HEALS: A Parallel eALS Recommendation System on CPU/GPU Heterogeneous Platforms

Qihan Wang*, Wei Niu*, Li Chen†, Ruoming Jin‡, and Bin Ren*

*Department of Computer Science, William & Mary, Williamsburg, VA
Email: {qwang19, wniu}@email.wm.edu, bren@wm.edu
†iLambda, Inc
Email: lchen@ilambda.com
‡Department of Computer Science, Kent State University, Kent, Ohio
Email: rjin1@kent.edu

2021 IEEE 28th International Conference on High Performance Computing, Data, and Analytics (HiPC) | 978-1-6654-1016-8/21/$31.00 ©2021 IEEE | DOI: 10.1109/HIPC53243.2021.00039

***Abstract*—Alternating Least Square (*ALS*) is a classic algorithm to solve matrix factorization widely used in recommendation systems. Existing efforts focus on parallelizing ALS on multi-/many-core platforms to handle large datasets. Recently, an optimized ALS variant called *eALS* was proposed, and it yields significantly lower time complexity and higher recommending accuracy than ALS. However, it is challenging to parallelize eALS on modern parallel architectures (e.g., CPUs and GPUs) because: 1) eALS' data dependence prevents it from fine-grained parallel execution, thus eALS cannot fully utilize GPU's massive parallelism, 2) the sparsity of input data causes poor data locality and unbalanced workload, and 3) its large memory usage cannot fit into GPU's limited on-device memory, particularly for real-world large datasets.**

This paper proposes an efficient CPU/GPU heterogeneous recommendation system based on fast eALS *for the first time* (called HEALS) that consists of a set of system optimizations. HEALS employs newly designed architecture-adaptive data formats to achieve load balance and good data locality on CPU and GPU. HEALS also presents a CPU/GPU collaboration model that can explore both task parallelism and data parallelism. HEALS also optimizes this collaboration model with data communication overlapping and dynamic workload partition between CPU and GPU. Moreover, HEALS is further enhanced by various parallel techniques (e.g., loop unrolling, vectorization, and GPU parallel reduction). Evaluation results show that HEALS outperforms other state-of-the-art baselines in both performance and recommendation quality. Particularly, HEALS achieves up to $5.75\times$ better performance than a state-of-the-art ALS GPU library. This work also demonstrates the possibility of conducting fast recommendations on large datasets with constrained (or relaxed) hardware resources, e.g, a single CPU/GPU node.

I. Introduction

A recommendation system is a fundamental building block of many real-world applications ranging from online shopping, social networking, to short video sharing and media business, etc [1], [2]. With rapidly increasing data volumes, exploiting more efficient (and accurate) recommendation models has been attracting attention from the fields of information retrieval, machine learning, and high-performance computing fields [3], [4] etc.

Matrix factorization and its variants [5]–[10] have been widely studied and among the most prevalent approaches for recommendation. In essence, the input data forms a sparse matrix whose elements either represent ratings (explicit), or implicit feedback from users (such as click or add-cart). The goal of the factorization aims to produce two (dense) matrices, a user matrix and an item matrix, whose product can be used to recover the sparse matrix. Especially, the missing elements in the sparse matrix can be then approximated by the multiplication between the user and item matrices. We can then recommend the top-rated missing items to each user.

There are different optimization methods for matrix-factorization based recommendation, and among them, **Alternating Least Square (ALS)** [11]–[13] is one of the state-of-the-art methods due to its simplicity, ease of implementation, and (recommendation) accuracy. It is also rather efficient if the data is not very large. Intuitively, ALS will fix one matrix (for instance, the user matrix), and then optimize the other matrix (for instance, the item matrix); then we will fix the latter matrix and optimize the first one. This alternating optimization will perform iterative until it converges.

However, when the data size becomes larger (i.e., the number of users and the number of items), ALS also requires computation efficiency and memory capacity. To efficiently train large datasets [14], various ALS-based parallel recommendation models are implemented on many-core architectures, such as LIBMF [7] and CuMF libraries [8], [15]. To further accelerate ALS algorithm, recently He's work [9] proposes an optimized ALS algorithm, fast element-wise Alternating Least Square (eALS) algorithm and shows its outperforming convergent speed (and recommendation accuracy) compared with other MF-based recommendation models.

However, it is challenging to parallelize eALS on modern parallel architectures like GPUs [16]: First, data dependence in the fast eALS algorithm prevents it from fine-grained workload partition, thus eALS cannot fully utilize GPU's massive parallelism. Second, implicit information constitutes large sparse matrices, resulting in poor data locality and workload unbalance. Third, addressing large datasets is challenging for GPUs with limited on-device memory.

Targeting these challenges, this work proposes a CPU/GPU heterogeneous recommendation framework (HEALS) based on implementing an efficient parallel fast eALS algorithm.

2640-0316/21/$31.00 ©2021 IEEE
DOI 10.1109/HiPC53243.2021.00039

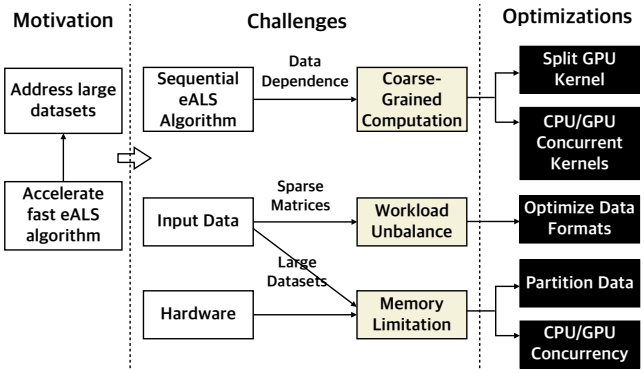

Fig. 1. **Problem Statement: Motivation, Challenges, and Optimizations.**

HEALS employs a hybrid CPU/GPU collaboration model to alleviate the impact of data dependence. HEALS accomplishes not only data movement overlapping for distinct GPU kernels but concurrent kernels between CPU and GPU as well. Moreover, HEALS reorganizes both sparse and dense matrices to new architecture-adaptive formats to improve workload unbalance and data locality. HEALS partitions data into several chucks to handle large datasets. HEALS also leverages several classic system techniques to further accelerate recommendation. The key contributions are summarized as follows:

- We propose a CPU/GPU heterogeneous recommendation system, HEALS, based on an efficient parallel eALS-based matrix factorization *for the first time*.
- We design an architecture-adaptive data format for GPU and CPU to solve workload unbalance. Additionally, HEALS unfolds and packs dense matrices for better data locality.
- We present a hybrid CPU/GPU collaboration model incorporating data parallelism, task parallelism, and overlapped data transfer. It also adopts a data partition adjustment approach to balance workload between concurrent kernels.
- We apply crucial hardware-based accelerating techniques to further accelerate kernel computation, including loop transformation and GPU parallel reduction.

HEALS is extensively evaluated on four datasets by comparing with three other state-of-the-art works. To further validate its prediction accuracy and recommendation quality, HEALS is also evaluated on two metrics, Root Mean Square Error (RMSE) and Normalized Discounted Cumulative Gain (NDCG). Evaluation results demonstrate that HEALS outperforms other ALS-based parallel libraries. Particularly, it runs $5.75\times$ faster than CuMF (a state-of-the-art GPU library), and $15.7\times$ faster than LIBMF (a state-of-the-art CPU library), respectively. HEALS also demonstrates the possibility of performing fast recommendations on large datasets with constrained (or relaxed) hardware resources.

