Parallel Training of Neural Networks for Speech Recognition

Karel Veselý, Lukáš Burget and František Grézl

Speech@FIT, Brno University of Technology, Czech Republic

xvesel39@stud.fit.vutbr.cz, {burget,grezl}@fit.vutbr.cz

Abstract

In this paper we describe parallel implementation of ANN training procedure based on block mode back-propagation learning algorithm. Two different approaches to parallelization were implemented. The first is *data parallelization* using POSIX threads, it is suitable for multi-core computers. The second is *node parallelization* using high performance SIMD architecture of GPU with CUDA, suitable for CUDA enabled computers.

We compare the speed-up of both approaches by learning typically-sized network on the real-world phoneme-state classification task, showing nearly 10 times reduction when using CUDA version, while the 8-core server with multi-thread version gives only 4 times reduction. In both cases we compared to an already BLAS optimized implementation. The training tool will be released as Open-Source software under project name *TNet*.

Index Terms: Artificial Neural Network, GPU, CUDA, Phoneme Classification, Fast Training

1. Introduction

State of the art speech recognition systems are based on acoustic-phonetic models of the words. Each acoustic unit is modeled by one or more states of a Hidden Markov Model (HMM). The systems based on the hybrid paradigm are using artificial Artificial Neural Networks (ANN) to estimate the probability density of the acoustic patterns associated to such HMM states [1]. The crucial advantage of using a hybrid ANN-HMM approach is that the computation of the posterior probabilities of HMM states takes a small fraction of time compared to the classical GMM-HMM approach. Moreover, the ANN models are inherently discriminative.

The most widely used ANNs in speech recognition are feed-forward Multi Layer Perceptron (MLP) networks. The MLPs are typically used as *phoneme classifiers*, where the network input is a vector of features and the output is a vector of phoneme-class membership probabilities.

These probabilities have been proven to be very useful for further processing such as direct LVCSR decoding [5], Keyword spotting [6] or Language identification. The probabilities can be called *posterior features*, because they are used as features for subsequent systems, or *phoneme-state probabilities*, because they correspond to emission probabilities in standard GMM-HMM approach.

The MLPs are trained by standard block-mode *stochastic gradient descent* algorithm with *error backpropagation*. Feature vectors together with phoneme labels are used for the training. The stochastic variant was chosen because it converges in lower number of epochs on data-sets with redundant data (e.g. with repeating or very similar segments) compared to the batch variant. Moreover, stochastic variant gives better possibility of escaping from local minima [2].

Even when using some degree of MLP training parallelization, for example as it is implemented in our current training tool *SNet* [4], typically the training time exceeds 24 hours, due to the huge quantities of training data (hundreds hours of speech). Such long training periods are uncomfortable for practical use.

In this paper, we describe *TNet* – a new faster implementation of parallel neural network training with two acceleration possibilities. The first is *data parallelization* version, which is based on POSIX threads. It is suitable for widespread multi-core computers. The second is *node parallelization* version, which exploits the high performance SIMD architecture of modern Graphical Processing Units (GPU) with Compute Unified Device Architecture (CUDA) computing engine; similar approach was reported in [9]. The performance of both approaches is compared on a real-world task.

The paper is organized as follows. Section 2 recalls the training algorithm and its parallelization methods. Section 3 details key aspects of both acceleration techniques from the implementation point of view. The experimental results are summarized in Section 4. Concluding remarks and future perspectives are reported in Section 5.

2. Feed-forward neural networks

Feed-forward ANN is an adaptive multivariate transform function with ability to "memorize" patterns by adjusting tunable parameters (neuron weights). It can be seen as a sequence of alternating linear and nonlinear transformations. Due to Kolmorogovs' theorem [2] we believe that the network is able to express any possible function when having enough layers and neurons per layer. In our particular case, we are interested in classification MLPs, the training algorithm will be explained on this class of ANN. The *Stochastic gradient descent algorithm* with *error backpropagation* is used for the training, while the weight update is performed per bunch (a block of N frames). In the MLP, *Sigmoid* nonlinearity is used for hidden layer and *Softmax* nonlinearity is used for the output layer.

