, ..., X^{(q)}\}$ and solve each $X^{(j)}$ partition in sequence $(1 \leq j \leq q)$. Given more GPUs, this sequential loop can further be parallelized.

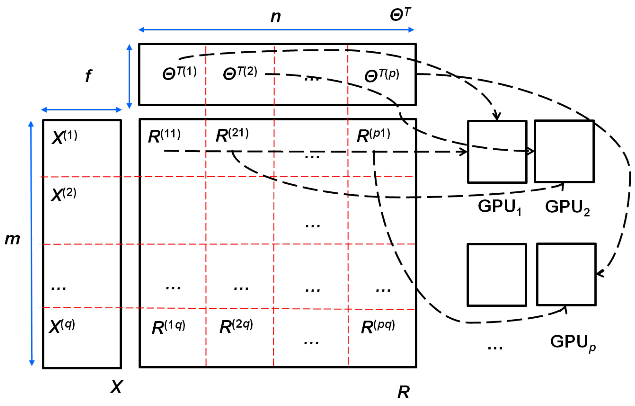


Figure 4: SU-ALS. Θ^T is partitioned evenly and vertically, and stored on p GPUs. X is partitioned evenly and horizontally, and solved in batches, achieving model parallelism. Each X batch is solved in parallel on p GPUs, each with Θ^T's partition on it, achieving data parallelism.

Algorithm 3 SU-ALS: Scale-Up ALS; update X on multiple GPUs.

```
 1: Given $p$ GPUs: $\text{GPU}_1$, $\text{GPU}_2$, ..., $\text{GPU}_p$.
 2: $\{\Theta^{T(1)}, \Theta^{T(2)}, ..., \Theta^{T(p)}\} \leftarrow VerticalPartition(\Theta^T, p)$
 3: $\{X^{(1)}, X^{(2)}, ..., X^{(q)}\} \leftarrow HorizontalPartition(X, q)$
 4: $\{R^{(11)}, R^{(12)}, ..., R^{(pq)}\} \leftarrow GridPartition(R, p, q)$
 5: parfor $i \leftarrow 1, p$ do            ▷ parallel copy to each $\text{GPU}_i$
 6:     copy $\text{GPU}_i \leftarrow \Theta^{T(i)}$
 7: end parfor
 8: for $j \leftarrow 1, q$ do                        ▷ model parallel
 9:     parfor $i \leftarrow 1, p$ do          ▷ data parallel on $\text{GPU}_i$
10:         copy $\text{GPU}_i \leftarrow R^{(ij)}$
11:         $(A^{(ij)}, B^{(ij)}) \leftarrow$ GET_HERMITIAN_X_MO$(R^{(ij)}, \Theta^{T(i)})$
12:         SYNCHRONIZE_THREADS()
13:         $\{A_1^{(ij)}, A_2^{(ij)}, ..., A_p^{(ij)}\} \leftarrow A^{(ij)}$
14:         $\{B_1^{(ij)}, B_2^{(ij)}, ..., B_p^{(ij)}\} \leftarrow B^{(ij)}$
15:         $A_i^{(j)} \leftarrow \sum_{k=1}^{p} A_i^{(kj)}$
16:         $B_i^{(j)} \leftarrow \sum_{k=1}^{p} B_i^{(kj)}$
17:         $X_i^{(j)} \leftarrow$ BATCH_SOLVE$(A_i^{(j)}, B_i^{(j)})$
18:     end parfor
19:     $X^{(j)} \leftarrow \{X_1^{(j)}, X_2^{(j)}, ..., X_p^{(j)}\}$
20: end for
```

Line 9-18: parallel loop over $\{\Theta^{T(i)}\}$ $(1 \leq i \leq p)$ to solve $X^{(j)}$. Without sufficient number of GPUs, this **parallel for** loop can degrade to a **sequential** one.

Lines 11: on GPU_i $(1 \leq i \leq p)$, for each row $\mathbf{x}_u$ in $X^{(j)}$, calculate the A_u local to GPU_i by only observing $\Theta^{T(i)}$ and $R^{(ij)}$:

$$A_u^i = \sum_{r_{uv} \neq 0}^{GPU_i} (\theta_v \theta_v^T + \lambda I) \tag{6}$$

Similarly, we calculate the local B_u matrix:

$$B_u^i = \Theta^{T(i)} \cdot (R_{u*}^{(ij)})^T \tag{7}$$

The collection of all A_u^is and B_u^is on GPU_i are denoted as $(A^{(ij)}, B^{(ij)})$.

Line 12: a synchronization barrier to wait for all parfor threads to reach this step.