II. Problem Statement

Fig. 1 illustrates the overview of the problem statement. The overall motivation is to accelerate recommendations for large datasets by designing a parallel fast eALS algorithm with the help of the superior computing capability of a heterogeneous CPU and GPU system. This section introduces three prerequisites of this work, explains the corresponding challenges, and summarizes some specific solutions. More specifically, it analyzes three sub-topics: coarse-grained computation, workload unbalance, and memory limitation.

A. Coarse-Grained Computation

As a variant of ALS, the sequential eALS algorithm [9] offers a theoretical guarantee of low computation cost and fast convergence, inspiring us to build our efficient parallel recommendation system based on this state-of-the-art algorithm. However, the original eALS algorithm leads to data dependence, which limits the performance of parallelism. Part of the kernel computation is prone to coarse-grained. More specifically, the kernel computation consists of three loops, and data dependence exists in the inner loops. We attempt to relax this inner-loop dependence by unrolling the inner loop, but this straightforward method significantly degrades the accuracy because it violates the theoretical guarantee. Therefore, this work focuses on utilizing a set of **system optimizations** to alleviate the influence of data dependence. This work splits the kernel computation into a CPU part and a GPU part and designs a CPU/GPU collaborative processing model.

B. Workload Unbalance

Workload unbalance is another bottleneck when calculating sparse matrices in eALS. The computation cost of different rows may vary dramatically for sparse matrices, directly leading to workload unbalance. A straightforward solution is to designate different numbers of threads to compute different rows; however, it will cause a significant overhead of threads scheduling. It is also time-consuming to quantify the workload of each row and decide which threads to use, especially for large datasets. Comparing with the above approach, it is more efficient to reorganize input data in advance. Thus, this work designs new data formats and divides the original data into groups with a more balanced workload. Threads are able to directly deal with reorganized groups, without any extra management in kernel computations.

C. Memory Limitation

Usually, large datasets cannot fit into the limited on-device memory of a single GPU. Two possible ways to solve this problem include scaling up to multiple GPUs and applying a CPU/GPU heterogeneous design. CuMF [15] selects the former to enlarge the whole memory size; however, it depends on the availability of hardware. In contrast, this work designs and implements a CPU/GPU concurrent execution model to leverage the large host memory efficiently **with relaxed hardware requirements**.

III. Algorithm Analysis

Before discussing the parallel implementation, this section explains the original eALS and its sequential implementation. It mainly explains eALS' theoretical definitions and analyzes

TABLE I
SYMBOL DEFINITIONS

Name	Definitions
R	Sparse rating matrix
M	Number of users
N	Number of items
U	Dense user matrix
V	Dense item matrix
W	Sparse weight matrix
K	Number of factors
T	Predicted training matrix
L	Loss function
λ	Parameter to control the regularization

its computation patterns. Tab. I illustrates the definitions of symbols used in the following sections.

A. Fast eALS Algorithm

Fast eALS algorithm aims to solve matrix factorization. Matrix factorization decomposes one matrix (usually a sparse matrix, e.g., a user-item rating matrix) into the product of two lower dimensional matrices (e.g., a user matrix and an item matrix). According to the Tab. I, $R \in R^{M\times N}$, $U \in R^{M\times K}$, $V \in R^{N\times K}$. The matrix factorization can be defined as:

$$R = U \times V^T \tag{1}$$

A loss function L calculates the loss value between the predicted training matrix (T) and the ground truth matrix (R). Let r, u, and v represent elements of matrices R, U, and V. The loss function is calculated as:

$$L = \sum_{i=1}^{M}\sum_{j=1}^{N} w_{ij}(r_{ij} - \hat{r_{ij}})^2 + \lambda(\sum_{i=1}^{M}||u_i||^2 + \sum_{j=1}^{N}||v_j||^2) \tag{2}$$

To optimize the ALS algorithm, eALS (**e**lement-wise ALS) [9] introduces two new attributes, the element-wise learner and popularity-aware strategy. With this optimization, eALS separates the rated and unrated elements in matrix R, then assigns unrated elements a confidence value c. Here is the optimized loss function L:

$$\begin{aligned} L = &\sum_{i,j\in R} w_{ij}(r_{ij} - \hat{r_{ij}})^2 + \sum_{i=1}^{M}\sum_{j\notin R_i} c_j \hat{r_{ij}})^2 \\ &+ \lambda(\sum_{i=1}^{M}||u_i||^2 + \sum_{j=1}^{N}||v_j||^2) \end{aligned} \tag{3}$$

The confidence c comes from the popularity fi that denotes the popularity of item i. The definition of c is:

$$c_i = c_0 \frac{f_i}{\sum_{j=1}^{N} f_j} \tag{4}$$

To minimize this loss function, we need to get the deviation and compute the updating rules of matrices U and V. The updating formulas of u_{if} and v_{jf} are defined as:

$$u_{if} = \frac{\sum_{j\in R}[w_{ij}r_{ij} - (w_{ij} - c_j)\hat{r}_{ij}^f]v_{jf} - \sum_{k\neq f} p_{ik} \times sv_{kf}}{\sum_{j\in R}(w_{ij} - c_i)v_{jf}^2 + sv_{ff} + \lambda} \tag{5}$$

$$v_{jf} = \frac{\sum_{i\in R}[w_{ij}r_{ij} - (w_{ij} - c_j)\hat{r}_{ij}^f]u_{if} - c_j\sum_{k\neq f} p_{ik} \times su_{kf}}{\sum_{j\in R}(w_{ij} - c_i)u_{if}^2 + c_j \times su_{ff} + \lambda} \tag{6}$$

Algorithm 1: Fast eALS Algorithm [9]

Input: R, W, c, K, λ
Output: Matrices U, V

```
Initialize U and V randomly;
for (i, j) ∈ R do
    r̂_ij = Eq.(1)
while Stopping criteria is not met do
    // Update user factors;
    S^v = Σ_{i=1^N} c_i v_i v_i^T;
    for i = 1; i ≤ M; i++ do
        for f = 1; f ≤ K; f++ do
            for j ∈ R_i do
                r̂^f_ij = r̂_ij − u_if v_jf
            u_if = Eq.(5);
            for j ∈ R_i do
                r̂^f_ij = r̂_ij + u_if v_jf
    // Update item factors;
    S^u = U^T × U;
    for j = 1; j ≤ M; j++ do
        for f = 1; f ≤ K; f++ do
            for i ∈ R_j do
                r̂^f_ij = r̂_ij − u_if v_jf
            v_jf = Eq.(6);
            for i ∈ R_j do
                r̂^f_ij = r̂_ij + u_if v_jf
return U and V;
```

As shown in Alg. 1, kernel computation is to update dense matrices U and V in formula (5) and (6). Take computing U as an example, after a matrix multiplication to compute S^v, the kernel consists of three *for* loops. The first loop is to compute every row of U, which is able to execute concurrently. In the middle *for* loop, every column is accessed in U, and the dependence exists in this loop. To be more specific, the sparse matrix R is updated before and after updating U. Then the updated R should be used in the next iteration to update the next element in one row of U. The data dependence limits the efficiency of the parallel fast eALS algorithm implementation.