For the sake of simplicity, the training algorithm will be explained on the case when the bunch has only one input datapoint. The general formula for gradient descent is:

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \mu \nabla E \tag{1}$$

which says that the current parameters $\mathbf{w}$ are iteratively moved in the opposite direction of the error function gradient ∇E, which is scaled by learning rate μ. The gradient ∇E of the error function E is a vector of its first derivatives with respect to all the model parameters $\mathbf{w}$:

$$\nabla E = \left[\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, ..., \frac{\partial E}{\partial w_M} \right]^T \tag{2}$$

To obtain the gradient, it is necessary to perform *data propagation*, *objective function evaluation* and *error backpropagation*. The data propagation is done from the first to the last layer. The linear and non-linear layers are alternating, while the first layer is linear and last layer is non-linear:

$$\begin{aligned} \text{Linearity:} \quad & \mathbf{y}_n = \mathbf{W}_n \mathbf{x}_n + \mathbf{b}_n \\ \text{Sigmoid:} \quad & y_{ni} = \frac{1}{1+\exp(-x_{ni})} \\ \text{Softmax:} \quad & y_{ni} = \frac{\exp(x_{ni})}{\sum_j \exp(x_{nj})} \end{aligned} \tag{3}$$

where n is the index of linear or nonlinear transformation, $\mathbf{y}_n$ is the output vector, $\mathbf{W}_n$ is a neuron-layer weight matrix, $\mathbf{x}_n$ is the input vector and $\mathbf{b}_n$ is the neuron-layer bias vector. Obviously, the output of the previous transformation is input of the next transformation, the input of the first transformation being the input data.

Then the *cross-entropy* error function is evaluated by using the MLPs' output vector $\mathbf{y}_n$ and *desired vector* $\mathbf{d}$. To be able to do the backpropagation, the first derivative of the error function E with respect to MLPs' output vector $\mathbf{y}_n$ must be evaluated. In the particular case when we have a coupled *cross-entropy* with *softmax*, the derivative of error function E with respect to softmax input vector $\mathbf{x}_{\text{Softmax}}$ leads to trivial solution which is called *global error*:

$$\begin{array}{ll} \text{Cross-entropy} & \text{Global error} \\ E = -\sum_j d_j \ln(y_j) & \frac{\partial E}{\partial \mathbf{x}_{\text{Softmax}}} = \mathbf{e}_n = \mathbf{y}_n - \mathbf{d} \end{array} \tag{4}$$

Now, the error backpropagation can be performed. We start at the last linearity which precedes *Softmax*, proceeding through the layers towards the first layer:

$$\begin{array}{ll} \text{Linearity} & \text{Sigmoid} \\ \mathbf{e}_{n-1} = \mathbf{W}_n^T \mathbf{e}_n & \mathbf{e}_{n-1} = \mathbf{y}_n(\mathbf{y}_n - \mathbf{1})\mathbf{e}_n \end{array} \tag{5}$$

Finally the *gradient descent* update formulas are used:

$$\begin{aligned} \text{Weights update:} \quad & \mathbf{W}_n(t+1) = \mathbf{W}_n(t) - \mu\, \mathbf{e}_n \mathbf{x}_n^T \\ \text{Bias update:} \quad & \mathbf{b}_n(t+1) = \mathbf{b}_n(t) - \mu\, \mathbf{e}_n \end{aligned} \tag{6}$$

During one training epoch (pass over whole training data-set), this procedure is performed for each block of the input data.

2.1. Parallelizations

The *Stochastic on-line learning* imposes strong data dependencies which makes the parallelization difficult. Two effective approaches to parallel network training have been reported [3]:

Data parallelization The training data is divided into disjoint sets. Each thread has its own network instance and works on its own data-set. Weight synchronization occurs periodically when N frames are processed. The weight difference matrices (e.g. error gradients) are gathered, summed up and a new set of weights is generated and distributed.

Node parallelization In this case, there is only one instance of the network. The network layers are divided into disjoint sets of neurons. Each thread has associated its own set. This method imposes higher frequency of synchronization than *data parallelization*. The threads are synchronized by a *barrier* before one can proceed to the next layer.

Both approaches were tested and compared. First, the multi-thread *data parallelization* was implemented, then the *node parallelization* using CUDA was implemented.

The problem of *data parallelization* is that the algorithm doesn't scale ideally and the overhead of weight synchronization increases by adding more slave threads.

The problem of CUDA *node parallelization* is that the data transfers between the host memory and the GPU memory can easily become a bottleneck and therefore should be minimized. However it is more efficient to transfer data in larger segments.

3. Implementation

The design of the tool was chosen with respect to both high performance and simple extensibility. The tool is capable of both *on-line* (Stochastic) and *batch* gradient descent, the only difference is that the weight update is not performed per-cycle but at the end of the epoch.

For the sake of speed, the data matrices are stored in such way that each matrix row starts on 16-byte aligned address both in the host memory as well as in the GPU memory.

