Efficient and Portable ALS Matrix Factorization for Recommender Systems

Jing Chen*, Jianbin Fang*, Weifeng Liu†, Tao Tang*, Xuhao Chen* and Canqun Yang*
*Software Institute, College of Computer, National University of Defense Technology, Changsha, China
Email: jingchen95@yeah.net, {j.fang, taotang84, chenxuhao, canqun}@nudt.edu.cn
†Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark
Email: weifeng.liu@nbi.ku.dk

***Abstract*—*Alternating least squares* (ALS) has been proved to be an effective solver of matrix factorization for recommender systems. To speedup factorizing performance, various parallel ALS solvers have been proposed to leverage modern multi-core CPUs and many-core GPUs/MICs. Existing implementations are limited in either speed or portability (constrained to certain platforms). In this paper, we present an efficient and portable ALS solver for recommender systems. On the one hand, we diagnose the baseline implementation and observe that it lacks the awareness of the hierarchical thread organization on modern hardware. To achieve high performance, we apply the thread batching technique and three architecture-specific optimizations. On the other hand, we implement the ALS solver in OpenCL so that it can run on various platforms (CPUs, GPUs, and MICs). Based on the architectural specifics, we select a suitable code variant for each platform to efficiently mapping it to the underlying hardware. The experimental results show that our implementation performs 5.5× faster on a 16-core CPU and 21.2× faster on K20c than the baseline implementation. Our implementation also outperforms `cuMF` on various datasets.**

***Keywords*-Matrix factorization; Alternating least squares; Performance**

I. Introduction

In a recommender system, we aim to build a model by training with observed incomplete rating data (i.e., a user's preference over all items) and then predict his/her preference over items not rated [1]. Among the recommendation approaches, *matrix factorization* was empirically shown to be a better solution than traditional nearest-neighbour approaches in the Netflix Prize competition [2]. Since then, there has been a large amount of work dedicated to the design of fast and scalable methods for large-scale matrix factorization problems [3], [1], [4].

Among the matrix factorization techniques, *alternating least squares* (ALS) has been proved to be an effective one [1]. Compared to *stochastic gradient descent* (SGD) [5], [6], the ALS algorithm is not only inherently parallel, but can incorporate implicit ratings [1]. Nevertheless, the ALS algorithm involves parallel sparse matrix manipulation [7] which is challenging to achieve high performance due to imbalanced workload [8], [9], random memory access [10] and task dependency [11]. This particularly holds when parallelizing and optimizing ALS on modern multi-/many-cores. To address the issue, researchers have investigated various solutions. In [12], Rodrigues et al. present a CUDA-based ALS implementation on GPU, which is claimed to run faster than the implementation on a multi-core CPU. In [13], Tan et al. provides a CUDA-based matrix factorization library (cuMF). It uses various techniques to maximize the performance on multiple GPUs.

In spite of the common efforts, these solutions are still very limited in speed and portability. In terms of speed, we observe that the CUDA implementation on K20c runs much slower than the OpenMP implementation on a 16-core CPU (Figure 1). We argue that this is possibly because the parallel ALS code has been mapped to the massive cores in an inappropriate manner. According to the architectural specifics, converting the code into a right form is highly required. In terms of portability, the available implementations are often limited to vendor-specific platforms. Running the code on emerging hardware often needs from-scratch code engineering. The two motivating observations are further detailed in Section II-C.

In this paper, we present an efficient and portable ALS solver. On the one hand, we diagnose the baseline implementation and observe that it is lack of awareness of the hierarchical thread organization on modern hardware. This leads to an inefficient use of hardware resources: *unbalanced thread use* and *scattered memory access*. Thus, we apply the thread batching technique and three architecture-specific optimizations to mine the hardware potentials. On the other hand, we implement the ALS solver in OpenCL so that it can run on various platforms (CPUs, GPUs, and MICs). Based on the architectural specifics, we select a suitable code variant for each platform to efficiently map it to the underlying hardware. The experimental results show that our implementation performs 5.5× faster on E5-2670 and 21.2× faster on K20c than the baseline implementation. Our implementation also outperforms `cuMF` for various datasets (`Netflix`, `Movielens`, `YahooMusic R1`, and `YahooMusic R4`).

To summarize, we make the following contributions.

- We present an efficient and portable ALS recommender system by applying the thread batching parallelization technique and the architecture-specific optimizations.
- We implement the recommender system with OpenCL and customize code variants for different architec-

978-1-5386-3408-0/17 $31.00 © 2017 IEEE
DOI 10.1109/IPDPSW.2017.91

tures. The portable implementation facilitates us to enable/disable an optimization in an easy way.

- We evaluate the ALS solver on various platforms (CPU, GPU and MIC) and datasets, and demonstrate that our ALS solver is an efficient and portable one.

The remainder of this paper is organized as follows. Section II describes the background and the motivation. We present our approach in Section III and evaluate it in Section IV and Section V. Section VI lists the related work and Section VII concludes our work.

II. BACKGROUND

In this section, we describe the matrix factorization problem and the ALS algorithm. Then we present the motivation of our work with two observations.

A. Problem Definition

The input of matrix factorization is a relation matrix between users and items, $R(m \times n)$, where m denotes the number of users and n denotes the number of items. Due to the sparsity of R, matrix factorization maps both users and items to a joint factor space of dimensionality k, a.k.a. *latent factor*, so that predicting unknown ratings can be estimated by the inner products of two vectors, x_u of matrix $X(m \times k)$ and y_i of matrix $Y(n \times k)$,

$$r_{ui} = x_u {y_i}^T, \tag{1}$$

where x_u denotes the extent of user's interest on items. Similarly, y_i denotes the extent to which the item owns these factors, r_{ui} denotes an entry of the rating matrix R. The key of the problem is how to obtain x_u and y_i so that $R \approx XY^T$. The basic idea for matrix factorization is to minimize the regularized squared error on the observed ratings to learn the factors,

$$L(X, Y) = \sum_{u,i \in \Omega} (r_{ui} - x_u^T y_i)^2 + \lambda(|x_u|^2 + |y_i|^2), \tag{2}$$

where Ω is the known nonzero ratings of R, and x_u^T are the u^{th} row vectors of the matrix X, y_i are i^{th} column vectors of matrix Y, the constant λ is the regularized coefficient to avoid over-fitting. Therefore, the key to solve this problem is to find approaches of getting the matrices X and Y.

