PSL: Exploiting Parallelism, Sparsity and Locality to Accelerate Matrix Factorization on x86 Platforms

Weixin Deng, Pengyu Wang, Jing Wang, Chao Li, Minyi Guo

Shanghai Jiao Tong University
{dengwxn,wpybtw,jing618}@sjtu.edu.cn
{lichao, guo-my}@cs.sjtu.edu.cn

Abstract. Matrix factorization is a basis for many recommendation systems. Although alternating least squares with weighted-λ-regularization (ALS-WR) is widely used in matrix factorization with collaborative filtering, this approach unfortunately incurs insufficient parallel execution and ineffective memory access. Thus, we propose a solution for accelerating the ALS-WR algorithm by exploiting parallelism, sparsity and locality on x86 platforms. Our PSL can process 20 million ratings and the speedup using multi-threading is up to 14.5x on a 20-core machine.

Keywords: Parallelism, Sparsity, Locality, Matrix Factorization, ALS-WR, Acceleration

1 Introduction

There are lots of work on AI benchmarking [13, 14, 16, 20, 23][1]. Some important topics include neural network inference on RISC-V [19] and on Cambricon [21], matrix factorization on x86 [17], 3D face recognition [27] and etc. Matrix factorization is a popular solution to recommendation systems. Many recommendation systems recommend items to users by collaborative filtering based on historical records of items that the users have rated or bought. For example, online shopping websites recommend goods to buyers and movie rating websites recommend movies to users.

Recently, there are lots of research interest in improving the performance of many applications like matrix factorization [6, 8, 15] and graph processing [7, 24]. One way is to accelerate through mathematical methods [6, 15]. Another approach is to optimize on architecture features [7, 8, 24, 26].

Alternating least squares with weighted-λ-regularization (ALS-WR) is proposed by Zhou et al. [28] to solve large-scale collaborative filtering. They use parallel Matlab on a Linux cluster as the experimental platform. Nowadays x86 has powerful vector instructions [1], Intel Math Kernel Library (Intel MKL) [3],

[1] The source code of AIBench is publicly available from http://www.benchcouncil.org/benchhub/AIBench/ (sign up to get access).

and multi-core computing resources. As a result, x86 platforms have shown great potential in performance tuning. Equipped with flexible C++ and OpenMP [5], we implement ALS-WR on x86 platforms by exploiting parallelism, sparsity, and locality to achieve high performance.

The contributions of this paper are as follows:

- PSL, a multi-threaded ALS-WR implementation that scales well on x86.
- Achieving high parallelism by parallelizing loops.
- Reducing memory consumption by leveraging sparsity of input data.
- Improving spatial locality by adopting suitable layouts.

We organize the rest of paper as follows. Section 2 introduces background. Section 3 describes our design and implementation. Section 4 shows our evaluation. Finally, Section 5 concludes this paper.

2 Background

2.1 Alternating-Least-Squares with Weighted-λ-Regularization

ALS-WR is widely used in matrix factorization problems, in which Root Mean Square Error (RMSE) is used to evaluate the convergence. Let $R = \{r_{ij}\}_{n_u \times n_m}$ denotes the user-item matrix, in which r_{ij} represents the rating score of item j rated by user i and is either a real number or missing, n_u is the number of users and n_m is the number of items. Many recommendation systems try to predict the missing r_{ij} values from known ones.

This approach assigns each user and each item a feature vector. Each predicted rating of an item is calculated by the inner product of the corresponding user and item feature vectors. More specifically, let $U = [\boldsymbol{u}_i]$ be the user feature matrix, in which $\boldsymbol{u}_i \in \mathbb{R}^{n_f}$ for $1 \leqslant i \leqslant n_u$, and let $M = [\boldsymbol{m}_j]$ be the item feature matrix, in which $\boldsymbol{m}_j \in \mathbb{R}^{n_f}$ for $1 \leqslant j \leqslant n_i$. Here n_f is the dimension of the feature space.

By introducing a regularization factor $\lambda \in \mathbb{R}$, we define the objective function

$$\begin{aligned} f(U, M) = & \sum_{ij}^{r_{ij} \text{ is known}} (r_{ij} - \boldsymbol{u}_i^{\mathrm{T}} \boldsymbol{m}_j)^2 \\ & + \lambda \left(\sum_i n_{u_i} \|\boldsymbol{u}_i\| + \sum_j n_{m_j} \|\boldsymbol{m}_j\| \right). \end{aligned} \tag{1}$$

The ALS-WR algorithm runs as follows:

Step 1. Initialize matrix M by setting the first row as the average ratings of every item, and assigning small random numbers for the rest.
Step 2. Fix M, solve U by minimizing the objective function.
Step 3. Fix U, solve M by minimizing the objective function similarly.
Step 4. Repeat Steps 2 and 3 until desired convergence of RMSE.