Lines 13-14: evenly partition $A^{(ij)}$ and $B^{(ij)}$ by rows of $X^{(j)}$. That is, $A^{(ij)}$ on GPU_i is evenly divided into p portions:

$$A_1^{(ij)}, A_2^{(ij)}, ..., A_p^{(ij)}$$

$B^{(ij)}$ is partitioned in the same manner into:

$$B_1^{(ij)}, B_2^{(ij)}, ..., B_p^{(ij)}$$

Lines 15-16: **parallel reduce** p $A^{(ij)}$s and $B^{(ij)}$s into the global $A^{(j)}$ and $B^{(j)}$, on p GPUs. GPU_i takes care of the reduction of partition i of all $A^{(kj)}$s $(1 \leq k \leq p)$. See Figure 5 (a) for an example where $j = 1$ and $p = 4$: GPU_1 reduces $\{A_1^{(11)}, A_1^{(21)}, A_1^{(31)}, A_1^{(41)}\}$, GPU_2 reduces $\{A_2^{(11)}, A_2^{(21)}, A_2^{(31)}, A_2^{(41)}\}$, and so on. $B^{(ij)}$s are reduced in the same manner.

Line 17: solve the p partitions concurrently on p GPUs. GPU_i solves the local partition $(A_i^{(j)}, B_i^{(j)})$ it reduces in *Lines 15-16*.

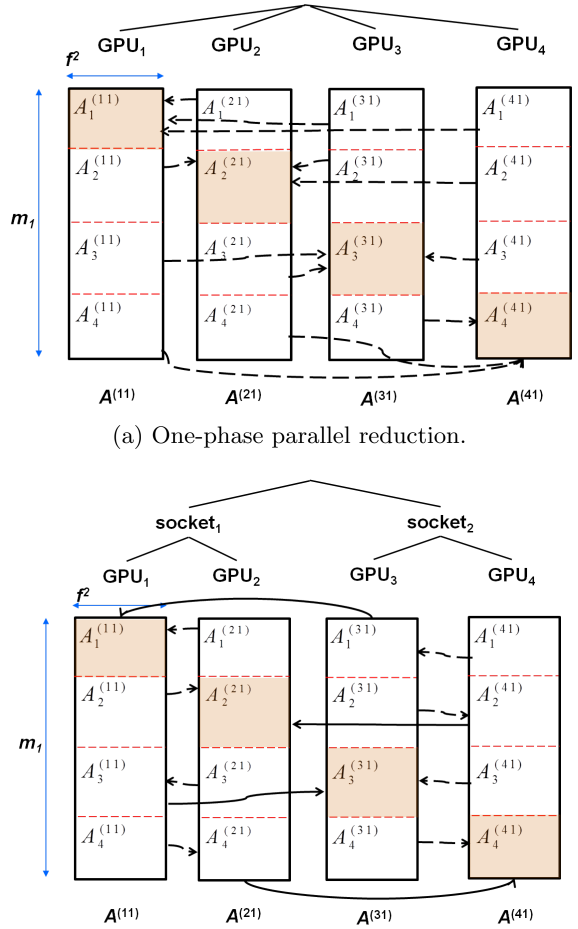
Line 19: collect p partitions $\{X_1^{(j)}, X_2^{(j)}, ..., X_p^{(j)}\}$ on p GPUs to obtain $X^{(j)}$.

4.2 Topology-aware parallel reduction to speed up SU-ALS

Parallel reduction. Refer to Lines 13-17 of Algorithm 3, $(A^{(j)}, B^{(j)})$ could have been reduced in one GPU (say, GPU_1) and $X^{(j)}$ solved there. However, this simple approach fails to parallelize either data transfer or computation. Moreover, multiple GPUs on a machine are usually connected through a PCIe bus. PCIe channels are full-duplex, meaning that data transfer in both directions can happen simultaneously without affecting each other. To leverage the bandwidth in both directions, we develop a parallel reduction scheme that evenly utilizes both incoming and outgoing channels of all GPUs, as shown in Figure 5 (a). Experiment on Hugewiki data set shows that this optimization is 1.7x as fast compared with the reducing-by-one-GPU approach. After this parallel reduction, *batch_solve* begins on p GPUs in parallel.

Topology-aware parallel reduction. Figure 5 (a) assumes a flat interconnection where all GPUs directly connect to a PCIe root. This assumption may not always hold. For example, in a two-socket machine with four GPUs, a typical configuration is that every two GPUs connect to one socket. Communications between the two GPUs in the same socket still go though the local PCIe bus, while communications between GPUs in different sockets go through the inter-socket connection. In this case, intra-socket transfers enjoy zero-copy and faster duplex PCIe channel, compared with inter-socket transfers. In such a topology, the scheme shown in Figure 5 (a) is not optimal.