B. The ALS Algorithm

Alternating least squares (ALS) is an efficient matrix factorization technique for recommender systems. Because Function 2 is not convex, the minimization principle of alternating least squares is *to keep one fixed while calculating the other*: we fix Y matrix to calculate X matrix to get vectors x_u, and vice versa. In this way, the problem becomes a quadratic function. The procedure iterates until it converges. First, we minimize the equation over X while fixing Y, and the function becomes

Algorithm 1 The ALS algorithm

procedure ALS(R, k, λ; X, Y)
  $X \leftarrow 0$, $Y \leftarrow random\ initial\ guess$
  repeat
    for row $u \leftarrow 1, m$ **do**
      $x_u \leftarrow (Y^T Y + \lambda I)^{-1} Y^T r_u$
    end for
    for column $i \leftarrow 1, n$ **do**
      $y_i \leftarrow (X^T X + \lambda I)^{-1} X^T r_i$
    end for
  until $reached\ max\ iterations$
end procedure

$$L(X) = \sum_{i \in \Omega_u} (r_{ui} - x_u^T y_i)^2 + \lambda |x_u|^2 \tag{3}$$

By calculating the partial derivative of x_u in Function 3 and letting the partial derivative equal zero, we can obtain

$$x_u = (Y^T Y + \lambda I)^{-1} Y^T r_u, \tag{4}$$

where I is the unit matrix ranked k, and r_u is the u^{th} rows of R. In the same way, we can obtain y_i

$$y_i = (X^T X + \lambda I)^{-1} X^T r_i. \tag{5}$$

The ALS algorithm is shown in Algorithm 1. We initialize Y with small random numbers instead of zeros when starting to update the X matrix. The algorithm iterates until it reaches the maximum specified cycles or error rate.

C. Motivation

When running the parallel ALS implementation on multi-/many-cores [12], we have the following two observations.

Observation 1: *ALS on CPUs runs faster than on GPUs.*

Thanks to a larger memory bandwidth and more hardware cores, using GPUs can often bring a much better performance than using a traditional multi-core CPU. This particularly holds for the data-intensive codes such as the ALS solver. However, we observe that this is not necessary the case. Figure 1 compares the performance of ALS on a 16-core CPU and on a K20c GPU. We see that ALS runs, on average, $8.4\times$ faster on the CPU than on the GPU. This unsatisfactory performance of the current implementation leads us to restructure the algorithm and customize optimizations according to the architectural specifics.

Observation 2: *The current implementation cannot run on the coprocessors such as Intel Xeon Phi.*

Nowadays platforms often incorporate specialized processing capabilities (e.g., GPUs, MICs, FPGAs and DSPs) to handle particular tasks. Adding the specialized units gains performance or energy efficiency. However, using such platforms is challenging. In particular, programmers

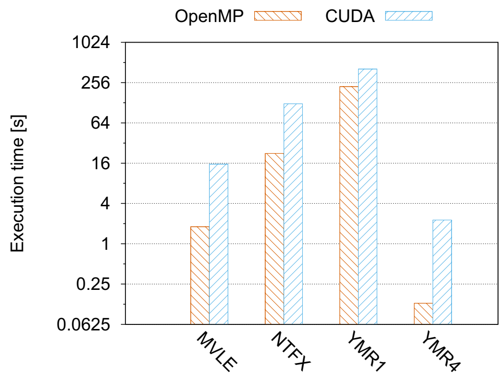


Figure 1. Performance comparison of an OpenMP implementation on a 16-core CPU versus a CUDA implementation on K20c.

have to use vendor-specific programming interface to exploit the diversity. This is the same for the ALS recommender systems, i.e., the OpenMP version of ALS can run only on the traditional multi-core CPUs, while the CUDA version is constrained to NVIDIA GPUs. The current implementation cannot be offloaded to run on Intel Xeon Phi and/or FPGAs. Porting it, which requires restructuring the code from scratch, is time-consuming and error-prone. Thus, a portable recommender system is required. Further, a simple code rewriting in portable programming interfaces such as OpenCL will again lead to a poor hardware utilization. Speed and portability need to be taken into account as a whole.

III. Design and Implementation

In this section, we give the baseline design of ALS and then present our approach. We customize the optimization techniques for different architectures and detail how to select an appropriate code variant.

A. Baseline Design

In [12], Rodrigues et al. present an ALS solver in CUDA and OpenMP, which is taken as our baseline implementation. Algorithm 2 illustrates the algorithm skeleton. Since updating X is similar to updating Y, we only show the former part. Lines 6–7 calculate $(Y^TY + \lambda I)$ and `smat` (a matrix sized of $k \times k$) is introduced to store the temporary results. Lines 8–15 evaluate $Y^T r_u$ which is stored temporally in a vector `svec` sized of k. The baseline implementation employs the Cholesky method to factorize `smat` shown in Line 16 and evaluates the current row (x_u) in Line 17. For the baseline design, each thread updates a row x_u or a column y_i. In total, we have m (or n) tasks and at most m (or n) threads can run concurrently.

Notation. To save memory space, we use the *compressed sparse row* (CSR) form to store the sparse rating matrix R. Three arrays are introduced to represent the original matrix: a *value array* stores the nonzero elements of R

Algorithm 2 The Baseline ALS algorithm (updating X).

