The following publication K. Xie, Y. Chen, X. Wang, G. Xie, J. Cao and J. Wen, "Accurate and Fast Recovery of Network Monitoring Data: A GPU Accelerated Matrix Completion," in IEEE/ACM Transactions on Networking, vol. 28, no. 3, pp. 958-971, June 2020 is available at https://doi.org/10.1109/TNET.2020.2976129.

Accurate and Fast Recovery of Network Monitoring Data: A GPU Accelerated Matrix Completion

Kun Xie[1,2,3], *Member, IEEE*, Yuxiang Chen[1], Xin Wang[4], *Member, IEEE*,
Gaogang Xie[5,6], *Member, IEEE*, Jiannong Cao[7], *Fellow, IEEE*, Jigang Wen[5]
[1] College of Computer Science and Electronics Engineering, Hunan University, China
[2] Cyberspace Security Research Center, Peng Cheng Laboratory, China
[3] Purple Mountain Laboratories, China
[4] Department of Electrical and Computer Engineering, State of New York University at Stony Brook
[5] Computer Network Information Center, Chinese Academy of Sciences, China
[6] School of Computer Science and Technology, University of Chinese Academy of Sciences, China
[7] Department of computing, The Hong Kong Polytechnic University, Hong Kong

***Abstract*—Gaining a full knowledge of end-to-end network performance is important for some advanced network management and services. Although it becomes increasingly critical, end-to-end network monitoring usually needs active probing of the path and the overhead will increase quadratically with the number of network nodes. To reduce the measurement overhead, matrix completion is proposed recently to predict the end-to-end network performance among all node pairs by only measuring a small set of paths. Despite its potential, applying matrix completion to recover the missing data suffers from low recovery accuracy and long recovery time.**

To address the issues, we propose MC-GPU to exploit Graphics Processing Units (GPUs) to enable parallel matrix factorization for high-speed and highly accurate Matrix Completion. To well exploit the special architecture features of GPUs for both task independent and data-independent parallel task execution, we propose several novel techniques: similar OD (origin and destination) pairs reordering taking advantage of the locality-sensitive hash (LSH) functions, balanced matrix partition, and parallel matrix completion. We implement the proposed MC-GPU on the GPU platform and evaluate the performance using real trace data. We compare the proposed MC-GPU with the state of the art matrix completion algorithms, and our results demonstrate that MC-GPU can achieve significantly faster speed with high data recovery accuracy.

I. Introduction

The monitoring of the end-to-end performance of the network path between two end nodes is essential to ensure the performance expected by many Internet applications [1]–[4]. For example, to ensure the transmission quality of real-time video conferences, we need to closely monitor the end-to-end network performance changes such as bandwidth and delay. The delay change in the unit of millisecond could impact the quality of experience of users. Other advanced network management and service applications include the content replica placement and selection, path setup, network provisioning, anomaly detection , and failure recovery. Recently, the monitoring of the end-to-end performance has seen growing interests and applications in content distribution networks (CDNs) and software defined networks (SDNs).

Despite the importance, the monitoring of the end-to-end performance of a path between an origin and a destination (OD), such as round-trip time (RTT), available bandwidth, and packet loss rate, often requires the periodic sending of probing packets along the path. The measurement overhead will increase quadratically with the number of network nodes.

To reduce the measurement overhead, sample-based network monitoring is often applied in current network systems where measurements are only taken at a small set of paths. In addition, measurement data may get lost due to severe communication and system conditions, including network congestion, node misbehavior, monitor failure, transmission of measurement information through an unreliable transport protocol.

As many network services and traffic engineering tasks require the complete network performance information and some are also highly sensitive to the missing data, an important problem in large-scale network monitoring is: ***given sampled measurement data of a small set of paths, recovering the end-to-end network performance data among all node pairs***. We term this problem as the monitoring data recovery problem.

With the rapid progress of sparse representation, matrix completion [5]–[7], a remarkable new field, has emerged very recently. According to the matrix completion theory, a low-rank matrix can be accurately reconstructed with a relatively small number of entries in the matrix. Several recent studies model the monitoring data as matrices and propose some matrix completion based algorithms [8]–[12] to recover the missing monitoring data with a small set of samples utilizing both spatial and temporal information. A popular method used to complete the matrix is through the matrix factorization, which decomposes the monitoring matrix into the product of multiple low-rank matrices. Despite its potential and big efficiency, applying matrix completion to monitoring data recovery suffers from the problems of low recovery accuracy and long recovery time.

- ***Low recovery accuracy***: In matrix completion, to successfully recover a low-rank matrix, the number of samples should be sufficient and meet some conditions. Although current matrix-based data recovery approaches present good performance when the sample ratio is high, their performance suffers when there are not enough sample data.
- ***Long recovery time***: Although several matrix completion algorithms are proposed to capture the spatial and temporal features of monitoring data for recovering the missing data elements (not measured) in a matrix, these algorithms involve an iterative and sequential process to train the parameters (i.e., factor matrices in matrix factorization methods), which would incur a high computational overhead thus delay when the matrix to recover is large for a large-scale network.

To conquer above challenges, we propose to investigate the

potential and methodologies of performing parallel processing for high-speed and high accuracy Matrix Completion over Graphics Processing Units (GPUs), termed as MC-GPU. There are some recent studies [13]–[18] on the parallel and distributed matrix completion to grid a large matrix into small sub-matrices with each iteratively executed using parallel and distributed machines. After one machine has updated a sub-factor matrix, there need communications among parallel/distributed machines or with the server center to obtain the updated factor matrices for the next iteration. With a high communication cost, these methods can not be directly applied to the GPU platform, as they cannot well exploit the special architecture features of GPU (special thread, thread block, and special memory architecture) to achieve high parallelism gain.

Facilitated by locality-sensitive hash (LSH) functions, we propose several novel techniques to take full advantage of the architecture features of GPU to provide both *task-independence* and *data-independence* parallelism for quick and highly accurate data recovery. The main techniques proposed in this paper are summarized as follows:

- ***Quick OD pair reorganization***. Reordering the matrix rows each corresponding to an OD pair can facilitate better matrix division for matrix completion. Taking advantage of the property of locality-sensitive hash functions, MC-GPU adopts an LSH table to quickly reorder the OD pairs in the network and buffers the indexes of similar OD pairs in the same hash bucket of the table.
- ***Balanced matrix partition***. Based on the LSH table, a *partition based on adjacent least load merge* mechanism is proposed to balancedly partition a large monitoring matrix into sub-matrices each containing the monitoring data of similar OD pairs. This will help bring close the OD pair correlation in the sub-matrices to more accurately recover the missing data in the sub-matrices, and facilitate uniform assignment of computational tasks into different GPU threadblocks. The later will further speed up the recovery process for a large matrix.
- ***Task independent assignment in GPU threadblocks***. To well utilize the GPU's thread/block architecture, each partitioned sub-matrix is assigned as the sub-task to execute in a threadblock, and the final recovered data can be obtained by combining the recovered data from multiple sub-matrices. All the sub-tasks assigned to execute in parallel are independent, so that our scheme can effectively avoid the large costs for communication and synchronization across threadblocks.
- ***Data independent assignment in GPU threads***. To well exploit thread resources in each block of GPU, we propose a data independent computation assignment algorithm to take advantage of the factorization structure of the sub-matrix and design a set of update rules based on the mini-batch SGD (Stochastic Gradient Descent) to train the factor matrices for more accurate data recovery.

We implement the proposed MC-GPU on the GPU platform using trace data for RTT (Harvard226 [19]) and traffic flow (GÈANT [20]) measurements. We compare the proposed algorithms with the state of art matrix completion algorithms, and our results demonstrate that MC-GPU can achieve significantly quicker recovery speed with a high data recovery accuracy.

The rest of this paper is organized as follows. We introduce the related work in Section II. We present the preliminary, problem formulation, and solution overview in Section III. The proposed matrix partition mechanism is presented in Section IV, and the parallel matrix completion algorithm is presented in Section V. Finally, we evaluate the performance of the proposed MC-GPU through extensive experiments in Section VI, and conclude the work in Section VII.

II. Related Work

As a typical sparse representation technique, matrix completion attracts lots of research interests recently. Some results in matrix completion have demonstrated that if the underlying data-matrix is low rank, then the missing entries can be inferred from a small fraction of the entries [5]–[7], [21]. According to [22], existing matrix completion algorithms can be classified into two categories: convex programming based and non-convex programming based.

To recover low-rank matrix, most matrix completion techniques [3], [23]–[33] are based upon convex relaxation with nuclear norm constraint or regularization to enforce the low rank structure. Nevertheless, solving these convex optimization problems can be computationally prohibitive in high dimensional regimes with large matrix sizes [34].

To scale to larger data sets, a variety of techniques have been proposed to factor a large scale matrix into smaller ones using incremental, non-convex heuristics [22], [35]–[37]. In particular, to complete a low-rank matrix M with the dimension $m \times n$, non-convex heuristics re-parameterize M in the optimization problem as UV^T with $U \in R^{m\times k}$ and $V \in R^{n\times k}$, and optimize over U and V. U and V are low-rank factor matrices each with the rank k. Such a re-parametrization automatically enforces the low rank structure and leads to low computational cost per iteration. Non-convex methods have been quite successful on real problems, such as the popular Netflix Prize problem.

As the size of the network monitoring matrix is usually large, to make our proposed technique scalable to a large-scale network environment, the missing data are recovered based on non-convex solution. Specially, the missing data recovery problem is modeled as a matrix factorization problem whose aim is to approximate the incomplete monitoring matrix X by AB, where A and B are rank-k factor matrices. As discussed in the introduction, despite its potentially big efficiency, applying a matrix completion technique to the recovery of monitoring data suffers from the problems of ***low recovery accuracy*** and ***long recovery time***.

IDES [38] and Dmfsgd [39] are two RTT prediction algorithms based on matrix factorization. IDES factorizes a small but full inter-landmark distance matrix, at a so-called information server, by using singular value decomposition (SVD). IDES is a Landmark-based system which suffers from several drawbacks, including single-point failures, landmark overloads, and potential security problems. Our techniques can infer the missing data without the need of Landmark. Dmfsgd is a distributed matrix factorization algorithm which only requires each distributed node to collect and process local measurements. Our work aims to design an algorithm that can fully exploit the GPU parallelism to speed up the data recovery speed. It requires independent and load balanced task assignment. The requirement for an algorithm to support parallel execution is different from that for a distributed algorithm.

To provide accurate and high-speed matrix factorization for missing monitoring data recovery, facilitated by locality-sensitive hash (LSH) functions, this paper proposes a novel partition-based Matrix Completion algorithm to enable efficient parallel computations over the GPU platform, named MC-GPU. The family of LSH functions [40]–[45] exhibits the locality-aware property. Under LSH functions, close items will collide with each other at a probability higher than the distant ones. LSH has been successfully applied in approximate queries of vector space and semantic access. To the

best of our knowledge, our paper is the first to exploit LSH table to reorder the OD pairs in the network field. Moreover, based on LSH table, we further design an algorithm to partition a large matrix into sub-matrices with each sub-matrix consisting of OD pairs with closer correlation to facilitate more accurate missing data recovery in the sub-matrix.

Although many matrix completion approaches have been proposed, originally developed for centralized in-memory computation on a single machine, these algorithms involve an iterative and sequential process to train the parameters, and are thus difficult to be executed in parallel under advanced architectures such as GPU, multi-core CPU or distributed clusters. Recently, a few studies investigate the parallel and distributed matrix completion. For example, a lock-free approach called HogWild is investigated in [46], in which HogWild randomly selects a subset of instances and applies SGD update rules in all available threads simultaneously without synchronization between threads. However, as the updates for the instances on the same row or the same column of the large matrix involve the same variables, units running in parallel could overwrite each others' results. To address the issue, [13] and [14] propose distributed SGD (DSGD) that grids the large matrix into blocks and updates a set of independent blocks in parallel at the same time. As follow on work, several parallel and distributed matrix completion techniques [15]–[18] are proposed recently. Under gridding-based approaches, different blocks may use the same sub-row space and sub-column space. Therefore, after a group of parallel/distributed machines have updated the sub-factor matrices in parallel, the parallel/distributed machines have to communicate with each other or the server center to obtain the updated factor matrices for its next iteration, which incurs a high communication cost.

