Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 349 – 358

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

Multicore and GPU Parallelization of Neural Networks for Face Recognition

Altaf Ahmad Huqqani[a,*], Erich Schikuta[a], Sicen Ye[a], Peng Chen[a]

[a] *University of Vienna, Faculty of Computer Science, Währinger Strae 29, A-1090 Vienna, Austria*

Abstract

Training of Artificial Neural Networks for large data sets is a time consuming task. Various approaches have been proposed to reduce the efforts, many of them by applying parallelization techniques. In this paper we develop and analyze two novel parallel training approaches for Backpropagation neural networks for face recognition. We focus on two specific parallelization environments, using on the one hand OpenMP on a conventional multithreaded CPU and CUDA on a GPU. Based on our findings we give guidelines for the efficient parallelization of Backpropagation neural networks on multicore and GPU architectures.

Additionally, we present a traversal method finding the best combination of learning rate and momentum term by varying the number of hidden neurons supporting the parallelization efforts.

Keywords: Parallel Simulation, multicores, GPUs, OpenMP, CUDA, Artificial Neural Network

1. Introduction

Neural networks are efficient to solve problems where mathematical modeling of the problem is difficult. They are used to overcome problems including predictions, classifications, feature extraction, image matching and noise reduction. Typically large amount of data is required to train and evaluate neural networks. As the size of the neural network increases the time required to train or evaluate increases exponentially. Many efforts are made to reduce the training time of the neural network by selecting initial values[1], controlling the learning parameters of a neural network[2] and determining weight adjustments[3]. However there exits still a greater need to exploit the capabilities of modern hardware and softwares interfaces, specifically by the exploiting the parallelization capabilities of multicore/multithreaded CPUs or of graphic processing units (GPUs), which earned a strong research focus today. A modern GPU provides hundred of streaming cores and handles thousands of threads, which makes it specifically suitable for compute intensive applications like neural network simulation.

Two software environments, CUDA and OpenMP, have established themselves as quasi-standard for GPU and multicore systemes respectively. The Compute Unified Device Architecture (CUDA) is a parallel computing platform and programming model invented by NVIDIA[4]. It provides set of extensions to standard programming languages, like C, that enables implementation of parallel algorithms. Since the launch of the CUDA architecture many applications were developed and more features were added to new GPUs. The OpenMP Application

*Corresponding author. Tel.: +43-6507505705 ; fax: +43-1-4277-9791 .
E-mail address: aahuqqani@gmail.com.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.198

Programming Interface is a powerful and compact programming model for shared memory parallelism in C/C++ and FORTRAN in multi-core servers [5]. It provides directives, library routines, and environment variables to manage a parallel program across the different processors in a server. OpenMP exits in industry since the 90's and became a de-facto standard to exploit the capabilities of available multi-core processors with virtualization and hyper-threading with shared memory.

In this paper we explore the features of CUDA on GPU and OpenMP on multicore CPUs for the parallelized simulation of a neural network based face recognition application. We analyze the application for different configurations of neural networks and give recommendations for their effective parallel simulation on both GPU and OpenMP versions.

The paper is organised as follows. Section 2 presents the state of the art and elaborates the recent research trends. Section 3 describes the details of the parallelization approach and implementation. In section 4 we discuss the results obtained and derive our recommendations. Finally, section 5 gives the conclusion and future directions of our work.

2. Related Work

With advancement in the technology we have multicore processors and powerful graphic processing units (GPU) in our desktop and servers, affordable to everyone. Applying parallelization techniques for neural network simulation became a promising research field by using modern hardware. In [6] the author gives a survey of the state-of-the-art parallel computer hardware from a neural networks user's point of view focusing on the implementation of artificial neural networks on common and highly accessible parallel computer hardware (PCs, workstations, clusters, multicore processor machines), while keeping in mind related software matters as well.

Lindholm et al. [7] and Ryoo et al. [8] concentrate on GPU-related issues regarding general parallel programs. They focus on NVIDIA's Tesla unified graphics and computing architecture that utilizes NVIDIA GPUs and allows to create parallel programs based on the CUDA programming API. In their work CUDA is described as a minimal extension of the C/C++ programming language, where programmers write serial programs that call kernels, which are executed in parallel across a set of threads. Moreover they discuss other approaches related to GPU like Brook, OpenCL or PGI Accelerator.