1: **procedure** Update_X_over_Y(R, X, Y, k, λ; X)
2: **for** $u \leftarrow 1, m$ **do** ▷ Foreach row
3: $x_u \leftarrow GetBaseAddr(X, u, k)$
4: $omegaSize \leftarrow CountNonZeros(R, u)$
5: **if** $omegaSize > 0$ **then**
6: $smat \leftarrow Y^TY$ ▷ smat: sub-matrix
7: $smat \leftarrow smat + \lambda I$
8: **for** $c \leftarrow 0, k$ **do**
9: **for** $idx \leftarrow row\_ptr[u], row\_ptr[u+1]$ **do**
10: $idx2 \leftarrow colMajored\_sparse\_id[idx]$
11: $idx3 \leftarrow col\_idx[idx] \times k + c$
12: $svec[c] \leftarrow svec[c] + R[idx2] \times Y[idx3]$
13: ▷ svec: sub-vector
14: **end for**
15: **end for**
16: $LL^T \leftarrow smat$ ▷ with Cholesky
17: solve $LL^T\mathbf{x} = svec$ for $\mathbf{x}$
18: **end if**
19: **end for**
20: **end procedure**

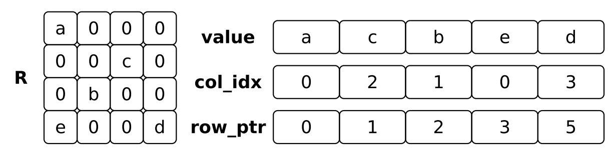


Figure 2. An example of the compressed sparse row storage (CSR) format. The matrix R has 5 rating scores out of 16 elements and three data structures are used in the representation.

in a row-major manner, and its size equals the number of nonzero elements; a *col_idx array* stores the column index of each nonzero element in R, and its size equals the number of nonzero elements; and a *row_ptr array* stores the index of each row's first element, and its size is the number of rows plus 1. Figure 2 illustrates the structure of CSR. The data structures (*value*, *col_idx*, *row_ptr*) are introduced to represent the rating matrix R (See Lines 8–15 of Algorithm 2). Note that we use the *compressed sparse column* (CSC) format when updating y_i. The CSC representation is similar to that of CSR, except that CSC stores the nonzero entries in a column-major manner.

B. Thread Batching Parallelization

As shown in Algorithm 2, the baseline implementation uses one thread to update a row of X or a column of Y. This straightforward implementation can provide sufficient parallelism to utilize the massive hardware threads on GPUs, MICs or multi-core CPUs. Nevertheless, the baseline implementation is unaware of the hierarchical thread organization (i.e., the two-level parallelism) of modern hardware architectures, which results in two major issues: unbalanced thread use and scattered memory access [14].

On the one hand, threads are organized in a hierarchical fashion on modern many-core architectures (Figure 4). On

```
float sum[k*k]={0};
for (int i = lx; i < k; i+=ws){
  for (int j = i; j < k; j++) {
    for (int z = 0; z < omegaSize; z++){
      int d = col_idx[row_ptr + z] * k;
      sum[i*k+j] += Y[d + i] * Y[d + j];
    }
    smat[(j*k)+i] = sum[i*k+j];
    smat[(i*k)+j] = sum[i*k+j];
  }
}
```

(a) Original code.

```
float sum0=0,sum1=0,sum2=0,sum3=0,sum4=0;
for (int z = 0; z < omegaSize; z++){
  int d = col_idx[row_ptr + z] * k;
  if(0<=lx<k) sum0 += Y[d + lx] * Y[d + 0];
  if(1<=lx<k) sum1 += Y[d + lx] * Y[d + 1];
  if(2<=lx<k) sum2 += Y[d + lx] * Y[d + 2];
  if(3<=lx<k) sum3 += Y[d + lx] * Y[d + 3];
  if(4<=lx<k) sum4 += Y[d + lx] * Y[d + 4];
  ...
}
// updating the smat matrix
```

(b) Unrolling the code.

Figure 3. An example of unrolling the code to calculate Y^TY. lx is the local work-item index, ws is the work-group size, k denotes the latent factor, $omegaSize$ is the number of non-zero entries of the current row, $smat$ is the allocated matrix to store temporary results, col_idx and row_ptr are the structures introduced in Figure 2. This example is the case when $k = 5$.

GPUs, a warp of threads are organized into a *SIMT core* (i.e., Streaming Multiprocessor, SM). When the threads diverge (i.e., follow different paths), they are serialized. Meanwhile, the threads from different groups can run concurrently. On CPUs or MICs, a group of fine-grained threads are to be vectorized/packed into a vector core thanks to the compilers or manual efforts. The threads within a group are similarly serialized when they diverge. For a typical recommender dataset, the number of nonzeros varies over rows/columns. When two neighbouring threads updating two continuous rows/columns, it is likely that the thread on the longer row takes more time while the other thread stays idle. The problem becomes severe when the length of rows/columns is significantly uneven, leading to unbalanced thread use.

On the other hand, this baseline implementation accesses the off-chip memory in an inefficient manner. On GPUs, the threads within a workgroup prefer accessing data elements near each other, i.e., coalesced memory accesses. On CPUs/MICs, the memory accessing requests are performed in a cacheline granularity. For the baseline implementation, each thread calculates a matrix ($smat$ sized of $k \times k$) and a vector (sized of k). Thus, the distance between two accesses is at least $(k+1) \times k$. The uncoalesced scattered accesses by neighbouring threads lead to a poor bandwidth utilization.

Therefore, we apply the *thread batching* technique and let a SIMT/SIMD core update a row or a column of ALS. This can not only avoid unbalanced thread use but batch the data accessing requirements. The thread batching technique is applicable on CPUs, GPUs, and MICs.

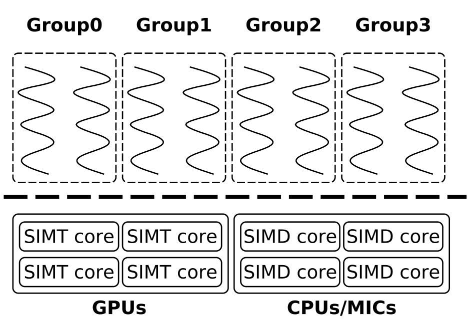


Figure 4. An illustration of a flat thread organization and a hierarchical hardware organization. The GPU cores are in a *SIMT* form while the CPU/MIC cores are in a *SIMD* form.

