Training Recurrent Neural Network Using Multistream Extended Kalman Filter on Multicore Processor and Cuda Enabled Graphic Processor Unit

Michal Čerňanský

Faculty of Informatics and Information Technologies, STU Bratislava, Slovakia
cernansky@fiit.stuba.sk

Abstract. Recurrent neural networks are popular tools used for modeling time series. Common gradient-based algorithms are frequently used for training recurrent neural networks. On the other side approaches based on the Kalman filtration are considered to be the most appropriate general-purpose training algorithms with respect to the modeling accuracy. Their main drawbacks are high computational requirements and difficult implementation. In this work we first provide clear description of the training algorithm using simple pseudo-language. Problem with high computational requirements is addresses by performing calculation on Multicore Processor and CUDA-enabled graphic processor unit. We show that important execution time reduction can be achieved by performing computation on manycore graphic processor unit.

1 Introduction

To process data with spatio-temporal structure recurrent neural networks (RNNs) were suggested. RNNs were successfully applied in many real-life applications where processing time-dependent information was necessary. Unlike feedforward neural networks, units in RNNs are fed by activities from previous time steps through recurrent connections. In this way contextual information can be kept in units' activities, enabling RNNs to process time series. Common algorithms usually used for RNN training are based on gradient minimization of the output error. Backpropagation through time (BPTT) [1] consists of unfolding a recurrent network in time and applying the well-known backpropagation algorithm directly. Another gradient descent approach, where estimates of derivatives needed for evaluating error gradient are calculated in every time step in forward manner, is the real-time recurrent learning (RTRL) [2].

Probably the most successful training algorithms are based on the Kalman filtration (KF) [3,4,5]. The standard KF can be applied to a linear system with Gaussian noise. A nonlinear system such as RNNs with sigmoidal units can be handled by extended KF (EKF). In EKF, linearization around current working point is performed and then standard KF is applied. In case of RNNs, algorithms similar to BPTT or RTRL can be used for linearization. Methods based on the Kalman filtration often outperform common gradient-based algorithms. Multistream EKF training proved to be very successful approach for training relatively large recurrent neural networks to the complex real-life tasks from industry [6,7]. Multiple instances of the same network are trained on different data streams in the same time and coordinate weight changes are performed.

C. Alippi et al. (Eds.): ICANN 2009, Part I, LNCS 5768, pp. 381–390, 2009.

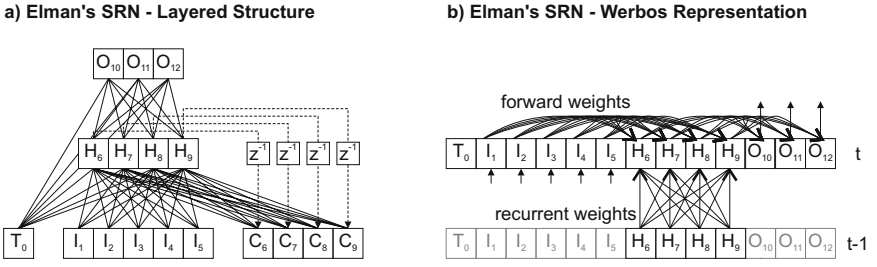
Here we address the problem of high computational requirements by using high performance of many-core processor of graphic unit. We use Compute Unified Device Architecture (CUDA) which is general purpose parallel computing architecture enabling developers to use NVIDIA's graphic processor units (GPUs) for solving complex computational problems [8]. GPUs are now highly parallel multithreaded many-core processors with high memory bandwidth capable to perform more than 10^{12} floating point operations per second. GPUs are well suited to address problems that can be expressed as data parallel computations - the same program is executed on many data elements in parallel. CUDA has been already applied in many applications such as video processing, pattern recognition or physics simulations. In [9] authors used CUDA CBLAS library for linear algebra operations of RNN EKF training.

