2014 IEEE 28th International Parallel & Distributed Processing Symposium Workshops

Training Large Scale Deep Neural Networks on the Intel Xeon Phi Many-core Coprocessor

Lei Jin, Zhaokang Wang, Rong Gu, Chunfeng Yuan and Yihua Huang
Department of Computer Science and Technology, Nanjing University
National Key Laboratory for Novel Software Technology, Nanjing University
Nanjing 210023, China
leinking1@gmail.com, wang.zk@foxmail.com, gurongwalker@gmail.com, {cfyuan,yhuang}@nju.edu.cn

***Abstract*—As a new area of machine learning research, the deep learning algorithm has attracted a lot of attention from the research community. It may bring human beings to a higher cognitive level of data. Its unsupervised pre-training step allows us to find high-dimensional representations or abstract features which work much better than the principal component analysis (PCA) method. However, it will face problems when being applied to deal with large scale data due to its intensive computation from many levels of training process against large scale data. The sequential deep learning algorithms usually can not finish the computation in an acceptable time. In this paper, we propose a many-core algorithm which is based on a parallel method and is used in the Intel Xeon Phi many-core systems to speed up the unsupervised training process of Sparse Autoencoder and Restricted Boltzmann Machine (RBM). Using the sequential training algorithm as a baseline to compare, we adopted several optimization methods to parallelize the algorithm. The experimental results show that our fully-optimized algorithm gains more than 300-fold speedup on parallelized Sparse Autoencoder compared with the original sequential algorithm on the Intel Xeon Phi coprocessor. Also, we ran the fully-optimized code on both the Intel Xeon Phi coprocessor and an expensive Intel Xeon CPU. Our method on the Intel Xeon Phi coprocessor is 7 to 10 times faster than the Intel Xeon CPU for this application. In addition to this, we compared our fully-optimized code on the Intel Xeon Phi with a Matlab code running on single Intel Xeon CPU. Our method on the Intel Xeon Phi runs 16 times faster than the Matlab implementation. The result also suggests that the Intel Xeon Phi can offer an efficient but more general-purposed way to parallelize the deep learning algorithm compared to GPU. It also achieves faster speed with better parallelism than the Intel Xeon CPU.**

Keywords:Deep learning;Unsupervised learning;Deep architecture;Sparse autoencoder; Restricted Boltzmann Machine; Parallel algorithm; Intel Xeon Phi;Many-core

I. INTRODUCTION

The term "Deep Learning" has gained great attention and become a landmark in machine learning area since Geoffrey Hinton and Ruslan Salakhutdinov published their paper in Science in 2006 [1].This paper proposed an idea to convert high-dimensional data to low-dimensional data by training a multilayer neural network with a small middle layer to reconstruct the high-dimensional input vectors [1].This low-dimensional data can be viewed as a code or extracted features to make it easier to learn tasks of interests. The so-called "deep" refers to multi-level representations of original data, where higher level representations are defined based on lower representations. The assumption underlying the deep learning is that there exists some invisible hidden structures behind the observed data which are helpful in future tasks. The idea of extracting the hierarchical features also has a biological basis that human visual cortex is hierarchical [2].

Since deep learning algorithms involve feature extraction or learning representations, they are often framed as unsupervised learning. It often uses the artificial neural networks (ANN) to perform unsupervised learning and then piles up many layers of neural networks like a stack. Sparse Autoencoder, Restricted Boltzmann Machine, Sparse Code and many variations of them are usually used as the unsupervised building block [3, 4, 5]. Since constructing labeled data can be very time-consuming and labor-intensive, unsupervised learning has an advantage of using more unlabeled data compared to supervised learning. At the meantime, more data lead to more computations which limits its wide use in real application.

Sparse Autoencoder, Restricted Boltzmann Machine (RBM) and Sparse Code are three of the most commonly used building blocks of deep architectures [3, 6, 7]. By stacking layers of Sparse Autoencoders, we can get stacked Autoencoder. Another deep learning network, Deep Belief Network (DBN), is constructed by stacking many layers of RBMs. Both of them can be considered as neural networks.

Sparse Autoencoder, as its name implies, plays the role of an encoder. It typically consists of three layers of neurons: input layer, hidden layer and output layer, and the number of neurons in the output layer is equal to that in the input layer. Using unsupervised training, we tune the weight of each connection and bias of each neuron so that the reconstructed data from the output of the hidden layer can be as close as to the original input. As a result, the output of the hidden layer can be recognized as a code of the input.

978-1-4799-4116-2/14 $31.00 © 2014 IEEE
DOI 10.1109/IPDPSW.2014.194

RBM is a generative stochastic neural network to learn the probability distribution of its input [8]. It has been applied to many areas effectively like phone recognition and some classification related domains [9], [10]. A typical RBM network consists of two layers of neurons with one layer representing the input or visible variables and another layer, called the hidden layer, for the unobserved variables. A well-trained RBM can get the unobserved structure of a certain input to help future work.

However, training a neural network is intractable [11], [12]. First, leveraging large number of unlabeled data to boost performance brings numerous workloads. Second, the training process of the neural network intrinsically includes inevitable large matrix multiplication which is very time-consuming. Third, quite a number of matrix multiplications are naturally sequential, which limits parallelism. Previous optimization work using graphic processors or a cluster of machines has proved quite successful in speeding up neural network by reducing weeks of training to a few days or hours [2, 13].

