Sameer Deshmukh

17M38101
deshmukh.s.aa@m.titech.ac.jp

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

(2014)

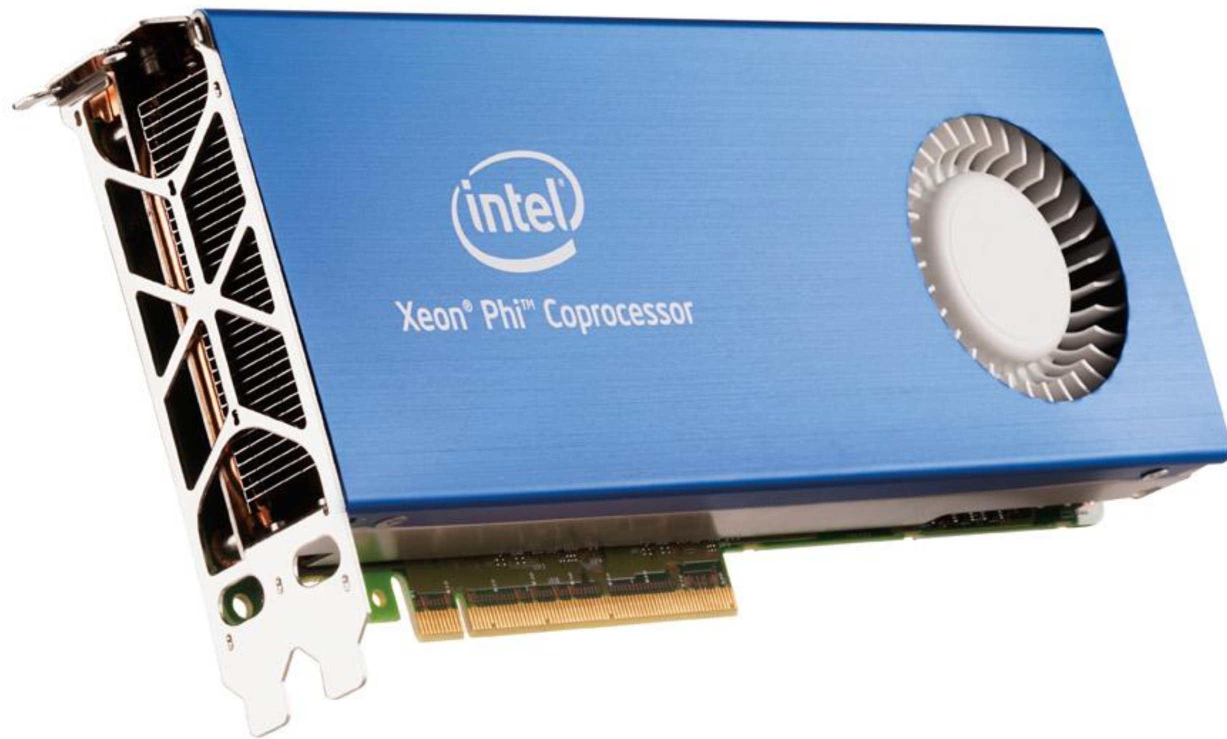
Talk Structure

- Key takeaways.
- Prerequisites.
- Elaboration of solution.
- Experiments for verifying result.
- Conclusion.

Key Takeaways

- Learn algorithms for fast unsupervised pre-training of Neural Networks using RBMs and Spare Auto-encoder on the Intel Xeon Phi.
- Understand the architecture of the Intel Xeon Phi co-processor and its advantages and disadvantages compared to GPUs.
- See benchmarks between a traditional sequential training algorithm vs. parallel algorithm on the Xeon Phi co-processor.

Use of the Intel Xeon Phi for training neural networks.



Prerequisites

Unsupervised pre-training

- When neural networks get deeper, they face the **vanishing gradient problem.**
- Basically when training a NN using back propagation, you might get zero or really high gradients sometimes.
- An additional **unsupervised pre-training** step before the actual training solves this problem.