Based on the GPU connection topology, we design a two-phase parallel reduction scheme shown in Figure 5 (b). In this scheme, each partition is first reduced intra socket (see the dash line). Afterward, the partial, intra-socket reduction results are moved across socket and generate the final reduction result (the solid line). Experiments show that this two-phase scheme enjoys an additional 1.5x speedup compared with the one-phase scheme shown in Figure 5 (a).



(a) One-phase parallel reduction.

(b) Two-phase parallel reduction considering PCIe hierarchy: phase-1 (intra-socket) in dash lines; phase-2 (inter-socket) in solid lines.

Figure 5: Parallel reduce $A^{(ij)}$ in SU-ALS when $j = 1$ and $p = 4$. For $1 \leq i \leq p$, on GPU$_i$, $A^{(ij)}$ is evenly partitioned into p pieces: $A_1^{(ij)}, A_2^{(ij)}, ..., A_p^{(ij)}$. Afterward GPU$_i$ reduces all $A_i^{(kj)}$ across p GPUs ($1 \leq k \leq p$). This not only achieves parallel get_hermitian_x and batch_solve, but also leverages cross-GPU bandwidth efficiently.

4.3 How to partition?

Assume a single GPU's memory capacity is C. According to Algorithm 3, one GPU needs to hold $X^{(j)}$, $\Theta^{(i)}$, $R^{(ij)}$, $A^{(j)}$, and $B^{(j)}$. Therefore the choices of p and q are subject to (8).

$$\frac{m \times f}{q} + \frac{n \times f}{p} + |R^{(ij)}| + \frac{m}{q} \times f^2 + \frac{m}{q} \times f + \epsilon < C \quad (8)$$

ϵ is a headroom space for miscellaneous small variables. In practice, when $C = 12$ GB we choose $\epsilon = 500$ MB.

Here are some best practices in choosing p and q:

1. If $p = 1$ can satisfy (8), you can solve X in a single GPU in sequential batches. In this case SU-ALS is equivalent to MO-ALS.
2. When q increases and $p = 1$ satisfies (8), q should not increase any more. At this time there is already no need to further partition X.
3. We usually start from p such that $\frac{n \times f}{p} \approx \frac{C}{2}$, and then choose the smallest q that satisfies (8).

4.4 Implementation of cuMF

This section describes selected details of cuMF. CuMF is implemented in C, using CUDA 7.0 and GCC OpenMP v3.0. It has circa 6,000 lines of code.

Out-of-core computation. As seen in Figure 2 and Table 5, rating and feature matrices can both have 100 billion entries. This goes far beyond the host and device memory limit. For such out-of-core problems, cuMF first generate a partition scheme, planning which partition to send to which GPU in what order. With this knowledge in advance, cuMF uses separate CPU threads to preload data from disk to host memory, and separate CUDA streams to preload from host memory to GPU memory. By this proactive and asynchronous data loading, we manage to handle out-of-core problems with close-to-zero data loading time except for the first load.

Elasticity to resources. Algorithm 3 is generic enough to cover many deployment scenarios where the number of GPUs are fewer or more than p or q. With more GPUs, the sequential **for** at *Line 8* can be parallelized; with fewer GPUs, the **parfor** at *Line 9* can be turned into a sequential for. This is similar to how MapReduce deals with resource elasticity: when there are fewer/more parallel tasks compared with task slots, tasks will be executed in fewer/more waves. By this design cuMF is able to solve ALS of any size.

Fault tolerance. Handling machine failure is straightforward in cuMF which uses a single machine. During ALS execution we asynchronously checkpoint X and Θ generated from the latest iteration, into a connected parallel file system. When the machine fails, the latest X or Θ (whichever is more recent) is used to restart ALS.

5. EXPERIMENTS

This section reports the performance evaluations on cuMF. We compare cuMF with multi-core solutions libMF [3] and NOMAD [5]. We also compare with distributed solutions including NOMAD (on multi-nodes), Factorbird [9], Spark ALS [8], and a Giraph based solution from Facebook [7]. We select these solutions because they either perform better than earlier studies [4, 10, 11, 24, 25], or are able to handle large data sets. Because none of existing GPU-based solutions [26, 27] can tackle big data sets, we do not compare with their results.

