GPU-Accelerated Parallel Hierarchical Extreme Learning Machine on Flink for Big Data

Cen Chen, Kenli Li, *Senior Member, IEEE,* Aijia Ouyang, Zhuo Tang, *Member, IEEE,*
Keqin Li, *Fellow, IEEE*

Abstract—Extreme learning machine (ELM) has become one of the most important and popular algorithms of machine learning, because of its extremely fast training speed, good generalization, and universal approximation/classification capability. The proposal of hierarchical extreme learning machine (H-ELM) extends ELM from single hidden layer feedforward networks (SLFNs) to multilayer perceptron (MLP), greatly strengthening the applicability of ELM. Generally speaking, during training H-ELM, large-scale datasets are needed. Therefore, how to make use of H-ELM framework in processing big data is worth further exploration. This paper proposes a parallel H-ELM algorithm based on Flink, which is one of the in-memory cluster computing platforms, and GPUs. Several optimizations are adopted to improve the performance, such as cache-based scheme, reasonable partitioning strategy, memory mapping scheme for mapping specific JVM objects to buffers. Most importantly, our proposed framework for utilizing GPUs to accelerate Flink for big data is general. This framework can be utilized to accelerate many other variants of ELM and other machine learning algorithms. To the best of our knowledge, it is the first kind of library, which combines in-memory cluster computing with GPUs to parallelize H-ELM. The experimental results have demonstrated that our proposed GPU-accelerated parallel H-ELM named as GPH-ELM can efficiently process large-scale datasets with good performance of speedup and scalability, leveraging the computing power of both CPUs and GPUs in the cluster.

Index Terms—Big data, deep learning, Flink, GPGPU, hierarchical extreme learning machine (H-ELM), parallel

1 Introduction

1.1 Motivation

With the rapid development of Internet and Internet of things technologies, recent years have witnessed a surge of data at a fast speed. Due to the development of big data technologies, many decision methods based on the traditional experience and intuition have been replaced by data analysis and data mining. Many researchers have focused on getting valuable information and knowledge from massive data with data mining or machine learning methods [1], [2].

During the past years, extreme learning machine (ELM) [3], [4], [5] has become one of the most important and popular algorithms of machine learning and artificial intelligence, because of its extremely fast training capacity, good generalization, and universal approximation/classification capability. Unlike other traditional learning algorithms, e.g., back propagation (BP)-based neural networks (NNs), ELM theories believe that the hidden layer of ELM does not need to be iteratively tuned and the parameters of the hidden layer can be generated randomly. Theoretically, Huang et al. [6], [7] have proved that with randomly generated hidden neurons, ELM still maintain its universal approximation capability. Tang et al. [8] proposed a hierarchical ELM (H-ELM) framework for multilayer perceptrons (MLPs), which is based on the universal approximation capability of the original ELM. This proposal has extended the original ELM algorithm from shallow architecture to deep architecture, largely strengthening the applicability of ELM. The H-ELM framework contains two main components: (a) several unsupervised feature encoding layers and (b) supervised feature classification based on the original ELM. Unlike the greedy layerwise training of deep learning (DL), once the previous layer is established, the weights of the current layer are fixed without fine-tuning, making it more efficient in learning performance than the DL [8].

Generally speaking, applications on MLPs (e.g., images/videos) usually require large-scale datasets. As we know, there are many hidden layers in H-ELM framework. These factors engender the need for large amounts of computing resources and cost plenty of time to train H-ELM framework, thus making it prohibitive to complete the training job on a single computer. How to make use of H-ELM algorithm in processing a large amount of data is worth further exploration, which constitutes one main challenge for researchers..

Hadoop, an open source MapReduce framework [9], has been highly successful in implementing large-scale data-intensive applications on commodity clusters. Xun et al. [10] proposed a parallel mining algorithm for frequent itemsets using MapReduce. Ding et al. [11] proposed a novel

- *Cen Chen, Kenli Li, Aijia Ouyang, Zhuo Tang and Keqin Li are with the College of Information Science and Engineering, Hunan University, Changsha, Hunan 410082, China; and the National Supercomputing Center in Changsha, Changsha, Hunan 410082, China.*
  E-mail: chencen@hnu.edu.cn; lkl@hnu.edu.cn; ztang@hnu.edu.cn; likq@hnu.edu.cn
- *Aijia Ouyang are with the Department of Information Engineering, Zunyi Normal College, Zunyi, Guizhou 563006, China.*
  E-mail: oyaj@hnu.edu.cn;
- *Corresponding author: Kenli Li*
- *Keqin Li is also with the Department of Computer Science, State University of New York, New Paltz, New York 12561, USA.*
  E-mail: lik@newpaltz.edu

attribute equilibrium dominance reduction accelerator (DCCAEDR) based on the distributed coevolutionary cloud model and MapReduce, aiming at the tremendous challenge of attribute reduction for big data mining and knowledge discovery.

However, it is a disk-based system and every MapReduce stage can only interact with other stages through the Hadoop Distributed File System (HDFS). In-memory cluster computing platforms (e.g., Flink [12] and Spark [13]) are designed to process data-intensive applications with distributed in-memory architecture, and provide similar scalability, and fault-tolerance characteristics to Hadoop. Because of their in-memory parallel execution model which saves huge amounts of disk I/O operations time, they are more suitable for data mining and machine learning that require many iterative operations.

Over the past few years, graphics processing units (GPUs) have emerged as parallel processors thanks to their high computational power and low price, especially for high-performance computing (HPC) area. In many supercomputers, such as Tianhe and Titan, CPUs and GPUs cooperate together to produce powerful computing. The trend of using heterogeneous CPU-GPU clusters is now mainstream. For example, each computing node of the Tianhe-1A has two Intel(R) Xeon X5670 CPUs and one NVIDIA(R) Tesla M2050 GPU. As with the high computing power of GPUs, many researchers have focused on utilizing GPUs to accelerate the deep learning and have obtained excellent effect. NVIDIA has developed deep neural network library (cuDNN) [14] which is a GPU-accelerated library for deep neural networks. Coates et al. [15] utilized a cluster of GPU servers with Infiniband interconnects and MPI to train extremely large networks (with over 1 billion parameters) within a couple of days.

1.2 Contributions

Existing in-memory cluster computing platforms have been proven to be outstanding platforms for processing big data with high performance, high fault tolerance, high-level and easy programming model and high compatibility with many open source stacks. Apache Flink is a new open source platform for both distributed stream and batch data processing. Spark and Flink are quite similar. As we know, H-ELM framework consists of many hidden layers, which are similar with the iterative operation. The in-memory parallel execution model provided by Flink can cache the data in the distributed memory in the cluster, thus saving huge amounts of disk I/O operations time between two layers. Through deep analysis of the details of H-ELM, we find that Flink is a very appropriate platform for parallelizing H-ELM framework.

Moreover, through analysing the execution process of H-ELM framework, we find that some time-consuming sub-processes are appropriate for being executed in GPUs. Through accelerating them by GPUs, a high speedup will be achieved. However, in-memory cluster computing platforms such as Flink and Spark can only run on CPUs now. That is to say, these platforms cannot leverage the available computing resources of GPUs, or benefit from the acceleration of GPUs, which may be present in the nodes of the cluster.

In this paper, we have proposed a novel parallel hierarchical extreme learning machine combining Flink and GPUs to process large-scale datasets. Our Our algorithms are built on top of Flink, thus inheriting the existing good reliability and expandability of Flink. Due to the fact that Spark and Flink are very similar, our design can be easily integrated into Spark. To the best of our knowledge, it is the first kind of library, which combines in-memory cluster computing with GPUs to parallelize H-ELM, inheriting the outstanding features of both in-memory cluster computing and GPU. The framework for utilizing GPUs to accelerate H-ELM on Flink is an extension of our previous work [16]. In this new framework, we proposed a heterogeneous task management for hybrid CPUs and GPUs, in which CPUs and GPUs cooperate together to fulfill the works assigned to them, thus achieving a better acceleration than our previous work. Our main contributions are as follows.

- The sub-operations of H-ELM which need to be parallelized are parallelized on Flink, thus benefiting from the in-memory cluster computing. In the meantime, several optimizations are adopted to improve the scalability and performance of our parallelization. Furthermore, we accelerate our proposed parallel algorithm through leveraging the high computing power of GPUs, which may be present in the clusters, thus improving the performance greatly.
- Our proposed parallel algorithm is based on both Flink and GPUs, thus inheriting the outstanding features of both Flink and GPUs. Our proposed parallel algorithm has good fault tolerance and reliability, supporting distributed file system, high performance for iterative computing, high computing power provided by GPUs.
- Our proposed framework for utilizing GPUs to accelerate applications on Flink for big data is widely applicable, in which CPUs and GPUs cooperate with each other to fulfill the tasks with high performance. Many other variants of ELM and other machine learning algorithms can benefit from our proposed framework.

The remainder of this paper is organized as follows. Section 2 and Section 3 review related work and provide more background information. Section 4 describes our proposed parallel H-ELM framework on Flink (PH-ELM). Section 5 presents details of accelerating our proposed parallel H-ELM framework by GPUs (GPH-ELM). Section 6 analyzes our proposed algorithms. Section 7 presents the performance results. Section 8 concludes this paper.

2 Related Works

Extreme learning machine (ELM) was fisrt proposed by Huang [4], [5] to train single-hidden layer feedforward Neural networks (SLFNs). Thanks to the efforts of different scholars and researchers, ELM algorithm has been greatly developed and improved. Many researchers have come up with different variants of ELM algorithm. Liang et al. [17] proposed an online sequential ELM (OS-ELM) algorithm, which could train the data block of either fixed or unfixed sizes in an incremental quantity. Thus, it provides a way of utilizing the ELM algorithm to train a large number of data samples. Rong et al. [18] proposed an online sequential fuzzy extreme learning machine (OS-FUZZY-ELM) for function approximation and classification problems. Huang et al. [19] has put forward the semi-supervised extreme learning machine (SS-ELM) based on manifold regularization and ELM.

