BALS: Blocked Alternating Least Squares for Parallel Sparse Matrix Factorization on GPUs

Jing Chen, Jianbin Fang, Weifeng Liu, Senior Member, IEEE and Canqun Yang

Abstract—Matrix factorization on sparse matrices has been proven to be an effective approach for data mining and machine learning. However, the prior parallel implementations for matrix factorization fail to capture the internal social property embedded in real-world use cases. This article presents an efficient implementation of the alternative least squares (ALS) algorithm called BALS built on top of a new sparse matrix format for parallel matrix factorization. The BALS storage format organizes the sparse matrix into 2D tiles to avoid repeated data loads and improve data reuses. We further propose a data reordering technique to sort sparse matrices according to nonzeros. The experimental results show that BALS can yield a superior performance than state-of-the-art implementations, i.e., our BALS generally runs faster than Gates’ implementation over different latent feature sizes, with a speedup of up to 2.08× on K20C, 3.72× on TITAN X and 3.13× on TITAN RTX. When compared with alternative matrix factorization algorithms, our BALS consistently outperforms CDMF, cuMF_CCD, and cuMF_SGD over various latent feature sizes and datasets. The reordering technique can provide an extra improvement of up to 23.68% on K20C, 19.87% on TITAN X and 20.38% on TITAN RTX.

Index Terms—Matrix factorization, Alternating least squares, Data reuse, Data reordering, Performance evaluation, GPGPUs

1 INTRODUCTION

Matrix factorization has been taken as one of the most successful realizations of latent factor models and thus has been widely used in the machine learning fields such as collaborative filtering recommender systems [8]. Its task is to fill in the missing entries of a partially observed matrix. The input of matrix factorization is an incomplete relation matrix \( R(m \times n) \). In recommender systems, \( m \) and \( n \) denote the number of users and items, respectively. Due to the sparsity of \( R \), matrix factorization maps both users and items to a joint factor space of dimensionality \( f \) (i.e., 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 \( Y(n \times f) \),

\[
r_{ui} = x_u y_i^T,
\]

where \( x_u \) denotes the extent of user’s interest on items, \( y_i \) denotes the extent to which the item owns these factors, and \( r_{ui} \) denotes an entry of \( R \). The essence of this problem is to obtain \( x_u \) and \( y_i \) so that \( R \approx XY^T \). Figure 1 illustrates an example for matrix factorization, where \( m=n=4 \) and \( f=2 \).

So far, there has been a large amount of work dedicated to the design of fast and scalable methods for large-scale matrix factorization problems [7, 8, 15, 24, 25, 34]. However, matrix factorization over extremely sparse matrices (e.g., recommender datasets) is still a challenging issue [22, 27]. Among the factorization techniques, alternating least squares (ALS) has been proved to be an effective one [8]. Compared to stochastic gradient descent (SGD) [5, 28], ALS is not only inherently parallel, but can incorporate implicit ratings [8]. Nevertheless, ALS involves sparse matrix manipulation [10] which is challenging to achieve high performance due to imbalanced workload [11], random memory access [17] and task dependency [13]. This particularly holds when parallelizing and optimizing ALS on GPGPUs [3]. For this, researchers have investigated various solutions. Rodrigues et al. present a CUDA-based ALS implementation on GPU, which runs faster than the implementation on a multicore CPU [20]. Tan et al. provide a CUDA-based matrix factorization library (CuMF), which uses various memory-related techniques to maximize the performance on one or multiple GPUs [26]. gates et al. propose a multi-core CPU implementation and a GPU ALS solver for implicit feedback datasets, which attains good performance through an algorithm-specific kernel and is, thus far, the fastest implementation among these ALS solvers on GPUs [4].

In spite of these efforts, the training speed of parallel sparse matrix factorization based on the ALS algorithm has not reached its optimum. Although the previous works have used fine-grained techniques to exploit the hierarchical resources on modern GPGPUs, they fail to capture and utilize the internal social property embedded in the real-world datasets [3, 7, 26]. This can be drawn from our observation

- J. Chen, J. Fang and C. Yang are with the College of Computer, National University of Defense Technology, Changsha, China, 410073. E-mail: chjing@chalmers.se, j.j.fang, canqun}@nudt.edu.cn
- W. Liu is with the Super Scientific Software Laboratory, Department of Computer Science and Technology, China University of Petroleum, Beijing, China, 102249. E-mail: weifeng.liu@cup.edu.cn
  (Corresponding author: Jianbin Fang and Weifeng Liu.)
that there exist many users who have ratings for the same item, and the number of such type of items are enormous in these datasets. When computing $X$ or $Y$, the prior approaches let a thread or a block of threads work on a user vector ($x_u$) or an item vector ($y_i$). But updating two user vectors might need the same item vectors from $Y$. As a result, the same vector would be loaded twice, leading to redundant data movements and a waste of memory bandwidth. The motivating observations will be detailed in Section 3.

We propose an efficient implementation for parallel sparse matrix factorization, BALS\(^1\), based on the alternating least squares algorithm on GPGPUs. BALS aims to avoid repeated data loads of row or column vectors from global memory to shared memory and exploit data reuse based on our observations. For this purpose, we propose a new blocked storage format for sparse matrices by partitioning a matrix into 2D tiles. To enhance data reuse, we further propose a data reordering technique in BALS by sorting rows and columns in the descending order of nonzeros.

The experimental results demonstrate a better performance than the state-of-the-art implementations on three generations of NVIDIA GPUs and six real-world datasets. Overall, BALS runs the fastest among competitive implementations. It outperforms Gates’ implementation over different latent feature sizes, with a maximum speedup of 2.08×, 3.72× and 3.13× on K20C, TITAN X and TITAN RTX, respectively. We also compare BALS with alternative matrix factorization algorithm implementations, seeing that it outperforms CDMF\(^2\), cuMF\(^3\) for CCD++, and cuMF\(^4\) for SGD on different latent feature sizes on three GPU platforms. Furthermore, data reordering brings an extra 8.73%, 8.32% and 10.08% performance improvement over various real-world datasets on K20C, TITAN X and TITAN RTX, respectively. Note that BALS is equally applicable to AMD GPUs, although its implementation is in CUDA and on NVIDIA GPUs.

This paper makes the following contributions.

- We propose a new compressed storage format which organizes a sparse matrix into 2D tiles to facilitate data reuse during matrix factorization.
- We develop a parallel ALS implementation (BALS) on GPGPUs with the storage format to avoid redundant data movements across memory hierarchies.
- We propose a data reordering technique to decrease the processing overhead and further enhance the benefits of data locality.
- We evaluate how BALS performs compared with the baseline implementations on three generations of NVIDIA GPUs and six real-world datasets.