For more efficient matrix completion, we would like to explore new opportunities to inexpensively shift the computing needs from CPUs to Graphics Processing Units (GPUs) for high-speed missing data recovery. As GPU has special architecture features (special thread, thread block architecture), high communication cost in gridding-based parallel matrix completion can not well exploit GPU architecture features to achieve high parallelism gain. Facilitated by locality-sensitive hash (LSH) functions, in MC-GPU, we propose a partition-based matrix completion scheme which can provide both task independent and data independent parallelism with synchronization free parallel computation scheduling and low communication cost.

III. Preliminary and Overview

In this section, we first introduce the preliminary on GPU architecture, then present the problem formulation and the solution overview.

A. GPU architecture

In this section, we use NVIDIA GPUs as an example to illustrate GPU architecture [47], [48]. Although AMD GPU uses different terms to describe its GPU architecture, the whole architecture is similar. There are three main components of a typical GPU architecture: threads, threadblocks (also called blocks) and memory hierarchy.

CUDA (Compute Unified Device Architecture) was released by NVIDIA to simplify the general-purpose programming of their GPUs. Threads in a CUDA kernel are organized in a two-level hierarchy, as shown in Fig. 1(a). At the top level, a kernel consists of a 1D/2D/3D grid of threadblocks where each thread block internally contains multiple threads organized in either 1, 2 or 3 dimensions. One grid corresponds to one kernel call. One or more thread blocks can be executed by a single computing unit called SM (Streaming Multiprocessor), but one block can not be executed across different SMs. In Fig.1(b), two blocks are executed in the Multiprocessor 0. An SM is responsible for all its internal thread management and can switch threads without the scheduling overhead. Once a threadblock is dispatched to an SM, it cannot be preempted and occupies that SM until all threads of the threadblock terminate.

A CUDA program can use various types of memory. Fig. 1(b) shows the GPU memory architecture with the features summarized in Table. 1.

Global memory is normally applied for copying the input and output data to and from the main memory of the host. Although the space is large, it is off-chip and slow. The use of this memory should be restricted to fully-coalesced access cases. Serialized access to the global memory is very expensive given that its latency is two orders of magnitude greater than the accesses to an on-chip memory.

The local variables used are independent for each thread, and their values are preferentially stored in registers for fast access. When registers are not enough to record all local variables, data are stored in the local memory. Both local memory and registers are visible to only individual threads and cannot be programmed directly by the programmer.

Constant memory is a limited off-chip memory space where a small number of constants (or non-modifiable input data) can be placed and used by all threads. *Texture memory* is a subset of global memory that is accessed via the texture fetch unit. This region of global memory must be bound through runtime functions before it can be used by a kernel. Different from the global memory, the constant and texture memory can only be read, but not written by a function.

Shared memory is a small on-chip software-managed memory on the NVIDIA platform. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. The latency of accessing a shared memory (assuming no bank conflicts) is fast and similar to a register access, thus the effective use of the shared memory can have a large impact on the performance.

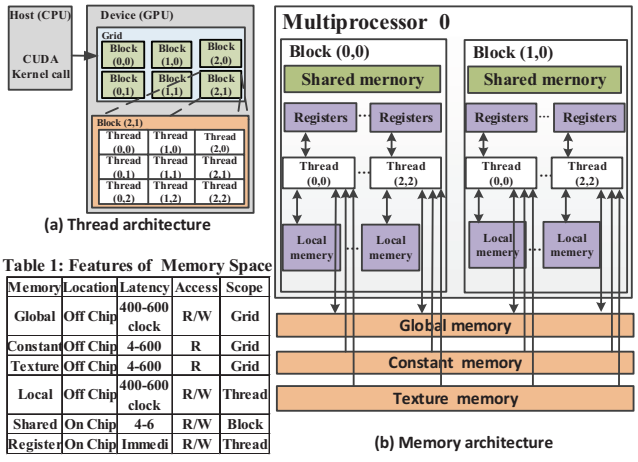


Table 1: Features of Memory Space

Memory	Location	Latency	Access	Scope
Global	Off Chip	400-600 clock	R/W	Grid
Constant	Off Chip	4-600	R	Grid
Texture	Off Chip	4-600	R	Grid
Local	Off Chip	400-600 clock	R/W	Thread
Shared	On Chip	4-6	R/W	Block
Register	On Chip	Immedi	R/W	Thread

Fig. 1. GPU architecture

As there is no explicit support for inter-block communication on the GPU, when needed, such a communication is performed only via the global memory and requires a barrier synchronization to complete, which is (inefficiently) implemented via the host CPU. Therefore, GPUs typically map well only to data or task parallel

applications whose execution requires minimal or even no inter-block communication.

We aim to design a parallel matrix completion algorithm that can well utilize the characteristics of the GPU architecture to maximize the parallelization gain for fast monitoring data recovery.

B. Empirical study with real monitoring data

For a network consisting of N nodes, there are $M = N \times N$ OD pairs. We define a monitoring data matrix, $P_{M\times T}$, to hold the end-to-end performance monitoring data, with the (ij)-th entry, p_{ij}, representing the performance monitoring data of OD pair i at the timeslot j. In the matrix, a row corresponds to an OD pair and a column corresponds to a timeslot. T denotes the total number of time slots captured in the matrix.

The low-rank feature is the prerequisite for using the matrix completion. In this subsection, we validate that the monitoring matrix data have the low rank feature. To examine whether the monitoring matrix has a good low-rank structure, we first apply the singular value decomposition (SVD). A monitoring matrix $P_{M\times T}$ can be decomposed as:

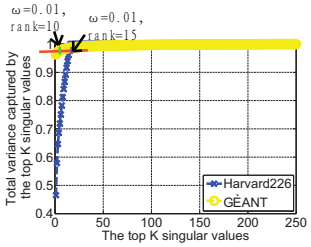


Fig. 2. $g(k)$:the fraction of the total variance captured by the top k singular values

$$P = U\Sigma V^T \tag{1}$$

where $U = [u_1, \cdots, u_M]$ is an $M \times M$ unitary matrix, $V = [v_1, \cdots, v_T]$ is a $T \times T$ unitary matrix, and Σ is an $M \times T$ diagonal matrix with the diagonal elements (i.e. the singular values) organized in the decreasing order (i.e. $\Sigma = diag(\sigma_1, \sigma_2, \cdots, \sigma_r, 0, \cdots, 0)$). The rank of a matrix P, denoted by r, is equal to the number of its non-zero singular values. In this paper, we call this rank definition as "precise rank". A matrix is low-rank if its $r \ll \min\{M, T\}$.

Although the definition of the precise rank is of high theoretical interest, it is not realistic to use this definition for the practical data. The calculation of the precise rank of the matrix is an ill-posed problem in a practical environment because arbitrary small perturbations of matrix elements may change the rank [49]. Instead of performing the matrix completion based on the precise rank, this paper adopts the approximate rank [49]. We say that X has ω-rank k if

$$\inf\{\|X - Y\| : Y \quad has \quad rank \quad k\} \leq \omega. \tag{2}$$

A theorem proof provided by Eckart and Young [50] shows that the error in approximating a matrix X by X_k can be written as: $\|X - X_k\|_F^2 \leq \|X - Y\|_F^2$ where Y is any matrix with rank k, X_k is the rank-k truncated SVD of matrix X, that is $X_k = \sum_{i=1}^{k} \sigma_i u_i v_i^T$.

The ratio $g(k) = \sum_{i=1}^{k} \sigma_i^2 / \sum_{i=1}^{r} \sigma_i^2$ indicates what fraction of the total variance (Frobenius norm) in X is explained by the rank k approximation of X, i.e., X_k. According to PCA (Principal components analysis), if a matrix has low-rank, its top k singular values occupy the total variance, that is, $\sum_{i=1}^{k} \sigma_i^2 \approx \sum_{i=1}^{r} \sigma_i^2$.

Fig.2 plots the fraction of the total variance captured by the top k singular values for different trace data (Harvard226 [19] and GÈANT [20]). Harvard226 records RTT data between OD pairs, while GÈANT records traffic flow values between OD pairs. We find that the top 20 singular values capture more than 95% variance in the real traces. These results indicate that the monitoring matrix P has a good low-rank approximation.

We design a matrix completion algorithm based on the matrix factorization, where a monitoring matrix is decomposed into two low-rank factor matrices. The factor matrices are trained with the sample data, and then integrated to recover the original matrix thus the missing data. As the sizes of the factor matrices are determined by the rank of the original matrix, we estimate the rank according to (2) by setting $\omega = 0.01$ to preserve 99% variance in the real traces. In Fig.2, the ω-ranks of Harvard226 and GÈANT are 15 and 10, respectively.

In practice, when we collect measurement samples and want to recover the missing data based on matrix completion, we can estimate the matrix rank based on the historical monitoring data.

C. Problem

This paper adopts matrix factorization to infer the missing entries of a matrix. Specifically, to recover the missing data, a monitoring matrix P is factored into a production of an $M \times k$ factor matrix A and a $T \times k$ factor matrix B by minimizing the loss function with k denoting the matrix rank as follows:

$$(A, B) = \arg\min_{A,B} \left\| \left(P - A \times B^T\right)_\Omega \right\|_F^2 \tag{3}$$

where $\left\| \left(P - A \times B^T\right)_\Omega \right\|_F^2$ is the loss function defined based on the Frobenius norm $\|\|_F$, Ω is the index set of the observed samples in the monitoring matrix.

Directly solving (3) may lead to an overfitting problem, where the errors corresponding to the matrix elements with data samples are very small while the errors for the inferred data items not having data for the training of matrices A and B are large. A common way to avoid the overfitting problem is through the regularization that penalizes the norms of the solutions:

$$(A, B) = \arg\min_{A,B} \left\| \left(P - A \times B^T\right)_\Omega \right\|_F^2 + \lambda \|A\|_F^2 + \lambda \|B\|_F^2 \tag{4}$$

where λ is the regularization coefficient that controls the extent of the regularization. After obtaining the factor matrices A and B, the monitoring matrix can be recovered through following calculations:

$$\hat{P} = A \times B^T \tag{5}$$

where $\hat{P}$ denotes the estimated monitoring matrix.

By using the observed samples to train the important parameters in (3) (i.e., factor matrices A and B), many past studies have proposed optimization methods to solve the problem. Despite its potentially big efficiency, current matrix completion methods face the problems of not enough sample data and scalability and can hardly provide accurate and quick matrix recovery, especially when the matrix size is large. The detailed issues are:

- ***Low data sample ratio***: As matrix P is generally sparse with very limited number of data samples to reduce the monitoring cost, the factor matrices trained can not contain enough information to capture characteristics and temporal features of OD pair, resulting in the low recovery accuracy.
- ***Scalability***: When the monitoring matrix P is large with the values of N and T very large, it will involve a high computation cost to recover the monitoring matrix and consequently result in slower processing speed. This makes it difficult to perform the matrix factorization on a traditional CPU platform.

To address the above issues, we propose a GPU-based parallel Matrix Completion scheme (MC-GPU) to take advantage of the correlations of OD pairs and the massively-parallel processing power of GPU for accurate and fast matrix completion.

