FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks

Yijin Guan[1], Zhihang Yuan[1], Guangyu Sun[1,3], Jason Cong[2,3,1]
[1]Center for Energy-Efficient Computing and Applications,Peking University, China
[2]Computer Science Department, University of California, Los Angeles, USA
[3]PKU/UCLA Joint Research Institute in Science and Engineering

Abstract— Long Short-Term Memory Recurrent neural networks (LSTM-RNNs) have been widely used for speech recognition, machine translation, scene analysis, etc. Unfortunately, general-purpose processors like CPUs and GPGPUs can not implement LSTM-RNNs efficiently due to the recurrent nature of LSTM-RNNs. FPGA-based accelerators have attracted attention of researchers because of good performance, high energy-efficiency and great flexibility. In this work, we present an FPGA-based accelerator for LSTM-RNNs that optimizes both computation performance and communication requirements. The peak performance of our accelerator achieves 7.26 GFLOP/S, which significantly outperforms previous approaches.

I. Introduction

In recent years, research in deep learning algorithms has achieved great progress in model accuracy and training methods, which makes deep learning a hot topic in computer science. When it comes to applications that process sequential data, such as speech recognition, the performance of conventional neural networks is not satisfactory, because they do not take timing information into account. As a result, it is natural to design a new network architecture to generate outputs based on previous input sequence.

Recurrent neural network (RNN), a well-known deep learning algorithm, has been extensively applied in various applications like speech recognition[7][14], text recognition[13], machine translation[16], scene analysis[4], etc. By taking advantage of previous outputs as inputs for current prediction, RNNs show a strong ability to learn and predict sequential data. To further improve the prediction accuracy of RNNs, Long Short-Term Memory (LSTM), a learned memory controller, is combined with standard RNN designs. In recent years, research on LSTM-RNNs has grown very fast due to the rapid development of modern applications based on deep learning algorithms.

Though the combination of LSTM and standard RNNs improves the prediction accuracy, it also makes the computation pattern and data access pattern more complex. Due to the recurrent nature of LSTM-RNNs, it is quite difficult for CPUs to accomplish LSTM-RNN computation in parallel. GPGPUs can explore little parallelism due to the branching operations and relatively small model size of LSTM-RNNs. The disappointing performance of LSTM-RNN on general-purpose processors can not meet the requirements of real-time inference in modern applications. It means that a high-performance accelerator is highly desired. Taking performance, energy-efficiency and flexibility into consideration, an FPGA-based accelerator is a good choice, and previous designs have showed great benefits brought by FPGA-based accelerators[5][10][17].

Typically in practice, an LSTM-RNN model must be trained off-line for a fairly good prediction accuracy, then it can be applied to various real-life applications. As a result, the processing speed of on-line inference is the key point of LSTM-RNN deployment, and we focus on accelerating the inference phase of LSTM-RNNs in this work. The inference phase requires carefully designed computation engines and data management modules. In this work, we carefully analyze the characteristics of LSTM-RNN inference, and propose several optimization strategies for hardware implementation. We implement an FPGA-based accelerator with significant performance improvement.

In summary, this paper makes following contributions:

- At the architecture level, we optimize the LSTM-RNN accelerator to meet both computation performance and communication requirements.
- Our implementation of the LSTM-RNN accelerator integrates a set of high-performance computation engines and a data dispatcher. These designs significantly improve the overall performance.
- As a case study, we implement an LSTM-RNN accel-

erator for a real-life speech recognition model. The peak performance of our design is 7.26 GFLOP/S, which outperforms all previous works.

The remainder of this paper is organized as follows: Section II provides the basics of RNN and LSTM. Section III presents our architecture level optimization and provides our analysis. Section IV provides the details of our hardware implementation. The experimental setup and results can be found in Section V, and Section VI compares our implementation with related work. Section VII concludes this paper and discusses future work.

II. Background

In this section, we introduce some basic concepts of RNN, LSTM cell and LSTM-RNNs.