With the development of ELM algorithm, many scholars have focused on the study of distributed ELM algorithm. Q. He et al. [20] proposed the parallel ELM algorithm (PELM) based on MapReduce. Wang et al. [21] put forward the parallel online sequential extreme learning machine (POS-ELM) based on OS-ELM and MapReduce model for training the incremental data samples in parallel. Xin came up with ELM* algorithm [22] which has higher efficiency than PELM.

We can find that there are many matrix operations in H-ELM framework. Many works have focused on efficient parallel matrix multiplication. Among traditional parallel matrix multiplication based on MPI, SUMMA [23] is the most popular distributed matrix multiplication algorithm. It is obvious that these methods have the disadvantage that all the data needs to be resided in the shared memory of the cluster. Furthermore, these solutions require designing complicated fault tolerance mechanisms. To solve the low programmability of traditional distributed approaches, Schmidt et al. [24] proposed an approach to integrate R into [25]. However, this approach still suffers the fault tolerance problems. Recently, HAMA [26], which is based on Hadoop and MapReduce model, provides distributed matrix computations. However, the execution performance of HAMA is not efficient due to the overhead and disk operations of the MapReduce jobs. Due to the high computational performance of GPUs, many researchers from both academia and industry have proposed GPU-based accelerated matrix operation libraries or algorithms, such as cuBLAS [27] and [28]. Yang et al. [29] proposed a partitioning scheme for SpMV on GPUs and multicore CPUs. Li et al. [29] proposed a probabilistic modeling method to improve the performance of SpMV on GPUs. However, they are not distributed operation libraries.

3 Background

3.1 ELM Learning Algorithm

During the past decades, many researchers have studied the universal approximation capability of SLFNs deeply [30], [31]. It is usually assumed that the activation function of the hidden neurons is continuous and differentiable, and the parameters of hidden neurons need to be adjusted during training. However, it has been proven that randomly generated networks with the outputs being solved by least mean square are able to maintain the universal approximation capability [6], [7].

ELM was put forward for ”generalized” single-hidden layer feedforward networks (SLFNs) where the hidden layer does not need to be neuron alike [3], [32]. The output function of ”generalized” SLFNs with l hidden nodes can be represented by Equation (1):

$$f_l(x) = \sum_{i=1}^{l} \beta_i G(a_i, b_i, x), x \in R^d, \beta_i \in R^o \tag{1}$$

where β_i is the output weight of the ith hidden node connecting with the output layer, a_i is the input weight vector connected the input layer to the ith hidden node, b_i is the bias weight of the ith hidden node, $G(x)$ denotes the activation function, d denotes the dimension of the sample and o denotes the dimension of the label of the sample.

For N arbitrary distinct samples $(x_i, t_i), x_i \in R^d, t_i \in R^m, i = 1, ..., N$, where x_i is the training data vector, t_i represents the target of each sample. Equation (1) can be expressed compactly as Equation (2):

$$H\beta = T \tag{2}$$

where H is the hidden layer output matrix (randomized matrix), T is the training data target matrix, β is the output weight vector. H, T and β are expressed as Equation (3):

$$H = \begin{bmatrix} G(a_1, b_1, x_1) & \cdots & G(a_l, b_l, x_1) \\ \vdots & & \\ G(a_1, b_1, x_N) & \cdots & G(a_l, b_l, x_N) \end{bmatrix}_{N \times l}$$

$$\beta = \begin{bmatrix} \beta_1^T \\ \vdots \\ \beta_l^T \end{bmatrix}_{l \times o} \quad T = \begin{bmatrix} t_1^T \\ \vdots \\ t_N^T \end{bmatrix}_{N \times m} \tag{3}$$

Different from traditional learning algorithms, ELM tends to reach not only the smallest training error but also the smallest norm of output weights [3]:

$$Minimize : ||H\beta - T||^2 + ||\beta|| \tag{4}$$

In the classical implementation of ELM [4], [5], the minimal norm least square method is used to obtain the output weight vector:

$$\beta^* = H^\dagger T \tag{5}$$

where $H^\dagger$ is the Moore-Penrose (MP) generalized inverse of matrix H. The orthogonal projection method can be efficiently used to calculate the MP inverse: $H^\dagger = (H^T H)^{-1} H^T$ if $H^T H$ is nonsingular, or $H^T (HH^T)^{-1}$ if HH^T is nonsingular. According to the ridge regression theory, it was recommended that a positive value $\frac{1}{\lambda}$ can be added during the calculation of the output weights, thus achieving a more stable solution and better generalization performance. Therefore, we can have:

$$\beta = \mathrm{H}^T \left(\frac{1}{\lambda} + HH^T \right)^{-1} T \tag{6}$$

And the corresponding output function of ELM is:

$$f(x) = h(x)\beta = h(x)\mathrm{H}^T \left(\frac{1}{\lambda} + HH^T \right)^{-1} T \tag{7}$$

Or we can have:

$$\beta = \left(\frac{1}{\lambda} + H^T H \right)^{-1} H^T T \tag{8}$$

$$f(x) = h(x)\beta = h(x) \left(\frac{1}{\lambda} + HH^T \right)^{-1} H^T T \tag{9}$$

Huang et al. [3] demonstrated that the solutions to Equation (6) and Equation (8) are actually consistent to minimize $||H\beta - T||^2 + \lambda||\beta||^2$, which is the essential target of ELM as mentioned before. As for big data circumstance, the number of hidden nodes is much less than that of training samples. According to the matrix theory, with SVD method, a small matrix $H^T H$ could be calculated instead of the large matrix HH^T.

3.2 ELM-Based Sparse Autoencoder

Representational learning, e.g., stacked autoencoder (SAE), is effective in learning useful features for achieving high generalization performance [33]. Apart from being used to train SLFNs, the ELM theory has also been applied to build an autoencoder for multilayer perceptron (MLP). Autoencoder aims at learning representations of the input that are robust to small irrelevant changes in input and always functions as some sort of feature extractor in a multilayer learning framework [34]. Mathematically, an autoencoder takes an input vector x, and first maps it to a higher level representation y through a deterministic mapping $y = h_\theta(x) = g(A \cdot x + b)$, parameterized by $\theta = \{A, b\}$, where $g(\cdot)$ is the activation function, A is a $d \times d'$ weight matrix and b is a bias vector. The resulting latent representation y is then mapped back to a reconstructed vector z in the input space $z = h_{\theta'}(y) = g(A' \cdot y + b)$ with $\theta' = \{A', b'\}$.

Tang et al. [8] proposed ELM-based sparse autoencoder. Through performing ℓ_1 optimization, more sparse and compact features of the inputs are generated for the establishment of ELM autoencoder. Unlike the autoencoders (i.e., BP-based algorithm) used in traditional deep learning (DL) algorithms, the input weights of the proposed ELM-based sparse autoencoder are established by searching the path back from a random space. The optimization model of the ELM sparse autoencoder proposed in [35] can be denoted as the following equation:

$$O_\beta = \underset{\beta}{argmin}\{p(\beta) + q(\beta)\} \tag{10}$$

where $p(\beta) = ||H\beta - X||^2$, and $q(\beta) = ||\beta||_{\ell_1}$ is the ℓ_1 penalty term of the training model.

Beck et al. [36] proposed a fast iterative shrinkage-thresholding algorithm (FISTA) to solve the problem in Equation (10). It has been proven that FISTA has a global rate of convergence which is significantly better, both theoretically and practically, especially for big data. The pseudocode of implementing FISTA for solving ELM-based sparse autoencoder is shown in Algorithm 3.1.

3.3 H-ELM Framework for Deep Learning

Tang et al. [8] proposed a hierarchical ELM (H-ELM) framework for multilayer perceptrons (MLPs). The H-ELM training architecture has two separate parts: unsupervised hierarchical feature representation and supervised feature classification. For the former phase, the ELM-Based sparse autoencoder described in Section 3.2 is performed to extract multilayer sparse features of the input data. While for the latter one, the original ELM algorithm is used for making the final decision.

Algorithm 3.1 ELM-based Sparse Autoencoder Overview

```
Input: The training dataset matrix $X$: $\{(x_i)|x_i \in R^d, i = 1, ..., N\}$;
    Hidden node output function: $G(a_i, b_i, x)$;
    The number of hidden nodes: $l$;
    The iteration size: $size$;
    Hidden node matrix $V$: $(a_i, b_i), i = 1, ..., l$;
Output: The hidden weight vector: $\beta$;
Calculate the hidden layer output matrix $H$;
Calculate the matrix $H^T H$;
Calculate the matrix $H^T X$;
Calculate the Lipschitz constant $\gamma$ of the the gradient of smooth convex function $\nabla p$ depends on the maximum eigenvalue of $H^T H$;
$y_1 \leftarrow \beta_0 \in R^n$ and $t_1 \leftarrow 1$;
for $(i = 0; i \le size; i++)$ do
    Calculate $\beta_k$ using Equation (11) and Equation (12):
        $\beta_k \leftarrow p_L(y_k) \leftarrow T_\alpha\left(y_k - 2\frac{1}{\gamma}H^T H y_k + 2\frac{1}{\gamma}H^T X\right);$   (11)
        $T_\alpha(x) \leftarrow (|\beta| - \alpha) + \mathrm{sgn}(\beta), \alpha \leftarrow \frac{\lambda}{\gamma}$   (12)
    Calculate $t_{k+1}$ using Equation (13):
        $t_{k+1} \leftarrow \frac{1 + \sqrt{1 + 4t_k^2}}{2}$   (13)
    Calculate $y_{k+1}$ using Equation (14):
        $y_{k+1} \leftarrow \beta_k + \left(\frac{t_k - 1}{t_{k+1}}\right)(\beta_k - \beta_{k-1})$   (14)
end for
return $\beta_k$.
```

During each forward feature representation layer, the input raw data should be firstly transformed into an ELM random feature space, which can help to exploit hidden information among training samples. Then, a multilayer unsupervised learning is performed to eventually obtain the high-level sparse features by the ELM-Based sparse autoencoder algorithm. Mathematically, the output of each hidden layer can be represented as:

$$O_i = G(O_{i-1} \times \beta_{i-1}) \tag{15}$$

where O_i is the output of the ith layer, O_{i-1} is the output of the $(i-1)$th layer, $G(\cdot)$ denotes the activation function of the hidden layers, and β represents the output weights of the $(i-1)$th hidden layer. Unlike the existing DL frameworks [37], [38], where all the hidden layers are put together as a whole system, each hidden layer of H-ELM is an independent module, and functions as a separated feature extractor. Once the feature of the previous hidden layer is extracted, the weights or parameters of the current hidden layer will be fixed, and do not need to be fine-tuned.