2 Background

This section introduces the ALS-based matrix factorization and analyzes its time and space efficiency.

2.1 ALS-based Matrix Factorization

Matrix factorization aims to learn the factors by minimizing the regularized squared error on the observed ratings,

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

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 $Y$.

The minimization principle of alternating least squares is to keep one fixed while calculating the other: fixing $Y$ to calculate $X$ so as to get vectors $x_u$, and vice versa. Thus 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

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

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

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

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

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

2.2 Algorithm Analysis

The ALS algorithm has three steps, which are (S1) $Y^T Y + \lambda I$, (S2) $Y^T r_u$, and (S3) solving the linear system when updating $x_u$. 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 nonzeros in the current row. Therefore, the total compute cost is $nnz \times f \times (f + 1)$, where $nnz$ denotes the total number of nonzeros in $R$. In terms of memory footprint, we need a matrix smat (sized of $f \times f$) to store the results of $Y^T Y$ 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 $R$. Thus, the total computing cost of S2 is $nnz \times f \times 2$. This step needs a vector $\text{vec}$ sized of $f$ to store the results of $Y^T r_u$, and thus the total
memory footprint for \( m \) rows is \( m \times f \). After obtaining \( \text{smat} \) and \( \text{svec} \), the Cholesky decomposition method is exploited to solve the linear system \( \text{smat} \cdot x_u = \text{svec} \) (S3). The time complexity of updating a row of \( R \) is \( O(f^3) \).

Figure 2 shows that \( Y^T Y \) is the most time-consuming step, which takes on average 90\% of the end-to-end execution time over various latent factors. Therefore, optimizing this step is the focus of our work. Similarly, calculating \( X^T X \) takes the most time of updating an item vector \((y_i)\).

### 3 Motivation

This section starts with our observations coming from the real-world datasets which motive our work.

#### 3.1 Data Reuse

According to our statistics on various recommender datasets, we observe that there exist popular items which have been rated by the majority of users and, at the same time, there exist users who have given a score to most, if not all, items. Figure 3 shows the total number of ratings (a.k.a. popularity) for each movie of the Netflix dataset. We see that some movies are watched by many users (i.e., the movie has a lot of ratings in one column), while other movies by only a few (i.e., there are very few ratings for the corresponding movie). This is why the diagram shows the vertical sparse lines when the movies are popular.

In alternating least squares, updating a user vector \((x_u)\) needs to load the data elements (i.e., \( \text{nnz}_i \) column vectors each sized of \( f \)) from \( Y \) according to the column indexes of the items that the user has rated. When updating a neighbouring user vector \((x_{u+1})\), the same column vectors of \( Y \) may be used again. This occurs when nonzeros from two distinct rows of \( R \) share the same column index. Thus, we can avoid the movement of one column vector in \( Y \), so as to save the memory bandwidth. Our observation from Figure 3 has shown that this is a common case in recommendation datasets. However, the previous implementations let different threads (or thread blocks) work on distinct user vectors \([4, 26]\). The already-loaded column vectors from \( Y \) are either evicted out of caches or be manually overwritten in scratch-pad memories. This is the same when we calculate item vectors. In this work, we aim to exploit such a data feature to avoid redundant movements.

**Defining data reuse.** If there exist two nonzeros of distinct rows sharing the same column index, updating the corresponding row vectors requires the same column vector from \( Y \). Actually, we can avoid one extra data movement of \( f \) data elements when updating the row vectors simultaneously. We regard this as a data reuse. Analytically, a data reuse occurs once Equation 6 holds:

\[
\text{col}_i[\text{row}_j[\text{start}_a + i]] = \text{col}_j[\text{row}_j[\text{start}_b + j]].
\]

where \( a \) and \( b \) are two rows in a row block of \( R \), \( i \) denotes the \( i^{th} \) nonzero of row \( a \) and \( j \) denotes the \( j^{th} \) nonzero of row \( b \). In BALS, we organize all rows into groups, each of which is defined as a row block. Figure 4a illustrates the concept of data reuse, where we group six rows into two row blocks each with three rows. The first nonzero of R0 has the same column index (C1) with the first nonzero of R1. Thus, we only have to load the column vector once (vector 1 shaded in yellow). In the same way, the column vectors (vectors 4 and 5) can be reused in the second row block.

#### 3.2 Data Reordering

Figure 5 shows the nonzero distribution of the six datasets. Each dot of the histogram represents the number of rows with a certain number of nonzeros. We observe that the number of nonzeros varies from very few to tens of thousand, e.g., there are 197,277 rows, each with one nonzero for the YahooMusic R1 dataset. We understand that most elements of the sparse matrix are zeros. Due to the data sparsity, there actually exists many scattered vacant tiles and/or vacant row segments that are all zeros in specific areas of the rating matrix. Dealing with such vacant segments comes at a cost. If a specific 2D tile is vacant, we only have to check it once and then skip it in the outer iteration. But if there are several vacant row segments in the tile, we have to enumerate the inner iteration to detect these vacant row segments for multiple times, which takes more time than checking a vacant tile. For this, we propose to use a data reordering technique by sorting the rows and columns of \( R \) in the descending order of nonzeros. By doing so, many scattered vacant row segments cluster to form new vacant tiles, which reduces the overhead of checking separate vacant row segments.

The red part of Figure 3 shows how the movie ratings distribute of Netflix with data reordering. We see that the popular movies move towards left and leave the unpopular movies on the right. Thus, data reordering can draw the nonzeros to be closer, decrease the number of vacant row segments and create more vacant tiles. Figure 4b illustrates...
the data distribution after reordering ratings. Most nonzeros move towards the top-left corner and the total data reuse times increase by 1. Therefore, it is significant to perform data reordering on the rating matrix to enhance data reuse and decrease the processing overhead of vacant segments.

4 BALS Design and Implementation

This section introduces BALS with its data structures and the ALS-based implementation details.

4.1 BALS Storage Format

To exploit data reuse for matrices with any sparsity structures, we organize the sparse matrix into 2D tiles of the same size \((xb \times yb)\), where \(xb\) and \(yb\) denotes the height and the width of a tile, respectively. Thus, BALS has two tuning parameters: \(xb\) and \(yb\). To facilitate the ALS computation, BALS uses five data structures: value, tile_colidx, tile_ptr, seg_colidx and seg_ptr.

Figure 6 shows an example matrix \(R\) with 9 users, 6 items and 21 nonzero elements. We partition \(R\) with \(xb=2\), \(yb=3\), and there are a total of 10 tiles which are differentiated with distinct colors. The value array stores all the nonzeros of \(R\), and the size of the array equals the total number of nonzeros. The difference from the conventional CSR format is that we store the nonzeros in a tiled fashion. The other four data structures are illustrated in Sections 4.1.1 and 4.1.2. The pseudocode of how to store rating data with our new data format is shown in Algorithm 1.

