Optimizing Deep Neural Network on CPU+MIC Hybrid Platform

Abstract

Machine learning foucs on acquiring new knowledge or skills by computing.

DNN(Deep Neural Network), as an important area of research in machine learning, is widely applied in image recognition, speech recognition, natural language understanding, weather forecasts, gene expression, content recommendations, and so on.

However, DNN has its own disadvantages.

The most serious disadvantage is that in order to achieve the desired training effect, DNN requires a lot of data for trainning, which means that it takes long time for trainning.

Based on traditional DNN theory, made use of multi-core and multi-computer and MIC based on Intel's many-core processor for DNN acceleration to achieve short term can produce satisfactory effect of the design of neural networks, and programming on the actual machine. The experienment results achieves satisfactory effect.

This paper is divided into 3 main sections.

In the first chapter, introduces traditional DNN algorithm and its implementation.

In the second chapter, introduces the acceleration strategy and its effects on CPU platform.

In the third chapter, introduces the accelerated strategy and its effect on hybrid CPU+MIC platform.

Keywords—DNN,MIC,parallel computing

1.1 ALGORITHM AND PROGRAM ANALYSIS

1.1.1 Brief Introduction of DNN

Deep Neural Network (DNN) is an Artificial Neural Network (ANN) with multiple hidden layers of units between the input and output layers. Usually, the algorithm includes three major parts: the forward calculation, the error back propagation, and update of weight and bias. Similar to shallow ANN, DNN can model complex non-linear relationships. As one of typical algorithms for deep learning, DNN has been successfully applied in many domains in recent years, such as speech recognition and image recognition.

1.1.2 Analysis of the Original DNN Program

The DNN used in ASC16 consists of eight layers, including an input layer, six hidden layers and an output layer. The detailed network structure is shown in Figure 1 .

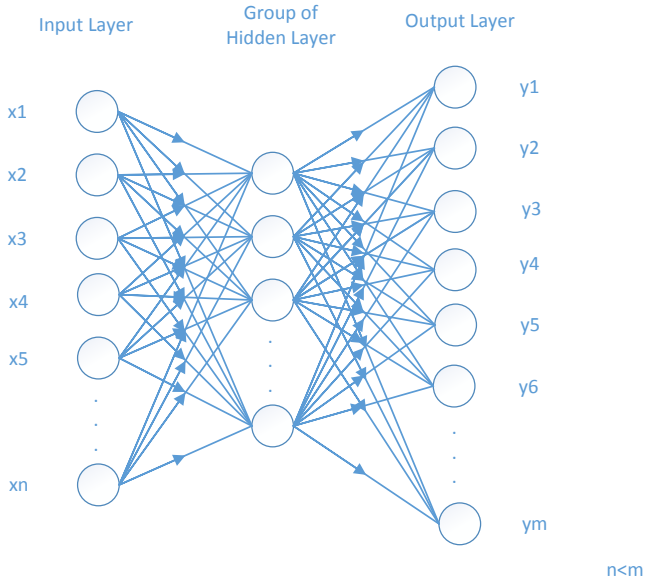


Figure 1 Detailed network structure

In order to make our description clear, here we simplify the representation of the parameters in nodeArg, as listed in the following table.

Table 1 Simplified representation of parameters in *nodeArg*

Original	Simplified representation	Meaning
nodeArg.numN	numN	Size of minibatch
nodeArg.dnnLayerNum	LayerNum	layer numbers
nodeArg.d_X	X	Input data of the input layer
nodeArg.d_T	T	Target label of input, used in backward
nodeArg.d_Y[i]	Y[i]	The neurons state of the layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)
nodeArg.d_W[i]	W[i]	The weight matrix of layer i (0<=i< NUM_LAYERS)
nodeArg.d_B[i]	B[i]	The bias of the Neurons on layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)
nodeArg.d_E[i]	E[i]	The training error of layer i+1 (hidden layers and output layer, 0<=i<NUM_LAYERS)

nodeArg.d_Wdelta[i]	Wdelta[i]	The weight delta matrix of layer i (0<=i<NUM_LAYERS)
nodeArg.d_Bdelta[i]	Bdelta[i]	The bias delta of the Neurons on layer i (hidden layers and output layer, 0<=i<NUM_LAYERS)

1.2 ANALYSIS OF FORWARD

(1) Basic idea

During forward calculation, DNN mainly use the data reading from the data file, the bias of the hidden layers and output layer, and the weight matrix to get the output of the hidden layers and output layer in order.

(2) Algorithm

The algorithm of Forward can be described as follows:

for i=0 to LayerNum-2 do

1. Initialize every row of Y[i] with B[i],

$$Y[i][j] = B[j\%col]$$

2.

$$Y[i] = \begin{cases} X \times W[i] + Y[i] & i = 0 \\ Y[i-1] \times W[i] + Y[i] & i > 0 \end{cases}$$

3. Sigmoid on hidden layers or Softmax on output layer

if(i==LayerNum-2) //output layer

$$Y[i][j] = \frac{e^{max(Y[i][k]) - Y[i][j]}}{\sum_{l=\frac{j}{col}}^{l=\frac{j}{col}+col-1} e^{max(Y[i][k]) - Y[i][l]}} \quad \frac{j}{col} \le k \le \frac{j}{col} + col - 1$$

else //hidden layer

$$Y[i][j] = \frac{1}{1+e^{-Y[i][j]}}$$

(3) Analysis

If we do not consider the difference inside add, sub, multiply, div and exp operation, it can be concluded that the computing complexity of this part is O (LayerNum*n^3).

Through our analysis, we could optimize the parts as shown in follows:

We should use parallel methods to compute Matrix multiplication.

In computer, we use Taylor expansion to compute e^x. The time it cost will be much, if we can compute it faster under premise of accuracy, that will be better.