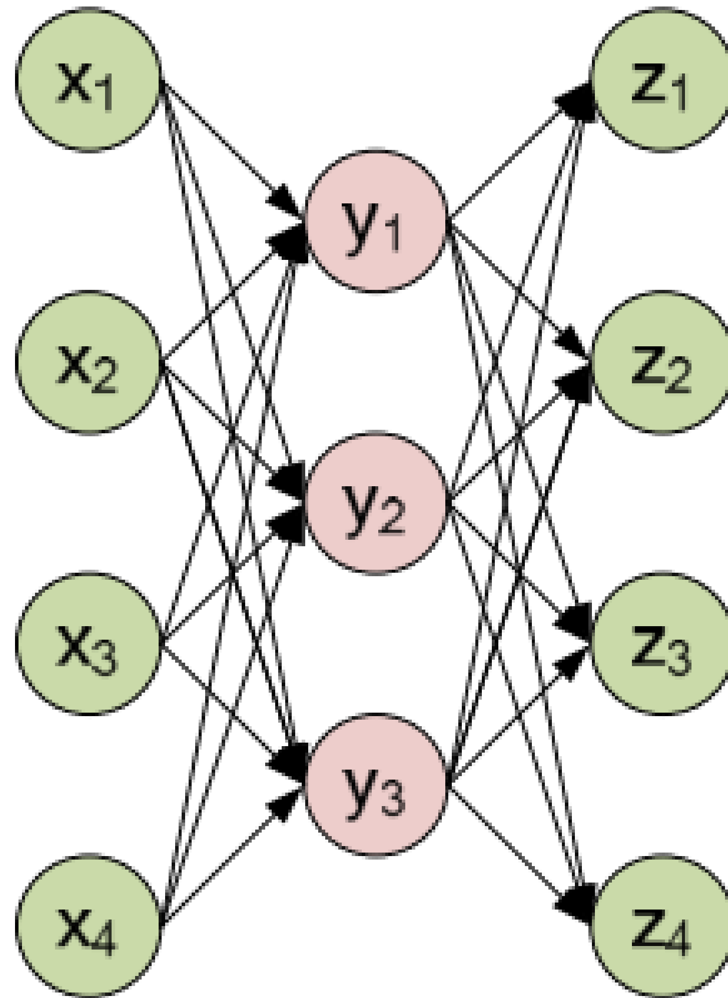
Unsupervised pre-training

- It finds patterns in the data by reconstructing the input.
- This is done with unlabeled data and forces the NN to decide which of the features are most important, eventually acting as a feature extraction engine.
- Sparse Auto Encoder and Restricted Boltzmann Machine are two main ways of achieving this.

Sparse Auto-encoder

A sparse auto-encoder is an unsupervised learning algorithm that applies back propagation, setting the target values to be equal to the inputs.

Top level structure



Input Layer | Hidden Layer | Output Layer

Forward propagation

Takes input $x \in R^m$ through input layer *x* and maps it to the hidden layer *y* with the function:

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

W^1 is the weight, b^1 is the bias of the layer and *s* is an activation function like sigmoid that brings *y* in the domain of [0,1].

Forward propagation

Takes input $y \in R^n$ through hidden layer *y* and maps it to the output layer *z* with the function:

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

W^2 is the weight, b^2 is the bias of the layer and *s* is an activation function like sigmoid that brings *z* in the domain of [0,1].

Loss function

The square error function is usually used as the loss function:

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

Cost function

We make use of the following cost function to train our neural network:

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

The goal of the algorithm is to minimize this function.

Cost function

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

m signifies the number of IID samples that will be used to train this neural network from a set of samples $\{x_1, x_2 \ldots x_m\}$.

Cost function

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

λ is the **weight decay parameter** that controls the relative importance of the two terms.

It tends to decrease the magnitude of the weights, and thereby prevent over-fitting.

The auto-encoder tries to approximate the identity function such that $z = f_{W,b}(x)$.

Sparse auto-encoder

The identity function seems like
a trivial function to learn.

But, by limiting the number of
hidden units, we can discover
interesting correlations in the data.

Sparsity parameter ρ

It is observed that structures in the data are better observed if the number of hidden neurons that fire is limited.

The parameter ρ determines how many neurons will fire. All hidden neurons firing does not lead to the best results.