This paper will focus on parallelizing the Autoencoder and Restricted Boltzmann Machine algorithms on a many-core platform. The many-core platform we choose is the Intel Xeon Phi coprocessor. Based on the Intel Many Integrated Core (MIC) architecture, the Intel Xeon Phi coprocessor achieves dramatic performance gains on some of the most demanding applications. Each coprocessor features many smaller cores, threads and wide vector units. The high degree of parallelism compensates for the relatively lower speed of each individual core. Moreover, The Intel Xeon Phi many-core platform provides a general–purpose programming model that allows programmers to develop their programs in an easier and more general way.

In this paper, we attempt to explore the potential of the many-core platform to see how much it can speed up the unsupervised pre-training process.

The rest of the paper is organized as follows. Section II describes the basic concepts of deep learning and the Intel Xeon Phi platform, plus a brief introduction to the algorithms of Sparse Autoencoder and RBM. Section III discusses related work. Section IV introduces our implementation of parallel algorithm on Intel Xeon Phi in details. Section Vevaluates the experimental results of our implementation.

II. BACKGROUND

A. Training Process of Deep Learning

The unsupervised pre-training process of a deep neural network consists of many layers of unsupervised learning processes. We take the Stacked Autoencoder as an example to show the unsupervised pre-training process of a deep neural network.

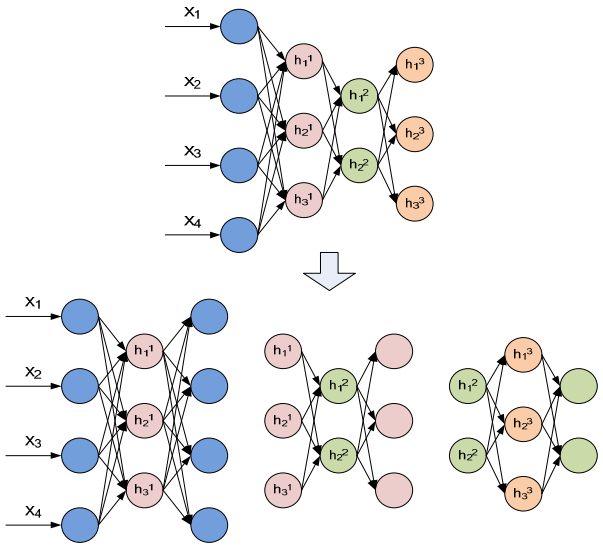


Figure 1. Trainging process of a Stacked Autoencoder

As shown in Fig. 1, a four-layer deep neural network can be decomposed into three Autoencoders. Given a dataset, we first use the dataset to train the first Autoencoder. Then we use the original dataset as input and get the output of the hidden layer. The output dataset is then used as the input training set of the second Autoencoder. The training processes of the second and third Autoencoders are the same as the first Autoencoder. The differences between them only lie in the training set. The pre-training of this deep network consists of three sequential unsupervised trainings.

B. Unsupervised Learning

The training process of a deep neural network contains unsupervised learning process. In machine learning, the purpose of unsupervised learning is to find the hidden structure of given unlabeled data. Given a dataset $X = \{x^1, x^2, x^3, ..., x^n\}$ with each input $x^i \in R^k$. Unsupervised learning aims to find the hidden structure or features $y^i \in R^m$ for each input data x^i so that the extracted representations can benefit subsequent work. Here, we introduce the specific unsupervised learning problems discussed in this paper.

We investigated two unsupervised learning problems: Sparse Autoencoder and Restricted Boltzmann Machine (RBM). They are two different building blocks of deep neural networks.

1) Sparse Autoencoder

Autoencoder is an artificial neural network that is often used to convert high-dimensional data to low-dimensional vectors. A typical autoencoder usually has three layers including one hidden layer.

An Autoencoder takes the input $x \in R^m$ and then maps it to a hidden representation with a deterministic mapping:

$$y = s(W^1 \bullet x + b^1) \quad (1)$$

where W^1 is the connection weight between the input layer and hidden layer and b^1 is the bias of the hidden layer. s is a non-linear mapping such as the sigmoid

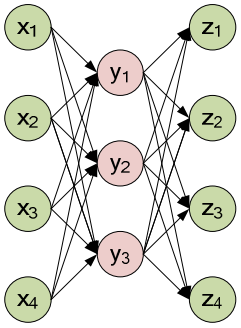


Figure 2. the architecture of anAutoencoder

function. $y \in R^n$ is usually considered as a compressed code of the input (if $n < m$) or over-complete feature representations (if $n > m$). Then y is mapped back to z which is of the same shape of x using a similar mapping:

$$z = s(W^2 \bullet y + b^2). \quad (2)$$

W^1, b^1, W^2, b^2 are the parameters to be tuned so that z decoded from y can be as close to x as possible. The square error function is usually used as the loss function:

$$J(W, b; x, z) = \frac{1}{2}\left\| z - x \right\|^2. \quad (3)$$

Given a dataset $\{x^1, x^2, ..., x^m\}$, the cost function is defined as follow:

$$J(W, b) = \frac{1}{m}\sum_{i=1}^{m} J(W, b, x^i, z^i) + \frac{\lambda}{2}(\left\| W^1 \right\|^2 + \left\| W^2 \right\|^2) \quad (4)$$

The second term is the regularization term to avoid over fitting.