In this work we provide experiments with RNNs trained by multistream extended Kalman filter (MSEKF). This algorithm was successfully used in several real world applications and is considered to be the state-of-the-art technique for training recurrent networks. We first describe simple and elegant way of encoding recurrent neural network into data structures inspired by [1,10]. Then we present MSEKF in the form of simple algorithm in pseudo-language similar to Pascal. Both forward propagation of the signal and MSEKF training algorithms are given in almost copy and paste form. We discuss details of two implementations: the first using standard CPU and the second using CUDA enabled GPU. Finally we compare executions times of both implementations for different number of hidden units and different number of streams.

2 Encoding Recurrent Neural Network into Data Structures

Elman's simple recurrent network (SRN) proposed in [11] is probably the most widely used RNN architecture. Context layer keeps activities of hidden (recurrent) layer from previous time step. Input layer together with context layer form extended input to the hidden layer. Elman's SRN composed of 5 input, 4 hidden a 3 output units is shown in Fig. 1a. Context units C6 to C9 hold activities of hidden units H6 to H9 from previous time step and together with input layer activities I1 to I5 they serve as extended input to the hidden units H6 to H9. All hidden and output (O10 to O12) units are connected to the special input unit T0 through threshold connections. Threshold unit T0 is set to constant value of 1.

In general, units of a neural network need not to be organized in layers. They can be randomly interconnected as soon as the directed graph representation of the network having vertices as units and edges as connections does not includes any cycle. In other words, activity of a unit can be calculated knowing activity of each unit from which a weight connection exist to the given unit. Since no cyclic dependencies exist, activities of all units can be calculated. This notion also holds for RNNs, although recurrent connections form a kind of cycle in the network. But recurrent connections are sourced by activities already calculated in previous time steps and hence pose no problem in calculating actual activities. Only non-recurrent forward connections must not form a cycle in the graph representation of an RNN. Although this condition may seem to be restrictive, it is met in all commonly used feed-forward and recurrent multilayer perceptron architectures.



c) Weight Connections

Weight Index		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
Source Unit	wSource	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
Destination U.	wDest	6	6	6	6	6	6	6	6	6	6	7	7	7	7	7	7	7	7	7	7	8	8	8	8	8	8	8	8	8	8
Time Delay	wDelay	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	1	1	1	1
Value	wValue																														

Weight Index		30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54
Source Unit	wSource	0	1	2	3	4	5	6	7	8	9	0	6	7	8	9	0	6	7	8	9	0	6	7	8	9
Destination U.	wDest	9	9	9	9	9	9	9	9	9	9	10	10	10	10	10	11	11	11	11	11	12	12	12	12	12
Time Delay	wDelay	0	0	0	0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Value	wValue																									

d) Units

Unit Index		0	1	2	3	4	5	6	7	8	9	10	11	12
First Weight	uFirstWeight							0	10	20	30	40	45	50
Last Weight	uLastWeight							9	19	29	39	44	49	54
Type	uType	T	I	I	I	I	I	H	H	H	H	O	O	O
Act. Function	uActFunc							SGM	SGM	SGM	SGM	LIN	LIN	LIN

Fig. 1. a) Layered structure of Elman's SRN. b) Werbos representation of the Elman's SRN. c) Table gathering information on weight connections. d) Table gathering information on units.

Werbos representation of the Emnan's SRN is shown in Fig. 1b. All units of the network can be indexed from 0 to $NU - 1$, where NU is the number of all units. The first unit - special input unit corresponding to the bias weight, is followed by input units, than by hidden and output units. Arranging hidden and output units is not necessary, every non-input unit can be output unit and desired activities can be specified during training phase for that unit. Unit with index i can have forward connections starting only from units with smaller indices (from index 0 to $i - 1$). Hence edges corresponding to forward weights in Fig. 1b are oriented from left to right. For every weight, indices of the source and destination unit are kept (arrays `wSource` and `wDest`), together with the time delay (`wDelay`) related to the weight connection. Weights can be also sorted, first by corresponding destination unit index then by connection time delay and finally by source unit index in ascending order (Fig. 1c). Forward weights have time delay of 0, what means that the actual step t activity of the source unit is fed through this connection to the destination unit. Recurrent connection have time delay t_d greater than 0 and activity from previous time step $t - t_d$ is fed through this connections. For every non-input unit the first and the last weight indices are stored (`uFirstWeight`, `uLastWeight`) together with other useful information such as unit's type (`uType`) and unit's activation function (`uActFunc`) (Fig. 1d). Usufulness of this network encoding can be seen in algorithmic description for forward propagation of the signal in custom RNN for multistream EKF (Fig. 2).