Tab. II shows detailed comparisons between the original ALS implementation in CuMF_ALS [15] and the fast eALS algorithm. ALS consists of two main kernels, matrix multiplication (`get_hermitian_x`) and matrix inverse (`batch_solve`). No data dependence exists as partitioning the input matrices into tiles for both kernels. ALS's matrix inverse (`batch_solve`) consists of two solving functions, LU and Conjugate Gradient (CG). LU's complexity is K-time higher than eALS. CG is an estimated method and its iteration number is set to be 6 in the programs, so the time complexity of the CG solver is the same as the initial prediction in the fast eALS algorithm. If $(M + N)K$ is much larger than $|R|$,

TABLE II
COMPARE CUMF_ALS AND FAST_EALS: TIME COMPLEXITY AND COMPUTATION PATTERNS

Algorithm	Function	Time Complexity	Computation Pattern
CuMF_ALS	`get_hermitian_x`	$\|R\|K^2$	Matrix multiplication, no dependence
CuMF_ALS	`batch_solve`	$(M+N)K^3/(M+N)K^2 * It\_CG$	Matrix inverse and multiplication, no dependence
fast_eALS	`updateUser/updateItem`	$\|R\|K$	Irregular access computation, have dependence
fast_eALS	`initialPredictions`	$(M+N)K^2$	Dense matrix multiplication, no dependence

$(M+N)K^2$ is dominant in time complexity. In summary, the original ALS has obvious higher time complexity than fast eALS, while the optimized CG solution of ALS in CuMF has the same time complexity as fast eALS. Besides time complexity, convergence speed is another critical factor to evaluate recommendation systems. In He *et al.*'s work [9], they conclude that the fast eALS algorithm has better recommendation quality than the ALS algorithm, which are compared in Section VIII.

B. Computation Pattern Analysis

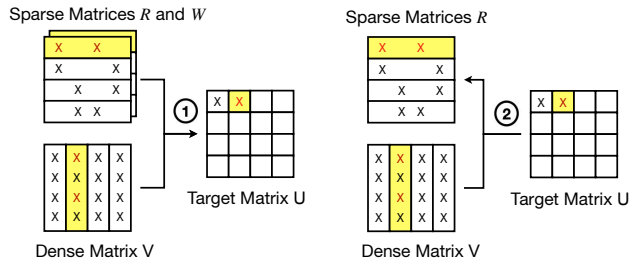

Fig. 2. **Kernel Computation Pattern.** Kernel computation mainly involves two stages: one is to utilize R, W, and V to update the target matrix U (①); the other is to calculate R based on two dense matrices U and V (②). Highlighted elements participate in calculating one element of U. Updating the second element (red cross one) in the first row of U requires the first row of R, but this row of R is necessarily updated after computing the first element (black cross one) of U. **Dependence exists in the same row of the matrix** U, i.e., one element of U depends on the updated values of all previous elements in the same row.

Compared with the original ALS, the fast eALS has a more complicated computation kernel with data dependency. Fig. 2 shows the access patterns of the fast eALS with an example highlighted with a yellow background. Target matrix and dense matrix are two dense matrices, defined as U and V. The objective is to use R to estimate U and V. In the left part of Fig. 2, the goal is to update the red cross element (in row 1 and column 2) of the target matrix U. The first row of two sparse matrices (R and W) and the second column of the dense matrix (V) will be accessed to compute the selected target element (①). Next, the updated red element of the target matrix (U) and the second column of the dense matrix (V) are used to update the first row of the sparse matrix (R) (②), as shown in the right part of Fig. 2.

Updating each row of U raises data dependence among columns, while no dependence occurs between rows. Take updating one row of U into consideration. Before computing the second element (i.e., the red cross element in the first row of U), the first row of R is necessarily updated. However, this row of R is computed after updating the first element (i.e., the black cross one in the first row of U). More specifically, computing one element of U requires the associated row of R, and this row of R is calculated based on the updated previous elements of U. Computing one element of U has to wait until all previous elements accomplish updating. Thus, data dependence exists in updating the same row of U. Different rows of U can execute concurrently without any dependence.

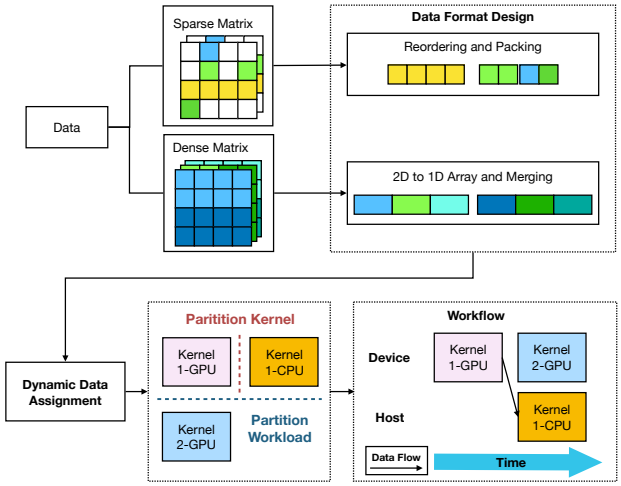

Fig. 3. **System Overview: architecture-adaptive data format and hybrid CPU/GPU collaboration model.** Sparse matrices are reordered and packed into groups. Dense matrices are reorganized into a merged one-dimensional array. The optimized data are fed into the hybrid CPU/GPU collaboration model with a dynamic data assignment. Kernel computation includes two types: (a) **partitioning kernel into Kernel 1-GPU (partial computation kernel w/o dependence on GPU) and Kernel 1-CPU (dependence part on CPU)**; (b) **Kernel 2-GPU (complete kernel on different data)**.

IV. SYSTEM OVERVIEW

Based on the background and algorithm analysis, this work presents an efficient parallel eALS recommendation framework on CPU/GPU heterogeneous systems (called HEALS). Fig. 3 illustrates HEALS's overview that consists of two main parts: architecture-adaptive data format and hybrid CPU/GPU collaboration model.

For input data, sparse matrices and dense matrices have different optimizations. On the one hand, HEALS reorders sparse matrices and pack them into groups for better workload balance. The new data formats are designed based on well-known CSR [17], [18]. Moreover, new data formats are designed to be adaptive for specific architectures, based on hardware characteristics. On the other hand, in order to improve data locality, HEALS linearizes dense matrices from 2D to 1D array and merges multiple matrices to one array.

After pre-processing input data, the data will be assigned to the hybrid CPU/GPU collaboration model. HEALS **partitions**

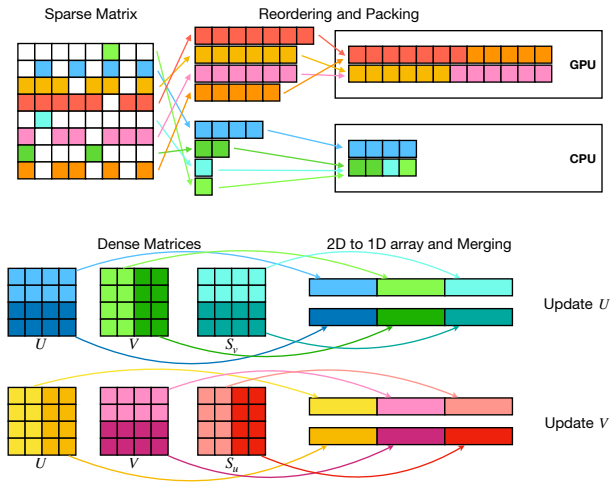

Fig. 4. **Data Format Design: sparse matrices and dense matrices.** HEALS designs new architecture-adaptive data format to solve workload unbalance on GPU and CPU. Sparse matrices on GPU are reorganized with balanced group size, while on CPU adopt uncertain size groups but more even computation cost. For dense matrices, HEALS translates multiple matrices into a combined one-dimensional array for better data locality.

kernel computation into Kernel 1-GPU (partial computation kernel without dependence on GPU) and Kernel 1-CPU (partial computation kernel with dependence on CPU). To further leverage concurrency, HEALS performs Kernel 2-GPU to hide the waiting latency of the device. **Kernel 2-GPU processes different data from Kernel 1-GPU/CPU and accomplishes the complete kernel computation**. Additionally, HEALS dynamically assigns data chunks with optimized proportion to achieve better workload balance.