After multilayer unsupervised feature learning, the resultant outputs of the Kth layer O_K, are viewed as the high-level features extracted from the input data. When used for classification, they are randomly perturbed, and then utilized as the inputs of the supervised ELM to obtain the final results of the whole network. The overall algorithm of training H-ELM framework is shown in Algorithm 3.2.

TABLE 1
Parallelization Suboperations

Subprocess	Suboperations	Description
SAT	H_i	Calculate H using Equation (3) by O_{i-1}
	${H_i}^T H_i$	Large matrix - large matrix multiplication
	${H_i}^T O_{i-1}$	Large matrix - large matrix multiplication
	$O_i = O_{i-1}\beta_i$	Large matrix - small matrix multiplication
OET	H_m	Calculate H using Equation (3) by O_m
	${H_m}^T H_m$	Large matrix - large matrix multiplication
	${H_m}^T T$	Large matrix - large matrix multiplication
SAP	$O_i = O_{i-1}\beta_i$	Large matrix (rows are enormous) - small matrix multiplication
OEP	H_m	Calculate H using Equation (3) by O_{i-1}
	$H_m\beta_m$	Large matrix - small matrix multiplication

Algorithm 3.2 H-ELM Framework for Training

```
Input: The training dataset matrix $X$: $\{(x_i)|x_i \in R^d, i = 1, ..., N\}$;
    The number of hidden layer for sparse autoencoder: $m$;
Output: Output weight vector of each layer: $\beta_i$;
Randomly generate hidden node matrix for each layer for sparse autoencoder $V_i, i = 1, ..., m$;
Randomly generate hidden node matrix for original ELM $V_{m+1}$;
$O_0 \leftarrow X$;
for $(i = 0; i < m; i++)$ do
    Calculate hidden weight vector $\beta_i$ by Algorithm 3.1 using $O_{i-1}$ and $V_i$ as its parameters;
    $O_i \leftarrow O_{i-1} \times \beta_i$;
end for
Calculate output weight vector $\beta_{m+1}$ by Equation (8) using $O_m$ and $V_m$ as its parameters;
return $\beta_i, i = 1, ..., m + 1$.
```

4 Parallel H-ELM Framework with Flink

In this section, we parallelize the training process and prediction process of H-ELM framework on Flink named as PH-ELM, taking advantage of the distributed in-memory computing platform. Flink is designed as a popular processing platform that is suitable for big data mining. The key programming model of Flink is the abstract distributed DataSet (DST), which is similar with Resilient Distributed Dataset (RDD) [13]. DST represents a collection of distributed items, which can be manipulated across many computing nodes concurrently. Programmers can define a series of user-defined actions (e.g., Map, Reduce, Join, Group) for the DSTs. Due to the fact that Spark and Flink are quite similar, our design can be easily integrated into Spark.

4.1 Selective Parallelization

Among many suboperations in training process described in Algorithm 3.1, Algorithm 3.2 and suboperations in prediction process, which operations should we parallelize? A naive approach is to parallelize all the operations. However, some operations run more quickly on a single machine rather than on multiple machines in parallel. That is because the overhead incurred by using distributed computing exceeds gains made by parallelizing the task. Therefore, simple tasks where the input data is very small are carried out faster on a single machine. Thus, we divide the suboperations into two groups: those to be parallelized and those to be run in a single machine.

As discussed in Section 3, there are four subprocesses in H-ELM framework: sparse autoencoder in training process (SAT); original ELM in training process (OET); sparse autoencoder in prediction process (SAP) and original ELM in prediction process (OEP). We go through the execution process of H-ELM framework to select the suboperations which contain large-scale inputs.

For processing large-scale datasets, N is very large, while l, d is small (N is the number of input data samples; d is the dimension of input data samples; l is the number of hidden nodes). $H^T H$ is an $l \times l$ matrix, H^T is an $l \times N$ matrix, H is an $N \times l$ matrix, while O is an $N \times m$ matrix. The computation cost of calculation shown in lines $5 - 10$ of Algorithm 3.1 is small. In addition, after the calculation of $H^T H$ and $H^T T$, the computation cost of the line 3 of Equation (8) is also small. Therefore, we implement these operations on a single machine.

Through deep analysis, all the suboperations of H-ELM framework which are required to be parallelized are listed in Table 1. These suboperations can be divided into three types: calculation of H, multiplication of H^T and another large matrix, multiplication of a large matrix and small matrix β.

4.2 PH-ELM Overview

According to the three types of suboperations presented above, we have proposed three basic parallel algorithms on Flink and adopted a series of optimizations to improve the performance. The parallel training process and prediction process of H-ELM are based on these three basic parallel algorithms: cache-based parallel hidden layer output matrix (CPHOM), cache-based parallel β matrix multiplication (CPBMM), and adaptive transpose hidden matrix multiplication (ATHMM).

The workflow of parallel H-ELM framework on Flink (PH-ELM) is as presented in Figure 1. During PH-ELM, the executions of both sparse autoencoder layers and the original ELM layer are parallelized on Flink. Before the training process, the data to be processed is stored in HDFS at first. Then it is loaded into Flink's distributed memory system as a DST object. In the meantime, the hidden node parameters of all the autoencoder layers and the original ELM layer are generated randomly. Suppose that there are m sparse autoencoder layers. Through our proposed parallel H-ELM framework which are based on our proposed three basic algorithms, hidden weight vectors $\beta_{i,...,m}$ of all autoencoder layers and β_{m+1} of the original ELM layer are

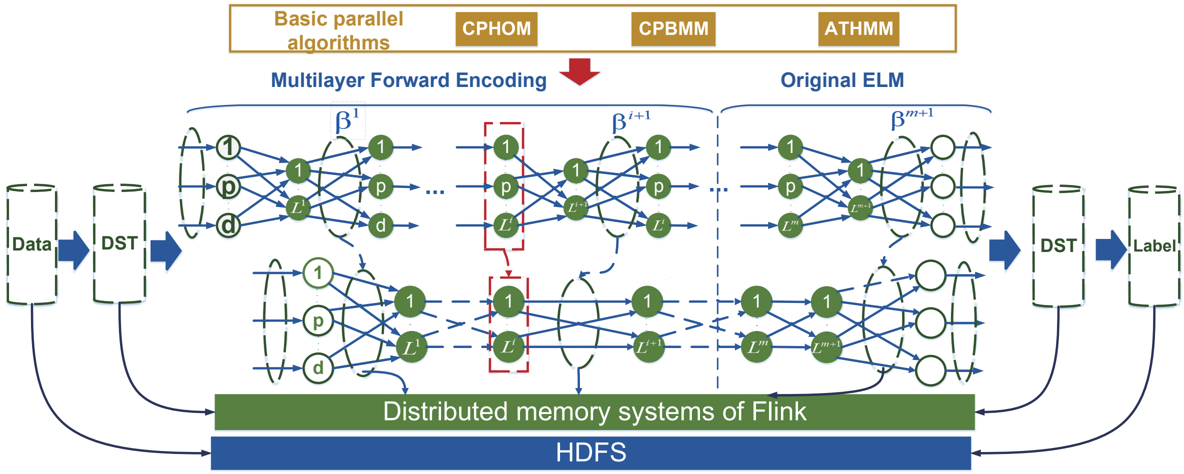
Basic parallel algorithms
CPHOM
CPBMM
ATHMM
Multilayer Forward Encoding
Original ELM
β1
βi+1
βm+1
Data
DST
DST
Label
Distributed memory systems of Flink
HDFS

Fig. 1. Parallel H-ELM on Flink (PH-ELM) overview.

calculated. These hidden weight vectors are stored in Flink's distributed memory system as DST objects. They can be also written into HDFS.

During the prediction process, the label of the data to be predicted is calculated by the hidden weight vectors $\beta_{i,\ldots,m+1}$ and the randomly generated hidden node parameters in a forward way.

4.3 Implementation Details of Basic Parallel Algorithms

4.3.1 *Adaptive Partition Scheme and Storage Formats*

Minimizing the volume of data exchanged between nodes is important to design efficient distributed algorithms. Generally speaking, there are two basic types of strategies for partitioning a matrix, which are based on submatrices and rows. A row may be split into different blocks using the strategies based on submatrices, leading to the need for accumulating computing results from different workers. Actually, the communication and synchronization among nodes are costly in the cluster computing environment. Therefor, partitioning the matrices based on submatrices will bring high overhead so that it is not appropriate in big data environment. On the contrary, a row does not need to be split into different blocks in rows-based strategies. The accumulation of intermedia results can be calculated locally in one worker so that there is no need of shuffling the intermedia results across different nodes. According to the above analysis, the best choice is to partition the matrix based on rows so that the multiplication of every row of the first matrix and the column of the second matrix is executed as a whole to decrease the overhead caused by shuffle phase. However, if the row of the first matrix is so large (e.g. 5G, 20G), it should be partitioned into different blocks.