From the perspective of mathematical, we could find that we do not have to compute $e^{maxY[i][k]}$ during softmax on output layer. Here is the detailed reason:

$$Y[i][j] = \frac{e^{max(Y[i][k]) - Y[i][j]}}{\sum_{l=j/com}^{l=j/col+col-1} e^{max(Y[i][k]) - Y[i][l]}} == \frac{e^{-Y[i][j]}}{\sum_{l=j/com}^{l=j/col+col-1} e^{-Y[i][l]}}$$

1.2.1.1 ANALYSIS OF BACKWARD

(1) Basic idea

During error back propagation, we mainly use target label of input, the output of the hidden layers and output layer got in the forward, and weight matrix to get the training error in reverse order.

(2) Algorithm

- Firstly, DNN uses the state of the output layer and the target label of input to get the training error of the output layer.

$$E[LayerNum-2][j] = \begin{cases} Y[LayerNum-2][j] - 1.0 & j\%col = T[j/col] \\ Y[LayerNum-2][j] & j\%col \neq T[j/col] \end{cases}$$

- Secondly, DNN uses the Back propagation to get the training error of the hidden layer in order.

  For i=LayerNum-3 to 0 do

  1. Propagate training error

     $E[i] = E[i+1] \times W[i+1]^T$

  2. Transform error, where $Y[i][j] \times (1 - Y[i][j])$ is the derivative value of sigmoid function on Y[i][j]

     $E[i][j] = E[i][j] \times Y[i][j] \times (1 - Y[i][j])$

(3) Analysis

Same to the forward part, computing complexity of this part is O (LayerNum*n^3).

Through our analysis, the parts we think we could optimize are shown as follows:

1) We should use parallel methods to compute matrix multiplication.
2) Also, since here we use matrix transposition, we should pay attention to the Cache hit ratio of matrix multiplication.

1.2.1.2 ANALYSIS OF UPDATE

(1) Basic idea

During this part, DNN mainly uses training error to compute bias delta and weight delta, as well as updating bias and weight matrix.

(2) Algorithm

- **Firstly**, DNN updates the bias and weight of the input layer.

  1. Update weight matrix

     $$Wdelta[0] = alpha \times X^T \times E[0]$$

     $$W[0][j] += Wdelta[0][j]$$

  2. Update bias

     $$Bdelta[0][j] = alpha \times \sum_{l=0}^{minibatch} E[0][l * col + j]$$

     $$B[0][j] += Bdelta[0][j]$$

- **Secondly**, DNN updates the bias and weight of the hidden layers.

  For i=1 to LayerNum-2 do

  1. Update weight matrix

     $$Wdelta[i] = alpha \times Y[i-1]^T \times E[i]$$

     $$W[i][j] += Wdelta[i][j]$$

  2. Update bias

     $$Bdelta[i][j] = alpha \times \sum_{l=0}^{minibatch} E[i][l * col + j]$$

     $$B[i][j] += Bdelta[i][j]$$

(3) Analysis

Similar to the forward part, the computing complexity of this part is also O (LayerNum*n^3).

Through our analysis, the parts we think we could optimize are shown as follows:

1) We should use parallel methods to compute Matrix multiplication.
2) In fact, we do not need to compute Wdelta and Bdelta separately which will waste our time.
3) Also, we need pay attention to matrix transposition and the computation of bias, here we need to notice the Cache hit ratio.
4) Interestingly, different from the forward calculation and the error back propagation, there is no data dependencies between adjacent layers in this part. So we can take parallelization strategy to update bias and weight of different layers.

1.2.1.3 OVERALL ANALYSIS

Based on the previous analysis, it can be concluded that:

(1) Major computations inside DNN algorithm is matrix multiplication. To achieve better performance, it is really necessary to use parallel methods to implement it;
(2) When updating W[i] or B[i], it only needs to get E[i] in the backward. Therefore, it is possible to parallelize these two parts.

1.3 PARALLELIZATION AND OPTIMIZATION ON CPU PLATFORM

1.3.1 Introduction

Based on previous analysis to DNN algorithm and the original program, we parallelize and optimize the DNN application on CPU platform in following aspects:

(1) We implement data parallelism for DNN application, that is, processing multiple mini-batches parallelly in multiple processes under restrictions of accuracy;
(2) Implement multithreading inside process to achieve fine-grained parallelism by using OpenMP;
(3) Use general-purpose optimizing approaches, such as data alignment, auto-vectorization, etc;
(4) Exploit optimized parameters and options for compilers and libraries.

Note: there are two workloads: one for debug and one for training. As the debug workload is suitable for analyzing the performance feature of the DNN application, the following experiments are based on the debug workload. And we only use training workload to compute speedup.

1.3.2 DATA PARALLELISM

1.3.3 DATA PARALLELIZABILITY OF DNN

The training process of DNN presents a non-convex optimization problem. Mini-batch gradient descent is used during this process. However, mini-batch gradient descent is a sequential process in nature. Thus, DNN is not well suited to parallel architectures.

To parallelize DNN, we use Asynchronous Stochastic Gradient Descent(ASGD) algorithm. The algorithm of ASGD is shown in Algorithm 1.

Algorithm 1 Asynchronous Stochastic Gradient Descent

Client	Server
Loop ***X*** = get next mini-batch ***W*** = fetch latest parameters from server **if *X*** is empty, **then** exit	**Loop** ***request*** = Wait for request from client **if *request*** is fetching data, **then** send ***X*** and ***W***.

