IEEE Access
Multidisciplinary : Rapid Review : Open Access Journal

Received November 18, 2019, accepted January 29, 2020, date of publication February 3, 2020, date of current version February 10, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2971036

High-Performance Homomorphic Matrix Completion on Multiple GPUs

TAO ZHANG[1], (Member, IEEE), HAN LU[1], AND XIAO-YANG LIU[2], (Student Member, IEEE)
[1]School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China
[2]Department of Electrical Engineering, Columbia University, New York, NY 10027, USA

Corresponding author: Xiao-Yang Liu (xl2427@columbia.edu)

The work of Tao Zhang was supported in part by the Natural Science Foundation of Shanghai under Grant 17ZR1409800.

ABSTRACT In various applications such as trajectory tracking in mobile social networks and online recommendation systems, the massive raw data are often incomplete due to various unpredictable or unavoidable reasons. Matrix completion algorithms are effective for reconstructing two-dimensional data, but sending raw data containing personal, sensitive information to cloud computing nodes for matrix completion may lead to privacy exposure issue. The homomorphic matrix completion is a promising approach to perform matrix completion while preserving privacy. However, CPU-based homomorphic matrix completion has low performance, making it impractical to process multiple or large-scale data completion tasks in real-time. In this paper, we propose a high-performance homomorphic matrix completion scheme by exploiting commodity GPUs (Graphics Processing Units) that are widely available in HPC servers and cloud computing nodes. First, we design and implement a baseline GPU-based homomorphic matrix completion, and propose techniques to optimize memory accesses, GPU utilization, and communications. Second, we propose a shard mode for large-scale matrices exceeding GPU memory capacity. Third, we propose a multi-GPU mode to fully utilize multiple GPUs in computing nodes. Experiment results show that the proposed scheme is both fast and accurate. On matrices of varying sizes, the proposed scheme running on a single Tesla V100 GPU achieves up to 116.23× speedups over the CPU MATLAB implementation running on dual Xeon CPUs. The multi-GPU mode achieves up to 1.84× speedups on two GPUs versus on a single GPU. For large-scale matrices, the shard mode achieves up to 174.92× speedups on a single GPU over the CPU MATLAB implementation on two CPUs, and further achieves up to 1.35× speedups when running on two GPUs using the multi-GPU mode.

INDEX TERMS GPU, homomorphic matrix completion, least squares minimization.

I. INTRODUCTION

In many applications such as video and image processing [2], out-door and in-door localization [3], [4], and recommendation systems [5], the massive data are often incomplete owing to various reasons in data acquisition and transmission [6]. For instance, the UC Berkeley INTEL project [7], [8] reported 40% data missings and 8% data errors. Many scientific researches such as geoexploration [9] depend on complete data to draw accurate conclusion, since data analysis on incomplete data leads to inaccurate and even wrong conclusions. Therefore, various data completion methods [8], [10], [11] have been designed to recover the incomplete data. For two-dimensional data (i.e., matrix), matrix completion approaches [8] [11] [12] [13] were proposed to recover the missing elements by exploiting the latent low-rank property of matrices. Alternating minimization [7] [11] [14] is an effective algorithm to address the matrix completion problem. Matrix completion approaches have been widely adopted in diverse applications including link prediction in social networks [15], image recovery [16], image classification [17], and action detection [18].

Existing matrix completion approaches have limitations in the cloud computing paradigm. Plain-data (i.e., not encrypted) based matrix completion on cloud computing nodes may have privacy issues. In cloud computing, users send requests with plain data from their mobile devices to cloud computing nodes, then cloud computing nodes execute the matrix completion algorithms, and finally responses are sent back from cloud computing nodes to users [4] [19].

The associate editor coordinating the review of this manuscript and approving it for publication was Tomás F. Pena.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

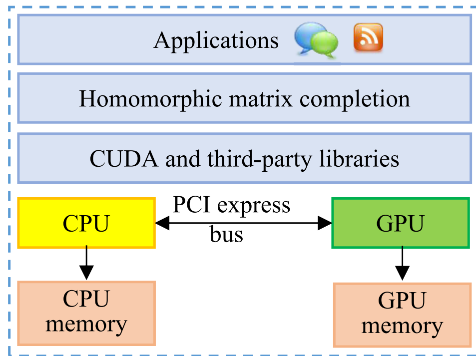

FIGURE 1. Overview of the homomorphic matrix completion on GPU.

In this process, users' sensitive information may be exposed to cloud computing nodes, such as locations and trajectories, preferences, and website navigation histories. Privacy exposure may lead to economic losses and even lawsuits. For example, the recommendation contest Netflix prize was canceled after the privacy lawsuit [20]. To address the privacy issue, the homomorphic matrix completion approach [19] was proposed to perform matrix completion while preserving privacy. Homomorphic matrix completion [19] adopts a three-step procedure and conduct matrix completion on encrypted data instead of plain data to preserve privacy. However, the major computation of the homomorphic matrix completion is tens of iterative least square minimizations, whose running time grows rapidly with the matrix size. As a result, it is impractical for a cloud computing node to process multiple or large-scale homomorphic matrix completion tasks in real-time.

In this paper, we propose high-performance and large-scale homomorphic matrix completion on GPU, as shown in Fig. 1. We design, implement, and optimize fast and accurate homomorphic matrix completion on single and multiple GPUs. First, we design and implement a baseline homomorphic matrix completion scheme on GPU. This baseline scheme is full functional, but its performance is unoptimized. We then investigate the bottlenecks of the baseline scheme, and optimize the memory accesses, GPU utilizations, and CPU-GPU communications to obtain an optimized scheme. Second, we design a multi-GPU mode to distribute the computations among multiple GPUs on a computing node. Third, we propose a shard mode to complete large-scale matrices exceeding the GPU memory capacity.

Our contributions are summarized as follows.

- We design, implement and optimize a GPU-based homomorphic matrix completion scheme to achieve high performance and accuracy. We propose techniques to optimize memory accesses, GPU utilizations, and CPU-GPU communications. These optimization techniques improve performance significantly.
- We propose a multi-GPU mode to fully utilize multiple GPUs in cloud computing nodes. This mode achieves up to 1.84× speedups on two GPUs versus on a single GPU. We further propose a shard mode to complete large-scale matrices exceeding the GPU memory capacity.
- We perform extensive experiments to evaluate the performance of the GPU-based homomorphic matrix completion. For small- or medium-sized matrices, the proposed scheme running on a Tesla V100 GPU achieves up to 116.23× speedup versus the CPU MATLAB implementation [19] running on dual Xeon CPUs. With the multi-GPU mode, the proposed scheme achieves up to 1.84× speedups on two GPUs versus on a single GPU. With the shard mode, the proposed scheme achieves up to 174.92× speedups on a Tesla V100 GPU versus the CPU MATLAB implementation [19] running on dual Xeon CPUs for large matrices, and further achieves up to 1.35× speedups when running on two GPUs using the multi-GPU mode. The proposed scheme achieves similar recovery error with the CPU MATLAB implementation [19]. The project is available at: https://github.com/hust512/Homomorphic_Matrix_Completion_Multiple_GPU.

The remainder of this paper is organized as follows. In Section II, we describe the notations and the homomorphic matrix completion algorithm [19]. Section III presents the design, implementation and optimization of the GPU-based homomorphic matrix completion scheme. Section IV presents the shard mode and the multi-GPU mode. In Section V, we evaluate the performance of the proposed scheme. Section VI discusses the related works. The conclusions are drawn in Section VII.

II. THE HOMOMORPHIC MATRIX COMPLETION ALGORITHM

We introduce the notations, summarize the homomorphic matrix completion algorithm [19], and provide parallel acceleration analysis for this algorithm.

A. NOTATIONS

We use lowercase and uppercase boldface letters to denote vectors (e.g., $\boldsymbol{x} \in \mathbb{R}^I$) and matrices (e.g., $\boldsymbol{X} \in \mathbb{R}^{I\times J}$), respectively. We use i, j to index the rows and columns of a matrix, respectively. We use $[I]$ to denote the set $\{1, 2, \ldots, I\}$. For a matrix $\boldsymbol{X} \in \mathbb{R}^{I\times J}$, $\boldsymbol{X}_j$ or $\boldsymbol{X}(:, j)$ denotes the j-th column of $\boldsymbol{X}$, the (i, j)-th element is $\boldsymbol{X}(i, j)$ or simply $\boldsymbol{X}_{ij}$, where $i \in [I]$ and $j \in [J]$. The transposed matrix of $\boldsymbol{X}$ is denoted as $\boldsymbol{X}^T$. We use $\boldsymbol{X}^i$ to denote the i-th matrix in a series of matrices.

1) FROBENIUS NORM OF A MATRIX

The Frobenius norm of a matrix $\boldsymbol{X}$ is $\|\boldsymbol{X}\|_F = \sqrt{\sum_{i=1}^{I}\sum_{j=1}^{J}|\boldsymbol{X}_{ij}|^2}$.

