Neural Network Training with Extended Kalman Filter Using Graphics Processing Unit

Peter Trebatický and Jiří Pospíchal⋆

Slovak University of Technology in Bratislava, Faculty of Informatics and Information Technologies, Ilkovičova 3, 842 16 Bratislava, Slovakia
{trebaticky,pospichal}@fiit.stuba.sk

Abstract. The graphics processing unit has evolved through the years into the powerful resource for general purpose computing. We present in this article the implementation of Extended Kalman filter used for recurrent neural networks training, which most computational intensive tasks are performed on the GPU. This approach achieves significant speedup of neural network training process for larger networks.

1 Introduction

The graphics processing unit (GPU) was and still is used mainly for speedup of graphical operations. It has recently evolved into the powerful resource for general purpose computing.

Recurrent neural network learning is a very difficult numerical problem, which approaches very poorly and slowly to satisfactory results when being solved with the classic gradient optimization methods on longer input sequences. In this paper we present the already studied [9,10] better alternative which is called Extended Kalman filter (EKF) and how to make it faster using modern but generally available graphics processing unit.

In the first section of this paper we explain the concept of Kalman filtering and how to apply it to the task of neural network training. The remaining sections describe in detail how to map the equations involved onto the graphics processing unit. On the chosen problem we demonstrate two things. Firstly, the already known fact that the EKF achieves better results than classical gradient descent methods and secondly, more importantly, the speedup of our implementation on the graphics processing unit as opposed to implementation on CPU.

2 Extended Kalman Filter

Kalman filter (which is the set of mathematical equations) is considered one of the important discoveries in the control theory principles. E. Kalman's paper [5] was published in the year 1960. Its most immediate applications were in control

⋆ This work was supported by Scientific Grant Agency of Slovak Republic under grants #1/0804/08 and #1/4053/07.

V. Kůrková et al. (Eds.): ICANN 2008, Part II, LNCS 5164, pp. 198–207, 2008.
© Springer-Verlag Berlin Heidelberg 2008

of complex dynamic systems, such as manufacturing processes, aircrafts, ships or spaceships (it was part of the Apollo onboard guidance system). However, the Extended Kalman filter started to appear in the neural network training applications only relatively recently, which was caused by the progress of computer systems development.

Original Kalman filter is targeted at the *linear* dynamic systems. However, when the model is *nonlinear*, which is the case of neural networks, we have to extend Kalman filter using linearization procedure. Resulting filter is then called extended Kalman filter (EKF) [3]. Neural network is a nonlinear dynamic system, that can by described by equations:

$$\mathbf{x}_k = \mathbf{x}_{k-1} + \mathbf{q}_{k-1} \tag{1}$$

$$\mathbf{y}_k = g(\mathbf{x}_k, \mathbf{u}_k, \mathbf{v}_{k-1}) + \mathbf{r}_k \tag{2}$$

The process equation expresses the state of neural network as a stationary process corrupted by the process noise $\mathbf{q}_k$, where the state of the network $\mathbf{x}$ consists of network weights. Measurement equation expresses the desired output of the network as a nonlinear function g of the input vector $\mathbf{u}_k$, of the weight vector $\mathbf{x}_k$ and for recurrent networks also of the activations of recurrent neurons from the previous step $\mathbf{v}_{k-1}$. This equation is augmented by a random measurement noise $\mathbf{r}_k$. The covariance matrix of the noise $\mathbf{r}_k$ is $\mathbf{R}_k = E\left[\mathbf{r}_k\mathbf{r}_k^T\right]$ and the covariance of the noise $\mathbf{q}_k$ is $\mathbf{Q}_k = E\left[\mathbf{q}_k\mathbf{q}_k^T\right]$.

The basic idea of the Extended Kalman filter lies in the linearization of the measurement equation at each time step around the newest state estimate $\hat{\mathbf{x}}_k$. We use for this purpose just the first-order Taylor approximation of non-linear equation, because of computational complexity.

