clMF: A Fine-Grained and Portable Alternating Least Squares Algorithm for Parallel Matrix Factorization

Jing Chen\textsuperscript{a,∗}, Jianbin Fang\textsuperscript{a,∗}, Weifeng Liu\textsuperscript{b,∗}, Tao Tang\textsuperscript{a}, Canqun Yang\textsuperscript{a}

\textsuperscript{a}College of Computer, National University of Defense Technology, Changsha, China

\textsuperscript{b}Department of Computer Science, Norwegian University of Science and Technology, Norway

Abstract

Alternating least squares (ALS) has been proved to be an effective solver for matrix factorization in recommender systems. To speed up factorizing performance, various parallel ALS solvers have been proposed to leverage modern multi-cores and many-cores. Existing implementations are limited in either speed or portability. In this paper, we present an efficient and portable ALS solver (clMF) for recommender systems. On one hand, we diagnose the baseline implementation and observe that it lacks of the awareness of the hierarchical thread organization on modern hardware. To achieve high performance, we apply the thread batching technique, the fine-grained tiling 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 $2.8 \times$–$15.7 \times$ faster on an Intel 16-core CPU, $23.9 \times$–$87.9 \times$ faster on an NVIDIA K20C GPU and $34.6 \times$–$97.1 \times$ faster on an AMD Fury X GPU than the baseline implementation. On the K20C GPU, our implementation also outperforms cuMF over different latent features ranging from 10 to 100 with various real-world recommendation datasets.

Keywords: Matrix factorization, Alternating least squares, Performance

∗Corresponding author

Email addresses: jingchen95@yeah.net (Jing Chen), j.fang@nudt.edu.cn (Jianbin Fang), weifeng.liu@ntnu.no (Weifeng Liu)
1. 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 (MF) was empirically shown to be a better solution than the traditional nearest-neighbour approaches in the Netflix Prize competition [2]. Since then, there has been much work dedicated to the design of fast and scalable methods for large-scale matrix factorization problems [3, 1, 4, 5, 6, 7].

Among the matrix factorization techniques, alternating least squares (ALS) has been proved to be an effective one [1]. Compared to stochastic gradient descent (SGD) [8, 9], the ALS algorithm is not only inherently parallel, but can incorporate implicit ratings [1]. Nevertheless, the ALS algorithm involves parallel sparse matrix manipulation [10] which is challenging to achieve high performance due to imbalanced workload [11, 12, 13], random memory access [14, 15], unpredictable amount of computations [16] and task dependency [17, 18, 19]. This particularly holds when parallelizing and optimizing ALS on modern multi-cores and many-cores [20]. To address the issue, researchers have investigated various solutions. 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 [21]. Tan et al. provide a CUDA-based matrix factorization library (cuMF), which uses various techniques to maximize the performance on one or multiple GPUs [22]. Gates et al. formulate ALS as a mix of cache-optimized algorithm-specific kernels and batched Cholesky factorization, and accelerate it on GPUs and multi-threaded CPUs [23].

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 a K20C GPU runs much slower than the OpenMP version 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. Thus, converting the code into a right form is highly required according to the architectural specifics. In terms of portability, the available implementations are often limited to vendor-specific platforms. Running the ALS code on emerging hardware often needs from-scratch code engineering. The motivating observations are further detailed in Section 2.4.

In this paper, we present an efficient and portable ALS solver (clMF).

---

1clMF is a name short for “an ALS solver for matrix factorization in OpenCL”, which corresponds to cuMF (“an ALS solver for matrix factorization in CUDA”). The source code
On one hand, we diagnose the baseline implementation and observe that it is unaware of the hierarchical thread organization on modern multi-cores and many-cores. This leads to an inefficient and unbalanced use of hardware resources: unbalanced thread use and scattered memory access. Thus, we apply the thread batching technique, the fine-grained tiling 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 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 $2.8 \times -15.7 \times$ faster on an Intel 16-core CPU, $23.9 \times -87.9 \times$ faster on an NVIDIA K20C GPU and $34.6 \times -97.1 \times$ faster on an AMD Fury X GPU than the baseline implementation. Our implementation also outperforms cuMF for six real-world recommender datasets (Netflix, Movielens 10M, Movielens 20M, YahooMusic R1, YahooMusic R4, and Delicious).

To summarize, we make the following contributions.

- We present an efficient and portable ALS recommender system by applying the thread batching parallelization technique, the fine-grained tiling technique and the architecture-specific optimizations.
- We implement the recommender system with OpenCL and customize code variants for different architectures. The portable implementation facilitates us to enable/disable an optimization in an easy manner.
- We evaluate the ALS solver on four modern multi-/many-core platforms (CPU, GPU and MIC) and six recommender real-world datasets, and demonstrate that our ALS solver is efficient and portable.

The remainder of this paper is organized as follows. Section 2 describes the background of matrix factorization and the ALS algorithm, and the motivation. We present our approach in Section 3 and evaluate it in Section 5. Section 4 introduces the experimental platforms and the recommender datasets. Section 6 lists the related work, and Section 7 concludes our work.