In Nickolls et al. [9] and Che et al. [10] the authors also deal with CPU-based parallel approaches such as OpenMP, Pthreads or MPI (Message Passing Interface). They elaborate the CUDA architecture in detail covering streaming processor cores interconnected to external DRAM partitions. In contrast to our approach most of the work treats GPU- and CPU-based parallelism on a very high and generic level only, e.g. not giving a concrete and applicable evaluation scenario for a specific purpose.

Jang et al. [11] describe a MLP-based (Multi-Layer Perceptron) text detection algorithm implemented using CUDA and OpenMP. They tried to simplify the usage of GPUs for those users who have very little knowledge of GPU programming. In their approach the MLP consists of one input layer, one or more hidden layers and one output layer. They outline difficulties and obstacles in their implementation, present some sample code and also depict a short evaluation section. In contrast, our work focuses on the generalized Backpropagation paradigm and concentrates more specficially on the evaluation details. Nevertheless, the authors in [11] also experienced improvements regarding the computation times of their implementation by making use of parallelization techniques.

Xavier et al. [12] describe the parallel training of Backpropagation of neural network using CUBLAS, a CUDA implementation of BLAS (Basic Linear Algebra Subprogram). They compared the CPU and CUDA implementations for varying hidden neuron and reported the performance of GPU against serial execution on a classical CPU. Ting He et al. [13] used GPU standard capabilities to train MLPs by using CUDA functionalities and gained an acceleration factor of 5.21 as compared to CPU by doing most of the matrix multiplications and vector operations on GPU. In Shetal et al. [14] the implementation of an algorithm for pattern recognition is presented, which is used to recognize hand written digits. They compare their GPU implementation with a Matlab (CPU) version and report on speedups on the GPU.

In literature many approaches can be found for the parallelization of neural networks that have been deployed on different hardware architectures. For example, Pethick et al. [15] reported several different approaches for the parallelization of the neural network training. A good survey can be found in [16], which describes and

categorises the various levels of parallelization of neural network. Specifically, this classification was further refined by [17], where the authors divided Data Parallel (DP) category to Structural DP and Topological DP. This paper complements our previous work where we developed different parallelization approaches of neural networks simulations for different hardware infrastructures, as hypercube[18], cluster[19, 20], and multiprocessor systems[21].

In our work we run our experiments with a varying number of hidden neurons and processing cores, for both setups (OpenMP and CUDA). We are close to [22] where the author implemented a block mode Backpropagation for training neural networks for speech recognition on multicore and GPU systems and stated a speedup of 10 for CUDA and 4 for the multithreaded version as compared to serial execution. In contrast, we follow the approach finding the maximum number of threads for multithreaded OpenMP version depending on the input size and then compare the results with both GPU and CPU version.

Summing up, in our work we analyze on the one hand the multicore versus GPU performance gains by parallelization, but on the other hand we also analyze the application behaviour of neural network face recognition by a sweep of problem specific parameters.

3. Methodology

The use case for our parallelization approaches is face recognition by Backpropagation neural networks.

The face images we used are from [23] and are available in a pgm-p2 format. These images contain faces of 20 people, in various head poses (left, right, straight, up), various expressions (neutral, happy, sad, angry) and different eye status (open, closed), as shown in Figure 1 and Figure 2. The images are available in different resolutions: $32x30$, $64x60$ and $128x120$ pixels. In our evaluation we used images in the resolutions $32x30$ and $128x120$. For each resolution 70 images were used as the trainset and 32 images were used as the testset – for the evaluation phase of a neural network – to verify the quality of the training.

Fig. 1. (a) left-angry-closed

(b) right-happy-open.

Fig. 2. (a) straight-sad-open

(b) up-neutral-open.

3.1. Mathematical Model

The underlying network paradigm of the parallel Backpropagation simulation is the Multi-Layer Perceptron. It is a multi-level neural network with a varying number of hidden layers. Each neuron of a layer is connected with each neuron of an adjacent layer. The Backpropagation paradigm is a gradient descent method to find combinations of connection weights between the layers to map input values to given output values (supervised learning). The feedforward evaluation of the net is defined by the propagation function $net_j = \sum o_i(t) \times w_{i,j}$ where $w_{i,j}$ represent the connection weights and o_i the output value of the neuron. The correlation between the input, hidden and output layer of the net is defined by the following system,

$$hidden = f(w1 \times inp) \text{ and}$$
$$out = f(w2 \times hidden),$$

where *inp*, *hidden* and *out* denotes the input pattern, the hidden layer and the output vector and *w1* the weight matrix between the input and hidden, and *w2* the weight matrix between hidden and output layer respectively. The activation function is represented by the sigmoid function $f(x) = \frac{1}{1+e^{-c \cdot x}}$.