4.1.1 Tile Information

We use two tile structures (tile_colidx and tile_ptr) to indicate the column vectors to be loaded for each tile, where tile_colidx stores the column indices of the nonzeros within a tile (Lines 10-17 in Algorithm 1) and tile_ptr stores the locations in the tile_colidx array that start a tile (Lines 10-17 in Algorithm 1).

The tile_colidx structure consecutively stores the column indices of each nonzero for the tiles of \(R\). We skip the ones that are redundant across rows of a tile. Note that the indices are the ones in the global space of \(R\), rather than the local space. We regard those columns that have the same column indices in a tile as redundant columns and thus record their first appearance only. The array size equals the number of nonzeros in \(R\) minus the number of redundant columns. In Figure 6, there are four nonzeros in tile 0, which are located in three distinct columns (i.e., columns 0, 1 and 2). Among them, the \((r0, c0)\) element, valued 3, shares an identical column index with the \((r1, c1)\) element, valued 2. Therefore, we take the column vector (from \(Y\)) indexed by the \((r1, c0)\) element as a redundant column. The elements of the tile_colidx array are 0, 0, 1 and 2 for tile 0. As for tile 1, there exists two nonzeros, i.e., the \((r0, c4)\) element and the \((r1, c3)\) element, shown in Figure 6. Since there is no redundant column in this tile, the following two elements of tile_colidx are 4 and 3. In the same way, we can fill tile_colidx for the other tiles. In BALS, tile_colidx determines which column(s) to load from global memory \(Y\) into shared memory \(sY\). Compared with the implementation in [4], we can avoid loading redundant columns from \(Y\).

The tile_ptr structure stores the beginning and ending locations of each tile in tile_colidx. Thus, it can be used to determine the starting and the ending indices of the tile_colidx array to be visited. Figure 6 shows that the starting index of tile 0 in tile_colidx is 0 and the ending address is 2. The size of tile_ptr is the number of tiles plus 1. When two adjacent elements in the tile_ptr array are identical, we know there is no nonzero in the current tile, which is defined as a vacant tile, e.g., tile 4 in Figure 6. Such vacant tiles are skipped when calculating ALS. The last element of tile_ptr denotes the number of nonzeros except the redundant ones, which can be used to calculate redundancy. Figure 6 shows there are 18 elements except the redundant ones. The redundancy equals the number of nonzeros minus the last element of tile_ptr, i.e., \(21 - 18 = 3\).
4.1.2 Segment Information

The tile structures tell us where to load column vectors from \( \mathbf{Y} \). Then we introduce another two segment structures (\( \text{seg}_\text{colidx} \) and \( \text{seg}_\text{ptr} \)) to indicate the locations of column vectors cached in the high-speed on-chip buffers (\( \text{sY} \)). Here we refer a row in a tile to be as a segment.

The \( \text{seg}_\text{colidx} \) structure stores the local indices of the nonzeros for each segment (Lines 18-26 in Algorithm 1). This structure allows us to determine which columns we should use in the local buffer (\( \text{sY} \)). The reason of storing such local indices is that we reorganize column vectors and skip the redundant ones in a tile-wise manner when loading data from global memory (\( \mathbf{Y} \)) to shared memory (\( \text{sY} \)). The local indices in \( \text{seg}_\text{colidx} \) are actually the tile-scoped locations of the corresponding column vectors in \( \text{tile}_\text{colidx} \). In Figure 6, the (r1, c1) element of \( \mathbf{R} \) is of value 4. The global index of the corresponding column vector is located in the third slot (outlined in red) of \( \text{tile}_\text{colidx} \). Thus, the local index of this column vector within the tile is located in the fourth slot (outlined in red) of \( \text{seg}_\text{colidx} \). Each nonzero of \( \mathbf{R} \) corresponds to an element of the \( \text{seg}_\text{colidx} \) array. Therefore, this array is sized of \( nyz \).

The \( \text{seg}_\text{ptr} \) structure stores the starting and the ending locations of each segment in the \( \text{seg}_\text{colidx} \) array (Lines 2-9 in Algorithm 1). In Figure 6, the starting index of segment 1 (marked r1 of tile 0) is 2, indicating that the segment starts with the third element in \( \text{seg}_\text{colidx} \). Then we access the corresponding elements in \( \text{seg}_\text{colidx} \) and load \( \text{seg}_\text{colidx}[2] \) and \( \text{seg}_\text{colidx}[3] \). Meanwhile, we note that the ending index of segment 0 is the starting index of segment 1. When two adjacent elements in \( \text{seg}_\text{ptr} \) are identical, we regard that the tile has a vacant segment. The size of the array equals (\( \text{rows} \times \text{segments} \) + 1).

### Algorithm 1 Storing a Rating Matrix in Our Format

1: \( h ← 0 \), \( v ← 1 \), \( \text{tile}_\text{number} ← \text{rows} \times \text{columns} \times \text{yb} \)
2: for \( i ← 0 \), \( \text{tile}_\text{number} \) do
3:   for \( j ← 0 \), \( \text{xb} \) do
4:     for \( k ← \text{row}_\text{ptr}[j], \text{row}_\text{ptr}[j + 1] \) do
5:       \( \text{seg}_\text{colidx}[h] ← \text{col}_\text{idx}[k] \)
6:     end for
7:   end for
8: end for
9: end for
10: for \( i ← 0 \), \( \text{tile}_\text{number} \) do
11:   for \( j ← \text{seg}_\text{ptr}[i], \text{seg}_\text{ptr}[i + 1] \) do
12:     if \( \text{seg}_\text{colidx}[j] = \text{seg}_\text{colidx}[\text{seg}_\text{ptr}[i \times \text{xb}]] \) then
13:       \( h ← \text{tile}_\text{colidx}[j] \)
14:       \( h ← \text{tile}_\text{colidx}[j + 1] \)
15: end if
16: end for
17: end for
18: for \( i ← 0 \), \( \text{tile}_\text{number} \) do
19:   for \( j ← \text{seg}_\text{ptr}[i], \text{seg}_\text{ptr}[i + 1] \) do
20:     for \( k ← \text{tile}_\text{ptr}[j \times \text{xb}], \text{tile}_\text{ptr}[j \times \text{xb} + 1] \) do
21:       if \( \text{seg}_\text{colidx}[j] = \text{seg}_\text{colidx}[k] \) then
22:         \( \text{seg}_\text{colidx}[h] ← \text{seg}_\text{pr}t[j \times \text{xb}] \)
23:       end if
24:     end for
25: end for
26: end for

4.2 BALS Implementation

4.2.1 Work Partitioning and Thread Mapping

BALS uses a batched implementation and updates batch rows one time. As shown in Figure 7, each batch of \( \mathbf{R} \) is partitioned into 2D tiles, sized of \( \text{xb} \times \text{yb} \), which are stored in our BALS format (Section 4.1). Each tile contains multiple row segments, and the temporary result (\( \mathbf{Y}^T \mathbf{Y} \)) for each row segment is stored in an \( f \times f \) matrix. The overall temporary results for each row will be accumulated from different segments of the row.