We can express the neural network training as a problem of finding the state estimate $\mathbf{x}_k$ that minimizes the least-squares error, using all the previous measurements. We can express the solution of this problem as:

$$\mathbf{K}_k = \mathbf{P}_k\mathbf{H}_k^T[\mathbf{H}_k\mathbf{P}_k\mathbf{H}_k^T + \mathbf{R}_k]^{-1} \tag{3}$$

$$\hat{\mathbf{x}}_{k+1} = \hat{\mathbf{x}}_k + \mathbf{K}_k[\mathbf{y}_k - g(\hat{\mathbf{x}}_k, \mathbf{u}_k, \mathbf{v}_{k-1})] \tag{4}$$

$$\mathbf{P}_{k+1} = \mathbf{P}_k - \mathbf{K}_k\mathbf{H}_k\mathbf{P}_k + \mathbf{Q}_k \tag{5}$$

where $\hat{\mathbf{x}}$ is a vector of all the weights, $g(\cdot)$ is a function returning a vector of actual outputs, $\mathbf{y}$ is a vector of desired outputs, $\mathbf{K}$ is the so called Kalman gain matrix, $\mathbf{P}$ is the error covariance matrix of the state and $\mathbf{H}$ is the measurement matrix (Jacobian). Matrix $\mathbf{H}$ contains partial derivatives of ith output with respect to jth weight. One can use for this purpose one of two main methods – *Real-Time Recurrent Learning* (RTRL) or *Backpropagation Through Time* (BPTT), or its truncated version BPTT(h) [8].

The RTRL is computationally intensive, therefore we will use the BPTT(h) method according to the recommendation in [8]. With this method for appropriately chosen depth h we obtain derivatives that are close to those obtained by the RTRL, and we significantly reduce the computational complexity.

At the beginning of the recursion (3–5) it is necessary to assign the initial values of $\hat{\mathbf{x}}_0$, $\mathbf{P}_0$, $\mathbf{Q}_0$ and $\mathbf{R}_0$. The initial values of weights should be set randomly

e.g. from interval $[-0.5, 0.5]$. According to recommendations in [3,8] $\mathbf{P}_0$ should be set in orders of magnitude $100\,\mathbf{I}$–$1000\,\mathbf{I}$, where $\mathbf{I}$ is identity matrix. $\mathbf{Q}_0$ should have values from $10^{-6}\,\mathbf{I}$ to $0.1\,\mathbf{I}$. Its nonzero value helps to override divergency [3]. For the parameter $\mathbf{R}_0$, values in orders of magnitude $10\,\mathbf{I}$–$100\,\mathbf{I}$ should be chosen.

3 Computational Complexity

In order to express the computational complexity of EKF and BPTT(h), let us summarize the notation and dimensions of every matrix and vector:

n_i	Number of input neurons, i.e. number of inputs to the network
n_o	Number of output neurons, i.e. number of outputs of the network
n_h	Number of hidden neurons
n_x	Number of weights in the network
$\mathbf{x}_{[n_x \times 1]}$	Vector of weights in the network
$g(\cdot)_{[n_o \times 1]}$	Function returning vector of actual outputs
$\mathbf{y}_{[n_o \times 1]}$	Vector of desired outputs
$\mathbf{K}_{[n_x \times n_o]}$	Kalman gain matrix
$\mathbf{P}_{[n_x \times n_x]}$	Error covariance matrix
$\mathbf{Q}_{[n_x \times n_x]}$	Process noise covariance matrix
$\mathbf{R}_{[n_o \times n_o]}$	Measurement noise covariance matrix
$\mathbf{H}_{[n_o \times n_x]}$	Measurement matrix (Jacobian)

Number of weights n_x in the Elman's architecture of recurrent neural network can be expressed as $(n_i+1)n_h+n_h^2+(n_h+1)n_o$, where ones represent bias weights.

Using given notation, we can express the computational complexity of the EKF (3–5) as $O\left(\left[n_o n_x^2 + n_o^2 n_x + n_o^2 + n_o n_x + n_x^2\right] + n_o^3 + h n_o n_x\right)$. The first term $[n_o n_x^2 + n_o^2 n_x + n_o^2 + n_o n_x + n_x^2]$ comes from matrix multiplications and additions, the second term n_o^3 is the matrix inversion in (3) and the third term $h n_o n_x$ represents the complexity of BPTT(h) used to compute measurement matrix $\mathbf{H}$, where h is the truncation depth. Typically $h, n_o \ll n_x$, so the complexity of the EKF is then $O(n_o n_x^2)$. As BPTT(h) alone has complexity $O(h n_o n_x)$, the speed difference between EKF and BPTT(h) is significant for large n_x.