Final cost equation

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

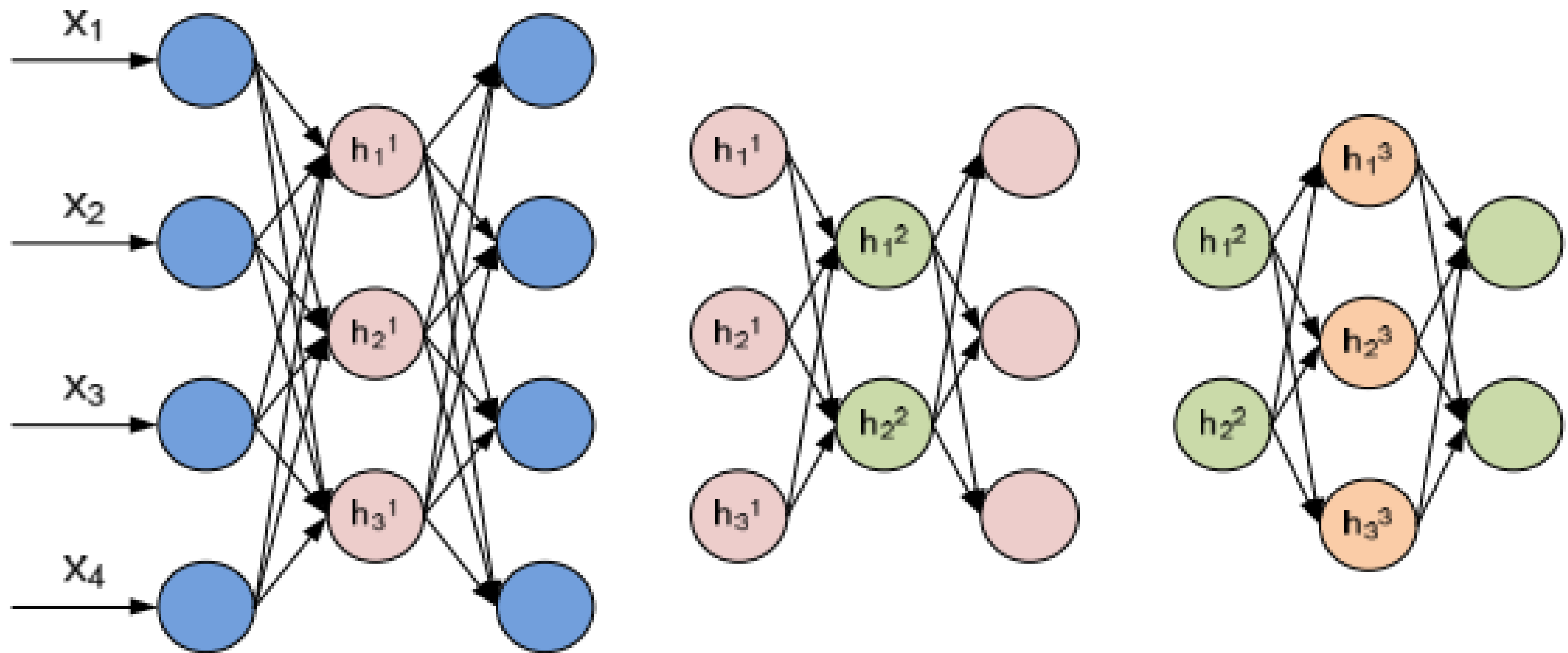
$$KL(\rho \| \rho_i) = \rho \log \frac{\rho}{\rho_i} + (1-\rho) \log \frac{1-\rho}{1-\rho_i}$$

ρ_i -> average activation of the hidden node *i* given the training set.

h -> number of hidden nodes.

β -> Additional learning rate parameter.

Stacked Auto-encoder



Restricted Boltzmann Machine

Top level structure

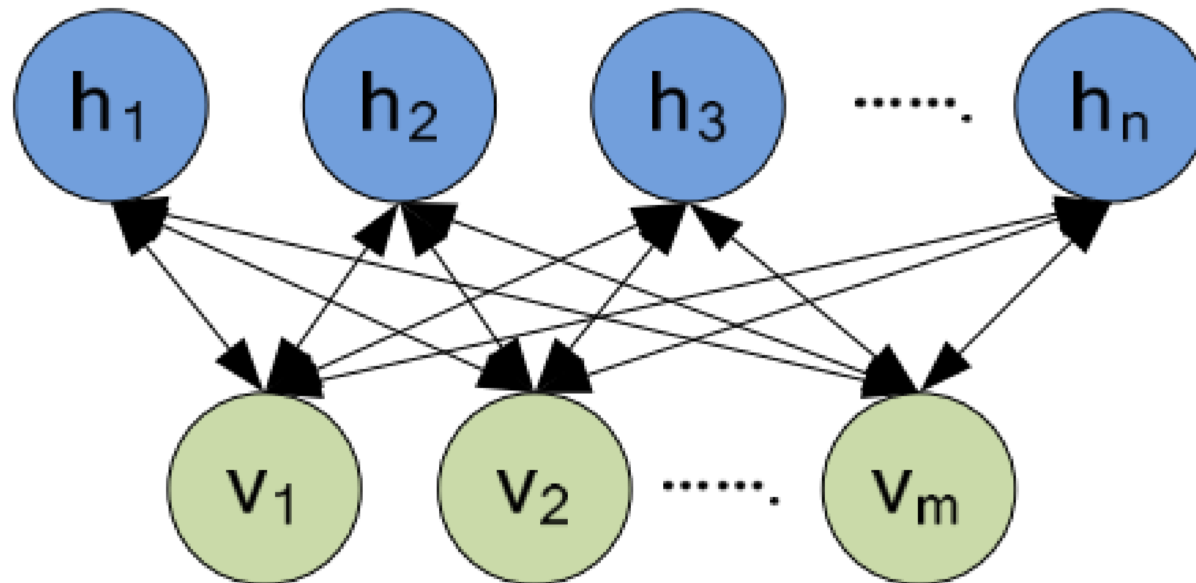


Figure 3. the architecture of a Restricted Boltamann Machine

RBM Overview

- Used for building Deep Belief Networks (DBM).
- Two-layer fully-connected network.
- Works based on 'Energy' equation of neurons back propagation using Contrastive Divergence.
- Trains the NN for each layer; layer by layer.
- Uses unlabeled data.

