RECURRENT NEURAL NETWORKS HARDWARE IMPLEMENTATION ON FPGA

Andre Xian Ming Chang, Berin Martini & Eugenio Culurciello
Department of Electrical and Computer Engineering
Purdue University, West Lafayette, IN 47907, USA
{amingcha,berin,euge}@purdue.edu

ABSTRACT

Recurrent Neural Networks (RNNs) have the ability to retain memory and learn data sequences, and are a recent breakthrough of machine learning. Due to the recurrent nature of RNNs, it is sometimes hard to parallelize all its computations on conventional hardware. CPUs do not currently offer large parallelism, while GPUs offer limited parallelism due to branching in RNN models. In this paper we present a hardware implementation of Long-Short Term Memory (LSTM) recurrent network on the programmable logic Zynq 7020 FPGA from Xilinx. We implemented a RNN with 2 layers and 128 hidden units in hardware and it has been tested using a character level language model. The implementation is more than 21× faster than the ARM CPU embedded on the Zynq 7020 FPGA. This work can potentially evolve to a RNN co-processor for future mobile devices.

1 INTRODUCTION

The phenomena of the universe may be represented with different dimensions and variables, but one dimension is always present in all of the universe: *time*. Things that happen now may be caused by what has happened in the past, and it may not make sense to analyze the present without accounting for the past.

A Neural Network, or NN, is a generic architecture used in machine learning that can map different types of information. Given an input, a trained NN can give the desired output. However, NNs do not take into account the time dimension, in the sense that they cannot give an output based on a sequence of inputs. Recurrent Neural Networks, or RNNs, address this issue by adding feedback to standard neural networks. Thus, previous outputs are taken into account for the prediction of the next output. RNNs are becoming an increasingly popular way to learn sequences of data (Zaremba et al., 2014; Graves, 2013), and it has been shown to be successful in various applications, such as speech recognition (Graves et al., 2013), machine translation (Sutskever et al., 2014) and scene analysis (Byeon et al., 2015). A combination of a Convolutional Neural Network (CNN) with a RNN can lead to fascinating results such as image caption generation (Vinyals et al., 2014; Mao et al., 2014; Fang et al., 2014).

Due to the recurrent nature of RNNs, it is sometimes hard to parallelize all its computations on conventional hardware. General purposes CPUs do not currently offer large parallelism, while small RNN models does not get full benefit from GPUs. Thus, an optimized hardware architecture is necessary for executing RNNs models on embedded systems.

Long Short Term Memory, or LSTM, is a specific RNN architecture that implements a learned memory controller for avoiding vanishing or exploding gradients (Bengio et al., 1994). The purpose of this paper is to present a LSTM hardware module implemented on the Zynq 7020 FPGA from Xilinx. Figure 1 shows an overview of the system. As proof of concept, the hardware was tested with a character level language model made with 2 LSTM layers and 128 hidden units. The next following sections present the background for LSTM, the implementation details of the hardware and driver software, the experimental setup and the obtained results.

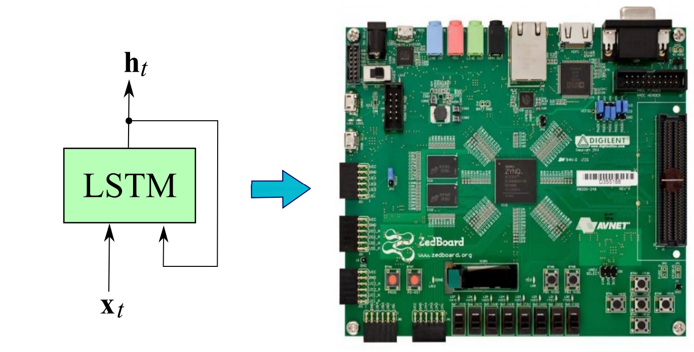

Figure 1: The LSTM hardware was implemented using a Zedboard Zynq ZC7020.

2 LSTM Background

One main feature of RNNs are that they can learn from previous information. But the question is how far should a model remember, and what to remember. Standard RNN can retain and use recent past information (Schmidhuber, 2015). But it fails to learn long-term dependencies. The problem is that RNNs do not have control of when to remember, when to forget and when to use stored information (Bengio et al., 1994). This is where LSTM comes into play. LSTM is an RNN architecture that adds memory controllers to decide when to remember, forget and output. This allows the model to learn long-term dependencies (Hochreiter & Schmidhuber, 1997).