2) INDEX MATRIX

The index matrix $\boldsymbol{\Phi}$ indicates whether an element $\boldsymbol{X}_{ij} \in \boldsymbol{X}$ is missing. It is defined as follows:

$$\boldsymbol{\Phi}_{ij} = \begin{cases} 0, & \text{if } \boldsymbol{X}_{ij} \text{ is missing,} \\ 1, & \text{otherwise.} \end{cases}$$

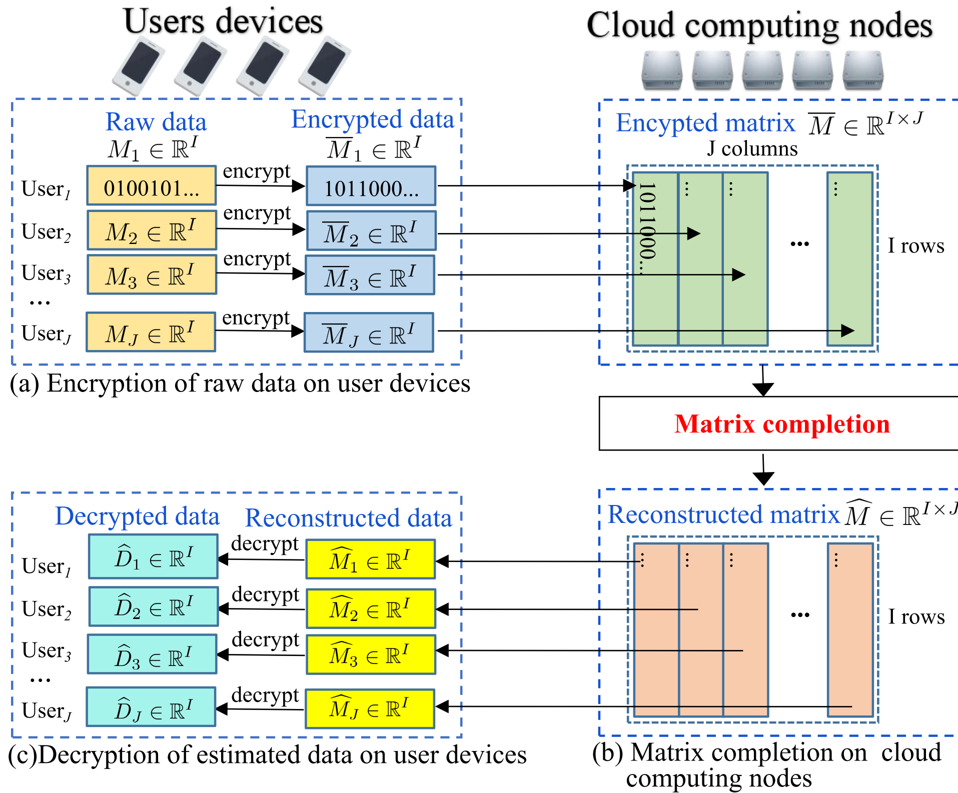

FIGURE 2. Overview of the homomorphic matrix completion algorithm [19].

Algorithm 1 Homomorphic Matrix Completion Algorithm on User Devices j, for $j \in [J]$

Input: The raw vector $\boldsymbol{M}_j \in \mathbb{R}^I$, index vector $\boldsymbol{\Phi}_j \in \mathbb{R}^I$.

1: **Initialize**: Randomly generate $\psi_0, \psi_1, \ldots, \psi_K \in [0, 1]$ and $\sum_{i=0}^{K} \psi_i = 1$,
2: Receive public matrix $\boldsymbol{P}$ from the cloud server (line 2 in Alg. 2),
3: Encrypt $\boldsymbol{M}_j$ to $\overline{\boldsymbol{M}}_j$ through $\overline{\boldsymbol{M}}_j = (\psi_0 \boldsymbol{M}_j + \psi_1 \boldsymbol{P}_1 + \cdots + \psi_K \boldsymbol{P}_K) \circ \boldsymbol{\Phi}_j$,
4: Upload the encrypted vector $\overline{\boldsymbol{M}}_j$ to the cloud server,
5: Receive the recovered vector $\widehat{\boldsymbol{M}}_j$ from the cloud server (line 8 in Alg. 2),
6: Decrypt $\widehat{\boldsymbol{M}}_j$ to obtain $\widehat{\boldsymbol{D}}_j$ through $\widehat{\boldsymbol{D}}_j = (\widehat{\boldsymbol{M}}_j - \psi_1 \boldsymbol{P}_1 - \cdots - \psi_K \boldsymbol{P}_K)/\psi_0$,

Output: $\widehat{\boldsymbol{D}}_j$.

3) HADAMARD PRODUCT (ELEMENT-WISE PRODUCT)

The Hadamard product $\boldsymbol{Z}$ of $\boldsymbol{X} \in \mathbb{R}^{I \times J}$ and $\boldsymbol{Y} \in \mathbb{R}^{I \times J}$ is $\boldsymbol{Z} = \boldsymbol{X} \circ \boldsymbol{Y}$, where $\boldsymbol{Z}_{ij} = \boldsymbol{X}_{ij}\boldsymbol{Y}_{ij}$.

4) RAW MATRIX

The raw (incomplete) matrix $\boldsymbol{M}$ can be represented as the Hadamard product of the true (complete) matrix $\boldsymbol{D} \in \mathbb{R}^{I \times J}$ and the index matrix $\boldsymbol{\Phi} \in \mathbb{R}^{I \times J}$, i.e., $\boldsymbol{M} = \boldsymbol{D} \circ \boldsymbol{\Phi} \in \mathbb{R}^{I \times J}$.

B. OVERVIEW OF THE HOMOMORPHIC MATRIX COMPLETION ALGORITHM

The homomorphic matrix completion algorithm [19] consists of three steps, as shown in Fig. 2:

1) Encryption of raw data on user devices as in Fig. 2(a): each user uses the public matrix $\boldsymbol{P} \in \mathbb{R}^{I \times K}$ from cloud computing nodes to encrypt her data vector on user devices such as mobile phones. These encrypted vectors are sent to a cloud computing node and form an encrypted matrix.
2) Matrix completion on cloud computing nodes as in Fig. 2(b): the encrypted matrix is recovered targeting a low rank R using the matrix completion algorithm on the cloud computing node.
3) Decryption of reconstructed data on user devices as in Fig. 2(c): the reconstructed matrix $\widehat{\boldsymbol{M}}$ is returned to users. Each user obtains her own reconstructed data vector by decrypting her column in the reconstructed matrix using her private keys on a user device.

In the first and third steps, homomorphic encryption and decryption are used to encrypt the raw (incomplete) data and decrypt the reconstructed data, so as to protect data privacy. In the second step, the alternating minimization algorithm [7] [11] is utilized for matrix completion. The parts of the homomorphic matrix completion algorithm on user devices and on cloud computing nodes are described in Algs. 1 and 2, respectively.

Algorithm 2 Homomorphic Matrix Completion Algorithm on Cloud Computing Nodes

Input: Index matrix $\boldsymbol{\Phi} \in \mathbb{R}^{I \times J}$, target low rank R, number of iterations L.

1: **Initialize**: Public matrix $\boldsymbol{P} \in \mathbb{R}^{I \times K}$, matrix $\boldsymbol{X}^0 \in \mathbb{R}^{I \times R}$,
2: Broadcast $\boldsymbol{P}$ to all J users,
3: Gather all J encrypted vectors and form the encrypted matrix $\overline{\boldsymbol{M}} \in \mathbb{R}^{I \times J}$,
4: **for** $\ell = 1$ to L **do**
5: $\quad \boldsymbol{Y}^\ell = \arg\min_{\boldsymbol{Y}^\ell \in \mathbb{R}^{R \times J}} \|\overline{\boldsymbol{M}} - \boldsymbol{\Phi} \circ (\boldsymbol{X}^{\ell-1}\boldsymbol{Y}^\ell)\|_F^2$,
6: $\quad \boldsymbol{X}^\ell = \arg\min_{\boldsymbol{X}^\ell \in \mathbb{R}^{I \times R}} \|\overline{\boldsymbol{M}} - \boldsymbol{\Phi} \circ (\boldsymbol{X}^{\ell}\boldsymbol{Y}^\ell)\|_F^2$,
7: **end for**
8: Compute the reconstructed matrix $\widehat{\boldsymbol{M}} = \boldsymbol{X}^L \boldsymbol{Y}^L$,
9: Broadcast the reconstructed vector $\widehat{\boldsymbol{M}}_j$ to the j-th user, for $j \in [J]$,

Output: $\widehat{\boldsymbol{M}}$.