A. RNN Basis

Recurrent neural networks (RNNs) were first invented to deal with sequential data, which requires the model to learn from previous states. Fig.1 compares the basic architectures of a standard feed-forward neural network and a standard RNN. As Fig.1 illustrates, a standard feed-forward neural network (Fig.1.a) connects all the layers in a uniform direction, while RNN (Fig.1.b) adds additional connections that pass the previous outputs of hidden layers back to the current input. Unlike deep neural networks, standard RNN exhibits a deep structure in time rather than in space. As a result, standard RNN can take the time dimension into account, and generates outputs on the basis of previous input sequence.

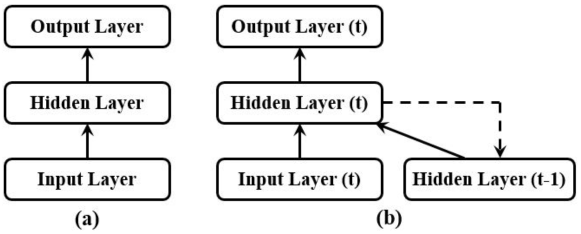

Fig. 1. Feed-forward NN and RNN

Given an input sequence $X = (x_1, \dots\ , x_T)$, RNN will compute the hidden layer vector sequence $H= (h_1, \dots ,h_T)$ and output vector sequence $Y = (y_1, \dots\ , y_T)$ by iterating the following equations from $t = 1$ to $t = T$:

$$h_t \ = \ \sigma(W_{xh}x_t \ + \ W_{hh}h_{t-1} \ + \ b_h) \tag{1}$$

$$y_t \ = \ W_{hy}h_t \ + \ b_y \tag{2}$$

Where W_{xh} represents the weight matrix between the input layer and the hidden layer, and W_{hy} represents the weight matrix between the hidden layer and the output layer. Additionally, W_{hh} denotes the weight matrix of the recurrent connections between two hidden layer states at two consecutive time steps. b_h and b_y represent the hidden bias vector and output bias vector respectively. σ is the hidden layer activation function, an element-wise *sigmoid()* function.

B. LSTM

Though RNN can learn from prior information, research later showed that it could not maintain long-term memory and the prediction accuracy was not very satisfactory. To overcome the problem that RNNs can not explore long range context [3], researchers proposed to combine the RNN architecture with Long Short-Term Memory (LSTM). LSTM was first designed in [9] as a memory cell to decide what to remember, what to forget and what to output. The architecture of a typical LSTM cell is shown in Fig.2.

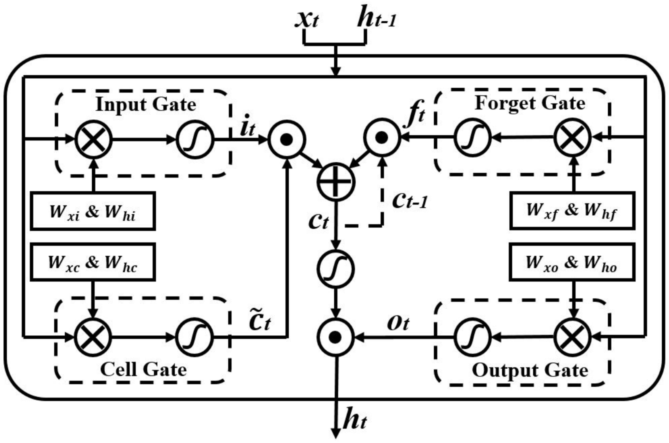

Fig. 2. Standard Long-Short Term Memory Cell [9]