Training stage

- Consist of ‘forward pass’ and ‘backward pass’.
- Works like a 2 way translator.
- Input -> encoding translation in the forward pass.
- Encoding -> input translation in the backward pass.

Feature detection

- The input vector corresponds to the visible units because they are observed.
- Feature vector corresponds to the hidden layer.
- Weights are converted into most important features due to weight adjustment.

Energy function

- RBM is an Energy Based Model (EBM) – It defines the probability via an energy function.
- The actual probability of firing is controlled by the weights between the neurons and their individual biases.
- An energy function assigns probabilities to different configurations of a system.

Energy function

- The energy function for a joint distribution of *(v,h)* can be defined as follows:

$$E(v,h) = -b'v - c'h - h'Wv$$

- The probability of firing in RBM is inversely proportional to the energy:

$$p(v,h) \propto \frac{1}{e^{E(v,h)}}$$

Probability from energy equation

The probability of visible vector v, is given by summing over all the probabilities:

$$p(v) = \frac{e^{-E(v,h)}}{\sum_h e^{-E(v,h)}}$$

Due to the nature of RBM, the probabilities of visible and hidden units are independent of each other.

Calculating conditional probabilities

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

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

B_i -> bias of the visible layer.
C_i -> bias of the hidden layer.
W_{ij} -> Weight from visible to hidden layer.
W_{ji} -> Weight from hidden to visible layer.

Contrastive Divergence

- RBM uses this technique to adjust weights during training.
- Calculates the partial derivatives of log likelihood of probability equation with respect to weight and biases.

Calculating partial derivatives

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

$$\frac{\partial \log p(v)}{\partial b_i} = \left\langle v_i \right\rangle_{data} - \left\langle v_i \right\rangle_{model}$$

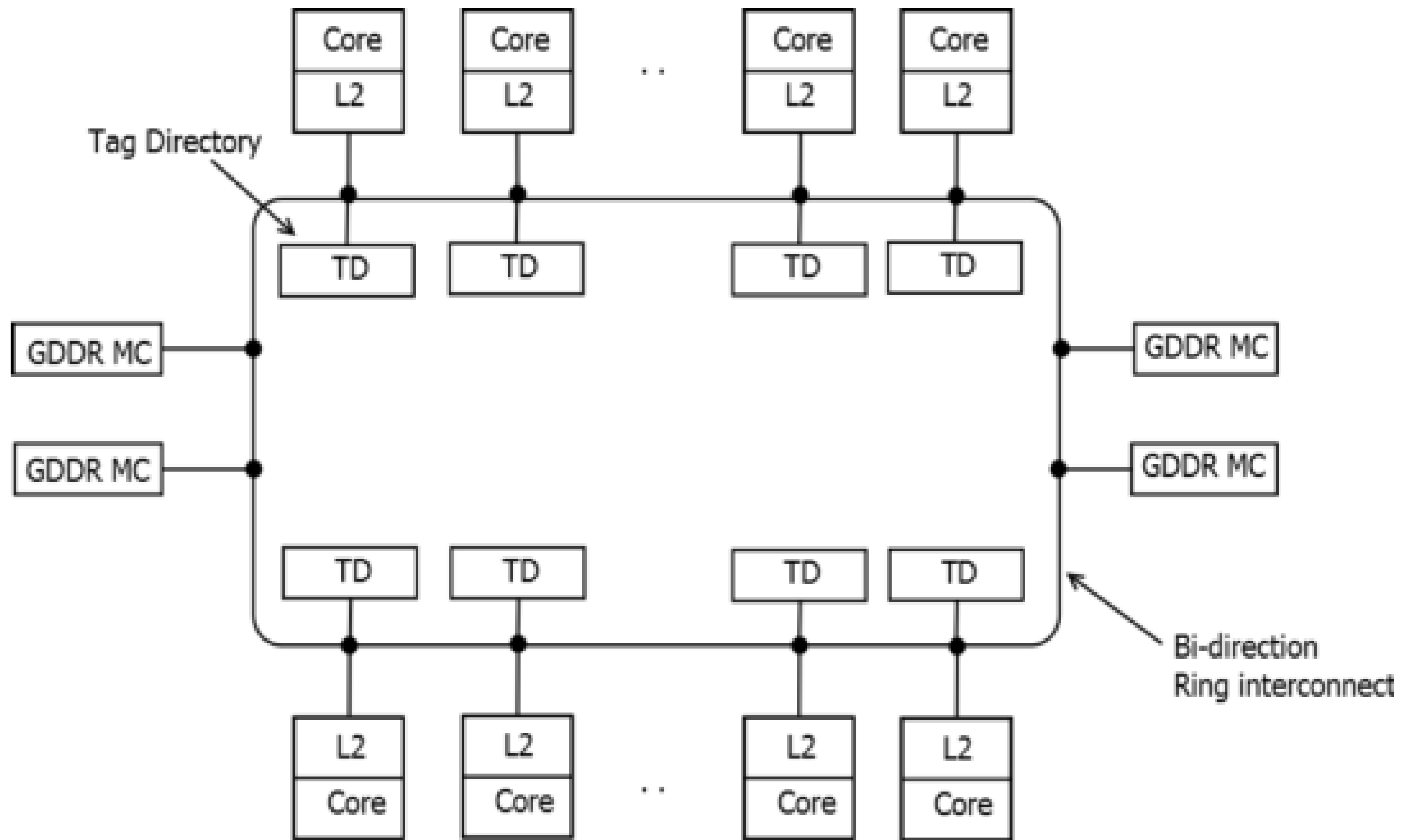
$$\frac{\partial \log p(v)}{\partial c_i} = \left\langle h_i \right\rangle_{data} - \left\langle h_i \right\rangle_{model}$$