2. Background

In this section, we describe the matrix factorization problem and the ALS algorithm. Then we analyze ALS in terms of time and space complexity, and introduce the motivation of our work with three observations.

of the clMF implementation is online available: https://github.com/jingchen95/clMF.
2.1. 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 $f$, a.k.a. latent feature, so that predicting unknown ratings can be estimated by the inner products of two vectors, $x_u$ of matrix $X(m \times f)$ and $y_i$ of matrix $Y(n \times f)$,

$$r_{ui} = x_u y_i^T,$$

(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, and $r_{ui}$ denotes an entry of the rating matrix $R$. The key of the problem is 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,

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

(2)

where $\Omega$ are 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 $\lambda$ 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 $Y$.

2.2. 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 the $Y$ matrix to calculate the $X$ matrix so as 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

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

(3)

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

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

(4)

where $I$ is the unit matrix ranked $f$, and $r_u$ is the $u$th rows of $R$. In the same way, we can obtain $y_i$. 

4
Algorithm 1 The ALS algorithm

1: procedure ALS($R$, $f$, $\lambda$; $X$, $Y$)
2: $X \leftarrow 0$, $Y \leftarrow$ random initial guess
3: repeat
4: for row $u \leftarrow 1$, $m$ do
5: $x_u \leftarrow (Y^T Y + \lambda I)^{-1} Y^T r_u$ by solving the linear system
6: $(Y^T Y + \lambda I)x_u = Y^T r_u$
7: end for
8: for column $i \leftarrow 1$, $n$ do
9: $y_i \leftarrow (X^T X + \lambda I)^{-1} X^T r_i$ by solving the linear system
10: $(X^T X + \lambda I)y_i = X^T r_i$
11: end for
12: until reaching the maximum iterations
13: end procedure

\[ y_i = (X^T X + \lambda I)^{-1} X^T r_i. \]  

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. Note that $x_u$ or $y_i$ can be obtained by solving linear systems (Lines 6 and 10). The algorithm iterates until it reaches the maximum specified cycles or error rate.

2.3. Algorithm Analysis

The algorithm consists of three steps when factorizing the rating matrix. When solving each user $x_u$ of $X$, the three steps are (S1) $Y^T Y + \lambda I$, (S2) $Y^T r_u$, and (S3) solving the linear system (Line 6 of Algorithm 1). When solving each item $y_i$ of $Y$, these three steps are (S1) $X^T X + \lambda I$, (S2) $X^T r_i$, and (S3) solving the linear system (Line 10 of Algorithm 1). A baseline implementation of the ALS algorithm is shown in Algorithm 2, and the three steps are located on Lines 6–7, Lines 8–15, and Lines 16–17, respectively.

As for S1, calculating $Y^T Y$ requires $nnz_i \times f \times (f + 1)/2$ multiply-add operations for a row of $R$, where $nnz_i$ denotes the number of nonzero entries in the current row. Therefore, the total computing cost is $nnz \times f \times (f + 1)$, where $nnz$ denotes the total number of non-zero elements in $R$. In terms of memory footprint, we need a matrix $smat$ sized of $f \times f$ (Line 6 of Algorithm 2) to store the results of $Y^T Y$ in global memory when updating a row. Thus, the total memory footprint for $m$ rows is $m \times f \times f$.

Calculating S2 requires $nnz_i \times f$ multiply-add operations when updating the $i$th row of $X$. Thus, the total computing cost of S2 is $nnz \times f \times 2$. This
step needs a vector $svec$ sized of $f$ (Line 12 of Algorithm 2) to store the results of $Y^T r_u$, and thus the total memory footprint for $m$ rows is $m \times f$.

For matrix factorization, *cholesky decomposition*, *LU decomposition* and *conjugate gradient method* (CG) are three means to solve the linear system of dense matrix. *Cholesky* and *LU decomposition* are direct methods solved by the highly optimized BLAS/LAPACK library, while *CG* solve a linear system in an iterative fashion. In this paper, we exploit the *cholesky decomposition* method to solve $sma\cdot x_u = svec$ (S3). The time complexity of *cholesky decomposition* is $O(f^3)$ for updating a row of $R$. To summarize, we notice that S1 is the most time-consuming step, which is confirmed by our experimental results.

### 2.4. Motivation

When running parallel ALS implementations SAC [21] and cuMF [22] on multi-cores and many-cores, we have the following three 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 in the context. Figure 1 compares the performance of ALS on a 16-core CPU and on a K20C GPU. We see that the ALS implementation (in [21]) runs, on average, $11.87 \times$ 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:** Unbalanced resource utilization of state-of-the-art accelerators leads to degraded factorizing performance.