We used the conventional formulas for the backpropagation learning algorithm, which are given by,

$$\Delta w^1_{ij} = \alpha \times \varepsilon_i \times a_i \times (1 - a_i) \times h_i \text{ and}$$
$$\Delta w^2_{ij} = \alpha \times \sum e_m \times a_m \times (1 - a_m) \times w^1_{mi} \times e_j,$$

where $w^1_{i,j}$ representes the weight values of matrix *w1* and $w^2_{i,j}$ of *w2* respectively.

3.2. Parallelization Approaches

As discussed in section 2 we apply two parallelization approaches motivated by our problem domain and the available hardware resources:

Structural Data Parallel. The structural data parallelization techniques is used for the multicore system. The training images are divided into disjoint sets. Identical copy of neural networks are created for each thread. Each thread is trained on its own data. After certain number of epochs the weights of all the threads are collected, updated and broadcasted to the threads again. This process is continued unless the error is less than the defined threshold value.

Topological Data Parallel. The topological node parallelization is deployed on GPU using CUDA. In this approach only one copy of the neural network is instantiated, which resides on GPU. Each thread on the GPU behaves like a neuron and executes independently. To speed up the implementation the training weights and input data are stored in an one dimensional array aligned with the host and the device memory for looping in GPU.

In both implementations the Backpropagation neural network (BPNN) is trained by a supervised learning mode. Output values range from 0.0 to 1.0, where a high value (above 0.9) indicate that the image matches an assumed person. A low value (below 0.1) indicates that the image does not belong to the assigned person. After a feed-forward operation, the output value is compared with the target value and classified to rather high (above 0.5) or rather low (below 0.5), to check whether the BPNN has correctly classified the face image. Both multithreaded and GPU implementations make use of the existing timer function `clock_gettime()` to record the system time and calculate the overall execution time of a simulation run of the algorithm.

3.3. Structural Data Parallel Multicore Implementation using OpenMP

The multithreaded implementation of the BPNN for a multicore CPU consists of three layers: *n* neurons in the input layer, *h* neurons in the hidden layer and a single neuron in the output layer. All layers are fully connected, as depicted in Fig. 3. Additionally, the input and hidden layer also contain a bias neuron, which value is fixed to 1.0.

As described above, the face images we used are in pgm-p2 format. Every pixel of an image has a gray value between 0 to 255. Thus a pixel value is divided by 255 and taken as input value of the BPNN. The image resolution determines the number of input neurons in the first layer of the BPNN, i.e. for $32x30$px images 960 input neurons

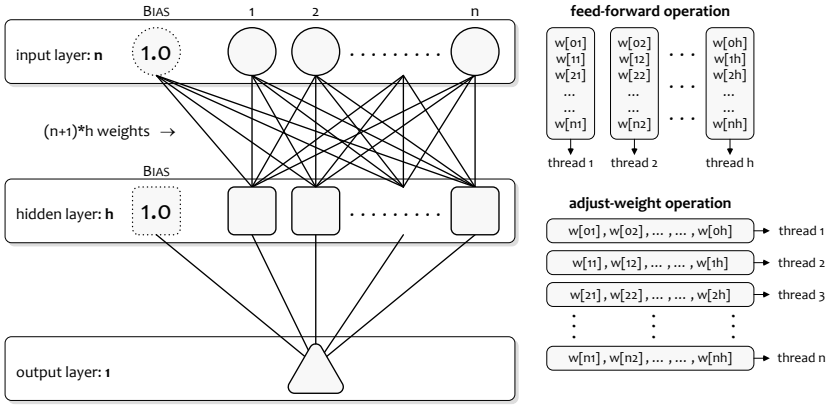

Fig. 3. Parallel BPNN Structure and Thread Allocation for Feed-Forward and Adjust-Weight Operations.

and for $128x120$px images $15,360$ input neurons are needed. Nevertheless, the number of hidden layer neurons can vary, while there is always exactly one output neuron in the last layer of the BPNN.

Considering the CPU multithreaded version we first have to define empirically the optimal number of threads per problem size. Therefore we run our program with 2, 4, 8, 16 or 32 threads (10 epochs, 512 hidden neurons) to determine the optimal number of threads. We found that for 960 inputs ($32x30$) 4 threads are the best choice and for $15,360$ inputs ($128x120$) 8 threads are optimal.

The following steps are carried out for the parallel execution of neural network in OpenMP:

Initialization.