V. Architecture-Adaptive Data Format

In order to reduce workload unbalance and improve data locality, HEALS presents architecture-adaptive data format for sparse matrices and dense matrices. Targeted workload unbalance, prior works propose various data format optimizations. For instance, Hong *et al.*'s paper [18] reorders the data and packs different rows. The data formats have two extra index arrays and a complicated data structure, which is suitable for simple kernel computation like matrix multiplication instead of matrix factorization. With respect to the kernel computation patterns, this work designs architecture-adaptive data format, including optimizing sparse matrices for GPU and CPU respectively, and dense matrix transformation.

A. Sparse Matrix

The sparse matrix optimization is shown in the upper part of Fig. 4. Before compressing the sparse matrix, HEALS reorders both rows and columns based on the number of rated elements. Then HEALS divides the whole sparse matrix into two parts: a **dense part** and a **sparse part**. Since GPU is beneficial for the computation-intensive kernel, the dense part is fed to GPU, while the sparse part to CPU.

As claimed in Section II, workload unbalance relies on the varied row lengths of sparse matrices. To improve workload balance, the key idea is to divide the rows into groups and balance the number of elements in each group for each thread. HEALS utilizes the *greedy algorithm* to distribute multiple rows into groups, and generate even groups. This new data format is named *Multiple Packed Compressed Sparse Row* (*MP-CSR*). The detailed steps are:

- **Step 1:** Reorder the matrix based on the row size.
- **Step 2:** Select the smallest row, and assign it to the current smallest group.
- **Step 3:** Continue selecting and assigning until all the rows are allocated. Each group contains an uncertain number of rows.

Since the data size of every group may be distinct, *MP-CSR* manages **an extra array** to record the first index position of every group. However, an extra array requires more memory space, resulting in extra access operations and more cache misses, especially for GPU. Thus, HEALS proposes another grouping approach, in which it assigns each group the same number of rows and balances the total number of elements in these rows. After reordering the rows, HEALS packs the longest one and shortest one together as a group. This new data format is named *Nested Packed Compressed Sparse Row* (*NP-CSR*). *NP-CSR* implies dividing multiple rows into even groups. The number of rows in the dense part is set to be even multiple numbers of the group. The number of groups depends on the input data size and GPU layouts. The detailed steps are:

- **Step 1:** Reorder the matrix based on the row size.
- **Step 2:** Select the largest row and smallest row, then assign them to one group.
- **Step 3:** Continue selecting and assigning until all the rows are allocated. Each group contains the same number of rows.

Compared with *NP-CSR*, *MP-CSR* has a better workload balance but a more complicated data structure. In contrast to GPU, CPU is more suitable to conduct an extra array with a larger cache and memory size. Therefore, **GPU employs *NP-CSR*, while CPU employs *MP-CSR***.

B. Dense Matrix

For dense matrices, HEALS linearizes a two-dimensional matrix into a one-dimensional array then merges dense matrices. When accessing data, the increased possibility of accessing near elements in different threads improves data locality. Packing three matrices together is also able to reduce data movements. For small datasets, dense matrices are easy to reorganize. For large dense matrices, HEALS partitions data to fit the limited global memory of GPU. As shown in Fig. 4, different color areas of dense matrices represent different partitions. Based on the access pattern in Fig. 2, light area in three matrices are loaded to deal with one partition of the target matrix. The partition organization depends on the input data size.

VI. Hybrid CPU/GPU Collaboration Model

This section introduces hybrid CPU/GPU collaboration model in HEALS. Some related works implement CPU/GPU

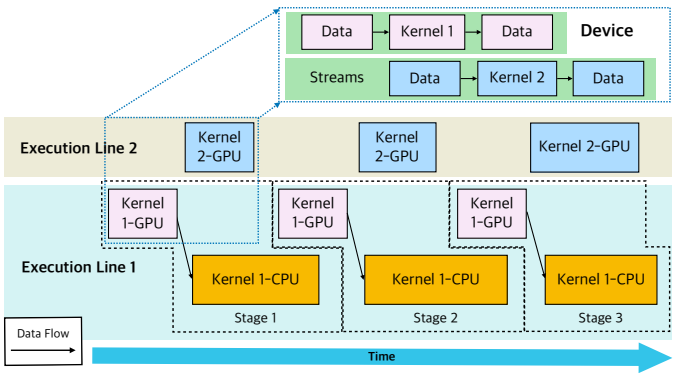

Fig. 5. **Framework Workflow: hybrid CPU/GPU collaboration model.** Multi-level concurrency includes: (a) two execution lines constitute **data parallelism**; (b) Kernel 1-GPU and Kernel 1-CPU achieve **task parallelism**; (c) **Overlapped data transfer** hides latency between Kernel 1-GPU and Kernel 2-GPU. Additionally, computation is partitioned into several stages to handle very large datasets.

concurrent schemes to solve matrix factorization. Tsai *et al.*'s work [19] separates kernel computation to solve QR matrix factorization with a much longer execution time on CPU than GPU. Teodoro *et al.*'s work [20] designs a performance-aware method for multi-GPUs with extra scheduling overhead. In contrast to prior approaches, HEALS exploits a hybrid CPU/GPU collaboration model, combining *Workload Partition (WP)* and *CPU-GPU Pipeline (CGP)* collaboration models [21]. HEALS dynamically adjusts data partitioning proportion for balance workload and optimal overlapping.

A. Multi-level Concurrency Design

Fig. 5 illustrates the design of hybrid CPU/GPU collaboration model. To process large datasets, HEALS partitions the whole data into chunks. The scheduling framework solves chunks stage by stage. In each stage, kernel computation is divided into two execution lines: one executes on Kernel 1-GPU and Kernel 1-CPU, and another runs on Kernel 2-GPU. The two execution lines process independent data.

Kernel 1 represents the first execution line, including two kernels: kernel without data dependence and the rest computation with dependence. The no dependence computation is named Kernel 1-GPU, executing on GPU. The rest of kernel computation is Kernel 1-CPU on CPU. They are combined together to solve a complete kernel for the same data chunks. Kernel 1-CPU can only start after Kernel 1-GPU.

However, Kernel 1-GPU is more than five times faster than Kernel 1-CPU to process identical data chunks. The significant GPU waiting time affects the performance. To improve GPU utilization, HEALS applies another execution line, aiming to concurrently execute with Kernel 1-CPU. To avoid any conflicts, this execution line deals with separate data chunks on GPU. This kernel is named Kernel 2-GPU to implement the whole kernel computation. Only executing Kernel 2-GPU is not efficient enough due to the data dependence. Therefore, the optimal design is to incorporate these two execution lines, to fully utilize both GPU and CPU. If the dataset is large, the computation will be partitioned into several stages. Due to the sequential execution of kernels on the device, multiple stages constitute a pipeline model.