The goals of our experiments are to provide key insights on the following questions:

1. how would cuMF on a single GPU compare with highly optimized multi-core methods, such as libMF and NOMAD, on medium-size problems? (Section 5.2)
2. are the memory optimization done by MO-ALS effective? (Section 5.3)
3. is SU-ALS scalable with multiple GPUs? (Section 5.4)
4. with four GPUs on one machine, how would cuMF compare with multi-node methods on large-size problems? (Section 5.5)

Table 5: Data sets

Data Set	m	n	N_z	f	λ
Netflix	480,189	17,770	99M	100	0.05
YahooMusic	1,000,990	624,961	252.8M	100	1.4
Hugewiki	50,082,603	39,780	3.1B	100	0.05
SparkALS	660M	2.4M	3.5B	10	0.05
Factorbird	229M	195M	38.5B	5	0.05
Facebook	1B	48M	112B	16	0.05
cuMF	1B	48M	112B	100	0.05

5.1 Experiment setting

Data Sets. We use three public data sets, i.e., Netflix [6], YahooMusic [13] and Hugewiki [3] to measure the convergence speed. For large-size problems, we synthesize the data sets used by SparkALS [8], Factorbird [9] and Facebook [7]. For these three systems, we compare the per iteration latency because their convergence speed are not reported. We also synthesize a data set to the size that is beyond any previous attempts. That is, we use the rating matrix of the Facebook data set, with an enlarged f of 100 from the original 16. Characteristics of these data sets are shown in Table 5.

Hardware. Unless otherwise mentioned, we use one to four Nvidia Titan X GPUs, each with 3072 CUDA cores and 12 GB memory, on one machine. The machine is with two Intel Xeon E5 CPUs, 256 GB RAM, and the GPFS [28] as the file system.

Parameters. The f and λ values for each data set are given in Table 5. Feature matrices are initiated with random numbers in $[0, 1]$. We focus on the speed and scalability of cuMF, and therefore did not spend much effort in hyper-parameter tuning to achieve the best accuracy.

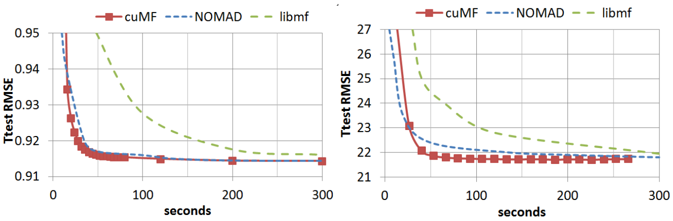
Evaluation. For Netflix, YahooMusic and Hugewiki, we evaluate the root-mean-square-error (RMSE) on test set. Performance of libMF and NOMAD is obtained from [3, 5]. For SparkALS, Factorbird and Facebook, since the data is synthetic and no test RMSE is reported, we compare the per iteration run time.

5.2 MO-ALS on a single GPU

We run cuMF on one GPU, measure the test RMSE w.r.t. training time, and compare with NOMAD and libMF on one machine with 30 cores [5]. We choose these two for comparison because they are among the fastest multi-core solutions. In Figure 6, on both Netflix and YahooMusic, cuMF performs slightly worse than NOMAD at the beginning but slightly better later, and constantly faster than libMF. CuMF use ALS where each iteration takes much longer than SGD based methods. This makes it slower at the beginning. Nevertheless cuMF catches up quickly and outperforms soon afterward.

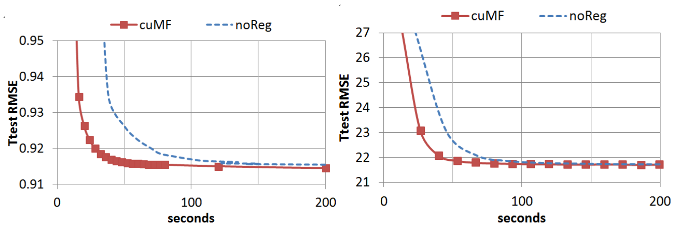
5.3 Benefit of using registers and texture memory in MO-ALS

We first measure the benefit of aggressively using registers in MO-ALS. Figure 7 compares cuMF's performance, with or without using register memory to aggregate A_u, on one GPU. On Netflix data, cuMF converges 2.5 times as slow (75 seconds vs. 30 seconds when RMSE reaches 0.92) without using registers. The result strongly supports the idea of aggressively using registers. Among all optimizations done in MO-ALS, using registers for A_u brings the greatest performance gain. Without using the registers, cuMF converges 1.7 times as slow on YahooMusic. YahooMusic has a smaller performance degradation without using registers than Netflix. This is because its rating matrix is more sparse. As a result, its GET_HERMITIAN_X() is less heavy-duty and occupy a smaller percentage of the overall run time.



(a) Netflix (b) YahooMusic

Figure 6: Test RMSE convergence speed in terms of number of iterations: cuMF (with one GPU), NOMAD and libMF (both with 30 CPU cores).

