Speedup of deep neural network learning on the MIC-architecture

Evgeniia Milova, Svetlana Sveshnikova, Ivan Gankevich
Dept. of Computer Modeling and Multiprocessor Systems
Saint Petersburg State University
Saint Petersburg, Russia
milova_evg@mail.ru, svetasvesh@yandex.ru, i.gankevich@spbu.ru

POSTER PAPER

***Abstract*—Deep neural networks are more accurate, but require more computational power in the learning process. Moreover, it is an iterative process. The goal of the research is to investigate efficiency of solving this problem on MIC architecture without changing baseline algorithm. Well-known code vectorization and parallelization methods are used to increase the effectiveness of the program on MIC architecture. In the course of the experiments we test two coprocessor data transfer models: explicit and implicit one. We show that implicit memory copying is more efficient than explicit one, because only modified memory blocks are copied. MIC architecture shows competitive performance compared to multi-core x86 processor.**

***Keywords*—DNN, optimisation, parallel computing, vectorization, offload, Xeon Phi, coprocessor, many-core architecture**

I. INTRODUCTION

A perceptron that features more than one hidden (learning) layer is called a Deep Learning Network. To train this network the method of backpropagation is usually employed, which is an iterative gradient algorithm, used for minimizing the errors of neural network learning.

An algorithm iteration consists of three main step functions: *dnnForward* puts a training sample through the network, yielding a certain result; *dnnBackward* determines the error, then, in each layer of the network, starting with the penultimate, calculates the correction of weight coefficients for each node; *dnnUpdate* updates the weight of neurons according to a previously calculated adjustment. Network learning ends when an error reaches its specified minimum accepted level. Such network demonstrates remarkable results in many areas, including those of voice and image recognition. However its deficiency is in a very lengthy learning process. Therefore, it has been decided to investigate the issue of effective functioning of this type of networks on parallel computational architectures. For testing purposes an 8-layered neural network was taken (1 input, 6 hidden, 1 output). In the interest of result analysis the following parameters were chosen: neural network learning speed and precision of object recognition.

The task was carried out on the Intel Xeon processor (see Table I for specifications). First, the task was carried out employing only one core. Then the code optimization was carried out to prepare for the launch on the parallel architecture. The decision was made to test the effectiveness of Many Integrated Core (MIC) architecture [**?**] in light of finding the solution to the task. This architecture features a large number of x86 cores in one co-processor, coupled with the main processor. Intel Xeon Phi co-processor specifications are also shown in Table I.

TABLE I
COMPUTATIONAL PLATFORM SPECIFICATIONS.

Processor	2xIntel Xeon CPU E5-2695 v2 (12 cores, 2 streams by core, 2.40 Ghz)
Coprocessor	Intel Xeon Phi-5110P (60 cores, 4 streams by core, 1.052 Ghz)

II. RELATED WORK

In [**?**] Geoffrey Hinton and Ruslan Salakhutdinov described deep neural network training algorithms making deep learning a new direction of research in the field of neural networks. The idea of deep learning is based on biological peculiarities of how human brain work. In essence, deep neural network is a perceptron with more than one hidden layer. An algorithm of error back-propagation is employed to train such neural network [**?**]. This algorithm represents iterative gradient descent which minimises training error [**?**].

There are a number of articles that describe how to speed up learning process. In [**?**], [**?**], [**?**] the speed up performed by increasing the number of cores that are running the neural network compared with the work of one core there was a significant speedup. Besides, there are other ways to increase the speed, such as vectorization.

In [**?**], [**?**] the author analyzes the effect of vectorization to the speed up, and it is shown that the set of parallelization and vectorization can also give a performance boost.

III. METHODS

A. Parallel architecture code optimization

Each Intel Xeon processor core and Intel Xeon Phi coprocessor core contains a vector processing unit. It is possible to process 16 32-bit integers or 8 64-bit integers in a single processor cycle. The code vectorization during array processing yields a significant potential for program acceleration when launching on parallel architectures. Vectorization was carried out by the technology of Array Notation extension Intel Cilk Plus. Intel Cilk Plus is a C/C++ extension for parallel support, implemented in the Intel compiler.

For working with the array the following construction is used instead of cycle *for* in Array Notation: `array[start_index : length]`. For example, the following code adds i[th] element of W array to each i[th] element of W_{delta} array

```
W[0:count] += Wdelta[0:count];
```

With Array Notation it is possible to vectorize an execution of more complicated operations. The search of maximum element in array is performed using the expression `__sec_reduce_max`

```
const float max =
__sec_reduce_max(in_vec[base:ncols]);
```

Summing the elements of the array performed by expression `__sec_reduce_add`

```
const float sumexp =
__sec_reduce_add(in_vec[base:ncols]);
```

After vectorization, the code was launched on the processor Intel Xeon on 12 cores (24 threads). Performance increased by 14.5 times, compared to launching the non-vectorized code on one core.