Based on Sun*et al.*'s work [21], HEALS combines two CPU/GPU collaboration models: **Workload Partition (WP)** and **CPU-GPU Pipeline (CGP)**. On one hand, Kernel 2-GPU and Kernel 1-CPU solve independent workload concurrently as a WP collaboration model. On other hand, the first execution line partitions kernel computation in multiple stages, formalized as a CGP collaboration model. Furthermore, HEALS achieves **multi-level concurrency**. As shown in Fig. 5, the concurrency consists of three major aspects: *task parallelism* between Kernel 1-CPU and Kernel 1-GPU, *data parallelism* between two execution lines (i.e., Kernel 1 and Kernel 2), and *overlapped data transferring* between two GPU kernels.

B. Adjusting Data Partition Dynamically

This work designs a dynamic partition model to calculate the ideal proportion and adjusts it in the following iterations. According to Fig. 5, HEALS partitions the data into two independent execution lines, so it is critical to balance the workload between the two execution lines. When training the complete parallel fast eALS model, the iteration number is at least 50 to achieve a satisfying accuracy. After executing on the first iteration, HEALS obtains execution time and builds bivariate linear equations to compute the ideal partition ratio. Subsequently, the adjusted proportion is applicable in the following iterations.

Take one stage as an example to explain how to compute the ideal partition ratio. This work defines the execution time of Kernel 1-GPU to be t_1, Kernel 2-GPU to be t_2, and Kernel 1-CPU to be t_3. First, HEALS allocates θ tiles to solve. Tiles mean the groups of rows of the sparse matrices. Second, α tiles are allocated to Kernel 2-GPU and β to Kernel 1. The most overlapped case is $t_1 + t_2 = t_3$, between Kernel 1 and Kernel 2. Last, the equations are built as follows: $\theta = \alpha + \beta$, $\alpha \times t_2 + \beta \times t_1 = \beta \times t_3$.

The partition ratio is set to be $\gamma = \frac{\alpha}{\beta}$. Execution time t_1, t_2, t_3, and the total number of tiles θ are given. The objective is to calculate γ, β and α. After solving the bivariate linear equations, results are shown as follows:

$$\gamma = (t_3 - t_1)/t_2 \tag{7}$$

$$\beta = \frac{t_2\theta}{t_3 - t_1 + t_2} \tag{8}$$

$$\alpha = \frac{(t_3 - t_1)\theta}{t_3 - t_1 + t_2} \tag{9}$$

If the data size is so large to be managed more than one stage, all stages will compute ratios individually after the first iteration and record them in a global array.

VII. Hardware-Based Accelerating Techniques

This section presents hardware-based accelerating techniques used in HEALS, including *loop transformation* and *GPU parallel reduction*. Fig. 6 illustrates the implementing details in Kernel 1-GPU, Kernel 2-GPU and Kernel 1-CPU.

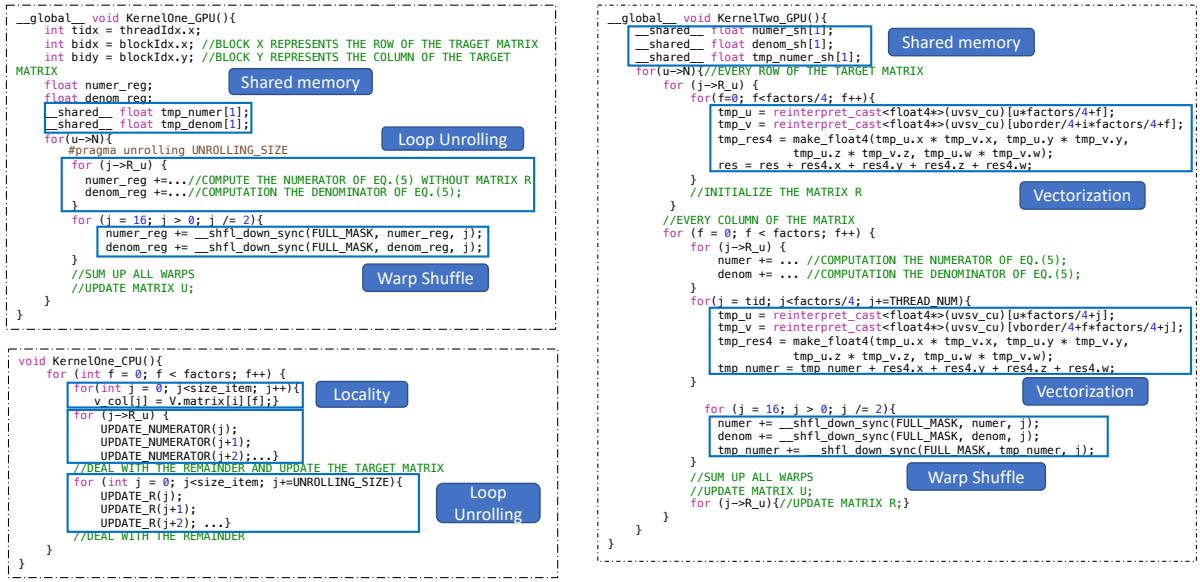

Fig. 6. **Implementation in Kernel 1-GPU, Kernel 1-CPU and Kernel 2-GPU: applying hardware-based accelerating techniques.** Loop transformation includes **loop unrolling** for a better locality and **vectorizd I/O** to leverage data parallelism. GPU parallel reduction is accelerated by **warp primitives (warp shuffle)**, which efficiently utilizes registers and shared memory for reduction.

A. Loop Transformation

HEALS applies loop transformation including vectorization and loop unrolling. Vectorization is commonly used to leverage data parallelism, as one instruction manages multiple data (SIMD). On CPU, the compiler automatically applies SIMD optimizations in most loops when setting `-O3`. On GPU, vectorized I/O cannot be managed by `nvcc`, providing chances to improve the kernel performance. CUDA supports vectorization instructions, including 64 or 128-bit load and store operations. Vectorization is used to update R, a data-dependent part. Thus HEALS conducts vectorization in Kernel 2-GPU. Likewise, loop unrolling is applied in Kernel 1-CPU and Kernel 1-GPU to further improve data locality.

During loop transformation, HEALS leverages shared memory to store temporal data for each block. Based on the access pattern in Fig. 6, when updating one row of the target matrix, the whole dense matrix has to be accessed, which cannot fit in the limited shared memory. As for the sparse matrix, each thread processes one row, of which the size varies dramatically. Thus, HEALS mainly stores temporal parameter values instead of dense or sparse matrices in the shared memory, both in Kernel 1-GPU and Kernel 2-GPU. These parameters are extended to arrays, making each thread process one element without any conflicts.

B. Accelerating GPU Parallel Reduction

GPU parallel reduction techniques aim to summarize data among threads efficiently. Based on the eALS algorithm implementation in Section III, summation operation is part of the kernel computation to calculate dense matrices. Therefore, HEALS performs efficient utilization of registers and shared memory to improve GPU parallel reduction.