1) ENCRYPTION OF RAW DATA

The encryption of raw data on user devices is shown in lines 2–4 of Alg. 1. Each user uses the public matrix $\boldsymbol{P} \in \mathbb{R}^{I \times K}$ from a cloud computing node to encrypt her own raw data vector $M_1 \in \mathbb{R}^I$ and obtain $\overline{M}_1 \in \mathbb{R}^I$ on a user device. Since user devices only have limited computation power, memory capacity and power budget, the homomorphic matrix completion algorithm [19] utilizes a light-weight encryption scheme as follows:

$$\overline{\boldsymbol{M}}_j = (\psi_0 \boldsymbol{M}_j + \psi_1 \boldsymbol{P}_1 + \cdots + \psi_K \boldsymbol{P}_K) \circ \boldsymbol{\Phi}_j \in \mathbb{R}^I, \quad (1)$$

where $\psi_0, \psi_1, \ldots, \psi_K \in [0, 1]$ are private keys generated on user devices and $\sum_{i=0}^{K} \psi_i = 1$. Each user generates her own private keys ψ and encrypt her own data vector. Therefore, user j encrypts a raw data vector $\boldsymbol{M}_j \in \mathbb{R}^I$ and produces an encrypted data vector $\overline{\boldsymbol{M}}_j \in \mathbb{R}^I$. The encrypted data vectors $\overline{M}_j$ from J users are sent to a cloud computing node and form an encrypted matrix $\overline{M} \in \mathbb{R}^{I \times J}$, where each encrypted data vector becomes a column of the encrypted matrix $\overline{M}$. Let K denote the rank of the public matrix $\boldsymbol{P}$, and $R - K$ denote the rank of the raw (incomplete) matrix before encryption, then the rank of the encrypted matrix $\overline{M}$ is denoted by R.

2) MATRIX COMPLETION ON CLOUD COMPUTING NODES

The matrix completion on cloud computing nodes is shown in Alg. 2. First, the cloud computing node generates a public matrix $\boldsymbol{P} \in \mathbb{R}^{I \times K}$ and an initial matrix $\boldsymbol{X}^0 \in \mathbb{R}^{I \times R}$ (in line 1), where R is the target rank of the encrypted matrix. The public matrix $\boldsymbol{P}$ is constructed by selecting K public vectors from all available public vectors or generated randomly. Public vectors are subsidiary data that are complete and without encryption, which are provided by some users [19]. The initial $\boldsymbol{X}^0$ is generated randomly for the specified size. Second, the cloud computing node broadcasts the public matrix $\boldsymbol{P}$ to all J users, then gathers all J encrypted vectors and form an encrypted matrix $\overline{\boldsymbol{M}} \in \mathbb{R}^{I \times J}$ (in lines 2–3). Third, the cloud computing node performs the compute-intensive matrix completion (in lines 4–8). We adopt the alternating minimization algorithm [7] [11] [14] to solve matrix completion. In each iteration, we fix matrix $\boldsymbol{X} \in \mathbb{R}^{I \times R}$ to solve $\boldsymbol{Y} \in \mathbb{R}^{R \times J}$ using (2) and then fix $\boldsymbol{Y}$ to solve $\boldsymbol{X}$ using (3). The core operation is least squares minimization. The reconstructed matrix $\widehat{\boldsymbol{M}} \in \mathbb{R}^{I \times J}$ is computed as the matrix multiplication of $\boldsymbol{X}^L$ and $\boldsymbol{Y}^L$ (line 8).

$$\boldsymbol{Y} = \arg\min_{\boldsymbol{Y} \in \mathbb{R}^{R \times J}} \|\overline{\boldsymbol{M}} - \boldsymbol{\Phi} \circ (\boldsymbol{XY})\|_F^2, \quad (2)$$

$$\boldsymbol{X} = \arg\min_{\boldsymbol{X} \in \mathbb{R}^{I \times R}} \|\overline{\boldsymbol{M}} - \boldsymbol{\Phi} \circ (\boldsymbol{XY})\|_F^2. \quad (3)$$

Finally, the cloud computing node broadcast the reconstructed vector $\widehat{\boldsymbol{M}}_j$ to the j-th user, for $j \in [J]$.

3) DECRYPTION OF RECONSTRUCTED DATA

The decryption of reconstructed data on user devices is described in line 6 of Alg. 1. The homomorphic matrix completion algorithm [19] utilizes a light-weight decryption scheme as follows:

$$\widehat{\boldsymbol{D}}_j = (\widehat{\boldsymbol{M}}_j - \psi_1 \boldsymbol{P}_1 - \cdots - \psi_K \boldsymbol{P}_K)/\psi_0 \in \mathbb{R}^I. \quad (4)$$

Each user utilizes her own private keys $\psi_0, \psi_1, \ldots, \psi_K$ to perform decryption on her reconstructed vector $\widehat{\boldsymbol{M}}_j$ to obtain decrypted data column $\widehat{\boldsymbol{D}}_j$.

C. PARALLEL ACCELERATION ANALYSIS

We profiled the CPU MATLAB implementation of the homomorphic matrix completion algorithm [19] and found that the second step took more than 95% of the total running time for most matrix sizes. Therefore, the efficiency of the algorithm can be improved by accelerating the second step.

In Alg. 2, the algorithm solves $\boldsymbol{Y}^\ell$ and $\boldsymbol{X}^\ell$ alternately within the for-loop (lines 4–7), then computes the reconstructed matrix (line 8). This process involves intensive computations, and we propose to offload these computations from CPU to high-performance GPU. GPUs are effective for accelerating massively parallel computations such as matrix operations. The computations in lines 4–8 involve matrix Hadamard product, matrix multiplication, matrix transpose, and least square minimization, which can be accelerated on GPU. Thus, we focus on the design, implementation and optimizations of the matrix completion (lines 4–8) implementation on GPU.

III. EFFICIENT HOMOMORPHIC MATRIX COMPLETION ON GPU

We parallelize the homomorphic matrix completion algorithm [19] in Alg. 2 on GPU. Then we propose techniques to optimize memory accesses, GPU utilizations and CPU-GPU communications.

A. DESIGN AND IMPLEMENTATION OF THE BASELINE GPU HOMOMORPHIC MATRIX COMPLETION

1) DATA STORAGE

We store the data in the column-major format because of two reasons. First, the homomorphic matrix completion algorithm

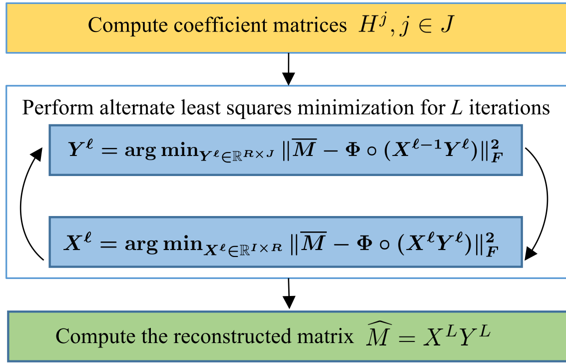

FIGURE 3. The three-step baseline of homomorphic matrix completion on GPU.

in Alg. 2 always accesses each column of matrices, therefore the column-major storage guarantees continuous memory accesses. Second, the GPU-based homomorphic matrix completion utilizes some routines in the CUDA libraries that use the column-major format.

2) PARALLELIZATION OF THE HOMOMORPHIC MATRIX COMPLETION

In Alg. 2, the major computations are the for-loop in lines 4–7 for alternate least square minimizations and the matrix multiplications in line 8. Lines 5 and 6 inside the for-loop perform the least squares minimizations in (2) and (3), respectively. Since the processes to obtain $\boldsymbol{Y}$ (line 5) and $\boldsymbol{X}$ (line 6) are quite similar, we only discuss the process to obtain $\boldsymbol{Y}$ in the following sections for conciseness. We use the following equation to solve the least squares minimization in (2) and get a column $\boldsymbol{Y}_j$ of $\boldsymbol{Y}$:

$$\boldsymbol{Y}_j = \arg\min_{\boldsymbol{Y}_j \in \mathbb{R}^R} \|\overline{\boldsymbol{M}}_j - \boldsymbol{H}^j \boldsymbol{Y}_j\|_F^2, \tag{5}$$

where $\boldsymbol{H}^j \in \mathbb{R}^{I\times R}$ is the j-th matrix in a series of coefficient matrices and R is the target rank of the encrypted matrix. $\boldsymbol{H}^j$ is calculated as the Hadamard product of $\boldsymbol{\Phi}_j$ and each column of matrix $\boldsymbol{X}$ as follows:

$$\boldsymbol{H}^j(i, r) = \boldsymbol{\Phi}(i, j) \circ \boldsymbol{X}(i, r). \tag{6}$$

We compute lines 4–8 in Alg. 2 on GPU in the following three steps, as shown in Fig. 3:

1) Computing coefficient matrices $\boldsymbol{H}^j$: Compute J coefficient matrices $\boldsymbol{H}^1, \boldsymbol{H}^2, \cdots, \boldsymbol{H}^J$ using (6). In the baseline GPU implementation, we compute these J coefficient matrices in parallel on GPU. We allocate multiple GPU threads and use one GPU thread to calculate each $\boldsymbol{H}^j \in \mathbb{R}^{I\times R}$.
2) Using (5) to solve the least squares minimizations in lines 5–6 of Alg. 2. We perform QR factorization on each $\boldsymbol{H}^j$ to obtain an orthogonal matrix and an upper triangular matrix. Therefore, a single least squares minimization is solved and a column $\boldsymbol{Y}_j$ of $\boldsymbol{Y}$ is obtained. For an encrypted matrix $\overline{\boldsymbol{M}} \in \mathbb{R}^{I\times J}$, the algorithm computes J least squares minimizations for $\boldsymbol{Y}$ and J least squares minimizations for $\boldsymbol{X}$ in each iteration. Therefore, the algorithm perform $(I+J)L$ least squares minimizations for L iterations.
3) Performing matrix multiplication of $\boldsymbol{X}^L$ and $\boldsymbol{Y}^L$ (in line 8 of of Alg. 2) to obtain the reconstructed matrix $\widehat{\boldsymbol{M}}$. We utilize the high-performance matrix multiplication routine in the cuBLAS library [21].