$$\Delta w_{ij} = \eta(\left\langle v_i h_j \right\rangle_{data} - \left\langle v_i h_j \right\rangle_{sample})$$

Intel Xeon Phi

- Upto 60 cores.
- 1.053 GHz per core.
- Cores connected by a ring bus.
- 8 GB GDDR5 memory.
- Each core supports 512-bit wide SIMD instructions.
- All tools and programs used on Intel x86 processors can be used with little change.

Hardware interconnect



Advantages of Xeon Phi

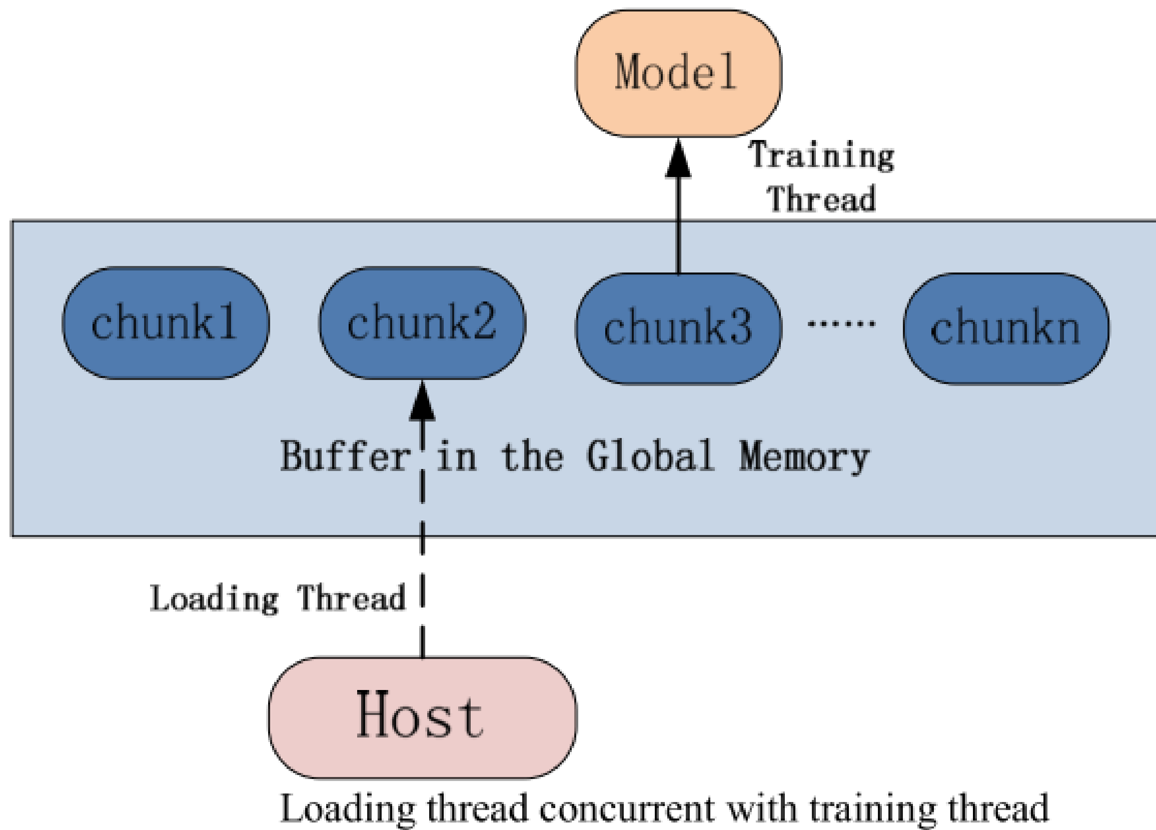
- Ability to login and run ‘native applications’ without intervention from host CPU. Very different from GPU!
- Run x86 code on the co-processor without major modifications.