Let I_i^u denotes the set of items j that user i rated and n_{u_i} is the size of I_i^u, and let I_j^m denotes the set of users who rated item j and n_{m_j} is the size of I_j^m. We directly give the solution to minimizing the objective function here. When we fix M and update U, we have

$$\begin{aligned} A_i &= M_{I_i^u} M_{I_i^u}^{\mathrm{T}} + \lambda n_{u_i} E, \\ V_i &= M_{I_i^u} R(i, I_i^u)^{\mathrm{T}}, \\ \boldsymbol{u}_i &= A_i^{-1} V_i, \ \forall i, \end{aligned} \tag{2}$$

where E is the $n_f \times n_f$ identity matrix, $M_{I_i^u}$ denotes the sub-matrix of M where columns $j \in I_i^u$ are selected, and $R(i, I_i^u)$ is the row vector where columns $j \in I_i^u$ of the i-th row of R is taken.

Similarly when we fix U and update M, we have

$$\begin{aligned} A_j &= U_{I_j^m} U_{I_j^m}^{\mathrm{T}} + \lambda n_{m_j} E, \\ V_j &= U_{I_j^m} R(I_j^m, j), \\ \boldsymbol{m}_j &= A_j^{-1} V_j, \ \forall j, \end{aligned} \tag{3}$$

where $U_{I_j^m}$ denotes the sub-matrix of U where columns $i \in I_j^m$ are selected, and $R(I_j^m, j)$ is the column vector where rows $i \in I_j^m$ of the j-th column of R is taken.

After we have N predicted values $\hat{r}_i$ and N real values r_i, we use RMSE to evaluate convergence. RMSE is calculated as

$$\mathrm{RMSE}(\hat{r}, r) = \left(\frac{1}{N} \sum_{i=1}^{N} (\hat{r}_i - r_i)^2 \right)^{1/2}. \tag{4}$$

Optimization Opportunities. Analyzing main components of the ALS-WR algorithm gives us opportunities to have optimizations. There is numerous memory access, while some layouts are less efficient. Besides, there are loops that can be parallelized. Our PSL carefully solves these problems and therefore achieves high performance.

2.2 Supported Techniques

We introduce the following three relative techniques, which are adopted by PSL.

Vector Instructions. Advanced Vector Extensions (AVX) [1] contain a set of instructions for doing Single Instruction Multiple Data (SIMD) operations on x86 architecture. The vector operations use a set of special vector registers. For example, the maximum size of each vector register is 256 bits if the AVX instruction set is available, and 512 bits if the AVX512 instruction set is available. On large-scale data, vector operations are useful when the same operation is performed on multiple data elements and the dataflow allows parallel calculations.

Intel Math Kernel Library (Intel MKL). Intel MKL [3] is a library of optimized math functions. It equips with industry-standard C and Fortran APIs, which is convenient to programmers. For the ALS-WR algorithm, there are lots of performing linear algebra operations and solving linear equations. We use Intel MKL to do mathematical calculations in ALS-WR.

Non-Uniform Memory Access (NUMA). NUMA is a memory architecture for multi-core machines where processors are directly attached to their own local memory. It is fast to access local memory but relatively slower to access remote memory. In contrast, in uniform memory access (UMA) multi-core machines, generally only one processor can control the memory bus at a time. Since processors block the memory bus when accessing memory, it can lead to significant performance degradation as the number of cores increases. There are lots of research interest on NUMA-aware optimizations [9, 10, 22].

3 Design and Implementation

Benefiting from techniques mentioned in Section 2.2, we explore optimizations from parallelism, sparsity and locality on multi-core CPU architecture.

3.1 Optimization Opportunity of ALS-WR Algorithm

Our revised ALS-WR is described in Algorithm 1. It describes the process of updating U and M in one epoch. The function `linalg.solve`(A, B) returns matrix X such that $AX = B$. As suggested in Intel Guide [2], Intel MKL should run on a single thread when called from a threaded region of an application to avoid over-subscription of system resources. So we let Intel MKL run on a single thread, and make loops in lines 1 and 8 be multi-threaded.