The activation function in each gate can be *sigmoid()* or *tanh()*. $\odot$ and $\oplus$ represent element-wise multiplication and addition respectively. Though there have been a great number of LSTM cell variants for different applications [2][6][12][14][15], the changes to the standard architecture are very small and their effects on the overall prediction accuracy are small enough to be ignored [8]. Replacing each hidden layer in RNN with an LSTM cell, we can get an LSTM-RNN. For the standard LSTM cell shown in Fig.2, the hidden layer vector sequence H is produced by the following functions:

$$i_t \ = \ \sigma(W_{xi}x_t \ + \ W_{hi}h_{t-1} \ + \ b_i) \tag{3}$$

$$f_t \ = \ \sigma(W_{xf}x_t \ + \ W_{hf}h_{t-1} \ + \ b_f) \tag{4}$$

$$\tilde{c_t} \ = \ tanh(W_{xc}x_t \ + \ W_{hc}h_{t-1} \ + \ b_f) \tag{5}$$

$$c_t \ = \ f_t \odot c_{t-1} \ + \ i_t \odot \tilde{c_t} \tag{6}$$

$$o_t \ = \ \sigma(W_{xo}x_t \ + \ W_{ho}h_{t-1} \ + \ b_o) \tag{7}$$

$$h_t \ = \ o_t \odot tanh(c_t) \tag{8}$$

where σ is the *sigmoid()* function, and f, i, $\tilde{c}$, o represent the output vectors of forget gate, input gate, cell gate, and output gate, respectively. W terms denote weight matrices (e.g. W_{xi} is the weight matrix between input gate i and input vector x), and b terms denote bias vectors (e.g. b_i is the bias vector of input gate i).

C. Real-life LSTM-RNN

Many LSTM-RNNs have been proposed in various applications, and they offer great prediction accuracy. Typically, deploying an LSTM-RNN in real-life applications includes two phases: training and inference. This characteristic is the same as other deep learning algorithms. Training is accomplished off-line, and all parameters (weights and bias) of LSTM-RNNs are adjusted to improve prediction accuracy. Once training is over, all parameters will not change any more, and they are stored for model inference. Inference is performed on-line for real-life time-sensitive applications, so the speed of inference computation is what really matters for the deployment of LSTM-RNNs. As a result, in this work we focus on accelerating the inference phase of LSTM-RNNs.

We use the LSTM-RNN (PRETRANS-3L-250H) introduced in [7] as a case study. This LSTM-RNN is designed for speech recognition, and achieves a test set error of 17.7% on the TIMIT phoneme recognition benchmark. This model consists of an input layer, an output layer, and three LSTM layers. The number of all its parameters is about 4.3M. Due to space constraints, please refer to [7] for detailed layer configurations.

III. Optimization Analysis

To optimize the overall performance, we need to consider two principal constraints: computation performance and communication requirements. In this section, we analyze the problems and possible optimizations of LSTM-RNN for these two constraints.

A. Computation Optimization

Our software-based profiling results show that the most time-consuming and resource-consuming part of LSTM-RNN inference is the computation inside each LSTM gate, so we focus on improving the computation performance of LSTM gates. Here we denote the length of the input vector and hidden layer vector at a certain time step as L_i and L_h respectively. Then the total number of operations (floating-point number multiplication and addition) during an LSTM-RNN inference within a single layer can be approximately estimated as: $(L_i + L_h) * L_h * 4 * 2 \ + \ L_h * (8 + 4 + 4)$. Usually, L_i is no larger than L_h, so the computation complexity is $O(L_h^2)$.

Considering the computation resource constraints of hardware platforms and scalability of design, it is natural to take advantage of computation tiling to fit LSTM-RNN models into certain hardware platforms. We tile the loop computation inside each gate, and every tile performs a small portion of the inference computation. The input vectors and weight matrices are also tiled correspondingly. At the inter-tile level, we execute tiles in a pipeline manner to maximize throughput. At the intra-tile level, we unroll the inner-most loops and perform computation in parallel to minimize latency.

The second frequently performed computation during LSTM-RNN inference is the activation functions. Typically, *sigmoid()* and *tanh()* are the two most used activation functions. Unfortunately, the exponentiation and floating-point divisions consume a large amount of hardware resource for implementation. Considering the computation resource constraints of hardware platforms, we choose to make a trade-off between computation accuracy and resource consumption: we replace the activation functions with some simple additions and shifting operations, by using a piecewise linear approximation of nonlinear function (PLAN) approach, which was introduced in [1]. Our experimental results show that, compared with original activation functions, the average error rate brought by our linear approximated activation functions is only 0.63%, which is small enough to be ignored during LSTM-RNN inference.