Given the fact that only a few of human visual neurons are activated when observing objects, a sparsity penalty term is added to restrict the number of neurons activated. If we use ρ_i to denote the average activation of the hidden node i given the training set, then the final cost function is in the shape of:

$$J(W, b, \rho) = J(W, b) + \beta\sum_{i=1}^{h} KL(\rho \,||\, \rho_i). \quad (5)$$

Where h is the number of nodes of the hidden layer and ρ is the sparsity parameter. The denotation KL means the KL divergence which is calculated as follows:

$$KL(\rho \,||\, \rho_i) = \rho \log\frac{\rho}{\rho_i} + (1-\rho)\log\frac{1-\rho}{1-\rho_i}. \quad (6)$$

To train the neural network, we use back propagation to search for the minimum point of the cost function [14]. The basic progress of back propagation is to compute the gradient layer by layer from the last level of the network to the first layer. The computations are correlated and thus we cannot compute the entire gradient all together.

2) Restricted Boltzmann Machine

Restricted Boltzmann Machine (RBM) is a generative probabilistic model which Hinton et al.[1] used to build up Deep Belief Network (DBN). It is another building block of deep neural network other than Sparse Autoencoder. Briefly, a Restricted Boltzmann Machine is a two-layer fully connected network. Fig. 3 shows the architecture of an RBM network.

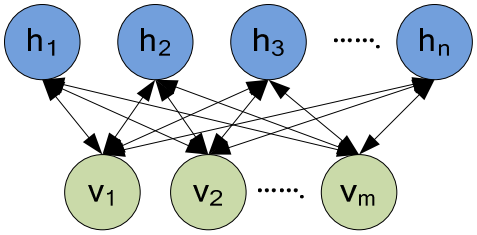


Figure 3. the architecture of a Restricted Boltamann Machine

Consider a set of binary vectors as our input. The set can be modeled by RBM in which the stochastic binary vectors are connected to the stochastic feature detectors using symmetric weighted connections [4, 15, 16]. The input vectors correspond to visible units because they are observed while the feature detectors correspond to the hidden units. The joint distribution (v, h) can be assigned an energy as follows [17]:

$$E(v, h) = -b^{'}v - c^{'}h - h^{'}Wv, \quad (7)$$

Where W is the weights connecting the visible and hidden units and b, c are the biases of visible and hidden units respectively. Because of the specific structure of RBM, the states of visible units and hidden units are independent given one another. Thus the conditional probability can be computed as follows:

$$p(v_i = 1 \,|\, h) = s(b_i + \sum_j W_{ij} h_j), \quad (8)$$

$$p(h_i = 1 \,|\, v) = s(c_i + \sum_j W_{ji} v_j). \quad (9)$$

Maximum likelihood learning is used to train an RBM with Contrastive Divergence that calculates the log likelihood gradients [15]. The derivative of the log probability of a training vector with respect to a weight or bias is computed according to the following formula:

$$\frac{\partial \log p(v)}{\partial w_{ij}} = \left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{model}, \quad (10)$$

$$\frac{\partial \log p(v)}{\partial b_i} = \langle v_i \rangle_{data} - \langle v_i \rangle_{model}, \quad (11)$$

$$\frac{\partial \log p(v)}{\partial c_i} = \langle h_i \rangle_{data} - \langle h_i \rangle_{model}, \quad (12)$$

where the angel brackets are used to denote the expectations under the distribution specified by the subscript that follows.

However, calculating the second term is difficult. Hinton proposed Contrastive Divergence to get an approximation by running Gibbs Sampling one step. This algorithm first sets the visible units to one of the training data. At each step, the binary states of all the hidden units (visible units) are computed following the equations above. Thus the change of a weight is:

$$\Delta w_{ij} = \eta(\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{sample}), \quad (13)$$

where η is the learning rate.

C. Computing with Intel Xeon Phi

The Intel Xeon Phi coprocessor provides up to 61 cores, 244 threads, and 1.2 teraflops of performance with all the computing cores connected by a ring bus. Equipped with 8GB of GDDR5 memory, it can provide a bandwidth of 325GB/s per coprocessor. In addition, each computing core consists of a double-wide (256-bit) vector engine supporting 512-bit SIMD instructions. Thus, the Intel Xeon Phi family is quite suitable for highly-parallel, vector-intensive and memory bound computation.

In addition to the features mentioned above, a set of programming languages, models and tools supporting the Intel x86 architecture can also be used on the Intel Xeon Phi coprocessor with little change. As a result, instead of redesigning new algorithms or models, developers now can reuse existing codes or applications and maintain common codes using familiar tools and methods. The vector-intensive algorithms in our method take advantage of this feature.

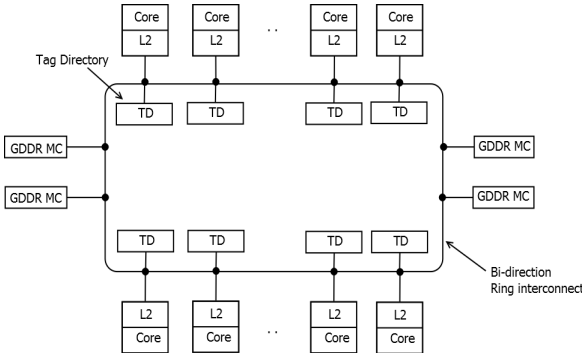


Figure 4. The architecture of Intel Xeon Phi

III. Related Work

Several methods have been proposed to parallelize deep learning algorithms. Recent work has shown that most of the work can benefit from parallel architecture by distributing datasets to different computing nodes and then combine all the results [18]. Some machine learning algorithms such as logistic regression and SVM are naturally divisible, which makes them suitable for parallel training. Standard Sparse Autoencoder or RBM, however, are intrinsically sequential. They often involve a large number of sequential computations in which the computation of new gradient is based on previous updates. This makes it hard to massively parallelize on a coarse data level [19].