Delta = calculate gradient based on ***X*** and ***W*** Upload ***Delta*** **end loop**	**if** request is upload ***delta***, **then** update ***W*** based on the uploaded ***delta***. **end loop**

The program runs faster when applying this algorithm. However, the accuracy rate decreases when we increase the number of processes. According to On Parallelizability of Stochastic Gradient Descent for Speech DNNs, there are two factors affecting the parallelizability: mini-batch size and data compression①. Both increasing mini-batch size and decreasing data compression can increase parallelizability. For us, we explore the former.

The main method to increase mini-batch is "AdaGrad", which leads to more mature models earlier, thus allowing for larger N. Our implementation is based on the description in Large Scale Distributed Deep Networks②. We applied AdaGrad to server side. To be more specific, we use a separate learning rate for each parameter. Let $\mu_{i,K}$ be the learning rate of the i-th parameter at iteration K and $\Delta w_{i,j}$ is its gradient. We set $\mu_{i,K} = \frac{\gamma}{\sqrt{\sum_{j=1}^{K} \Delta w_{i,j}{}^2}}$. The value of γ is the constant scaling factor for all learning rates. We choose a suitable γ from a lot of valid values by using an automatic script.

1.3.3.1 Implementation of Asynchronous SGD (with AdaGrad) Based on MPI

Message Passing Interface (MPI) is widely used technology in parallel software field. It enables programmers to write programs that run on distributed systems. The first version of our asynchronous stochastic gradient descent implementation is based on MPI.

In our implementation (its concept architecture is shown in Figure 2), there are two kinds of processes: server and client. The process whose process identifier(PID) is zero behaves as a server process. It reads chunk data, distributes mini-batch to clients and maintains the latest value of parameters. Other processes are client processes, who compute the data received and transfer the data to server and get new mini-batch and new references. When there is no data, client processed end and server writes result.

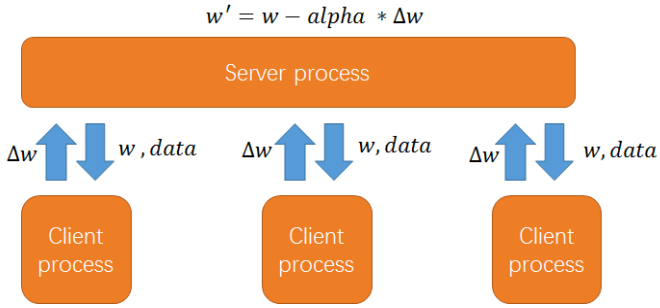


Figure 2 Data parallelization of MPI model

We tested our MPI version of ASGD, and collected the accuracy data of it. The result can be seen from Figure 3 and Figure 4.

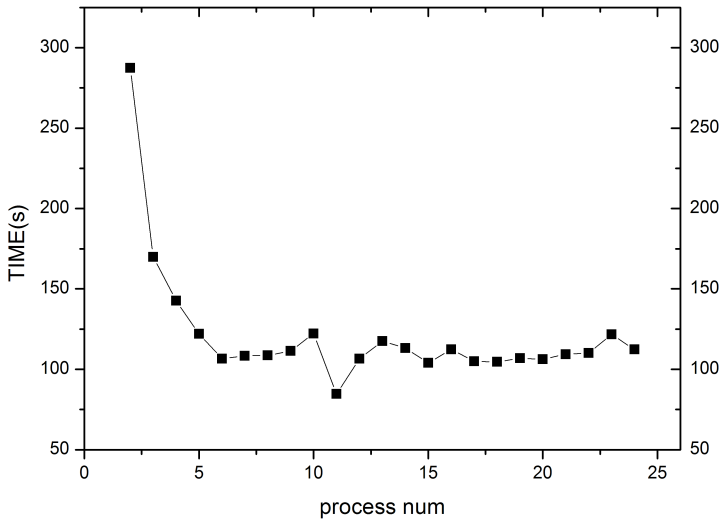


Figure 3 time of ASGD using MPI with different work threads

Figure 3 shows that data parallelism could speed up the original program remarkable. More specifically, the time which program costs reduce sharply when the process number is less than 6. At the same time. the performance of the program keeps stable when the process number is greater than 5.

According to Figure 4, the accuracy is unsatisfactory when work threads exceeds 8. In summarize, our experiment validate that data parallelism is an effective method to boosting the performance of the program under the condition that the number of threads cannot exceed the parallelizability of the program.

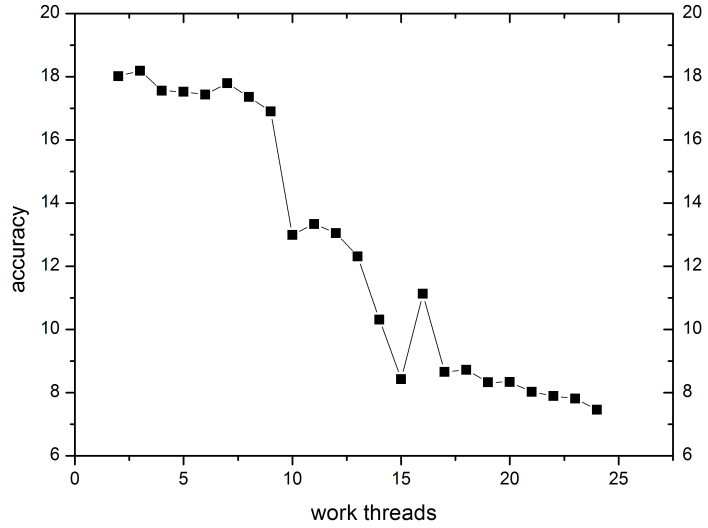


Figure 4 accuracy of ASGD using MPI with different work threads

1.3.3.2 OPENMP IMPLEMENTATION OF ASGD

After tuning the performance of MPI version, we find that the necklace of our implementation is communication cost. In our design, parameter server sends bunch data to clients, which causes several copy operations. To avoid unnecessary copy operations, we rewrite ASGD in OpenMP. Here is our Data parallelization of OMP model.