D. Solution overview

To solve the problem in (4), ALS (Alternative Least Squares) and SGD (Stochastic Gradient) can be utilized. With ALS, the recovery loss $L(A,B) = \left\|\left(P - A \times B^T\right)_\Omega\right\|_F^2 + \lambda \|A\|_F^2 + \lambda \|B\|_F^2$ is minimized with the iterative solving of two sub-problems, $B \leftarrow \arg\min_B L(A,B)$ and $A \leftarrow \arg\min_A L(A,B)$ with either A or B fixed, and each sub-problem further contains multiple independent ridge regression problems. According to [51], the time complexity of one ALS iteration step is $O\left(2\left|\Omega\right| k^2 + (T+M)k^3\right)$.

As we will introduce in Section V-B, with SGD, an (m,n) entry $p_{m,n}$ from the loss summation L is randomly selected and the corresponding local gradient is applied to update A_m and B_n, where A_m and B_n denote the m-th and n-th rows of factor matrices A and B. Each local gradient calculation requests one inner product of vectors at the time complexity $O(k)$. Therefore, it involves the time complexity $O(2k)$ to train A_m and B_n corresponding to one sample in each iteration. We refer the overall time duration taken to pass over all the observed sample data as an epoch, and the complexity of one epoch is $O(2|\Omega|k)$.

Although ALS is easy to parallelize as it includes independent ridge regression problems, its computation cost is much higher than that of SGD. Thus we design our parallel matrix completion algorithm based on SGD in Section V-B. In Section VI-G, we will compare the computation time under our MC-GPU and ALS using two data traces collected from the real networks.

In order to exploit all GPU units to run the completion process, a straight forward way is to uniformly divide the data entries into multiple parts. However, the relationships among data entries are different. Our recent study [28] demonstrates that network paths starting from nearby end nodes often have overlapping path segments or go through some common network nodes. This is especially the case in the Internet core that has simple topology. As a result, the monitoring data from network measurements often have correlations. For example, the congestion at a certain link would cause higher delay for all paths that traverse this link. Therefore, the samples of similar OD pairs have higher impacts on each other and can be applied for inferring the missing ones.

In order to exploit some of these correlations, we propose MC-GPU scheme to partition the original monitoring matrix into sub-matrices with each containing similar OD pairs, and assign sub-matrices to threadblocks in the GPU platform to realize the matrix completion in parallel. MC-GPU mainly includes the following three technique components:

- An LSH table, which reorders and buffers OD pairs based on locality-sensitive hashing (LSH). The good property of the LSH guarantees that similar OD pairs are packed into the same bucket in the LSH table.
- A partition algorithm, which partitions the large monitoring matrix into multiple sub-matrices based on the LSH table. This partition reduces the matrix scale and provides the task independent parallelism to avoid inter-block communication. By putting OD pairs with higher correlation into the same sub-matrix, the partition further helps for the factored sub-matrices to better capture the useful characteristics and temporal features of OD pairs for more accurate inferring of the missing data.
- A GPU-based parallel matrix completion execution scheme, which achieves the fast matrix factorization by exploiting the resources of threads on the GPU platform.

With these strategies, our MC-GPU can provide quick and high quality missing data recovery compared with the method of directly factorizing the original large monitoring matrix. In the following two sections, we present our two key techniques on matrix partition and GPU-based parallel matrix completion.

IV. Matrix Partition Based On Similar OD Pair Grouping

To exploit the correlations among OD pairs for more accurate matrix recovery, similar OD pairs need to be grouped together. Moreover, to well utilize GPU's parallel resources, our matrix partition algorithm will sub-divide a large monitoring matrix into sub-matrices with balanced number of items and similar OD pairs in each.

The matrix partition can be achieved with a clustering algorithm. However, to handle network monitoring data with high dimensional OD pair vector, conventional clustering algorithms such as k-means [52] and gaussian mixture model (GMM) [53] would incur a high computation cost with their need of iteratively calculating the distances among items. Instead, we exploit LSH for low cost and fast clustering to facilitate the matrix partition.

Our partition algorithm has following two steps : 1) We first build a LSH table to reorder OD pairs and put together the similar OD pairs with a higher probability, and 2) We design an adjacent least load merge mechanism to facilitate balanced matrix partition, where the LSH table is adjusted that each bucket of the table contains a balanced number of similar OD pairs. Following, we will present the detailed techniques.

A. Similar OD pair grouping using LSH table

According to [45], the LSH function family is defined as follows.

Definition 1: **(LSH Function Family) [45]:** $\mathbb{H} = \{g : S \rightarrow U\}$ is called (R, cR, P_1, P_2) - sensitive for any $p, q \in S$

- If $||p,q||_s \leq R$ then $\Pr_{\mathbb{H}}[g(p) = g(q)] \geq P_1$.
- If $||p,q||_s \geq cR$ then $\Pr_{\mathbb{H}}[g(p) = g(q)] \leq P_2$.

where $\|p,q\|_s$ is the distance of elements p and q, S is the domain of elements. In this paper, OD pairs are the elements that need to be reordered according to their distances. In the LSH, $c > 1$ and $P_1 > P_2$. This indicates that LSH is locality-sensitive, with its probability of collision (by hashing two items to the same hash value) higher for nearby items.

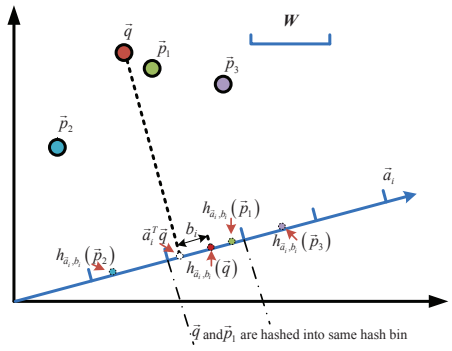


Fig. 3. Geometry result of LSH hash functions

To group similar OD pairs together, we use the row vector of the monitoring matrix as the OD pair vector and apply LSH to reorder OD pairs. As demonstrated by [45], [54], LSH function can project points that are close-by in its space to neighboring points on the line of the projection vector, as long as the elements of the projection vector are taken following the Gaussian distribution. We would like to exploit LSH to quickly cluster OD pairs into different buckets to facilitate the sub-division of matrix for parallel

matrix completion, with each sub-matrix containing items of higher correlation to increase the recovery accuracy.

Specially, given an OD pair $\vec{\mu}_j \in R^T\,(1 \le j \le M)$, we define the following LSH hash function $h_{\vec{a},b} : R^T \to R$ to map an OD pair into a bucket:

$$h_{\vec{a},b}\left(\vec{\mu}\right) = \left\lfloor \frac{\vec{a}^T\vec{\mu} + b}{W} \right\rfloor \tag{6}$$

where $\vec{a}$ is a T-dimensional random vector with each component chosen independently from a Gaussian distribution $\mathcal{N}\left(0,1\right)$, W is the width of a bucket, and b is a real number randomly selected from the interval $[0, W)$.

Fig. 3 further illustrates the geometry result of the adopted LSH functions in a 2D space. Given a vector $\vec{a}$ and the query OD pair $\vec{q}$, $\vec{a}^T\vec{q}$ is the dot product of the two, which projects $\vec{q}$ onto the vector $\vec{a}$ (also called as projection line). The value b in $h_{\vec{a},b}\left(\vec{q}\right)$ is applied to further shift the projected point over a distance b. The vector line is divided into multiple buckets with the length of each being W.

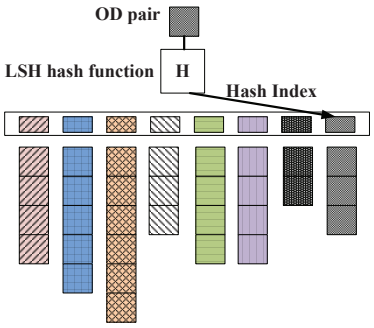


Fig. 4. OD pair reorder in LSH table

In the domain of hash function, given two OD pairs $\vec{p}$ and $\vec{q}$, if the pairs are mapped to the same bucket, there is a collision in hashing. We analyze the properties of the hash table through the calculation of the collision probability.

According to [45], [54], we know that Gaussian distribution $\mathcal{N}\left(0,1\right)$ is 2-stable distribution. Thus the distribution of $\vec{a}\cdot\vec{p} - \vec{a}\cdot\vec{q}$ follows that of dX, where $d = \|\vec{p} - \vec{q}\|$ is the distance between OD pairs $\vec{p}$ and $\vec{q}$ and X is a random variable drawn from $\mathcal{N}\left(0,1\right)$ with the density $f\left(x\right) = \frac{1}{\sqrt{2\pi}}e^{-x^2/2}$.

Since b is a real number randomly selected from the interval $[0, W)$, the probability that the OD pairs $\vec{p}$ and $\vec{q}$ hashed into the same bucket (i.e., collide in LSH table) is computed as follows [45], [54]:

$$Pr_{\vec{a},b}\left[h_{\vec{a},b}(\vec{p}) = h_{\vec{a},b}(\vec{q})\right] = \int_{t=0}^{W} \tfrac{1}{d} f(\tfrac{t}{d}) \left(1 - \tfrac{t}{W}\right) dt \tag{7}$$

where $\frac{1}{d}f(\frac{t}{d})$ is the probability that the distance of the projected points equal to t, that is, $\vec{a}\cdot\vec{p} - \vec{a}\cdot\vec{q} = t$. The collision probability (7) depends only on the distance d. As this probability increases when distance reduces, our LSH function in Eq(6) has a good property of mapping similar OD pairs with short distance into the same bucket.

To facilitate the matrix partitioning, in this paper, we use the LSH table to reorder the OD pairs. We need a key to find the hashed location of an OD pair. Instead of directly using the row vector corresponding to an OD pair, we use the normalized row vector. Given an OD pair $\vec{\mu}_j \in R^T\,(1 \le j \le M)$, it can be normalized as

$$\vec{\mu}_j^* = \frac{\vec{\mu}_j}{\|\vec{\mu}_j\|_2}, \tag{8}$$

where $\|\vec{\mu}_j\|_2$ is an L_2-norm of the $\vec{\mu}_j$.

Taking two OD pair vectors ($\vec{r}_1 = [2,3,4]$ and $\vec{r}_2 = [4,6,8]$) as an example. The values of these two vectors are obviously different, but both OD pairs have the same traffic access pattern and thus the similar traffic trend. The traffic from the two OD pairs must be closely related, and the OD pairs should be grouped into the same hash bucket. The normalized vectors of the two are the same $[2/\sqrt{29}, 3/\sqrt{29}, 4/\sqrt{29}]$ and can be hashed into the same table bucket.

According to Definition 1, with one LSH hash function, two OD pairs $\vec{p}$ and $\vec{q}$ have up to P_2 probability of being hashed into the same bucket when their distance $\|\vec{p} - \vec{q}\| > cR$. To reduce the probability of hashing distant OD pairs into the same hash bucket, we could apply multiple hash functions. Given an LSH hash function $h_{\vec{a}_i,b_i}$ and a normalized OD vector $\vec{u}_j^*$, we denote the hash index as $h_{\vec{a}_i,b_i}(\vec{u}_j^*)$. With n hash functions, we have the hash index of $\{h_{\vec{a}_1,b_1}(\vec{u}_j^*), h_{\vec{a}_2,b_2}(\vec{u}_j^*), ..., h_{\vec{a}_n,b_n}(\vec{u}_j^*)\}$. Straight forwardly, this would require n hash tables to store the OD vectors. Given an OD pair, to search for its close-by neighbor, n corresponding hash buckets in the n hash table should be checked to find the one that can be found in all the tables. Obviously, this solution requires $O(n)$ to store the hash index and also $O(n)$ to search for a close-by neighbor. Without a physical bucket to hold all items closely-related, this storage and search scheme is difficult to support clustering.