Overall, we partition the \( \text{sY} (\text{Y}) \) column vectors into subvectors sized of \( \text{nb} \). In this case, the \( f \times f \) matrix \( \mathbf{Y}^T \mathbf{Y} \) is divided into \( (f/\text{nb}) \times (f/\text{nb}) \) submatrices, each sized of \( \text{nb} \times \text{nb} \). We use a 3D grid of thread blocks: \( (f/\text{nb}, f/\text{nb}, \text{tb}) \), where \( (f/\text{nb}) \times (f/\text{nb}) \) thread blocks are used to deal with the computing task of a row block, as Figure 7 shows. A row block can be further divided into a row of \( 2D \) tiles. The third dimension of the thread-block configuration is \( \text{tb} \), which corresponds the number of row blocks of a batch. Meanwhile, a 3D grid of thread configuration is exploited to work on a single tile: \( (\text{dx}, \text{dy}, \text{dz}) \), where \( \text{dx} \times \text{dy} \) threads are employed to update a row segment of a tile and \( \text{dz} \) corresponds to the number of row segments in the tile. Note that \( \text{dz} \) is limited by the maximum number of threads per block (1024). For instance, when \( \text{dx} = \text{dy} = 4 \), the maximum number of \( \text{dz} \) is 64. Thus, BALS can execute 64 rows of a tile concurrently in this case. In Figure 7, we assume that \( f = 8, \text{nb} = 4, \text{dx} = \text{dy} = 2 \). The first batch of \( \mathbf{R} \) is partitioned into \( 3 \times 5 \) tiles. Thus, we exploit \( 2, 2, 3 \) thread blocks in this example. The way of mapping a block of threads to a single tile is shown in the left bottom of Figure 7. There are a total of \( 2, 2, 4 \) threads in a block and every \( 2 \times 2 \) threads are spawned to work on a row segment.

4.2.2 Implementation Details

Algorithm 2 shows the BALS implementation, which takes three steps to solve \( \mathbf{Y}^T \mathbf{Y} \): (1) loading data into shared memory from global memory (Lines 5-7 in Algorithm 2), (2) calculating \( \mathbf{Y}^T \mathbf{Y} \) with the data staged in shared memory (Lines 8-14), and (3) storing results into global memory (Line 17). Figure 8 shows an example of the BALS kernel, where \( \text{xb} = 6, \text{yb} = 6, \text{nonzeros} = 8, f = 32 \) and \( \text{nb} = 16 \).

The first step is to load columns from \( \mathbf{Y} \) in global memory to \( \text{sY} \) in shared memory. BALS accesses \( \text{tile}_\text{colidx} \)
and tile_ptr to determine which columns to load from Y to sY. We check each row segment of the tile consecutively and obtain the indices of columns to be loaded. For the first row segment in Figure 8, there are two nonzeros having distinct column indices of 1 and 3, respectively. For the second row segment, the nonzero shares the same column index with the second element of the first row segment. Therefore, we only have to load the column vector once into shared memory. In the same way, we enumerate the remaining segments of this tile. By querying tile_colidx and tile_ptr, we only have to load the column vectors (C0, C1, C3, C4) of Y into the corresponding slots (C3’, C0’, C1’, C2’) of sY. The storing order in shared memory is determined by the element order in tile_colidx.

The second step is to calculate sY’ sY, which is shown in the right part of Figure 8. After obtaining sY, BALS visits the seg_colidx and seg_ptr arrays to determine which columns of sY to use for the calculation. There are two elements in the first row segment, so the first and second elements of seg_ptr array are 0 and 2, respectively. The seg_colidx array stores the local column indices of each nonzero in sY. For example, row segment 0 has two elements, whose column indice corresponds to the first and second columns of sY. Then, we use four thread blocks to calculate sY^T sY (f * f matrix) for each segment of a tile.

Algorithm 2 BALS Implementation

1: procedure BALS(Y, xb, yb, data structure; Y^T Y)
2:   tile_number ← columns/yb
3:   for tn ← 1, tile_number do
4:     if tile_ptr[tn] > tile_ptr[tn-1] then → Check vacant tile
5:       for c ← tile_ptr[tn], tile_ptr[tn+1] do → Load data
6:         sY[c−tile_ptr[tn]] ← Y[c]
7:       end for
8:     end for
9:     for r ← 0, xb do → Calculate sY^T × sY
10:    if seg_ptr[r+1] > seg_ptr[r] then
11:      for k ← seg_ptr[r], seg_ptr[r+1] do
12:       r’ = sY[seg_colidx[k]] * sY[seg_colidx[k]]
13:      end for
14:    end if
15:   end for
16:  end for
17:  Y^T Y ← r → Store temporary results
18: end procedure

In BALS, we allocate registers r sized of nb × nb × [xb/dx] to store and accumulate the temporary results of all the tiles. The third step is to save the temporary results from r back to the global memory.

4.3 BALS Tile Size Selection

Due to the uneven data distribution of the rating matrix, using BALS will not bring a performance improvement when there is insufficient data reuse in 2D tiles. Thus, we need to pick a right block size of xb and yb for the tiles of a given sparse matrix. Through running a large number of experiments, we observe that BALS can achieve its best performance when nb is the greatest common divisor of f and dx (dy) is the lowest common multiple (greater than 2) of f.

On the other hand, the parameter yb is dependent on nb and the size of shared memory. For instance, there are 48-KB shared memory on K20C and each element sized of float takes up four bytes. Therefore, the maximum value of yb is 192 when f=64 and nb=32. In this case, we use four thread blocks to deal with the f × f matrix and need to allocate two local buffers sY1 and sY2. That is,

$$48KB/(4B \times 2 \times 32) = 192.$$ 

However, for this problem, the choice yb=192 does not yield the best performance. We argue that a larger on-chip memory will lead to a better performance. Figure 9 shows the performance when using various tile configurations on two GPUs (both with 48KB shared memory). We observe that yb=192, nb=32 and dx=dy=4 is the best configuration while f=64, xb=1280 on both K20C and TITAN X.