GPU parallel reduction in HEALS is broadcast through *warp-level (registers)*, *block-level (shared memory)*, and *global-level (global memory)*. To accomplish summation operations in Kernel 1-GPU and Kernel 2-GPU, HEALS takes advantage of registers based on warp-level primitives, `shfl_down_sync`. Warp shuffle directly fetches data from other threads' registers within one warp, which benefits broadcasting data without `__syncthreads()`. The detailed implementing steps are: *First*, HEALS utilizes primitive function `shfl_down_sync` to get the sum of one warp, and the first thread of every warp obtains the result; *Second*, HEALS employs `atomicAdd` operation to sum up all warps and store the result in shared memory; *Last*, HEALS gathers the values in the shared memory and stores the final result in the global memory. Overall, efficient GPU parallel reduction yields great benefits on performance improvements, which will be explained in Section VIII.

VIII. Evaluation

This section evaluates HEALS with a set of experiments. The evaluation objectives include: (a) demonstrating that HEALS outperforms other state-of-the-art related works; (b) evaluating the effects of proposed optimizations including architecture-adaptive data format, hybrid CPU/GPU collaboration model, and hardware-based accelerating techniques; (c) validating the benefits of HEALS on prediction accuracy and recommendation quality.

A. Experiment Settings

1) Environment: Experiments are conducted on a heterogeneous CPU/GPU platform with an Intel Xeon CPU with 40-cores, and an NVIDIA Tesla P100 GPU (as shown in Tab. III). This platform is configured with Ubuntu 18.04.4, icpc (ICC) 19.1.0.166, and CUDA V10.1.105. All HEALS runs use all computing resources (i.e., 40 threads w/o hyper-threading and the whole GPU).

TABLE III
MACHINE INFORMATION

Hardware	Descriptions
CPU	Intel(R) Xeon(R) Gold 6138 CPU @ 2.00GHz
GPU	NVIDIA Corporation GP100GL Tesla P100 PCIe 16GB

TABLE IV
EXPERIMENT DATASETS

Name	Users	Items	Ratings	Sparsity
Yelp	25677	25815	698472	99.8947
Amazon	117176	75389	3790245	99.9571
YahooMusic	1127446	136736	431596064	99.7211
Friendster	37551359	124836179	1768515776	99.9999

2) Datasets: This work is tested on four datasets: Yelp, Amazon, YahooMusic, and Friendster. Yelp and Amazon datasets come from the original eALS paper [9]. Dataset YahooMusic [22] includes the ratings of songs with artists and albums from the Verizon Media Labs. Friendster is a sparse matrix dataset from the Stanford Large Network Dataset Collection. Friendster contains 1.7 billion ratings to show HEALS's capacity of handling large data. It requires more than 56 GB memory for kernel computation, out of GPU's on-device memory. This original dataset is pre-processed to match our recommendation evaluation needs, including adding random rated values from one to five and eliminating all repeated edges and null elements. The number of users, items, and training ratings are shown in Tab. IV. This work randomly selects 90% data to train and 10% to test. Training datasets are used for measuring performance while testing datasets are used for measuring accuracy and recommendation quality.

3) State-of-the-art Works to Compare: HEALS is compared with three state-of-the-art implementations: **Original Java implementation** [9] is implemented in a sequential way. **LIBMF** [7] is a state-of-the-art matrix factorization library on CPU. This work compares with its ALS implementation (with all 40 CPU threads). **CuMF_ALS** [5], [8], [15] is a state-of-the-art library to solve matrix factorization on (multiple) GPUs. Each test runs 10 times. Because different runs do not vary significantly, this work only reports the average time for readability.

B. Overall Improvement

The overall performance improvement is evaluated in terms of the execution time per iteration (as shown in Tab. V). CuMF library is evaluated with two different batch solver functions: CG and LU. Factor values (f) are commonly set to be 60 or 100 as suggested in Xie *et al.*'s work [5], so our experiments cover both factors. Tab. V shows the minimum and maximum speedup in the last column. `OOM` donates that CuMF cannot execute Friendster on a single GPU because of the limited on-device memory. Our evaluation results show that HEALS achieves significant speedups on all datasets, outperforming all other state-of-the-art works by a speedup from $1.48\times$ to $23.6\times$. Particularly, comparing with the state-of-the-art CPU implementation (LIBMF with all 40 CPU threads), HEALS achieves $15.65\times$, $14.43\times$, $8.47\times$, and $8.44\times$ speedup on Yelp, Amazon, YahooMusic, and Friendster, respectively. Comparing with the fastest version of the latest GPU implementation (CuMF), HEALS achieves $3.42\times$, $2.39\times$, and $1.48\times$ speedup on Yelp, Amazon, and YahooMusic, respectively.

C. Performance Analysis: Optimization Breakdown

Fig. 7 to Fig. 10 illustrate the impact of different optimizations on performance improvements. The speedup is compared with the original eALS implementation in Fig. 7 and Fig. 8. The partition ratio in Fig. 8 represents the workload ratio of Kernel 2 over Kernel 1. The baseline of Fig. 9 and Fig. 10 is the non-optimized kernel computation. The proposed hardware-based accelerating techniques are divided into `Vectorized I/O`, and `Warp shuffle + Vectorized I/O`.

Fig. 7 shows the impact of architecture-adaptive data format. Sparse matrix data format optimization brings $1.1\times$ benefits on average while dense matrix optimization brings additional $1.2\times$ gains, e.g., the speedup of Amazon improves from $13.01\times$ with sparse matrix optimization only to $16.98\times$ with both sparse and dense data format optimizations. Experiments demonstrate that dense matrix optimization yields slightly more benefits than the sparse matrix. Fig. 8 shows the benefits of dynamic partition ratio, achieving a $1.25\times$ to $2.54\times$ speedup over other selected fixed partition ratios. The improved results demonstrate great benefits of the hybrid CPU/GPU collaboration model. Fig. 9 and Fig. 10 show the speedup of hardware-based accelerating techniques compared with non-optimized GPU kernel implementation. Warp shuffle yields more benefits than loop transformation, e.g., the GPU vectorized I/O brings $1.09\times$ performance gains on average while warp shuffle brings additional $1.23\times$ benefits on average (in Fig. 9).

D. Recommendation Efficiency

This work evaluates recommendation efficiency in two major aspects: **matrix factorization precision** and **recommendation equality**. The performance metrics are Root Mean Square Error (**RMSE, the lower the better**) and Normalized Discounted Cumulative Gain (**NDCG, the higher the better**) [9]. RMSE represents the average accuracy of rated scores, illustrating matrix factorization precision. NDCG aims to measure the recommending quality without considering the values of rated scores. The observations involve not only *convergence point values* but *convergence speed* as well to measure the recommendation efficiency.

Fig. 11 shows the evaluation results. HEALS is measured with $f = 64$. HEALS is compared with CG solver-based CuMF with two factors ($f = 60$, $f = 100$) in three datasets. For Friendster, HEALS is compared with LIBMF ($f = 64$).

For RMSE, HEALS and CuMF achieve close factorization precision but HEALS obtains faster convergence speed. As shown in Fig. 11, HEALS achieves $1.45\times$ convergence speedup over CuMF ($f = 60$) and $1.77\times$ over CuMF ($f = 100$) on average, respectively. On Friendster, HEALS outperforms LIBMF, achieving $3.75\times$ convergence speedup. Experiment results prove HEALS has better recommendation efficiency than other ALS-based parallel libraries.