According to the above analysis, we propose rows-based format named as RDST and columns-based format named as CDST if the rows do not need to be split. During RDST, the matrix is partitioned by rows, and each row is expressed by the pair with row index as its key and a vector as its value. While during CDST, the matrix is partitioned by columns and each column takes column index as its key and a vector as its value. To further improve the performance, we can combine several rows or columns into a group, which are named as blocked RDST (BRDST) or blocked CDST (BCDST). If the content of each row is too large, it then needs to be split. Therefore, a split rows-based format named as SRDST and a split columns-based format named as SCDST are proposed. RDST, BRDST and SRDST are presented in Figure 2. All these formats are based on Flink's abstract model DataSet (DST) to form the distributed in-memory datasets.

4.3.2 *CPHOM: Cache-based Parallel Hidden Layer Output Matrix*

In the calculation of the matrix H, the naive method is to join matrix elements with parameters of the hidden nodes in the Map and Shuffle stage, and then execute $G(a_i, b_i, x)$ in the Reduce stage. However, the Shuffle phase of this method involves large-scale communication overhead. Generally speaking, the parameters (w_i, b_i) of the hidden nodes are small. CPHOM utilizes the fact that the small dataset can fit into a machine's main memory, and can be distributed to all the Mappers by the distributed cache functionality of Flink. The advantage of the small dataset being available in Mappers is that, during the calculation of H, the execution of $G(a_i, b_i, x)$ can be done inside the Mappers, and the Shuffle phase can be omitted, thus greatly improving the performance.

Algorithm 4.1 provides the pseudo code of calculating the hidden layer output matrix. The input dataset is stored in the Flink's distributed cache as a DataSet object M in a sequence of $< key, value >$ pairs, each of which represents a sample in the datasets, where key represents the index of the sample and $value$ represents the content of the sample. The hidden parameters (w_i, b_i) of the hidden nodes are generated and then are created as a DST object P. We define a Map function named as *CPHOMMap* for DST M with P as its broadcast variable. Therefore, the hidden parameters are distributed to the nodes of the cluster by the distributed cache functionality of Flink. In the *CPHOMMap* function, $G(a_i, b_i, x)$ is executed for each sample.

As discussed in Section 4.3.1, to improve the performance of the matrix multiplication, the best choice is to partition the matrix H^T by rows. That is to say, we need to partition the matrix H by columns. To partition the matrix H^T

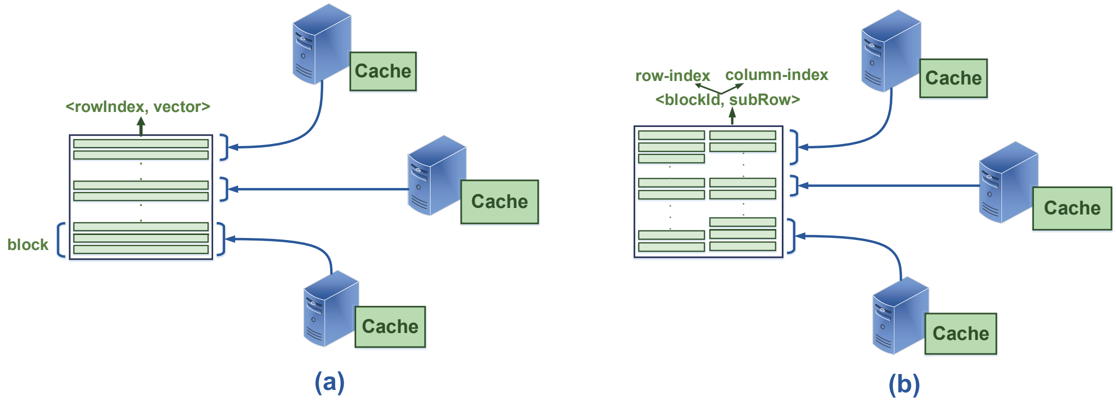

Fig. 2. Proposed format. (a) rows based DST (RDST) and blocked rows based DST (BRDST) (b) split rows based format DST (SRDST) .

in RDST format, the output of the Map function will be $< colIndex, hValue >$, where $colIndex$ is the key which represents the column index, $hValue$ represents one column. To partition the matrix H^T to BRDST format, the output of the *Map* function will be $< blockID + colIndex, hValue >$. After the Mapper, *groupBy* function is executed to partition the matrix to the corresponding format.

Algorithm 4.1 CPHOM Algorithm

Input: A distributed DST object which contains a sequence of pairs in the form of $< index, sample >$ pair, where the the $index$ is the key which represents the index of the $sample$, $sample$ is the value which represents the content of the sample.

Output: $< key, hValue >$: key is the index of the content in the hidden output matrix, $hValue$ represents the content in the hidden output matrix.

```
CPHOM($DataSet < index, sample > M$) //main function
Randomly generate the hidden node parameters $(a_i, b_i)$;
Create DST object $P$ from $(a_i, b_i)$;
Set $P$ as a Broadcast Variable;
$H \leftarrow M$.map(new CPHOMMap().withBroadcastSet($P$)
    .groupBy(0);
return $H$.

CPHOMMap($< index, sample >$) //map function
Init $h$
$(x, t) \leftarrow$ parse($sample$)
for $(i = 0; i \leq l; i++)$ do
    $h \leftarrow G(a_i, b_i, x)$;
    if partition $H^T$ by rows then
        return ($i$, $h$);
    else
        Get $blockIndex$ from $i$;
        $key \leftarrow blockIndex + ',' + i$;
        return ($key$, $h$);
    end if
end for
```

4.3.3 CPBMM: Cache-based Parallel β Matrix Multiplication

To calculate the multiplication of a large matrix and the small matrix β, such as $O_{i-1}\beta_i$ and $H_m\beta_m$, a cache-based parallel β matrix multiplication named as CPBMM algorithm is proposed. In general, matrix-matrix multiplication is very expensive. A standard, yet naive way of multiplying two matrices A and B in MapReduce is to join the corresponding elements of A and B together at first. And then the joined pair is multiplied in Mappers. After that, a Shuffle phase and a Reduce phase are executed to sum the intermediate results. This naive algorithm is very inefficient since it generates huge communication overhead and occupies huge storage spaces. Fortunately, when one of the matrices is very small, we can distribute one matrix using the distributed cache functionality provided by Flink.

As for the the multiplication of a large matrix and small matrix β as presented in Table 1, the content of rows are generally small. Therefore, we can partition the large matrix by rows in the form of RDST or BRDST. Take RDST as a example, a *Map* function is defined which takes the pair $< rowIndex, vector >$ as its input. The user-defined Map function is employed to the RDST object. During each Mapper, the pair is multiplied by all the columns of the β.

4.3.4 ATHMM: Adaptive Transpose Hidden Matrix Multiplication

In terms of the large matrix - large matrix multiplication, such as ${H_i}^T H_i$, ${H_i}^T O_{i-1}$, ${H_m}^T H_m$ and ${H_m}^T T$ as presented in Table 1, an adaptive algorithm is proposed. Let us take $A \times B$ as an example. If one of these two matrices is smaller than a threshold so that one matrix is appropriate to be stored in a single computer node, CPBMM algorithm can be utilized. While, if these two matrices are so large that it is inappropriate to cache them in a single node, these two matrices are needed to be partitioned. As discussed in Section 4.3.1, the best choice is to partition A by rows as the RDST format and B by columns as the CDST format. To further improve the performance, we can combine several rows of A into a group (BRDST format), and combine several columns of B into a group (BCDST format). Let us take RDST and CDST as an example. Firstly, each row of A is joined with all the columns of B. After that, the user-defined Mappers are invoked to multiply the row vector of A and the column vector B.

However, if the width of A is so large that it is inappropriate to be processed in a single Mapper, we need to split each row of A into blocks. SBDST and SCDST format can be utilized to deal with this situation. Firstly, the corresponding splits of A and B are joined. Then, the user-defined Mappers are used to conduct the vector multiplication. After that, an extra Reduce phase is needed to sum up the intermediate results.

5 GPU-ACCELERATED H-ELM FRAMEWORK

5.1 Combination of Flink and GPUs

As we know, CUDA kernels running on GPUs could only be invoked by host applications or libraries programmed by C/C++ or Python. However, the tasks of Flink are executed in Java virtual machines (JVMs). Therefore, to integrate GPUs into the existing architecture of Flink, the first problem to be solved is to provide an efficient strategy for communication between JVM and GPUs. A number of issues complicate the efficient communication strategy. Firstly, CPUs and GPUs have separate memory spaces, requiring explicit data transfers between CPU and GPU memory. What's worse, during the classical implementations of CUDA programming model, data transfers from the host to GPUs are in the form of buffers. The native solution is to transform the JVM objects to buffers manually, which decreases the performance greatly.

5.1.1 *Architecture*

To overcome the challenges described above, we designed the strategy for combining Flink and GPU carefully. It is an extension of our previous work [16]. Our architecture is based on Flink's cluster computing environment such as HDFS, Job Manager. Java native interface (JNI) is utilized to communicate between JVM and GPUs and invoke the CUDA kernels in work nodes.