B. OPTIMIZATIONS

We performed preliminary experiments to evaluate the baseline GPU homomorphic matrix completion scheme, and found that it achieved merely similar performance with the CPU MATLAB implementation. We profiled the baseline scheme using the profiler in CUDA and observed three major bottlenecks: low memory access efficiency and big memory footprint, low utilization of GPU compute units, and high communication cost between CPU and GPU. Therefore, we optimize the baseline scheme on these aspects.

1) OPTIMIZING MEMORY ACCESSES

We optimize the memory access efficiency in GPU-based computation. The first step of lines 4–8 in Alg. 2 is to compute a series of coefficient matrices $\boldsymbol{H}^j$ using (6). In the baseline GPU implementation, $\boldsymbol{\Phi}$ and $\boldsymbol{X}$ are stored in the GPU global memory which has the largest capacity on GPU. During the computing, each column of $\boldsymbol{\Phi}$ in the GPU global memory is accessed J times. However, the access latency of the GPU global memory is high and introduce high time cost. To improve memory access efficiency, we store $\boldsymbol{\Phi}$ into low-latency, small shared memory inside each GPU streaming multiprocessor (SM). To compute J coefficient matrices, we launch J thread blocks and copy $\boldsymbol{\Phi}_j$ from the GPU global memory to the shared memory on j-th block. In this way, the algorithm accesses the shared memory J times to calculate $\boldsymbol{H}^j$, which is much faster than accessing the GPU global memory.

We reduce the memory footprint in GPU computation. The matrix form of (5) is $\boldsymbol{H}^j\boldsymbol{Y}_j = \overline{\boldsymbol{M}}_j$, where $\boldsymbol{H}^j \in \mathbb{R}^{I\times R}$ and $\overline{\boldsymbol{M}}_j \in \mathbb{R}^{I\times 1}$. To solve all J columns of $\boldsymbol{Y}$, the space complexity is $O(IRJ)$, which increases rapidly as the grow of encrypted matrix size $I \times J$ and imposes pressure on limited GPU memory capacity. We multiply the transposed matrix $(\boldsymbol{H}^j)^T$ to the matrix form of (5) and derive the following equation:

$$\boldsymbol{H}^j\boldsymbol{Y}_j = \overline{\boldsymbol{M}}_j \quad \Rightarrow \quad (\boldsymbol{H}^j)^T\boldsymbol{H}^j\boldsymbol{Y}_j = (\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j. \tag{7}$$

In (7), $(\boldsymbol{H}^j)^T\boldsymbol{H}^j$ is $R\times R$ and $(\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j$ is $R\times 1$, so the space complexity becomes $O(R^2J)$ for all J columns of $\boldsymbol{Y}$. Since the encrypted matrix $\overline{\boldsymbol{M}}$ holds the low-rank property, the target rank R is much smaller than I and J. Therefore, the memory consumption is significantly reduced. Since we store $(\boldsymbol{H}^j)$ in the column major format, we can obtain $(\boldsymbol{H}^j)^T$ through fetch the elements of $(\boldsymbol{H}^j)$ without explicit matrix transpose.

2) IMPROVING GPU UTILIZATION

In (7), there are two matrix multiplications to calculate $(\boldsymbol{H}^j)^T\boldsymbol{H}^j$ and $(\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j$, and a least square minimization.

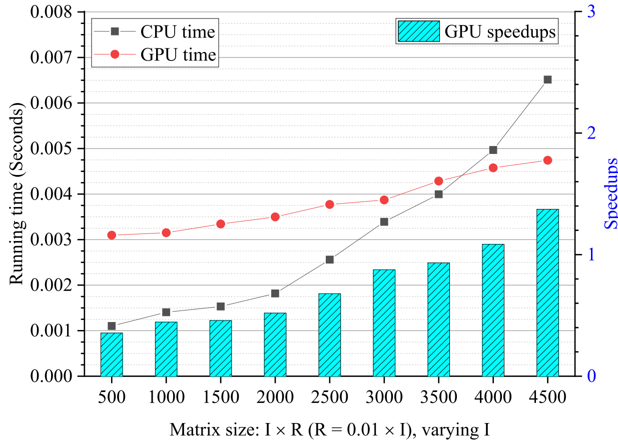

FIGURE 4. The performance of a single least squares minimization on two CPUs and the Tesla V100 GPU, respectively.

So there are $2J$ matrix multiplications and J least square minimizations to calculate all J columns of $\boldsymbol{Y}$. Similarly, there are multiple matrix multiplications and least square minimizations to obtain $\boldsymbol{X}$. In regular GPU implementation, these matrix multiplications or least square minimizations are computed one by one on GPU. However, since GPUs have thousands of cores, a single matrix multiplication or least squares minimization is often inadequate to fully utilize all GPU cores. We performed preliminary experiment to evaluate the performance of a single least squares minimization in (5). The performance under varying matrix size of $I \times R$ on two Xeon CPUs or a Tesla V100 GPU is shown in Fig. 4. The least squares minimization on the Tesla V100 GPU poorly achieved an average of $0.75\times$ and up to $1.37\times$ speedups versus two Xeon CPUs, due to the fact that a single least squares minimization is inadequate to fully utilize all GPU resources.

We improve GPU utilization by organizing and batching multiple matrix computation or the least squares minimizations to run parallelly on GPU. Since there are no dependency between these computations, we organize and compute multiple matrix computation or the least squares minimizations in batches. We compute $(\boldsymbol{H}^j)^T\boldsymbol{H}^j,\ j \in [J]$ in a batch and $(\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j,\ j \in [J]$ in another batch by exploiting the batched matrix multiplication routine in the cuBLAS library [21]. After these batched matrix multiplications, we obtain J pairs of data from $\left[(\boldsymbol{H}^1)^T\boldsymbol{H}^1, (\boldsymbol{H}^1)^T\overline{\boldsymbol{M}}_1\right]$ to $\left[(\boldsymbol{H}^J)^T\boldsymbol{H}^J, (\boldsymbol{H}^J)^T\overline{\boldsymbol{M}}_J\right]$. Each pair of data is used for solving a single least squares minimization in (7). We organize and solve these J least squares minimizations parallelly in a batched by exploiting the batched Lower-Upper (LU) decomposition routine in the MAGMA library [22], as illustrated in Fig. 5.

The pseudo code of the optimized GPU matrix completion algorithm is described in Alg. 3. The most outer for-loop in lines 2–15 executes the matrix completion for L iterations. There are two parfor-loop in lines 3–6 and 8–11, in which the computations are in parallel (i.e., batched). The function getCoefMatrix($\cdot$) in lines 4 and 9 calculates a coefficient

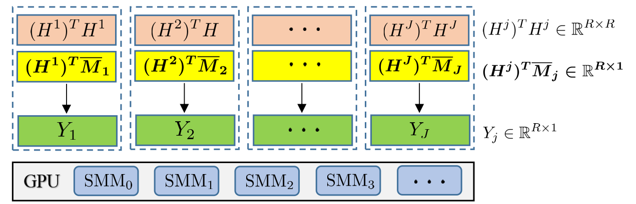

FIGURE 5. Batched least squares minimizations.

Algorithm 3 Pseudo code of the optimized GPU matrix completion

Input: Encrypted matrix $\overline{\boldsymbol{M}} \in \mathbb{R}^{I\times J}$, target rank R, index matrix $\boldsymbol{\Phi} \in \mathbb{R}^{I\times J}$, maximum iterations L.

1: **Initialize**: Matrix $\boldsymbol{X}^0 \in \mathbb{R}^{I\times R}$,
2: **for** $\ell = 1$ to L **do**
3:   **parfor** $j = 1$ to J **do**
4:     $\boldsymbol{H}^j$ = getCoefMatrix($\boldsymbol{X}^{\ell-1}, \boldsymbol{\Phi}_j$),
5:     $\boldsymbol{Y}_j^\ell \leftarrow$ LSM($(\boldsymbol{H}^j)^T\boldsymbol{H}^j\boldsymbol{Y}_j^\ell = (\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j$),
6:   **end parfor**
7:   $(\boldsymbol{Y}^\ell)^T$ = Transpose($\boldsymbol{Y}^\ell$),
8:   **parfor** $i = 1$ to I **do**
9:     $\boldsymbol{U}^i$ = getCoefMatrix($(\boldsymbol{Y}^\ell)^T, \boldsymbol{\Phi}_i^T$),
10:     $(\boldsymbol{X}^\ell)_i^T \leftarrow$ LSM($(\boldsymbol{U}^i)^T\boldsymbol{U}^i(\boldsymbol{X}^\ell)_i^T = (\boldsymbol{U}^i)^T\overline{\boldsymbol{M}}_i^T$),
11:   **end parfor**
12:   $\boldsymbol{X}^\ell$ = Transpose($(\boldsymbol{X}^\ell)^T$),
13: **end for**
14: $\widehat{\boldsymbol{M}} = \boldsymbol{X}^L\boldsymbol{Y}^L$,

Output: $\widehat{\boldsymbol{M}}$.

matrix H^j, and all J coefficient matrices are calculated parallelly on GPU. The function LSM($\cdot$) in lines 5 and 10 is the least squares minimization function. The transpose($\cdot$) function in lines 7 and 12 performs matrix transposition.