4 Reducing Complexity Using GPU

In this paper we aim to reduce the impact of the most computationally intensive tasks of the EKF by exploiting the parallel nature of the graphics processing unit. As the number of parallel processors is fixed – concretely 128 in G8800 GTX – this does not have the impact on asymptotical complexity of EKF. But any constant factor speedup makes the EKF more useable and for smaller numbers of n_x can actually make it faster than BPTT(h).

For implementation we chose CUBLAS library readily available for nVidia GPUs (as part of CUDA Toolkit[12]), because it provides basic linear algebra operations, especially matrix multiplication. This library is a significant step in general

purpose computing on the GPU as one does not have to overcome steep learning curve of detailed GPU functionality. As it turns out, the provided functionality of CUBLAS library is all we need to speed up matrix equations (3–5).

The only thing that is not readily available is matrix inversion needed to compute $n_o \times n_o$ matrix in (3). This is a symmetric positive definite matrix, that is why we chose to use Cholesky factorization to compute its inverse. Cholesky factorization on GPU has already been studied [4], but because it is usually small matrix ($n_o \ll n_x$) we can simply compute it on the CPU. This requires an additional transfer of data between GPU and CPU which is a costly operation, but we already have to transfer a much bigger measurement matrix and weight vector, so this should not have a big impact.

Cholesky factorization of a symmetric positive definite matrix $\mathbf{M}$ is a lower triangular matrix $\mathbf{N}$ for which holds $\mathbf{N}\mathbf{N}^T = \mathbf{M}$. Inverse of matrix $\mathbf{M}$ can then be efficiently and numerically stable computed:

$$\mathbf{M}\mathbf{M}^{-1} = \mathbf{I} \tag{6}$$

$$\mathbf{N}\mathbf{N}^T\mathbf{M}^{-1} = \mathbf{I} \tag{7}$$

$$\mathbf{N}^T\mathbf{M}^{-1} = \mathbf{X} \tag{8}$$

$$\mathbf{N}\mathbf{X} = \mathbf{I} \tag{9}$$

where $\mathbf{I}$ is an identity matrix and $\mathbf{X}$ is used for substitution. The inverse of $\mathbf{M}$ is computed by first solving (9) for unknown $\mathbf{X}$ and then by solving (8) for unknown $\mathbf{M}^{-1}$.

5 Implementation Details

This section describes in detail the implementation of EKF on GPU as well as on CPU. In order to explain the meaning of auxiliary matrices $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$ and $\mathbf{Z}$ used in pseudocode we rewrite equation (3):

$$\mathbf{K} = \mathbf{P}\mathbf{H}^T\left[\mathbf{H}\mathbf{P}\mathbf{H}^T + \mathbf{R}\right]^{-1} = \mathbf{C}\left[\mathbf{A}\mathbf{H}^T + \mathbf{R}\right]^{-1} = \mathbf{C}\mathbf{B}^{-1} = \mathbf{C}\mathbf{Z}$$

The pseudocode of initialization of the EKF on GPU follows:

On CPU Initialize matrices $\mathbf{P}$, $\mathbf{R}$, $\mathbf{Q}$, $\mathbf{I}$
Note: we store only diagonal elements of $\mathbf{R}$, $\mathbf{Q}$ and $\mathbf{I}$
On CPU Fill vector $\mathbf{x}$ with uniformly random weights from $[-0.5, 0.5]$
Transfer $\mathbf{P}$, $\mathbf{R}$, $\mathbf{Q}$, $\mathbf{I}$ and $\mathbf{x}$ from system memory to GPU

The pseudocode of one time step of the EKF on GPU follows. The mentioned function names are those that were used from CUBLAS library.