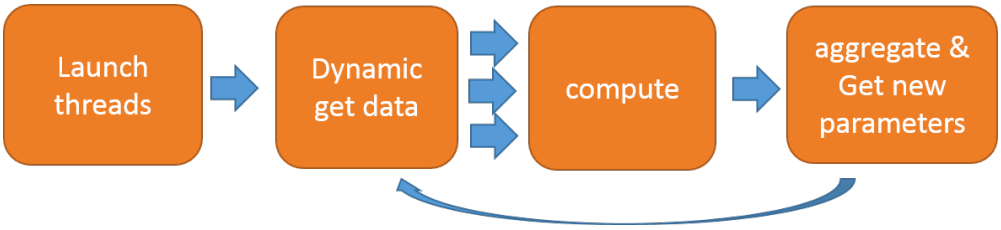


Figure 5 Data parallelization of OMP model

OpenMP based ASGD implementation does not have a server thread. Instead, all of threads share one central model which contains the latest parameters. In addition, each thread has its own local replica.

We tested the performance of our implementation based on OpenMP when using 1-8 to eight work threads. (Due to the results of our MPI experiments, we do not test the situation that using more than 8 threads, for its accuracy is unsatisfied.). From this, we can find that when threads are around 3, the program works well.

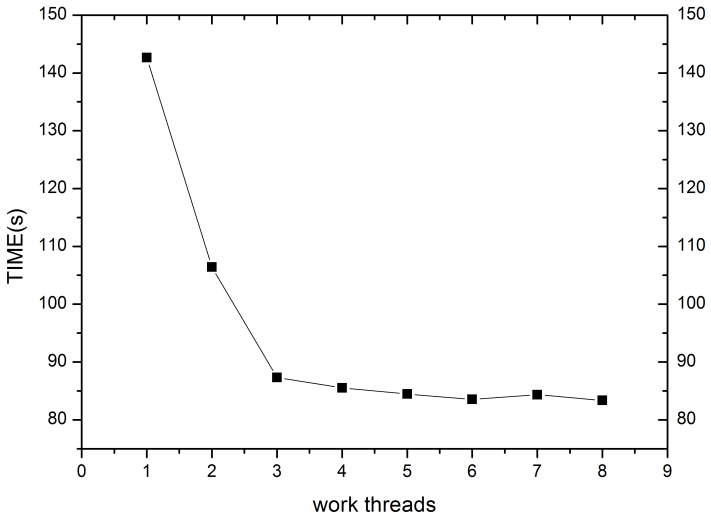


Figure 6 Time when using different number of work threads

And there is the accuracy statistics when using different number of work threads. As we can see from Figure 7, there are few differences when using 1-3 work threads. Then, the accuracy drops down rapidly when we launch more than 4 threads, which indicates the number of threads exceeds the parallelizability.

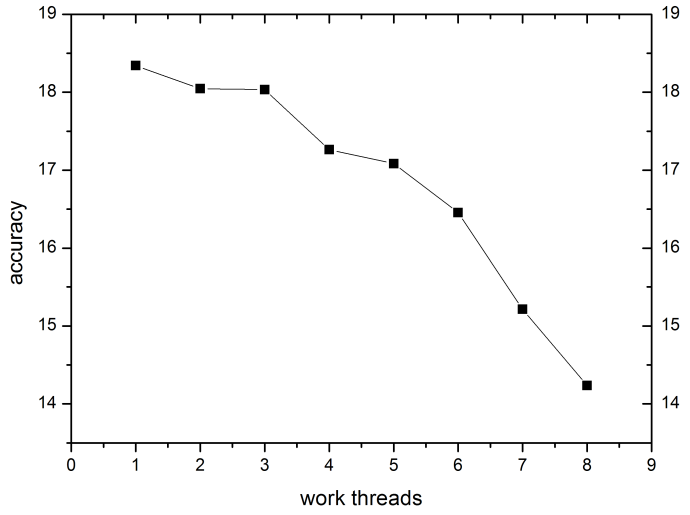


Figure 7 Accuracy when using different number of work threads

To maximize the number of threads, we use 3 work threads as it isn't exceeds the limit. Then we tunend the number of MKL work threads. Intel MKL can dynamiclly lunch suitable number of threads. However, it is not perfect. We set the number of MKL threads staticlly, testing the performance of the program. Interestingly, we find 8 MKL threads reachs the highest performance.

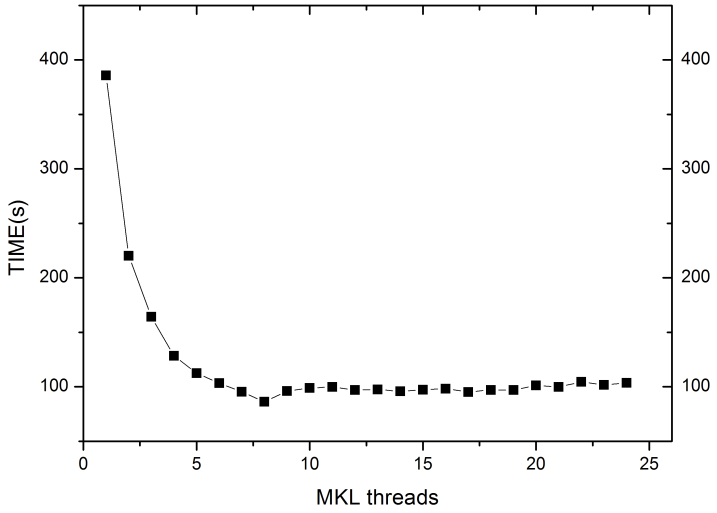


Figure 8 the time of different MKL threads

1.3.4 COMPILER AND ENVIRONMENT RELATED OPTIMIZATION

1.3.4.1 COMPILER

The default compiler of the origin code is g++ which supports various platforms. An alternate option is ICPC, a compiler from Intel. ICPC can generate highly optimized binary code on Intel platforms. We compared the performance of compiled programs of g++ and ICPC. The result is shown in Figure 9(linked with sequential version of Intel MKL).