Modern GPUs contain rich memory resources (global memory, texture memory, shared memory, and registers) and thread resources (each multi-processor can run thousands of threads). How to take advantage of these resources in a balanced manner (*neither too many nor too few*) is critical to the overall performance. In cuMF, a thread block is used to update a row $(Y^T Y)$ or a column $(X^T X)$ [22]. The entire task of calculating a $sma\cdot x_u$ (S1) is partitioned into multiple tiles, each sized of $10 \times 10$. Then cuMF lets each thread work on such a data tile. Instead of using a loop to iterate a $10 \times 10$ data tile, it fully unrolls the loop and allocates 100 registers to store the temporary results of $sma\cdot x_u$. Taking $f = 10$ for example, cuMF uses only one thread to calculate the temporary results of $Y^T Y$. On one hand, this approach leaves many threads to be idle and cannot make the best of thread resources per warp. On the other hand, the completely unrolled loop consumes too many registers and may reduce the number of active warps [24].
Observation 3: The current implementation cannot run on the coprocessors such as Intel Xeon Phi or AMD GPUs.

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 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 traditional multi-/many-cores, while the CUDA version is constrained to NVIDIA GPUs. The current implementation cannot be offloaded to run on Intel Xeon Phi and/or AMD GPUs. 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.

3. Design and Implementation

In this section, we give the baseline design of ALS and then present our approach (clMF). We customize the optimization techniques for different architectures and explain how to select an appropriate code variant in detail.
Algorithm 2 The Baseline ALS algorithm (updating $X$).

```plaintext
1: procedure Update X over Y (R, X, Y, f, $\lambda$; X)
2:  for $u \leftarrow 1, m$ do ▷ Foreach row
3:      $x_u \leftarrow $ GetBaseAddr($X, u, f$)
4:      omegaSize $\leftarrow $ CountNonZeros($R, u$)
5:      if omegaSize $> 0$ then
6:         $smat \leftarrow Y^T Y$ ▷ smat: sub-matrix
7:         $smat \leftarrow smat + \lambda I$
8:         for $c \leftarrow 0, f$ 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] $\times$ f + 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 $LL^T x = svec$ for $x$
18: end if
19: end for
20: end procedure
```

3.1. Baseline Design

In [21], 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 $(Y^T Y + \lambda I)$ and $smat$ (a matrix sized of $f \times f$) is introduced to store the temporary results. Lines 8–15 evaluate $Y^T r_u$ which is stored temporarily in a vector $svec$ sized of $f$. 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 data structures are introduced to represent the original matrix. A value array stores the nonzero elements of $R$ 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 also equals the number of nonzero elements. A row_ptr array stores the index of each row’s first nonzero element in value, and the difference between two continuous elements in row_ptr represents
Figure 2: An example of the compressed sparse row storage (CSR) format. $R$ has 5 rating scores out of 16 elements and three data structures are used in the representation.

The number of nonzeros in the current row. Thus, the size of $\text{row ptr}$ is the number of rows plus 1. Figure 2 illustrates the structure of CSR. The data structures ($\text{value}$, $\text{col idx}$, $\text{row ptr}$) are introduced to represent $R$ (See Lines 8–15 of Algorithm 2). Note that we use the compressed sparse column (CSC) format when updating $y_i$. This representation is similar to that of CSR, except that it stores the nonzero entries in a column-major manner.

3.2. ALS Parallelization on Modern Hardware

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 [25].

3.2.1. Basic Parallelization Technique

Modern many-core architectures organize threads in a hierarchical fashion. On GPUs, a warp of threads are organized to run on a SIMT core. When the threads within a warp diverge, they are serialized. Meanwhile, the threads from different warps can run concurrently. On CPUs or MICs, a group of fine-grained threads are expected to be packed to run on a SIMD core. To mine the hardware potentials, the threads on either a SIMT core or SIMD core have to follow the same execution path. For a typical recommender dataset, the number of nonzeros varies over rows (or columns). When two neighboring threads updating two continuous rows (or 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 (or columns) is significantly uneven, resulting in unbalanced thread use.

In terms of memory accesses, the threads within a GPU warp prefer accessing data elements near each other to guarantee coalesced memory accesses. On CPUs (or MICs), the memory accessing requests are performed
in a cacheline granularity. Nonetheless, the baseline implementation accesses the global memory space in an inefficient manner. Specifically, each thread calculates a matrix (\texttt{smat} sized of \( f \times f \)) and a vector (\texttt{svec} sized of \( f \)). Thus, the distance between two accesses of neighboring threads is at least \((f + 1) \times f\). This scattered memory accesses lead to a poor bandwidth use.