C. Architecture-Specific Optimizations

CPUs, GPUs and MICs share a lot in common, but they differ in many details. To exploit such details, we need to customize optimizations according to the architectural differences. In this section, we investigate the architecture-oriented optimization techniques.

1) Using Registers: The recent GPUs feature abundant registers with a very small accessing latency. For example, each SM of K20c has 256 KB registers and this architecture increases the maximum number of registers addressable per thread from 63 to 255. Factorizing rating matrix is a typical bandwidth-limited kernel. Thus, an efficient utilization of these registers can improve the kernel performance. When calculating Y^TY (Line 6 of Algorithm 2), the original code uses a private array ($sum[k * k]$) to store the temporary results before updating $smat$ (Figure 3). Depite that the structure is private to a thread, register spilling occurs with a large k. We observe that allocating a $k * k$ buffer per thread is not required. In fact, a buffer sized of k is sufficient. The restructured code is shown in Figure 3(b).

2) Using the Scratch-pad Memory: Compared with the off-chip memory, the scratch-pad memory, which is termed *local memory* in OpenCL, is a high-speed memory unit located on-chip. Staging data with scratch-pads can enhance performance by (1) data reusing, and/or (2) increasing the data moving bandwidth between the off-chip memory space and the on-chip memory space [15].

As shown in Algorithm 2 (Lines 8–15), calculating $Y^T r_u$ needs to load data from R (i.e., the *value* array) and Y. Specifically, updating $svec$ of the row r_u requires the columns of Y identified by the non-zero elements in r_u. Due to the sparsity of R, the data columns are often not contiguous. Thus, staging the data columns is necessary. Figure 5 shows we allocate a local memory buffer (3×5) to cache the required data columns of Y. At the same time, updating $svec$ requires all the non-zero entries of the current row. Loading them into the scratch-pad will improve data sharing for the threads within a workgroup. Figure 5 shows

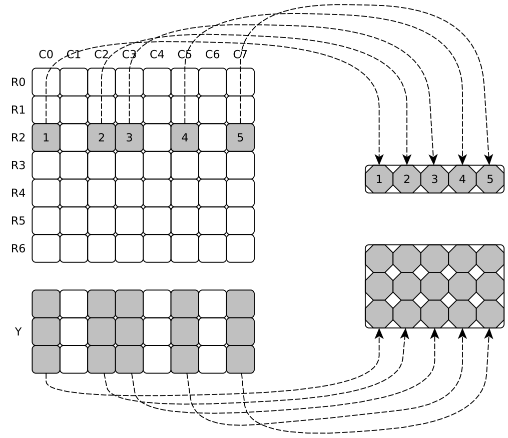


Figure 5. Using local memory to stage the R and Y matrix. R is sized of 7×8, and Y is sized of 3×8 when $k = 7$.

how a local memory vector is allocated to store all the non-zero entries of r_u.

3) Using Vector Units: Both the traditional multi-core CPUs and Intel MIC have vector cores. Merely relying on compilers is difficult to fully use the vector units and explicit vectorization is often required [16]. OpenCL provides *vector data types* to exploit the vector cores, e.g., `float16` is a vector containing 16 scalar data elements typed of `float`. The arithmetic operators can perform the corresponding operations in an element-wise manner. We use `vload` to fill vectors while using `vstore` to write results to memory.

D. Code Variant Selection

Code variants represent alternative implementations of a computation. Each code variant has the same interface, and is functionally equivalent to the other variants but may employ fundamentally different algorithms or implementation strategies [17], [18]. Based on the thread batching version, we will yield 8 versions of code variants by individually applying different optimization techniques or combining them. To achieve high performance, it is necessary to select the most appropriate implementation for a specific execution context (target architecture and input dataset).

In this context, we use an empirical approach to select a right code variant. In total, we provide 8 code variants of the ALS solver by combining different optimizations. Evaluating different code variants and various datasets shows the optimization has an 'unpredictable' impact on the factorization performance (Figure 6). For example, due to the missing scratch-pad on CPU/MIC, using local memory cannot theoretically bring a performance increase on CPU/MIC. But our evaluation results show that using local memory gives a performance boost on these two architectures. This 'unpredictable' performance motivates us to use a machine-learning based approach to select a code variant in future.

IV. Experimental Setup

In this section, we introduce the hardware and software configurations used in the context. We also present the details of the datasets used to evaluate our implementation.

A. Platform Configurations

We use three multi-/many-core platforms in the experiment: Intel Xeon CPU, NVIDIA Tesla GPU and Intel MIC, where the GPU and the MIC are connected to the CPU with different PCIe slots. The Intel CPU is a dual-socket Intel Xeon E5-2670, each with 8 cores running at 2.60GHz. The NVIDIA GPU is Tesla K20c, which contains 13 streaming multiprocessors (SM), and 192 CUDA cores on each SM. The Intel Many Integrated Cores (MIC) is Intel Xeon Phi 31SP, with 57 cores and 6GB global memory.

Our ALS solver is implemented in OpenCL (v1.2) and is then installed on the experimental platforms. The OpenCL implementations for the three devices are from their vendors respectively. The host CPU runs Redhat Linux (v7.0) and uses GCC (v4.9.2), while the MIC coprocessor runs a customized uOS (v2.6.38.8). Intel MPSS (v3.6) is used as the driver and the communication backbone between the host and the coprocessor. The Intel OpenCL SDK for both CPU and MIC is of version `14.1_x64_4.5.0.8`. Also, we use NVIDIA CUDA (v7.5) for Tesla K20c to run the `cuMF` code and the baseline code.