TABLE V

OVERALL PERFORMANCE: EXECUTION TIME (S) PER ITERATION. HEALS is compared with original java implementation of eALS, LIBMF, and CuMF with CG and LU solvers. CuMF is measured with two factors: $f = 100$ and $f = 60$. 'Speedup' illustrates the minimum and maximum speedup of HEALS over all others. 'OOM' donates that CuMF cannot execute Friendster on a single GPU due to the limited on-device memory.

Datasets	Original eALS	LIBMF	CuMF(CG,f=100)	CuMF(CG,f=60)	CuMF(LU,f=100)	CuMF(LU,f=60)	HEALS (f=64)	Speedup
Yelp	0.284	0.18776	0.06942	0.04104	0.2902	0.1456	**0.0120**	**3.42X-23.6X**
Amazon	1.223	1.71973	0.4186	0.2852	0.8176	0.4102	**0.1192**	**2.39X-14.4X**
YahooMusic	89.811	70.905	19.55	12.4054	22.7412	13.3947	**8.3740**	**1.48X-10.7X**
Friendster	940.8	663.7	OOM	OOM	OOM	OOM	**78.6**	**8.4X-11.97X**

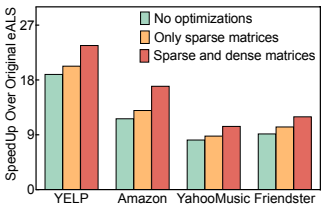

Fig. 7. **Speedup: impact of architecture-adaptive data format.**

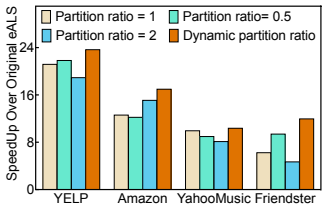

Fig. 8. **Speedup: impact of partition ratio in hybrid CPU/GPU collaboration model**

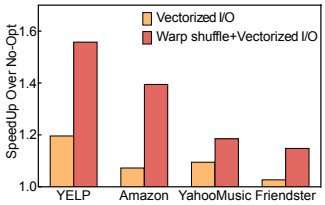

Fig. 9. **Speedup in Kernel 1-GPU: impact of hardware-based accelerating techniques.**

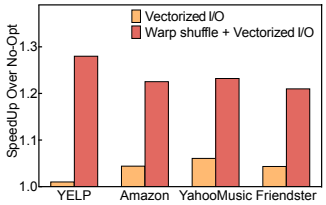

Fig. 10. **Speedup in Kernel 2-GPU: impact of hardware-based accelerating techniques.**

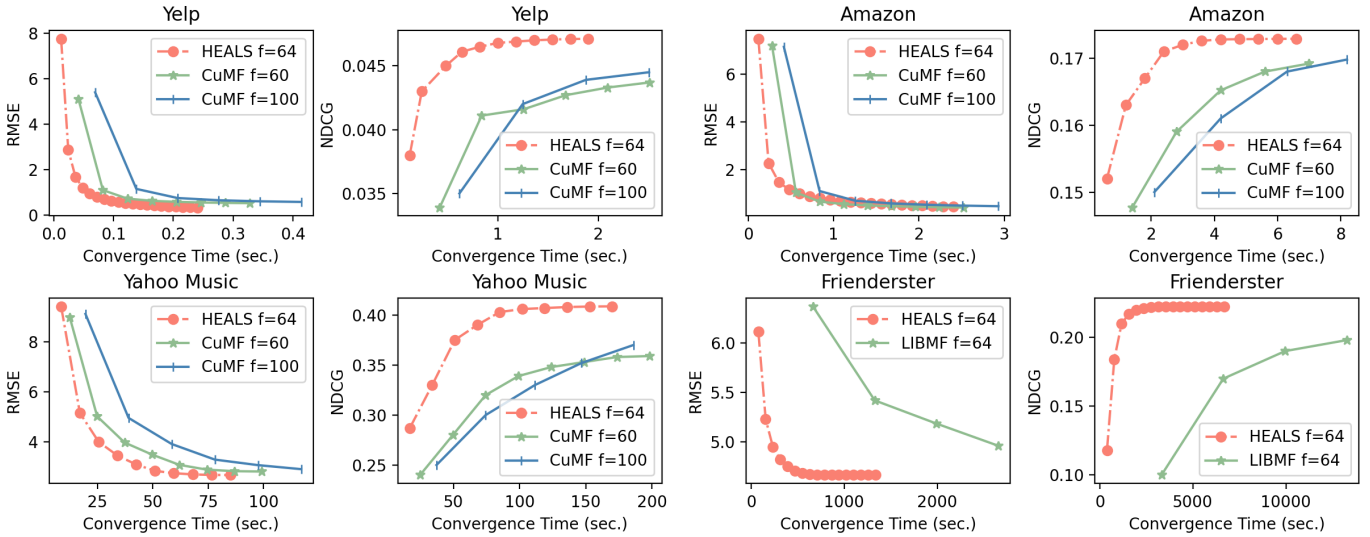

Fig. 11. **Recommendation Efficiency: Convergence Speed of RMSE and NDCG.** Compare HEALS ($f = 64$) with CG Solver based CuMF ($f = 60$, $f = 100$) in datasets: Yelp, Amazon, and YahooMusic. Compare HEALS ($f = 64$) with LIBMF ($f = 64$) in Friendster.

For NDCG, HEALS yields benefits on both recommendation equality and convergence speed. For instance, HEALS obtains a 1.14× convergence point value over CuMF on YahooMusic. HEALS achieves a 2.78× convergence speedup over CuMF on Amazon. For Friendster, HEALS achieves a 1.12× NDCG absolute value and a 4.83× convergence speedup over LIBMF. In summary, HEALS shows great benefits on recommendation efficiency, outperforming others.

IX. RELATED WORK

Matrix factorization-based recommendation algorithms. Various matrix factorization-based algorithms have been applied in recommendation systems, e.g., Stochastic Gradient Descent (SGD) [1], [23], Cyclic Coordinate Descent (CCD) [6], [20], and ALS [7], [9], [15], [24]. He *et al.*'s eALS work [9] has proved that the optimized eALS-based recommendation model outperforms other matrix factorization-based approaches (e.g., SGD, CCD, and original ALS) in both computation cost and recommendation accuracy. That is why this work focuses on building *the first* efficient parallel recommendation system based on eALS.

Parallel ALS-based recommendation systems. Many existing research efforts focus on parallelizing ALS-based recommendation. LIBMF [7] offers an efficient ALS C++ library on CPUs. Chen *et al.* [25] also propose an optimized ALS algorithm based on Weighted-Regularization (WR) on CPU. Tan *et al.* [8], [15] employ iterative conjugate gradient solver to optimize parallel ALS, and present a parallel ALS implementation on GPU (CuMF). Closely related to HEALS, Teodoro *et al.* [20] propose a performance-aware CPU/GPU heterogeneous framework and optimize ALS by managing GPU layouts and partitioning. Chen *et al.* [26] implement a portable ALS framework to optimize original ALS by applying system techniques, e.g. thread batching techniques. Many other ALS-based parallel recommendation systems are

implemented on distributed systems. Xie *et al.* [27] design a new loss function of ALS algorithm on Spark platform. Aljunid and Manjaiah [28] optimize conventional ALS algorithm based on Apache Spark. Winlaw *et al.* [29] propose an optimized ALS by using a nonlinear conjugate gradient (NCG) on Spark. Compared with all of these efforts, HEALS targets a more advanced algorithm (eALS) that is more challenging to be parallelized than the original ALS, and *for the first time* presents a set of new system optimizations on heterogeneous CPU/GPU systems.