B. Porting the code on MIC architecture

An offload-model of data transfer was used for working with Intel Xeon Phi. In offload mode the code block highlighted by the directive `#pragma offload target (mic)` is executed on the coprocessor, the rest of the code is executed on the main processor. The size of the coprocessor memory for each variable must be specified. Offload mode supports 2 data transfer models: explicit and implicit.

1) Explicit data transfer model: By using the explicit model, a programmer specifies which variables should be copied onto coprocessor. The copy destination is also specified. The advantage of this model is the possibility of successful code compilation by any compiler besides Intel Compiler. Unknown directives will be simply ignored, generating no errors, the code will be compiled and ready for work on x86 architecture only.

The functions of neural network learning are called within two nested loops. Inner loop was marked for execution on the coprocessor.

```
while (FetchOneChunk(cpuArg, chunk)) {
    ...
    #pragma offload target (mic:0)
    while (FetchOneBunch(chunk, bunch)) {
        dnnForward (bunch);
        dnnBackward(bunch);
        dnnUpdate  (bunch);
    }
}
```

One problem that we faced during optimization was that there is no simple way to transfer two-dimensional arrays to MIC and back. All in all, we managed to do this with help of preprocessor macros and careful calculation of array sizes from source code analysis. Unfortunately, an explicit data transfer model contains a drawback: It supports a bitwise data copy only, and a structure containing field-pointers cannot be copied. In this program all characteristics of neural network are contained within the `bunch` structure. It is specified as an argument for functions being sent to the coprocessor for execution. This structure contains field-pointers. In order to copy the `bunch` structure properly to the coprocessor, its each field must be copied separately and then the whole structure assembled again on the coprocessor.

```
#define COPY_FLOAT_ARRAY_IN(arr) \
    float* arr ## 0 = bunch.arr[0]; \
    float* arr ## 1 = bunch.arr[1]; \
    float* arr ## 2 = bunch.arr[2]; \
    float* arr ## 3 = bunch.arr[3]; \
    float* arr ## 4 = bunch.arr[4]; \
    float* arr ## 5 = bunch.arr[5]; \
    float* arr ## 6 = bunch.arr[6]

#define COPY_FLOAT_ARRAY_OUT(arr) \
    bunch.arr[0] = arr ## 0; \
    bunch.arr[1] = arr ## 1; \
    bunch.arr[2] = arr ## 2; \
    bunch.arr[3] = arr ## 3; \
    bunch.arr[4] = arr ## 4; \
    bunch.arr[5] = arr ## 5; \
    bunch.arr[6] = arr ## 6

...
COPY_FLOAT_ARRAY_IN(d_B);
COPY_FLOAT_ARRAY_IN(d_Wdelta);
COPY_FLOAT_ARRAY_IN(d_Bdelta);
COPY_FLOAT_ARRAY_IN(d_Y);
COPY_FLOAT_ARRAY_IN(d_E);

#pragma offload target(mic:0) \
```

```
mandatory \
inout(d_W0: length(
bunch.dnnLayerArr[0] *
bunch.dnnLayerArr[1])) \

inout(d_W1: length(
bunch.dnnLayerArr[1] *
bunch.dnnLayerArr[2])) \

inout(d_W2: length(
bunch.dnnLayerArr[2] *
bunch.dnnLayerArr[3])) \

inout(d_W3: length(
bunch.dnnLayerArr[3] *
bunch.dnnLayerArr[4])) \

inout(d_W4: length(
bunch.dnnLayerArr[4] *
bunch.dnnLayerArr[5])) \

inout(d_W5: length(
bunch.dnnLayerArr[5] *
bunch.dnnLayerArr[6])) \

inout(d_W6: length(
bunch.dnnLayerArr[6] *
bunch.dnnLayerArr[7]))

//similarly for d_B, d_Wdelta,
d_Bdelta, d_Y, d_E

{

// loop of training

COPY_FLOAT_ARRAY_OUT(d_W);

//similarly for d_B, d_Wdelta,
d_Bdelta, d_Y, d_E

}
```

Experiments on test dataset demonstrated that this data transfer model is not adequate for this task. The program runs slightly faster, than on one core processor and 12 times slower than on all of the cores (coprocessor II). Therefore it has been decided to use implicit data transfer model on coprocessor.

2) Implicit data transfer model: The basic principle for the implicit model is the usage of memory shared between CPU and MIC in the virtual address space. This method allows transferring of complex data types, thus ridding of the limitation of bitwise copying occurring on explicit model. Program conversion implemented as follows:

1) Data was marked by the `_Cilk_shared` keyword that allows allocating it in the shared memory.
   ```
   bunch.d_B[i-1]=(_Cilk_shared
   float*)_Offload_shared_malloc(size);
   ```
2) Functions used inside the learning cycle were marked as shared:
   ```
   #pragma offload_attribute(push,\
       _Cilk_shared)
   ...
   #pragma offload_attribute(pop)
   ```
3) A separate function was created for the neuron network learning loop for using it in shared memory:
   ```
   _Cilk_shared void
   dnn(Bunch& bunch, Chunk& chunk)
   {
       while(FetchOneBunch(chunk, bunch))
       {
           dnnForward (bunch);
           dnnBackward(bunch);
           dnnUpdate (bunch);
       }
   }
   ```
4) A function sent for execution to the coprocessor was marked by the command `_Cilk_offload`:
   ```
   _Cilk_offload dnn(bunch, chunk);
   ```

IV. RESULTS

It is worth mentioning, that implicit working model proved to be easier to program, compared to the explicit model and enabled to reach an acceptable time of learning. It was accelerated by 13.5 compared to the sequential version.