B. Input Datasets

We use four datasets (`Movielens`[1], `Netflix`[2], `YahooMusic R1`, and `YahooMusic R4`[3]) to measure the factorization performance. The format of each dataset is

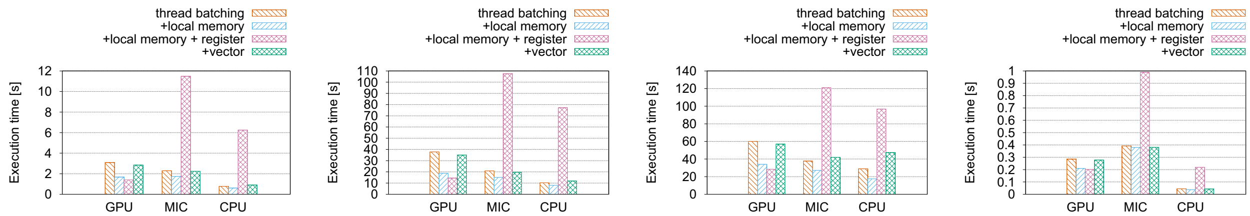
$$< userID, itemID, rating > .$$

We preprocess each dataset according to this format. The details of the four datasets are shown in Table I. m is the number of users, n is the number of items, and N_z is the number of non-zero entries in the rating matrix R. In the context, $k = 10$ and $\lambda = 0.1$ unless otherwise specified.

Table I
DATASETS

	Abbr.	m	n	Training N_z
Movielens10M	MVLE	71567	65133	8000044
NetFlix	NTFX	480189	17770	99072112
YahooMusic R1	YMR1	1948882	98212	115248575
YahooMusic R4	YMR4	7642	11916	211231

[1]http://files.grouplens.org/datasets/movielens/
[2]http://www.select.cs.cmu.edu/code/graphlab/datasets/
[3]http://webscope.sandbox.yahoo.com



(a) Movielens (b) Netflix (c) YahooMusic R1 (d) YahooMusic R4

Figure 6. A performance comparison of the ALS solver on different architectures and datasets. We use the thread configuration of 8192×32 and 5 iterations, while $k = 10$.

V. Performance Results

In this section, we first show how our ALS solver performs by comparing with the state-of-the-art implementations. Then we evaluate the performance impact of the optimization techniques and how we apply optimizations. We also demonstrate the performance results on the three platforms and the performance sensitivity to thread blocks.

A. Comparing with State-of-the-Art

We compare the performance of our ALS implementation with that in SAC15 [12] and HPDC16 [13]. On the E5-2670 CPU, our implementation runs $5.5\times$ faster than the SAC15 OpenMP implementation, while it runs $21.2\times$ faster on the K20c GPU. This significant performance improvement comes from the usage of the thread batching parallelization and the architecture-specific optimizations.

Compared with the HPDC16 implementation, we also notice a remarkable speedup ranging from $2.2\times$ to $6.8\times$. The performance differences are due to several factors. First, we observe that the latent factor k has an impact on the overall performance. The HPDC16 implementation has been specially tuned for the $k = 100$ case, while it is a generic one for the other cases. Second, the HPDC16 implementation employs the `cusparse` library (e.g., `cusparseScsrmm2` and `cublasSgeam`) while each step of our implementation is particularly customized and highly tuned according to the architectures and the datasets. In particular, we achieve the largest speedup for `YahooMusic R4`. Although this dataset is small, our Cholesky-based approach plays a key role in reducing the time of factorizing $smat$ to be LL^T.

B. Evaluating Optimizations

Figure 6 shows how our ALS solver performs on the K20c GPU, the Intel MIC, and the Intel Xeon E5 CPU when using our optimization techniques. Starting with using *thread batching*, we incrementally apply the optimizations of *registers*, *local memory* and *vectors*. On GPUs, we observe that using registers and local memory can significantly improve the factorizing performance (by upto $2.6\times$), while using vectors brings very little change on performance.

On MIC and CPU, using local memory brings a performance increase for `Movielens`, `Netflix`, `YahooMusic R1`, and `YahooMusic R4`. The performance boost is upto $1.4\times$ for MIC and $1.6\times$ for CPU. Furthermore, using both registers and local memory degrades the overall performance remarkably. Therefore, it is not recommended to combine these two optimization techniques on MIC or CPU. We also notice a slight performance improvement by explicitly vectorizing the ALS code. As can be seen in Figure 6, the performance impact on the CPU resembles that on MIC because of the architectural similarities.

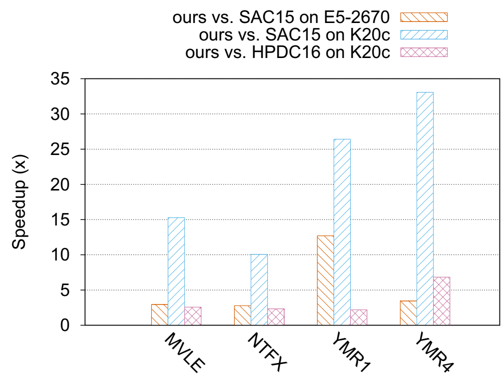


Figure 7. A performance comparison of our implementation versus the state-of-the-art implementations. We use the thread configuration of 8192×32 and 5 iterations, while $k = 10$.

C. Applying Optimizations

Our implementation consists of three steps when factorizing the rating matrix (Algorithm 2): (S1) $Y^TY + \lambda I$ (Lines 6–7), (S2) $Y^T r_u$ (Lines 8–15), and (S3) solve the linear system (Lines 16–17). When applying the optimization techniques, we give a priority to the most time-consuming step. Figure 8 shows an illustrative example on how we apply the optimization techniques in a step-by-step manner. Figure 8(a) shows the execution time percent of S1–S3, while Figure 8(b) is the number when applying *thread batching* on all the three steps. Although the percentage changes very slightly, the execution time of each step is reduced significantly. After applying the optimization, we notice that S1 takes up around 70% of the total execution time (i.e., the hotspot).