4.4 Data Reordering

By reordering rows or columns according to the number of nonzeros in each row or column, the nonzeros will walk towards the top-left corner of the rating matrix. Thus, we can further increase the benefits of data reuse, and eliminate the overhead of processing vacant segments. Algorithm 3 demonstrates how we reorder rows.

The inputs of data reordering algorithm are the original rating matrix R in the conventional CSR format. The output is the reordered matrix R' represented with three updated data structures (dr_value, dr_rowptr, and dr_colidx). We first initialize the intermediate variables and the temporary arrays to be zero. Then, we obtain the longest row according to the nonzeros of each row in R by calculating row_ptr[u+1] − row_ptr[u]. After that, we store its information into
Algorithm 3 Data Reordering

1: procedure DATA REORDERING(R (val, row_ptr, col_idx); R′
   (dr_val, dr_rowptr, dr_colidx))
2:    dr_val, dr_rowptr, dr_colidx, h ← 0
3:    for u ← 1, m do
4:       find longest row lr ← row_ptr[u + 1] − row_ptr[u]
5:       for i ← row_ptr[lr], row_ptr[lr + 1] do
6:          dr_val[h][i] ← value[i]
7:       dr_colidx[h] ← col_idx[i]
8:       h ← h + 1
9:    end for
10:   dr_rowptr[u + 1] ← dr_rowptr[u] + length
11: end procedure

This section introduces the hardware and software platforms, and describes the real-world recommender datasets.

GPU Hardware and Software. NVIDIA Tesla K20C GPU, TITAN X Pascal and TITAN RTX Turing are utilized for the following experiments. Tesla K20C contains 13 streaming multiprocessors (SM), each with 192 CUDA cores. The theoretical peak floating point performance of K20C is 3.52 Tflops in single precision and 1.17 Tflops in double precision. TITAN X Pascal has 3840 CUDA cores spread across 30 streaming multiprocessors (SM) and six graphics processing clusters (GPCs) from which 3584 are enabled on the TITAN X Pascal. TITAN RTX Turing includes 4608 CUDA cores across 72 SMs, 576 Tensor cores, 72 RT cores, 288 texture units, and 36 PolyMorph engines. Not only does Titan RTX sport more CUDA cores than GeForce RTX 2080 Ti, it also offers a higher GPU Boost clock rating (1,770 MHz vs. 1,635 MHz). As such, its peak single-precision rate increases to 16.3 Tflops. Besides, we use CUDA v7.5 for K20C, CUDA v8.0 for TITAN X and CUDA v10.2 for TITAN RTX respectively, which is taken as the communication backbone between CPU and GPU.

Input Datasets. We use six datasets (Movielens 10M, Movielens 20M, Netflix\textsuperscript{3}, YahooMusic R1, Sls, Rucci\textsuperscript{5}) to evaluate the ALS performance. The format of each dataset is (userID, itemID, rating). We pre-process each dataset according to this format. The details of the datasets are shown in Table 1. Note that m is the number of users, n is the number of items, and nnz is the number of the nonzero entries in the rating matrix R. The sparsity of a rating matrix is calculated by nnz/(m×n). In the experiments, the nonzero entries are in single precision.

Competitive Approaches. Rodrigues et al. introduce a basic ALS implementation in CUDA [20], where each GPU thread updates a row x\textsubscript{u} of X (Equation 4) or a column y\textsubscript{i} of Y (Equation 5). Thus, the implementation has a total of m (or n) independent tasks and at most m (or n) threads can run concurrently. As one GPU thread is used to update a row of the X matrix, all the temporary data of Y\textsuperscript{T}Y is allocated dynamically in the kernel function. However, when f becomes large, there is insufficient global memory space remained for dynamic allocation and thus the kernel failed to run. Therefore, the implementation does not scale well over the latent factor. Given that their work is a baseline implementation and only supports f=10, we only compare other three ALS implementations in this section.

CuMF uses a thread block to update a row of the X matrix or a column of the Y matrix [26]. The entire task of calculating Y\textsuperscript{T}Y is partitioned into multiple tiles, each sized of 10 × 10. Then CuMF lets each thread work on such a data tile. Instead of using a loop to iterate a 10 × 10 data tile, it fully unrolls the loop and allocates 100 registers to store the temporary results of smat. Taking f=10 as an example, CuMF uses only one thread to calculate the temporary results of Y\textsuperscript{T}Y.

Gates et al. present an ALS solver in CUDA [4]. They leverage a batched implementation and use a 3D grid of thread blocks, (\lceil \frac{f}{nb} \rceil, \lceil \frac{f}{nb} \rceil, batch). The product of Y\textsuperscript{T}Y is an f × f matrix for each row of R, and this task is divided into sub-tiles sized of nb × nb. Each sub-tile is mapped to a thread block. The approach moves the corresponding columns of Y sized of nb × f into shared memory s\textsuperscript{Y} (or s\textsuperscript{Y}T) and exploits ((f/nb) × ((f/nb) thread blocks to calculate Y\textsuperscript{T}Y. Meanwhile, they use a 2D grid of thread configuration (i.e., each thread block has dx × dy threads) and allocates a register file sized of (nb/dx) × (nb/dy) for each thread to save the temporary results.

6 Performance Evaluation

This section reports how well BALS performs, by comparing BALS with the state-of-the-art ALS implementations, and evaluating the performance impact of the reordering techniques and BALS tuning parameters.

6.1 Comparison to State-of-the-Art Implementations

Figure 10 presents the comparison with other two state-of-the-art ALS implementations (Gates’ and CuMF) over different f (ranging from 10 to 100) on six datasets. BALS generally runs the fastest among the three implementations and Gates’ implementation runs the second on the three GPUs. Specifically, our BALS implementation runs, on average, 1.26×, 1.46×, 1.30×, 2.19×, 2.30× and 2.25× faster than Gates’ for Movielens 10M, Movielens 20M, Netflix, YahooMusic R1, Sls and Rucci on K20C, respectively.
CCD (x)

SGD iterations. On SGD. That is, BALS converges with around 5 CCD [16] for CCD++ and cuMF SGD. Figure 12

CCD and BALS over different latent feature sizes for SGD [30].

CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

CuMF_CCD scale linearly, while BALS fluctuates slightly due to the different best observed parameters chosen for different f. Table 2 lists the average speedups of BALS over CDMF and CuMF_CCD, which shows different level of performance speedups on six datasets.