On CPU Set weights to $\mathbf{x}$ and propagate network with actual input
On CPU Compute measurement matrix $\mathbf{H}$ using BPTT(h)
On CPU Compute $\mathbf{yg} = \mathbf{y} - g(\cdot)$
Transfer $\mathbf{H}$ from system memory to GPU

On GPU $\mathbf{A} = \mathbf{HP}$, $\mathbf{P}$ is symmetric – function `cublasSsymm()`
On GPU $\mathbf{C} = \mathbf{PH}^T$ – function `cublasSgemm()`
On GPU $\mathbf{B} = \mathbf{AH}^T$ – function `cublasSgemm()`
On GPU $\mathbf{B} = \mathbf{B} + \mathbf{R}$ – function `cublasSaxpy()`
Transfer $\mathbf{B}$ from GPU to system memory
On CPU Compute Cholesky factor of $\mathbf{B}$
Note: it is stored in the lower triangular part of $\mathbf{B}$
Transfer $\mathbf{B}$ and $\mathbf{yg}$ from system memory to GPU
On GPU $\mathbf{Z} = \mathbf{I}$ – function `cublasScopy()`
On GPU $\mathbf{BZ} = \mathbf{Z}$ – function `cublasStrsm()`
Note: solves $\mathbf{Z}$ in equation $\mathbf{BZ} = \mathbf{I}$
On GPU $\mathbf{B}^T\mathbf{Z} = \mathbf{Z}$ – function `cublasStrsm()`
Note: $\mathbf{Z}$ now contains inverse of $\mathbf{HPH}^T + \mathbf{R}$
On GPU $\mathbf{K} = \mathbf{CZ}$, $\mathbf{Z}$ is symmetric – function `cublasSsymm()`
On GPU $\mathbf{x} = \mathbf{Kyg} + \mathbf{x}$ – function `cublasSgemv()`
On GPU $\mathbf{P} = -\mathbf{KA} + \mathbf{P}$ – function `cublasSgemm()`
On GPU $\mathbf{P} = \mathbf{P} + \mathbf{Q}$ – function `cublasSaxpy()`
Transfer $\mathbf{x}$ from GPU to system memory

We used our own implementation of BPTT(h) and the pseudocode from [4] to compute Cholesky factorization on CPU. Each variable that was transferred to GPU used single precision floating point, which is the limitation of present GPUs. We used double precision floating point for everything else.

The implementation on CPU was essentially the same. The only difference was removal of transfers between CPU and GPU and the substitution of CUBLAS library functions by corresponding ATLAS library functions [11], which is straightforward as they both comply with Basic Linear Algebra Subprograms (BLAS) standard [2]. When configured to support threading, ATLAS library automatically takes advantage of multiple CPU cores to speed up its functions.

The hardware and software specifications used for all the tests:

- Intel Core2 Quad CPU Q6600 2.4 GHz – 4 cores
- nVidia GeForce 8800 GTX GPU – 128 parallel processors
- Ubuntu 7.10
- gcc 4.2.1
  compile flags: `-fomit-frame-pointer -mfpmath=sse -mmmx -msse -msse2 -msse3 -O3`
- ATLAS 3.8.1
  configured with `-b 32 -t 4 -D c -DPentiumCPS=2400`
- nVidia Toolkit 1.1
- nVidia Graphics Driver 169.12

ATLAS library was configured to support threading on CPU. The number of maximum threads was chosen to be 4 which is the number of cores on our test machine.

6 Experiment Description

The main goal of the experiment was to compare the performance of various implementations of the EKF and for comparison also of the BPTT(h) in a task typically used for training of recurrent neural networks.

We trained the recurrent neural network on the next symbol prediction. The predicted sequence[1] is based on real data obtained by quantization of activity changes of laser in chaotic regime. The bounds for quantization were chosen for positive small and big activity change and for negative small and big activity change. One symbol is assigned for each of these categories. The sequence therefore consists of four distinct symbols. This sequence contains relatively predictable subsequences followed by much less predictable events. The sequence length is 10000 symbols, we can therefore predict 9999 symbols.