Despite of the difficulties mentioned above, several ways to speed up or optimize deep learning have been proved effective in practical application. We can classify them into two categories.

For the first category, some algorithms focus on adaptive strategies for the learning rate to make it faster to converge [20]. Using changing learning rate instead of constant learning rate has reduced the iterations needed to converge [21, 22] and thus can speed up the training. Online Stochastic Gradient Descent (SGD) is a common-used optimization method to minimize the cost function of one kind of deep learning algorithm. It performs its update for each training example. This sequential feature makes it hard to parallelize. In order to overcome the weakness of online Stochastic Gradient Descent (SGD) which is inherently sequential, the batch methods like limited memory BFGS (L-BFGS) or Conjugate Gradient (CG) has been proposed [23, 24]. These methods make it easier to parallelize the deep learning algorithms. However, these methods are slower to converge since one update of parameters involves much more computations than SGD.

For the second category, more hardware resources are devoted to find the internal parallelism of a certain algorithm. Google has distributed a very large deep network to hundreds of computing nodes and uses lock—less asynchronous update to speed up the procedure. Google concludes that the MapReduce platform is ill-suited for iterative tasks like neural network training. Also, GraphLab which is designed for general graph computation is unable to exploit the computing efficiency found in structured graph like deep neural network [13]. In addition, GPU has also shown great potential in training modest- sized neural network [19, 25, 26].

In this paper, we try to exploit the potential of the novel architecture, the Intel Xeon Phi many-core coprocessor platform, to parallelize deep learning and evaluate its performance. We try to leverage its advantage on general-purpose programming to train neural network and offer another efficient option other than GPU. As far as we know, our work is the first attempt to speed up the training of deep learning using the Intel Xeon Phi platform.

IV. Design of our Parallel Algorithm

A. Basic Process

The training algorithms of Sparse Autoencoder and Restricted Boltzmann Machine run in a similar way. During every epoch, they pick a small batch of unlabeled data, compute the gradient (using Back Propagation for

Sparse Autoencdoer or Contrastive Divergence for RBM) and then update all the parameters. Although the basic process is short and concise, to parallelize it successfully and efficiently on the Intel Xeon Phi platform requires careful consideration on many aspects. There are two key points that will play important roles in parallelizing and optimizing the performance of our algorithm.

First, memory transfers between the host and the Intel Xeon Phi is relatively slow. Thus, the number of data transfers between the host and Intel Xeon Phi should be minimized as much as possible. Therefore, we load the training data into the global memory of Intel Xeon Phi in a large chunk.

Second, we use a thread to load the data chunk from the host to the Intel Xeon Phi so that our algorithm does not need to wait for loading new data when finishing the process of training one large chunk of data. This is a key point to keep all cores busy all the time.

Fig. 5 illustrates the basic data loading and processing strategy.

To make full use of the potential of the Intel Xeon Phi, we tried to design a fine-grained algorithm. A fine-grained algorithm may make full use of the Intel Xeon Phi. However, it also leads to more synchronization cost. For this reason, we should consider the granularity of our algorithm carefully. Besides, the data transfer rates between the host machine and the Intel Xeon Phi is relatively slow and thus we should avoid data transfer between the host and Intel Xeon Phi as much as possible.

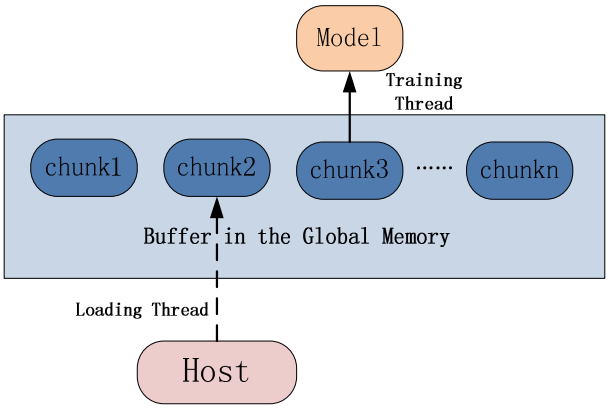


Figure 5. Loading thread concurrent with training thread

Based on the ideas above, we discribe the basic design of our parallel algorithm. Algorithm 1 is the pseudo-code for the basic process of our parallel algorithm for unsupervised learning.

As we have discussed above, the transferring speed between the host and Intel Xeon Phi is relatively slow. Our test shows that it costs 132s to transfer 10,000*4096 samples from the host to Intel Xeon Phi and our training time is about 628s. This means that about 17% of the total time is spent on transferring training data. During the period of loading data, most of our computing cores are idle, which wastes computing resources and thus undermines the performance. We can cut down this transferring time by setting a loading thread and a loading buffer in the global memory on Intel Xeon Phi. We make part of the global memory as the loading buffer and set its size as several times as that of a data chunk. While the loading thread is loading data into the i th data chunk, our training thread can use the $i-1$ th data chunk to train. The proportion of time cost in transferring can be notably reduced when we train a large number of data.