B. Communication Optimization

Conventionally, parameters and inputs of deep learning models are too large to be stored in limited FPGA on-chip memory. As a result, FPGA-based accelerators usually store the parameters, inputs and outputs in the external DRAM, and load data onto the FPGA for computation during run-time inference. However, the parameters of modern LSTM-RNNs are usually less than 10M [7] [14], which can be stored in FPGA on-chip memory partially or even entirely. This will greatly reduce the long latency of off-chip memory accessing, and improve the communication requirements. Thus, we apply an eclectic approach in our implementation.

There are other optimizations that make the communication speed keep up with the computation modules. With the expressions and analysis in Section III.A, the amount of data transferred between the external DRAM and FPGA chip during an inference is $(L_i + L_h) * L_h * 4 \ + \ L_h * 4$, which indicates that the space complexity for storing these parameters is $O(L_h^2)$.

During the inference phase of LSTM-RNN, the computation between input vectors and parameters needs to transport parameters from the external DRAM to the accelerator. In addition, these matrices or vectors need to be transposed and tiled for computation optimization. As a result, the pattern of data access is pretty irregular, which makes it more difficult to meet the required data bandwidth of the accelerator. To avoid the additional overhead brought by random DDR reads, we reshape the parameter matrices to insure that they can be accessed sequentially for the tiled computation. The reshaping operations are done off-line, and we store the reshaped parameters in the external DRAM before performing model inference, so these reshaping operations do not adversely affect the performance of the accelerator. In the meantime, we implement two input buffer groups and two output buffer groups to work in a ping-pong manner during data accessing, and this scheme can also help to largely improve communication requirements. In addition, we design a data dispatcher to maximize the utilization of data bandwidth between the external DRAM and FPGA on-chip buffers, and this dispatcher is orthogonal to the hardware platform.

IV. Implementation

In this section, we first present an overview of the whole accelerator system. Then we describe implementation of the LSTM accelerator in detail.

A. System Overview

Fig.3 shows an overview of the whole implementation on an FPGA board. This system consists of a single FPGA, with a DDR3 DRAM as the external memory for storing input vectors, output vectors, and parameters of the LSTM-RNN model. The modules on-chip are connected to AXI4 bus or AXI4Lite bus. The data communications among different modules are done on the AXI4 bus, and the AXI4Lite bus is used to transfer commands. The LSTM Accelerator is packaged as a hardware IP. We use MicroBlaze, a RISC processor, to initialize the LSTM Accelerator, measure execution time, and control the data communication between accelerator and DRAM through the AXI4Lite bus. The Data Dispatcher communicates with the AXI4 bus, and uses two FIFOs to communicate with LSTM Accelerator. According to [19], only more IP interfaces added to AXI4 bus can efficiently improve bandwidth nearly linearly. Thus, Data Dispatcher uses multi-IP interfaces to fully utilize the physical data width of the AXI4 bus (512 bits). The UART module transfers the results returned by the LSTM Accelrator to host, and Timer is used to measure the execution time of the LSTM Accelerator.

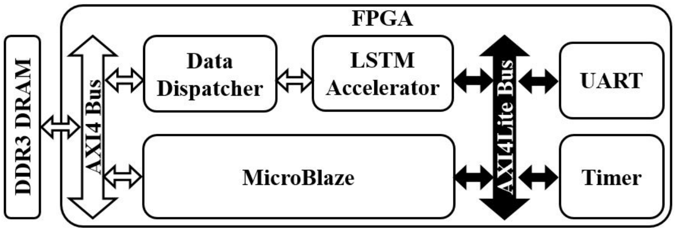

Fig. 3. System Overview

B. LSTM Accelerator

The architecture of the LSTM Accelerator is shown in Fig.4. On-chip data buffers are evenly divided into four groups: two for input data buffering, and two for output data buffering. These buffers work in a ping-pong manner to overlap the time of data communication with inference computation.