- Creation of a disjoint training data set.
- Setting of execution threads and respective neural network instantiations.
- Transfer of input data into memory for initialization of neural network.
- Weight matrix initialization with random numbers between -1.0 to 1.0.

Computation.

- Execution of computation for feed forward, weight adjustment, record output errors and others operations.
- Transfer back of weight matrix after certain epochs.
- Weight adjustment and broadcast to execution threads.

Finish & Persist.

- Saving of errors and trained neural network on the hard disk.

3.4. Topological Data Parallel GPU Implementation using CUDA

The parallel execution of the neural network in CUDA consists basically of two phases, the forward and the weight adaptation phase.

Feed-Forward Phase. The CUDA program aims for a fully parallelization of the feed-forward operation using a huge number of threads to avoid any looping. This means every GPU-thread is therefore responsible for the computation of only a single weight value. Thus $(15,360 + 1) * 1,024$ threads were launched for a BPNN with $15,360$ inputs and $1,024$ hidden neurons in the feed-forward phase from the input to the hidden layer. Hereby, each thread has read access to a single input neuron, read access to a single weight value and write access to a hidden neuron, resulting in $(n + 1) * h$ write operations in the hidden layer. In fact this setup causes a tremendous bottleneck at the hidden layer and therefore increases the execution time.

To overcome this burden we decided to use a smaller number of threads was used, where each thread is responsible for all weights related to a specific hidden neuron. So the number of threads was reduced to $1,024$ threads for the above mentioned use-case. Hereby, each thread has read access to all input layer neurons, read access to a column of weights concerning a specific hidden neuron (edges from each input neuron to this particular hidden neuron) and write access to the same single hidden neuron. Therefore the write accesses to the hidden layer are reduced to h, each thread having only to perform a single write operation.

Adjust-Weight Phase. In the CUDA implementation two versions of the adjust-weight operation were implemented and analyzed: a *row-oriented* version, where all weights of an input node (layer 1) are handled by a thread, and a *column-oriented* version, where all weights of a hidden node (layer 2) are handled by a thread. The row-oriented version results in a lower execution time than the column-oriented version due to the given physical layout of the weight matrix and its values that are allocated row-wise in memory. The row-oriented version reaches a better spatial locality concerning the write operations and was therefore chosen for the experiments.

4. Performance Observations and Discussion

4.1. Computation Environment

For the two multithreaded versions of the BPNN face recognition algorithm we used the following hard and software environment: The multithreaded CPU program was compiled by GCC 4.3.3 and runs on a dual Xeon X5570 machine (2x 2.93GHz quad-cores with hyper-threading, each 6GB memory at 1333MHz). Thus totally 16 logical cores can be used. The multithreaded GPU program was compiled by CUDA NVCC 3.0 and runs on a Tesla C1060 graphics card (240x 1.296GHz streaming cores, 4GB memory at 800MHz).

4.2. Performance Analysis

Both implementations produced similar results regarding the error rate in the training phase as shown in Figure 4 and evaluation phase as shown in Figure 5. For our simulation runs the CPU and GPU programs used the same BPNN configuration (learning rate, momentum, number of neurons, initialized weights and number of epochs). Therefore we can directly compare the execution times of the different runs. For all runs we set the learning rate (0.3) and the momentum (0.3) and vary only the number of epochs (100 epochs for 960 inputs, 20 epochs for $15,360$ inputs) and the number of hidden neurons (from 8 to $1,024$)

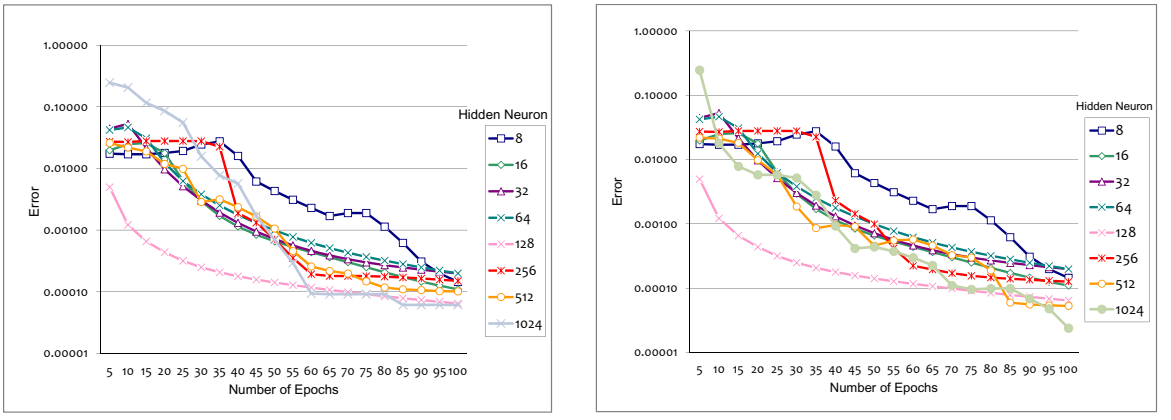

Fig. 4. Backpropagation neural network: Error Rates for varying Numbers of threads for training (a) Training multithreaded (CPU) (b) Training on GPU.