The next symbol prediction procedure in general is the following: we present in every time step the first symbol in order, and the desired network's output is the next symbol in sequence order. The predictive performance was evaluated by means of a normalized negative log-likelihood (NNL), calculated over symbolic sequence from time step $t = 1$ to T [1]:

$$\mathrm{NNL} = -\frac{1}{\mathrm{T}} \sum_{\mathrm{t}=1}^{\mathrm{T}} \log_{|\mathrm{A}|} \mathrm{p}^{(\mathrm{t})}(\mathrm{s}^{(\mathrm{t})}) \tag{10}$$

where the base of the logarithm $|A|$ is the number of symbols in the alphabet A and $p^{(t)}(s^{(t)})$ is the probability of predicting symbol $s^{(t)}$ in the time step t. If NNL $= 0$, then the network predicts next symbol with 100% accuracy, while NNL ≥ 1 corresponds to a very inaccurate prediction (random guessing).

We chose the following initial values of parameters for the EKF: covariance matrix $\mathbf{P}_0 = 1000\ \mathbf{I}$, measurement noise covariance matrix $\mathbf{R}_0 = 100\ \mathbf{I}$ and process noise covariance matrix $\mathbf{Q}_0 = 0.0001\ \mathbf{I}$. We have not altered the covariance matrices $\mathbf{R}$ and $\mathbf{Q}$ during the training process. These values were inspired by [1] and were chosen also because they correspond with recommendations for EKF.

For the BPTT(h) method, we chose the learning parameter $\alpha = 0.2$ and parameter $h = 10$. The same value of parameter h was used also for the EKF. This choice is in accordance with the recommendations in [7].

We have used the training and testing procedure for the next symbol from this sequence prediction from [1]. We do not update the weights for the first 50 steps, in order to lessen the impact of initial recurrent neurons output values. The training then takes place during next 7949 symbols. The remaining 2000 symbols form the test data set, through which we compute the NNL. That terminates one cycle. A few cycles are usually sufficient for the EKF, much more for the BPTT(h) to converge to its best result (for details see [9,10]). However, we chose 20 training cycles in order to compare the precision of various EKF implementations in longer run.

[1] This sequence is available at `http://www2.fiit.stuba.sk/~cernans/main/download.html`

We have used one-hot encoding for inputs as well as for outputs, so we have 4 inputs as well as outputs – one input/output for each of 4 symbols. With this setup the prediction probability of the desired symbol is determined as the value of that element of the output vector which is reserved for given symbol, after normalizing the output vector (i.e. the sum of its elements equals 1). We chose the Elman's network architecture – i.e. the network with one hidden layer, which is recurrent.

7 Results

We conducted the described experiment with different implementations of EKF and with BPTT(h). We will use following abbreviations of corresponding implementations in this section:

EKF GPU EKF implemented using CUBLAS library
EKF ATLASf EKF implemented using ATLAS library with single precision functions and utilizing single thread
EKF ATLASft EKF implemented using ATLAS library with single precision functions and utilizing four threads
EKF ATLASd EKF implemented using ATLAS library with double precision functions and utilizing single thread
EKF ATLASdt EKF implemented using ATLAS library with double precision functions and utilizing four threads
BPTT(h) Truncated BPTT with double precision – h is truncation depth

Table 1. Elapsed time in seconds for 10 cycles of training and testing of recurrent neural network with various numbers of hidden neurons and thus weights

hidden neurons number of weights	4 56	8 140	12 256	16 404	30 1174	60 4144
EKF GPU	67s	108s	179s	279s	951s	4752s
EKF ATLASf	35s	96s	187s	325s	2218s	19858s
EKF ATLASft	35s	103s	199s	332s	1483s	13614s
EKF ATLASd	35s	96s	192s	340s	2005s	28677s
EKF ATLASdt	40s	107s	197s	354s	2026s	23315s
BPTT(h)	31s	76s	136s	214s	621s	2184s

Firstly, we present the results for the elapsed time during training of neural networks with various numbers of hidden neurons by each method in absolute numbers in Table 1, as well as relative to EKF GPU in Fig. 1. From these results we can see that the implementation of EKF on GPU provides significant speedup for larger networks, but is not beneficial for smaller networks. This stems from