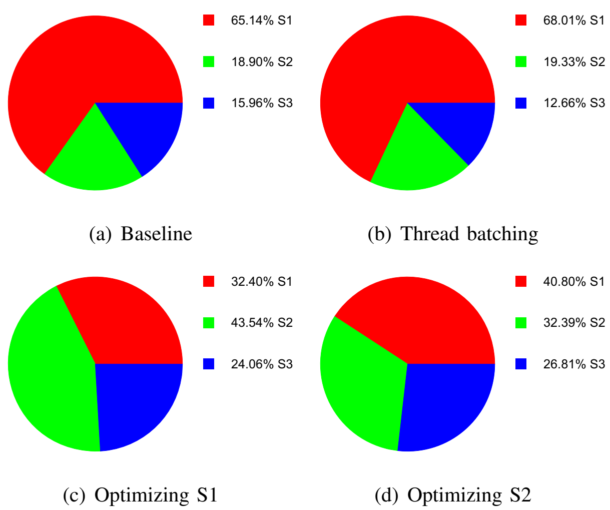


Figure 8. Applying optimization techniques in a step-by-step manner. The data is measured with `Netflix` on K20c.

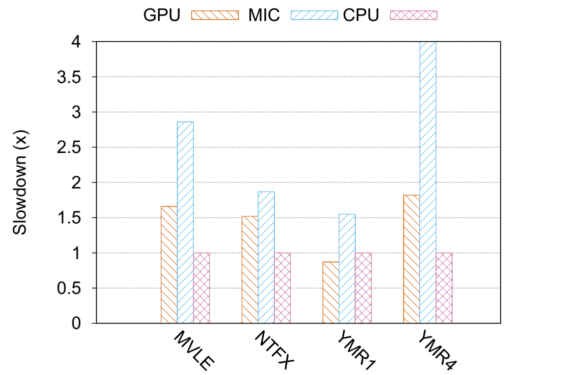


Figure 9. A performance comparison of our ALS code on various architectures. We use the thread configuration of 8192×32 and 5 iterations, while $k = 10$.

As indicated in Section III-C, local memory and registers are used to reduce the Y^TY time from 26 seconds to 6 seconds. Then the time consumption is shown in Figure 8(c). We see that S2 becomes the most time-consuming step. When calculating Y^Tr_u, local memory is used to stage the columns of Y. After that, Figure 8(d) shows that S1 dominates the factorization once again and becomes the new tuning focus. Besides, we can optimize S3 with the Cholesky method so that the overall running time (S1+S2+S3) is reduced to 12 seconds from 15 seconds. To summarize, we apply the optimization techniques and tune the ALS performance in a hotspot-guided manner.

D. Comparing Different Architectures

Figure 9 compares how our implementation performs on various architectures. We see that the 16-core CPU performs the best, GPU runs the second and then MIC follows. Specifically, our code on the K20c GPU runs 1.5× slower than it on the E5-2670 CPU, whereas it runs 4.1× slower on the Intel Xeon Phi. For the large datasets (`Movielens`, `Netflix` and `YahooMusic R1`), the performance gap between the GPU and the CPU is not so large. When working on `YahooMusic R1`, our ALS solver on the K20c GPU outperforms that on the 16-core CPU. Note that our optimized ALS on the K20c GPU can run 3× as fast as the OpenMP version on the 16-core CPU. In the future, we will further investigate the performance gap between platforms and push the factorizing performance to the hardware limit.

E. Sensitivity to Thread Blocks

Figure 10 shows the performance changes when using various thread block configurations. On the GPU, the execution time reaches its minimum when the block size equals 16 or 32, whereas the execution time increases when the block size is 8 or 64. We set k to be 10 in the experiment and thus two iterations are required to calculate $smat$ or $svec$. On the other hand, *warp* is the smallest unit of execution on the device and each warp contains 32 threads on the K20c GPU. Thus, the threads within each warp are under-utilized when the block size is 8. When the block size is 16 or 32, only one iteration is required to calculate $smat$ or $svec$ and the warp utilization is better than the case when the block size is 8. At the same time, the block size (16 or 32) is still smaller than the warp size and thus the execution time remains. Further increasing the block size (e.g., 64 threads per block) results in idle warps, leading to a performance drop. Therefore, it is recommended that the block size be the minimum integer number larger than the latent factor.

Different from GPU, the execution time on the CPU stabilizes over the size of thread block for `Movielens`, `Netflix`, and `YahooMusic R4`. To be more specific, the smaller the block size is, the better the factorization performance. We believe this is due to a better utilization of local memory. On MIC, we see that the thread block size has a significant impact on the execution time. The optimal block size varies for different datasets. For `YahooMusic R4`, using a block sized of 8 gives the best performance, whereas, for `YahooMusic R1`, 16 is better.

VI. Related Work

In this section, we discuss the main *matrix factorization* algorithms for recommender systems and the parallelization approaches on both multi-cores, many-cores and distributed platforms. As stated in [1], matrix factorization is regarded as the most successful realization of latent factor models in recommender systems. When factorizing a rating matrix, ALS (altering least squares), SGD (stochastic gradient descent) and CCD (cyclic coordinate decent) are the three most commonly used techniques.