To reduce the storage and query cost, and more importantly to support our adjacent least load merge algorithm in Algorithm 1 for load balanced partitioning, we will instead use a super index which involves n hash functions:

$$H\left(\vec{\mu}_j^*\right) = \sum\nolimits_{i=1}^{n} r_i h_{\vec{a}_i,b_i}\left(\vec{\mu}_j^*\right) \tag{9}$$

where r_i is a random integer. This design is inspired by MIT's E2LSH [55]. However, a mod operation is applied to the sum of hash values in E2LSH, which may change the sequence of items. In our design, we apply the hash operation to facilitate the grouping of items with higher similarity. We use multiple hash functions to improve the hashing accuracy and reduce the chance of hash collision, where the collision is resulted when two distant OD vectors are hashed to the same bucket. So our index is calculated without the mod operation. In Section IV-C1, we will demonstrate that with our hash index design, the items in adjacent buckets are more similar than those from far away buckets, which provides the base for our merging operation.

With this super index, two OD vectors $\vec{p}$ and $\vec{q}$ have the same super index under Equ. (9) only if all n hash indexes are the same, that is, $h_{\vec{a}_i,b_i}(\vec{p}) = h_{\vec{a}_i,b_i}(\vec{q})$ for $(1 \le i \le n)$. According to Definition 1, using a single hash function, the probability for two distant OD vectors with the distance $\|\vec{p} - \vec{q}\| > cR$ to have the same hash index is not larger than P_2. Thus, the probability in Eq.(9) is not larger than $(P_2)^n$. As $0 < P_2 < 1$, we have $(P_2)^n < P_2$. Thus the use of a group of hash functions helps to reduce the probability of hashing uncorrelated OD pairs to the same bucket to create the collision.

The use of multiple hash functions to find the index allows users to choose a proper number of hash functions (i.e., n) to meet different application needs. There is a tradeoff in determining the value of n. A larger n can reduce the probability for distant vectors to be hashed into the same hash bucket (i.e., $(P_2)^n$), but will incur a large a computation cost. In Section VI-C1, we use experiment to set a proper n in our scenario.

After applying the LSH-based function to the OD pairs, the OD pairs in the network are reordered and buffered in the LSH table. An index calculated from (9) corresponds a bucket in the LSH table. Rather than physically moving the measurement data for OD pairs, each bucket in the hash table only keeps the chain of IDs of correlated OD pairs, as shown in Fig.4. Only requiring the hash

calculations, our LSH table reorders and buffers OD pairs in a fast and effective way.

B. Pre-interpolate the matrix

Although applying LSH is a promising way to group similar OD pairs, data of a row which corresponds to an OD pair in the matrix may be a sparse vector with some entries missing and can not be used as an input for LSH.

Our goal is to quickly group data with higher correlation into sub-matrices for parallel processing. Although k-NN and SVD are two typical methods to interpolate the missing monitoring matrix, these two methods may introduce high computation cost and not suitable for this paper. As shown in [2], network monitoring data have the feature of temporal stability and usually change slowly over time. This makes the use of a simple and fast pre-interpolation along the temporal dimension reasonable, and it also provides some accuracy guarantee. More specifically, for a missing entry, we take the interpolated value to be the average of the sample entries close to the missing one in the temporal domain, as illustrated in Fig.5. A pre-interpolation only facilitates the use of LSH for OD pair mapping. The missing entries will be more accurately recovered through iterative calculations in GPUs, taking advantage of both the spatial and temporal features hidden in the monitoring data. The initial values interpolated do not impact the final accuracy of the matrix recovery, as demonstrated in our performance studies later.

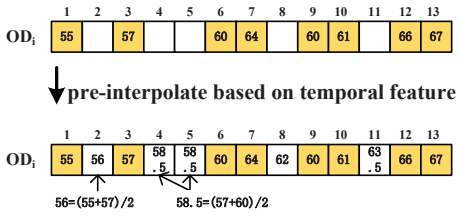


Fig. 5. pre-interpolation method

C. Large matrix partition

Based on the OD pair reordering in Section IV-A, to increase the missing data recovery performance, we propose to partition the original large-scale matrix into sub-matrices thus dividing a large matrix completion task into multiple small sub-tasks. The sub-tasks are then assigned to threadblocks in GPU for parallel execution. The general characteristics and benefits of partition-based matrix completion are:

- It allows for a quick and high quality missing data recovery because the scale of the sub-matrix is significantly reduced and similar OD pairs with close correlations are contained in the same sub-matrix.
- The partition-based matrix completion provides the possibility of task-independent parallelism to well exploit the resources of multiple threadblocks in GPU.
- As each sub-matrix is executed independently in different threadblcoks, the large monitoring matrix can be recovered by combining data recovered from sub-matrices using different threadblocks. As each sub-task is independent, such combining does not need the communication and data synchronization across different threadblocks. Therefore, the low cost parallel execution can be achieved at the GPU platform.

LSH table is applied to reorder and buffer similar OD pairs in close-by positions. It is straight-forward to let a hash bucket which contains similar OD pairs to correspond to one sub-matrix of the original large-scale matrix. Although feasible, this matrix partition method may not be able to fully exploit the GPU resources to maximize the performance. If the number of OD pairs in different buckets are different, it will result in different computation load assigned to different threadblocks. As the large matrix is recovered by combining sub-matrices recovered in different threadblocks, the overall matrix recovering time depends on the overloaded threadblocks. ***To achieve the maximum parallel processing gain in GPU platform, threadblocks should be assigned with tasks of balanced load, that is, a large matrix should be partitioned into sub-matrices with similar load.***

Following, we first present the two good properties hidden in our LSH table, then design an adjacent merge algorithm to well utilize the properties for load-balanced matrix partition.

1) Properties of LSH table

Eq(7) implies that our LSH table has a property: proximate items are hashed into the same bucket. In this section, we will validate that our LSH table has another property: the items in adjacent buckets are more similar than those from far away buckets.

Let $\vec{v}_i$ denote the centroid vector of hash bucket i. The centroid of a LSH bucket is calculated as the average of the vectors that are hashed into the bucket. The distance between two buckets can be calculated as

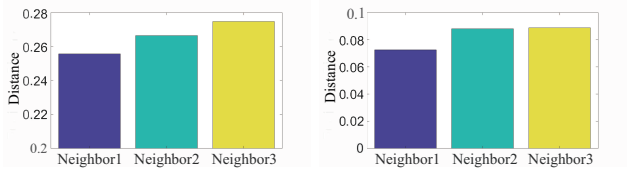
$$d_{\vec{v}_i \vec{v}_j} = \|\vec{v_i} - \vec{v_j}\|_2 \tag{10}$$

where $\vec{v}_i$ and $\vec{v}_j$ are the centroid vectors of bucket i and j.

We build two LSH tables by applying the hash function in Eq(9) with $n = 10$ to reorder and buffer the OD pairs from two public data sets Harvard226 [19] and GÈANT [20]. For every bucket in LSH table, we calculate the following three types of distances and draw the average results in Fig.6.

- Neighbor1: the distance between one-hop neighboring buckets.
- Neighbor2: the distance between two-hop neighboring buckets.
- Neighbor3: the distance between three-hop neighboring buckets.

Obviously, in Fig.6, the distance under Neighbor1 is smaller than that under Neighbor2 and Neighbor3. Therefore, one-hop adjacent buckets are more similar than with other remote buckets.



(a) Harvard226 (b) GÈANT

Fig. 6. Cosine similarity of neighbor bucket.

2) Adjacent merge algorithm

Based on the properties of our LSH table, we propose Algorithm 1 to facilitate the partition of a large matrix into sub-matrices with the load balanced. For a lightly loaded bucket, Algorithm 1 will enlarge the bucket by merging the bucket with its least-loaded one-hop adjacent bucket. The algorithm iteratively executes the merge operations until the number of buckets is equal to the number of the threadblocks in the GPU system.

We denote the length of table as the number of buckets in the hash table, denoted by w. According to the LSH table, we know that if the table size w in the LSH function becomes larger, the hash table will have a larger number of buckets with each bucket

buffering fewer OD pairs with closer correlation. As a result, w should not be too large. Moreover, to exploit the LSH table for the load-balanced partition of the matrix, the parameter w is usually set to satisfy $w > s$ where s is the number of threadblocks in the GPU. In this paper, we set $w = 30$.

Algorithm 1 iteratively adjusts the least load bucket by merging the bucket with its least load one-hop adjacent bucket until $w = s$ where s is the number of threadblocks. Finally, according to the adjusted LSH table, the large matrix is partitioned into s sub-matrices.

Algorithm 1 Adjacent least-load merge

1: **while** $w \neq s$ **do**
2: Update the least load bucket by merging the bucket with its least-load one-hop adjacent bucket.
3: Update the hash table with $w = w - 1$
4: **end while**

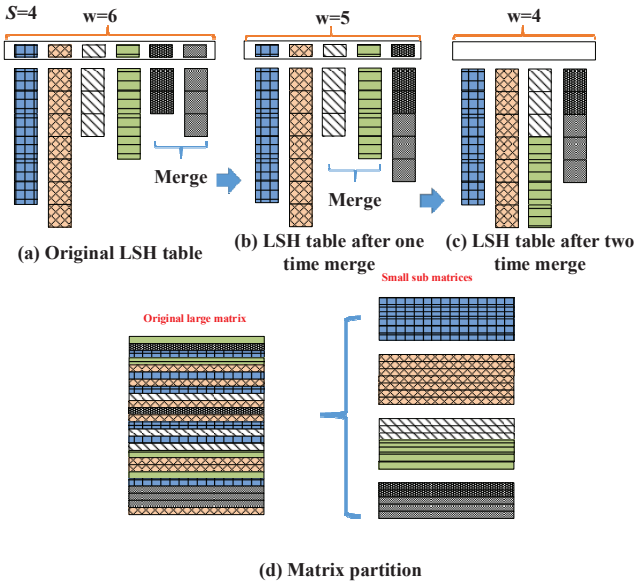


Fig. 7. Partition based on adjacent least load merge

Fig.7 presents an example to illustrate the merge procedure of line 1-4 in Algorithm 1. There are totally $s = 4$ threadblocks in the GPU and $w = 6$ buckets in the LSH table. In Fig.7(a), originally, all the OD pairs are hashed into $w = 6$ buckets in the hash table. According to Algorithm 1, the fifth bucket has the least load with only 2 OD pairs, we merge this bucket with its least-load adjacent bucket (the 6th bucket) and form the bucket distribution in Fig.7(b). After further merging the 3rd and 4th bucket in the LSH table shown in Fig.7(b), we have Fig.7(c) with the number of buckets w exactly equal to s. Finally, Fig.7(d) shows the matrix partition result according to the updated hash table (in Fig.7(c)).

V. Parallel Matrix Completion

As GPU has gird-threadblock-thread architecture and each threadblock has multiple threads, to maximize the parallelism gain brought by GPU, the sub-task (corresponding to the sub-matrix) is further partitioned into data independent computations to run in parallel using threads in the threadblock.

In this section, we first introduce the basic method we take to solve the parallel matrix completion problem, based on which, we outline our methodologies in achieving data independent parallel calculation in threadblock in a GPU environment.

A. Matrix factorization for the sub-matrix

We consider a GPU consisting of s threadblocks. According to the partition mechanism, the large monitoring matrix will be partitioned into s sub-matrices. Let $P^{(i)} \in R^{M_i \times T}$ denote the i-th sub-matrix with $1 \leq i \leq s$. $P^{(i)}$ includes the monitoring data of M_i OD pairs over T timeslots. To recover the missing data in the sub-matrix, the sub-problem to be solved at the i-th threadblock is

$$(A^{(i)},B^{(i)}) = \underset{A^{(i)},B^{(i)}}{\arg\min} \left\| \left(P^{(i)} - A^{(i)} \times (B^{(i)})^T\right)_{\Omega^{(i)}} \right\|_F^2 + \lambda \left\|A^{(i)}\right\|_F^2 + \lambda \left\|B^{(i)}\right\|_F^2 \tag{11}$$

where $\left\| \left(P^{(i)} - A^{(i)} \times (B^{(i)})^T\right)_{\Omega_i} \right\|_F^2$ is the loss function defined based on the Frobenius norm $\|\|_F$, $\Omega^{(i)}$ is the index set of samples in the sub-matrix $P^{(i)}$, $A^{(i)} \in R^{M_i \times k}$ and $B^{(i)} \in R^{T \times k}$ are the factor matrices for the sub-problem.