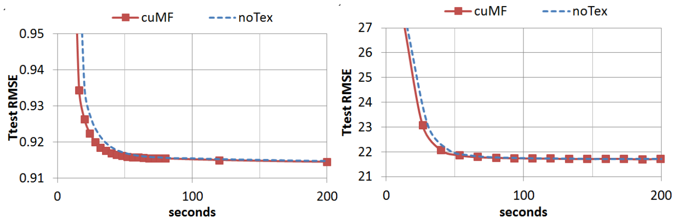


(a) Netflix (b) YahooMusic

Figure 7: The convergence speed of cuMF, with or without aggressively using registers on one GPU.

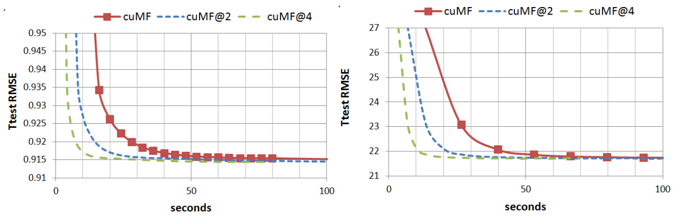
Figure 8 compares cuMF's performance with or without using texture memory. Using texture memory, the convergence speed is 25% to 35% faster. The reason for the gain is due to the fact that Algorithm 2 updates Θ and X in an alternating manner, i.e., Θ is read-only when updating X, and X is read-only when updating Θ. This feature enables us to leverage the read-only texture memory in GPU to speed up memory access. Since YahooMusic data is more sparse, the penalty of not using texture memory is also smaller.

5.4 Scalability of SU-ALS on multiple GPUs



(a) Netflix (b) YahooMusic

Figure 8: The convergence speed of cuMF, with or without texture memory on one GPU.



(a) Netflix

(b) YahooMusic

Figure 9: The convergence speed of cuMF on one, two, and four GPUs.

This section first studies how a problem with the fixed size data set can be accelerated with multiple GPUs. In both Netflix and YahooMusic, X and Θ can both fit on one GPU. As a result only model parallelism is needed. We run Netflix and YahooMusic data on one, two and four GPUs, respectively, on one machine. As seen from Figure 9, close-to-linear speedup is achieved. For example, the speedup is 3.8x when using four GPUs, measured at RMSE 0.92. Detailed profiling shows that, the very small overhead mainly comes from PCIe IO contention when multiple GPUs read from host memory simultaneously.

In contrast, NOMAD observed a sub-linear speedup on certain data sets, due to cache locality effects and communication overhead [5]. CuMF achieves better scalability due to the optimized memory access and inter-GPU communication. An advantage of cuMF is that, it consolidates massive computation on a single machine, so that it only uses PCIe connections which are faster than any existing network.

We also tested Hugewiki data on four GPUs. We compare with multi-node NOMAD (on 64-node HPC cluster and 32-node AWS cluster) because it outperforms DSGD [10] and DSGD++ [11]. Hugewiki is a relatively large data set where $m \approx 50\text{M}$, $n \approx 40\text{K}$, and $N_z \approx 3\text{B}$. When using X to solve Θ, X is too big to fit on one GPU. According to Algorithm 3 we partition X evenly into four GPUs and apply data parallelism. We use the two-phase parallel reduction scheme shown in Figure 5 (b), because our machine has two sockets each connecting to two GPUs. With all the intra- and inter-GPU optimizations, cuMF performs slightly better than NOMAD on a 64-node HPC cluster (again, with a slower start), and much better than NOMAD on a 32-node AWS cluster, as shown in Figure 10. This result is very impressive, as a 64-node HPC cluster is outperformed by only one node plus four GPUs. This indicates that cuMF brings a big saving in infrastructure and management cost.

5.5 Solve extremely large-scale problems

We conduct experiments on three extremely large problems. In this experiment we use four Nvidia GK210 cards on one machine. Each card is with 2496 CUDA cores (slightly fewer than Titan X) and 12 GB memory, and every two cards are encapsulated as one K80 GPU.

The results for the following experiments are shown in Figure 11. SparkALS [8] is a benchmark of Spark MLlib ALS. Its rating matrix is from the 100-by-1 duplication of the *Amazon Reviews* [29] data. It uses 50×m3.2xlarge AWS nodes with Spark MLlib 1.1, and takes 240 seconds per ALS iteration. We synthesize the data in the same way as [8], apply model parallelism solving X, and apply data parallelism solving Θ. CuMF with four GPUs completes one iteration in 24 seconds, which is **ten times as fast** as SparkALS.