The *TNet* is compatible with the HTK data formats. It accepts STK[1] transforms for feature extraction, the network is stored in it's own format with possible conversion to the STK format.

After finishing the *TNet* development, the tool will be distributed as Open Source software. The pre-release version can be downloaded at BUT Speech@FIT web-site[2].

Currently, both the multi-threaded *data parallelization* and CUDA *node parallelization* are implemented in *TNet*, while one option excludes the other:

3.1. Multi-threaded version

Here, the GotoBLAS[3] library is used to accelerate linear algebra operations. The `cblas_sgemm` function is used for linearity propagation, backpropagation and for the evaluation of weight difference matrices. This function represents 80% of training time in case of single thread training.

The network parameters are shared for all the threads which improves the processor cache hit-rate.

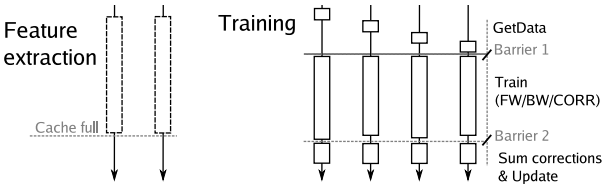

Figure 1: Thread synchronization (FW - forward pass, BW - backward pass, CORR - weight difference matrices calculation)

As can be seen in Fig. 1, the training process is divided into several training threads and two feature extraction threads. The training cycle consists of three phases:

1. Data distribution
2. Training (forward pass, objective function evaluation, error backpropagation and evaluation of weight difference matrices)

[1] BUT Speech ToolKit http://merlin.fit.vutbr.cz/svn/STK/trunk/src/
[2] TNet http://speech.fit.vutbr.cz/files/software/tnet/TNet.tar.gz
[3] GotoBLAS http://www.tacc.utexas.edu/tacc-projects/

3. Merging of weight difference matrices, weight update

The *data distribution* is done in series, because it is a simpler and more universal solution. If we did not distribute the data deterministically, we would not be able to repeat the training procedure and obtain the same network, also the merging of weight update matrices must be deterministic. The *training* and *weight difference matrix merging* is parallel. Two barriers are used to synchronize, one before the training starts and one before the merging starts. Partial summing is used for merging; every thread is responsible for summing several lines of the weight difference matrices from all the network instances.

The two background threads are loading the training data and performing STK feature extraction transform till the cache is full. Two caches are used, one is used for training, while the second is being filled and randomized on background.

3.2. CUDA version

Here, the CUBLAS library is used to accelerate the linear algebra operations, the nonlinear transformations are implemented as separate CUDA kernels. Two threads are used, the background data-preparation thread and the main thread which calls all the CUDA routines.

The feature extraction is partly done on the CPU in the background thread, and partly on the GPU in the main thread, also the feature cache randomization is done in the main thread.

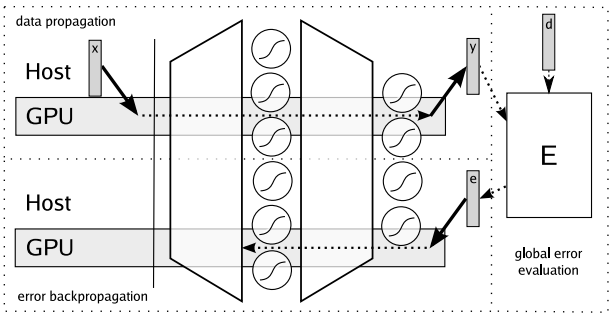

Figure 2: Host to GPU I/O operations

As can be seen in Fig. 2, a block of randomized input data is uploaded to the GPU by the function `cudaMemcpy` and propagated through the network. Then, the network output is downloaded from the GPU again by `cudaMemcpy` and the error is evaluated on host CPU. Then the error is uploaded to the GPU, backpropagated and finally, the network parameters are updated.

Currently, the CUDA acceleration is feasible due to its C-like language interface and availability of CUBLAS and CULA libraries, alternatives to popular *cblas* and *clapack*, which are very useful for linear algebra acceleration.

The function `cublasSgemm` is used to accelerate the matrix multiplication, which is used for linearity propagation, back-propagation and neuron-layer weight update. The nonlinear transformations (sigmoid, softmax) and neuron-layer bias update have been implemented as CUDA kernels. The kernels are run as a grid of 16x16-cells blocks or as a grid of 16-rows blocks. The mode depends on the type of kernel function. From the integration point of view, the CUDA kernels are wrapped in C functions and compiled by `nvcc` as separate C library, which is then linked to the project. In order to become familiar with CUDA we suggest to read [10].