Algorithm 1 Parallelize Autoencoder / RBM on Intel Xeon Phi

1:**Initialize parameters of our unsupervised network**
2:**While** stop condition is not satisfied
3: get a chunk of data from the buffer area in global memory
4 : split the chunk into many smaller training batches
5: **For** each small training batch
6: compute the gradient accordingly
7: update the parameters
8: **EndFor**
9:**EndWhile**

Among the steps mentioned in our algorithm, computing the gradient is the most time-consuming for both Sparse Autoencoder and RBM. We then introduce our method to parallelize the computing gradient step of RBM and Sparse Autoencoder.

B. Parallelizing methods for RBM and Sparse Autoencoder

1) Parallelizing RBM

To parallelize the computation of RBM as much as possible, we design several optimizations to reduce the time.

First, since the size of our model is moderate, we keep all the parameters including W, b, c in our global memory permanently. In addition to these parameters, several temporary variables needed by each gradient computation are also kept permanently to avoid unnecessary reallocation and release. Second, we can use the 512-bit wide Vector Processing Unit (VPU) of Intel Xeon Phi to speed up several loops. Thus, we vectorize the sampling and update step of RBM training. Specifically, the equation of the sampling step should be rewrited in vector form:

$$p(v \mid h) = vectorsig(b + W \bullet h), \quad (14)$$

$$p(h \mid v) = vectorsig(c + W \bullet v). \quad (15)$$

And the sampling step can also be vectorized. Since the updates of each parameter are independent, the

updating can also be vectorized and the update step can be written in vector form:

$$W = W + \Delta W, \quad (16)$$

$$b = b + \Delta b, \quad (17)$$

$$c = c + \Delta c. \quad (18)$$

Third, despite of all optimizations described above, the eventual optimizing effect would be very limited if we did not focus on the matrix operations that exist in our algorithm.

In our algorithm, we use the Intel MKL (Math Kernel Library) packages to perform all the time-consuming matrix operations. The Intel MKL is a library of optimized mathematical operation. Its core functions contain BLAS, LAPACK and so on. The Intel MKL packages greatly speed up common mathematical operations by exploiting modern many-core processors and wide vector units. This has been proved to be very efficient in our algorithm.

Fourth, some matrix operations can also be calculated concurrently based on the sequence of the computations. We take the computation of RBM as an example and illustrate the parallelism we can excavate.

Fig. 6 shows the dependency of all the variables in computing the gradient of the parameters needed in one iteration. The detonation Vb , Vc and Vw are the gradients needed to be calculated and others are the temporary variables. Each arrow pointing from A to B denotes that the calculation of B depends on the calculation of A and thus A and B cannot be calculated concurrently. We try to find all the computations that can be computed concurrently. From the dependency graph above, we can find some computation bodies for parallelization. Once $V1$ is calculated, then we can only compute $H1$ because all other computations need some variables that have not been calculated yet. After getting the result of $H1$, the computations of $V2$ and $C2$ can run in parallel. Similarly we can compute Vb , $H2$ and $C2$ after $V2$, and compute Vb , Vc and Vw after $H2$ in parallel because all preconditions of computing the gradient are satisfied.

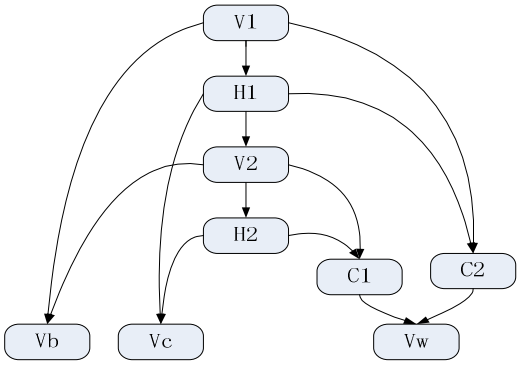


Figure 6. the dependency of all temporary variables in computing the gradient of a RBM network.

2) Parallelizing Sparse Autoencoder

Comparing with parallelizing RBM, parallelizing Sparse Autoencoder is more complicated given the complexity of back propagation algorithm. However, the basic ideas to parallelize it are the same. We also need to find the dependencies of each matrix operation like Fig. 6 and use OpenMP and Intel MKL packages to parallelize them.

As mentioned above, the granularity of parallelism impacts a lot. The calculation of the back propagation has the features as follows:

- Many matrix multiplications and they have been well tackled by the Intel MKL packages.
- There are some loops that cannot be transformed effectively into matrix operations. As to these loops, we can simply use OpenMP to parallelize them. However, it turned out to be ineffective since the loop body is relatively small and the time cost in synchronization accounts most of the total time. We finally combine several loops together to make the granularity more suitable for our platform.

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our method. We conducted five different experiments to analyze the performance of algorithm on Intel Xeon Phi from different aspects.

First, we ran our full-optimized algorithm on both single Intel Xeon CPU core and Intel Xeon Phi to evaluate the advantage of Intel Xeon Phi over single CPU core. We compare the performance of Intel Xeon Phi and single CPU core in three aspects: network size, dataset size and batch size.

Second, we ran our full-optimized algorithm of Autoencoder on Intel Xeon Phi and ran a Matlab code of it using Matlab on the same single Xeon CPU. Since Matlab has done a great job in optimizing matrix operations, we conducted this experiment to show that our optimization work on Intel Xeon Phi has desired effect.

Third, we evaluated our optimization process on Intel Xeon Phi. We first implemented an algorithm of stacked-up Autoencoder on Intel Xeon Phi without using Intel MKL packages or any other skills to speed up. Then we optimize it step by step until we get the full-optimized code today.

A. Platforms and Datasets

1) Platforms and Software

We ran our algorithm on both the Intel Xeon Phi platform and Single Xeon CPU core.