There are some variations on the LSTM architecture. One variant is the LSTM with peephole introduced by Gers et al. (2000). In this variation, the cell memory influences the input, forget and output gates. Conceptually, the model peeps into the memory cell before deciding whether to memorize or forget. In Cho et al. (2014), input and forget gate are merged together into one gate. There are many other variations such as the ones presented in Sak et al. (2014) and Otte et al. (2014). All those variations have similar performance as shown in Greff et al. (2015).

The LSTM hardware module that was implemented focuses on the LSTM version that does not have peepholes, which is shown in figure 2. This is the vanilla LSTM (Graves & Schmidhuber, 2005), which is characterized by the following equations:

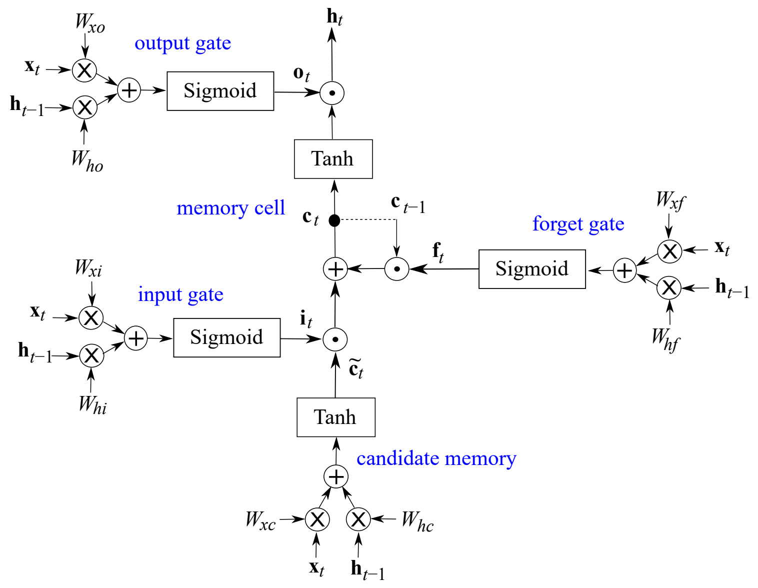

Figure 2: The vanilla LSTM architecture that was implemented in hardware. $\otimes$ represents matrix-vector multiplication and $\odot$ is element-wise multiplication.

$$\mathbf{i}_t = \sigma(W_{xi}\mathbf{x}_t + W_{hi}\mathbf{h}_{t-1} + \mathbf{b}_i) \quad (1)$$
$$\mathbf{f}_t = \sigma(W_{xf}\mathbf{x}_t + W_{hf}\mathbf{h}_{t-1} + \mathbf{b}_f) \quad (2)$$
$$\mathbf{o}_t = \sigma(W_{xo}\mathbf{x}_t + W_{ho}\mathbf{h}_{t-1} + \mathbf{b}_o) \quad (3)$$
$$\tilde{\mathbf{c}}_t = \tanh(W_{xc}\mathbf{x}_t + W_{hc}\mathbf{h}_{t-1} + \mathbf{b}_c) \quad (4)$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t \quad (5)$$
$$\mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t) \quad (6)$$

where σ is the logistic sigmoid function, $\odot$ is element wise multiplication, $\mathbf{x}$ is the input vector of the layer, W is the model parameters, $\mathbf{c}$ is memory cell activation, $\tilde{\mathbf{c}}_t$ is the candidate memory cell gate, $\mathbf{h}$ is the layer output vector. The subscript $t-1$ means results from the previous time step. The $\mathbf{i}$, $\mathbf{f}$ and $\mathbf{o}$ are respectively input, forget and output gate. Conceptually, these gates decide when to remember or forget an input sequence, and when to respond with an output. The combination of two matrix-vector multiplications and a non-linear function, $f(W_x\mathbf{x}_t + W_h\mathbf{h}_{t-1} + \mathbf{b})$, extracts information from the *input* and *previous output* vectors. This operation is referred as gate.