ALS (Alternative Least Squares) can be utilized to solve problem in (11). As ALS for matrix completion is usually computation intensive, to reduce the computation cost, we design our matrix completion algorithm based on SGD.

B. SGD

Let $A_m^{(i)}, B_n^{(i)}$ denote the m-th row, n-th row of the factor matrices $A^{(i)}$ and $B^{(i)}$. Based on the definition of the Frobenius norm $\|\|_F$, we can rewrite the loss function as a sum of local losses as

$$\begin{aligned} &\left\|(P^{(i)} - A^{(i)} \times (B^{(i)})^T)_{\Omega^{(i)}}\right\|_F^2 + \lambda \left\|A^{(i)}\right\|_F^2 + \lambda \left\|B^{(i)}\right\|_F^2 \\ &= \sum_{(m,n)\in\Omega^{(i)}} \left(\left(P_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)^2 + \frac{\lambda}{N_m^{(i)}} A_m^{(i)} \left(A_m^{(i)}\right)^T + \frac{\lambda}{N_n^{(i)}} B_n^{(i)} \left(B_n^{(i)}\right)^T \right) \end{aligned} \tag{12}$$

where $N_m^{(i)}$ and $N_n^{(i)}$ denote the number of entries on the m-th row and n-th column in the $P^{(i)}$. $L_{m,n}^{(i)}\left(A_m^{(i)}, B_n^{(i)}\right) = \left(\left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)^2 + \frac{\lambda}{N_m^{(i)}} A_m^{(i)} \left(A_m^{(i)}\right)^T + \frac{\lambda}{N_n^{(i)}} B_n^{(i)} \left(B_n^{(i)}\right)^T \right)$ is the local loss over the sample $P_{m,n}^{(i)}$.

Instead of exploiting the gradient method to solve the problem in (11), we adopt the stochastic gradient descent (SGD) to solve the matrix factorization problem. The basic idea of SGD is that, it randomly selects an (m, n) entry from the summation and calculates the corresponding local gradient $\frac{\partial}{\partial A_m^{(i)}} L_{m,n}\left(A_m^{(i)}, B_n^{(i)}\right)$. After obtaining the local gradient, the factor matrices are updated by the following rules iteratively

$$A_m^{(i)} \leftarrow A_m^{(i)} - \varepsilon \frac{\partial}{\partial A_m^{(i)}} L_{m,n}\left(A_m^{(i)}, B_n^{(i)}\right) \tag{13}$$

$$B_n^{(i)} \leftarrow B_n^{(i)} - \varepsilon \frac{\partial}{\partial B_n^{(i)}} L_{m,n}\left(A_m^{(i)}, B_n^{(i)}\right) \tag{14}$$

where ε is the learning rate.

The overall procedure of SGD is to iteratively select an instance $p_{m,n}^{(i)}$ and apply the update rules (13) and (14). After obtaining the optimal factor matrix to minimize the loss function, the sub-matrix can be recovered through the following calculations:

$$\hat{P^{(i)}} = A^{(i)} \times (B^{(i)})^T \tag{15}$$

where $\hat{P^{(i)}}$ denotes the estimated sub-matrix.

C. mini-batch SGD

The SGD algorithm in (13) and (14) is sensitive to the learning rate ε. Too large an ε results in a large step of update and may lead to the solution overflow [56], whereas too small an ε will result in a slow convergence. Enlightened by [39], we can alleviate the sensitivity problem by using more training samples at the same time. We propose to design an algorithm for parallel matrix completion based on mini-batch SGD in this work.

As only the samples on the m-th row and n-th column in $P^{(i)}$ impact A_m and B_n, to utilize multiple samples in one iteration step as well as facilitate the parallel SGD execution on GPU-platform (described in Section V-D), we define a row loss function and a column loss function of the entry (m,n):

$$l_m^{(i)} = \sum_{n:(m,n)\in\Omega^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)^2 + \lambda A_m^{(i)}\left(A_m^{(i)}\right)^T \quad (16)$$

$$l_n^{(i)} = \sum_{m:(m,n)\in\Omega^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)^2 + \lambda B_n^{(i)}\left(B_n^{(i)}\right)^T \quad (17)$$

That is, when selecting an entry (m,n), the row loss function is the sum of the local loss function of the sample entries sharing the same row with the entry (m,n). Similarly, the column loss function is the sum of the local loss function of the sample entries sharing the same column with the entry (m,n). Based on the row and column loss functions, the factor matrices are updated by the following rules iteratively through the mini-batch SGD:

$$A_m^{(i)} \leftarrow A_m^{(i)} - \varepsilon\frac{l_m^{(i)}}{\partial A_m^{(i)}} \quad (18)$$

$$B_n^{(i)} \leftarrow B_n^{(i)} - \varepsilon\frac{l_n^{(i)}}{\partial B_n^{(i)}} \quad (19)$$

where

$$\frac{\partial l_m^{(i)}}{\partial A_m^{(i)}} = \sum_{n:(m,n)\in\Omega^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right) B_n^{(i)} + \lambda A_m^{(i)} \quad (20)$$

and

$$\frac{\partial l_n^{(i)}}{\partial B_n^{(i)}} = \sum_{m:(m,n)\in\Omega^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right) A_m^{(i)} + \lambda B_n^{(i)} \quad (21)$$

D. Data independent parallel execution

To solve the problem in (11), the mini-batch SGD iteratively selects an instance $p_{m,n}^{(i)}$ and applies the update rules (18) and (19). The iteration process of applying (18) and (19) is inherently sequential, so it is difficult to parallelize the mini-batch SGD under advanced architectures such as GPU, multi-core CPU or distributed clusters. In this subsection, we first analyze the possibility of executing the mini-batch SGD in parallel, and then design a parallel matrix completion algorithm over the GPU platform.

If we check carefully, in each iteration, only one row of the factor matrix $A^{(i)}$ and one row of the factor matrix $B^{(i)}$ are updated. Specifically, for a randomly selected entry $(m,n)\in\Omega^{(i)}$, the basic procedures in (18) and (19) indicate that only the m-th row of the factor matrix $A^{(i)}$ (i.e., $A_m^{(i)}$) and the n-th row of the factor matrix $B^{(i)}$ (i.e., $B_n^{(i)}$) are updated. In a new iteration, if an entry (m',n') with $m'\neq m$ and $n'\neq n$ is selected, the m'-th row of the factor matrix $A^{(i)}$ and n'-th row of the factor matrix $B^{(i)}$ can be updated in parallel with the previous iteration. Therefore, if we know in advance that we are going to first take a local gradient at $p_{m_1,n_1}^{(i)}$ and then at $p_{m_2,n_2}^{(i)}$ with $m_1\neq m_2$ and $n_1\neq n_2$, we could perform these two calculations in parallel. These instances $p_{m_1,n_1}^{(i)}$ and $p_{m_2,n_2}^{(i)}$ are called independent samples.

We refer the time duration taken to pass over all the sample data as an epoch. Given a threadblock with η threads, maximum η independent samples can be operated in parallel. In this paper, one pass of η independent samples is called as a round. As the number of samples in a sub-matrix is generally larger than the number of threads in a threadblock, that is, $|\Omega_i|\geq\eta$, one epoch may consist of multiple rounds.

To take full advantage of the GPU thread resources for parallel processing, our task assignments will consider the following criteria for fast matrix factorization:

- REQ_1. To speed up the convergence process, the group of η tasks for the mini-batch SGD updates in the sequence of rounds within an epoch should be independent. That is, let S_1 and S_2 denote the two groups of tasks for two rounds, these two groups should satisfy that $S_1\neq S_2$.
- REQ_2. The tasks assigned to execute in parallel through multiple threads in a round should be independent with the equal load from each other.

According to the above criteria, we propose Algorithm 2 for data independent task assignment, where the task sequence is generated for each thread over multiple epochs. According to the sequence, a thread applies (18) and (19) to take sample one by one to train and update the factor matrices.

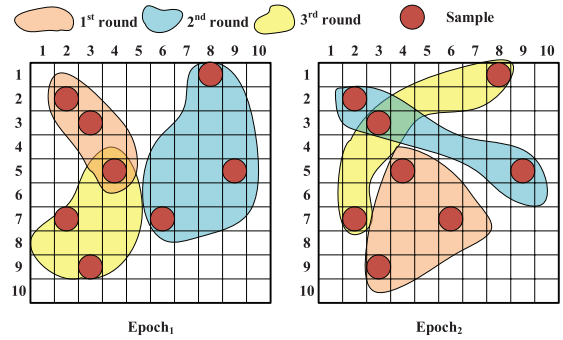
Algorithm 2 Data Independent Task Assignment

Input: the sub-matrix P with the sample set Ω
initial factor matrix A and B for the sub-matrix
the total number of threads in a threadblock, η
the maximum number of epochs to run, K

1: **for** $o=0; o<K; o++$ **do**
2: I=NULL; // the set of samples already run in an epoch
3: **while** $|\Omega - I|\neq 0$ **do**
4: R=NULL; // recording the set of samples in a round
5: $i=0$; // the number of samples already selected in a round
6: **while** $i<\eta$ **do**
7: select a sample in $\Omega - I$ randomly satisfying that it is independent from the samples in R
8: **if** successively find the independent sample **then**
9: $i=i+1$; insert the sample into I; insert the sample into R
10: **else**
11: ***Avoid Thread Idle:*** select the sample in I randomly that it is independent from the samples in R
12: $i=i+1$; insert the sample into R
13: **end if**
14: **end while**
15: **for** each thread j parallely **do**
16: ***Assign Computation task:*** Let the row index and the column index of sample $R[j]$ be $r(R[j]), c(R[j])$. Update corresponding $A_{r(R[j])*}$ and $B_{*c(R[j])}$ using (13) and (14) based on this sample
17: **end for**
18: R=NULL
19: **end while**
20: **end for**

The algorithm takes a random process (line 6-14) to assign the computation tasks in each round to train the factor matrices in parallel. We use I and R to denote the set of samples selected inside an epoch and a round. For each round, to quickly run over all the samples in Ω (i.e., the index set of the observed samples in the sub-matrix), the samples are randomly selected from the remaining sample space $\Omega - I$ to form R (line 7-9). However, the number of samples that can be selected independently in one round may be fewer than the number of threads and thus some threads may be left idle, resulting in a ***a thread idle problem***. To well exploit the thread resource for the maximum parallelism gain, instead of leaving the thread idle, we allow the thread to rerun samples that have been selected in the previous rounds but are independent from the samples to take in the current round (line 11-12).

Fig.8 shows an example to illustrate how we solve the thread idle problem. There are 8 samples and one threadblock consists of 3 threads. In $epoch_1$, the 1^{st} round selects samples p_{22}, p_{33}, p_{54}, and the 2^{nd} round selects samples p_{18}, p_{59}, and p_{76}. After that,



(a) Task assignment in two epochs

Access order

	Epoch1			Epoch2		
	1st round	2nd round	3rd round	1st round	2nd round	3rd round
Thread1	p_{23}, A_2*,B*$_2$	p_{18}, A_1*,B*$_8$	p_{72}, A_7*,B*$_2$	p_{54}, A_5*,B*$_4$	p_{23}, A_2*,B*$_2$	p_{72}, A_7*,B*$_2$
Thread2	p_{23}, A_2*,B*$_3$	p_{99}, A_9*,B*$_9$	p_{93}, A_9*,B*$_3$	p_{76}, A_7*,B*$_6$	p_{33}, A_3*,B*$_3$	p_{18}, A_1*,B*$_8$
Thread3	p_{54}, A_5*,B*$_4$	p_{76}, A_7*,B*$_6$	p_{54}, A_5*,B*$_6$	p_{93}, A_9*,B*$_3$	p_{33}, A_3*,B*$_3$	p_{33}, A_3*,B*$_3$

(b) memory accesses performed by threads in a threadblock

Fig. 8. Illustrate of the data independent execution

only two independent samples p_{72}, p_{93} are left for the 3^{rd} round and $thread_3$ may be idle. To fully exploit the thread resource, the sample p_{54} is selected again to execute at the $thread_3$. Similarly, in the 3^{rd} round of the $epoch_2$, besides samples p_{72} and p_{18}, the sample p_{33} already used in the 2^{nd} round is selected again to run by the potentially idle thread.