We also compare BALS to cuMF_SGD when f=128. The performance results (in terms of GFlops) show that BALS achieves an average speedup of 5.3× (5.6×) over six datasets on TITAN RTX (K20C), compared to cuMF_SGD. Figure 12 shows the RMSE comparison with regards to the training time on TITAN RTX with Netflix dataset. Since ALS requires more computations per iteration, it runs slower than SGD. We also see that, the execution time of a single ALS iteration, while each red square dot represents 10 SGD iterations).

6.2 Comparison to SGD and CCD++

This section compares BALS with three state-of-the-art CCD and SGD implementations: CDMF [32], CuMF_CCD [16] for CCD++ and cuMF_SGD for SGD [30].

Figure 11 shows the performance comparison of CDMF, CuMF_CCD and BALS over different latent feature sizes (from 8 to 96) on K20C, TITAN X and TITAN RTX. We observe that BALS outperforms CDMF and CuMF_CCD over different latent feature sizes on six datasets, while not changing the prediction accuracy (i.e., RMSE). We see that

The performance improvements (geometric mean) of BALS over Gates’ implementation reach 24%, 28%, 11%, 28%, 15% and 19% on TITAN X, and 20%, 30%, 11%, 49%, 31% and 39% on TITAN RTX on these six datasets, respectively.

When f is small, the performance gap between BALS and Gates’ implementation over various datasets is small. But when f increases, the performance gap becomes larger. This is because a larger f allows for better data reuse, e.g., BALS achieves the maximum speedup of 4.38× over Gates’ when f=100 on K20C for YahooMusic R1. Note that CuMF exploits a specially customized kernel for the case when f = 100, which leads to a dramatic performance improvement.

<table>
<thead>
<tr>
<th>Platform</th>
<th>Comparison</th>
<th>ML10M</th>
<th>ML20M</th>
<th>NTFX</th>
<th>YHR1</th>
<th>SLS</th>
<th>RUCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20C</td>
<td>vs. CDMF</td>
<td>7.10</td>
<td>6.89</td>
<td>9.37</td>
<td>15.26</td>
<td>3.04</td>
<td>21.64</td>
</tr>
<tr>
<td></td>
<td>vs. CuMF_CCD</td>
<td>2.75</td>
<td>2.31</td>
<td>4.04</td>
<td>9.09</td>
<td>1.79</td>
<td>11.39</td>
</tr>
<tr>
<td>X</td>
<td>vs. CDMF</td>
<td>6.13</td>
<td>6.64</td>
<td>7.35</td>
<td>5.21</td>
<td>5.39</td>
<td>5.79</td>
</tr>
<tr>
<td></td>
<td>vs. CuMF_CCD</td>
<td>3.50</td>
<td>7.18</td>
<td>9.01</td>
<td>7.28</td>
<td>9.68</td>
<td>17.88</td>
</tr>
<tr>
<td>RTX</td>
<td>vs. CDMF</td>
<td>2.51</td>
<td>2.44</td>
<td>3.13</td>
<td>3.04</td>
<td>4.45</td>
<td>3.80</td>
</tr>
<tr>
<td></td>
<td>vs. CuMF_CCD</td>
<td>2.09</td>
<td>3.86</td>
<td>3.22</td>
<td>5.53</td>
<td>10.03</td>
<td>14.40</td>
</tr>
</tbody>
</table>

Table 2: The speedup of BALS over CDMF and CuMF_CCD (x)
Fig. 13. The performance impact over \( \text{x}_b \) and \( \text{y}_b \) and the performance comparison (GFlops) of BALS and Gates’ implementation (labeled on the colorbar) before data reordering on various datasets, where \( f=32, \ n_b=16, \ dx=dy=4 \). X-axis is \( \text{x}_b \), from 256 to 7936, step increase is 512. Y-axis is \( \text{y}_b \), from 144 to 384, step increase is 16. A more visible version is available at: https://bit.ly/35YUQvW.

Fig. 14. The performance impact over \( \text{x}_b \) and \( \text{y}_b \) and the performance comparison (GFlops) between BALS and Gates’ implementation (labeled on the colorbar) after data reordering on various datasets, where \( f=32, \ n_b=16, \ dx=dy=4 \). X-axis and y-axis keep the same as Figure 13. A more visible version is available at: https://bit.ly/35YUQvW.

6.3 Impact of \( \text{x}_b \) and \( \text{y}_b \)

The tile size \( \text{x}_b \) and \( \text{y}_b \) have a dramatic performance impact on BALS, which determines the data reuse. Figures 13 shows the performance impact of \( \text{x}_b \) and \( \text{y}_b \) on various datasets and three GPUs. BALS’s performance changes gradually with \( \text{x}_b \) and \( \text{y}_b \), and the best tile size typically appears on the top-left corner of heatmaps. Theoretically, for a fixed \( \text{y}_b \) (\( \text{x}_b \)), the data reuse linearly increases with \( \text{x}_b \) (\( \text{y}_b \)). However, in order to achieve the best performance in BALS, it is necessary to balance the benefit from column reuse and the computing cost due to increased number of rows in a specific tile. As Figure 13 heatmaps show, the performance changes from red to blue in most of cases when \( \text{x}_b \) increases, which indicates that the calculation cost caused by increasing \( \text{x}_b \) overtakes the benefit of data reuse. Also, this observation hints best parameter choices. Note that there exists a dramatic performance change for Movielens 20M and YahooMusic R1 on K20C and TITAN X, when \( \text{x}_b \) ranges from 256 to 1280, whereas this change is minor on TITAN RTX. This comes from these datasets differing from
very little for them when datasets and measure the performance benefits of skipping the percentage of vacant tiles and segments of the six of vacant tiles/segments. Thus, we introduce an inspection Rucci performance improvement of 60.37% Using the inspection mechanism yields the largest perfor-
ing in the six datasets. As a result, we obtain various levels percentage in rating matrices, with an average 35.88% of see that the vacant tiles and vacant segments occupy a large vacant tiles and vacant segments, i.e., those segments within 6.5 Impact of Data Reordering

The percentage of vacant tiles and segments and performance gain

<table>
<thead>
<tr>
<th>Before Reordering</th>
<th>After Reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacant Tiles</td>
<td>Vacant Segments</td>
</tr>
<tr>
<td>ML10M</td>
<td></td>
</tr>
<tr>
<td>43.67%</td>
<td>49.44%</td>
</tr>
<tr>
<td>ML20M</td>
<td></td>
</tr>
<tr>
<td>18.30%</td>
<td>63.85%</td>
</tr>
<tr>
<td>NTEX</td>
<td></td>
</tr>
<tr>
<td>2.36%</td>
<td>24.10%</td>
</tr>
<tr>
<td>YHR1</td>
<td></td>
</tr>
<tr>
<td>52.26%</td>
<td>45.73%</td>
</tr>
<tr>
<td>SLS</td>
<td></td>
</tr>
<tr>
<td>26.35%</td>
<td>71.85%</td>
</tr>
<tr>
<td>RUCI</td>
<td></td>
</tr>
<tr>
<td>72.32%</td>
<td>26.92%</td>
</tr>
</tbody>
</table>