3) REDUCING COMMUNICATIONS

The profiling result on the baseline GPU implementation showed a high communication cost between CPU and GPU due to a large amount of data transfers. In the optimized GPU matrix completion implementation in Alg. 3, the communication could be a bottleneck if not optimized. We reduce communication cost by eliminating data transfers of intermediate results. We transfer $\boldsymbol{X}^0$, $\boldsymbol{\Phi}$ and $\overline{\boldsymbol{M}}$ from CPU memory to GPU memory and continuously calculate lines 4–6 of Alg. 3 on the GPU without transferring intermediate results back to the CPU memory. Similar method is applied to lines 10–12.

IV. LARGE-SCALE AND MULTI-GPU HOMOMORPHIC MATRIX COMPLETION

A. LARGE-SCALE HOMOMORPHIC MATRIX COMPLETION ON A SINGLE GPU

Although external GPUs have much higher computation capability than integrated GPUs, their GPU memory capacity is limited. For large matrices, the memory required for matrix completion exceeds GPU memory capacity, limiting

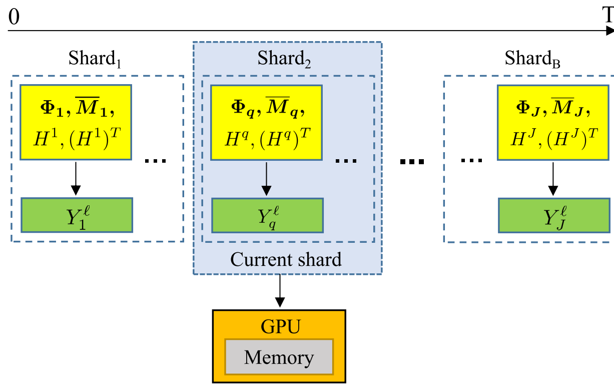

FIGURE 6. Shard mode for large-scale homomorphic matrix completion.

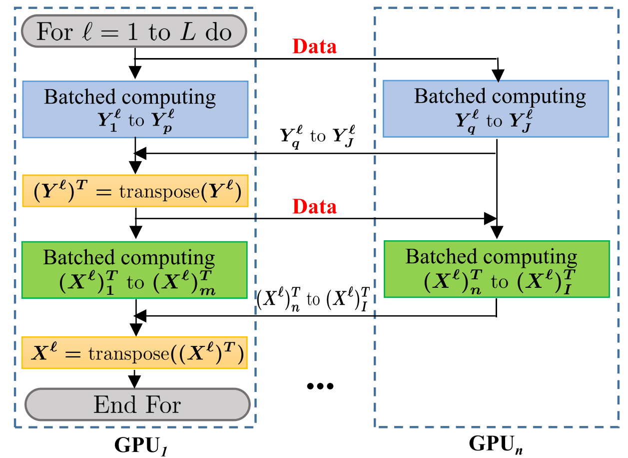

FIGURE 7. Multi-GPU scheme for homomorphic matrix completion.

the applicability of external GPUs to small- or medium-sized matrices. By exploiting the separability of the matrix completion algorithm, we design a shard mode to perform large-scale homomorphic matrix completion on a single GPU.

1) SHARDS

As shown in Alg. 3, the parfor-loop in lines 3–6 to solve all columns of $\boldsymbol{Y}^{\ell}$ and the parfor-loop in lines 8–11 to solve all columns of $(\boldsymbol{X}^{\ell})^T$ consume most of the GPU global memory during computing. Take $\boldsymbol{Y}^{\ell}$ as example, J coefficient matrices $\boldsymbol{H}^j \in \mathbb{R}^{I\times R}$ are computed parallelly (in line 4). Then there are two batched multiplications to compute J parallel $(\boldsymbol{H}^j)^T\boldsymbol{H}^j$ and $(\boldsymbol{H}^j)^T\overline{\boldsymbol{M}}_j$, and a batched least squares minimization to obtain all columns of $\boldsymbol{Y}^{\ell}$ (in line 5). Since the computations to obtain different column of $\boldsymbol{Y}^{\ell}$ are independent of each other (i.e., computing $\boldsymbol{Y}_i^{\ell}$ is independent of computing $\boldsymbol{Y}_j^{\ell}$, for $i \neq j$). Therefore, we split the computations and data into shards (i.e., partitions) and each partition computes several columns of $\boldsymbol{Y}^{\ell}$ or $(\boldsymbol{X}^{\ell})^T$. The shards to compute columns of $\boldsymbol{Y}^{\ell}$ are called $\boldsymbol{Y}^{\ell}$ shards while those to compute columns of $\boldsymbol{X}^{\ell}$ are called $\boldsymbol{X}^{\ell}$ shards. Fig. 6 shows example $\boldsymbol{Y}^{\ell}$ shards.

Suppose the GPU has S bytes of available memory and the required memory for computing $\boldsymbol{Y}^{\ell}$ or $(\boldsymbol{X}^{\ell})^T$ on an $I \times J$ encrypted matrix is a function of I and J denoted by f(I, J), then the number of $\boldsymbol{Y}^{\ell}$ shards or $\boldsymbol{X}^{\ell}$ shards B is determined as $B = \mathrm{f}(I, J)/S$.

2) LARGE-SCALE HOMOMORPHIC MATRIX COMPUTING WITH THE SHARD MODE

We propose a shard mode to perform large-scale homomorphic matrix completion exceeding the GPU memory capacity. Using this mode, we perform the computation in the ℓ-th iteration (in lines 2–13 of Alg.3) as follows:

- Computing $\boldsymbol{Y}^{\ell}$ shards one by one on the GPU to obtain $\boldsymbol{Y}^{\ell}$, as shown in Fig. 6;
- Transposing $\boldsymbol{Y}^{\ell}$ to obtain $(\boldsymbol{Y}^{\ell})^T$;
- Computing $\boldsymbol{X}^{\ell}$ shards one by one on the GPU to obtain $(\boldsymbol{X}^{\ell})^T$;
- Transposing $(\boldsymbol{X}^{\ell})^T$ to obtain $\boldsymbol{X}^{\ell}$;

B. MULTI-GPU HOMOMORPHIC MATRIX COMPLETION

Cloud computing nodes are often equipped with multiple GPUs, therefore we design a multi-GPU scheme to fully utilize multiple GPUs in a node for homomorphic matrix completion, as illustrated in Fig. 7. Since the computing of columns of $\boldsymbol{Y}^{\ell}$ or $(\boldsymbol{X}^{\ell})^T$ is independent of each other, we can distributed the computation among multiple GPUs to compute in parallel. Assuming there are n GPUs, denoted by GPU_1, ..., GPU_n, in a computing node and the size of the encrypted matrix is $I \times J$, we perform the computation in the ℓ-th iteration (in lines 2–13 of Alg. 3) as follows:

- GPU_j computes C_j continuous columns of $\boldsymbol{Y}^{\ell}$, where C_j is proportional to the performance of the GPU (e.g., GFLOPs when performing matrix multiplication) and $\sum_{j=1}^{n} C_j = J$, where n is the number of GPUs. GPU_1 sends the data for computing to other GPUs using direct GPU to GPU communication. Each GPU computes its assigned columns of $\boldsymbol{Y}^{\ell}$ using batched matrix multiplications and batched least square minimization. Upon completion, all other GPUs send their results to GPU_1 using direct GPU to GPU communication to form $\boldsymbol{Y}^{\ell}$;
- GPU_1 performs matrix transpose on $\boldsymbol{Y}^{\ell}$ to obtain $(\boldsymbol{Y}^{\ell})^T$;
- GPU_i computes C_i continuous columns of $(\boldsymbol{X}^{\ell})^T$, where C_i is proportional to the performance of the GPU (eg., GFLOPs when performing matrix multiplication) and $\sum_{i=1}^{n} C_i = I$, where n is the number of GPUs. GPU_1 sends the data for computing to other GPUs using direct GPU to GPU communication. Each GPU computes its assigned columns of $(\boldsymbol{X}^{\ell})^T$ using batched matrix multiplications and batched least square minimization. Upon completion, all other GPUs send their results to GPU_1 using direct GPU to GPU communication to form $(\boldsymbol{X}^{\ell})^T$;
- GPU_1 performs matrix transpose on $(\boldsymbol{X}^{\ell})^T$ to obtain $\boldsymbol{X}^{\ell}$.

In this scheme, the direct GPU to GPU communication is implemented by exploiting the asynchronous, GPU to GPU memory copying routine in the CUDA library [21], which supports the communication between two and more GPUs.