Fig.8(b) shows the memory accesses from all threads in a threadblock of the example in Fig.8(a). The threads are listed in the vertical direction, with the time of access increasing to the right in the horizontal direction. Our example has 2 epochs with each consisting of 3 rounds. As the sample set is selected independently in each round, the updated partial factor matrices are independent from each other in each round.

E. Mini-batch SGD based on load balancing

In each execution round in our algorithm (Algorithm 2), each thread selects an independent sample and applies (18) and (19) to update the factor matrices. (18) and (19) further invoke (20), (21) to calculate the local gradients, respectively. As shown in (20), (21), different selected entry has a different number of sampled entries that share the same row or column with it, which may bring a different task load to a different thread. To avoid exploiting the explicit synchronization among threads for efficient parallel execution, we expect that each thread takes the equal load.

Therefore, instead of selecting all the entries that sharing the same row and same column with the entry (m,n), we randomly select K sample entries to calculate the row loss function and the column loss function, which further changes the local gradient calculations, expressed as follows:

$$\frac{\partial l_m^{(i)}}{\partial A_m^{(i)}} = \sum_{n:(m,n)\in\Omega_{row-K}^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)B_n^{(i)} + \frac{\lambda K}{N_m^{(i)}}A_m^{(i)} \quad (22)$$

$$\frac{\partial l_n^{(i)}}{\partial B_n^{(i)}} = \sum_{m:(m,n)\in\Omega_{col-K}^{(i)}} \left(p_{m,n}^{(i)} - A_m^{(i)}(B_n^{(i)})^T\right)A_m^{(i)} + \frac{\lambda K}{N_n^{(i)}}B_n^{(i)} \quad (23)$$

where $\Omega_{row-K}^{(i)}$ and $\Omega_{col-K}^{(i)}$ are the index set of K sample entries that share the row index and the column index with the entry (m,n), respectively, with $(m,n)\in\Omega_{row-K}^{(i)}$ and $(m,n)\in\Omega_{col-K}^{(i)}$.

Obviously, a large K brings more samples to train in an iteration step and thus a fast convergence speed while at the cost of a higher computation overhead in one step. In our performance studies in Section VI-B, we will vary K to test how it impacts the recovery performance. Moreover, when $K=1$, only the entry (m,n) itself is invoked in the gradient calculation and it is a standard SGD solution (shown in (13) and (14)). If K is larger than $N_m^{(i)}$ or $N_n^{(i)}$ (where $N_m^{(i)}$ and $N_n^{(i)}$ are the number of entries observed on the m-th row and n-th column in the $P^{(i)}$), we can select the entries on the row or column multiple times until the total number of entries taken reaches K.

VI. Experiment

We use the public data set Harvard226 [19] and GÈANT [20] to evaluate the proposed techniques in MC-GPU. We first present the experiment setup, then the experiment results.

A. Experiment setup

We denote the raw monitoring data as a matrix P. For more efficient data processing, data normalization [57] is often applied to scale the variables or features of data that the resulted values are within the range [0,1]. In this paper, given a data item $p_{i,j}$, we adopt the following equation to normalize the data

$$p_{i,j} = \frac{p_{i,j} - \min\limits_{i,j}\{p_{i,j}\}}{\max\limits_{i,j}\{p_{i,j}\} - \min\limits_{i,j}\{p_{i,j}\}} \quad (24)$$

where $\max\limits_{i,j}\{p_{i,j}\}$ and $\min\limits_{i,j}\{p_{i,j}\}$ are the maximum value and minimum value of the monitoring data, respectively.

In the experiment, we apply the proposed matrix completion schemes to recover the full performance monitoring data from partial measurement samples. Then, using the raw monitoring trace data as reference, we calculate the performance metrics by comparing the recovered data with the raw data trace. In all the experiments, we set the default sample ratio to be 60%.

To evaluate the accuracy of different matrix completion algorithms, we use two relative error metrics: **Error(sample)** $= \frac{\sqrt{\sum_{(i,j)\in\Omega}(p_{i,j}-\hat{p}_{i,j})^2}}{\sqrt{\sum_{(i,j)\in\Omega}(p_{i,j})^2}}$ and **Error(un-sample)** $= \frac{\sqrt{\sum_{(i,j)\in\overline{\Omega}}(p_{i,j}-\hat{p}_{i,j})^2}}{\sqrt{\sum_{(i,j)\in\overline{\Omega}}(p_{i,j})^2}}$, where $1\le i\le M$, $1\le j\le T$. $p_{i,j}$ and $\hat{p}_{i,j}$ denote the (i,j)-th element of the raw data matrix P and the matrix $\hat{P}$ recovered through the matrix completion. The first metric is the relative error to evaluate the impact of matrix completion on the data elements with observed values already, and the second is error for the matrix element locations with the values inferred from the matrix completion.

To evaluate the speed of different matrix completion algorithms, we use **Computation Time**, a metric for measuring the average number of seconds taken to recover the monitoring matrix.

The parameter ε in (18) and (19) is the learning rate. We employ a bold driver method to determine and set the gradient descent. Starting from an initial learning rate ε^0, we (1) increase the learning rate by a small percentage (say, 5%) whenever the loss decreases over an epoch, and (2) drastically decrease the learning rate (say, by 50%) if the loss increases. Within each epoch, the learning rate remains fixed. For each scheme implemented, we try the learning rates 1, 1/2, 1/4, $\cdots$, $1/2^{d-1}$, and set the learning rate ε^0 to the one that gives the best result for a scheme.

As all the matrix recovery approaches are executed iteratively to train the parameters needed, for a fair comparison, we adopt the same two stop conditions: 1) The difference in the recovery loss between two consecutive iterations is smaller than a given

threshold value, set to 10^{-7} in this paper; 2) The maximum number of epochs is reached, set to 100 in this paper. The iteration process will continue until either of the two stop conditions is satisfied.

We denote these implementations as miniSGD(1)-p, miniSGD(10)-p, SRMF-p, NMF-p, LMaFit-p). Then, for performance comparison, we also implement the matrix completion algorithms directly using the whole large matrix without matrix partition, denoted as miniSGD(1)-w, miniSGD(10)-w, SRMF-w, SVT-w, LMaFit-w.

We implement MC-GPU on a NVIDIA GPU, GeForce GTX 960, which runs CUDA SDK 7.0. Each SM (Streaming Multiprocessor) has 48 KB shared memory and 65536 registers.

B. Impact of K in the mini-batch SGD

As discussed in Section V-E, in each iterative round, our mini-batch SGD selects one sample $p^{(i)}_{m,n}$, which further involves K samples that share the same row index with $p^{(i)}_{m,n}$ and K samples that share the same column index with $p^{(i)}_{m,n}$ to calculate the local gradient in a batch. In this section, we investigate how K impacts the convergence speed and the recovery performance.

Although with the increase of K, the total number of iteration steps needed decreases (Fig.9(c)), the overall computation time (Fig.9(d)) becomes larger as each iteration step takes longer time to calculate the local gradient. When K is small, the loss function is not as well minimized as when K is large. As a result, with the increase of K, the error(sample) and error(un-sample) both decrease thus the matrix recovery accuracy increases. Therefore, there is a tradeoff to determine an appropriate K. When $K = 1$, the mini-batch SGD reduces to standard SGD, a special case. For performance comparison with other state of art matrix completion algorithms, in the experiment, we implement two mini-batch SGD cases with $K = 1$ and $K = 10$ (denoted as miniSGD(1) and miniSGD(10) in following experiment results).

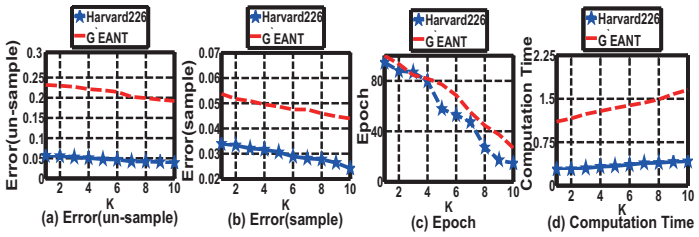


Fig. 9. Convergence behavior.

C. Clustering performance based on LSH table

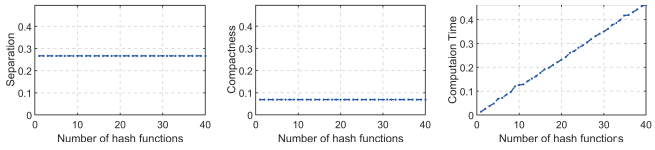
In this paper, we build an LSH table to reorder and cluster OD pairs for efficient matrix partition.

To evaluate the clustering performance, two clustering metrics (Compactness and Separation) are defined as follows. Denote s clusters as $c_1, c_2, \cdots, c_s$. The centroid of cluster c_i is calculated as $\vec{u}_i = \frac{1}{|c_i|}\sum_{\vec{p}_j \in c_i} \vec{p}_j$. The average distance between each OD pair in cluster c_i to its centroid is $CP_i = \frac{1}{|c_i|}\sum_{\vec{p}_j \in c_i}(\vec{p}_j - \vec{u}_i)$. The Compactness (CP) is defined as $CP = \frac{1}{s}\sum_{k=1}^{s} CP_i$. Separation (SP) is defined as $\frac{2}{s^2-s}\sum_{i=1}^{s}\sum_{j=i+1}^{s}||\vec{u}_i - \vec{u}_j||_2$. Low compactness value and large separation value indicate better and more compact clusters.

In our simulation, the clustering algorithms are firstly applied to the matrix data operated by our pre-interpolation technique in section IV-B. We record each OD pair's cluster ID and we use the raw monitoring data to calculate the compactness value and large separation value.

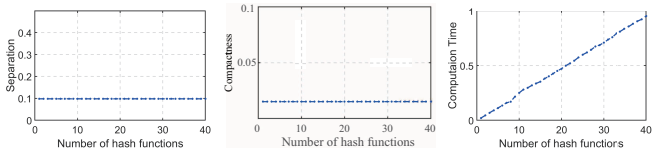
1) Impact of the number of hash functions

To locate the OD pair in the LSH table, according to Eq.(9), the super index of an OD pair is calculated using n hash functions. To investigate how n impacts the performance, we vary n from 1 to 40. Fig.10 and Fig.11 show the clustering performance.



(a) Separation (b) Compactness (c) Computation Time

Fig. 10. Harvard226.



(a) Separation (b) Compactness (c) Computation Time

Fig. 11. GÈANT.

Using more hash functions may reduce the chance of hashing items with larger distance into the same bucket, it also involves higher computation delay. From Fig.10 and Fig.11, for both network monitoring data sets, we don't observe big difference in clustering performance (in terms of separation and compactness) between using one hash function and multiple hash functions. As one hash function requires much smaller computation time, we set $n = 1$ according to the experiment result.