For 960 inputs the execution time of a single-threaded CPU program increases dramatically as the number of hidden neurons increases. In this case the multithreaded CPU program outperforms the GPU program, having a better runtime for each variation of hidden neurons as shown in Figure 6(a). In contrast, Figure 6(b) shows with

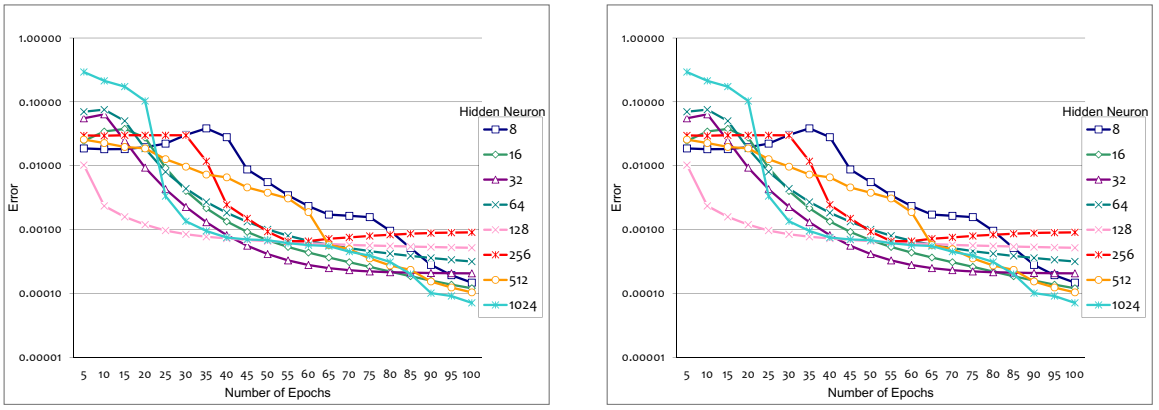

Fig. 5. Backpropagation neural network: Error Rates for varying Numbers of threads for evaluation (a) Evaluation multithreaded (CPU) (b) Evaluation GPU.

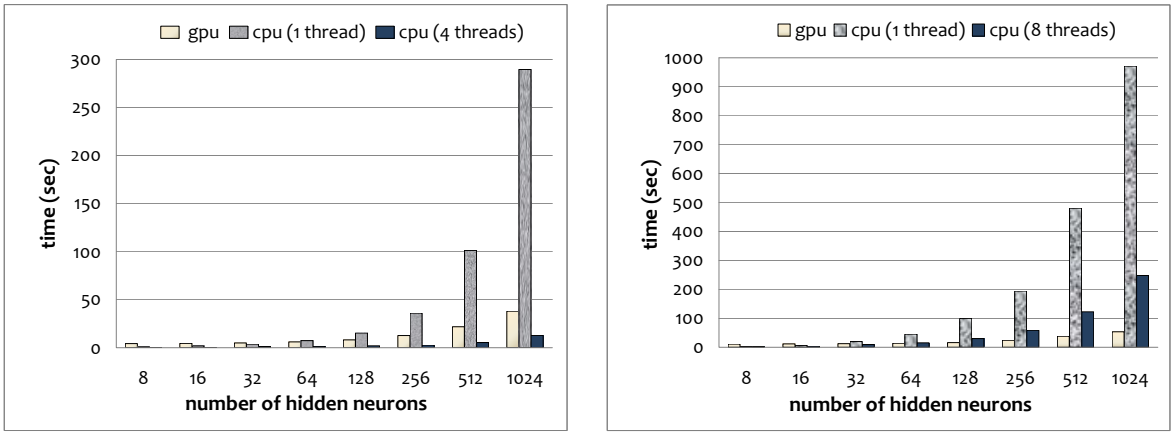

Fig. 6. BPNN Execution Time Analysis: (a) Execution time (32x30) - 100 Epochs (b) Execution Time (128x120) - 20 Epochs .