4. Experimental results

Database The training set is a subset of AMIDA meeting data corpus[4]. The total size of the AMIDA corpus is 150 hours. A 135h subset was taken as training data-set, the cross-validation was performed on a 15h subset. The corpus is labeled by 45 phonemes. The phonemes are considered context independent with three sub-states, which leads to 135 classes.

Parameterisation We use the long-context parameterisation which was proposed in [7]. The parameters are log Mel filterbank outputs derived using 25ms window, 10ms shift; 23 filters were used. Such parameters were normalized by Per-Speaker Cepstral Mean-Variance Normalization and Vocal Tract Length Normalization. Then a 51 frames long context (510ms) was taken for each filter. Each context was separately scaled by Hamming Window and compressed by Discrete Cosine Transform to 26 coefficients. By re-concatenation we get vectors of 23×26 coefficients. Such network inputs were finally globally Mean-Variance normalized.

Network structure Two-layer feed-forward MLP with one hidden layer was used. The layers are fully connected, the dimensionality of input is 598, the hidden layer has 1000 neurons, the output layer has 135 neurons. The activation function of the first layer was Sigmoid, while activation function of the second layer was Softmax.

The network has 0.7M tunable parameters and it is a part of a state of the art speech recognition system [8].

Training Standard backpropagation algorithm with the "newbob" learning rate scheduling was used: The learning rate is kept fixed as long as the increment in cross-validation accuracy is bigger than a threshold. For the subsequent epochs, the learning rate is being halved till the cross-validation increment is inferior to some stopping threshold.

The ANN weight update was performed per *bunch* (fixed-size block of data-points). In case of multi-thread training, the slave bunch-size is different for each order of parallelization. Since the weight update is done per aggregated bunch-size of all slaves, which is equal to original bunch-size of serial training, the two training algorithms are equal.

We believe that the optimal aggregated bunch-size is around 1000 frames, but it should be verified for each different training task. Too small bunch-size causes training slowdown, too big bunch-size may have impact on the quality of the global minima which will be found.

Results The performance of the *CUDA version* was evaluated on HW setup marked ♡: Desktop PC with 1x Intel Core2Duo E8400 3.0GHz, 2GB RAM and the NVidia GTX 285 GPU with 240 shaders at 1.476GHz. The system is running CentOS 5.4 64bit Linux.

The performance of the *multi-thread version* was evaluated on HW setup marked ♢: Blade server SuperTwin2 6026TT-TF with 2x Intel Xeon E5520 2.26GHz Nehalem, 12 GB RAM running also CentOS 5.4 64bit Linux.

For each training configuration, the single-epoch training time was measured till the network was fully trained, which took typically 8-10 epochs. The mean training-epoch durations and the obtained speed-ups are in Tab. 1.

[4] http://www.amiproject.org/ami-scientific-portal/meeting-corpus

The *baseline* for the speed-up evaluation is one-thread CUDA-disabled training on the desktop PC ♡; Tab. 1, line 1. This was already accelerated by GotoBLAS linear algebra library with disabled multi-threading, a pure C implementation would be even approximately 4x slower [9].

Our fastest *mutli-thread* training is in Tab. 1, line 2. Six training threads are used together with two background feature extraction threads; one thread was found to be insufficient. By profiling we discovered that the data distribution represents 4% of the running time, the training is 87%, the waiting on barrier before merging the weight update matrices is 5% and merging the weight update matrices is 4%. During this experiment we exploited the whole 8-core machine socketed with two state-of-the-art Nehalem processors and obtained 4.4x speed-up.

However the *CUDA version* of the training Tab. 1, line 3 greatly outperformed even the 8-core machine. One main thread is used for all the CUDA calls and one background thread for loading data and doing undemanding part of preprocessing. The speed-up compared to the baseline is very good: 9.7x, the 8-core machine was outperformed 2.2x. The profiling has revealed that 37% of running time was spent on data preprocessing, 8% on CPU-GPU data transfers, 23% on forward pass, 3% on objective function evaluation, 5% on backward pass and 24% on update of network parameters.

Table 1: *Training configurations*

HW	Training Impl.	Thr-eads	T-epoch h:mm	Speed-up
♡	1-thread, no CUDA	1+1	4:41	-
♢	Multi-thread	6+2	1:04	4.4x
♡	CUDA	1+1	0:29	9.7x

Scaling on 32 cores A very interesting graph was produced by running the *multi-thread version* with different parallelizations on a 32 core server. For this experiment, we used HP ProLiant DL785 G5 server, which is fully socketed with 8 quad-core AMD Opteron 8356 processors and 128GB of RAM, while running CentOS 5.4 64bit Linux with NUMA optimized kernel.