This experiment demonstrates that using one hash function with merging operation can achieve nearly the same good clustering performance as the use of multiple hash functions. This is because our use of LSH is not for looking for nearby neighbors which require the exact mapping of neighboring items into the same bucket, but for the clustering of data. Even though similar items may not be hashed into the same bucket, they have a high possibility of being hashed into the near buckets. To balance the number of data items (i.e., OD pairs) in different clusters, we merge adjacent buckets with fewer data items into a large one, and this merge will largely reduce the inaccuracy. On the other hand, even though distant data points may have a higher chance of being hashed into the same hash bucket when using one hash function, the probability of such an inaccuracy is generally low (i.e., $P_2 < P_1$) and reduces with the distance. This property of LSH is shown in its definition in the Section IV-A and reference [45].

2) Comparing with other clustering algorithm

To compare the performance of matrix partition with LSH and other clustering algorithms, we implement k-means [52], and gaussian mixture model (GMM) [53]. To make them comparable with LSH in the partition, we set their cluster numbers to be the number of sub-matrices in our method. To support balanced partition, we also enhance the three peer algorithms with a least load merge mechanism similar to our Algorithm 1. That is, we first divide the items into w clusters (w is the LSH table size), then iteratively merge the cluster having the least OD pairs with its nearest neighboring cluster until the total number of clusters reaches s.

Table.I shows the clustering performance under different cluster algorithms. Besides compactness and separation, the computation

TABLE I
CLUSTERING METRICS

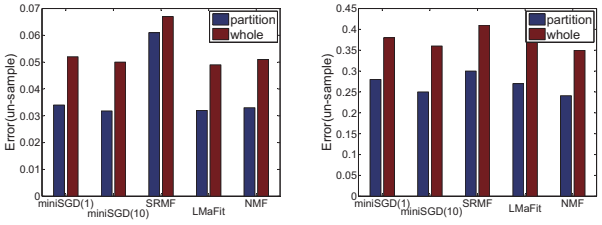
Harvard226			
	LSH	K-means	GMM
Compactness	0.0695	0.1197	0.0546
Separation	0.1615	0.1354	0.0104
Computation Time	2.85 s	3.41 s	26.52 s
GÈANT			
	LSH	K-means	GMM
Compactness	0.5217	0.6955	0.49
Separation	0.6677	0.0463	0.39
Computation Time	0.48 s	0.55 s	915 s

time under different algorithms is also listed. It takes much less time for LSH to partition a matrix, as it only requires a quick hash computation to group similar OD pairs while the two reference methods need to run iteratively and involve significantly higher computational cost. Among all the clustering algorithms, our LSH achieves the similar compactness value and the largest separation value with much lower computation time, thus better clustering performance. Moreover, although all the clustering algorithms are applied to the pre-interpolated data, the compactness values under all clustering algorithms are low. This demonstrates that clustering under pre-interpolation in this paper is effective.

D. Effectiveness of partition based on LSH table

Fig. 12 compares the accuracy of matrix completion with the facilitation of matrix partition based on LSH table. Besides the matrix completion algorithm based on mini-batch SGD, we implement other three matrix completion algorithms $SRMF$ [8], NMF [58], and $LMaFit$ [21]. Firstly, these algorithms are applied to complete the sub-matrices partitioned from the large matrix. For performance comparison, we also implement the matrix completion algorithms directly using the whole large matrix without matrix partition.

Compared with the matrix completion algorithms executed using the whole data, our partition algorithm can bring closer the OD pair correlation in the sub-matrices. Thus matrix completion algorithms based on our partition can more accurately recover the missing data. All the performance results (in Fig. 12) demonstrate that our LSH-facilitated partition scheme is very effective in improving the recovery accuracy, and our partition algorithm is general without depending on the underlying matrix completion algorithms.

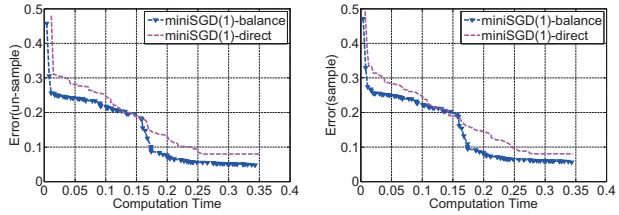


(a) Accuracy under Harvard226 (b) Accuracy under GÈANT

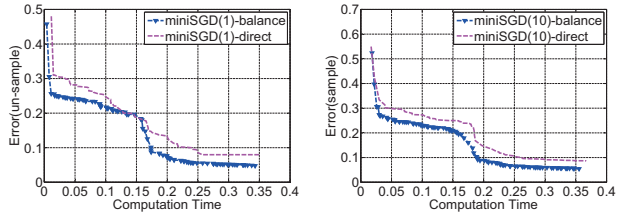
Fig. 12. Recovery accuracy under different matrix completion algorithms.

E. Effectiveness of balanced matrix partition

In section IV-C2, facilitated by our adjacent least-load merge algorithm, we can achieve balanced matrix partition. To evaluate the effectiveness of balanced matrix partition, we also implement another partition algorithm that directly reorder the OD pairs in a LSH table with s buckets (denoted as direct). As our SGD based algorithm executed iteratively, we shows the recovery performance in Fig. 13-14 with x axis denoting computation time. As expected, to achieve same recovery accuracy, matrix completion with our balanced matrix partition needs less time than the other direct partition scheme. This is because that each sub-matrix is executed in

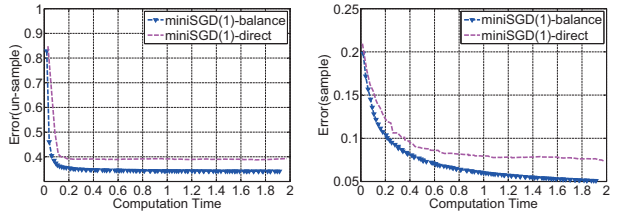


(a) miniSGD(1) Error(un-sample) (b) miniSGD(1) Error(sample)

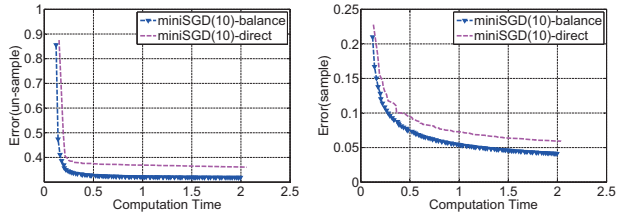


(c) miniSGD(10) Error(un-sample) (d) miniSGD(10) Error(sample)

Fig. 13. Effectiveness of balanced matrix partition(Harvard226).



(a) miniSGD(1) Error(un-sample) (b) miniSGD(1) Error(sample)



(c) miniSGD(10) Error(un-sample) (d) miniSGD(10) Error(sample)

Fig. 14. Effectiveness of balanced matrix partition (GÈANT).

a threadblock, our balanced matrix partition can help threadblocks in GPU platform be assigned with tasks of balanced load to achieve good parallel processing gain, while direct matrix partition may assign different computation load to different threadblocks and thus the overall matrix recovering time depends on the overloaded threadblocks. As a result, it takes much longer time for the direct method to reach the same recovery error rate in all test cases.

F. Comparing with other GPU based matrix completion

Besides our miniSGD(1)-p and miniSGD(10)-p, we implement other two GPU based matrix completion algorithms cumf_sgd [59] and ALS (denoted as cumf_ALS [60]). To make cumf_sgd and cumf_ALS comparable with our algorithms, in cumf_sgd and cumf_ALS, the large matrix is partitioned into sub-matrices by randomly selected OD pairs. Consistent with the complexity analysis in Section III-D and result in [59], to achieve same matrix recovery accuracy, cumf_ALS is much slower than cumf sgd, miniSGD(1)-p, and miniSGD(10)-p because of its large computation cost. Although our algorithms require hash computation to partition the matrix, the convergence speed under our algorithms are similar to cumf_sgd. When converges, compare with cumf_sgd, the error(sample) is lower under our miniSGD(1)-p and miniSGD(10)-p because of our

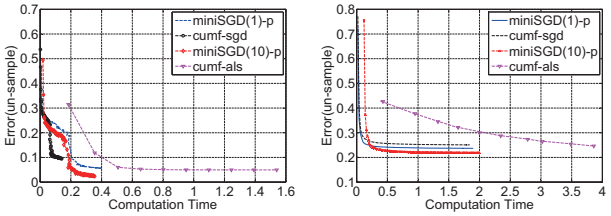
partition algorithm can bring closer the OD pair correlation in the sub-matrices.

G. Speed gain brought by GPU parallelism

To evaluate the speed gain brought by GPU platform, we implement our matrix partition based matrix completion algorithm on CPU platform. Following our matrix partition, we feed each sub-matrix one by one to CPU platform(Intel I5-8400). The computation time is counted as the total computation time of all the sub-matrices. Compared with CPU implementations, the GPU based implementations are more than 20 times faster.

TABLE II
COMPUTATION TIME COMPARISON (GPU AND CPU)

	Harvard226		GÈANT	
	GPU	CPU	GPU	CPU
miniSGD(10)-p	0.368 s	19.5 s	2.12 s	76.5 s
miniSGD(1)-p	0.34 s	15.6 s	1.91 s	56.8 s



(a) un-sample (b) un-sample

Fig. 15. SGD vs. ALS under trace GÈANT on the GPU platform.

VII. CONCLUSION

We present MC-GPU, a high performance scheme for accurate and high-speed matrix completion with Graphics Processing Units (GPUs). Facilitated by locality-sensitive hash (LSH) functions, we propose one elegant mechanism to partition the large matrix into small sub-matrices with each sub-matrix consisting of OD pairs with closer correlation, which helps to more accurately recover the missing data in the sub-matrices. Besides, to well exploit the special architecture of GPU to maximize its parallelism gain, we further propose a parallel matrix completion mechanism to provide both task independent and data-independent parallel task execution for high speed data recovery.

With real trace data, we have conducted extensive experiments over GPU platform and commodity CPU platform, respectively. We compare the performance of MC-GPU with three state of art matrix completion algorithms. Our results demonstrate that MC-GPU can achieve significantly higher recovery speed with high recovery accuracy. Moreover, the experimental results also demonstrate that our LSH-facilitated partition scheme is very effective in improving the recovery performance, and it is general and dose not depend on the underlying matrix completion algorithms.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China under Grant 61972144, 61572184, 61725206, and 61976087, in part by the Hunan Provincial Natural Science Foundation of China under Grant 2017JJ1010, in part by U.S. NSF under Grant ECCS 78929 and CNS 1526843, in part by the Open Project Funding (CARCH201809) of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences, and in part by the Peng Cheng Laboratory Project of Guangdong Province under Grant PCL2018KP004.

REFERENCES

[1] K. Xie, X. Li, X. Wang, J. Cao, G. Xie, J. Wen, D. Zhang, and Z. Qin, "On-line anomaly detection with high accuracy," *IEEE/ACM transactions on networking*, vol. 26, no. 3, pp. 1222–1235, 2018.

[2] K. Xie, C. Peng, X. Wang, G. Xie, J. Wen, J. Cao, D. Zhang, and Z. Qin, "Accurate recovery of internet traffic data under variable rate measurements," *IEEE/ACM transactions on networking*, vol. 26, no. 3, pp. 1137–1150, 2018.

[3] K. Xie, X. Li, X. Wang, G. Xie, J. Wen, J. Cao, and D. Zhang, "Fast tensor factorization for accurate internet anomaly detection," *IEEE/ACM transactions on networking*, vol. 25, no. 6, pp. 3794–3807, 2017.

[4] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pp. 561–575, 2018.

[5] E. J. Candès and B. Recht, "Exact matrix completion via convex optimization," *Foundations of Computational mathematics*, vol. 9, no. 6, pp. 717–772, 2009.

[6] R. H. Keshavan, A. Montanari, and S. Oh, "Matrix completion from a few entries," *IEEE Transactions on Information Theory*, vol. 56, no. 6, pp. 2980–2998, 2010.

[7] B. Recht, M. Fazel, and P. A. Parrilo, "Guaranteed minimum-rank solutions of linear matrix equations via nuclear norm minimization," *SIAM review*, vol. 52, no. 3, pp. 471–501, 2010.