Figure 3(a) shows the architecture of a work node. Flink communicates with GPUs by calling JAVA interfaces. While, the JAVA interfaces communicate with CUDA host which is programmed by C/C++ through JNI. CUDA host controls the management of GPUs and invokes CUDA kernels to execute operations on GPUs. Figure 3(b) shows the heterogeneous task management for hybrid CPUs and GPUs. During the programming model of GPGPU, the CPU and GPU work in master-slave mode, with the CPU as master and the GPU as slave. Part of the task can be assigned and executed on the CPU in parallel on the original task slots provided by Flink. The rest of the task can be assigned and be executed on the GPU in parallel with support from the CPU on the GPU task slots.

5.1.2 *Memory Mapping Scheme*

A memory mapping scheme is utilized to avoid manual transformation from JVM objects to buffers, thus improving the performance. We use the fact that, the *Tuple* objects in Flink are stored in the cache of JVM in a sequential way. During our scheme, we transfer the buffers in JVM in raw bytes to the device memory of GPUs by JNI without any modifications. After that, a user-defined *Struct* pointer according to the definition of *Tuple* can be utilized to indicate the content of a *Tuple* as the code presented as follows:

```
#pragma pack (1)
typedef struct Point{
    long  x;
    double    y;
};

Point *ptr;
long  point_x = ptr->x;
double  point_y = ptr->y;
```

5.2 Algorithm Design

CPUs and GPUs have different properties and should take on computing tasks matching their abilities to obtain the maximum benefit. The CPU is good at complexity control and lowering the latency of computing while the GPUs do well in high throughput. In other words, various aspects of each task should be assigned to the CPU and GPU based on their properties to achieve the effect of ”one plus one is larger than two”.

To design efficient algorithms for H-ELM Framework on Flink and GPUs, we need to find out the parts which are appropriate for being accelerated by GPUs at first. And then, the corresponding CUDA kernels will be developed. After that, we need to write programs about transferring the data to be processed in Flink to GPUs, invoking the kernels by JNI in an asynchronous model, and transferring the results from GPUs to Flink. Through analysing the parallel parts of H-ELM Framework on Flink, we find that the parts which are time-consuming and requires large-scale computing are those in Mappers and Reducers. Therefore, these parts are accelerated by GPUs to improve the performance. While other parts (e.g., join, group) are in need of large-scale complexity control and memory operations, thus making them inappropriate for GPUs.

5.3 Implementation Details of Acceleration

5.3.1 *Acceleration of CPHOM by GPUs (GCPHOM)*

Through analysing the CPHOM algorithms, we find that the time-consuming subprocess is to apply $G(a_i, b_i, x)$ execution for each sample as described in Algorithm 4.1. As GPUs can process many items concurrently, the *mapPartition* function, which is similar to *Map* function, but runs separately on each partition of the DST, is utilized in our implementation.

Algorithm 5.1 Accelerate CPHOM algorithm by GPUs

```
Input: A distributed DataSet which contains a sequence of
    pairs in the form of $< index, sample >$ pair, where the
    $index$ is the key which represents the index of the $sample$,
    $sample$ is the value which represents the content of the
    sample.
    Distributed cache: parameters of the hidden nodes $(a_i, b_i)$.
Output: $< key, hValue >$: $hValue$ represents one value in the
    hidden output matrix.
MapPartition ($list(index, sample)$) //map function
if There is no free GPU then
    Call the Map function as described in Algorithm 4.1 for
        all the elements in the list;
    return ;
end if
Select an appropriate GPU; //need lock
Transfer the buffers in the list to the selected GPU;
Transfer the buffers of hidden nodes to the selected GPU;
Invoke $cuCPHOM$ kernel to process the data by GPU;
Transfer the partial resulted vector $valueList$ to main mem-
    ory;
Release the selected GPU; //need lock
Convert and insert $valueList$ to $pU$;
return $pU$;
```

Firstly, we check if there is a free GPU or not. If there are no appropriate GPUs, *Map* function is then adopted to execute the operations on CPUs. If there are appropriate GPUs,

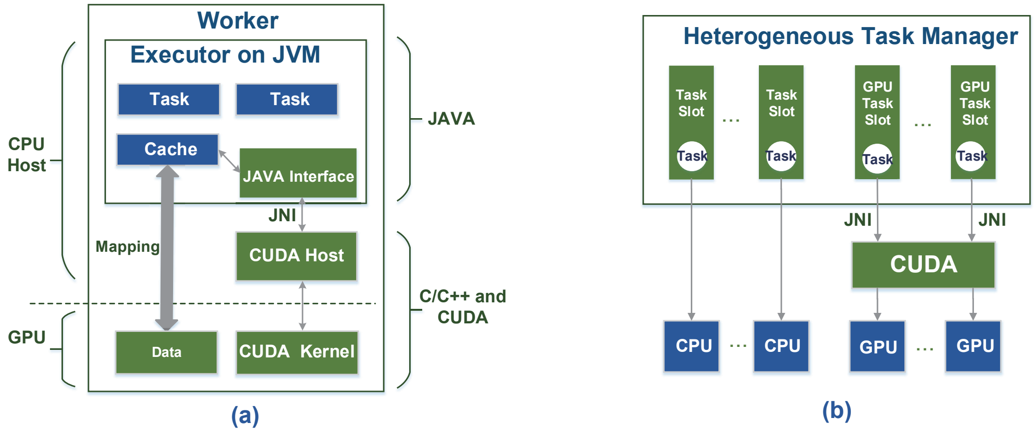

Fig. 3. GFlink architecture (a) Architecture of a work node. (b) Heterogeneous task management.

a GPU is selected and marked as busy. Then the buffers to be processed are transferred to GPUs and the *cuCPHOM* kernel is invoked. After the executions are finished, the contents are transferred from GPUs to the main memory. Lastly, the selected GPU is released.

5.3.2 Acceleration of CPBMM by GPUs (GCPBMM)

In terms of CPBMM algorithm, we find that suboperation which is the most time-consuming and appropriate for being accelerated by GPUs is the multiplication of submatrices and hidden matrix β. The procedure of accelerating CPBMM algorithm by GPUs is as shown in Figure 4. The large matrix is stored in RDST. Firstly, matrix β is broadcasted to all the work nodes in the cluster. Then the map phase is invoked. During the map phase, *mapPartiton* function provided by Flink is implemented, which is similar to *Map* function. By this means, each partition is processed together without the need of processing elements one by one. The CUDA kernel utilized to multiply the submatrices and β is *cublasCgemm* interface in cuBLAS [27].

As shown in in Figure 4(a), we take $A \times B$ as an example. where A is a large matrix and B represents the matrix β. Suppose that there are three workers in the cluster: *worker0* with 1 CPU and 1 GPU, *worker1* with 1 CPU and *worker2* with 1 CPU and 2 GPUs. Firstly, we partition the sparse matrix to RDST format. Suppose that there are 6 partitions (e.g., A_0, A_1, A_2, A_3, A_4, A_5). During the execution, the matrix B is broadcasted to every worker in the cluster. Each *partition* alike to a submatrix of the huge matrix is processed in *mapPartiton* function by a task. In terms of our example, 2 tasks in *worker0*, 1 task in *worker1* and 3 tasks in *worker2* are invoked by the master. As for the executions on GPUs, our implemented kernel is invoked to execute matrix multiplication in one GPU.

5.3.3 Acceleration of ATHMM by GPUs (GATHMM)

In terms of ATHMM, if one row of the matrix is not very large, the first matrix is partitioned by RDST format or BRDST format and the second matrix is partitioned by CDST or BCDST format. We take this format as an example and the work flow is as described in Figure 4(b). For $A \times B$, suppose that A has 2 rows and B has 2 columns. Suppose that there are two workers in the cluster: *worker0* with 1 CPU and 1 GPU, *worker1* with 1 CPU and 1 GPU. Like GCPBMM, to multiply two submatrices in GPUs, *cublasCgemm* interface is adopted.

TABLE 2
Symbols of cost model

Symbol	Definitions
N	The number of samples.
d	The dimension of samples.
o	The dimension of labels.
l	The number of hidden nodes of layers: $1 - m$.
k	The dimension of label.
n	The number of computing nodes in the cluster.
$IOM()$	The cost of memory read and write.
$Network()$	The cost of network transfers.
$Compute()$	The cost of computation.

The rows in the first matrix with the same indexes as the columns in the second matrix are joined together and are grouped into 4 groups. Secondly, these groups are processed in *mapPartiton* function. In terms of our example, 2 tasks in *worker0*, 1 task in *worker1* are invoked by the master. Then, $A_0 \times B_0$ is executed in *CPU0* in *worker0*, $A_0 \times B_1$ in $GPU0$ in $worker0$, $A_1 \times B_0$ in *CPU0* in *worker1*, and $A_1 \times B_1$ in *GPU0* in *worker1* respectively. Then all the partial results are shuffled across networks and accumulated together as a new matrix by *reduceByKey* procedure.

6 Algorithms Analysis

In this section, we build a brief time cost model to evaluate our proposed algorithms. The symbols used in the cost model are defined in Table 2.

