CuMF: scale matrix factorization using just ONE machine with 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 widely used in recommendation systems. We present cuMF, a highly-optimized matrix factorization tool with supreme performance on graphics processing units (GPUs) by fully utilizing the GPU compute power and minimizing the overhead of data movement. Firstly, we introduce a memory-optimized alternating least square (ALS) method by reducing discontiguous memory access and aggressively using registers to reduce memory latency. Secondly, we combine data parallelism with model parallelism to scale to multiple GPUs. Results show that with up to four GPUs on one machine, cuMF can be up to ten times as fast as those on sizable clusters on large scale problems, and has impressively good performance when solving the largest matrix factorization problem ever reported.

1 Introduction

Matrix factorization (MF) is a key algorithm in recommender systems [1]. Given a rating matrix R (m by n) with some observed entries and many missing ones, it approximates R by the multiplication of two low-rank matrices X (m by f) and Θ^T (f by n), in the form of $R \approx X \cdot \Theta^T$. With recommendation being pervasive in Internet applications including e-commerce, digital content streaming, and search engine advertising, MF is regarded as one of the best methods for collaborative filtering [1]. The challenges of matrix factorization lie in two aspects: **scale** and **speed**.

1. **Scale.** While many solutions [2, 3, 4] target at medium-sized problems [5], the industry-scale recommendation problems have evolved to two orders of magnitude larger (see Figure 1). As an example, Facebook's MF is with 1 billion users, millions of items and 100+ billion ratings [6]. None of the existing systems except [6] has tried to tackle problems at this scale.
2. **Speed.** MF is used in many online applications where recommendations need to evolve promptly. Approaches including MPI [4], Spark [7, 8] and parameter server [9] tackle extremely large-scale MF problems. However, they all require big (e.g., 50-node) distributed clusters that incur high cost, and are still with suboptimal performance.

Recent advances in the field of deep learning indicates that, a few GPUs can yield similar or better performance of a big distributed CPU cluster [10]. GPU's success in deep learning strongly moti-

*Work done while the author was at IBM.

vates us to pursue **a fast and scalable MF solution on GPUs**. This paper proposes **cuMF** (short for CUDA Matrix Factorization), a novel *scale-up* solution. CuMF uses a handful of GPUs on a single machine to tackle the largest MF problem.

Experimental results show that with up to four Nvidia GPU cards, the performance of cuMF on a single machine can be **up to ten times as fast** as those on big clusters (e.g., with 50 nodes) on large-scale problems, and has impressively good performance while **solving the largest matrix factorization** problem ever reported. In terms of cost, thanks to its faster speed and lower per machine cost, cuMF is merely 1%-5% of the baseline systems compared. As a result, we believe cuMF's speed and cost is very competitive.

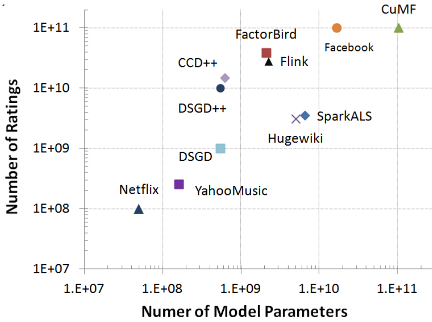

Figure 1: The scale of MF data sets[1]. Y-axis is the N_z of R, and x-axis is $(m + n) \times f$. See Table 2 for more details.

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%**

Table 1: Speed and cost (on cloud) of cuMF on one machine compared with three multi-node CPU systems. Note: Experiment details are in Section 3. NOMAD [4] used m1.xlarge which is now superseded by m3.xlarge by Amazon. Factorbird's node is similar to AWS c3.2xlarge [9]. CPU and GPU systems' cost is calculated by (price per node per hr)*(#nodes)*(execution time), with unit price taken when submitting this paper (AWS price: https://aws.amazon.com/ec2/pricing/; GPU machine price: http://www.softlayer.com/gpu).

2 Problem Definition

CuMF adopts the alternating least square (ALS) algorithm for MF, because it is inherently parallel and able to exploit thousands of GPU cores. ALS is computationally more expensive but requires fewer iterations compared with stochastic gradient descent (SGD), which nicely fits the nature of GPU computation.

2.1 ALS algorithm for matrix factorization

Matrix factorization or completion is to decompose a sparse matrix R with two lower-rank, dense matrices X and Θ, such that $R \approx X \cdot \Theta^T$. Suppose r_{uv} is the non-zero element of matrix R at position (u, v), we want to minimize the following cost function. To avoid overfitting we use weighted-λ-regularization proposed in [5], where n_{x_u} and n_{θ_v} denote the number of total ratings on user u and item v, respectively.

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

Many optimization methods, including Alternating Least Square (ALS) [5], CGD [3], and SGD [2] 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

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

which lead to the following equation to update X:

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

[1] CCD++ [3], DSGD [11], DSGD++ [12], Facebook [6], Factorbird [9], Flink [13], Hugewiki [2], Netflix [5] SparkALS [14], and YahooMusic [15].

together with the following equation to update Θ:

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

ALS updates X and Θ in an alternating manner. The formalism of ALS enables solving the rows of X (and Θ) in parallel so as to harness the power of GPU. In the rest of this paper, we explain our method using update-X. The same method is applicable to update-Θ.

2.2 Challenges and solution of speedy and scalable ALS

Challenge 1. Computation bounded by irregular and intensive memory access. Eq. (2) involves *sparse*, *irregular* and *intensive* memory access. Details are as follow:

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 spread **sparsely** and **discontiguously** across the n columns in Θ^T. For example, in the Netflix data set [5], one user rates around 200 items on average, leading to a discontiguous access of 200 columns among 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.