Elaboration of solution

Key considerations for design

- Memory transfers between Host and Phi are relatively slow. Thus, we load training data into Phi memory in large chunks.
- Use threads to load data into the Phi so that our algorithm does not need to wait for data loading.

Data loading overview



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**

Parallelize RBM

Parallelization steps (1)

1. Since size of model is small, keep all parameters including W, b and c in the Phi global memory.
2. Vectorize the sampling and update step of RBM training using 512-bit wide VPU.

Vectorizing equations

Vectorizing sampling step:

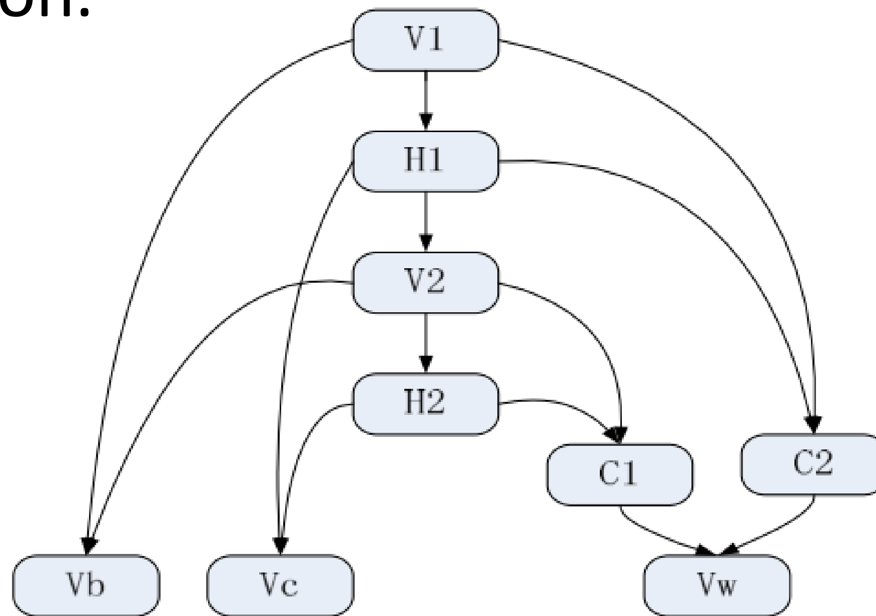
$$p(v \mid h) = vectorsig(b + W \bullet h)$$
$$p(h \mid v) = vectorsig(c + W \bullet v)$$

Vectorizing the update step:

$$W = W + \Delta W$$
$$b = b + \Delta b$$
$$c = c + \Delta c$$

Parallelization steps (2)

3. Parallelize matrix operations using Intel MKL.
4. Parallelize matrix operations based on sequence of execution.



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

Parallelize
Sparse Auto-encoder

Parallelization steps

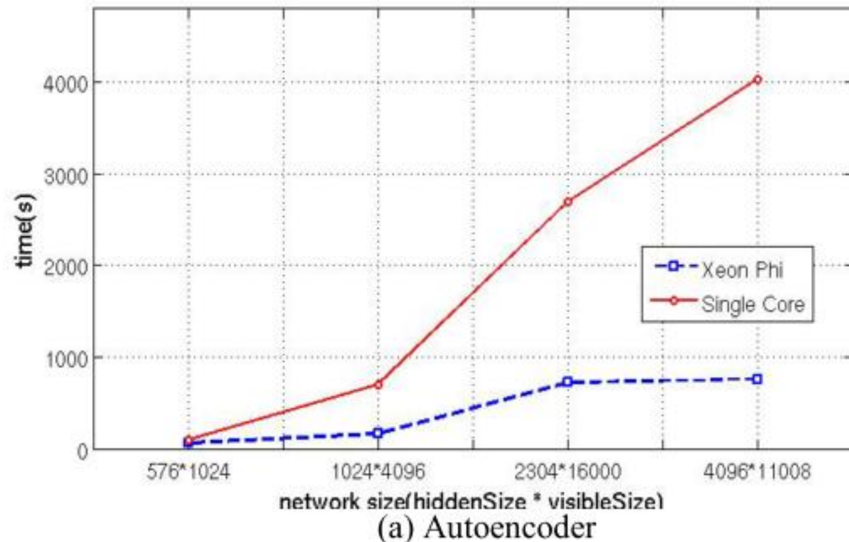
1. Limited scope in parallelization due to complexity of back propagation algorithm.
2. Use matrix multiplications tackled by Intel MKL packages.
3. Parallelize loops with OpenMP.