6.4 Vacant Tiles and Segments Analysis

The rating matrix $R$ is divided into multiple 2D tiles in BALS. Due to the sparsity of $R$, there exists a large amount of vacant tiles/segments. Thus, we introduce an inspection mechanism of identifying vacant tiles/segments. We count the percentage of vacant tiles and segments of the six datasets and measure the performance benefits of skipping them when $f=32$ (Table 3). Note that we differentiate the vacant tiles and vacant segments, i.e., those segments within a vacant tile are excluded from pure vacant segments. We see that the vacant tiles and vacant segments occupy a large percentage in rating matrices, with an average 35.88% of vacant tiles and 46.98% of vacant segments before reordering in the six datasets. As a result, we obtain various levels of performance improvement when skipping these vacancy. Using the inspection mechanism yields the largest performance improvement of 60.37% Rucci, while the gain is very little for Netflix. This is because Netflix is a denser matrix among the recommender datasets, whose sparsity is shown Table 1). The percentages of vacant tiles and vacant segments are much less than those of other datasets.

6.5 Impact of Data Reordering

BALS clusters the nonzeros of the rating matrix to exploit data locality with the data reordering technique. Table 3 shows that the proportions of vacant tiles and segments change significantly after reordering. The percentages of vacant tiles for most datasets are larger, while the percentages of vacant segments drop except Netflix and Sls. When the nonzeros cluster as much as possible, the scattered vacant segments without reordering form new vacant tiles, and thus we have more vacant tiles and fewer vacant segments with the reordering technique.

Figures 13 and 14 show the performance impact of data reordering. First, the performance ($x_b, y_b$) configurations (red) move towards the left parts after data reordering. Thus, the best tile configuration changes by reordering rows or columns. For Movielens 10M, BALS performs the best when $x_b=1280$, $y_b=384$ before reordering on K20C and $x_b=768$, $y_b=384$ on TITAN X, whereas the best configuration is of $x_b=256$, $y_b=384$ after reordering. Second, the best observed tile configuration differs across datasets. The best performance is achieved at $x_b=2816$, $y_b=384$ on K20C before reordering and at $x_b=768$, $y_b=384$ after reordering for Movielens 20M. Third, we summarize that achieving the best performance without data reordering requires more rows within a tile to exploit data reuses. Fourth, enlarging $x_b$ cannot improve the performance after reordering $R$, which leads to broader blue fields in Figure 14 compared with Figure 13. This is because most data move to left upper part with reordering, and the data reuse occurs in most left upper part. Therefore, it has much less data and no extra significant data reuse by using a larger $x_b$, which leads to a lower performance.

Our data reordering technique brings further performance improvement, compared to Gates’ implementation. Table 4 lists the performance improvements of the data reordering technique over the six datasets in BALS when compared with the performance without data reordering. We obtain different levels of performance improvements, with an average improvement of 8.73%, 8.32% and 10.08% on K20C, TITAN X and TITAN RTX, respectively. The performance improvement for Movielens 10M increases by 23.68% on K20C, 19.87% on TITAN X and 14.67% on TITAN RTX. However, we observe that the data reordering technique brings a rather small performance improvement for the Sls and Rucci datasets. This is because their matrix shape is more structured, and their nonzeros distribute more evenly across rows or columns than the other four datasets.

6.6 Impact of Feature Space Size

The first step of $Y^T Y$ is to load corresponding column vectors each sized of $f$ from $Y$ to $s Y$. A larger $f$ allows more data to be loaded to on-chip memories. And using BALS can avoid redundancy $\times f$ repeated data loads. As a result, BALS can avoid a large amount of data loads from off-chip to on-chip when $f$ is large. Figure 15 shows the performance impact of the feature space size ($f$ ranging from 8 to 96) on the three platforms. We note the BALS performance trend being mostly independent of the architecture, while there exists a significant difference in the actual performance numbers. We observe minor fluctuations between $f=8$ to 56 on K20C and TITAN RTX. But after $f=56$, it is obvious that BALS has a dramatic performance improvements due to more data uses and fewer data loads.

<table>
<thead>
<tr>
<th>Before Reordering</th>
<th>After Reordering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vacant Tiles</td>
<td>Vacant Segments</td>
</tr>
<tr>
<td>ML10M</td>
<td></td>
</tr>
<tr>
<td>43.67%</td>
<td>49.44%</td>
</tr>
<tr>
<td>ML20M</td>
<td></td>
</tr>
<tr>
<td>18.30%</td>
<td>63.85%</td>
</tr>
<tr>
<td>NTEX</td>
<td></td>
</tr>
<tr>
<td>2.36%</td>
<td>24.10%</td>
</tr>
<tr>
<td>YHR1</td>
<td></td>
</tr>
<tr>
<td>52.26%</td>
<td>45.73%</td>
</tr>
<tr>
<td>SLS</td>
<td></td>
</tr>
<tr>
<td>26.35%</td>
<td>71.85%</td>
</tr>
<tr>
<td>RUCI</td>
<td></td>
</tr>
<tr>
<td>72.32%</td>
<td>26.92%</td>
</tr>
</tbody>
</table>

TABLE 4

The performance improvements of data reordering

<table>
<thead>
<tr>
<th></th>
<th>ML10M</th>
<th>ML20M</th>
<th>NTEX</th>
<th>YHR1</th>
<th>SLS</th>
<th>RUCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20C</td>
<td>23.68%</td>
<td>11.72%</td>
<td>9.77%</td>
<td>6.26%</td>
<td>0.64%</td>
<td>0.33%</td>
</tr>
<tr>
<td>TITAN X</td>
<td>19.87%</td>
<td>11.92%</td>
<td>8.96%</td>
<td>5.87%</td>
<td>1.29%</td>
<td>2.02%</td>
</tr>
<tr>
<td>TITAN RTX</td>
<td>14.67%</td>
<td>14.07%</td>
<td>10.35%</td>
<td>20.38%</td>
<td>0.83%</td>
<td>0.18%</td>
</tr>
</tbody>
</table>

TABLE 3

The percentage of vacant tiles and segments and performance gain

<table>
<thead>
<tr>
<th></th>
<th>ML10M</th>
<th>ML20M</th>
<th>NTEX</th>
<th>YHR1</th>
<th>SLS</th>
<th>RUCI</th>
</tr>
</thead>
<tbody>
<tr>
<td>K20C</td>
<td>23.68%</td>
<td>11.72%</td>
<td>9.77%</td>
<td>6.26%</td>
<td>0.64%</td>
<td>0.33%</td>
</tr>
<tr>
<td>TITAN X</td>
<td>19.87%</td>
<td>11.92%</td>
<td>8.96%</td>
<td>5.87%</td>
<td>1.29%</td>
<td>2.02%</td>
</tr>
<tr>
<td>TITAN RTX</td>
<td>14.67%</td>
<td>14.07%</td>
<td>10.35%</td>
<td>20.38%</td>
<td>0.83%</td>
<td>0.18%</td>
</tr>
</tbody>
</table>

We observe major fluctuations between $f=8$ to 56 on K20C and TITAN RTX. But after $f=56$, it is obvious that BALS has a dramatic performance improvement due to more data uses and fewer data loads.