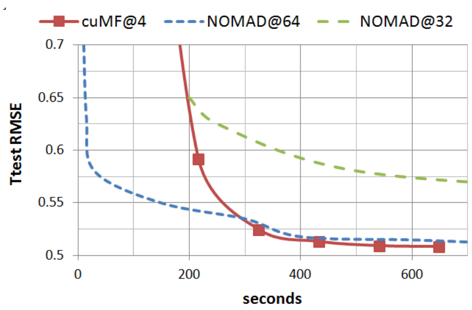


Figure 10: CuMF@4GPU, vs. NOMAD on a 64-node HPC cluster and a 32-node AWS cluster, with Hugewiki data. CuMF converges similar to NOMAD with 64 nodes, and 10x as fast as NOMAD with 32 nodes.

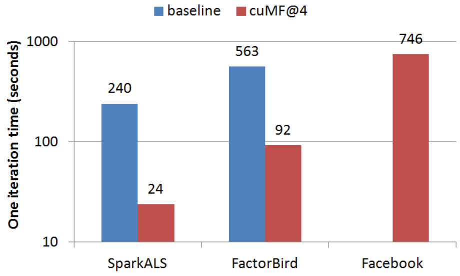


Figure 11: CuMF@4GPU on three very large data sets, compared with the their original implementations as baselines.

Factorbird [9] is a parameter server system for MF. It trains a data set ($m = 229\text{M}$, $n = 195\text{M}$, $f = 5$, and $N_z = 38.5\text{B}$) on a cluster of 50 nodes. We synthesize the data using the method described in [11]. We use only model parallelism in solving X and Θ because they both fit into one GPU. CuMF with four GPUs completes one iteration in 92 seconds. Factorbird needs 563 seconds per iteration, and with SGD it may need more iterations than ALS.

Facebook [7] recently revealed that its MF system deals with 1 billion users, millions of items and over 100 billion ratings. Given this hint we did a 160-by-20 duplication of the Amazon Review data, yielding a data set with $m = 1056\text{M}$, $n = 48\text{M}$, $f = 16$, and $N_z = 112\text{B}$. We use data parallelism to solve both X and Θ. Especially, when solving Θ, because X is huge (1056M×16 floats) and cannot fit on 4 GPUs, we change the **parfor** in Line 9-18 of Algorithm 3 into a **sequential for** with many batches. By doing this, cuMF completes one ALS iteration in 746 seconds. [7] does not report its speed on 50 Giraph workers, but we believe cuMF is competitive given the size of the problem and the low cost of one machine with GPUs. We further try a larger $f = 100$, and cuMF completes one iteration in 3.8 hours. To the best of our knowledge, this is by far the largest matrix factorization problem ever reported in literature.

As a summary, on two extremely large data sets, CuMF with four GPUs significantly outperforms the original distributed implementations. CuMF is also able to factorize the largest collaborative filtering matrix ever reported.

6. RELATED WORK

SGD, Coordinate Gradient Descent (CGD) and ALS are the three main algorithms for MF. This section firstly reviews the three algorithms and then the methods to parallelize them. Subsequently, we review GPU-based MF solutions.

6.1 MF algorithms

SGD based algorithms [1] have been often applied to matrix factorization. SGD handles large scale problems by splitting the rating matrix into blocks along with sophisticated conflict-avoiding updates. CGD based algorithms update along one coordinate direction in each iteration. [25] improved the default cyclic CGD scheme by prioritizing the more important coordinates. ALS algorithms [6,30] have advantages in easy to parallelize, converging in fewer iterations, and dealing with non-sparse rating matrices [1]. CuMF is based on ALS.

6.2 Parallel computing paradigms

Parallel SGD. SGD has been parallelized in environments including multi-core [3], multi-node MPI [5,11], MapReduce [10, 24] and parameter-server [9, 31]. These studies are inspired by HOGWILD! [32], which shows how to avoid expensive memory locking in memory sharing systems for some optimization problems with sparse updates. These methods partition the rating matrix into blocks with no overlapping rows or columns, and work on these blocks in parallel. They also use asynchronous communication, overlapping of communication and computation, and shared memory to achieve further speedup.

LibMF [3] is a very efficient SGD based library for matrix factorization on multi-cores. It has out performed nearly all other approaches on a 12-core machine. However, our experimental results show that libMF stops scaling beyond 16 cores, similar to the observation of [33]. Moreover, libMF is a single-machine implementation, which limits its capability to solve large-scale problems. NOMAD [5] extends the idea of block partitioning, adding the capability to release a portion of a block to another thread before its full completion. It performs similar to libMF on a single machine, and can scale out to a 64-node HPC cluster.