The Intel Xeon Phi platform we used comes with Xeon Phi 5110p many-core coprocessor. It is equipped with 60 active cores, each core with a frequency of 1.053 GHz, memory bandwidth of 320 GB/s and global memory of 8GB.

The CPU we used to do experiments is Intel(R) Xeon(R) E5620, with frequency of 2.4GHz and 4 cores, and cache size of 12288 KB.

The Matlab version we used is Matlab 7.14.0.739 (R2012a).

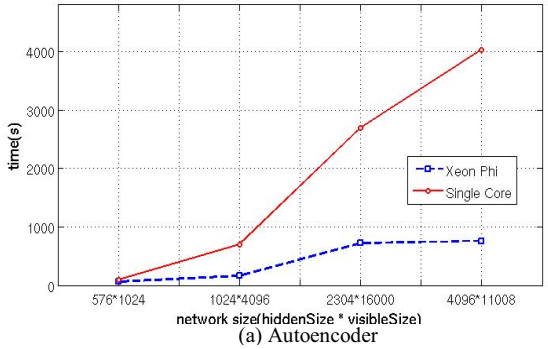
2) Datasets

Our dataset comes from a large of handwritten digit images and natural images [27, 23]. We obtain the training examples by randomly extracting patches of required sizes from these images.

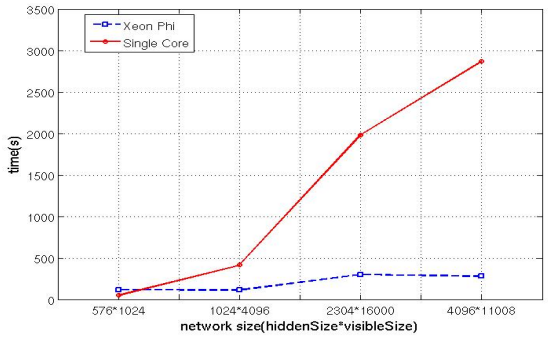
B. The Performance and Analysis

1) Impact of Network Size

Firstly, we evaluate the performance when the network size goes up. The size of the training dataset for Sparse Autoencoder is about 1 million training examples. The dimension of the examples accords with the visible size of our network. We conducted back propagation updates in batches of 10000 examples. The total size of training examples and batch size for RBM are100,000 and 200 respectively. Fig. 7 shows the time costs when the size of our network goes from 576*1024 to 4096 * 11008.



(a) Autoencoder



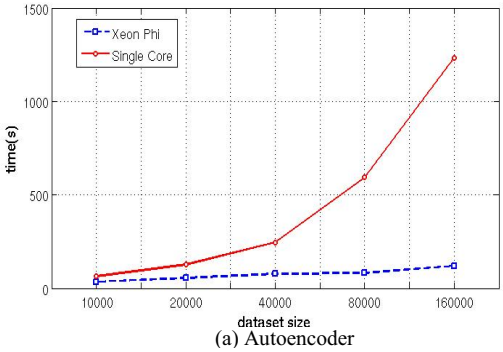
(b) RBM

Figure 7. Performance of parallel Autoencoder and RBM algorithms running on Intel Xeon Phi compared with sequential one on single CPU core on host

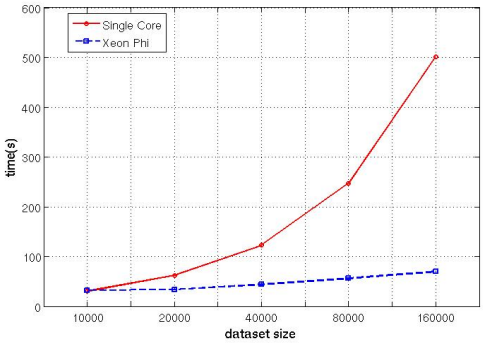
Fig. 7 shows that when the size of the network goes larger and larger, the time costs of single CPU core on host increases sharply. However, the time growth of our implementation on Intel Xeon Phi is mild. The time growth of single CPU core increases almost linearly. It also demonstrates that the difference between single CPU core and Intel Xeon Phi is small when the size of network is small. This is because the benefit brought by many cores is neutralized by the synchronization of threads when the network size is not big enough.

2) Impact of Dataset Size

To measure the performance of dataset size, we fixed the network size of RBM and Autoencoder to 1024 * 4096 and the size of our dataset varies from 10000 examples to 160000 examples. The batch size equals 1000 examples.



(a) Autoencoder



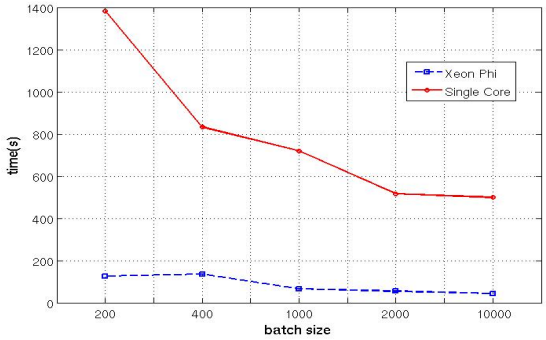
(b)RBM

Figure8. performances when the size of dataset goes up. Network size: 1024 * 4096. Batch size: 1000

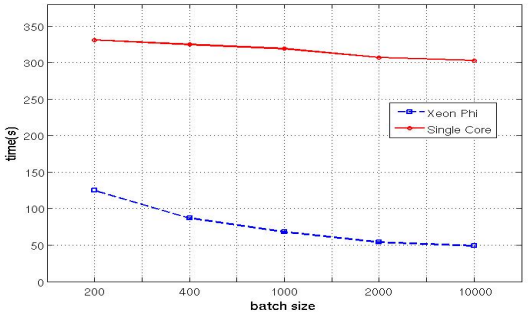
When the size of dataset increases, the time cost by single CPU core increases much faster than Intel Xeon Phi while the time cost by Intel Xeon Phi does not change much. It shows that Intel Xeon Phi works much better when dealing with large dataset size.