To address the issues, we apply the \textit{thread batching} technique and use a SIMT/SIMD core to update a row or a column of \( R \). For S1 of Algorithm 2, we exploit a 1D thread configuration and let each thread work on calculating a row of \texttt{smat}. To fully exploit the register file, we allocate a register array (sized of \( f \)) for each thread to store the temporary results. Once finished, the temporary results are written into the corresponding rows of \texttt{smat}. Figure 3 illustrates the basic parallelization technique on the thread usage, where there are 3 non-zeros and \( f = 4 \). We move the corresponding columns of \( Y \) according to the column indices of these non-zeros into shared memory \( sY \). Each thread deals with the computing task of a row in \( sY \) with all the columns of \( sY^T \), e.g., thread 0 calculates the \textit{multiply-add} product of the first row of \( sY \) with all the columns of \( sY^T \) (Figure 3). Compared with S1, S2 computes the multiplication of a matrix \( sY \) sized of \( f \times nnz \) with a vector \( v \) sized of \( nnz \). In the same way, we exploit a 1D grid of threads configuration and allocate one register for each thread. This allocated register stores the temporary result of each row in matrix \( sY \) with the vector \( v \). In addition, a thread block is applied to factorize a \texttt{smat} matrix (i.e., solving S3). In this way, the \textit{thread batching} technique can not only avoid unbalanced thread use but batch the data accessing requirements. Meanwhile, it is applicable not only on CPUs, but also on GPUs and MICs.
3.2.2. Fine-Grained Tiling Technique

The aforementioned technique is equally used to update $X$ or $Y$ in cuMF. Nevertheless, we observe that their approach leaves many threads to be idle and cannot make the best of thread resources per warp [24]. Also, cuMF completely unrolls the loop with 100 registers, which consumes too many resources and may reduce the number of concurrent warps. These result in poor performance particularly when $f$ is small (i.e., $f < 70$), and lead us to exploit the hardware resources in a balanced manner.

Based on the basic technique, we further partition $\text{smat}$ into multiple tiles, each updated by a thread block, to increase the data parallelism. Specifically, we use a 3D grid (instead of a 1D grid in basic parallelization technique) of thread blocks: $(bx, by, bz)$, in which $bz$ corresponds to the batch size of rows (or columns) in the $R$ matrix. At the same time, we divide $\text{smat}$ into $bx \times by$ tiles, each of which is solved by one thread block. Thus, there are a total of $bx \times by$ thread blocks to update a row (or a column). Due to the independence of the data tiles, these $bx \times by$ thread blocks can run concurrently. Further, we use a 3D grid of threads per block to calculate each data tile: $(tx, ty, tz)$. Such a thread organization can achieve coalesced memory accesses when loading the corresponding columns of $Y$ into shared memory.

Figure 4 illustrates an example of the fine-grained tiling technique used in updating a specific row. Here we introduce another parameter $nnz\_batch$ to denote the number of column vectors that are moved from global memory to shared memory per time. By doing so, we aim to avoid allocating a too large on-chip buffer and increase the number of active warps. We set $f=100$, $nnz_i=4$ (i.e., the number of non-zeros in the current row), and $nnz\_batch=2$. As for the thread configuration, we set $bx=by=2$, $tx=ty=25$. Thus, there are...
Figure 5: An example of unrolling the code to calculate $Y^TY$. $lx$ is the local work-item index, $ws$ is the work-group size, $f$ 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 $f = 5$.

3.3. 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.

3.3.1. Using Registers

The recent GPUs feature a large amount of 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 a rating matrix is a typical bandwidth-limited kernel. Thus, an efficient utilization of these registers can improve the kernel performance. When calculating $Y^T Y$ (Line 6 of Algorithm 2), the original code uses a private array ($sum[f \times f]$) to store the temporary results before updating $\text{smat}$ (Figure 5). Despite that the structure is private to a thread, register spilling occurs with a large $f$. We observe that allocating a $f \times f$ buffer per thread is not required. In fact, a buffer sized of $f$ for each thread is sufficient if we use the basic parallelization strategy. The restructured code is shown in Figure 5(b). In cuMF, they allocate 100 registers for each threads, which probably consumes too many registers and reduces the number of active warps [24]. In contrast, clMF needs $(\text{tile} \_\text{size}/tx) \times (\text{tile} \_\text{size}/ty)$ registers to store the temporary results (Figure 4). This parameterized code enables us to use the register resource in a more balanced manner.

3.3.2. 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 [26, 27, 28].

As shown in Algorithm 2 (Lines 8–15), calculating $Y^T r_u$ needs to load data from $R$ (i.e., the $\text{value}$ array) and $Y$. Specifically, updating $\text{svec}$ 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 6 shows that we allocate a local memory buffer ($3 \times 5$) to cache the required data columns of $Y$. At the same time, updating $\text{svec}$ 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 6 shows how a local memory vector is allocated to store all the non-zero entries of $r_u$.

3.3.3. 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 [29]. OpenCL provides vector data types to exploit the vector cores, e.g., $\text{float16}$ is a vector containing 16 scalar data elements typed of $\text{float}$. The arithmetic operators can perform the corresponding operations in an element-wise manner. We use $\text{vload}$ to fill vectors while using $\text{vstore}$ to write results to memory.
3.4. 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 [30, 31]. Based on the thread batching version, we will yield eight 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) [32, 33].

In this context, we use an empirical approach to select a right code variant. In total, we provide eight 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 9). For example, due to the missing scratchpad 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 [34].