Other parallel recommendation systems based on matrix factorization. Many other matrix factorization-based approaches are implemented on multi-core or many-core architectures. CuMF_SGD [5] scales up the SGD kernel computation on multiple GPUs, and Nisa *et al.* [6] present an optimized implementation of CCD++ on GPU. Li *et al.* [24] mainly focus on non-negative matrix factorization and present a multi-GPU implementation. All these efforts explore GPU optimizations without fully utilizing CPU resources or CPU/GPU collaboration. Zinkevich *et al.* [23] mainly optimize SGD in data parallelism. In contrast, HEALS exploits hybrid CPU/GPU collaboration model and combines data parallelism and task parallelism together. Again, different from all of these efforts, HEALS targets a more advanced eALS algorithm with better computation cost and recommendation accuracy while more challenges to be parallelized.

X. Conclusion

This work presents HEALS, an efficient CPU/GPU parallel recommendation system, *for the first time* building on top of fast eALS. To alleviate workload unbalance, HEALS employs a new architecture-adaptive data format for both GPU and CPU. HEALS is also equipped with a new hybrid CPU/GPU collaboration model with an adjustable partition ratio. HEALS supports multi-level concurrency including data parallelism, task parallelism, and overlapped data transferring. Moreover, HEALS takes advantage of various hardware-based accelerating techniques to further optimize kernel computation, including vectorized I/O, loop unrolling, and efficient GPU parallel reduction. HEALS outperforms other baselines by a speedup of $1.48\times$ to $23.6\times$. HEALS also achieves significantly better recommendation accuracy and quality, and faster convergence than other state-of-the-art libraries. In the future, we plan to extend HEALS to execute on multiple GPUs.

Acknowledgment

The authors would like to thank the anonymous reviewers for making innumerable helpful suggestions and comments. This research is in part supported by National Science Foundation CCF-2047516 (CAREER) and Jeffress Trust Awards in Interdisciplinary Research to William & Mary, and an industry sponsorship research grant from iLambda to Kent State University. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, or Thomas F. and Kate Miller Jeffress Memorial Trust.

References

[1] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *COMPSTAT*. Springer, 2010.
[2] A. Liu, Q. Wu, Z. Liu, and L. Xia, "Near-neighbor methods in random preference completion," in *AAAI*, 2019.
[3] P. Nagarnaik and A. Thomas, "Survey on recommendation system methods," in *ICECS*. IEEE, 2015.
[4] Q. Wu, A. Hare, S. Wang, Y. Tu, Z. Liu, C. G. Brinton, and Y. Li, "Bats: A spectral biclustering approach to single document topic modeling and segmentation," *ACM TIST*, 2021.
[5] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "Cumf_sgd: Parallelized stochastic gradient descent for matrix factorization on gpus," in *HPDC*, 2017.
[6] I. Nisa, A. Sukumaran-Rajam, R. Kunchum, and P. Sadayappan, "Parallel ccd++ on gpu for matrix factorization," in *GPGPUs*, 2017.
[7] W.-S. Chin, B.-W. Yuan, M.-Y. Yang, Y. Zhuang, Y.-C. Juan, and C.-J. Lin, "Libmf: a library for parallel matrix factorization in shared-memory systems," *JMLR*, 2016.
[8] W. Tan, S. Chang, L. Fong, C. Li, Z. Wang, and L. Cao, "Matrix factorization on gpus with memory optimization and approximate computing," in *ICPP*, 2018.
[9] X. He, H. Zhang, M.-Y. Kan, and T.-S. Chua, "Fast matrix factorization for online recommendation with implicit feedback," in *SIGIR*, 2016.
[10] H. Li, K. Li, J. An, and K. Li, "Msgd: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on gpus," *TPDS*, 2017.
[11] B. Hidasi and D. Tikk, "Fast als-based tensor factorization for context-aware recommendation from implicit feedback," in *ECML PKDD*. Springer, 2012.
[12] I. Pilászy, D. Zibriczky, and D. Tikk, "Fast als-based matrix factorization for explicit and implicit feedback datasets," in *RecSys*, 2010.
[13] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *ICDM*. Ieee, 2008.
[14] P. Lu and M. Allam, "Hybrid collaborative filtering recommendation algorithm for als model based on a big data platform," in *IAEAC*, 2021.
[15] W. Tan, L. Cao, and L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on gpus," in *HPDC*, 2016.
[16] H. Liu, S. Pai, and A. Jog, "Why gpus are slow at executing nfas and how to make them faster," in *ASPLOS*, 2020.
[17] B. Wheatman and H. Xu, "Packed compressed sparse row: A dynamic graph representation," in *2018 HPEC*. IEEE, 2018.
[18] C. Hong, A. Sukumaran-Rajam, I. Nisa, K. Singh, and P. Sadayappan, "Adaptive sparse tiling for sparse matrix multiplication," in *PPoPP*, 2019.
[19] Y. M. Tsai, W. Wang, and R.-B. Chen, "Tuning block size for qr factorization on cpu-gpu hybrid systems," in *MCSoC*. IEEE, 2012.
[20] G. Teodoro, T. M. Kurc, T. Pan, L. A. Cooper, J. Kong, P. Widener, and J. H. Saltz, "Accelerating large scale image analyses on parallel, cpu-gpu equipped systems," in *IPDPS*. IEEE, 2012.
[21] Y. Sun, X. Gong, A. K. Ziabari, L. Yu, X. Li, S. Mukherjee, C. McCardwell, A. Villegas, and D. Kaeli, "Hetero-mark, a benchmark suite for cpu-gpu collaborative computing," in *IISWC*. IEEE, 2016.
[22] G. Dror, N. Koenigstein, Y. Koren, and M. Weimer, "The yahoo! music dataset and kdd-cup'11," in *KDD Cup 2011*. PMLR, 2012.
[23] M. Zinkevich, M. Weimer, L. Li, and A. J. Smola, "Parallelized stochastic gradient descent," in *NIPS*, 2010.
[24] H. Li, K. Li, J. Peng, and K. Li, "Cusnmf: A sparse non-negative matrix factorization approach for large-scale collaborative filtering recommender systems on multi-gpu," in *ISPA/IUCC*. IEEE, 2017.
[25] M. Chen, T. Chen, and Q. Chen, "An efficient implementation of the als-wr algorithm on x86 cpus," in *BMO*. Springer, 2019.
[26] J. Chen, J. Fang, W. Liu, T. Tang, and C. Yang, "clmf: A fine-grained and portable alternating least squares algorithm for parallel matrix factorization," *Future Generation Computer Systems*, 2018.
[27] L. Xie, W. Zhou, and Y. Li, "Application of improved recommendation system based on spark platform in big data analysis," *Cybernetics and Information Technologies*, 2016.
[28] M. F. Aljunid and D. Manjaiah, "An improved als recommendation model based on apache spark," in *ICSCS*. Springer, 2018.
[29] M. Winlaw, M. B. Hynes, A. Caterini, and H. De Sterck, "Algorithmic acceleration of parallel als for collaborative filtering: Speeding up distributed big data recommendation in spark," in *ICPADS*, 2015.