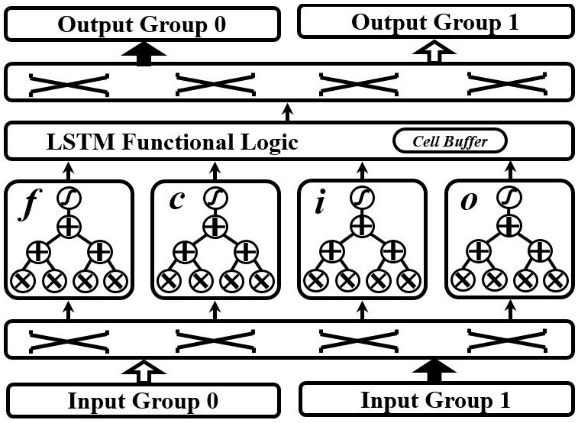

Fig. 4. LSTM Accelerator

The central part of the LSTM Accelerator includes four LSTM gate modules (Forget Gate, Cell Gate, Input Gate and Output Gate) of the LSTM cell, which output the gate vectors of LSTM-RNN. All four gate modules fetch data from input buffer groups through a crossbar. With the fetched data, these four gate modules perform LSTM-RNN inference, and transport results to LSTM Functional Logic to perform the remaining computation (element-wise multiplication and addition of gate vectors, activations, etc.). Then, the final results are loaded to output buffer groups through a crossbar. The current state of the LSTM cell is stored in am on-chip buffer, called *Cell Buffer*.

Inside each gate module, gate vector is calculated in a tiling scheme. Tiled input vectors and the corresponding parameters are transferred into the LSTM gate module in parallel to perform inference. Inside each LSTM gate module, all multiplications between input elements and parameters are performed in parallel. The results are then summed up through a addition tree to minimize latency. The whole architecture is also pipelined to further improve throughput. The outputs are fed into activation nodes to generate the final output vectors of each gate.

V. Evaluation

We introduce the setups of our experiments in this section. Then we present and carefully analyze the experimental results.

A. Experimental Setup

We implement the LSTM-RNN (PRETRANS-3L-250H) discussed in Section II.C as a case study. We finish the design of the LSTM-RNN accelerator with the help of

TABLE I
Performance Comparison with Software Implementations

	CPU-1th	**CPU-16th**	**FPGA**
LSTM #1	2.15 s	0.58 s	0.11 s
LSTM #2	2.87 s	0.77 s	0.15 s
LSTM #3	2.85 s	0.76 s	0.13 s
Total	7.87 s	2.11 s	0.39 s
Speedup	1.00 x	3.73 x	20.18 x

a high-level synthesis design tool, Vivado HLS (v2015.4). Then we export the accelerator as a hardware IP core for implementation. The whole system is designed in Vivado (v2015.4), which performs RTL synthesis and implementation, then generates the final binary file to configure the FPGA.

The FPGA board we use is Xilinx VC707, which has a Xilinx Virtex7-485t FPGA chip on it. The working frequency of the accelerator is set to 150MHz. For performance comparison, we also have a software implementation of LSTM-RNN inference, which runs on an Intel Xeon CPU E5-2430. The working frequency of this CPU is 2.20GHz.

B. Experimental Results

We apply a uniform hardware configuration to all the LSTM layers, and perform LSTM-RNN inference layer by layer on the FPGA. For the software part, we have two versions of software implementations: one runs in 1 thread (*CPU-1th*), the other runs in 16 threads (*CPU-16th*) using OpenMP. Both software implementations are compiled by gcc with -O3 optimization option. The performance comparison between our accelerator and software implementations is shown in Table I. Table I lists the execution time of each layer and overall performance for an input sequence with a length of 1K (typical value for real-life applications). As Table I shows, our FPGA implementation (*FPGA*) is 20.18x faster than *CPU-1th*, and also achieves a speedup of 5.41x over *CPU-16th.* Considering the run-time power of our FPGA board (19.63W, measured by a power monitor), our implementation outperforms *CPU-1th* 1∼2 orders of magnitude in energy-efficiency (GFLOP/J, giga floating operations per joule).