Parameter Server with SGD. More recently, the idea of "parameter server" [31, 34] emerges for extremely large-scale machine learning problems. In this paradigm, the *server nodes* store parameters, while the *worker nodes* store training data and compute on them. The parameter-server framework manages asynchronous communication between nodes, flexible consistency models, elastic scalability, and fault tolerance. Following this idea, Petuum [31] runs Netflix data on a 512 cores cluster using SGD. Factorbird [9] is a parameter server specifically implemented for matrix factorization, also based on SGD.

Parallel CGD. CCD++ [4] 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. In practice, CCD++ behaves well in the early stage of optimization, but then becomes slower than libMF.

Parallel ALS. As discussed in Section 2.2, PALS [6] and SparkALS [20] parallelize ALS by feature matrix replication and partial replication, respectively. These approaches does not work when feature matrices get extremely large. Facebook [7] tackles this issue by feeding a feature matrix in parts to a node. For example, when solving X, X is partitioned disjointedly across nodes; Θ is also partitioned and rotated across the same set of nodes. When a Θ partition $\Theta^{(j)}$ meets X partition $X^{(i)}$, $X^{(i)}$ is updated by observing $\Theta^{(j)}$; $X^{(i)}$ completes an iteration of update after it meets all $\Theta^{(j)}$s. This is somewhat similar to SU-ALS but SU-ALS does not use rotation, as GPUs do not have sufficient memory to do rotation.

GraphLab [35] implements ALS in such a way that when Θ is big, it is distributed among multiple machines. When updating a $\mathbf{x}_u$ in a node, all needed θ_vs are fetched on-the-fly from all nodes. This involves a lot of cross-node traffic and puts a high requirement on network bandwidth.

6.3 GPU approaches

[26] employs GPU-based restricted Boltzmann machine for collaborative filtering, which gives relative performance compared with a CPU implementation on Netflix data. [27] implements both SGD and ALS on GPU to solve MF. It uses a mini-batch-based and sequential version of SGD, and a variant of ALS that adjusts (rather than re-calculates) the inverse of the Hermitian matrices in each iteration. They neither optimize the memory access to fully utilize GPU's compute power, nor scale to multiple GPUs to handle large-scale problems.

Compared with CPU-based approaches, cuMF has better performance with a fraction of hardware resources. Compared with GPU-based approaches, our optimization in memory access and parallelism yields higher performance and scalability.

7. CONCLUSION

Advances in GPU computing opens new possibilities to accelerate high performance parallel and large scale distributed applications. GPUs enable us to consolidate huge compute power and memory bandwidth on one or few machines, which may reduce the demand for big distributed clusters. This scale-up approach provides an alternative to the scale-out systems in distributed applications. Evidently, cuMF using a single machine with GPUs is faster and cheaper to solve matrix factorization, compared with distributed CPU systems. CuMF achieves this by optimizing memory access, combining data and model parallelism, and applying topology-aware parallel reduction.

In future work we plan to extend cuMF to deal with other sparse problems such as graph algorithms [36], and use it to accelerate Hadoop/Spark framework [15].

8. REFERENCES

[1] Y. Koren, R. M. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, pp. 30–37, 2009.

[2] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *EMNLP*, 2014, pp. 1532–1543.

[3] Y. Zhuang, W. Chin, Y. Juan, and C. Lin, "A fast parallel SGD for matrix factorization in shared memory systems," in *RecSys*, 2013, pp. 249–256.

[4] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *ICDM*, 2012, pp. 765–774.

[5] H. Yun, H.-F. Yu, C.-J. Hsieh, S. Vishwanathan, and I. S. Dhillon, “NOMAD: Non-locking, stochastic multi-machine algorithm for asynchronous and decentralized matrix completion,” in *VLDB*, 2014, pp. 975–986.

[6] Y. Zhou, D. M. Wilkinson, R. Schreiber, and R. Pan, “Large-scale parallel collaborative filtering for the netflix prize,” in *AAIM*, 2008, pp. 337–348.

[7] M. Kabiljo and A. Ilic, “Recommending items to more than a billion people,” https://code.facebook.com/posts/861999383875667, 2015, [Online; accessed 17-Aug-2015].

[8] B. Yavuz, X. Meng, and R. Xin, “Scalable Collaborative Filtering with Spark MLlib,” https://databricks.com/blog/2014/07/23/scalable-collaborative-filtering-with-spark-mllib.html, 2014, [Online; accessed 15-Aug-2015].

[9] S. Schelter, V. Satuluri, and R. B. Zadeh, “Factorbird-a parameter server approach to distributed matrix factorization,” in *NIPS Workshop on Distributed Matrix Computations*, 2014.

[10] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, “Large-scale matrix factorization with distributed stochastic gradient descent,” in *KDD*, 2011, pp. 69–77.