3) Impact of Batch Size

It also indicates that the batch size impacts a lot on both single CPU core and Intel Xeon Phi. To assess the impact of the batch size, we fixed the network size to 1024 * 4096 and the dataset size to 100,000 examples. The batch size of an update varies from 200 to 10000. When the batch size goes larger, we need less iterations to train a fixed-sized data chunk.



(a)Autoencoder



(b)RBM

Figure 9. the impactof batch size when batch size goes larger

No matter what the batch size is, the time cost by Intel Xeon Phi maintains at a low level while the single CPU core costs much more time. When the batch size goes larger, both of them decrease since the number of iterations decreases. The experiment shows that the time cost of Autoencoder decreases by two thirds when the batch size increases from 200 to 10,000.

As to RBM, the time decreases on single CPU core is not obvious while time cost by Intel Xeon Phi drops by about two thirds. This is another proof that our method on Intel Xeon Phi works much better when dealing with large data and large network.

4) Comparison with Matlab

In addition to the experiments above, we also compared our algorithm with the Matlab implementation of Autoencoder and the parallel implementation on Intel Xeon Phi. Our Matlab code ran on the single CPU platform and Matlab has its own optimization of matrix operations. The dataset contains 1 million examples and the mini batch we used contains 10,000 examples. The Matlab code ran on the single CPU platform and we did not restrict the number of cores it used.

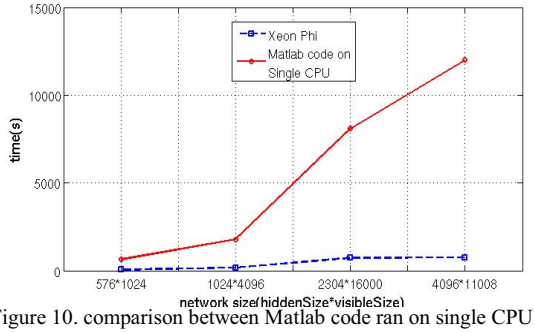


Figure 10. comparison between Matlab code ran on single CPU and Intel Xeon Phi

As we can see from Fig. 10, our version runs much faster than the Matlab version we implemented. It achieved about 16-fold speed up even if Matlab has an efficient implementation of matrix operations. We performed this experiment to show the effect of both Intel Xeon Phi and our optimization work on the training time of deep neural network.

5) Impact of Each Optimization Step on Intel Xeon Phi

At last, we show the impact of each optimization step we used in our experiment on Intel Xeon Phi. The network we used is different from previous ones. We used a four-layer network and the size of each layer is 1024, 512, 256, 128. The training process is exactly the same as the unsupervised pre-training process of a deep network in which the training examples of higher layer come from the output of the previous layer. The batch size we used to train each layer is 10000 examples and each layer ran 200 iterations.

TABLE I. PERFORMANCE AFTER EACH OPTIMIZATION STEP ON XEON PHI

	60 cores	30 cores
Baseline	16024s	15960s
OpenMP	892s	2122s
OpenMP+MKL	97s	120s
Improved OpenMP+MKL	53s	81s
Speedup(fully-optimized compared with baseline)	302	197

The baseline code did not use Intel MKL packages or any other speedup methods. We then used OpenMP to parallelize all the loops. After that, we used MKL to perform the matrix operations and some speedup skills in section Ⅳ. At last, we combined some loops to reduce synchronization cost. The result is shown in Table I. Apparently, there is a remarkable disparity between the baseline version code and our optimized code on Intel Xeon Phi. The result shows that Intel Xeon Phi gained an approximately 302-fold speedup compared with the sequential algorithm. The right column shows how Intel Xeon Phi performs when restricting the number of cores by half. This table shows the result of speedup of our algorithm on Intel Xeon Phi.

As far as the practical speedup of our work is concerned, our algorithm should have the same effect on real world data as it has on experimental data because the optimization work is irrelevant to specific data type and data distribution.

VI. CONCLUSION AND FUTURE WORK

In this paper, we designed and implemented parallel algorithm for unsupervised pre-training process of deep network on Intel Xeon Phi many-core platform and gained 302-fold speedup compared with the un-optimized sequential algorithm .

The MapReduce framework has done a great work in many machine learning algorithms but it relies too much on data parallelism [28]. Meanwhile, GPU has also shown its power in unsupervised pre-training [19]. However, the programmability of GPU has always been an obstacle. Our study on this paper suggests that Intel Xeon Phi shows its strength in training these networks. Also due to the general-purpose programming model for Intel Xeon Phi, programmers can quickly transplant their original program on host machine to the Intel Xeon Phi platform. This significantly increases the programmability for programmers.

However, the speedup is obtained largely by more cores and our implementation is still relatively coarse to make full use of Intel Xeon Phi. So there is some future work for us to do. First, a balance should be found between parallelism and synchronization. For now, we need to adjust the number of threads manually in our implementation. Second, a further combination between Xeon and Intel Xeon Phi can bring us higher efficiency. Since the transferring speed between Xeon and Intel Xeon Phi is slow, the transferring cost can be intolerable when the model becomes large. Third, we need to make our algorithm more efficient to deal with mini batch because online SGD is more common in practical use.