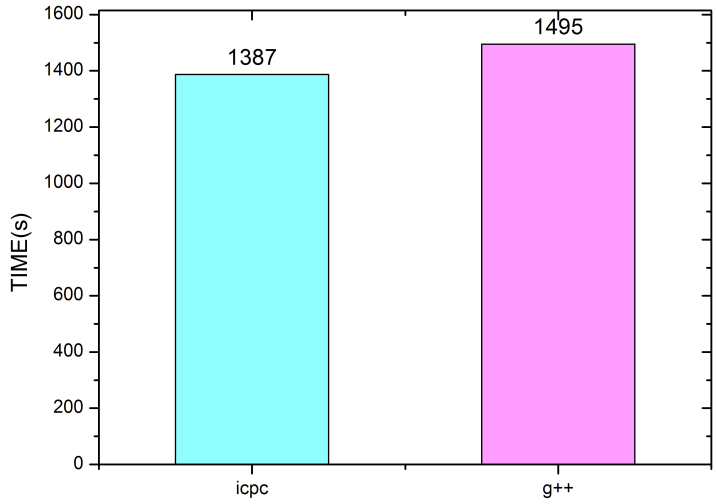


Figure 9 Execution time of the program compiled by *G++* and *ICPC*

As can be seen in Figure 9, compiling program using ICPC leads to performance improvement. Therefore, we replace g++ with ICPC in order to produce more effective binary code.

1.3.4.2 NUMBER OF MKL WORKING THREADS

The original program uses BLAS (Basic Linear Algebra Subprograms) for computing matrix multiplication. We choose Intel MKL (Math Kernel Library), which is a high performance library developed by intel, as the implementation of BLAS. MKL implements multiple threads matrix multiplication. Thus, the number of work threads used by MKL is an important factor that influences the speed of the program. We tested the performance of the program under different number of MKL threads. The results are shown in Figure 10.

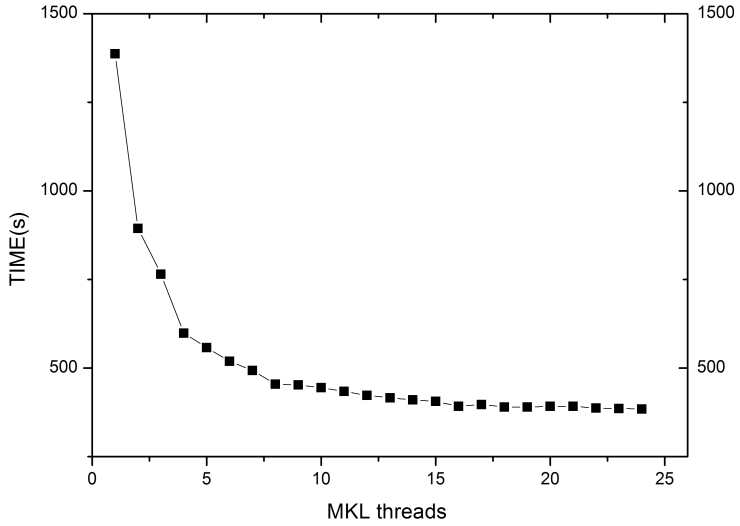


Figure 10 Execution time under different number of MKL threads

It is obvious that the more threads MKL uses, the faster the program runs. But MKL cannot archive linear speedup. The speedup is not very obvious when the number of threads exceeds around 8. So it may reach higher performance if we can execute three processes (each process has 8 threads for computing matrix multiplication) in parallel.

1.3.4.3 COMPILER OPTIMIZATION OPTIONS

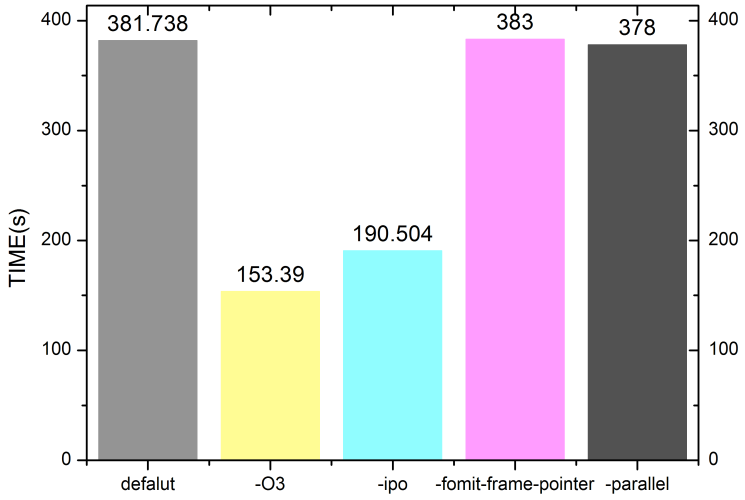
Modern compilers support various optimization strategies. Some aggressive optimizations may cause worse performance and others may lead to better performance. To find the most suitable optimization options, we did some experiments. All of the options we tested are listed in Table 2.