4. Experimental Setup

In this section, we first introduce the hardware and software configurations used in the context, and then describe the details of the real-world datasets used to evaluate our implementation.
Table 1: Datasets

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Abbr.</th>
<th>m</th>
<th>n</th>
<th>nnz</th>
<th>sparsity</th>
<th>avg. nnz/row</th>
<th>avg. nnz/col</th>
</tr>
</thead>
<tbody>
<tr>
<td>Movielens10M</td>
<td>MV10</td>
<td>71567</td>
<td>63133</td>
<td>8000044</td>
<td>0.0017</td>
<td>111.78</td>
<td>122.83</td>
</tr>
<tr>
<td>Movielens20M</td>
<td>MV20</td>
<td>138493</td>
<td>27278</td>
<td>20000263</td>
<td>0.0053</td>
<td>144.41</td>
<td>733.20</td>
</tr>
<tr>
<td>NetFlix</td>
<td>NTFX</td>
<td>490189</td>
<td>17770</td>
<td>99072112</td>
<td>0.0116</td>
<td>206.32</td>
<td>5575.25</td>
</tr>
<tr>
<td>YahooMusic R1</td>
<td>YMR1</td>
<td>194882</td>
<td>98212</td>
<td>115248575</td>
<td>0.0006</td>
<td>59.14</td>
<td>173.47</td>
</tr>
<tr>
<td>YahooMusic R4</td>
<td>YMR4</td>
<td>7642</td>
<td>11916</td>
<td>211231</td>
<td>0.0023</td>
<td>27.64</td>
<td>17.71</td>
</tr>
<tr>
<td>Delicious</td>
<td>DELI</td>
<td>107253</td>
<td>65000</td>
<td>487131</td>
<td>0.00007</td>
<td>4.54</td>
<td>7.49</td>
</tr>
</tbody>
</table>

4.1. Platform Configurations

We use four multi-/many-core platforms in the experiment: Intel Xeon CPU, NVIDIA Tesla K20C GPU, AMD Fury X 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 2.60 GHz. NVIDIA GPU is a Tesla K20C, which contains 13 streaming multiprocessors (SM), and 192 CUDA cores on each SM. The AMD Fury X GPU (based on GCN Fiji) features 4096 radeon cores. This GPU also has high-bandwidth memory (4 GB) with a 4096-bit memory interface. The Intel Many Integrated Cores (MIC) is Intel Xeon Phi 31SP, with 57 cores and 6 GB GDDR 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) to run the cuMF code and the baseline code on GPU. The driver version of the AMD GPU is v15.12.

4.2. Input Datasets

We use six real-world recommender datasets (Movielens 10M, Movielens 20M,

R1, YahooMusic R4, Netflix,

Delicious) to measure the factorization performance. The entry format of each dataset is

http://files.grouplens.org/datasets/movielens/

http://webscope.sandbox.yahoo.com

http://www.select.cs.cmu.edu/code/graphlab/datasets/

http://grouplens.org/datasets/hetrec-2011/
(user\textit{ID}, item\textit{ID}, rating). We preprocess each dataset according to this format. The details of the six datasets are shown in Table 1, where \( m \) is the number of users, \( n \) is the number of items, and \( nnz \) is the number of non-zero entries in the dataset. The sparsity of a rating matrix is calculated by \( nnz/(m \times n) \). In the context, \( \lambda = 0.1 \) unless otherwise specified.

5. Performance Results

In this section, we first show how clMF 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 compare the performance results across four many-core platforms. Finally, we empirically tune the parameters to obtain the best performance for clMF.

5.1. Comparing with State-of-the-Art

We compare the performance of our clMF implementation with two state-of-the-art implementations: SAC [21] and cuMF [22].

Comparing with SAC. As for SAC, one GPU thread is used to update a row of the \( X \) matrix, where all the temporary data of \( Y^T Y \) is allocated dynamically in the kernel function. But when \( f \) becomes large, there is insufficient global memory space remained for dynamic allocation and thus the kernel failed to run. In this case, this implementation does not scale over the latent feature. Thus, we focus on comparing clMF and SAC when \( f = 10 \), which is shown in Figure 7. We see that clMF performs significantly better than SAC on all datasets, with a speedup ranging from 23.9\( \times \) to 87.9\( \times \) on K20C, from 34.63\( \times \) to 97.1\( \times \) on Fury X and from 2.8\( \times \) to 15.7\( \times \) on CPU. Also, we notice that clMF runs particularly fast on the small datasets such as Yahoomusic R4 and Delicious. This significant performance improvement comes from the usage of the appropriate parallelization technique and the architecture-specific optimizations (Section 5.2).