Algorithm 1 Updating U and M in one epoch

1: **for** $i = 0$ **to** n_u **do in parallel**
2: $M_i \leftarrow M[:, I_i^u]$
3: $R_i \leftarrow R[i, I_i^u]$
4: $V_i \leftarrow M_i R_i^{\mathrm{T}}$
5: $A_i \leftarrow M_i M_i^{\mathrm{T}} + \lambda n_{u_i} E$
6: $U[:, i] \leftarrow$ `linalg.solve`(A_i, V_i)
7: **end for**
8: **for** $j = 0$ **to** n_m **do in parallel**
9: $U_j \leftarrow U[:, I_j^m]$
10: $R_j \leftarrow R[I_j^m, j]$
11: $V_j \leftarrow U_i R_j^{\mathrm{T}}$
12: $A_j \leftarrow U_j U_j^{\mathrm{T}} + \lambda n_{m_j} E$
13: $M[:, j] \leftarrow$ `linalg.solve`(A_j, V_j)
14: **end for**

3.2 Optimization Opportunity of RMSE Function

According to formula (4), the computation in ALS-WR only involves known ratings. So we should enumerate all known ratings r_{ij} and calculate the difference

between r_{ij} and $\boldsymbol{u}_i^{\mathrm{T}}\boldsymbol{m}_j$, rather than first calculate $U^{\mathrm{T}}M$. In this way, we can save unnecessary overheads on unknown ratings. Similar to ALS-WR, we use multi-threading on the enumerating loop.

Our revised RMSE is presented in Algorithm 2. For the summation variable s in line 6, maintaining thread local copies of s and adding them up at last can avoid atomic operations.

Algorithm 2 Calculating RMSE

1: $s \leftarrow 0$
2: **for** r_{ij} **do in parallel**
3: $\quad U_i \leftarrow U[:, i]$
4: $\quad M_j \leftarrow M[:, j]$
5: $\quad \hat{r}_{ij} \leftarrow U_i^{\mathrm{T}} M_j$
6: $\quad s \leftarrow s + (\hat{r}_{ij} - r_{ij})^2$
7: **end for**
8: **return** $(s/N)^{1/2}$

3.3 Sparsity and Locality Opportunity

First, since operations in lines 3 and 10 only access known ratings in rows and columns, R should be stored as a sparse matrix with compressed sparse row (CSR) and compressed sparse column (CSC) formats. By leveraging sparsity of input data, we reduce memory consumption. Second, U and M are feature matrixes and thus dense matrixes. As operations in lines 2, 6, 9 and 13 access columns of U and M, they should be stored in column-major order. These layouts of R, U and M improve spatial locality by increasing cache hits.

In all multi-threaded loops, the default first touch placement policy of NUMA allocates all new data in the memory closest to the loop thread [25], which is beneficial for memory access. For different threads, there are no race conditions on global data. Thus, no synchronization on global data is needed.

4 Evaluation

Experimental Setup. Our experiments are performed on a 20-core (2 sockets, each with 10 cores) machine with Intel Xeon Silver 4114 (hyper-threaded, 40 threads in total). The version of Intel Compiler and Libraries is 2019.4. The compiler options are `-mkl -O3 -qopenmp -std=c++17`. We also use Intel Vtune Amplifier 2019.4 to analyze core utilization and hotspots. The training parameters of ALS-WR are set as follows. The dimension of feature matrixes n_f is 100. The number of epochs is 30. The train-test data ratio is 4:1.

Data. MovieLens [4] is rating data collected from MovieLens website. We use ml-20m as our input data, which has 20 million ratings and 465,000 tag applications applied to 27,000 movies by 138,000 users.

4.1 Scalability

Fig. 1 shows the speedup of execution time using different number of threads. As we can see, performance scales linearly as more threads are added (except the 40 threads hyper-threaded from 20 cores). The speedup is up to 14.5x when using 40 threads.

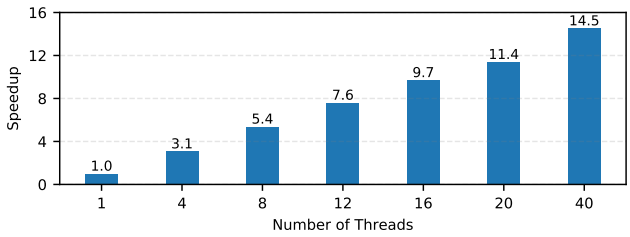

Fig. 1. Speedup of Multi-threaded ALS-WR

Fig. 2 shows the core utilization using different number of threads. It represents how efficiently the application utilized the available CPU cores and helps evaluate the parallel efficiency. As the number of threads rises from 1 to 20, the core utilization is increasing.