Table 2 Optimization Options and Details

Optimization Options	Description
default	Does not add any extra optimization flags.
-O3	Enables more aggressive loop transformations. Note that the O3 optimizations may not cause higher performance.
-ipo	Enables inter-procedural optimization.
-fomit-frame-pointer	Enables EBP as a general purpose register
-parallel	Auto parallelize

As we can see from Note:linked with MKL multi-threads version

Figure 11, -O3 option cause higher performance because of its aggressive optimization. In addition, using EBP as a general purpose register does not lead to better performance, either. As for inter-procedural optimization, it really has obvious effects. Inter-procedural optimization focus on analyzing the possible value sets of pointers, which enables compiler to generate more effective binary code. As a result, the performance of the program is obviously improved. Interestingly, as a result of auto parallelize optimization, the performance of the program is improved although speedup is less than 1%.



Note:linked with MKL multi-threads version

Figure 11 The effects of different optimization options

In conclusion, we enable -O3, –ipo and –parallel options for better performance (it only takes 149.504 seconds).

1.3.5 Data Alignment and Vectorization

By analyzing source code, we find that there are lots of operation applying for space during dnn_utility.cpp file. To our dismay, the operation is new(), which doesn't make data forced alignment. In order to reduce the CPU access to memory, we change it to _mm_malloc(). Besides, we find that most of the loop statements don't have data dependencies, for example:

```
1  for (int i = 0;i < row*col; ++i)
2  {
3          Y[i] = 1.0f/(1.0f + expf(-Y[i]));
4  }
```

We could easily find there is no dependencies between Y[i] and Y[j] when i! = j. So we can take

compiler options to make it automatic vectorization, like -xHost, so that we can make our program faster.

Based on the above analysis, we did some related experiment, the results are as follows:

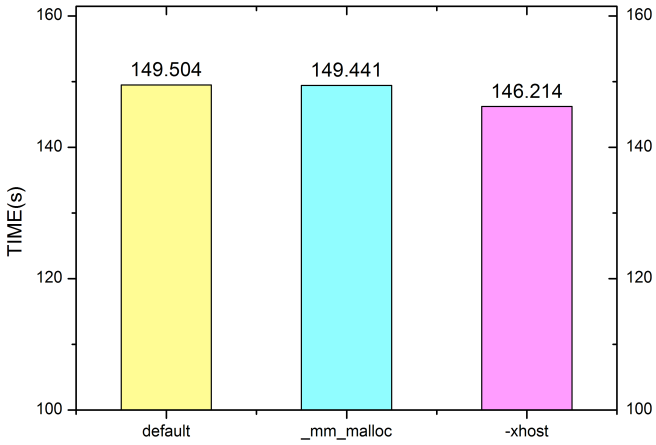


Figure 12 the effects of data alignment and vectorization

-xHost option enables all of the instruction sets supported by the host, which means it enables auto-vectorization, an optimization method that rewrite loops in SIMD instructions. As can been seen from Figure 12, auto-vectorization leads to a small performance improvement. Note that the default version used here applies -O3, -ipo and -parallel.

1.3.6 Final Execution Time and Speedup on ASC16 Remote CPU Platform

After all the experiments and optimizations on CPU, it takes 87.042s for our CPU DNN program to execute the debug workload. It takes 310.152s to execute the training workload. Comparing to the original sequential program, the final speedup of DNN on CPU platform is:

(1) Debug workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU} = \frac{1495.367}{87.042} = 17.180$$

(2) Training workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU} = \frac{4860.571}{310.152} = 15.672$$

1.4 Hybrid Parallelization and Optimization on CPU+MIC Platform

In this section, we will introduce our optimization on MIC. Mainly talking about the asynchronous transfer optimization and data parallelization's difference with it on CPU. Here is our model:

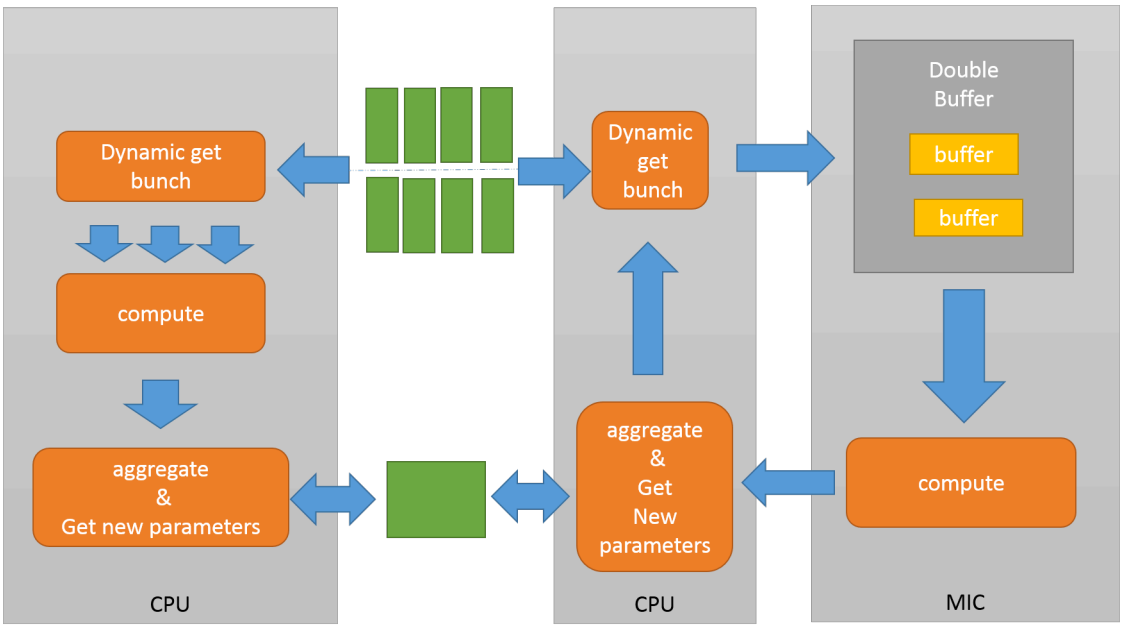


Figure 13 CPU+MIC model

1.4.1 Dynamic Load Balance

For CPU+MIC Heterogeneous computing platform, the strategy of distributing work to the work threads on CPU and MIC is an important point.