Performance Evaluation

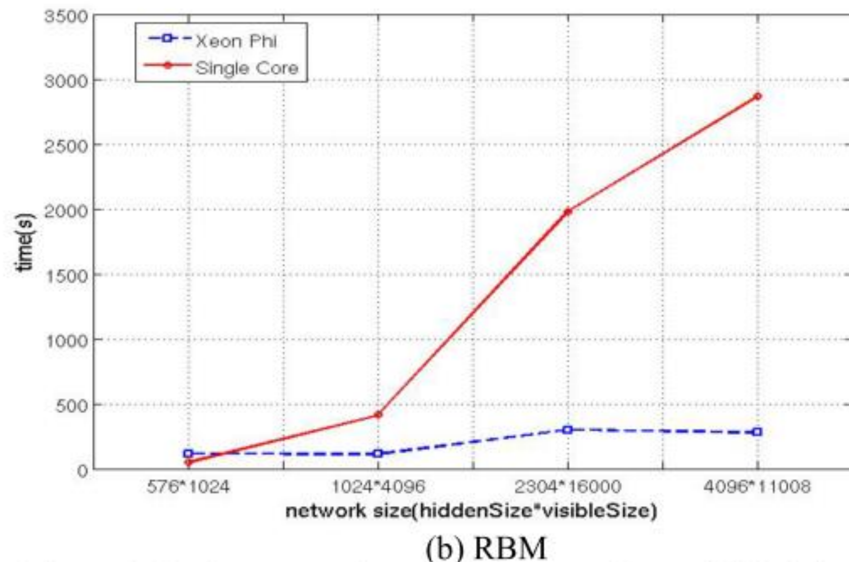
Evaluation criteria

- Comparison of Intel Xeon Phi vs. Intel Xeon CPU core.
- Comparison done in three aspects of network size, dataset size and batch size.
- Dataset consists of a range of handwritten and nature images. Training samples extracted by randomly taking patches from images.