Comparing with cuMF. Figure 8 shows the performance comparison between clMF and cuMF on the GPU. We observe that, clMF performs better than cuMF for most datasets. In particular, clMF always runs faster than cuMF for the small-scale datasets. cuMF scales linearly as \( f \) changes from 10 to 60 and 70 to 90 on all datasets. This is because cuMF partitions a \( f \times f \) matrix into multiple 10 \( \times \) 10 data tiles, but it uses only one thread to update a tile and uses distinct threads within a warp to work on different tiles. To update a row (or a column) of \( X \) (or \( Y \)), they need a total of \( f/2 \) threads. Accordingly, cuMF allocates 100 registers per thread to stage the temporary results. Therefore, the number of active threads increases slightly and the
Figure 7: A performance comparison of clMF versus SAC implementation, where GPU1 denotes K20C and GPU2 denotes Fury X. SAC implementation uses the thread configuration of \(8192 \times 32\), while clMF exploits \((1,1,16384)\) thread blocks and \((5,5,1)\) threads, where \(f = 10\) and iteration=1.

Performance gap between different \(f\) ranging from 10 to 60 and from 70 to 90 is not very dramatic. In other words, cuMF leaves many threads to be idle, which can be avoided by our fine-grained tiling technique. This is why clMF outperforms cuMF when the latent factor is small. Also, we observe that when \(f=70\), cuMF sees a sharp rise in the execution time. The reason is that the warp size is 32, but cuMF exploits 35 threads, which is right larger than the warp size. This results in unbalanced thread use in GPU architecture and degraded performance. Due to the customized kernel in cuMF when \(f=100\), the execution time of cuMF is less than clMF on the three large datasets.

5.2. Evaluating Optimizations

Figure 9 shows how our ALS solver performs on the NVIDIA GPU, the AMD 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 K20C, we observe that using registers and local memory can significantly improve the factorizing performance (by up to 2.6\(\times\)). Meanwhile, using local memory on Fury X brings the most significant performance improvement (by up to 12.58\(\times\)), compared with the ALS implementation with the basic batching technique. We also observe that further applying the register optimization degrades the overall factorization performance on the AMD GPU.

On MIC and CPU, using local memory brings a performance increase for Movielens 10M, Netflix, YahooMusic R1, and YahooMusic R4. The performance boost is up to 1.4\(\times\) for MIC and 1.6\(\times\) for CPU. Furthermore, using registers and local memory simultaneously degrades the overall performance.
Figure 8: A performance comparison of clMF and cuMF over different latent feature size and various datasets with one iteration on K20C. The datasets are divided into two groups: the left figure contains large datasets (Netflix, Yahoo music R1, Movielens 20M) while the right figure uses small datasets (Movielens 10M, Yahoo music R4, Delicious). cuMF uses batch size thread blocks each with $f/2$ threads when $f \in [10, 90]$, whereas each thread block has 64 threads when $f=100$. The thread configurations of clMF are listed in Table 2.

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 9, the performance impact on the CPU resembles that on MIC because of the architectural similarities.

5.3. Applying Optimizations

Algorithm 2 shows that our implementation consists of three steps when factorizing the rating matrix: (S1) $Y^T Y + \lambda I$ (Lines 6–7), (S2) $Y^T r_u$ (Lines 8–15), and (S3) solving the linear system (Lines 16–17). When applying the optimization techniques, we give a priority to the most time-consuming step. Figure 10 shows an illustrative example on how we apply the optimization techniques in a step-by-step manner. Figure 10(a) shows the execution time percent of S1–S3, while Figure 10(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 and thus becomes the tuning hotspot.

As indicated in Section 3.3, local memory and registers are used to reduce the $Y^T Y$ time from 26 seconds to 6 seconds. Then the time consumption is shown in Figure 10(c). We see that S2 becomes the most time-consuming step. When calculating $Y^T r_u$, local memory is used to stage the columns of $Y$. After that, Figure 10(d) shows that S1 dominates the factorization once
Figure 9: A performance comparison of the ALS solver on different architectures and datasets, where GPU1 denotes NVIDIA GPU and GPU2 denotes AMD GPU. We set $f = 10$ and use 8192 thread blocks. We exploit 32 and 64 threads per block on K20C and Fury X respectively.

again and becomes the new tuning spot. 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.

5.4. Comparing Different Architectures

Figure 11 compares how c1MF performs on various architectures and datasets. Note that the most suitable code variants and the best thread configurations are used for each hardware when measuring the performance results. We see that the AMD GPU performs the best, the NVIDIA GPU runs the second, the 16-core CPU runs the third and then MIC follows. Specifically, c1MF achieves a speedup of up to $3.4\times$ on the NVIDIA GPU and up to $7.6\times$ on the AMD GPU, compared with the performance on the E5-2670 CPU. To summarize, we argue that GPUs are the promising
platform for the ALS workload when taking both performance and power consumption into account. In the future, we will further investigate the performance gap between platforms and push the factorizing performance to the hardware limit (in particular on newer Intel Xeon Phi processors with on-package high bandwidth memory [35, 36], newer GPUs on warp-level [37, 38], CTA-level [39] and cache-level [40], and other emergent accelerators such as Matrix-2000 [41]).

5.5. Tuning Knobs for clMF

Selecting suitable parameters is key to achieve high performance for clMF. In this section, we empirically evaluate how thread configurations have an impact on the overall performance when using the basic parallelization technique (Section 3.2.1) and the fine-grained tiling technique (Section 3.2.2).