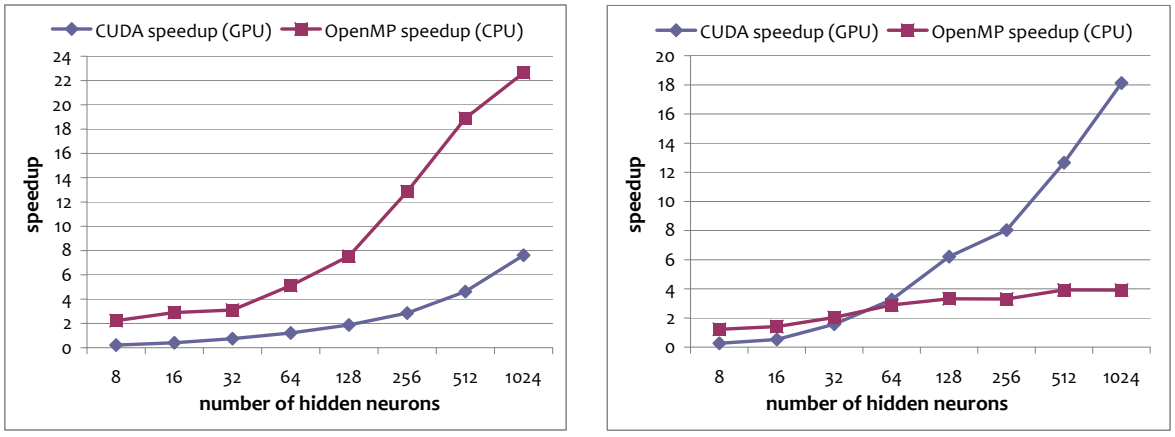

Fig. 7. BPNN Speedup Analysis:(a) Speed up (32x30). (b) Speed up (128x120).

15, 360 inputs that the GPU program performs better than the CPU program due to its superior way of utilizing the streaming cores.

With more than 64 hidden neurons the GPU program begins to outperform the CPU OpenMP program considering the speedup. As compared to serial execution on CPU ($speedup = 1$) for a BPNN with 1, 024 hidden neurons the GPU program ($speedup = 22$) outperforms the multithreaded CPU program (4 threads, $speedup = 7$) as shown in Figure 7(a). In contrast having 15, 360 inputs the GPU program reveals its great advantage when the size of the input neurons of the BPNN increases. Then a *speedup* of 18 (GPU program) in comparison to 4 (CPU program, 8 threads) is achieved, while the speedup of the CPU program stagnates (see Figure7(b)).

4.3. Guidelines for Environment Selection

In a feed forward neural network the complexity of neural network increases as the number of neurons in hidden layer increases for a fixed neurons in input layer. We have defined the problem size as the product of neurons in input layer to hidden layer. We have described [24] rules for the parallelization for neural networks on clusters, however in this paper we presented to choose a particular hardware and software configuration by taking in considerations of problem size as shown in Figure 8. The figure represents the absolute execution time of the neural network training as we changed number of neurons in input and hidden layers for openMP (optimal number of threads) and GPU implementations.