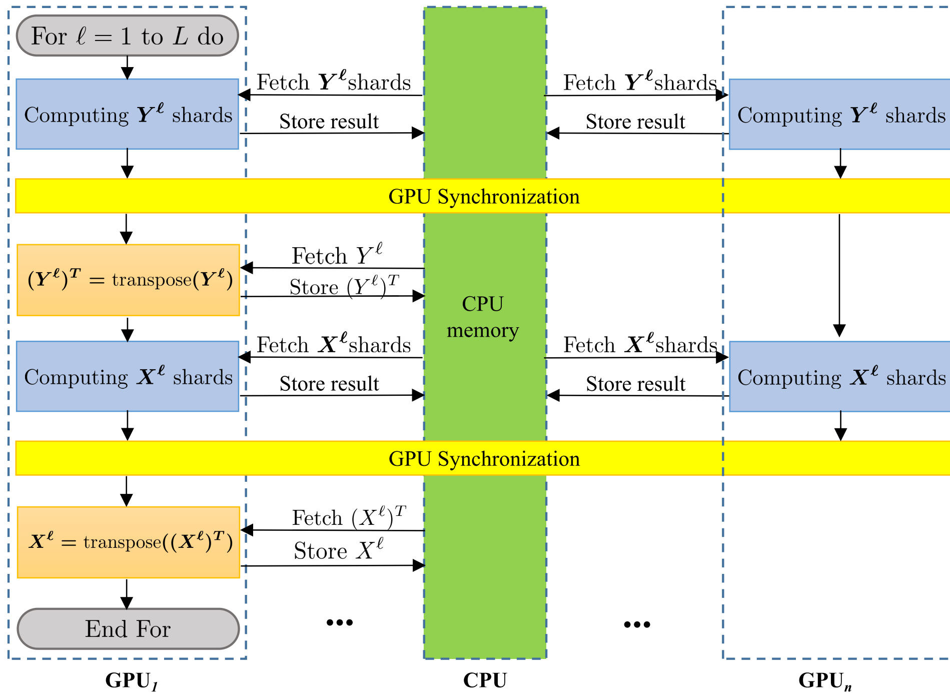

FIGURE 8. Large-scale homomorphic matrix completion on multiple GPUs.

C. LARGE-SCALE HOMOMORPHIC MATRIX COMPLETION ON MULTIPLE GPUs

We further design a scheme to perform large-scale homomorphic matrix completion exceeding GPU memory capacity on Multiple GPUs, as illustrated in Fig. 8. The basic idea is to exploit large CPU memory to store all the data, then distribute shards among multiple GPUs, and utilize GPU synchronization to ensure correct algorithm flow. Assuming there are n GPUs, denoted by GPU$_1$, ..., GPU$_n$, in a computing node, we perform the computation in the ℓ-th iteration (in lines 2–13 of Alg. 3) as follows:

- Each GPU fetches a $\boldsymbol{Y}^\ell$ shard from CPU memory, computes it and stores results (i.e., columns of $\boldsymbol{Y}^\ell$) into CPU memory. Then the GPU that finishes computing continues to fetch the next shard if there are remaining $\boldsymbol{Y}^\ell$ shards;
- Performing GPU synchronization across all GPUs to ensure the computing of all $\boldsymbol{Y}^\ell$ shards are completed;
- GPU$_1$ fetches $\boldsymbol{Y}^\ell$ from CPU, performs matrix transpose on $\boldsymbol{Y}^\ell$ to obtain $(\boldsymbol{Y}^\ell)^T$, and store $(\boldsymbol{Y}^\ell)^T$ back to CPU;
- Each GPU fetches a $\boldsymbol{X}^\ell$ shard from CPU memory, compute it and store results (i.e., columns of $(\boldsymbol{X}^\ell)^T$) into CPU memory. Then the GPU that finishes computing continues to fetch the next shard if there are remaining $\boldsymbol{X}^\ell$ shards;
- Performing GPU synchronization across all GPUs to ensure the computing of all $\boldsymbol{X}^\ell$ shards are completed;
- GPU$_1$ fetches $(\boldsymbol{X}^\ell)^T$ from CPU, performs matrix transpose on $(\boldsymbol{X}^\ell)^T$ to obtain $\boldsymbol{X}^\ell$, and store $\boldsymbol{X}^\ell$ back to CPU.

V. PERFORMANCE EVALUATION

A. EVALUATION SETTINGS

The experiment platform has two Intel Xeon E5-2640V4 CPUs, an NVIDIA Tesla V100 GPU (Volta architecture), a Quadro RTX6000 GPU (Turin architecture), and 80 GB DDR memory. Each E5-2640V4 CPU has 10 cores that support 20 threads with hyperthreading technology. The V100 GPU has 5120 CUDA cores with 32 GB DDR memory, achieving 14 TFLOPs single-precision performance and 900 GB/s memory bandwidth. The RTX6000 GPU has 4608 CUDA cores with 24 GB DDR memory, achieving 16.3 TFLOPs single-precision performance and 672 GB/s memory bandwidth. The two GPUs are running on driver 430.40. The operating system is Ubuntu 18.04 64bit. The GPU implementation and the CPU implementation are executed on CUDA 10.1 and MATLAB 2017b, respectively.

We generate low-rank matrices of varying size ($I \times J$) for performance evaluation. The rank of matrices R is set to $0.01 \times \min(I, J)$. The data missing rate of matrices is set to 50%. For single GPU performance, we run experiments on the Tesla V100 GPU. For multi-GPU performance, we run experiments on Tesla V100 and Quadro RTX6000 parallelly. Both CPU and GPU implementations executes the homomorphic matrix completion algorithm for $L = 10$ iterations. All experiment are executed for five times and we report the average results.

We adopt two metrics for performance evaluation: running time and recovery error. We compare the running time of the CPU MATLAB implementation and the GPU implementation on varying matrix sizes, and calculate speedups as

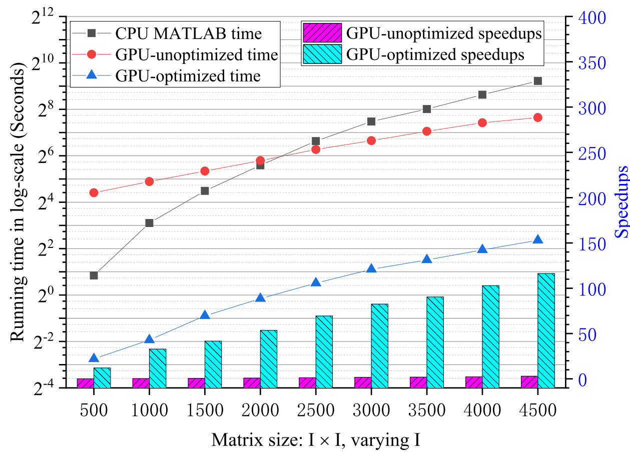

FIGURE 9. Running time and speedups of the homomorphic matrix completion algorithm on the Tesla V100 GPU and two Xeon CPUs for small- or medium-scale matrices.

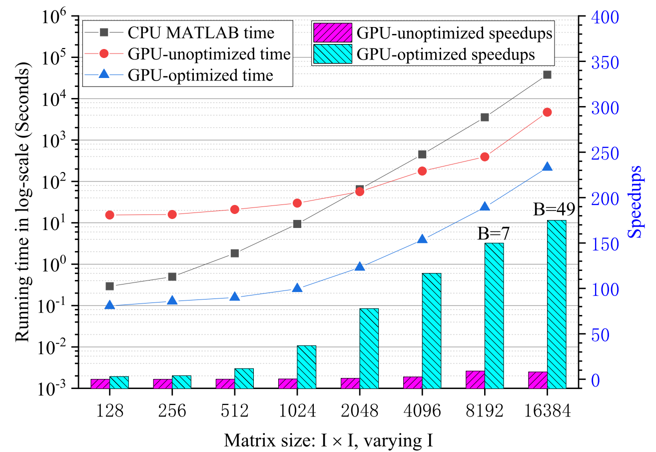

FIGURE 11. Running time and speedups of the large-scale GPU homomorphic matrix completion on a single GPU.

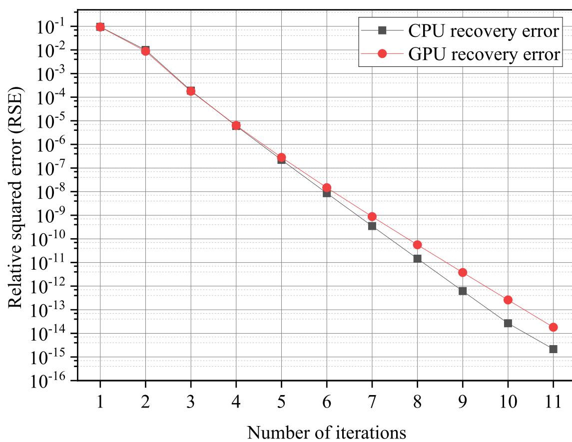

FIGURE 10. Recovery error of the homomorphic matrix completion algorithm on the Tesla V100 GPU and two Xeon CPUs, respectively.

(longer running time / shorter running time). The recovery error is defined as RSE $= \|\boldsymbol{D}-\widehat{\boldsymbol{D}}\|_F/\|\boldsymbol{D}\|_F$, where $\widehat{\boldsymbol{D}}$ is given in (4).