One needs to train the model to get the parameters that will give the desired output. In simple terms, training is an iterating process in which training data is fed in and the output is compared with a target. Then the model needs to backpropagate the error derivatives to update new parameters that minimize the error. This cycle repeats until the error is small enough (Bishop, 2006). Models can become fairly complex as more layers and more different functions are added. For the LSTM case, each module has four gates and some element-wise operations. A deep LSTM network would have multiple LSTM modules cascaded in a way that the output of one layer is the input of the following layer.

3 Implementation

3.1 Hardware

The main operations to be implemented in hardware are matrix-vector multiplications and non-linear functions (hyperbolic tangent and logistic sigmoid). For this design, the number format of choice is Q8.8 fixed point. The matrix-vector multiplication is computed by a Multiply ACcumulate (MAC) unit, which takes two streams: vector stream and weight matrix row stream. The same vector stream is multiplied and accumulated with each weight matrix row to produce an output vector with same size of the weight's height. The MAC is reset after computing each output element to avoid accumulating previous matrix rows computations. The bias $\mathbf{b}$ can be added in the multiply accumulate by adding the bias vector to the last column of the weight matrix and adding an extra vector element set to unity. This way there is no need to add extra input ports for the bias nor add extra pre-configuration step to the MAC unit. The results from the MAC units are added together. The adder's output goes to an element wise non-linear function, which is implemented with linear mapping.

The non-linear function is segmented into lines $y = ax + b$, with x limited to a particular range. The values of a, b and x range are stored in configuration registers during the configuration stage. Each line segment is implemented with a MAC unit and a comparator. The MAC multiplies a and x and accumulates with b. The comparison between the input value with the line range decides whether to process the input or pass it to the next line segment module. The non-linear functions were segmented into 13 lines, thus the non-linear module contains 13 pipelined line segment modules. The main building block of the implemented design is the gate module as shown in figure 3.

The implemented module uses Direct Memory Access (DMA) ports to stream data in and out. The DMA ports use valid and ready handshake. Because the DMA ports are independent, the input streams are not synchronized even when the module activates the ports at same the time. Therefore, a stream synchronizing module is needed. The sync block is a buffer that caches some streaming data until all ports are streaming. When the last port starts streaming, the sync block starts to output synchronized streams. This ensures that vector and matrix row elements that goes to MAC units are aligned.

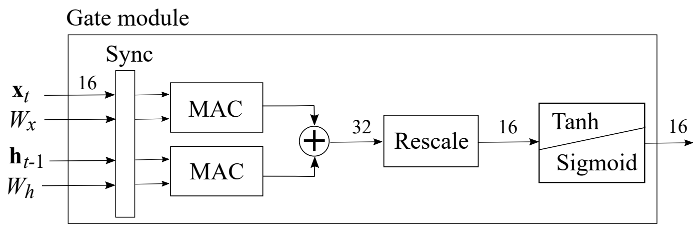

Figure 3: The main hardware module that implements the LSTM gates. The non-linear module can be configured to be a tanh or logistic sigmoid.

The gate module in figure 3 also contains a rescale block that converts 32 bit values to 16 bit values. The MAC units perform 16 bit multiplication that results into 32 bit values. The addition is performed using 32 bit values to preserve accuracy.

All that is left are some element wise operations to calculate $\mathbf{c}_t$ and $\mathbf{h}_t$ in equations 5 and 6. To do this, extra multipliers and adders were added into a separate module shown in figure 4.

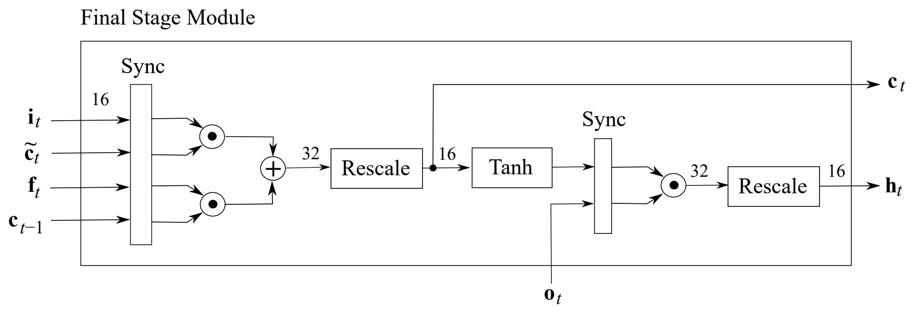

Figure 4: The module that computes the $\mathbf{c}_t$ and $\mathbf{h}_t$ from the results of the gates. $\odot$ is element-wise multiplication.