[8] M. Roughan, Y. Zhang, W. Willinger, and L. Qiu, "Spatio-temporal compressive sensing and internet traffic matrices (extended version)," *Networking IEEE/ACM Transactions on*, vol. 20, no. 3, pp. 662 – 676, 2012.

[9] M. Mardani and G. Giannakis, "Robust network traffic estimation via sparsity and low rank," in *IEEE ICASSP*, 2013.

[10] R. Du, C. Chen, B. Yang, and X. Guan, "Vanet based traffic estimation: A matrix completion approach," in *IEEE GLOBECOM*, 2013.

[11] G. Gürsun and M. Crovella, "On traffic matrix completion in the internet," in *ACM IMC 2012*.

[12] Y.-C. Chen, L. Qiu, Y. Zhang, G. Xue, and Z. Hu, "Robust network compressive sensing," in *ACM MobiCom*, 2014.

[13] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis, "Large-scale matrix factorization with distributed stochastic gradient descent," in *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 69–77, ACM, 2011.

[14] B. Recht and C. Ré, "Parallel stochastic gradient algorithms for large-scale matrix completion," *Mathematical Programming Computation*, vol. 5, no. 2, pp. 201–226, 2013.

[15] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin, "A fast parallel sgd for matrix factorization in shared memory systems," in *Proceedings of the 7th ACM conference on Recommender systems*, pp. 249–256, ACM, 2013.

[16] R. Kannan, G. Ballard, and H. Park, "A high-performance parallel algorithm for nonnegative matrix factorization," *arXiv preprint arXiv:1509.09313*, 2015.

[17] C. Teflioudi, F. Makari, and R. Gemulla, "Distributed matrix completion," in *Data Mining (ICDM), 2012 IEEE 12th International Conference on*, pp. 655–664, IEEE, 2012.

[18] H.-F. Yu, C.-J. Hsieh, S. Si, and I. S. Dhillon, "Parallel matrix factorization for recommender systems," *Knowledge and Information Systems*, vol. 41, no. 3, pp. 793–819, 2014.

[19] J. Ledlie, P. Gardner, and M. I. Seltzer, "Network coordinates in the wild.," in *NSDI*, vol. 7, pp. 299–311, 2007.

[20] S. Uhlig, B. Quoitin, J. Lepropre, and S. Balon, "Providing public intradomain traffic matrices to the research community," *ACM SIGCOMM Computer Communication Review*, vol. 36, no. 1, pp. 83–86, 2006.

[21] Z. Wen, W. Yin, and Y. Zhang, "Solving a low-rank factorization model for matrix completion by a nonlinear successive over-relaxation algorithm," *Mathematical Programming Computation*, vol. 4, no. 4, pp. 333–361, 2012.

[22] T. Zhao, Z. Wang, and H. Liu, "Nonconvex low rank matrix factorization via inexact first order oracle," *Advances in Neural Information Processing Systems*, 2015.

[23] J.-F. Cai, E. J. Candès, and Z. Shen, "A singular value thresholding algorithm for matrix completion," *SIAM Journal on Optimization*, vol. 20, no. 4, pp. 1956–1982, 2010.

[24] S. Ji and J. Ye, "An accelerated gradient method for trace norm minimization," in *Proceedings of the 26th annual international conference on machine learning*, pp. 457–464, ACM, 2009.

[25] Z. Liu and L. Vandenberghe, "Interior-point method for nuclear norm approximation with application to system identification," *SIAM Journal on Matrix Analysis and Applications*, vol. 31, no. 3, pp. 1235–1256, 2009.

[26] S. Ma, D. Goldfarb, and L. Chen, "Fixed point and bregman iterative methods for matrix rank minimization," *Mathematical Programming*, vol. 128, no. 1-2, pp. 321–353, 2011.

[27] K.-C. Toh and S. Yun, "An accelerated proximal gradient algorithm for nuclear norm regularized linear least squares problems," *Pacific Journal of Optimization*, vol. 6, no. 615-640, p. 15, 2010.

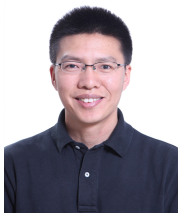
[28] K. Xie, L. Wang, X. Wang, G. Xie, G. Zhang, D. Xie, and J. Wen, "Sequential and adaptive sampling for matrix completion in network monitoring systems," in *IEEE Infocom*, 2015.

[29] K. Xie, L. Wang, X. Wang, J. Wen, and G. Xie, "Learning from the past: Intelligent on-line weather monitoring based on matrix completion," in *IEEE ICDCS*, 2014.
[30] K. Xie, X. Ning, X. Wang, D. Xie, J. Cao, G. Xie, and J. Wen, "Recover corrupted data in sensor networks: A matrix completion solution," *IEEE Transactions on Mobile Computing*, vol. 16, no. 5, pp. 1434–1448, 2017.
[31] K. Xie, L. Wang, X. Wang, G. Xie, and J. Wen, "Low cost and high accuracy data gathering in wsns with matrix completion," *IEEE Transactions on Mobile Computing*, vol. 17, no. 7, pp. 1595–1608, 2018.
[32] X. Wang, C. Xu, G. Zhao, K. Xie, and S. Yu, "Efficient performance monitoring for ubiquitous virtual networks based on matrix completion," *IEEE Access*, vol. 6, pp. 14524–14536, 2018.
[33] H. Li, K. G. Li, J. An, and K. G. Li, "An online and scalable model for generalized sparse non-negative matrix factorization in industrial applications on multi-gpu," *IEEE Transactions on Industrial Informatics*, pp. 1–1, 2019.
[34] C.-J. Hsieh and P. Olsen, "Nuclear norm minimization via active subspace selection," in *International Conference on Machine Learning*, pp. 575–583, 2014.
[35] Y. Koren, R. Bell, and C. Volinsky, "Matrix factorization techniques for recommender systems," *Computer*, vol. 42, no. 8, 2009.
[36] J. D. Lee, B. Recht, N. Srebro, J. Tropp, and R. R. Salakhutdinov, "Practical large-scale optimization for max-norm regularization," in *Advances in Neural Information Processing Systems*, pp. 1297–1305, 2010.
[37] J. D. Rennie and N. Srebro, "Fast maximum margin matrix factorization for collaborative prediction," in *ACM ICML*, 2005.
[38] Y. Mao, L. K. Saul, and J. M. Smith, "Ides: An internet distance estimation service for large networks," *IEEE Journal on Selected Areas in Communications*, vol. 24, no. 12, pp. 2273–2284, 2006.
[39] Y. Liao, W. Du, P. Geurts, and G. Leduc, "Dmfsgd: A decentralized matrix factorization algorithm for network distance prediction," *IEEE/ACM Trans. Netw.*, vol. 21, pp. 1511–1524, Oct. 2013.
[40] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, Morgan Kaufmann Publishers Inc.
[41] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe LSH: efficient indexing for high-dimensional similarity search," in *VLDB*, VLDB Endowment, 2007.
[42] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *ACM STOC*, 1998.
[43] A. P. de Vries, N. Mamoulis, N. Nes, and M. Kersten, "Efficient k-nn search on vertically decomposed data," in *ACM SIGMOD*, 2002.
[44] J. Gan, J. Feng, Q. Fang, and W. Ng, "Locality-sensitive hashing scheme based on dynamic collision counting," in *ACM SIGMOD*, 2012.
[45] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *ACM SCG*, 2004.
[46] B. Recht, C. Re, S. Wright, and F. Niu, "Hogwild: A lock-free approach to parallelizing stochastic gradient descent," in *Advances in Neural Information Processing Systems*, pp. 693–701, 2011.
[47] C. Nvidia, "Nvidia cuda c programming guide," *Nvidia Corporation*, vol. 120, no. 18, p. 8, 2011.
[48] D. B. Kirk and W. H. Wen-Mei, *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2016.
[49] I. Markovsky, *Low rank approximation: algorithms, implementation, applications*. Springer Science & Business Media, 2011.
[50] C. Eckart and G. Young, "The approximation of one matrix by another of lower rank," *Psychometrika*, vol. 1, no. 3, pp. 211–218, 1936.
[51] Y. Hu, Y. Koren, and C. Volinsky, "Collaborative filtering for implicit feedback datasets," in *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pp. 263–272, Ieee, 2008.
[52] H. Spath, *The cluster dissection and analysis theory FORTRAN programs examples*. Prentice-Hall, Inc., 1985.
[53] G. McLachlan and D. Peel, *Finite mixture models*. John Wiley & Sons, 2004.
[54] V. M. Zolotarev, *One-dimensional stable distributions*, vol. 65. American Mathematical Soc., 1986.
[55] A. Andoni and P. Indyk, "E 2 lsh 0.1 user manual," 2005.
[56] S. Ruder, "An overview of gradient descent optimization algorithms," *CoRR*, vol. abs/1609.04747, 2016.
[57] S. Aksoy and R. M. Haralick, "Feature normalization and likelihood-based similarity measures for image retrieval," *Pattern recognition letters*, vol. 22, no. 5, pp. 563–582, 2001.
[58] D. D. Lee and H. S. Seung, "Algorithms for non-negative matrix factorization," in *Advances in neural information processing systems*, pp. 556–562, 2001.
[59] X. Xie, W. Tan, L. L. Fong, and Y. Liang, "Cumf_sgd: Fast and scalable matrix factorization," *arXiv preprint arXiv:1610.05838*, 2016.
[60] W. Tan, L. Cao, and L. Fong, "Faster and cheaper: Parallelizing large-scale matrix factorization on gpus," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, pp. 219–230, ACM, 2016.

Kun Xie received the Ph.D. degree in computer application from Hunan University, Changsha, China, in 2007. She is currently a Professor with Hunan University, Peng Cheng Laboratory, and Purple Mountain Laboratory. She has published over 60 articles in major journals and conference proceedings, including journals IEEE/ACM TON, IEEE TMC, IEEE TC, IEEE TWC, and IEEE TSC, and conferences, including SIGMOD, INFOCOM, ICDCS, SECON, DSN, and IWQoS. Her research interests include network measurement, network security, big data, and AI.

Yuxiang Chen is now a PhD candidate in Hunan University. His research interests include parallel computing and matrix completion.

Xin Wang received the Ph.D. degree in electrical and computer engineering from Columbia University, USA. She is currently an Associate Professor with the Department of Electrical and Computer Engineering, The State University of New York at Stony Brook, USA. Her research interests include algorithm and protocol design in wireless networks and communications, mobile and distributed computing, and networked sensing and detection. She was a member of ACM in 2004. She received the NSF Career Award in 2005 and the ONR Challenge Award in 2010.



Gaogang Xie received the B.S. degree in physics, the M.S. and Ph.D. degrees in computer science from Hunan University in 1996, 1999, and 2002, respectively. He is currently a Professor with the Computer Network Information Center (CNIC), Chinese Academy of Sciences (CAS), and the University of Chinese Academy of Sciences and the Vice President of CNIC. His research interests include Internet architecture, packet processing and forwarding, and Internet measurement.

Jiannong Cao (M'93-SM'05-FM'14) received the Ph.D. degree in computer science from Washington State University, Pullman, WA, USA, in 1990. He is currently a Chair Professor with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, where he is also the Director of the Internet and Mobile Computing Lab. He is also the Director of University's Research Facility in Big Data Analytics. His research interests include parallel and distributed computing, wireless sensing and networks, pervasive and mobile computing, and big data and cloud computing.

Jigang Wen received the Ph.D. degrees in computer application from Hunan University, China, in 2011. He was a Research Assistant with the Department of Computing, The Hong Kong Polytechnic University, from 2008 to 2010. He is currently with the Computer Network Information Center (CNIC), Chinese Academy of Sciences. His research interests include wireless networks and mobile computing, and high-speed network measurement and management.