B. BASIC PERFORMANCE OF THE GPU HOMOMORPHIC MATRIX COMPLETION

Fig. 9 shows the performance of the baseline (i.e., unoptimized) and the optimized GPU homomorphic matrix completion running on the Tesla V100 GPU and the CPU MATLAB implementation running on two Xeon CPUs, respectively. The optimized GPU homomorphic matrix completion achieved up to 116.23× speedups versus two Xeon CPUs. In contrast, the baseline GPU homomorphic matrix completion achieved up to 2.98× speedups versus two Xeon CPUs. The baseline GPU implementation adopts none of the optimization techniques in Section III-B. As a result, it has poor resource utilization and high CPU-GPU communication cost. The optimized GPU homomorphic matrix completion achieved higher speedups on bigger matrix sizes. However, it supports up to $4,500 \times 4,500$ matrix due to limited GPU memory capacity.

Fig. 10 shows the recovery error of the optimized GPU homomorphic matrix completion on the Tesla V100 GPU and the CPU MATLAB implementation on two Xeon GPUs, respectively. The matrix is $1,000 \times 1,000$ with a 50% data missing rate. Both GPU and CPU MATLAB implementations used double precision float. The GPU and CPU implementation achieved similar recovery errors under different number of iterations. The recovery error keep decreasing with the number of iterations till $10e-15$ which is the limit of double precision float in MATLAB.

C. PERFORMANCE OF THE LARGE-SCALE AND MULTI-GPU HOMOMORPHIC MATRIX COMPLETION

Fig. 11 shows the performance on large-scale matrices using the shard mode. Due to the long running time of the CPU MATLAB implementation for comparison, we tested matrix sizes up to $16,384 \times 16,384$. The memory requirements for performing homomorphic matrix completion on two largest matrices $8,192 \times 8,192$ and $16,384 \times 16,384$ exceed the memory capacity of Tesla V100 GPU, therefore they are split into 7 and 49 shards for computing, respectively. On all matrix sizes, the optimized homomorphic matrix completion achieved up to 174.92× speedups versus the CPU MATLAB implementation. In contrast, the unoptimized GPU implementation poorly achieved up to 9.03× speedups over the CPU MATLAB implementation.

Fig. 12 shows the performance of the multi-GPU GPU homomorphic matrix completion. There are three running time curves to contrast the performance on single Tesla V100 GPU, single Quadro RTX6000 GPU, and two GPUs parallelly. In the two-GPU experiment, we set V100 as GPU_1 and RTX6000 as GPU_2, and the multi-GPU homomorphic matrix completion distributed the computation onto two GPUs and computed parallelly. On matrices of $1,000 \times 1,000$ or bigger, the GPU implementation running on RTX6000 was on average 1.13× faster than

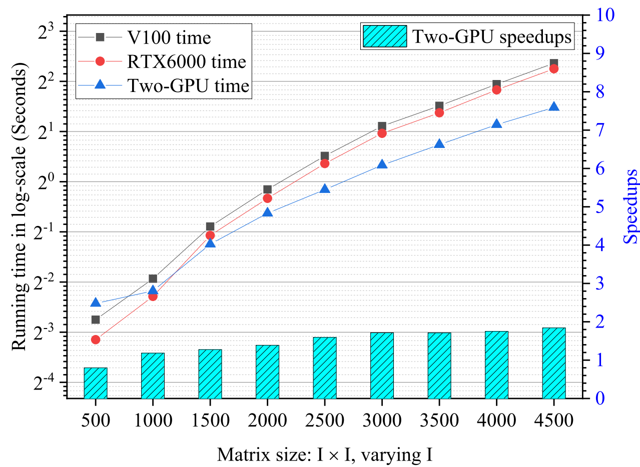

FIGURE 12. Running time and speedups of the multi-GPU homomorphic matrix completion.

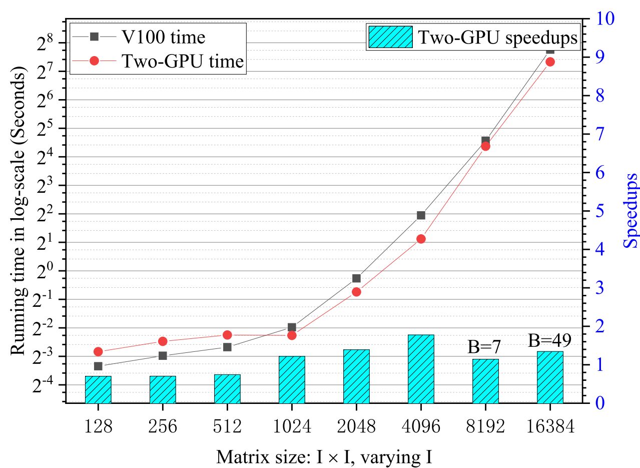

FIGURE 13. Running time and speedups of the large-scale GPU homomorphic matrix completion on multiple GPUs.

the GPU implementation running on V100. The multi-GPU homomorphic matrix completion achieved up to 1.84× speedups against the single-GPU homomorphic matrix completion running on V100. On small matrix of 500 × 500, the multi-GPU homomorphic matrix completion is slower than the single-GPU homomorphic matrix completion running on V100, because the running time is small and the cost to partition computation and the communication between multiple GPUs exceeds the performance benefit from multiple GPUs. Therefore, the multi-GPU scheme is more advantageous for medium and large matrices.

Fig. 13 compares the performance of the large-scale GPU homomorphic matrix completion on multiple GPUs. Two largest matrices $8,192 \times 8,192$ and $16,384 \times 16,384$ exceed the memory capacity of Tesla V100 GPU, therefore they are split into shards for computing. In the two-GPU experiment, we set V100 as GPU_1 and RTX6000 as GPU_2, and the large-scale, multi-GPU homomorphic matrix completion distributed all shards onto two GPUs and computed parallelly. On two matrices $8,192 \times 8,192$ and $16,384 \times 16,384$, the homomorphic matrix completion achieved 1.14× and 1.35× speedups on two GPUs versus on a single V100 GPU, respectively. These speedups are lower than the multi-GPU speedups in Fig. 12 because of two major reasons. First, the large-scale, multi-GPU scheme requires two synchronizations between GPUs in each iteration to ensure the completion of computing, which introduces time overhead. Second, there can be workload imbalance between GPUs when GPUs process different number of shards and lead to GPU waiting time overhead.

VI. RELATED WORKS

Homomorphic encryption [23], [24] is a form of encryption that allows computation on ciphertexts, generating an encrypted result which, when decrypted, matches the result of the operations as if they had been performed on the plaintext. Homomorphic encryption approaches have been applied to diverse fields including secure voting systems, private information retrieval schemes, and collision-resistant hash functions [25], [26]. Kong *et al.* [19] proposed a homomorphic encryption approach for low-rank matrices. Using this approach, incomplete data from multiple users can be reconstructed on cloud computing nodes while still preserving user privacy.

Matrix completion methods are effective in reconstructing two dimensional data. Low rank is often a necessary hypothesis for multi-dimensional data completion to avoid being an undetermined and intractable problem. Low-rank, incomplete matrices can be reconstructed using the alternating minimization algorithm [7] [11] [14]. Candès and Recht [13] proposed a convex optimization method for matrix completion and obtained the sampling lower bound. For data reconstruction of three-dimension and higher-dimension, researchers proposed various tensor completion algorithms, including decomposition-based algorithms, nuclear (trace) norm-based algorithms, and other variants [27].

Data completion algorithms are compute-intensive. Many data completion algorithms are iterative, whose computation complexity grows rapidly with the size and dimension of the data. Therefore, researchers proposed to utilize high-performance GPUs for acceleration. Shah and Majumdar [28] proposed an efficient matrix completion implementation based on stochastic gradient descent (SGD) on GPUs. Later, researchers designed parallel SGD [29] and multi-stream SGD [30] on GPUs that can be used for matrix factorization and completion. For the completion of third-dimensional data, we proposed efficient GPU tensor completion implementations [31], [32]. Different with these works, we propose a high-performance GPU homomorphic matrix completion based on the alternating minimization.

The GPU homomorphic matrix completion in this paper involves matrix operations including matrix multiplication, matrix decomposition (for solving least squares minimization), and matrix transpose. Because matrix algebra is fundamental and widely used, designing high-performance matrix routines could benefit diverse applications. Tao *et al.* [33] accelerated Sparse matrix-vector

multiplication and matrix-transpose vector multiplication. Deveci *et al.* [34] designed efficient GPU sparse matrix-matrix multiplication. Tomov *et al.* [35] accelerated dense linear algebra on hybrid GPU systems. Our work considers how to organize the computation and exploiting matrix routines in batched, large-scale or multi-GPU conditions.

VII. CONCLUSION

In this paper, we propose a GPU-based homomorphic matrix completion scheme for high-performance matrix completion while preserving user privacy. We optimized memory accesses, GPU utilizations, and CPU-GPU communications. The optimized scheme achieved up to 116.23× speedups over the CPU MATLAB implementation running on two Xeon CPUs. To fully utilize multiple GPUs within a computing node, we designed a multi-GPU mode by exploiting the separatability of the homomorphic matrix completion algorithm. To compute large-scale matrices exceeding GPU memory capacity, we designed a shard mode by exploiting large CPU memories. The multi-GPU mode achieves up to 1.84× speedups on two GPUs versus on a single GPU. For large-scale matrices, the shard mode achieves up to 174.92× speedups on a single GPU over the CPU MATLAB implementation on two CPUs, and further achieves up to 1.35× speedups when running on two GPUs using the multi-GPU mode. The proposed scheme can be employed in diverse applications.