Parallelizing ALS. GraphLab implements ALS by distributing matrix on multiple machines while the matrix is

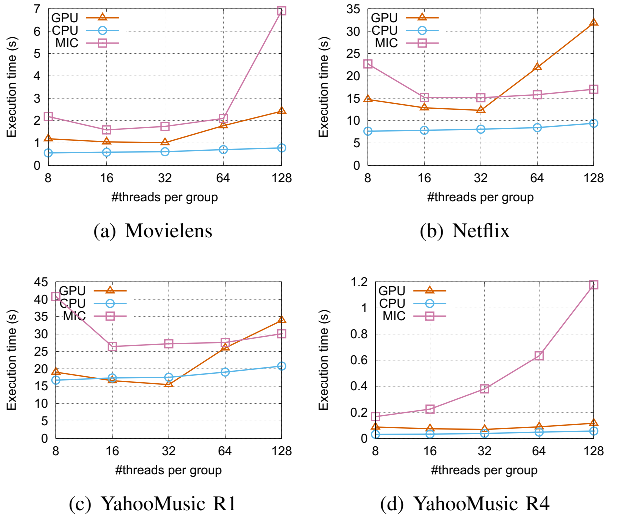


Figure 10. The performance changes over the thread block configuration. We use the thread configuration of 8192×32 and 5 iterations, while $k = 10$. We use *thread batching* + *local memory* + *registers* on the GPU while we only use *thread batching* + *local memory* on the CPU/MIC.

large, which results in heavy cross-node traffic and pretty high network bandwidth [19]. In [20], Spark MLlib leverages partial matrix replication to parallelize ALS. CuMF, a CUDA-based matrix factorization library, implements memory-optimized ALS to solve very large-scale MF by using a variety set of techniques to maximize the performance on either single or 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 between computing units, and utilizing a novel topology-aware parallel reduction scheme [13]. Gates et al. formulate ALS as a mix of cache-optimized algorithm-specific kernels and batched Cholesky factorization [21], and accelerate it on GPUs and multi-threaded CPUs [22]. Zhou et al. introduce a new parallel algorithm ALS-WR (weighted Regulation) for large-scale problems by using parallel Matlab on linux cluster [3].

Parallelizing CCD. Yu et al. propose a scalable and efficient method CCD++ which have the different update sequence from basic CCD and update rank-one factors one by one. The algorithm has two versions of parallelization on different machines: one version for multi-core shared memory systems and the other for distributed systems. If the matrices (A, W, H) fit in a single machine, they choose multi-core shared memory systems to parallelize CCD++ by diving the updating task into several subtasks that can be handled by different cores in parallel. When the matrices exceed the memory capacity of a single machine, a distributed system is used, and the parallelization method is same as the multi-core version [2]. Recently Nisa et al. [23] improved CCD++ method for GPU platform.

Parallelizing SGD. In [24], Paine et al. present an asynchronous SGD to speed up the neural network training on GPUs. In [25], [26], the authors propose the delayed update scheme and bootstrap aggregation scheme to parallelize SGD, respectively. HogWild uses a lock-free approach to parallelize SGD, which is shown to be more efficient than the delayed update scheme [27]. *Distribute SGD (DSGD)* partitions the ratings matrix into several blocks and updates a set of independent blocks in parallel at the same time [5]. Rashid Kaleem et al. show that parallel SGD can execute efficiently on GPU and the dynamically scheduled implementation on GPU is comparable to a 14-thread CPU implementation [28]. Jinoh et al. propose *MLGF-MF*, which is robust to skewed matrices and runs efficiently on block-storage devices (e.g., SSD disks) as well as shared-memory platforms. The implementation leverages *Multi-Level Grid File (MLGF)* to partition the rating matrix and minimizes the cost for scheduling parallel SGD updates on the partitioned regions by exploiting partial match queries processing [29]. *CuMF_SGD*, a CUDA-enabled SGD solution for large-scale matrix factorization problems, uses two workload scheduling schemes (*batch-Hogwild!* and *wavefront-update*) and a partitioning scheme to utilize multiple GPUs. At the same time, the authors address the well-known convergence issue when parallelizing SGD [30]. `Factorbird` uses a parameter server in order to scale models that exceed the memory of an individual machine, and employs lock-free *Hogwild!-style* learning with a special partitioning scheme to drastically reduce conflicting updates [31]. In [32], Sallinen et al. explore serveral modern parallelization methods of SGD on a shared memory system. In particular, they present a scalable, communication-avoiding implementation of SGD and demonstrate near linear scalability on a system with 14 cores.

To summarize, our work relates closely with [13], [12]. Different from these two works, our focus is the speed and portability of recommender systems on various architectures. The experimental results demonstrate that our implementation overtakes the cuMF code and the baseline code and is performance portable on various architectures.

VII. Conclusion

In this paper, we present an efficient and portable ALS solver. On the one hand, we diagnose the baseline implementation and observe that it is lack of awareness of the hierarchical thread organization on modern hardware. This leads to inefficient use of hardware resources: *unbalanced thread use* and *scattered memory access*. Thus, we apply the thread batching technique and three architecture-specific optimizations. On the other hand, we implement the ALS solver in OpenCL so that it can run on various platforms (CPUs, GPUs, and MICs). Based on the architectural specifics, we select a suitable code variant for each platform to efficiently map it to the underlying hardware. The experimental results show that our implementation performs 5.5× faster on E5-2670 and 21.2× faster on K20c than the baseline implementation. Our implementation also outperforms `cuMF` for various datasets (`Netflix`, `Movielens`, `YahooMusic R1`, and `YahooMusic R4`).

For future work, we will introduce the machine learning technique to select an appropriate code variant according to the target architecture and input dataset. Also, we will use more datasets to evaluate our ALS solver and extend our technique to other matrix factorization solvers such as SGD.

Acknowledgment

The authors would like to thank the anonymous reviewers for their constructive comments. This work was partially funded by the National Natural Science Foundation of China under Grant No.61402488, No.61502514 and No.61602501, the National Key Research and Development Program of China under Grant No. 2016YFB0200400.

References

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

[2] H. Yu, C. Hsieh, S. Si, and I. S. Dhillon, "Scalable coordinate descent approaches to parallel matrix factorization for recommender systems," in *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, pp. 765–774, 2012.

[3] Y. Zhou, D. M. Wilkinson, R. Schreiber, and R. Pan, "Large-scale parallel collaborative filtering for the netflix prize," in *Algorithmic Aspects in Information and Management, 4th International Conference, AAIM 2008, Shanghai, China, June 23-25, 2008. Proceedings*, pp. 337–348, 2008.