Table II describes results of experiments. Test number column indicates a test case from the following list:

1) Initial non-optimized version that was run on one-core processor.
2) Optimized version that was run on multi-core x86 processor (OpenMP directives and vectorization).
3) Optimized version that was run on MIC architecture (OpenMP directives, vectorization and explicit data transfer).
4) Optimized version that was run on MIC architecture (OpenMP directives, vectorization and implicit data transfer).

Other columns in order from left to right: architecture, the number of threads, time of execution, speed up (compared with initial non-optimized version) and learning accuracy for each test.

V. DISCUSSION

In the course of research, deep neural network learning has been tested on a variety of computer architectures. Results are shown on Table II. The MIC version does not increase performance compared to parallel version of the processor. It is

TABLE II
COMPARING WORKING TIME AND PRECISION OF LEARNING.

Test number	Arch.	Threads	Time, s	Speed up	Accuracy
1	x86	1	7952	1.0	19.19
2	x86	48	542	14.7	18.99
3	MIC	240	6889	1.2	20.05
4	MIC	240	589	13.5	20.05

affected by many factors associated with particular properties of the algorithm and limitations placed on the task. Algorithm iteration leaves little potential for parallelization. Optimization is only possible on each step associated with calculations on matrix. Compared to the sequential version, MIC architecture-based acceleration is 13.5 faster, which correlates with other research [**?**], [**?**]. On a side note, the coprocessor native-mode has not been considered: when the whole code is launched on the coprocessor without the main processor application. Presumably, that will allow an improved acceleration, but this question remains an open ground for further research.

The choice of data transfer model depends on the nature of the data that needs to be copied to the device. Explicit model is good enough for copying contiguous data block. The model provides many options to control data transfer which are manually selected by a programmer. However, manual copying is difficult to manage for complex structures with many fields and does not allow to send only those parts of arrays that were changed since the last transfer. In case of implicit model, it is the system that takes responsibility for managing data transfer. Changes to data residing in shared memory are synchronized automatically on entering and leaving offload region. So, this model is preferable for complex data structures and for structures with unpredictable or complex memory access pattern.

VI. CONCLUSION

The problem of research has been studied, regarding the possibility of neural networks acceleration with sequential learning algorithm. The optimization for parallel architectures has been carried out; the factors influencing the effectiveness of parallelization have been presented. The matter of MIC architecture effectiveness in this task has been addressed, as well.

ACKNOWLEDGEMENTS

The research was carried out using computational resources of Resource Centre "Computational Centre of Saint Petersburg State University" (T-EDGE96 HPC-0011828-001) within frameworks of grants of Russian Foundation for Basic Research (projects no. 16-07-01111, 16-07-00886, 16-07-01113) and Saint Petersburg State University (project no. 0.37.155.2014).

REFERENCES

[1] A. Duran and M. Klemm, "The Intel Many Integrated Core architecture," in *International Conference on High Performance Computing and Simulation (HPCS)*. IEEE, 2012, pp. 365–366.

[2] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, no. 5786, pp. 504–507, 2006.

[3] Y. Bengio, Y. LeCun *et al.*, "Scaling learning algorithms towards ai," *Large-scale kernel machines*, vol. 34, no. 5, 2007.

[4] P. Werbos, "Beyond regression: New tools for prediction and analysis in the behavioral sciences," Ph.D. dissertation, Harvard University, 1974.

[5] L. Jin, Z. Wang, R. Gu, C. Yuan, and Y. Huang, "Training large scale deep neural networks on the intel xeon phi many-core coprocessor," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*. IEEE, 2014, pp. 1622–1630.

[6] A. Viebke, "Accelerated deep learning using intel xeon phi," Ph.D. dissertation, Linnaeus University, 2015.

[7] M. Dixon, D. Klabjan, and J. H. Bang, "Implementing deep neural networks for financial market prediction on the intel xeon phi," in *Proceedings of the 8th Workshop on High Performance Computational Finance*. ACM, 2015, p. 6.

[8] M. Stanic, O. Palomar, I. Ratkovic, M. Duric, O. Unsal, A. Cristal, and M. Valero, "Evaluation of vectorization potential of graph500 on intel's xeon phi," in *High Performance Computing & Simulation (HPCS), 2014 International Conference on*. IEEE, 2014, pp. 47–54.

[9] C. Calvin, F. Ye, and S. Petiton, "The exploration of pervasive and fine-grained parallel model applied on intel xeon phi coprocessor," in *P2P, Parallel, Grid, Cloud and Internet Computing (3PGCIC), 2013 Eighth International Conference on*. IEEE, 2013, pp. 166–173.