To show the benefits of our proposed framework, we compare an existing FPGA-based LSTM-RNN accelerator design [5](denoted as *Ref.*) with ours in Table II. To make a fair comparison, we listed the FPGA chip, run-time frequency, total number of operations in each model, data precision, and overall performance in detail. However, the detailed resource utilization is not reported in [5], so we omit the comparison of overall performance density. From Table II, we can see that the overall performance of our implementation is 15.45x better than the implemented design in [5], and 1.92x better than their predicted performance.

TABLE II
Comparison with Previous Implementation

	Ref.[5]	**Ours**
FPGA chip	Zynq 7020	Virtex7-vx485t
Frequency	142 MHz	150 MHz
Model Size	0.48 MOP	2.76 MOP
Precision	fixed-16	float-32
Performance	0.47 GOP/S^a 3.78 GOP/S^b	7.26 GFLOP/S

^{a}implemented
^{b}predicted

TABLE III
Resource Utilization

Resource	**BRAM**	**DSP**	**FF**	**LUT**
Used	112	1176	181634	189871
Total	2060	2800	607200	303600
Utilization	5.44%	42.00%	29.91%	62.54%

TABLE IV
Resource Utilization (Memory Optimized)

Resource	**BRAM**	**DSP**	**FF**	**LUT**
Used	1072	1176	182646	198280
Total	2060	2800	607200	303600
Utilization	52.04%	42.00%	30.08%	65.31%

The on-chip resource utilization shown in Table III is reported by Vivado (v2015.4) after implementation. Since fixed-point computing units can achieve better performance and utilize less resource [19], the overall performance can even be better if we use fixed-point computing units instead of floating-point computing units. However, this will definitely bring some errors to the predictions. Research on LSTM-RNNs is still on-going, and we have not found any research which shows that LSTM-RNNs have strong robustness when data precision changes. As a result, we use floating-point numbers in our implementations. We leave the inference accuracy test of fixed-point implementations to future work, and we can potentially use fixed-point computing units in future hardware designs.

It is worth noticing that we do not use much BRAM in our implementation, so we consider storing a small portion of parameters on-chip using BRAMs, as mentioned in Section III.B. In this way, we can utilize the on-chip memory more and further optimize the overall performance of the accelerator. We test this scheme on the first layer only, and our experimental results show that this approach can achieve a further overall speedup of about 1.47x. The resource utilization is shown in Table IV, and this strategy can be an option for our future design.

VI. Related Work

Much previous work focuses on accelerating the training phase or inference phase of standard RNNs. Since LSTM-RNN is an advanced version of the standard RNN, the optimization strategies may be similar, and may need to be carefully discussed. The work in [18] [11] [10] are representative. Work in [18] focuses on implementing a RNN-based MUD (multiuser detection) for CDMA, and work in [11] focuses on accelerating a new RNN training scheme on FPGAs. In [10], Li et al. realize a training framework for an RNN-based language model. These three designs all used fixed-point data, and each achieves fairly good performance. However, the RNN models that they implement do not perform well enough in prediction accuracy, which prevents them from being applied to real-life applications.

Since LSTM-RNN has been an emerging architecture in recent years, few FPGA-based accelerators were studied or proposed for it. We find the designs in [17] and [5] to be representational work. Work in [17] focuses on replacing the LSTM training algorithm with simultaneous perturbation stochastic approximation (SPSA), which is more suitable for FPGA implementation. But our work focuses on accelerating the inference phase to deploy LSTM-RNN in real-life applications. In [5], Chang et al. propose an FPGA-based accelerator for a 2-layer LSTM-RNN. The data format they chose is 16bit fixed-point, and their accelerator explores coarse-grained computation parallelism during LSTM-RNN inference. Compared with their work, our implementation uses floating-point data, explores both computation and communication optimizations, and achieves better performance.

VII. Conclusions and Future Work

In this paper, we propose an FPGA-based accelerator for LSTM-RNN. We optimize both computation performance and communication requirements, and implement an accelerator on a Xilinx VC707 FPGA board. The experimental results show that our design achieves significant speedup over software implementations, and it outperforms previous LSTM-RNN accelerators as well.