ACKNOWLEDGMENT

This work is funded in part by China NSF Grants (No. 61223003), and the USA Intel Labs University Resea rch Program.

REFERENCES

[1] Hinton, Geoffrey E., and Ruslan R. Salakhutdinov. "Reducing the dimensionality of data with neural networks." *Science* 313.5786 (2006): 504-507.

[2] Y. Bengio and Y. LeCun, "Scaling learning algorithms towards AI," in *Large Scale Kernel Machines*, (L. Bottou, O. Chapelle, D. DeCoste,and J. Weston, eds.), MIT Press, 2007.

[3] Olshausen, Bruno A., and David J. Field. "Sparse coding with an overcomplete basis set: A strategy employed by V1?." *Vision research* 37.23 (1997): 3311-3325.

[4] Smolensky, Paul (1986). "Chapter 6: Information Processing in Dynamical Systems: Foundations of Harmony Theory". In Rumelhart, David E.; McLelland, James L. Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations. MIT Press. pp. 194–281. ISBN 0-262-68053-X.

[5] M. Ranzato, C. Poultney, S. Chopra, and Y. LeCun, "Efficient learning of sparse representations with an energy-based model," in *Advances in Neural Information Processing Systems 19 (NIPS'06)*, (B. Schölkopf, J. Platt, and T. Hoffman, eds.), pp. 1137-1144, MIT Press, 2007

[6] Ackley, David H., Geoffrey E. Hinton, and Terrence J. Sejnowski. "A learning algorithm for Boltzmann machines." *Cognitive science* 9.1 (1985): 147-169.

[7] Cochocki, A., and Rolf Unbehauen. *Neural networks for optimization and signal processing*. John Wiley & Sons, Inc., 1993.

[8] Dahl, George, Abdel-rahman Mohamed, and Geoffrey E. Hinton. "Phone recognition with the mean-covariance restricted Boltzmann machine." *Advances in neural information processing systems*. 2010.

[9] Larochelle, Hugo, and YoshuaBengio. "Classification using discriminative restricted Boltzmann machines." *Proceedings of the 25th international conference on Machine learning*.ACM, 2008.

[10] Ng, Andrew. "Sparse autoencoder."*CS294A Lecture notes* (2011): 72.

[11] Hecht-Nielsen, Robert. "Theory of the backpropagation neural network."*Neural Networks, 1989.IJCNN., International Joint Conference on*. IEEE, 1989.

[12] Paul J. Werbos. Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. PhD thesis, Harvard University, 1974

[13] J. Dean , G. Corrado , R. Monga , K. Chen , M. Devin , Q. Le , M. Mao , M. Ranzato , A. Senior , P. Tucker , K. Yang and A. Ng "Large scale distributed deep networks", *Proc. Adv. Neural Inf. Process. Syst.*, 201Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. NeuralComputation, 14(8):1711−1800.

[14] LeCun, B. Boser, et al. "Handwritten digit recognition with a back-propagation network." *Advances in neural information processing systems*. 1990.

[15] G. E. Hinton, "A Practical Guide to Training Restricted Boltzmann Machines," in Technical report 2010-003, Machine Learning Group, University of Toronto, 2010.

[16] J. J. Hopfield "Neural Networks and Physical Systems with Emergent Collective Computational Abilities", Proc. Natl. Acad. Sci. USA, vol. 79, pp.2554 -2558 1982

[17] Chu, Cheng, et al. "Map-reduce for machine learning on multicore." Advances in neural information processing systems 19 (2007): 281.

[18] Raina, Rajat, Anand Madhavan, and Andrew Y. Ng. "Large-scale deep unsupervised learning using graphics processors." ICML. Vol. 9. 2009.

[19] Ngiam, Jiquan, et al. "On optimization methods for deep learning." Proceedings of the 28th International Conference on Machine Learning (ICML-11). 2011.

[20] Shalev-Shwartz, Shai, et al. "Pegasos: Primal estimated sub-gradient solver for svm." Mathematical Programming 127.1 (2011): 3-30.

[21] Hazan, Elad, Alexander Rakhlin, and Peter L. Bartlett. "Adaptive online gradient descent."*Advances in Neural Information Processing Systems*. 2007.

[22] Dahl, George E., et al. "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition." *Audio, Speech, and Language Processing, IEEE Transactions on* 20.1 (2012): 30-42.

[23] Hestenes, Magnus Rudolph, and Eduard Stiefel. "Methods of conjugate gradients for solving linear systems." (1952): 1.

[24] Liu, Dong C., and Jorge Nocedal. "On the limited memory BFGS method for large scale optimization."*Mathematical programming* 45.1-3 (1989): 503-528.

[25] ClaudiuCiresan, Dan, et al. "Deep Big Simple Neural Nets Excel on Handwritten Digit Recognition." *arXiv preprint arXiv:1003.0358* (2010).

[26] Salakhutdinov, Ruslan, AndriyMnih, and Geoffrey Hinton. "Restricted Boltzmann machines for collaborative filtering."*Proceedings of the 24th international conference on Machine learning*.ACM, 2007.

[27] Olshausen, Bruno A. "Emergence of simple-cell receptive field properties by learning a sparse code for natural images." *Nature* 381.6583 (1996): 607-609.

[28] Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-11