A sample strategy is dividing training data statically, which means the data that will be computed by CPU or MIC has already been decided before the training process. As a result, the workload on CPU and MIC may be unbalanced.

To avoid this problem, we apply a dynamic strategy to distribute training data. To be more specific, the training data will be requested by idle work threads. In detail, when a thread requests data, it will lock the data buffer and get a bunch of samples from it. After that, it will unlock the data buffer, enabling other threads to read data. We implement a double-buffer mechanism on MIC platform. has a double-buffer to enable asynchronous data transfer from CPU to MIC.

Here is the result of the experiment about the speed and accuracy when there are 2,3, and 4 work threads.

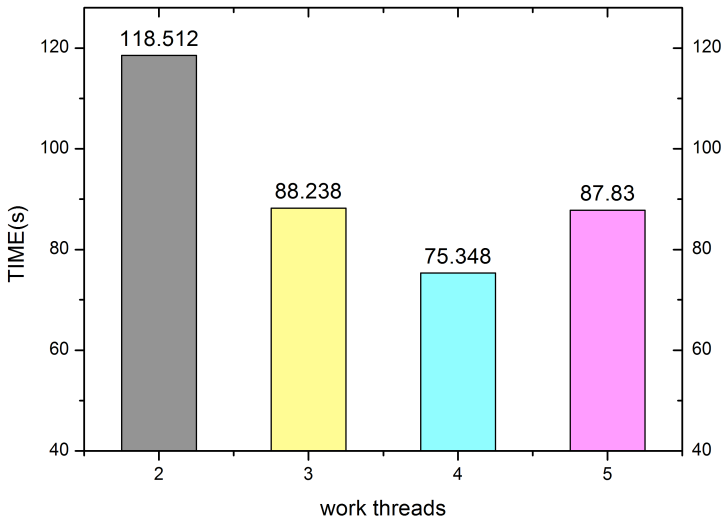


Figure 14 Performance under different number of work threads on CPU+MIC

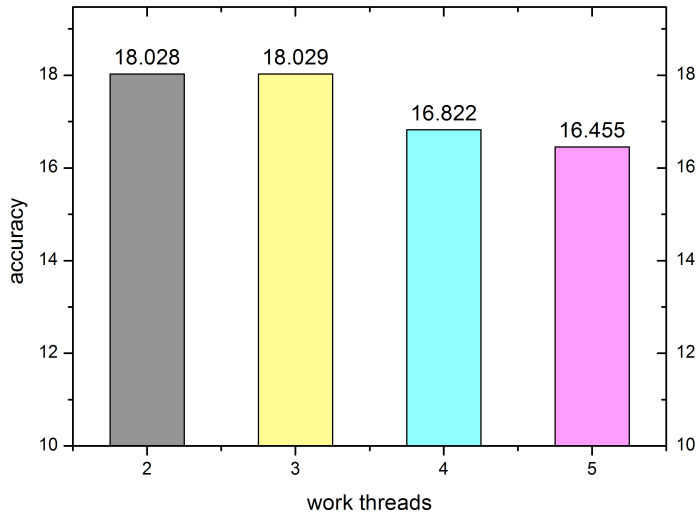


Figure 15 Accuracy under different number of work threads on CPU+MIC

1.4.2 Asynchronous Transfer Optimization

When computing on MIC, we apply two main methods to parallelizing it: setting double-buffer to get new data and parameters and merging data asynchronous. We will discuss them in the following sections.

1.4.3 Double-Buffer

We use a double-buffer scheme to support asynchronous data transfer between CPU and MIC. One of the buffers is used by the work thread, and the other is used for storing data that is asynchronous transferred from host. Once computing and data transfer completed, the two buffers are swapped.

1.4.3.1 Asynchronous Aggregate Gradient

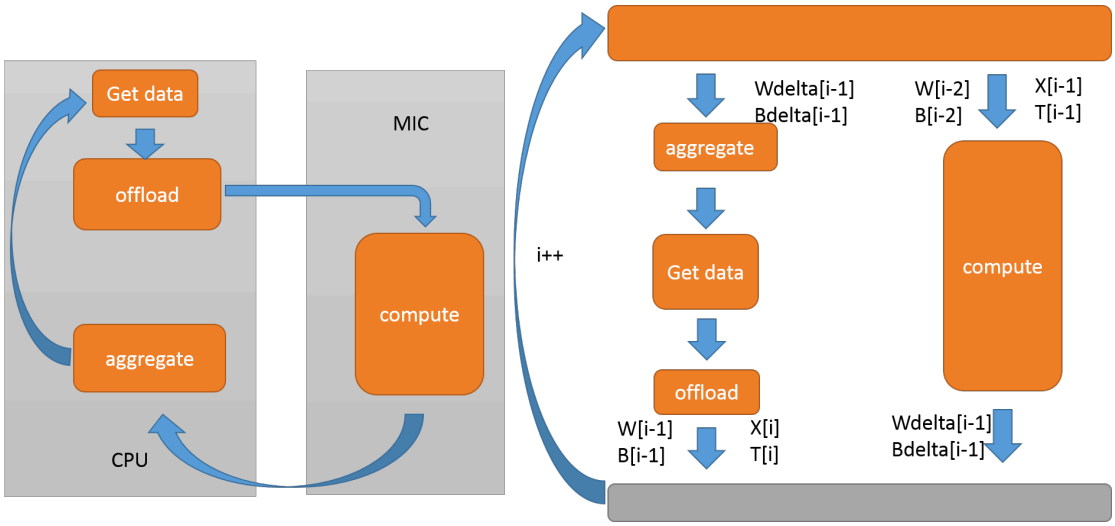


Figure 16 Difference between synchronous and asynchronous

On the left side of Figure 16 shows the synchronous aggregate gradient process. Even through the data transfer from CPU to MIC is asynchronously, we have no way but to wait the time when CPU have already get the data from MIC. And only after that time, MIC could get its new parameters from CPU. That will waste a lot of time.