Impact of Network Size



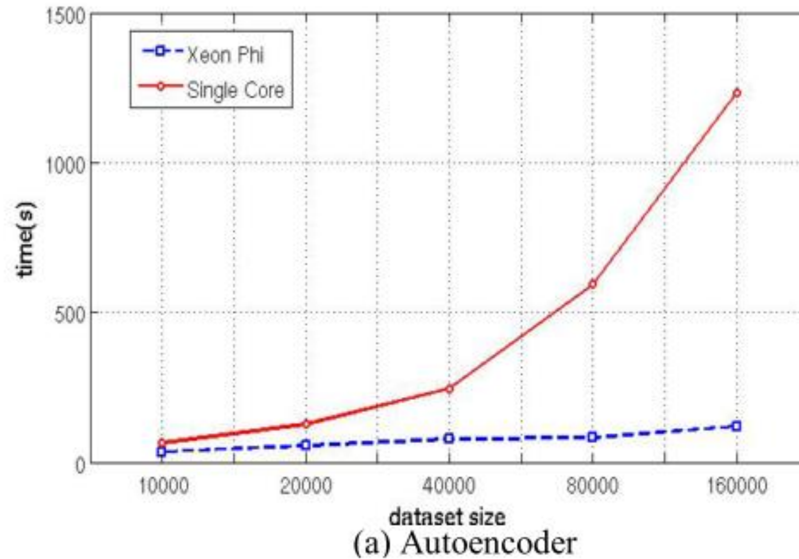
(a) Autoencoder



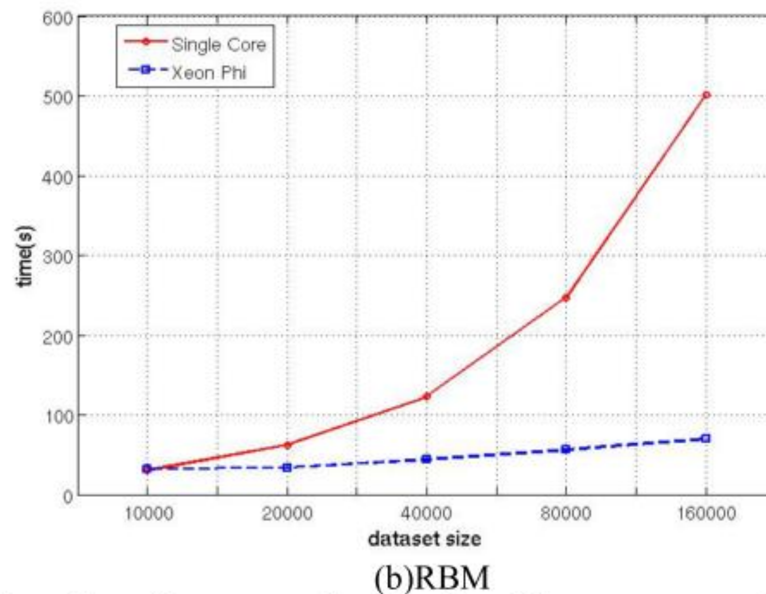
(b) RBM

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

Impact of dataset size



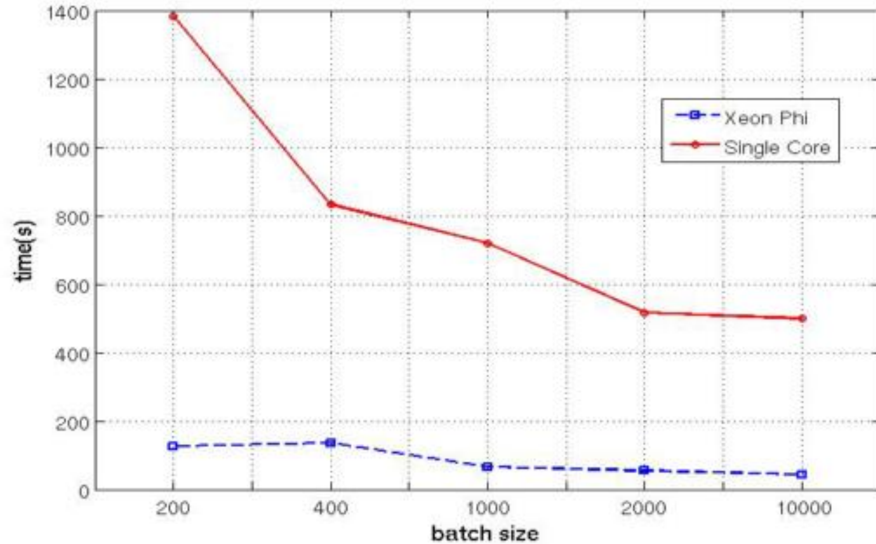
(a) Autoencoder



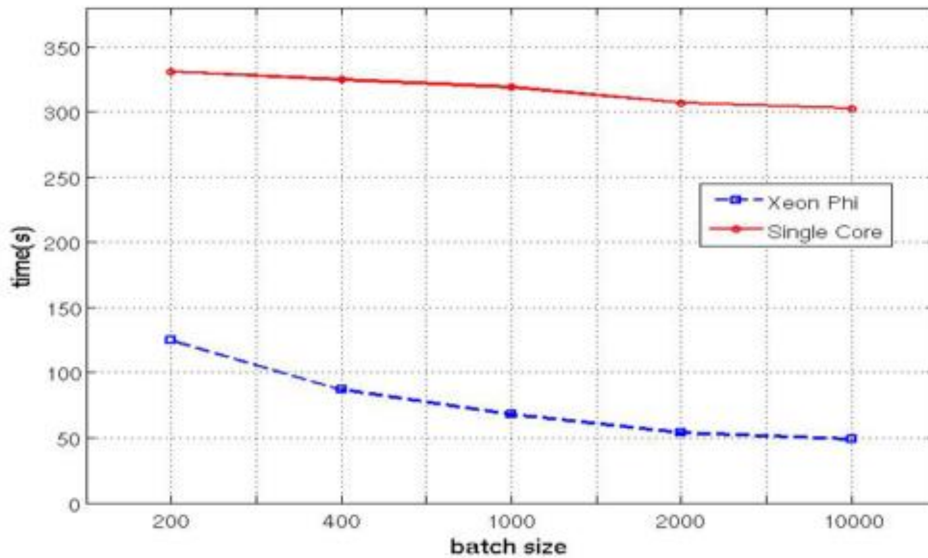
(b)RBM

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

Impact of batch size



(a)Autoencoder



(b)RBM

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

Successive performance optimizations

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

Conclusion

- Xeon Phi provides almost 300-fold increase in computation speed compared to sequential algorithm.
- Due to the general-purpose programming model for Xeon Phi, programmers can quickly transplant their original program on host machine to the Intel Xeon Phi platform

THANK YOU!