3 Multistream Extended Kalman Filter

Training of Elman's SRN, and generally any other multilayer perceptron network (recurrent or not), can be regarded as an optimal filtering [10]. The set of EKF equations for the network training can be formulated as follows:

$$\mathbf{K}(t) = \mathbf{P}(t-1)\mathbf{H}^T(t)[\mathbf{H}(t)\mathbf{P}(t-1)\mathbf{H}^T(t) + \mathbf{R}(t)]^{-1}, \tag{1}$$

$$\mathbf{P}(t) = \mathbf{P}(t-1) - \mathbf{K}(t)\mathbf{H}(t)\mathbf{P}(t-1) + \mathbf{Q}(t), \tag{2}$$

$$\mathbf{W}(t) = \mathbf{W}(t-1) + \mathbf{K}(t)[\mathbf{D}(t) - \mathbf{O}(t)]. \tag{3}$$

Let n_w and n_o denote the number of all network weights and number of output units, respectively. $\mathbf{W}$ is a vector of all weights of the length n_w. $\mathbf{H}$ is the Jacobian matrix, $n_o \times n_w$, calculated in every time step and containing in rows the derivatives of corresponding output activity with respect to all weights. These derivatives can be calculated similarly to the RTRL or BPTT algorithms. $\mathbf{P}$ is the $n_w \times n_w$ error covariance matrix, it holds error covariances corresponding to each pair of network weights. The $n_w \times n_o$ matrix $\mathbf{K}$ called the Kalman gain is used in updating the weights $\mathbf{W}$ according to the difference between the desired output vector $\mathbf{D}$ and actual network output $\mathbf{O}$. The $n_o \times n_o$ matrix $\mathbf{R}$ stands for the measurement noise covariance matrix and similarly to the learning rate in RTRL or BPTT can control the training speed of EKF. Note, that small process noise is still considered: the $n_w \times n_w$ matrix $\mathbf{Q}$ stands for the process noise covariance matrix. Nonzero process noise improves convergence of the filter.