Case 1. Figure 12 shows the performance changes when using the basic parallelization technique on four datasets. Since we use a 1D grid, there is only one tuning knob (i.e., the number of threads per block) in this case. 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 \( f \) to be 10 in the experiment and thus two iterations are required to calculate \( \text{smat} \) or \( \text{svec} \). On the other hand, \( \text{warp} \) 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 \( \text{smat} \) or \( \text{svec} \) 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 10M, 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 best 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.

**Case 2.** When using the fine-grained tiling technique, we have seven tuning knobs: three on the thread block configuration \((tx, ty, tz)\), three on the number of thread blocks \((bx, by, bz)\) and one on the size of a batch \((nnz\_batch)\). Selecting a right \( nnz\_batch \) depends on the size of the on-chip shared memory. Figure 13 shows how the factorizing performance changes with tuning.
Figure 12: The performance changes over the thread block configuration. We use the thread configuration of $8192 \times 32$ and 5 iterations, while $f = 10$. We use thread batching + local memory + registers on the GPU while we only use thread batching + local memory on the CPU/MIC.

Figure 13: How the performance changes over the various tuning knobs on the various dataset, where $f = 100$, $bz = 16384$, $tz = 1$, and the number of iterations in $clMF$ is 1. The format of the X labels is ($bx$, $by$, $tx$, $ty$) and we exploit the same thread configuration for $clMF$ on K20C and Fury X.
knobs on the two GPUs. We observe that the performance changes are dramatic over different thread configurations, i.e., the execution time of Netflix dataset is of $5.0 \times$ and $3.2 \times$ difference between the worst and the best configuration on K20C and Fury X, respectively. Particularly, we find that the configuration of $(bx=2, by=2, bz=16384, tx=25, ty=25, tz=1)$ is the best configuration for all the datasets on K20C. Note that \texttt{clMF} can not run on Fury X when the number of threads per dimension exceeds 16. Thus, we only have five thread configurations on the AMD GPU (Figure 13(b)). We observe that the configuration of $(bx=5, by=5, bz=16384, tx=10, ty=10, tz=1)$ on Fury X performs the best on most of the datasets. The superior performance comes from the fact that it makes use of the hierarchical resources (e.g., registers and local memory) of modern accelerators in a balanced manner.

In this context, we empirically search the tuning space for the best configuration. On K20C, we use 4 thread blocks to update a row (or a column) of $X$ (or $Y$) when $f \in [20, 100]$, while we use 25 thread blocks to update a row (or a column) of $X$ (or $Y$) for the same $f$ on Fury X. Moreover, we only use 1 thread block to do the same thing when $f=10$. Through a large number of experiments, we observe that in most cases, \texttt{clMF} can achieve the best performance when $tx$ (or $ty$) is the greatest common divisor of the tile size. For instance, when $f=100$, we use 4 thread blocks to update a row of $X$ on K20C and therefore, the tile size is $50 \times 50$. The greatest common divisor of 50 is 25, so in this case we use a thread configuration of $25 \times 25$ to work on a tile. On Fury X, 25 thread blocks are used to update a row or a column. Thus, the tile size is $20 \times 20$ and we use a thread block sized of $10 \times 10$ to compute a tile, which is the greatest common divisor of tile size. The tuning knobs for K20C used in Section 5.1 are listed in Table 2.

### Table 2: Tuning knobs for K20C

<table>
<thead>
<tr>
<th></th>
<th>$f=10$</th>
<th>$f=20$</th>
<th>$f=30$</th>
<th>$f=40$</th>
<th>$f=50$</th>
<th>$f=60$</th>
<th>$f=70$</th>
<th>$f=80$</th>
<th>$f=90$</th>
<th>$f=100$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$bx$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$by$</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$bz$</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
<td>16384</td>
</tr>
<tr>
<td>$tx$</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>7</td>
<td>20</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>$ty$</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>25</td>
<td>15</td>
<td>7</td>
<td>20</td>
<td>15</td>
<td>25</td>
</tr>
<tr>
<td>$tz$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$nnz$ batch</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
</tbody>
</table>
6. Related Work

In this section, we discuss the matrix factorization algorithms for recommender systems and their implementations on 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.

The ALS solver. GraphLab implements ALS by distributing matrix on multiple machines while the matrix is large, which results in heavy cross-node traffic and pretty high network bandwidth [42]. Spark MLlib leverages partial matrix replication to parallelize ALS [43]. 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 [22]. Gates et al. formulate ALS as a mix of cache-optimized algorithm-specific kernels and batched Cholesky factorization [44], and accelerate it on GPUs and multi-threaded CPUs [23]. Zhou et al. introduce a new parallel algorithm ALS-WR (weighted regulation) for large-scale problems by using parallel Matlab on a linux cluster [3].