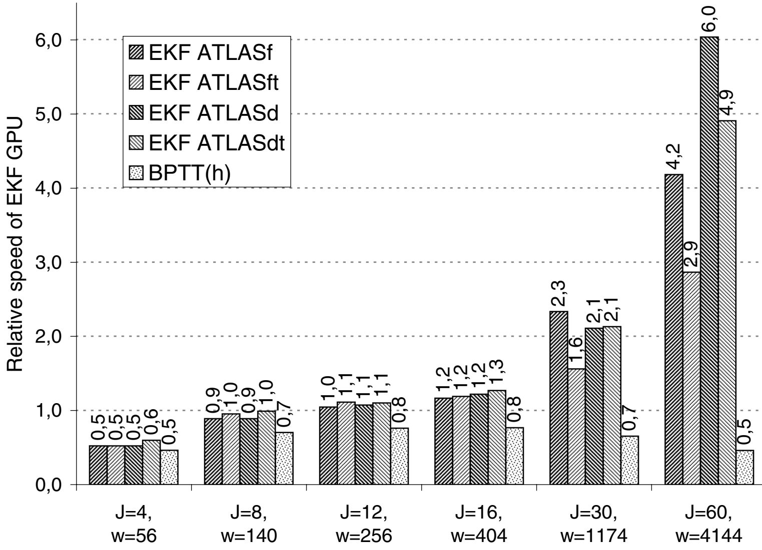

Fig. 1. Graphical representation of Tab. 1 – the speed comparison of EKF GPU relative to other implementations. J is number of hidden neurons and w the number of weights in recurrent neural network. The speedup is significant for networks with many weights even when compared to the threaded CPU version with single floating point precision. On the other hand it is not beneficial for small networks. Gradient descent method BPTT(h) is still faster but converges slowly and achieves worse results [9,10].

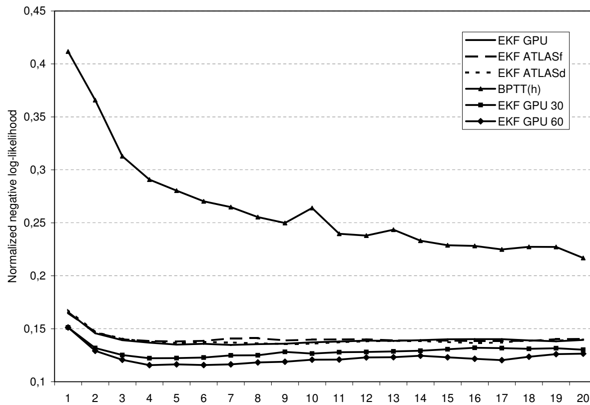

Fig. 2. The NNL dependence on the training cycles for various implementations and methods used for recurrent network with 16 hidden neurons. The exception is *EKF GPU 30* and *EKF GPU 60* which is a result for 30 and 60 hidden neurons respectively. The results show that (a) EKF is superior to gradient descent method BPTT(h) (for details see [9,10]); (b) the various EKF implementations achieve comparable results (see Fig. 3); (c) the increasing of hidden neurons is beneficial for this problem and was made more feasible by achieved speedup.

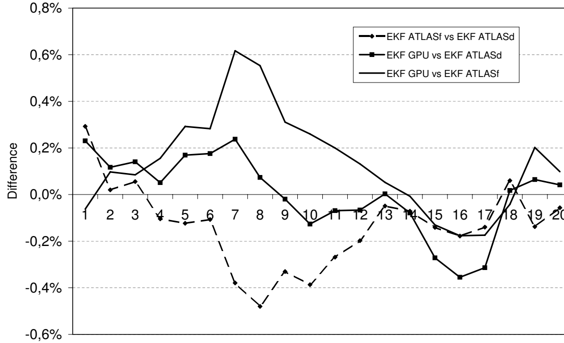

Fig. 3. Detailed view on results depicted in Fig. 2. Difference between achieved results of used implementations of EKF in every training cycle as a percentage of valid NNL interval length (i.e. 1). Positive values mean better achievement of first implementation than second implementation. The differences are negligible, all within 0.6%.

the fact, that the overhead of copying data from system memory to GPU and of parallelization alone is not worth when there is not much to compute.

The most "fair" comparison of EKF GPU is with EKF ATLASft, because both utilize parallelization and work with single precision floating point arithmetic. EKF GPU achieves nearly 3 times speedup for largest network when compared to EKF ATLASft. The most significant acceleration by using computation with EKF GPU – 6 times on largest network – is achieved when compared to EKF ATLASd, which is probably most similar to existing implementations of EKF.