For multistream EKF training the training dataset should be split into several parts [10]. Training dataset in practical application is frequently already naturally partitioned, since it often consists of multiple subsets of different nature corresponding to different working conditions. For a chosen number of streams n_s the training sequences of the same length for each stream are selected from the different parts of the training set. For each stream the different instance of the same network is presented with the actual input-output pattern from the corresponding stream. Propagation for each stream is performed as in the case of several single networks trained on different patterns. Also derivatives of network outputs with respect to network weight are calculated in the same way as for independent networks. The procedure could follow by performing n_s independent EKF steps and by calculating overall weight changes by averaging partially calculated weight changes from each stream. But this is not the case of multistream EKF. Instead calculated derivatives of each network instance are concatenated into the single measurement update matrix $\mathbf{H}(t)$. If $\mathbf{H}_i(t)$ would be matrix of partial derivatives of output units with respect to weight for single-stream EKF training corresponding to the stream i in time step t, the matrix $\mathbf{H}(t)$ can be expressed as $\mathbf{H}(t) = (\mathbf{H}_1(t)\mathbf{H}_2(t)\dots\mathbf{H}_{n_s}(t))$. In the similar way vector of desired value $\mathbf{D}(t)$ and vector of calculated output activities $\mathbf{O}(t)$ are formed by concatenating corresponding elements, $\mathbf{D}(t) = (\mathbf{D}_1(t)^T\mathbf{D}_2(t)^T\dots\mathbf{D}_{n_s}^T(t))^T$ and $\mathbf{O}(t) = (\mathbf{O}_1(t)^T\mathbf{O}_2(t)^T\dots\mathbf{O}_{n_s}^T(t))^T$. Except for these changes the EKF equations remain the same (Eq. 1 to Eq. 3). Multistream EKF approach with BPTT routine for the derivative calculation is provided to make this approach clearer. First, the forward propagation of multiple streams is given in Fig. 2 and then multistream EKF training step is decribed in Fig. 3.

```
NS                                      - number of streams
NW                                      - number of weight connections
NU                                      - number of units (threshold unit and input units also count)

wSource[0..NW-1]                        - source node indices
wDest[0..NW-1]                          - destination node indices
wDelay[0..NW-1]                         - connection time delays
wValue[0..NW-1]                         - weight connection strengths

uFirstWeight[0..NU-1]                   - indices of the first weight connection associated with the unit
uLastWeight[0..NU-1]                    - indices of the last weight connection associated with the unit
uType[0..NU-1]                          - unit types (THRESHOLD, INPUT, HIDDEN and OUTPUT)
uActFunc[0..NU-1]                       - unit activation-function types (SGM, LIN, ...)

ACT[0..NU-1,0..NSTEPS-1,0..NS-1]  - all unit activities in all time steps for all streams
ACTD[0..NU-1,0..NSTEPS-1,0..NS-1] - act. func. derivatives in all time steps for all streams

ActFunc(iact,actf)                      - activation function calculation based on the act. func. type
ActFuncDer(iact,actf)                   - derivative of the activation function

ts                                      - actual time step

Input(ui,ts,si)       - returns value for the input unit ui in time step ts for stream si
Output(ui,act,ts,si) - set network output to the value act for stream si
Target(ui,ts,si)      - returns desired output unit ui activity in time step ts for stream si
```

```
1  for si=0 to NS-1 do
2    for ui=0 to NU-1 do
3    begin
4      if uType[ui] = THRESHOLD then ACT[ui,ts,si] := 1.0;
5      else if uType[ui] = INPUT then ACT[ui,ts,si] := Input(ui,ts,si);
6      else
7      begin
8        iact := 0.0;
9        for wi := uFirstWeight[ui] to uLastWeight[ui] do
10         iact +:= wValue[wi]*ACT[wSource[wi],ts-wDelay[wi],si];
11       ACT[ui,ts,si] := Sgm(iact);
12       ACTD[ui,ts,si] := SgmDer(iact);
13     end;
14     if uType = OUTPUT then Output(ui,ACT[ui],ts,si);
15   end;
```

Fig. 2. Forward propagation for multistream EKF

Multistream forward propagation consists of NS consecutive single-stream propagations (cycle on lines 1 to 15). The first unit with index 0 correspond to the bias weight, its activity is always set to the value of 1 (line 4). Activities of input units are set to the actual input to the network (line 5). Internal activities of hidden and output units are calculated by multiplying unit's weights with source activities from corresponding time step (lines 8 to 10), operator "+:=" stand for addition of the right-hand side expression to the variable on the left-hand side. For forward weights having time delay of 0 already calculated activity from actual time step are used, for recurrent weights activities calculated in past time steps are used. Unit's activity is calculated by passing internal activity to the activation function (line 11). The derivative (line 12) calculation is only needed when gradient-based adaptation step such as backpropagation follows the forward propagation. Activities calculated for the output units are sent as the network output to the exterior (line 14).

```
NS                                  - number of streams
NO                                  - number of network output units

oIndex[0..NO]                       - indices of output units
winSize                             - unfolding window size

DO_DW[0..NO-1,0..NW-1,0..NS-1] - derivatives of net. outputs with respect to all weights
DO_DNA[0..NU-1,0..winSize-1]   - backpropagated signal

H[0..NO*NS-1,0..NW-1]               - augmented Jacobian matrix
P[0..NW-1,0..NW-1]                  - error covariance matrix
Q[0..NW-1,0..NW-1]                  - process noise covariance matrix
R[0..NO-1,0..NO-1]                  - measurement noise covariance matrix
W[0..NW,0..0]                       - estimated vector of network weights (EKF state vector)
K[0..NW-1,0..NO*NS-1]               - augmented Kalman gain
D[0..NO*NS-1,0..0]                  - augmented vector of desired output values
O[0..NO*NS-1,0..0]                  - augmented of calculated output values

Tr(X)                               - transpose of the matrix X
Inv(X)                              - inverse of the matrix X
```

```
1  for oui:=0 to NO-1 do for wi:=0 to NW-1 do for si:=0 to NS-1 do
2    DO_DW[oui,wi,si] := 0.0;
3
4  for si:=0 to NS-1 do
5    for oui:=0 to NO-1 do
6    begin
7      for hi:=0 to winSize-1 do for ui:=0 to NU-1 do DO_DNA[ui,hi] := 0.0;
8      DO_DNA[oIndex[oui],0] := 1.0;
9
10     for hi:=0 to winSize-1 do
11       for ui:=NU-1 downto 0 do
12       begin
13         if uType[ui] = INPUT then break;
14         DO_DNA[ui,hi] := DO_DNA[ui,hi]*ACTD[ui,ts-hi,si];
15         for wi := uLastWeight[ui] downto uFirstWeight[ui] do
16         begin
17           if (uType[wSource[wi]] <> INPUT) AND
18              (uType[wSource[wi]] <> THRESHOLD) AND
19              (wDelay[wi]+hi < winSize)) then
20                DO_DNA[wSource[wi],wDelay[wi]+hi] +:=
21                  wValue[wi]*DO_DNA[ui,hi];
22           DO_DW[oui,wi,si] +:=
23             DO_DNA[ui,hi]*ACT[wSource[wi],ts-hi-wDelay[wi],si];
24         end;
25       end;
26   end;
27
28 for wi:=0 to NW-1 do W[wi,0] := wValue[wi];
29
30 for si:=0 to NS-1 do
31   for oui:=0 to NO-1 do
32   begin
33     D[oui+si*NS,0] := Target(ui,ts,si);
34     O[oui+si*NS,0] := ACT[oIndex[oui],tsm,si];
35     for wi:=0 to NW-1 do H[oui+si*NS,wi] := DO_DW[oui,wi,si];
36   end;
37
38 K := P * Tr(H) * Inv(H * P * Tr(H) + R);
39 P := P - K * H * P  + Q;
40 W := W + K * (D - O);
41
42 for wi:=0 to NW-1 do wValue[wi] := W[wi,0];
```