[4] G. Takács, I. Pilászy, B. Németh, and D. Tikk, "Scalable collaborative filtering approaches for large recommender systems," *Journal of Machine Learning Research*, vol. 10, pp. 623–656, 2009.

[5] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Diego, CA, USA, August 21-24, 2011*, pp. 69–77, 2011.

[6] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *12th IEEE International Conference on Data Mining, ICDM 2012, Brussels, Belgium, December 10-13, 2012*, pp. 655–664, 2012.

[7] W. Liu, *Parallel and Scalable Sparse Basic Linear Algebra Subprograms*. PhD thesis, University of Copenhagen, 2015.

[8] W. Liu and B. Vinter, "A framework for general sparse matrixmatrix multiplication on {GPUs} and heterogeneous processors," *Journal of Parallel and Distributed Computing*, vol. 85, pp. 47–61, 2015. {IPDPS} 2014 Selected Papers on Numerical and Combinatorial Algorithms.

[9] W. Liu and B. Vinter, "Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication," in *Proceedings of the 29th ACM International Conference on Supercomputing*, ICS '15, (New York, NY, USA), pp. 339–350, ACM, 2015.

[10] W. Hao, L. Weifeng, H. Kaixi, and F. Wu-chun, "Parallel transposition of sparse data structures," in *Proceedings of the 2016 International Conference on Supercomputing*, ICS '16, (New York, NY, USA), pp. 33:1–33:13, ACM, 2016.

[11] W. Liu, A. Li, J. Hogg, I. S. Duff, and B. Vinter, "A Synchronization-Free Algorithm for Parallel Sparse Triangular Solves," in *Euro-Par 2016: Parallel Processing: 22nd International Conference on Parallel and Distributed Computing, Grenoble, France, August 24-26, 2016, Proceedings*, pp. 617–630, 2016.

[12] A. V. Rodrigues, A. Jorge, and I. Dutra, "Accelerating recommender systems using gpus," in *Proceedings of the 30th Annual ACM Symposium on Applied Computing, Salamanca, Spain, April 13-17, 2015*, pp. 879–884, 2015.

[13] W. Tan, L. Cao, and L. L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on gpus," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing, HPDC 2016, Kyoto, Japan, May 31 - June 04, 2016*, pp. 219–230, 2016.

[14] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011*, pp. 267–276, 2011.

[15] J. Fang, H. J. Sips, and A. L. Varbanescu, "Aristotle: A performance impact indicator for the opencl kernels using local memory," *Scientific Programming*, vol. 22, no. 3, pp. 239–257, 2014.

[16] J. Fang, A. L. Varbanescu, X. Liao, and H. J. Sips, "Evaluating vector data type usage in opencl kernels," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 17, pp. 4586–4602, 2015.

[17] S. Muralidharan, A. Roy, M. W. Hall, M. Garland, and P. Rai, "Architecture-adaptive code variant tuning," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pp. 325–338, 2016.

[18] L. Chang, H. Kim, and W. W. Hwu, "Dysel: Lightweight dynamic selection for kernel-based data-parallel programming model," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '16, Atlanta, GA, USA, April 2-6, 2016*, pp. 667–680, 2016.

[19] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Distributed graphlab: A framework for machine learning in the cloud," *PVLDB*, vol. 5, no. 8, pp. 716–727, 2012.

[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. Kurzak, H. Anzt, M. Gates, and J. Dongarra, "Implementation and tuning of batched cholesky factorization and solve for nvidia gpus," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, pp. 2036–2048, July 2016.

[22] M. Gates, H. Anzt, J. Kurzak, and J. Dongarra, "Accelerating collaborative filtering using concepts from high performance computing," in *2015 IEEE International Conference on Big Data, Big Data 2015, Santa Clara, CA, USA, October 29 - November 1, 2015*, pp. 667–676, 2015.

[23] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, "Parallel ccd++ on gpu for matrix factorization," in *Proceedings of the General Purpose GPUs*, GPGPU-10, (New York, NY, USA), pp. 73–83, ACM, 2017.

[24] T. Paine, H. Jin, J. Yang, Z. Lin, and T. S. Huang, "GPU asynchronous stochastic gradient descent to speed up neural network training," *CoRR*, vol. abs/1312.6186, 2013.

[25] A. Agarwal and J. C. Duchi, "Distributed delayed stochastic optimization," in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pp. 873–881, 2011.

[26] M. Zinkevich, M. Weimer, A. J. Smola, and L. Li, "Parallelized stochastic gradient descent," in *Advances in Neural Information Processing Systems 23: 24th Annual Conference on Neural Information Processing Systems 2010. Proceedings of a meeting held 6-9 December 2010, Vancouver, British Columbia, Canada.*, pp. 2595–2603, 2010.

[27] B. Recht, C. Ré, S. J. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems 24: 25th Annual Conference on Neural Information Processing Systems 2011. Proceedings of a meeting held 12-14 December 2011, Granada, Spain.*, pp. 693–701, 2011.

[28] R. Kaleem, S. Pai, and K. Pingali, "Stochastic gradient descent on gpus," in *Proceedings of the 8th Workshop on General Purpose Processing using GPUs, GPGPU@PPoPP 2015, San Francisco, CA, USA, February 7, 2015*, pp. 81–89, 2015.

[29] J. Oh, W. Han, H. Yu, and X. Jiang, "Fast and robust parallel SGD matrix factorization," in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, Sydney, NSW, Australia, August 10-13, 2015*, pp. 865–874, 2015.

[30] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "Cumf_sgd: Fast and scalable matrix factorization," *CoRR*, vol. abs/1610.05838, 2016.

[31] S. Schelter, V. Satuluri, and R. Zadeh, "Factorbird - a parameter server approach to distributed matrix factorization," *CoRR*, vol. abs/1411.0602, 2014.

[32] S. Sallinen, N. Satish, M. Smelyanskiy, S. S. Sury, and C. Ré, "High performance parallel stochastic gradient descent in shared memory," in *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pp. 873–882, 2016.