ACKNOWLEDGMENT

A conference version [1] of this article was presented at the 2019 IEEE International Conference on High Performance Computing and Communications.

REFERENCES

[1] H. Lu, T. Zhang, and X.-Y. Liu, "High-performance homomorphic matrix completion on GPUs," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun., IEEE 17th Int. Conf. Smart City, IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Aug. 2019, pp. 1627–1634.

[2] H. Wu, X.-Y. Liu, L. Fu, and X. Wang, "Energy-efficient and robust tensor-encoder for wireless camera networks in Internet of Things," *IEEE Trans. Netw. Sci. Eng.*, vol. 6, no. 4, pp. 646–656, Oct. 2019.

[3] X.-Y. Liu, S. Aeron, V. Aggarwal, X. Wang, and M.-Y. Wu, "Adaptive sampling of RF fingerprints for fine-grained indoor localization," *IEEE Trans. Mobile Comput.*, vol. 15, no. 10, pp. 2411–2423, Oct. 2016.

[4] S. Rallapalli, L. Qiu, Y. Zhang, and Y.-C. Chen, "Exploiting temporal stability and low-rank structure for localization in mobile networks," in *Proc. 16th Annu. Int. Conf. Mobile Comput. Netw.*, 2010, pp. 161–172.

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

[6] X.-Y. Liu, Y. Zhu, L. Kong, C. Liu, Y. Gu, A. V. Vasilakos, and M.-Y. Wu, "CDC: Compressive data collection for wireless sensor networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 8, pp. 2188–2197, Aug. 2014.

[7] L. Kong, M. Xia, X.-Y. Liu, G. Chen, Y. Gu, M.-Y. Wu, and X. Liu, "Data loss and reconstruction in wireless sensor networks," *IEEE Trans. Parall. Distr. Syst.*, vol. 25, no. 11, pp. 2818–2828, Nov. 2014.

[8] X.-Y. Liu and X. Wang, "LS-decomposition for robust recovery of sensory big data," *IEEE Trans. Big Data*, vol. 4, no. 4, pp. 542–555, Dec. 2018.

[9] S. Liao, X.-Y. Liu, F. Qian, M. Yin, and G.-M. Hu, "Tensor super-resolution for seismic data," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2019, pp. 8598–8602.

[10] M. L. Stein, *Interpolation of Spatial Data: Some Theory for Kriging*. New York, NY, USA: Springer, 2012.

[11] L. Kong, M. Xia, X.-Y. Liu, M.-Y. Wu, and X. Liu, "Data loss and reconstruction in sensor networks," in *Proc. IEEE INFOCOM*, Apr. 2013, pp. 1654–1662.

[12] B. Recht, "A simpler approach to matrix completion," *J. Mach. Learn. Res.*, vol. 12, pp. 3413–3430, Jan. 2011.

[13] E. Candès and B. Recht, "Exact matrix completion via convex optimization," *Commun. ACM*, vol. 55, no. 6, p. 111, Jun. 2012.

[14] P. Jain, P. Netrapalli, and S. Sanghavi, "Low-rank matrix completion using alternating minimization," in *Proc. 45th Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2013, pp. 665–674.

[15] C.-J. Hsieh, N. Natarajan, and I. S. Dhillon, "PU learning for matrix completion," in *Proc. Int. Conf. Mach. Learn.*, vol. 2015, pp. 2445–2453.

[16] W. Li, L. Zhao, Z. Lin, D. Xu, and D. Lu, "Non-local image inpainting using low-rank matrix completion," *Comput. Graph. Forum*, vol. 34, no. 6, pp. 111–122, Sep. 2015.

[17] R. Cabral, F. D. L. Torre, J. P. Costeira, and A. Bernardino, "Matrix completion for weakly-supervised multi-label image classification," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 37, no. 1, pp. 121–135, Jan. 2015.

[18] E. Adeli-Mosabbeb and M. Fathy, "Non-negative matrix completion for action detection," *Image Vis. Comput.*, vol. 39, pp. 38–51, Jul. 2015.

[19] L. Kong, L. He, X.-Y. Liu, Y. Gu, M.-Y. Wu, and X. Liu, "Privacy-preserving compressive sensing for crowdsensing based trajectory recovery," in *Proc. IEEE 35th Int. Conf. Distrib. Comput. Syst.*, Jun. 2015, pp. 31–40.

[20] S. Lohr, "Netflix cancels contest after concerns are raised about privacy," New York Times, Mar. 13, 2010, p. B3.

[21] Nvidia Corporation. (2019). *NVIDIA CUDA SDK 10.1, NVIDIA CUDA Software Download*. [Online]. Available: https://developer.nvidia.com/cuda-downloads

[22] A. Abdelfattah, A. Haidar, S. Tomov, and J. Dongarra, "Performance, design, and autotuning of batched GEMM for GPUs," in *Proc. Int. Conf. High Perform. Comput.* Cham, Switzerland: Springer, 2016, pp. 21–38.

[23] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proc. 41st Annu. ACM Symp. Symp. Theory Comput. (STOC)*, 2009, pp. 169–178.

[24] M. Van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully homomorphic encryption over the integers," in *Proc. Annu. Int. Conf. Theory Appl. Cryptograph. Techn.* Berlin, Germany: Springer, 2010, pp. 24–43.

[25] B. Chor, O. Goldreich, E. Kushilevitz, and M. Sudan, "Private information retrieval," in *Proc. IEEE 36th Annu. Found. Comput. Sci.*, 1995, pp. 41–50.

[26] H. Sun and S. A. Jafar, "The capacity of private information retrieval," *IEEE Trans. Inf. Theory*, vol. 63, no. 7, pp. 4075–4088, Jul. 2017.

[27] Q. Song, H. Ge, J. Caverlee, and X. Hu, "Tensor completion algorithms in big data analytics," *ACM Trans. Knowl. Discov. Data*, vol. 13, no. 1, pp. 1–48, Jan. 2019.

[28] A. Shah and A. Majumdar, "Accelerating low-rank matrix completion on GPUs," in *Proc. Int. Conf. Adv. Comput., Commun. Informat. (ICACCI)*, Sep. 2014, pp. 182–187.

[29] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "CuMF_SGD: Parallelized stochastic gradient descent for matrix factorization on GPUS," in *Proc. 26th Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2017, pp. 79–92.

[30] H. Li, K. Li, J. An, and K. Li, "MSGD: A novel matrix factorization approach for large-scale collaborative filtering recommender systems on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, no. 7, pp. 1530–1544, Jul. 2018.

[31] T. Zhang, X.-Y. Liu, X. Wang, and A. Walid, "CuTensor-tubal: Efficient primitives for tubal-rank tensor learning operations on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 31, no. 3, pp. 595–610, Mar. 2020.

[32] H. Li, T. Zhang, R. Zhang, and X.-Y. Liu, "High-performance tensor decoder on GPUs for wireless camera networks in IoT," in *Proc. IEEE 21st Int. Conf. High Perform. Comput. Commun., IEEE 17th Int. Conf. Smart City, IEEE 5th Int. Conf. Data Sci. Syst. (HPCC/SmartCity/DSS)*, Aug. 2019, pp. 1619–1626.

[33] Y. Tao, Y. Deng, S. Mu, Z. Zhang, M. Zhu, L. Xiao, and L. Ruan, "GPU accelerated sparse matrix-vector multiplication and sparse matrix-transpose vector multiplication," *Concurrency Comput., Pract. Exper.*, vol. 27, no. 14, pp. 3771–3789, Sep. 2015.

[34] M. Deveci, C. Trott, and S. Rajamanickam, "Multithreaded sparse matrix-matrix multiplication for many-core and GPU architectures," *Parallel Comput.*, vol. 78, pp. 33–46, Oct. 2018.

[35] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Comput.*, vol. 36, nos. 5–6, pp. 232–240, Jun. 2010.

TAO ZHANG (Member, IEEE) received the bachelor's and master's degrees in computer science and technology from Xidian University, China, in 2001 and 2006, respectively, the Ph.D. degree in computer engineering from The University of New Mexico, USA, and the Ph.D. degree in computer science from Shanghai Jiao Tong University, China, in 2015.

He is currently a Researcher and a Lecturer with the School of Computer Engineering and Science, Shanghai University, China. His research interests include big data processing, GPU heterogeneous computing, machine learning algorithms, and applications.

HAN LU received the B.Eng. degree in computer science and technology from the Hebei University of Engineering, China, in 2017. He is currently a Graduate Student with the School of Computer Engineering and Science, Shanghai University, China.

His research interests include GPU parallel computing, tensor computing, and optimization acceleration of algorithm.

XIAO-YANG LIU (Student Member, IEEE) received the B.Eng. degree in computer science from the Huazhong University of Science and Technology, China, in 2010. He is currently pursuing the Ph.D. degree with the Department of Electrical Engineering, Columbia University. He graduated from the Ph.D. Program in the Department of Computer Science and Engineering, Shanghai Jiao Tong University, in 2017.

His research interests include tensor theory and high performance tensor computation, deep learning, optimization algorithms, and big data analysis and data privacy.