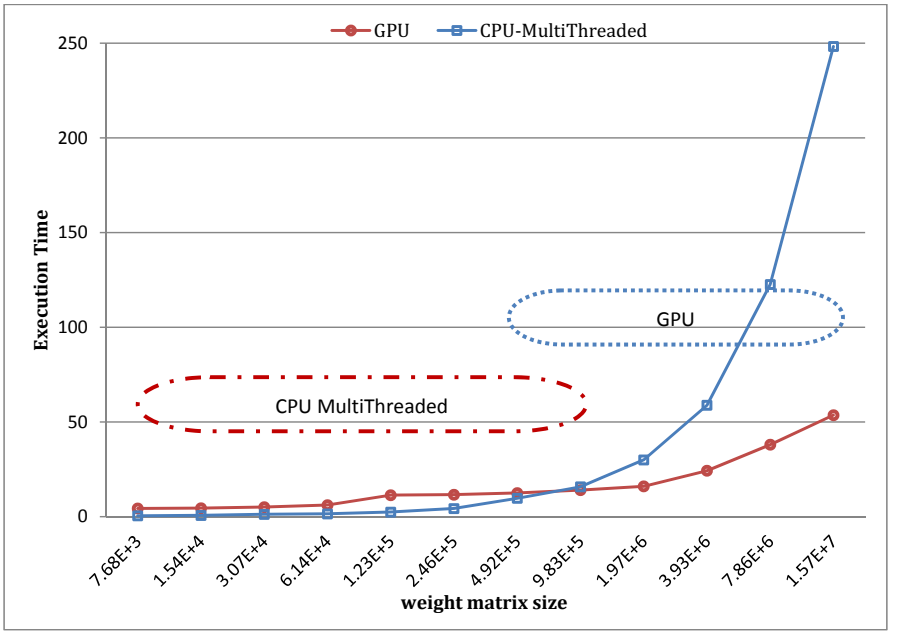

Fig. 8. Selection of openMP or GPU

Finally we used a traversal method to find the best combination of learning rate (LR) and momentum (M) for our BPNN considering a varying number of hidden neurons and an input size of 960. Therefore we did a parameter sweep by setting LR and M with values from 0.1 to 0.9 (in 0.1 steps) and got about 81 possibilities to run our algorithm. This allowed us to seek for the best combination, where the output error converges below 0.0001 and the number of epochs is minimal. As outlined in Tab. 1 the best setup is using 128 hidden neurons, a learning rate of 0.4 and a momentum of 0.1.

5. Conclusion & Future Work

In this paper we compared GPU performance to multithreaded CPU performance by implementing a neural network face recognition scenario. From the results we derived the following findings: for simulation (train-

Table 1. Traversal Method: Varying Learning Rate and Momentum for 960 Inputs (32*x*30)

number of hidden neurons	*LR*	*M*	**epochs (error <= 0.0001)**
8	0.5	0.4	62
16	0.4	0.6	55
32	0.3	0.6	69
64	0.5	0.3	63
128	**0.4**	**0.1**	**33**
256	0.4	0.1	99
512	0.1	0.5	75

ing and evaluation) of BPNN algorithm GPU based parallelization should be preferred generally to CPU based multithreaded program. However, for BPNN simulations with small input and few hidden neurons CPU based execution is better. But, if the input size and thus the number of input neurons increases GPU-based parallelization is suitable to reduce training and evaluation. In [25, 26] authors have implemented Convolutional Neural Networks (CNNs) for face detection and document processing on GPU and achieved performance gain of $11 - 13$ and $3.1 - 4.1$ respectively as compared to ours which is 22. Ho et al.[27] presented a GPU-based Cellular Neural Network simulator which can run $8 - 17$ times faster than CPU-based simulator.

As we have learned from this application that a compute intensive application can be implemented using different environment, like serial, CUDA, OpenMP, with varying performance depending on the input data. We envision to exploit the capabilities of GPU and multicores in cloud environments and offering them as services, where users can query and select them, depending on respective service level agreements (SLA) and the system is providing high performance by automatic parallelization. First prototypes of this system vision exist already for Grids[28] and Clouds[29].

References

[1] D. Nguyen, B. Widrow, Improving the learning speed of 2-layer neural networks by choosing initial values of the adaptive weights, in: Proceedings of the international joint conference on neural networks, Vol. 3, Washington, 1990, pp. 21–26.

[2] H. Kanan, M. Khanian, Reduction of Neural Network Training Time Using an Adaptive Fuzzy Approach in Real Time Applications, International Journal of Information and Electronics Engineering 2 (3).

[3] P. Wu, S.-C. Fang, H. Nuttle, Curved search algorithm for neural network learning, in: Neural Networks, 1999. IJCNN '99. International Joint Conference on, Vol. 3, 1999, pp. 1733 –1736 vol.3. doi:10.1109/IJCNN.1999.832638.

[4] CUDA Specifications and Documentation.
URL `http://docs.nvidia.com/cuda/index.html`

[5] OpenMP Specifications.
URL `http://www.openmp.org/mp-documents/OpenMP3.1.pdf`

[6] U. Seiffert, Artificial neural networks on massively parallel computer hardware, Neurocomputing 57 (2004) 135–150, new Aspects in Neurocomputing: 10th European Symposium on Artificial Neural Networks 2002.

[7] E. Lindholm, J. Nickolls, S. Oberman, J. Montrym, NVIDIA Tesla: A Unified Graphics and Computing Architecture, IEEE Micro 28 (2008) 39–55.

[8] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk, W.-m. W. Hwu, Optimization principles and application performance evaluation of a multithreaded GPU using CUDA, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP '08, ACM, New York, NY, USA, 2008, pp. 73–82.

[9] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable Parallel Programming with CUDA, Queue - GPU Computing 6 (2008) 40–53.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, K. Skadron, A performance study of general-purpose applications on graphics processors using CUDA, Journal of Parallel and Distributed Computing 68 (10) (2008) 1370–1380, general-Purpose Processing using Graphics Processing Units. doi:DOI: 10.1016/j.jpdc.2008.05.014.