6.1 ATHMM Analysis

Suppose that the rows of the first matrix are divided into r blocks, the columns of the first matrix are divided into s blocks, and the columns of the second matrix are divided into t blocks. We take ${H_i}^T H_i$ as an example. As Figure 4(b) shows, the cost of the whole procedure of ATHMM algorithm can be divided into 4 steps:

- **Partition stage**: During the partition stage, each block in matrix ${H_i}^T$ will be emitted t times, while each block in matrix H_i will be emitted r times, thus

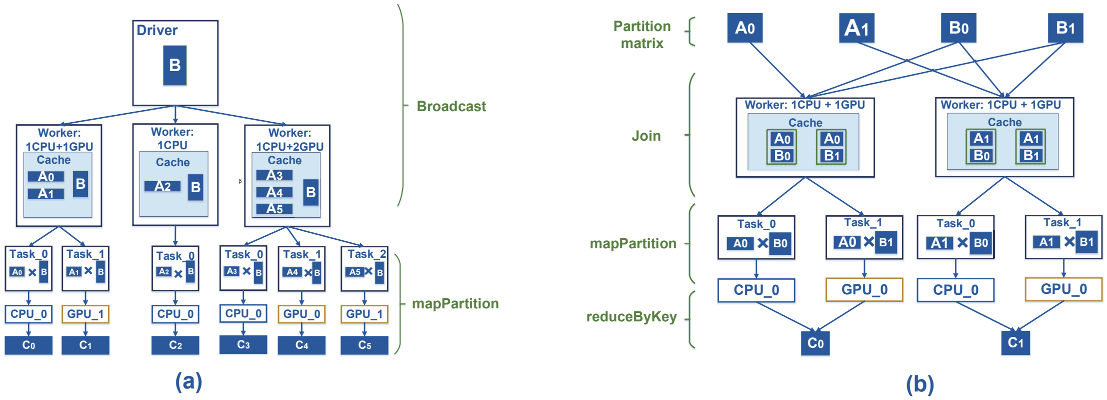

Fig. 4. Acceleration of CPBMM and ATHMM by GPUs (a) GCPBMM (b) GATHMM

the cost can be denoted as $IOM(((t+r)) \times |H_i|) = IOM((t+r) \times N \times l_i)$.

- **Join stage:** During the Join stage, only one matrix (H_i) is shuffled. through the network. The cost spending on the network communication can be derived as $Network(r \times |H_i|) = Network(r \times N \times l_i)$.
- **Map stage:** Once two related submatrices are gathered together by fetching from the network and reading locally, the matrix multiplication would be conducted locally. It is clear that the computing cost of this step is $Compute(N \times l_i \times N)$.
- **Reduce stage:** Finally, during the Reduce stage, the related submatrices are fetched through network, and then $s-1$ addition is performed. The cost is close to $Network(s \times l_i \times N)$.

The cost model of ATHMM algorithm is total cost of these 4 steps:

$$\begin{aligned} Cost(\text{ATHMM}) = IOM&((t+r) \times N \times l_i) \\ &+Network(r \times N \times l_i) \\ &+Compute(N \times l_i \times N) \\ &+Network(s \times l_i \times N) \end{aligned} \tag{16}$$

6.2 CPHOM Analysis

The total execution time of CPHOM consists two parts: broadcasting the hidden parameters (w_i, b_i) of the hidden nodes and the conducting $G(a_i, b_i, x)$ execution in all the nodes. The cost model of CPHOM algorithm can be derived as:

$$\begin{aligned} Cost(\text{CPHOM}) = &Network(2 \times n \times l_{i-1} \times l_i) \\ &+Compute(N \times l_{i-1} \times l_i) \end{aligned} \tag{17}$$

where l_0 is equal to the dimension of samples d.

6.3 CPBMM Analysis

Like CPHOM, firstly, the matrix β is broadcasted to all the computing nodes in the cluster. Then the sub-matrix of the first large matrix is multiplied with β in the computing nodes locally. As for the execution of $O\beta$ and $H\beta$, it is clearly that the computing cost of the second step is $O(N \times l_i \times l_{i+1})$. The cost model of CPBMM algorithm can be derived as:

$$\begin{aligned} Cost(\text{CPBMM}) = &Network(n \times l_i \times l_{i+1}) \\ &+Compute(N \times l_i \times l_{i+1}) \end{aligned} \tag{18}$$

6.4 GPU Acceleration Analysis

During our scheme, GPU is only utilized to accelerate the Compute stage. As for the GPU execution, the data is firstly transferred from the main memory to the device memory of GPUs before being processed in the GPUs. After that, the results are transferred from GPUs to the main memory. The execution time of a GPU can be denoted as:

$$T_g = T_{gm\_data} + T_{gp} + T_{gm\_result} + T_{gf} \tag{19}$$

where T_g represents the total execution time on a GPU, T_{gm_data} refers to the moving data buffers between the CPU and the GPU, T_{gp} represents the real execution time on a GPU, T_{gm_result} represents the moving results from GPUs to the main memory, while T_{gf} denotes the fixed time for invoking GPU.

6.5 Overall Analysis

When processing large-scale datasets, N is much larger than other parameters. Through analysing these three basic algorithms and suboperations, we can find that the bottleneck of PH-ELM and GPH-ELM is ATHMM algorithm (including ${H_i}^T H_i$, ${H_i}^T O_{i-1}$, ${H_m}^T H_m$ and ${H_m}^T T$). The number of hidden nodes has an effect on the execution of ${H_i}^T H_i$, ${H_i}^T O_{i-1}$ and ${H_m}^T T$. From the cost model of ATHMM, the execution time of ATHMM is linearly with the number of hidden nodes theoretically. The dimension of samples has an impact on the execution of H and ${H_i}^T O_{i-1}$. The influence of dimension is smaller than that of the number of hidden nodes. However, the dimension of labels merely affect the performance of ${H_m}^T T$. Due to the fact that one row of matrix T only contains one element, the dimension of labels has little influence on the overall performance.

7 Experiments

In this section, some experiments are conducted to evaluate the performance of PH-ELM and GPH-ELM. Firstly, a series of experiments are conducted to evaluate the execution efficiency by comparing them with the original H-ELM framework in terms of the average running time and speedup. Then, for comparison, we also evaluate the performance of other parallel ELM algorithms. Lastly, the performance of our three basic parallel algorithms, and the effects of the

TABLE 3
Datasets from the UCI machine learning repository

Datasets	Instances	Dimensions	Classes
MINST	60000	784	10
GISETTE	6000	5000	2
ADULT	48842	14	2
WINE	178	13	3

acceleration by GPUs are evaluated in detail by comparing them with other distributed matrix algorithms.

7.1 Experimental Setup

All the experiments are performed on a Flink cluster, in which each test computer in the cluster is equipped with one Intel(R) Core(TM)i5-4590 CPU which contains 4 cores running at 3.30 GHz , 12GB memory and 2 NVIDIA GeForce GTX 750 GPUs. Each GPU has 512 CUDA processor cores, working on 1020 MHz clock and 1 GB global memory with 128 bits bus. As for the software, the test machine runs in the UBUNTU 14.04, NVIDIA CUDA toolkit 7.5 and Flink 0.10.1. The datasets used in the experiments are from the UCI machine learning repository as shown in Table 3.

7.2 Results Overview

This section presents the results of the overall performance of our proposed PH-ELM and GPH-ELM algorithms. The performance speedup on small datasets is presented in Figure 5(a). For a clear observation of the results, three groups of dataset are utilized: small data samples, medium data samples and large-scale data samples. Four sparse encoder layers and the original ELM layer are utilized. The number of hidden nodes of all layers is set as $1,000$.

During the experiment for small data samples, the number of instances of all datasets is $50,000$ created by replicating the original datasets and the cluster contains 10 computing nodes. Both PH-ELM and GH-ELM do not get an ideal speedup. That is because, in terms of small datasets, the communication overhead in PH-ELM and GH-ELM greatly affects the efficiency of the parallelization. The running time of serial executions in PH-ELM and GPH-ELM also occupies a large proportion. Moreover, there also exists a fixed time for the application submission and configuration, it is reasonable that the whole speedup of processing small datastes is small. Figure 5(b) presents the speedup of the medium dataset with $400,000$ instances. We can see that the speedup of all these 4 datasets is higher than the speedup with small data samples. The speedup of PH-ELM achieves almost $4\times$, while GPH-ELM achieves almost $9\times$.

Figure 5(c) shows the average running time of large-scale training samples with $2,000,000$. H-ELM even fails to finish because of limited memory. Therefore, we cannot show the speedup of PH-ELM and GH-ELM. From this figure, we can find that the average running time of GISETTE dataset is longer than that of other datasets. That is because, the dimension of GISETTE dataset is the largest. However, the running time of GISETTE dataset is not much larger than that of other datasets. That is because, as discussed in Section 6, different dimensions of samples just affect parts of the execution of one hidden layer.

7.3 Rseults Under Different Circumstances

7.3.1 Results for Different Sizes of Records

Figure 6(a) illustrates the average running time of MINST. The number of samples is increased from $1,000,000$ to $5,000,000$ gradually. 4 sparse encoder layers and the original ELM layer are utilized, and the number of hidden nodes of all layers is set as $1,000$. H-ELM even fails to finish because of limited memory. From this figure, we can find that GPH-ELM has higher speedup (about $3\times$ to $4\times$) over PH-ELM than processing small datasets. As for PH-ELM and GPH-ELM, the running time increases faster than the increase of the number of instances.

7.3.2 Results for Different Numbers of Slave Nodes

In this case, the effects of our proposed PH-ELM and GPH-ELM under different numbers of slave nodes are evaluated. The number of data samples is $5,000,000$, both the number of the hidden nodes of the sparse autoencoder layers and the original ELM layer are set as $1,000$. The number of slave nodes is increased from 10 to 50. Figure 6(b) illustrates the average running time under different numbers of slave nodes. From this figure, we can find that the average running time of PH-ELM and GPH-ELM algorithm decreases basically in a linear manner with the increase of the number of slave nodes. It demonstrates that our proposed algorithms have good scalability.

7.3.3 Results for Different Numbers of Hidden Nodes

In this section, the effects of different numbers of the hidden nodes on the test results are examined. The number of slave mode computers is set as 10 and the size of records is $1,000,000$. The number of hidden nodes of all layers is increased from $1,000$ to $5,000$ gradually. For different numbers of the hidden nodes, the time consumed by PH-ELM algorithm and GPH-ELM algorithm are shown in Figure 6(c). We can see that the consumed time almost grows linearly along with the increase of the number of the hidden nodes, which is in accord with the analysis in Section 6.