The CCD solver. Yu et al. propose a scalable and efficient method CCD++ which has a different update sequence from the basic CCD and updates 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 [2]. Recently Nisa et al. improve the CCD++ method on GPUs with loop fusion and tiling [45]. Yang et al. present an efficient and portable CDMF solver on modern multi-core and many-cores [46]. In particular, they balance the factorization loads by re-organizing the non-zero entries of rating matrices.

The SGD solver. Paine et al. present an asynchronous SGD to speed up the neural network training on GPUs [47]. In [48, 49], the authors propose a delayed update scheme and a bootstrap aggregation scheme to speed up SGD. HogWild uses a lock-free approach to parallelize SGD, which is shown to be more efficient than the delayed update scheme [50]. DSGD (Distribute SGD) partitions the ratings matrix into several blocks and updates a set of independent blocks concurrently [8]. Kaleem et al. show that the parallel SGD can run efficiently on GPU, and their implementation on GPU is
comparable to a 14-thread CPU implementation [51]. 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 to partition the rating matrix and minimizes the cost of scheduling parallel SGD updates on the partitioned regions [52]. CuMF, 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 [53]. Factorbird uses a parameter server in order to scale models that exceed the memory of an individual machine, and employs a lock-free Hogwild!-style learning with a special partitioning scheme to drastically reduce conflicting updates [54]. Sallinen et al. explore several modern parallelization methods of SGD on a shared memory system [55]. In particular, they present a scalable, communication-avoiding implementation of SGD and demonstrate near-linear scalability on a system with 14 cores.

The SVD solver. Matrix factorization models map both users and items to a joint latent factor space of dimensionality $f$, such that user-item interactions can be modeled as inner products in that space. Therefore, the recommendation problem is how to compute a mapping of items and users to factor vectors [1, 56]. In the collaborative filtering domain, singular value decomposition (SVD) [57, 58] is also a well-established technique of identifying latent feature factors. However, the conventional SVD is often unapplicable in matrix factorization of the recommendation field due to the high percentage of missing entries in the sparse user-item matrix. When the matrix is incomplete, it is not possible to achieve the factoring task. Moreover, overfitting would occur if we address the sparse matrix carelessly. Therefore, we need an approach that can simply ignore the missing ratings in the sparse matrix, modeling directly the observed ratings. To this end, researchers have performed intensive research to improve the applicability of SVD in collaborative filtering. For example, in [59], Chih-chao proposed four variants of SVD to solve large-scale matrix of collaborative filtering instead of the conventional SVD, including incomplete incremental learning, complete incremental learning, complete incremental learning, batch learning with a momentum, SVD with biases. He observed that complete incremental learning which updates feature values after scanning a single training score of $R$, may be a good choice for collaborative filtering with millions of training instances. The method minimizes the objective function and addresses the negative gradients for each user and item according to each non-zero elements of the $R$ matrix per time. Therefore, it has $nnz$ (i.e. total number of non-zero elements in $R$) iterations. We focus on using the ALS algorithm in this work.
and will compare ALS, CCD, SGD, SVD for future work. The extensions to our previous work [60] are three-fold. We further propose an efficient fine-grained technique (see Section 3) and demonstrate the performance improvement over state-of-the-art implementations in Section 5. To further show the portability of clMF, we run all the experiments on a new many-core architecture (i.e., an AMD GPU) and perform an in-depth analysis on the performance results (Section 5). We have also used two more real-world datasets of recommender systems to quantify the performance of clMF.

To summarize, our work relates closely with [21, 22, 23]. By using the thread batching technique and the architecture-specific optimizations, clMF remarkably outperforms SAC on both multi-cores and many-cores [21]. By introducing a fine-grained tiling technique, our clMF can achieve better performance than cuMF, which is now constrained to the CUDA-compatible platforms [22]. Gates et al. present a highly optimized CUDA kernel for recommender systems with implicit ratings [23]. Although borrowing the idea of the fine-grained tiling technique, we focus on the explicit rating matrices. Above all, our focus is on both 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.

7. Conclusion

In this paper, we present an efficient and portable ALS solver. On 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 and unbalanced use of hardware resources: unbalanced thread use and scattered memory access. Thus, we apply the thread batching technique, the fine-grained tiling 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 $2.8 \times -15.7 \times$ faster on a 16-core CPU, $23.9 \times -87.9 \times$ faster an NVIDIA K20C GPU and $34.6 \times -97.1 \times$ faster on an AMD Fury X GPU than the baseline implementation. Our implementation also outperforms cuMF for various datasets (Netflix, Movielens 10M, Movielens 20M, YahooMusic R1, YahooMusic R4, and Delicious).

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.

8. Acknowledgments

The authors would like to thank our anonymous reviewers for their invaluable comments and suggestions. This research was supported by the National Key R&D Program of China under Grant No. 2017YFB0202003, the National Natural Science Foundation of China under Grant No. 61602501, and the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie project under Grant No. 752321. For any correspondence, please contact Jianbin Fang (Email: j.fang@nudt.edu.cn).

References


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, pp. 81–89.