Fig. 3. Multistream EKF with derivatives calculated by BPTT

The quantities DO_DW (derivatives of output activities with respect to weights) are initialized to 0 on lines 1 and 2. Then for every stream and every output unit the backpropagation through time is performed (cycle on lines 4 to 26). For one output unit the "error" signal of corresponding to this output unit is set to 1 (line 8), remaining elements of the array DO_DNA (derivatives of output activity with respect to the internal units' activities) were set to 0 (line 7) and are to be calculated in the following cycle of truncated backpropagation throught time (lines 10 to 25). Computation of truncated BPTT consists of backpropagating the "error" signal winSize time steps back (lines 10 to 25). For each unit ui (cycle on lines 11 to 25) and all its connections (cycle on lines 15 to 24) derivatives DO_DNA[ui,hi] are backpropagated through corresponding connections to the derivatives in corresponding time (lines 20 and 21). Signal is not backpropagated to the input or threshold units (condition on lines 17 and 18). The test (line 19) is also performed to ensure, that the array DO_DNA on lines 20 and 21 is accessed properly. Quantities DO_DW are built on lines 22 and 23.

Augmented arrays for desired and calculated output activities (D and O) are filled on lines 33 and 34 and the Jacobian matrix H on line 35. EKF update is then performed and weights are updated (line 42). In this section (lines 38 to 40) operators *,+ and - denote matrix multiplication, addition and subtraction respectively. Since network weights are changed only by EKF part of the algorithm they do not need to be filled up in every time step and line 28 can be removed. We keep it there as a reminder than array wValueW and vector W are the same quantities.

4 Implementation Details

Time complexity of the algorithm can be easily determined from the pseudo-language description of one cycle provided in Fig. 3. The training step can be divided into two parts: first truncated BPTT is done for every stream and every output and than KF step is performed. Hence time complexity can be estimated as:

$$T_{\mathrm{MSEKF}} = T_{\mathrm{BPTT}} + T_{\mathrm{KF}} = O(n_s \times n_o \times n_w \times h) + O(n_o \times n_w^2). \tag{4}$$

Since for huge networks number of streams n_s, number of output units n_o and the truncated BPTT window size h are smaller than the number of weights n_w, the time complexity of MSEKF can be expressed as:

$$O_{\mathrm{MSEKF}} \approx O(n_w^2). \tag{5}$$

Alhough in practice multiplicative constants (not revealed in big-O notation) do matter we can already see that the computation is dominated by the time complexity of the KF part.