But when we use the model on the right, the problem has been solved. When MIC is computing, we can fetch the parameters on the MIC got in the last iteration. And when MIC has ended its

computation, what it needs to do is just change its nodeArg's d_X, d_T, d_W and d_B 's pointer and let it point to the buffer where contains new data which has been transferred by CPU. And then it can start its next iteration. Thus, MIC just need to wait the pointer changed. It will save much time!

But there is also a problem that we can found. It is that our d_W, d_B is a little more delayed than it on CPU. So when we aggregate the d_Wdelta and d_Bdelta to the nodeArg shared by all threads, it will lose some accuracy. So we aggregate the d_Wdelta and d_Bdelta computed by CPU a little different from MIC. According to the results of our experiments (shown in Figure 17), it works better.

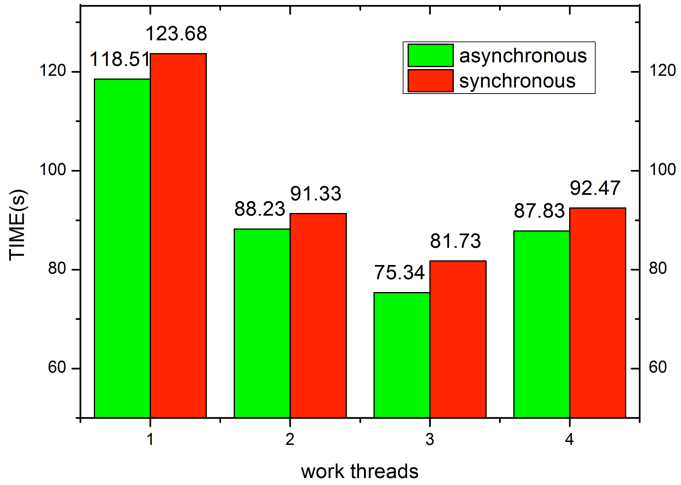


Figure 17 comparison between synchronous and asynchronous aggregate gradient

1.4.4 Miscellaneous Optimization

1.4.4.1 Reuse Memory

Through a series of experiments, we found that memory allocation on MIC is an expensive operation. If the code requires memory allocation in each iteration, the benefit of computing data on MIC would be affected. To avoid this problem, we only execute memory allocation just the first iteration and reuse them during the following iterations.

1.4.4.2 Cache Optimization

The cache architecture of MIC is different from that of CPU. Each core of MIC has 32KB L1 instruct cache and 32KB L1 data cache, the same as CPU. In addition, MIC has 512KB L2 cache shared by all cores, which is bigger but slower than L2 cache on CPU. So, we optimize cache hit ratio on MIC, getting remarkable effect on MIC.We can find that improving cache hit ratio is a good way to optimize the performance of MIC.

1.4.5 Final Execution Time and Speedup on Remote CPU+MIC Platform

After all the experiments and optimizations, it takes 75.348s for our CPU+MIC DNN program to execute the debug workload. As for the training workload, it takes 285.265s. Comparing to the original sequential program, the final speedup of DNN on CPU+MIC platform is:

(1) Debug workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU+MIC} = \frac{1495.367}{75.348} = 19.846$$

(2) Training workload

$$\text{Speedup} = \frac{Execution\ time\ of\ original\ sequencial\ program}{Excution\ time\ of\ optimized\ parallel\ program\ on\ CPU + MIC} = \frac{4860.571}{285.265} = 17.039$$

.

① H. F. J. D. G. L. D. Y. Frank Seide, "ON PARALLELIZABILITY OF STOCHASTIC GRADIENT DESCENT FOR SPEECH DNNS," from IEEE International Conference on Acoustic, Speech and Singal Processing, 2014

② J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M.A.Ranzato, A. Senior, P. Tucker, K. Yang, A. Y. Ng, "Large Scale Distributed Deep Networks," NIPS, 2012

Optimizing Deep Neural Network on CPU+MIC Hybrid Platform

（中文提纲）

题目：CPU+MIC 高性能计算平台上的深度神经网络的并行加速优化

主要内容：

机器学习（Machine Learning）是一门专门研究计算机怎样模拟或实现人类的学习行为，以

获取新的知识或技能，重新组织已有的知识结构使之不断改善自身的性能的学科。

DNN 深度神经网络作为机器学习的一个重要研究领域，在图像识别、语音识别、自然语言理解、天气预测、基因表达、内容推荐等等方面都有着极其广泛而有效的应用。

但是，DNN 深度神经网络也有着自己生而存在的缺陷。

其中最大的缺陷，就在于要想达到理想的训练效果，需要大量的数据来训练神经网络。但是

由于 DNN 天生存在的缺陷，导致训练时间过于漫长，不利于在实际应用中推广。

本文在基于传统 DNN 的基础之上，提出了利用多核以及多机，并基于英特尔的 MIC 众核处理器

对于 DNN 加速，以达到短期就可训练出较为满意的神经网络的效果的设计方案，并在实际机器上进行了编程实现。效果基本符合预期，可以达到比较满意的训练效果。

本文分为 3 个主要章节

在第一章，主要介绍了传统 DNN 的算法以及其编程实现

在第二章，主要介绍了在 CPU 之上的加速策略及其效果

在第三章，主要介绍了在加入 MIC 卡之后新增的加速策略及其效果