The performance of Gates’ implementation [4] is labeled on the colorbar of Figure 13. We see that BALS can outperform Gates’ implementation over various datasets on K20C, TITAN X and TITAN RTX. The performance improvements on TITAN RTX based on Gates’ implementation are up to 14.6%, 9.3%, 13.2%, 38.6%, 23.9% and 53.2% for Movielens 10M, Movielens 20M, Netflix, YahooMusic R1, Sls and Rucci, respectively. And BALS obtains the largest performance improvement (by 99%) on TITAN X for YahooMusic R1 when $f=32$, while using the same configuration can achieve only half of that on K20C. For Sls and Rucci, BALS achieves an average performance improvement of 49.2% on TITAN X and 29.8% on K20C compared to Gates’ implementation.

6.6 Impact of Feature Space Size

The first step of $Y^T Y$ is to load corresponding column vectors each sized of $f$ from $Y$ to $s Y$. A larger $f$ allows more data to be loaded to on-chip memories. And using BALS can avoid redundancy $\times f$ repeated data loads. As a result, BALS can avoid a large amount of data loads from off-chip to on-chip when $f$ is large. Figure 15 shows the performance impact of the feature space size ($f$ ranging from 8 to 96) on the three platforms. We note the BALS performance trend being mostly independent of the architecture, while there exists a significant difference in the actual performance numbers. We observe minor fluctuations between $f=8$ to 56 on K20C and TITAN RTX. But after $f=56$, it is obvious that BALS has a dramatic performance improvement due to more data uses and fewer data loads.
7 Related Work

This section discusses four matrix factorization algorithms and their state-of-the-art parallel implementations.

The ALS solver. GraphLab implements ALS by distributing matrix on multiple machines for large matrices, which results in heavy cross-node traffic and high network bandwidth [12]. Spark MLlib leverages partial matrix replication to parallelize ALS [14]. CuMF is a CUDA-based matrix factorization library, which implements memory-optimized ALS to solve large-scale matrix factorization on either single or multiple GPUs. Gates et al. formulate ALS as a mix of cache-optimized algorithm-specific kernels and batched Cholesky factorization [9], and accelerate it on GPUs and multi-threaded CPUs [4]. Zhou et al. introduce a new parallel algorithm ALS-WR (weighted regulation) for large-scale problems by using parallel Matlab on a linux cluster [34].

The CCD solver. Yu et al. propose a scalable method (CCD++), which has a different update sequence from the conventional CCD (Cyclic Coordinate Decent) and updates rank-one factors one by one. The algorithm has two parallel implementations: one for multi-core shared memory systems and the other for distributed systems [33]. Recently Nisa et al. improve the CCD++ method on GPUs with loop fusion and tiling [16]. Yang et al. present an efficient and portable CDMF solver on multi-core CPUs and GPUs [32]. They balance the factorization loads by organizing the nonzeros of rating matrices.

The SGD solver. Paine et al. present an asynchronous SGD to speed up the neural network training on GPUs [18]. In [1, 35], 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, that is more efficient than the delayed update scheme [19]. DSGD (Distribute SGD) partitions the ratings matrix into several blocks and updates a set of independent blocks concurrently [5]. Kaleem et al. show that the parallel SGD can run efficiently on GPU, and their GPU implementation is comparable to a 14-thread CPU implementation [6]. CuMF_SGD is a CUDA-enabled SGD solution for large-scale matrix factorization problems, which uses two workload scheduling schemes and a partitioning scheme to utilize multiple GPUs [31]. Factorbird uses a parameter server to scale models that exceed the memory of an individual machine, and employs a lock-free learning with a special partitioning scheme to reduce conflicting updates [23]. Sallinen et al. present a scalable, communication-avoiding implementation of SGD and demonstrate near-linear scalability on a 14-core system [21].

The SVD solver. The recommendation problem is how to compute a mapping of items and users to factor vectors [8, 29]. In the collaborative filtering domain, singular value decomposition (SVD) is a well-established technique of identifying latent feature factors. However, the conventional SVD is often inapplicable in matrix factorization of the recommendation field due to the high percentage of missing entries in the sparse user-item matrix. Moreover, overfitting can occur if the sparse matrix is addressed carelessly. Prior works leverage an approach of simply ignoring the missing ratings in the sparse matrix, and directly modeling the observed ratings. Ma proposed four variants of SVD to solve large-scale matrix of collaborative filtering instead of the conventional SVD [2]. They observed that complete incremental learning which updates feature values after scanning a single training score of R, is the best choice for collaborative filtering with millions of training instances. This method minimizes the object 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 requires a total of nnz iterations.

8 Conclusion

In this work, we have proposed BALS, an efficient implementation of the least squares algorithm for large-scale matrix factorization on GPUs. Through analyzing the algorithm and the recommendation datasets, we observed that there exist many repeated data loads during the ALS factorization. BALS aims to improve the data-moving efficiency across memory hierarchies. At the core of BALS is a new compressed blocked storage format for sparse matrices, which is used to build a blocked ALS implementation. We have further developed a data reordering technique to enhance the data locality. Our experiments reveal that our approach can outperform state-of-the-art implementations. Our implementation generally runs faster than Gates’ implementation with a speedup of up to $2.08 \times$ on K20C, $3.72 \times$ on TITAN X and $3.13 \times$ on TITAN RTX. BALS also outperforms CDMF, CuMF_CCD and CuMF_SGD over different latent feature sizes on K20C. Furthermore, reordering brings another performance improvement of up to 23.68% on K20C, 19.87% on TITAN X and 20.38% on TITAN RTX.

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. 2018YFB0204301, the
National Natural Science Foundation of China under Grant No. 61972408 and 61972415, the Science Challenge Project under Grant No. ZZST2016002, and the Science Foundation of China University of Petroleum, Beijing under Grant No. 2462019YJRC004, 2462020XKJS03.

REFERENCES