The LSTM module uses three blocks from figure 3 and one from figure 4. The gates are pre-configured to have a non-linear function (tanh or sigmoid). The LSTM module is shown in figure 5.

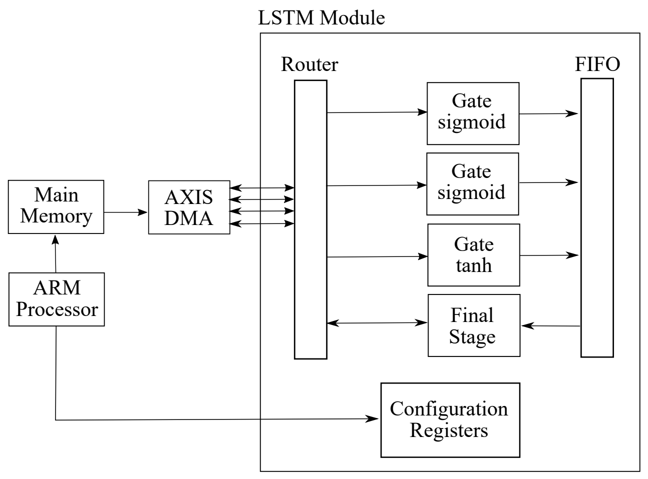

Figure 5: The LSTM module block diagram. It is mainly composed of three gates and one final stage module.

The internal blocks are controlled by a state machine to perform a sequence of operations. The implemented design uses four 32 bit DMA ports. Since the operations are done in 16 bit, each DMA port can transmit two 16 bit streams. The weights W_x and W_h are concatenated in the main memory to exploit this feature. The streams are then routed to different modules depending on the operation to be performed. With this setup, the LSTM computation was separated into three sequential stages:

1. Compute $\mathbf{i}_t$ and $\tilde{\mathbf{c}}_t$.
2. Compute $\mathbf{f}_t$ and $\mathbf{o}_t$.
3. Compute $\mathbf{c}_t$ and $\mathbf{h}_t$.

In the first and second stage, two gate modules (4 MAC units) are running in parallel to generate two internal vectors ($\mathbf{i}_t$, $\tilde{\mathbf{c}}_t$, $\mathbf{f}_t$ and $\mathbf{o}_t$), which are stored into a First In First Out (FIFO) for the next stages. The final stage consumes the FIFO vectors to output the $\mathbf{h}_t$ and $\mathbf{c}_t$ back to main memory. After the final stage, the module waits for new weights and new vectors, which can be for the next layer or next time step. The hardware also implements an extra matrix-vector multiplication to generate the final output. This is only used when the last LSTM layer has finished its computation.

This architecture was implemented on the Zedboard, which contains the Zynq-7000 SOC XC7Z020. The chip contains Dual ARM Cortex-A9 MPCore, which is used for running the LSTM driver C code and timing comparisons. The hardware utilization is shown in table 1. The module runs at $142\,\mathrm{MHz}$ and the total on-chip power is $1.942\,\mathrm{W}$.

Table 1: FPGA hardware resource utilization for Zynq ZC7020.

Components	**Utilization [/]**	**Utilization [%]**
FF	12960	12.18
LUT	7201	13.54
Memory LUT	426	2.45
BRAM	16	11.43
DSP48	50	22.73
BUFG	1	3.12

3.2 Driving Software

The control and testing software was implemented with C code. The software populates the main memory with weight values and input vectors, and it controls the hardware module with a set of configuration registers.

The weight matrix have an extra element containing the bias value in the end of each row. The input vector contains an extra unity value so that the matrix-vector multiplication will only add the last element of the matrix row (bias addition). Usually the input vector $\mathbf{x}$ size can be different from the output vector $\mathbf{h}$ size. Zero padding was used to match both the matrix row size and vector size, which makes stream synchronization easier.