There are several opportunities for further research, such as storing parameters in carefully quantized fixed-point data to reduce resource utilization and improve overall performance. Additionally, we can try to extend this acceleration framework to some other variants of LSTM-RNNs.

VIII. Acknowledgment

This work is supported by the National Natural Science Foundation of China (No.61572045) and the NSFC International collaboration project (No.61520106004). We also would like to thank Ningyi Xu at Microsoft Research Asia for many inspiring discussions.

References

[1] Hesham Amin, K Memy Curtis, and Barrie R Hayes-Gill. Piecewise linear approximation applied to nonlinear function of a neural network. In *Circuits, Devices and Systems, IEE Proceedings-*, volume 144, pages 313–317. IET, 1997.

[2] Justin Bayer, Daan Wierstra, Julian Togelius, and Jürgen Schmidhuber. Evolving memory cell structures for sequence learning. In *Artificial Neural Networks–ICANN 2009*, pages 755–764. Springer, 2009.

[3] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *Neural Networks, IEEE Transactions on*, 5(2):157–166, 1994.

[4] Wonmin Byeon, Marcus Liwicki, and Thomas M Breuel. Scene analysis by mid-level attribute learning using 2d lstm networks and an application to web-image tagging. *Pattern Recognition Letters*, 63:23–29, 2015.

[5] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on fpga. *arXiv preprint arXiv:1511.05552*, 2015.

[6] Patrick Doetsch, Michal Kozielski, and Hermann Ney. Fast and robust training of recurrent neural networks for offline handwriting recognition. In *Frontiers in Handwriting Recognition (ICFHR), 2014 14th International Conference on*, pages 279–284. IEEE, 2014.

[7] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[8] Klaus Greff, Rupesh Kumar Srivastava, Jan Koutník, Bas R Steunebrink, and Jürgen Schmidhuber. Lstm: A search space odyssey. *arXiv preprint arXiv:1503.04069*, 2015.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[10] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. Fpga acceleration of recurrent neural network based language model. In *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*, pages 111–118. IEEE, 2015.

[11] Yutaka Maeda and Masatoshi Wakamura. Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation. *Neural Networks, IEEE Transactions on*, 16(6):1664–1672, 2005.

[12] Sebastian Otte, Marcus Liwicki, and Andreas Zell. Dynamic cortex memory: Enhancing recurrent neural networks for gradient-based sequence learning. In *Artificial Neural Networks and Machine Learning–ICANN 2014*, pages 1–8. Springer, 2014.

[13] Qinru Qiu, Qing Wu, Martin Bishop, Robinson E Pino, and Richard W Linderman. A parallel neuromorphic text recognition system and its implementation on a heterogeneous high-performance computing cluster. *Computers, IEEE Transactions on*, 62(5):886–899, 2013.

[14] Hasim Sak, Andrew Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Proceedings of the Annual Conference of International Speech Communication Association (INTERSPEECH)*, 2014.

[15] Jürgen Schmidhuber, Daan Wierstra, Matteo Gagliolo, and Faustino Gomez. Training recurrent networks by evolino. *Neural computation*, 19(3):757–779, 2007.

[16] Ilya Sutskever, Oriol Vinyals, and Quoc VV Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[17] Rok Tavcar, Joze Dedic, Drago Bokal, and Andrej Zemva. Transforming the lstm training algorithm for efficient fpga-based adaptive control of nonlinear dynamic systems. *INFORMACIJE MIDEM-JOURNAL OF MICROELECTRONICS ELECTRONIC COMPONENTS AND MATERIALS*, 43(2):131–138, 2013.

[18] WG Teich, A Engelhart, W Schlecker, R Gessler, and HJ Pfleiderer. Towards an efficient hardware implementation of recurrent neural network based multiuser detection. In *Spread Spectrum Techniques and Applications, 2000 IEEE Sixth International Symposium on*, volume 2, pages 662–665. IEEE, 2000.

[19] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM, 2015.