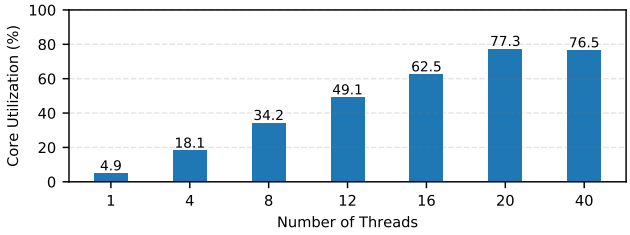

Fig. 2. Core Utilization of Multi-threaded ALS-WR

These show our implementation scales well. Besides, Intel hyper-threading is most effective when each thread performs different types of operations and there are under-utilized resources on the processor. However, Intel MKL uses most of the available resources and performs identical operations on each thread [2]. So we see the performance does not improve much when switching from 20 threads to 40 threads.

4.2 Hotspots

The top five hotspots of the 40-threaded ALS-WR are shown in Table 1. The first two functions are matrix multiplications and solving linear equations respectively. The overheads of the third function occur when accessing columns of dense matrixes. As for the fourth and the fifth, they relate to memory allocation. Therefore, further optimizing memory access and memory management might help gain better performance.

Table 1. Hotspots of the 40-threaded ALS-WR

Hotspots	Function	CPU Time
1	`cblas_sgemm`	36.1%
2	`LAPACKE_sgesv_work`	23.7%
3	`__intel_avx_rep_memcpy`	10.8%
4	`operator new`	7.5%
5	`[MKL SERVICE]@malloc`	7.1%

4.3 Speedup of Different Optimization

Fig. 3 shows the speedup of different optimization. The weakest baseline only implements sparse matrix function and is single threaded. Firstly, improving spatial locality brings 1.8x speedup. Secondly, using multi-threading with atomic operations offers 4.9x speedup. At last, when we replace atomic operations with thread local copies, we get a 2.9x speedup. Thus, it is vital to remove atomic operations. In total, using multi-threading provides up to 14.5x speedup.

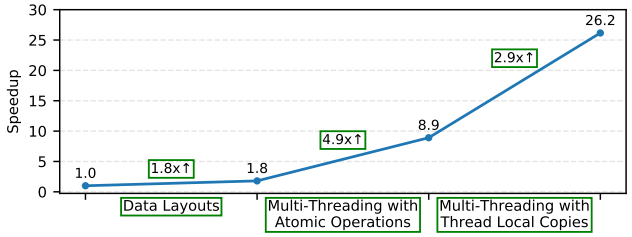

Fig. 3. Speedup of Different Optimization

5 Conclusion

This paper proposes a high performance multi-threaded implementation PSL for matrix factorization. We exploit parallelism, sparsity and locality for acceleration

on x86 platforms. In particular, We analyze the speedup of different optimization. In the future, we will research on how to optimize memory access and memory management to gain better performance.

References

1. Advanced vector extensions, `https://en.wikipedia.org/wiki/Advanced_Vector_Extensions`
2. Intel guide for developing multithreaded applications, `https://software.intel.com/sites/default/files/m/d/4/1/d/8/GDMA_2.pdf`
3. Intel math kernel library, `https://software.intel.com/en-us/mkl`
4. Movielens, `https://grouplens.org/datasets/movielens/`
5. Openmp, `https://en.wikipedia.org/wiki/OpenMP`
6. Ang, A.M.S., Gillis, N.: Accelerating nonnegative matrix factorization algorithms using extrapolation. Neural Computation **31**(2), 417–439 (Feb 2019). https://doi.org/10.1162/neco_a_01157
7. Balaji, V., Lucia, B.: Combining data duplication and graph reordering to accelerate parallel graph processing. In: Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing. pp. 133–144. HPDC '19, ACM, New York, NY, USA (2019). https://doi.org/10.1145/3307681.3326609
8. Chen, J., Fang, J., Liu, W., Tang, T., Chen, X., Yang, C.: Efficient and portable als matrix factorization for recommender systems. In: 2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW). pp. 409–418 (May 2017). https://doi.org/10.1109/IPDPSW.2017.91
9. Chen, S., Fang, J., Chen, D., Xu, C., Wang, Z.: Adaptive optimization of sparse matrix-vector multiplication on emerging many-core architectures. In: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS). pp. 649–658 (June 2018). https://doi.org/10.1109/HPCC/SmartCity/DSS.2018.00116
10. Elafrou, A., Karakasis, V., Gkountouvas, T., Kourtis, K., Goumas, G., Koziris, N.: Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms. ACM Trans. Math. Softw. **44**(3), 26:1–26:32 (Jan 2018). https://doi.org/10.1145/3134442
11. Eyerman, S., Heirman, W., Bois, K.D., Fryman, J.B., Hur, I.: Many-core graph workload analysis. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis. pp. 22:1–22:11. SC '18, IEEE Press, Piscataway, NJ, USA (2018). https://doi.org/10.1109/SC.2018.00025
12. Fog, A.: Optimizing software in c++, `https://www.agner.org/optimize/optimizing_cpp.pdf`
13. Gao, W., Luo, C., Wang, L., Xiong, X., Chen, J., Hao, T., Jiang, Z., Fan, F., Du, M., Huang, Y., Zhang, F., Wen, X., Zheng, C., He, X., Dai, J., Ye, H., Cao, Z., Jia, Z., Zhan, K., Tang, H., Zheng, D., Xie, B., Li, W., Wang, X., Zhan, J.: Aibench: towards scalable and comprehensive datacenter ai benchmarking. In: 2018 BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18). pp. 3–9. Springer (2018)
14. Gao, W., Tang, F., Wang, L., Zhan, J., Lan, C., Luo, C., Huang, Y., Zheng, C., Dai, J., Cao, Z., Tang, H., Zhan, K., Wang, B., Kong, D., Wu, T., Yu, M., Tan, C., Li, H., Tian, X., Li, Y., Lu, G., Shao, J., Wang, Z., Wang, X., Ye, H.:

Aibench: An industry standard internet service ai benchmark suite. arXiv preprint arXiv:1908.08998 (2019)
15. Gillis, N., Glineur, F.: Accelerated multiplicative updates and hierarchical als algorithms for nonnegative matrix factorization. Neural Comput. **24**(4), 1085–1105 (Apr 2012). https://doi.org/10.1162/NECO_a_00256
16. Hao, T., Huang, Y., Wen, X., Gao, W., Zhang, F., Zheng, C., Wang, L., Ye, H., Hwang, K., Ren, Z., Zhan, J.: Edge aibench: Towards comprehensive end-to-end edge computing benchmarking. 2018 BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18) (2018)
17. Hao, T., Zheng, Z.: The implementation and optimization of matrix decomposition based collaborative filtering task on x86 platform. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer (2019)
18. Hollowell, C., Caramarcu, C., Strecker-Kellogg, W., Wong, T., Zaytsev, A.: The effect of numa tunings on cpu performance, `https://indico.cern.ch/event/304944/contributions/1672535/attachments/578723/796898/numa.pdf`
19. Hou, P., Yu, J., Miao, Y., Tai, Y., Wu, Y., Zhao, C.: Rvtensor: A light-weight neural network inference framework based on the risc-v architecture. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer (2019)
20. Jiang, Z., Gao, W., Wang, L., Xiong, X., Zhang, Y., Wen, X., Luo, C., Ye, H., Lu, X., Zhang, Y., Feng, S., Li, K., Xu, W., Zhan, J.: Hpc ai500: A benchmark suite for hpc ai systems. 2018 BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18) (2018)
21. Li, G., Wang, X., Ma, X., Liu, L., Feng, X.: Xdn: Towards efficient inference of residual neural networks on cambricon chips. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer (2019)
22. Li, S., Hoefler, T., Snir, M.: Numa-aware shared-memory collective communication for mpi. In: Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing. pp. 85–96. HPDC '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2493123.2462903
23. Luo, C., Zhang, F., Huang, C., Xiong, X., Chen, J., Wang, L., Gao, W., Ye, H., Wu, T., Zhou, R., Zhan, J.: Aiot bench: Towards comprehensive benchmarking mobile and embedded device intelligence. 2018 BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench18) (2018)
24. Mukkara, A., Beckmann, N., Abeydeera, M., Ma, X., Sanchez, D.: Exploiting locality in graph analytics through hardware-accelerated traversal scheduling. In: Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture. pp. 1–14. MICRO-51, IEEE Press, Piscataway, NJ, USA (2018). https://doi.org/10.1109/MICRO.2018.00010
25. van der Pas, R.: How to befriend numa, `https://www.openmp.org/wp-content/uploads/SC18-BoothTalks-vanderPas.pdf`
26. Wang, P., Zhang, L., Li, C., Guo, M.: Excavating the potential of gpu for accelerating graph traversal. In: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 221–230 (May 2019). https://doi.org/10.1109/IPDPS.2019.00032
27. Xiong, X., Wen, X., Huang, C.: Improving rgb-d face recognition via transfer learning from a pretrained 2d network. In: International Symposium on Benchmarking, Measuring and Optimization (Bench'19). Springer (2019)
28. Zhou, Y., Wilkinson, D., Schreiber, R., Pan, R.: Large-scale parallel collaborative filtering for the netflix prize. In: Proceedings of the 4th International Conference on Algorithmic Aspects in Information and Management. pp. 337–348. AAIM

'08, Springer-Verlag, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68880-8_32