7.3.4 Comparison Results for Other Parallel ELM Algorithms

To the best of our knowledge, our proposed algorithm is the first kind of distributed versions of H-ELM. Therefore, in this section, we compare the performance of the original ELM layer of our proposed PH-ELM and GPH-ELM with other parallel ELM algorithms, including PELM [20] and ELM* [22] which are both built on top of Hadoop. The number of hidden is 200, the cluster has 10 slave mode computers and each dataset contains $1,000,000$ samples. From the comparison results as presented in Figure 7, we can find that our proposed algorithms outperform both PELM and ELM*. That is because, every MapReduce in Hadoop needs to interact with other stages through the Hadoop Distributed File System (HDFS), which costs much time for disk I/O operations. It can be estimated that the speedup of our proposed approaches will increase as the number of MapReduce stage increases. As we know, many MapReduce stages are required in H-ELM. That is an important reason why we choose Flink as the platform rather than Hadoop. Moreover, PELM and ELM* cannot utilize the high computing power of GPUs in the cluster.

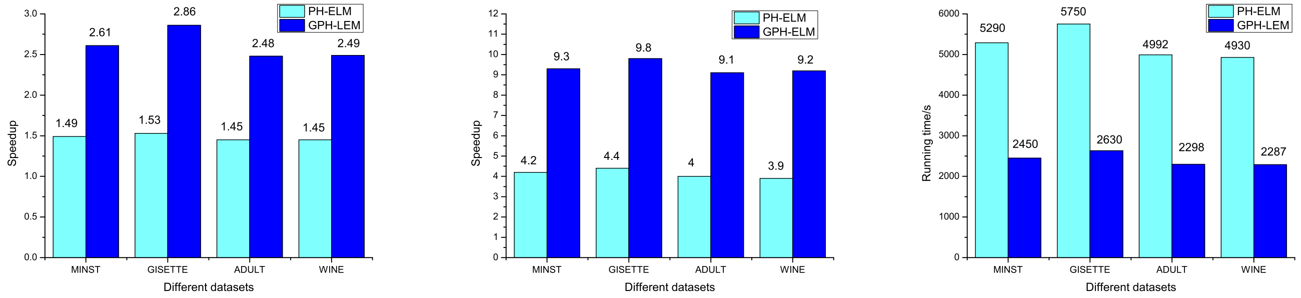

Fig. 5. Performance results overview. (a) Performance speedup of small datasets. (b) Performance speedup of medium datasets. (c) Average running time of large-scale datasets.

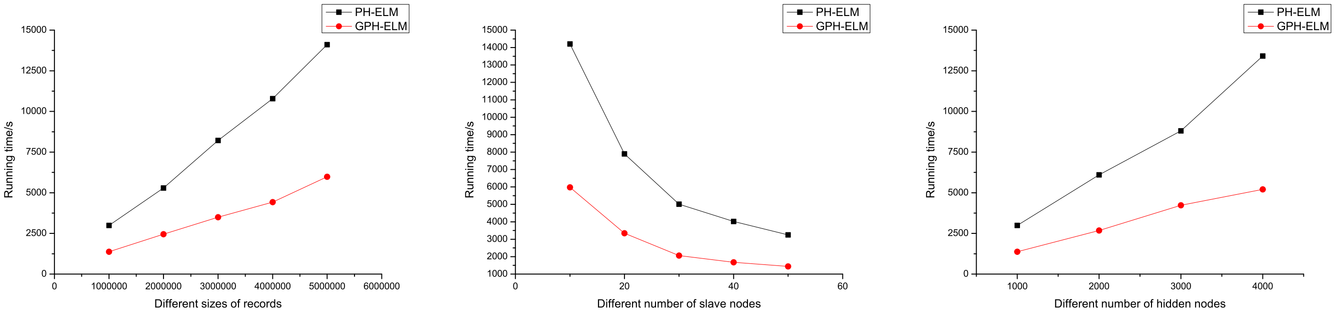

Fig. 6. Performance results under different circumstances. (a) Average running time under different data sizes. (b) Average running time under different numbers of slave nodes. (c) Average running time under different numbers of hidden nodes.

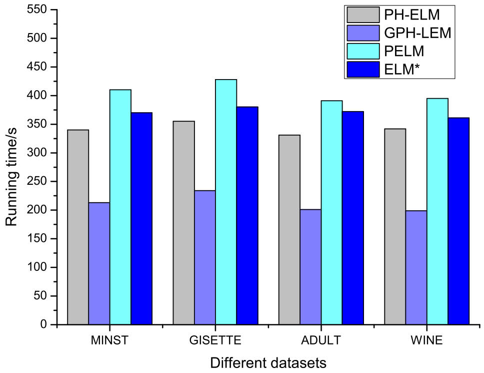

Fig. 7. Performance comparison with PELM and ELM*.

7.4 Results for Basic Parallel Algorithms

In this section, we firstly evaluate the results of our CPBMM, ATHMM, GCPBMM and GATHMM on a Flink cluster with 20 computing nodes by comparing them with other distributed matrix systems, including SUMMA and HAMA. SUMMA is a popular MPI-based matrix computation algorithm, while HAMA is a famous matrix computation library based on Hadoop. MINST dataset is utilized. As for the CPBMM algorithm, $H_m\beta_m$ is taken as an example, where the number of hidden nodes is set as $1,000$. While in ATHMM algorithm, H^TH is implemented, where BRDST is utilized for H^T and BCDST is utilized for H. The number of records increases from $50,000$ to $250,000$, and the number of the hidden nodes is $1,000$.

Figure 8 illustrates the comparison of CPBMM and GCPBMM with SUMMA and HAMA, while Figure 9 presents the comparison results of ATHMM and GATHMM. We find that the performance of CPBMM and ATHMM are comparable to SUMMA. Our GPU-accelerated algorithms GCPBMM and GATHMM are much faster than SUMMA. It can also be seen that the our proposed algorithms are much faster than HAMA. That is because, compared with our proposed algorithms, HAMA has many network synchronization operations and I/O overhead.

Then, we evaluate the results of our ATHMM and GATHMM algorithms on different GPUs (including NVIDIA C2050, NVIDIA GeForce GTX 750 and NVIDIA K40) on a single node. The number of hidden nodes is set as $1,000$, and the number of records is $10,000$. Figure 10 shows that the executions on K40 has the highest speedup, while the performance on C2050 and GTX 750 is almost the same. Therefore, our GPU-accelerated algorithms will get a higher speedup if the GTX 750 is replaced with K40.

8 Conclusion

The proposed H-ELM has adopted a novel MLP training scheme based on the universal approximation capability of the original ELM, which achieves high-level representation with layerwise encoding, and outperforms the original ELM in various simulations. However, the capability of utilizing H-ELM to process large-scale datasets is an urgent and challenging issue confronting researchers. This paper has proposed an efficient parallel algorithm based on Flink named as PH-ELM, benefiting from the high performance, good reliability and expandability of in-memory cluster computing. During PH-ELM, several optimizations have been adopted to improve the efficiency and scalability of

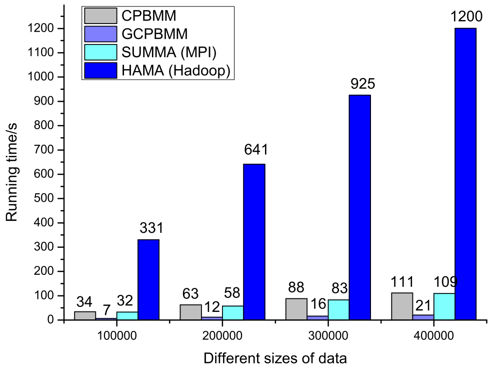

Fig. 8. Performance comparison of CPBMM and GCPBMM.

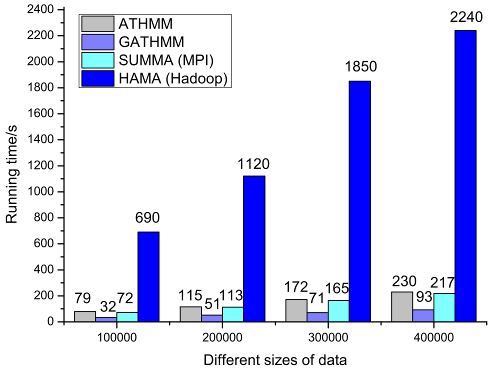

Fig. 9. Performance comparison of ATHMM and GATHMM.

parallelism. To further improve the performance, the existing high computing power of GPUs is leveraged to accelerate the PH-ELM. Experiments have demonstrated that our proposed PH-ELM and GPH-ELM are able to process large-scale datasets, with excellent performance in speedup and scalability.

Acknowledgments

The authors would like to express their gratitude to three anonymous reviewers for their constructive comments which have helped to improve the quality of the manuscript. The research was partially funded by the Key Program of National Natural Science Foundation of China (Grant Nos. 61133005, 61432005), the National Natural Science Foundation of China (Grant Nos. 61370095, 61472124), and the International Science & Technology Cooperation Program of China (Grant No. $2015DFA11240$), the Key Technology Research and Development Programs of Guangdong Province (Grant No. $2015B010108006$).

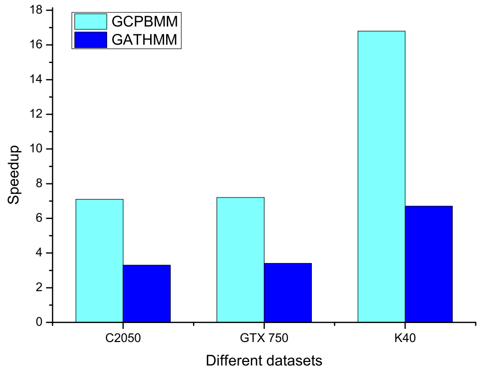

Fig. 10. Performance comparison of different GPUs.