Standard way how implement operations with matrices is the usage of some linear algebra package. All major processor vendors offer high-speed parallel multithreaded versions of BLAS (Basic Linear Algebra Subprograms). Intels solution is MKL (Math Kernel Library) offering BLAS and LAPACK (Linear Algebra PACKage) interfaces. Other choices are ACML (AMD Core Math Library) tailored for AMD processors and GotoBLAS library which is probably the fastest BLAS library available. Since the target

platform is a computer equipped with Intel Core i7 quad core processor and GotoBLAS has not yet been optimized for Core i7 architecture we have chosen MKL as a linear algebra package for CPU implementation of MSEKF.

CUDA platform contains CUBLAS (CUDA BLAS) library benefiting from parallelism of many-core GPU architecture but providing standard BLAS application interface. Whole KF part was implemented to be performed on the GPU including implementation of cholesky solver used for finding Kalman gain K. Only vector $\mathbf{E} = \mathbf{D} - \mathbf{O}$ and matrix $\mathbf{H}$ are transfered into the GPU device and weight vector $\mathbf{W}$ is transfered back from the CUDA device.

Other source of parallelism is the truncated BPTT step since iterations are independent. $n_o \times n_s$ iterations are performed in the truncated BPTT part of the MSEKF step. The only modification is that `NO_DNA` array cannot be shared between iterations executed in paralel. There are several programming models that can be used to improve performance of this part of the algorithm by splitting computation into multiple threads. We have chosen Intel's Threading Building Blocks (TBBs) and "parallel_for" construct was used in straightforward way. Implementation of BPTT step using CUDA is the task for near future.

5 Results

Tests for both CPU and GPU implementations of MSEKF were conducted on the same machine running Microsoft Windows XP SP 3 equipped with Intel Core i7 Nehalem processor operating at 2.67GHz, 3GB of RAM and Nvidia GeForce N280 graphic card. Intels hyperthreading technology was turned off in BIOS as recommended in MKL manual.

We performed tests with Elman's simple recurrent network trained for the next value prediction of Santa Fe laser sequence. Since RNNs are also frequently used by cognitive science community for modeling symbolic sequences the second dataset was generated by Elman's grammar [12]. In this case the networks were trained on the next symbol prediction task. The length of both sequences was 10000 values, while training on Laser sequence RNNs has single input and output unit, for Elman datasets RNN's input and output layer consist of 24 units.

We present simulation results for various number of streams and number of hidden units. MSEKF training run times in seconds are summarized in the Tab. 1 and correspond to one epoch - one presentation of the training sequence. For a given number of streams n_s the training sequence is divided into n_s parts of the same length and the network perform $10000/n_s$ MSEKF training cycles per epoch. Hence higher number of streams does not mean that more operations were performed during training.

We provide results for computation performed in single precision only for two reasons. First using double precision did not brink any difference considering resulting performance (similarly to [9]). We also encountered no problems with numerical stability. The second reason is that GPUs performance in double precision is lower since graphic hardware is optimized for single precision computation. For applications where numerical precision is crucial performing computation on standard multicore processor may be a better choice.

Table 1. Simulations times in seconds for BPTT and EKF parts of CPU and CUDA implemetations of MSEKF