Solution to Challenge 1. We optimize memory access in ALS, including reducing discontiguous memory access, retaining hotspot variables in faster memory, and aggressively using registers, as to get close to the roofline performance of a single GPU. CuMF attempts to fully harness the computing power of a single GPU. Due to the sparseness of R and the consequent irregular memory access, it is difficult to utilize the massive FLOPS (floating point operations per second) offered by a GPU. We exploit the GPU register file which is larger and has higher bandwidth compared to its shared memory [16]. For example, a Nvidia Maxwell stream multiprocessor has a 256 KB register file and only 96 KB shared memory. In cuMF, we aggressively use registers in sparse matrix multiplication. We had to use macro expansion in C to generate verbose paragraphs of code because CUDA does not allow declaring arrays in a register file.

Challenge 2. Scalability bounded by limited memory capacity.

When problem size get larger, ALS is bounded by the memory capacity of a single GPU. The current Nvidia Maxwell and Kepler GPUs have 12 GB memory per card. Each card would only be able to load 3 billion (3×10^9) single precision floats. However, the smallest data set, i.e., Netflix, in Figure 1, has $m = 480$K. When $f = 100$, m Hermitian matrices are with size $mf^2 = 480\text{K} \times 100^2 = 4.8$ billion floats > 3 billion.

Solution to Challenge 2.

In distributed machine learning, **model parallelism** and **data parallelism** are two common schemes [17]. 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. ALS is inherently suitable for model parallelism, as the updates of each $\mathbf{x}_u$ and θ_v are independent. On top of that, we design a data-parallel approach. CuMF distributes the computation of any single Hermitian matrix $\sum(\theta_v \theta_v^T + \lambda I)$ to multiple GPUs. Instead of transferring all θ_vs to one GPU, it calculates a local $\sum(\theta_v \theta_v^T + \lambda I)$ on each GPU using the local θ_vs, and reduce (aka., aggregate) many local $\sum(\theta_v \theta_v^T + \lambda I)$s later. By this design cuMF is able to solve ALS of any size.

3 Experiments

We compare cuMF with state-of-art distributed solutions including NOMAD [4], Factorbird [9], Spark ALS [14], and a Giraph based solution from Facebook [6].

Data Sets. We use Hugewiki [2] data and compare the convergence speed on test set; we also synthesize the data sets used by SparkALS [14], Factorbird [9] and Facebook [6], and compare the per iteration execution time.

Table 2: Data sets

Data Set	m	n	#ratings (N_z)	f	λ
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

Hardware. Unless otherwise mentioned, we use one to four Nvidia Titan X GPUs, each with 3072 CUDA cores and 12 GB on-chip memory, on one machine. The machine is with two Intel Xeon CPU E5 CPUs, 256 GB RAM, and the GPFS [18] as the file system.

Parameters. The f and λ values for each data set are given in Table 2. Feature matrices are initiated with random numbers in $[0, 1]$. We focus on performance and did not spend much effort in hyper-parameter tuning to improve accuracy.

Performance on 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 [11] and DSGD++ [12]. With CuMF we partition X evenly into four GPUs and apply data parallelism. CuMF performs slightly better than NOMAD on a 64-node HPC cluster (with a slower start because of ALS's heavier iteration), and much better than NOMAD on a 32-node AWS cluster, as shown in Figure 2.

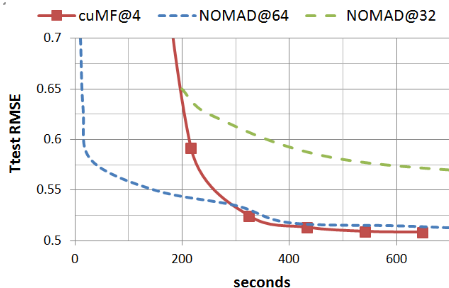

Figure 2: Hugewiki with cuMF@4GPU, vs. NOMAD on a 64-node HPC cluster and a 32-node commodity cluster.

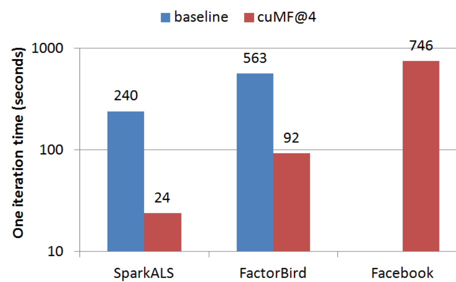

Figure 3: CuMF@4 on very large data sets, compared with their original implementations.

Solve extremely large-scale problems. We conduct experiments on three extremely large problems. 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 3. SparkALS [14] is a benchmark of Spark MLlib ALS with m = 660M, n = 2.4M, f = 10, and N_z = 3.5B. We apply model parallelism when solving X and data parallelism when solving Θ. CuMF finishes an iteration in 24 seconds, which is **ten times as fast** as SparkALS.

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

For the Facebook data set [6], we use data parallelism to solve both X and Θ. cuMF completes one ALS iteration in 746 seconds. [6] did not report its speed on 50 Giraph workers, but we believe cuMF is competitive given the difficulty 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 three 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.

4 Conclusion

Advances in GPU computing inspire us to use them to accelerate large-scale matrix factorization. By optimizing memory access and combining data parallelism with model parallelism, cuMF can be up to ten times as fast as those on sizable clusters on large-scale problems. It also solves the largest matrix factorization problem ever reported in a reasonable run-time. We plan to enhance cuMF to one machine with more GPUs or multiple machines, to deal with even larger data sets in future.

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] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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].

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

[8] 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.

[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] 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.

[11] 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.

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

[13] 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].

[14] 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].

[15] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The Yahoo! Music Dataset and KDD-Cup '11," in *KDD Cup 2011 competition*, 2012.

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

[17] 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.

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