The server is not equipped with the last generation processors, but still we can observe here the scaling of multi-thread version. As can be seen in Fig. 3 the training does not scale ideally. Adding more cores produces speed-up until the critical 12 cores are reached, then performance degradation occurs.

The bottleneck was identified as RAM to CPU bandwidth, which is actually limited to 10GB/s. During our benchmark, we have measured the bandwidth of 90GB/s by considering all the read and write data-flows, however this number is artificially "boosted" by the performance of processor cache hierarchy.

5. Conclusion and future work

We have studied and implemented two different approaches to the parallelization of the ANN training procedure for sequential patterns. First approach is the *data-parallelization* optimized for multiprocessor servers, showing a 4 times reduction of the training time on an 8-core server. The second approach is the *node-parallelization* optimized for a regular PC equipped with a modern GPU card. This approach showed nearly 10 times reduction of the training time. In both cases, we compared to the BLAS optimized single-thread training without GPU acceleration.

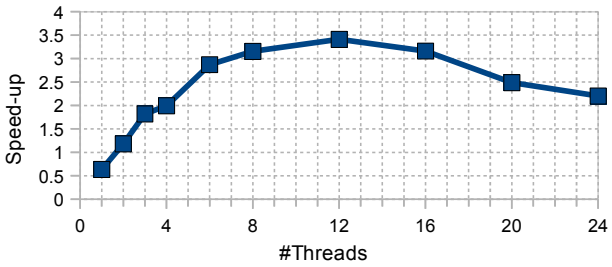

Figure 3: *The scaling of multi-thread version on 32-core server. The y-axis baseline is in Tab. 1, line 1.*

The acceleration of the ANN training will not only facilitate the generation of the acoustic models, but it will also lead to even more intensive research activity on finding better network structures and topologies for various speech tasks. Reduced training time also allows training of larger models with huge training corpora. Further work can be focused on scaling by using more graphic cards in one PC. As reported in [9], it is feasible to host multiple instances of the CUDA training in one system, but it might be also possible to accelerate one instance of the training by multiple cards. For example, when needed, the time consumed by feature preprocessing can be saved by using second GPU in order to get even better speed-up.

6. Acknowledgements

This work was partly supported by Grant Agency of Czech Republic project No. 102/08/0707, and by Czech Ministry of Education project No. MSM0021630528, and by Czech Ministry of Interior project No. VD20072010B16.

7. References

[1] Bourlard, H., Morgan, N.: Connectionist Speech Recognition a Hybrid Approach. Norwell MA USA, Kluwer Academic Publishers 1993, ISBN 0-79-239396-1

[2] Bishop, Ch.: Neural Networks for Pattern Recognition. Oxford University Press 2004, ISBN 0-19-853864-2

[3] Pethick, M., Liddle, M., Werstein, P., Huang, Z.: Parallelization of a Backpropagation Neural Network on a Cluster Computer. In Parallel and Distributed Computing and Systems, IASTED/ACTA Press, 2003.

[4] Kontár, S.: Parallel training of neural networks for speech recognition. FIT VUT Brno, 2006, diploma project

[5] Kingsbury, B.: Lattice-based Optimization of Sequence Classification Criteria for Neural-Network Acoustic Modeling. Proceedings of ICASSP'09, ISBN 978-1-4244-2353-8

[6] Szöke, I., Schwarz, P., Matějka, P., Burget, L., et al.: Comparison of Keyword Spotting Approaches for Informal Continuous Speech. Proceedings of Interspeech'05 - Eurospeech, ISSN 1018-4074

[7] Schwarz, P., Matějka, P., Černocký, J.: Towards Lower Error Rates in Phoneme Recognition. Proceedings of TSD'04, ISBN 3-540-23049-1

[8] Grézl, F., Karafiát, M., Burget, L.: Investigation into bottle-neck features for meeting speech recognition. Proceedings of Interspeech'09, ISSN 1990-9772

[9] Scanzio, S., Cumani, S., Gemello, R., Mana F., Laface, P.: Parallel Implementation of Artificial Neural Network Training. Proceedings of ICASSP'10, ISSN 0167-8655

[10] Seland, J.: CUDA Programming. SINTEF Winter school, http://heim.ifi.uio.no/ knutm/geilo2008/seland.pdf