References

[1] J. Kim and W. Lee, "Stochastic decision making for adaptive crowdsourcing in medical big-data platforms," *IEEE Transactions on Systems Man & Cybernetics Systems*, vol. 45, no. 11, pp. 1471–1476, 2015.

[2] L. Liu and H. Jia, "Trust evaluation via large-scale complex service-oriented online social networks," *IEEE Transactions on Systems Man & Cybernetics Systems*, vol. 45, no. 11, pp. 1–1, 2015.

[3] G.-B. Huang, H. Zhou, X. Ding, and R. Zhang, "Extreme learning machine for regression and multiclass classification," *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on*, vol. 42, no. 2, pp. 513–529, 2012.

[4] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: a new learning scheme of feedforward neural networks," in *Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on*, vol. 2. IEEE, 2004, pp. 985–990.

[5] G. B. Huang, Q. Y. Zhu, and C. K. Siew, "Extreme learning machine: theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, 2006.

[6] G. B. Huang, L. Chen, and C. K. Siew, "Universal approximation using incremental constructive feedforward networks with random hidden nodes." *IEEE Transactions on Neural Networks*, vol. 17, no. 4, pp. 879–92, 2006.

[7] G. B. Huang, M. B. Li, L. Chen, and C. K. Siew, "Incremental extreme learning machine with fully complex hidden nodes," *Neurocomputing*, vol. 71, no. 4-6, pp. 576–583, 2008.

[8] J. Tang, C. Deng, and G. B. Huang, "Extreme learning machine for multilayer perceptron," *IEEE Transactions on Neural Networks & Learning Systems*, vol. 27, no. 4, pp. 809–821, 2016.

[9] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[10] Y. Xun, J. Zhang, and X. Qin, "Fidoop: Parallel mining of frequent itemsets using mapreduce," *IEEE Transactions on Systems Man & Cybernetics Systems*, vol. 46, no. 3, pp. 1–1, 2015.

[11] W. P. Ding, C. T. Lin, M. Prasad, and S. B. Chen, "Attribute equilibrium dominance reduction accelerator (dccaedr) based on distributed coevolutionary cloud and its application in medical records," *IEEE Transactions on Systems Man & Cybernetics Systems*, vol. 46, no. 3, pp. 384–400, 2016.

[12] "Flink programming guide," http://flink.apache.org/, 2016, online; accessed 1-July-2016.

[13] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2.

[14] "cudnn," https://developer.nvidia.com/cudnn, 2016, online; accessed 1-July-2016.

[15] A. Coates, B. Huval, T. Wang, D. J. Wu, A. Y. Ng, and B. Catanzaro, "Deep learning with cots hpc systems," in *International Conference on Machine Learning*, 2013, pp. 1337–1345.

[16] C. Cen, L. Kenli, O. Aijia, T. Zhuo, and L. Keqin, "Gflink: An in-memory computing architecture on heterogeneous cpu-gpu clusters for big data," in *International Conference on Parallel Processing*, 2016.

[17] N.-Y. Liang, G.-B. Huang, P. Saratchandran, and N. Sundararajan, "A fast and accurate online sequential learning algorithm for feedforward networks," *IEEE Transactions on Neural networks*, vol. 17, no. 6, pp. 1411–1423, 2006.

[18] H.-J. Rong, G.-B. Huang, N. Sundararajan, and P. Saratchandran, "Online sequential fuzzy extreme learning machine for function approximation and classification problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 4, pp. 1067–1072, 2009.

[19] G. Huang, S. Song, J. N. Gupta, and C. Wu, "Semi-supervised and unsupervised extreme learning machines," *IEEE transactions on cybernetics*, vol. 44, no. 12, pp. 2405–2417, 2014.

[20] Q. He, T. Shang, F. Zhuang, and Z. Shi, "Parallel extreme learning machine for regression based on mapreduce," *Neurocomputing*, vol. 102, pp. 52–58, 2013.

[21] B. Wang, S. Huang, J. Qiu, Y. Liu, and G. Wang, "Parallel online sequential extreme learning machine based on mapreduce," *Neurocomputing*, vol. 149, pp. 224–232, 2015.

[22] J. Xin, Z. Wang, C. Chen, L. Ding, G. Wang, and Y. Zhao, "Elm*: distributed extreme learning machine with mapreduce," *World Wide Web*, vol. 17, no. 5, pp. 1189–1204, 2014.

[23] R. A. V. D. Geijn and J. Watts, "Summa: scalable universal matrix multiplication algorithm," *Concurrency and Computation: Practice and Experience*, vol. 9, no. 4, p. 255C274, 1997.

[24] D. Schmidt, G. Ostrouchov, W. C. Chen, and P. Patel, "Tight coupling of r and distributed linear algebra for high-level programming with big data," 2012, pp. 811–815.

[25] J. Choi, J. J. Dongarra, R. Pozo, and D. W. Walker, "Scalapack: a scalable linear algebra library for distributed memory concurrent computers," in *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, 1992, pp. 120–127.

[26] S. Seo, E. J. Yoon, J. Kim, S. Jin, J. S. Kim, and S. Maeng, "Hama: An efficient matrix computation with the mapreduce framework," in *Cloud Computing, Second International Conference, CloudCom 2010, November 30 - December 3, 2010, Indianapolis, Indiana, USA, Proceedings*, 2010, pp. 721–726.

[27] "Cublas programming guide," http://docs.nvidia.com/cuda/cublas/index.html, 2016, online; accessed 1-November-2016.

[28] "Cusparse programming guide," http://docs.nvidia.com/cuda/cusparse/index.html, 2016, online; accessed 1-November-2016.

[29] W. Yang, K. Li, Z. Mo, and K. Li, "Performance optimization using partitioned spmv on gpus and multicore cpus," *IEEE Transactions on Computers*, vol. 64, no. 9, pp. 2623–2636, 2015.

[30] T. Poggio and F. Girosi, "Networks for approximation and learning," *Proceedings of the IEEE*, vol. 78, no. 9, pp. 1481–1497, 1990.

[31] H. White, *Artificial neural networks: approximation and learning theory*. Blackwell Publishers, Inc., 1992.

[32] G.-B. Huang and L. Chen, "Convex incremental extreme learning machine," *Neurocomputing*, vol. 70, no. 16, pp. 3056–3062, 2007.

[33] G. Huang, G.-B. Huang, S. Song, and K. You, "Trends in extreme learning machines: a review," *Neural Networks*, vol. 61, pp. 32–48, 2015.

[34] Y. Bengio, "Learning deep architectures for ai," *Foundations and trends® in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[35] J. Tang, C. Deng, and G. B. Huang, "Extreme learning machine for multilayer perceptron." *IEEE Transactions on Neural Networks & Learning Systems*, p. 1, 2015.

[36] A. Beck and M. Teboulle, "A fast iterative shrinkage-thresholding algorithm for linear inverse problems," *SIAM journal on imaging sciences*, vol. 2, no. 1, pp. 183–202, 2009.

[37] Y. Bengio, A. Courville, and P. Vincent, "Representation learning: A review and new perspectives," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 8, pp. 1798–1828, 2013.

[38] G. E. Hinton, S. Osindero, and Y.-W. Teh, "A fast learning algorithm for deep belief nets," *Neural computation*, vol. 18, no. 7, pp. 1527–1554, 2006.

Cen Chen is currently a PhD candidate in Computer Science, Hunan University, China. His research interest includes parallel and distributed computing systems, machine learning on big data. He has published several research articles in international conference and journals of machine learning algorithms and parallel computing.

Kenli Li received the PhD degree in computer science from the Huazhong University of Science and Technology, China, in 2003. He was a visiting scholar at the University of Illinois at Urbana- Champaign from 2004 to 2005. He is currently a full professor of computer science and technology at Hunan University and deputy director of the National Supercomputing Center in Changsha. His major research areas include parallel computing, high-performance computing, and grid and cloud computing. He has published more than 130 research papers in international conferences and journals such as *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, *Journal of Parallel and Distributed Computing*, *ICPP*, *CCGrid*. He is an outstanding member of CCF. He is a senior member of the IEEE and serves on the editorial board of the *IEEE Transactions on Computers*.

Aijia Ouyang received the PhD degree in Computer Science, Hunan University, China, in2015.His research interests include parallel computing, cloud computing and big data. He has published more than 20 research papers in international conference and journals of intelligence algorithms and parallel computing.

Zhuo Tang received the Ph.D. in computer science from Huazhong University of Science and Technology, China, in 2008. He is currently an associate professor of the College of Computer Science and Electronic Engineering at Hunan University, and he is the sub-dean of the department of computing science. His majors are distributed computing system, cloud computing, and the parallel process for big data. He is a member of ACM and CCF.

Keqin Li is a SUNY Distinguished Professor of computer science. His current research interests include parallel computing and high-performance computing, distributed computing, energy-efficient computing and communication, heterogeneous computing systems, cloud computing, big data computing, CPU-GPU hybrid and cooperative computing, multicore computing, storage and file systems, wireless communication networks, sensor networks, peer-to-peer file sharing systems, mobile computing, service computing, Internet of things and cyber-physical systems. He has published over 460 journal articles, book chapters, and refereed conference papers, and has received several best paper awards. He is currently or has served on the editorial boards of *IEEE Transactions on Parallel and Distributed Systems*, *IEEE Transactions on Computers*, *IEEE Transactions on Cloud Computing*, *IEEE Transactions on Services Computing*, *IEEE Transactions on Sustainable Computing*. He is an IEEE Fellow.