In Fig. 1 we can further see that method BPTT(h) is consistently the fastest, whereas EKF ATLASd is generally the slowest one. The obvious question is if the achieved results are on the one hand worth the speed degradation when compared with BPTT(h) and on the other hand worth the significant speedup when compared with potentially more precise EKF ATLASd. The answer is in Fig. 2 which for each method shows the average results when used for training 10 randomly initialized networks. In this graph we can see the superior convergence and achieved result of EKF when compared with BPTT(h) (see also [9,10]). We can also see the comparable results of various implementations of EKF, which is more obvious from Fig. 3. It means the used floating point arithmetic does not play a significant role in EKF performance in this experiment.

The achieved speedup of EKF GPU made it also more feasible to conduct thorough experiments with larger networks, as seen in Fig. 2. This also justifies the increasing of number of hidden neurons for this problem, as the best achieved result is by networks with 60 hidden neurons (NNL=0.1156 in training cycle 4).

8 Conclusion

In this paper we have shown that the implementation of the most computational intensive tasks in the Extended Kalman filter using the CUBLAS library for

nVidia graphics processing units can significantly speed up the recurrent neural network training process. What is important, the speedup was achieved only by means of available library functions, one does not have to overcome steep learning curve of GPU architecture and its efficient parallel programming.

The drawback in current GPUs is the lack of double precision floating point arithmetic. This affects the numerical stability as well as overall achieved result. In our experiments we experienced the numerical stability problem when computing Cholesky factorization of matrix which became non positive definite. This was remedied by restarting the training process with different initial weights. Since the problem usually arose in the early training cycles, and the training is fast, the restarting did not cause significant delays.

The reduced precision was not significant problem in our experiments. However, the recommended practice for the time being would be to tune the training parameters using EKF on GPU and to use CPU only version with double precision for final experiments.

The presented method can be further enhanced by implementing more parts of algorithm on the GPU and reduce thus the need to copy data between CPU and GPU which is a costly operation. This will be the topic of our further research, namely implementing Cholesky factorization [4], neural network propagation [6] and BPTT(h) on graphic processing units.

References

1. Čerňanský, M., Makula, M., Beňušková, L.: Processing Symbolic Sequences by Recurrent Neural Networks Trained by Kalman Filter-Based Algorithms. In: SOFSEM 2004 (2004) ISBN 80-86732-19-3 58–65
2. Dongarra, J.: Basic Linear Algebra Subprograms Technical Forum Standard. Int. J. of High Performance Applications and Supercomputing 16(1), 1–111 (2002)
3. Haykin, S.: Kalman Filtering and Neural Networks. John Wiley & Sons, Inc., New York (2002) ISBN: 0-471-36998-5
4. Jung, J.: Cholesky Decomposition and Linear Programming on a GPU. Sholarly Paper. University of Maryland
5. Kalman, R.E.: A New Approach to Linear Filtering and Prediction Problems. Trans. of the ASME, Series D, Journal of Basic Engineering 82, 35–45 (1960)
6. Kyoung-Su, O., Keechul, J.: GPU Implementation of Neural Networks. Pattern Recognition 37, 1311–1314 (2004)
7. Patel, G.S.: Modeling Nonlinear Dynamics with Extended Kalman Filter Trained Recurrent Multilayer Perceptrons. McMaster University (2000)
8. Prokhorov, D.V.: Kalman Filter Training of Neural Networks: Methodology and Applications. Ford Research Laboratory (2002)
9. Trebatický, P.: Recurrent Neural Network Training with the Kalman Filter-based Techniques. Neural network world 15(5), 471–488 (2005)
10. Trebatický, P.: Recurrent Neural Network Training with the Extended Kalman Filter. IIT SRC: Proc. In: Informatics and Info. Technologies, 57–67 (2005)
11. Automatically Tuned Linear Algebra Software (ATLAS), `http://math-atlas.sourceforge.net`
12. Compute Unified Device Architecture (CUDA), `http://www.nvidia.com/object/cuda_home.html`