[11] C. Teflioudi, F. Makari, and R. Gemulla, “Distributed matrix completion,” in *ICDM*, 2012, pp. 655–664.

[12] T. Rohrmann, “How to factorize a 700 GB matrix with Apache Flink,” http://data-artisans.com/how-to-factorize-a-700-gb-matrix-with-apache-flink/, 2015, [Online; accessed 15-Aug-2015].

[13] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, “The Yahoo! Music Dataset and KDD-Cup ’11,” in *KDD Cup 2011 competition*, 2012.

[14] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, “Design and evaluation of the gemtc framework for gpu-enabled many-task computing,” in *HPDC*, 2014, pp. 153–164.

[15] A. Sabne, P. Sakdhnagool, and R. Eigenmann, “Heterodoop: A mapreduce programming system for accelerator clusters,” in *HPDC*, 2015, pp. 235–246.

[16] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew, “Deep learning with COTS HPC systems,” in *ICML*, 2013, pp. 1337–1345.

[18] Nvidia, “cuSPARSE,” http://docs.nvidia.com/cuda/cusparse/#cusparse-lt-t-gt-csrmm2, 2015, [Online; accessed 4-Aug-2015].

[19] ——, “cuBLAS,” http://docs.nvidia.com/cuda/cublas/, 2015, [Online; accessed 17-Aug-2015].

[20] X. Meng, J. K. Bradley, B. Yavuz, E. R. Sparks, S. Venkataraman, D. Liu, J. Freeman, D. B. Tsai, M. Amde, S. Owen, D. Xin, R. Xin, M. J. Franklin, R. Zadeh, M. Zaharia, and A. Talwalkar, “MLlib: Machine Learning in Apache Spark,” *CoRR*, vol. abs/1505.06807, 2015.

[21] J. Canny, D. L. W. Hall, and D. Klein, “A multi-teraflop constituency parser using GPUs,” in *EMNLP*, 2013, pp. 1898–1907.

[22] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, and W.-m. W. Hwu, “Optimization principles and application performance evaluation of a multithreaded GPU Using CUDA,” in *PPoPP*, 2008, pp. 73–82.

[23] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng, “Large scale distributed deep networks,” in *NIPS*, 2012, pp. 1223–1231.

[24] B. Li, S. Tata, and Y. Sismanis, “Sparkler: Supporting large-scale matrix factorization,” in *EDBT*, 2013, pp. 625–636.

[25] C.-J. Hsieh and I. S. Dhillon, “Fast coordinate descent methods with variable selection for non-negative matrix factorization,” in *KDD*, 2011, pp. 1064–1072.

[26] X. Cai, Z. Xu, G. Lai, C. Wu, and X. Lin, “GPU-accelerated restricted boltzmann machine for collaborative filtering,” in *ICA3PP*, 2012, pp. 303–316.

[27] D. Zastrau and S. Edelkamp, “Stochastic gradient descent with GPGPU,” in *KI 2012: Advances in Artificial Intelligence*. Springer, 2012, pp. 193–204.

[28] IBM, “General Parallel Filesystem,” http://www-01.ibm.com/support/knowledgecenter/?lang=en#!/SSFKCN_4.1.0.4/gpfs.v4r104_welcome.html, 2014.

[29] Stanford SNAP Lab, “Web data: Amazon reviews,” https://snap.stanford.edu/data/web-Amazon.html, 2015, [Online; accessed 18-Aug-2015].

[30] I. Pillaszy, D. Zibriczky, and D. Tikk, “Fast ALS-based matrix factorization for explicit and implicit feedback datasets,” in *RecSys*, 2010, pp. 71–78.

[31] H. Cui, J. Cipar, Q. Ho, J. K. Kim, S. Lee, A. Kumar, J. Wei, W. Dai, G. R. Ganger, P. B. Gibbons, G. A. Gibson, and E. P. Xing, “Exploiting bounded staleness to speed up big data analytics,” in *USENIX ATC*, 2014, pp. 37–48.

[32] F. Niu, B. Recht, C. Re, and S. J. Wright, “HOGWILD!: A lock-free approach to parallelizing stochastic gradient descent,” in *NIPS*, 2011, pp. 693–701.

[33] Y. Nishioka and K. Taura, “Scalable task-parallel sgd on matrix factorization in multicore architectures,” in *ParLearning*, 2015.

[34] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su, “Scaling distributed machine learning with the parameter server,” in *OSDI*, 2014, pp. 583–598.

[35] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, “Distributed GraphLab: a framework for machine learning and data mining in the cloud,” in *VLDB*, 2012, pp. 716–727.

[36] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, “Cusha: Vertex-centric graph processing on gpus,” in *HPDC*, 2014, pp. 239–252.