[11] H. Jang, A. Park, K. Jung, Neural Network Implementation Using CUDA and OpenMP, Digital Image Computing: Techniques and Applications 0 (2008) 155–161.

[12] Sierra-Canto, Xavier, Madera-Ramirez, Francisco, V. Uc-Cetina, Parallel training of a back-propagation neural network using cuda, in: Proceedings of the 2010 Ninth International Conference on Machine Learning and Applications, ICMLA '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 307–312. doi:10.1109/ICMLA.2010.52.
URL `http://dx.doi.org/10.1109/ICMLA.2010.52`

[13] T. He, Z. Dong, K. Meng, H. Wang, Y. Oh, Accelerating multi-layer perceptron based short term demand forecasting using graphics processing units, in: Transmission & Distribution Conference & Exposition: Asia and Pacific, 2009, IEEE, 2009, pp. 1–4.

[14] S. Lahabar, P. Agrawal, P. J. Narayanan, High performance pattern recognition on gpu, in: National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG'08), 2008, pp. 154–159.
URL http://cvit.iiit.ac.in/papers/Sheetal08High.pdf
[15] M. Pethick, M. Liddle, P. Werstein, Z. Huang, Parallelization of a backpropagation neural network on a cluster computer, in: International conference on parallel and distributed computing and systems (PDCS 2003), 2003.
[16] N. Serbedzija, Simulating artificial neural networks on parallel architectures, Computer 29 (3) (1996) 56 –63. doi:10.1109/2.485893.
[17] E. Schikuta, Structural data parallel neural network simulation, Proceedings of 11th Annual International Symposium on High Performance Computing Systems (HPCS97), Winnipeg, Canada.
[18] E. Schikuta, C. Weidmann, Data parallel simulation of self-organizing maps on hypercube architectures, Proceedings of WSOM 97 (1997) 4–6.
[19] H. Schabauer, E. Schikuta, T. Weishaupl, Solving very large traveling salesman problems by som parallelization on cluster architectures, in: Parallel and Distributed Computing, Applications and Technologies, 2005. PDCAT 2005. Sixth International Conference on, IEEE, 2005, pp. 954–958.
[20] T. Weishaupl, E. Schikuta, Parallelization of cellular neural networks for image processing on cluster architectures, in: Parallel Processing Workshops, 2003. Proceedings. 2003 International Conference on, IEEE, 2003, pp. 191–196.
[21] E. Schikuta, H. Wanek, T. Fuerle, Structural data parallel simulation of neural networks, Journal of Systems Research and Information Science 9 (1) (2000) 149–172.
[22] K. Vesel, L. Burget, F. Grzl, Parallel training of neural networks for speech recognition, in: Prof. Text, Speech and Dialogue 2010, no. 9 in LNAI 6231, Springer Verlag, 2010, pp. 439–446.
[23] Shufelt, Jeff, A Neural Network Face Recognition Assignment (1994).
URL http://www.cs.cmu.edu/afs/cs.cmu.edu/user/avrim/www/ML94/face_homework.html
[24] T. Weishupl, E. Schikuta, Cellular Neural Network Parallelization Rules, in: CNNA '04: Proceedings of the 8th IEEE International Biannual Workshop on Cellular Neural Networks and their Applications, IEEE Computer Society, Los Alamitos, CA, USA, 2004.
[25] F. Nasse, C. Thurau, G. Fink, Face detection using gpu-based convolutional neural networks, in: Computer Analysis of Images and Patterns, Springer, 2009, pp. 83–90.
[26] K. Chellapilla, S. Puri, P. Simard, et al., High performance convolutional neural networks for document processing, in: Tenth International Workshop on Frontiers in Handwriting Recognition, 2006.
[27] T.-Y. Ho, P.-M. Lam, C.-S. Leung, Parallelization of cellular neural networks on gpu, Pattern Recognition 41 (8) (2008) 2684–2692.
[28] E. Schikuta, T. Weishupl, N2Grid: Neural Networks in the Grid, in: Neural Networks, 2004. Proceedings. 2004 IEEE International Joint Conference on, Vol. 2, 2004, pp. 1409–1414.
[29] A. A. Huqqani, L. Xin, P. P. Beran, E. Schikuta, N2Cloud: Cloud based Neural Network Simulation Application, in: Neural Networks (IJCNN), The 2010 International Joint Conference on, 2010, pp. 1–5.