LASER	BPTT - no TBB	BPTT - TBB	EKF - CPU	EKF - CUDA
MSEKF-16HU-1S	0.001	0.001	1.01	5.02
MSEKF-16HU-4S	0.25	0.004	0.51	1.51
MSEKF-16HU-16S	0.18	0.065	0.52	0.76
MSEKF-16HU-64S	0.17	0.045	0.89	0.76
MSEKF-64HU-1S	2.00	2.02	210.51	124.87
MSEKF-64HU-4S	2.25	0.51	60.16	33.63
MSEKF-64HU-16S	2.18	0.56	20.67	11.29
MSEKF-64HU-64S	2.25	0.55	21.00	8.84
ELMAN	BPTT - no TBB	BPTT - TBB	EKF - CPU	EKF - CUDA
MSEKF-16HU-1S	11.00	3.01	46.51	35.59
MSEKF-16HU-2S	10.99	3.06	67.18	37.63
MSEKF-16HU-4S	11.49	3.02	111.74	69.21
MSEKF-16HU-8S	11.72	2.92	205.48	120.62
MSEKF-32HU-1S	27.99	7.15	184.08	95.12
MSEKF-32HU-2S	27.99	7.67	217.80	104.51
MSEKF-32HU-4S	29.00	7.42	322.48	177.40
MSEKF-32HU-8S	29.32	7.32	548.07	319.98

We provide results for both unparallelized (no TBB) and parallelized (TBB) BPTT part. Please note that the time requirements of BPTT part are the same for fixed number of streams, since the same number of backward propagations through time is performed for one epoch. Unsuprisingly parallelized version of the BPTT takes much less time than its unparallelized counterpart. Almost 4 times better performance was achieved, hence the performance of this part scales well with the number of cores.

As can be seen from Tab. 1 significant run time reduction for larger networks can be obtained by performing MSEKF training on CUDA-enabled GPU. On the other side standard CPU performs well for smaller networks since massive parallelism of many-core GPUs is not used. In general smaller networks benefit neither from multi-core nor many core processors because of small level of parallelism when doing calculations with small matrices.

6 Conclusion

Multistream extended Kalman filter is probably the most successful algorithm for training recurrent neural networks. The main drawback of MSEKF is sever computational requirements in comparing with common approaches such as BPTT or RTRL. This usually prevents thorough search for better parameters or other experimentation when performing model selection. In this paper we first provide detailed description of the algorithm using simple pseudo-language. Algorithm is almost in a copy and paste form. Then we provide results of implementations targeting CPU and CUDA-enabled GPU platforms. We show that significant reduction of execution time can be achieved by performing calculations on graphical processing units when training large networks.

Acknowledgments. This work was supported by the grants Vega 1/0848/08, Vega 1/0822/08 and APVV-20-030204.

References

1. Werbos, P.: Backpropagation through time; what it does and how to do it. Proceedings of the IEEE 78, 1550–1560 (1990)
2. Williams, R.J., Zipser, D.: A learning algorithm for continually running fully recurrent neural networks. Neural Computation 1, 270–280 (1989)
3. Williams, R.J.: Some observations on the use of the extended Kalman filter as a recurrent network learning algorithm. Technical Report NU-CCS-92-1, Northeastern University, College of Computer Science, Boston, MA (1992)
4. Čerňanský, M., Beňušková, Ľ.: Simple recurrent network trained by RTRL and extended Kalman filter algorithms. Neural Network World 13(3), 223–234 (2003)
5. Trebatický, P.: Recurrent neural network training with the kalman filter-based techniques. Neural network world 15(5), 471–488 (2005)
6. Feldkamp, L., Prokhorov, D., Eagen, C., Yuan, F.: Enhanced multi-stream Kalman filter training for recurrent networks. In: Suykens, J., Vandewalle, J. (eds.) Nonlinear Modeling: Advanced Black-Box Techniques, pp. 29–53. Kluwer Academic Publishers, Dordrecht (1998)
7. Prokhorov, D.V.: Toyota prius hev neurocontrol and diagnostics. Neural Networks 21, 458–465 (2008)
8. NVIDIA: NVIDIA CUDA programming guide. Technical report (2008)
9. Trebatický, P.: Neural network training with extended kalman filter using graphics processing unit. In: Kůrková, V., Neruda, R., Koutník, J. (eds.) ICANN 2008, Part II. LNCS, vol. 5164, pp. 198–207. Springer, Heidelberg (2008)
10. Prokhorov, D.V.: Kalman filter training of neural networks: Methodology and applications. In: Tutorial on IJCNN 2004, Budapest, Hungary (2004)
11. Elman, J.L.: Finding structure in time. Cognitive Science 14, 179–211 (1990)
12. Elman, J.: Distributed representations, simple recurrent networks, and grammatical structure. Machine Learning 7, 195–225 (1991)