Due to the recurrent nature of LSTM, $\mathbf{c}_t$ and $\mathbf{h}_t$ becomes the $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$ for the next time step. Therefore, the input memory location for $\mathbf{c}_{t-1}$ and $\mathbf{h}_{t-1}$ is the same for the output $\mathbf{c}_t$ and $\mathbf{h}_t$. Each time step $\mathbf{c}$ and $\mathbf{h}$ are overwritten. This is done to minimize the number of memory copies done by the CPU. To implement a multi-layer LSTM, the output of the previous layer $\mathbf{h}_t$ was copied to the $\mathbf{x}_t$ location of the next layer, so that $\mathbf{h}_t$ is preserved in between layers for error measurements. This feature was removed for profiling time. The control software also needs to change the weights for different layers by setting different memory locations in the control registers.

4 Experiments

The training script by Andrej Karpathy of the character level language model was written in Torch7. The code can be downloaded from Github[1]. Additional functions were written to transfer the trained parameters from the Torch7 code to the control software.

[1] `https://github.com/karpathy/char-rnn`

The Torch7 code implements a character level language model, which predicts the next character given a previous character. Character by character, the model generates a text that looks like the training data set, which can be a book or a large internet corpora with more than 2 MB of words. For this experiment, the model was trained on a subset of Shakespeare's work. The model is expected to output Shakespeare look like text.

The Torch7 code implements a 2 layer LSTM with hidden layer size 128 (weight matrix height). The character input and output is a 65 sized vector one-hot encoded. The character that the vector represents is the index of the only unity element. The predicted character from last layer is fed back to input $\mathbf{x}_t$ of first layer for following time step.

For profiling time, the Torch7 code was ran on other embedded platforms to compare the execution time between them. One platform is the Tegra K1 development board, which contains quad-core ARM Cortex-A15 CPU and Kepler GPU 192 Cores. The Tegra's CPU was clocked at maximum frequency of 2320.5 MHz. The GPU was clocked at maximum of 852 MHz. The GPU memory was running at 102 MHz.

Another platform used is the Odroid XU4, which has the Exynos5422 with four high performance Cortex-A15 cores and four low power Cortex-A7 cores (ARM big.LITTLE technology). The low power Cortex-A7 cores was clocked at 1400 MHz and the high performance Cortex-A15 cores was running at 2000 MHz.

The C code LSTM implementation was ran on Zedboard's dual ARM Cortex-A9 processor clocked at 667 MHz. Finally, the hardware was ran on Zedboard's FPGA clocked at 142 MHz.

5 Results

The text generated by sampling 1000 characters (timestep $t = 1$ to 1000) from the LSTM hardware module is shown in figure 6. The result shows that the LSTM model was able to generate personage dialog, just like in one of Shakespeare's book.

```
ezWhan I have still the soul of thee
That I may be the sun to the state,
That we may be the bear the state to see,
That is the man that should be so far off the world.

KING EDWARD IV:
Why, then I see thee to the common sons
That we may be a countering thee there were
The sea for the most cause to the common sons,
That we may be the boy of the state,
And then the world that was the state to thee,
That is the sea for the most contrary.

KING RICHARD II:
Then we shall be a surper in the seas
Of the statue of my sons and therefore with the statue
To the sea of men with the storesy.

DUKE VICENTIO:
I cannot say the prince that hath a sun
To the prince and the season of the state,
And then the world that was the state to thee,
That is the sea for the most cause to the common sons,
That we may be the sun to the state to thee,
That is the man that should be so far off the world.

QUEEN MARGARET:
And then the world that was the state to thee,
That is the man that should be so far off the worl
```

Figure 6: The output text from the LSTM hardware. The model predict each next character based on the previous characters. The model only gets the first character (seed) and generates the entire text character by character.

Figure 7 shows the timing results. One can observe that the implemented hardware LSTM was significantly faster than other platforms, even running at lower clock frequency of 142 MHz (Zynq ZC7020 CPU uses 667 MHz). Scaling the implemented design by replicating the number of LSTM modules running in parallel will provide faster speed up. Using 8 LSTM cells in parallel can be $16\times$ faster than Exynos5422. Figure 8 shows the expected speed up, assuming the data throughput is high enough to handle the parallel processing. Figure 9 shows the projected FPGA resources utilization in the Zynq ZC7020. It is possible to fit 8 LSTM cells, for this particular device. Given the necessary resources, the LSTM module can be further replicated to improve performance, since the modules can operate independently.

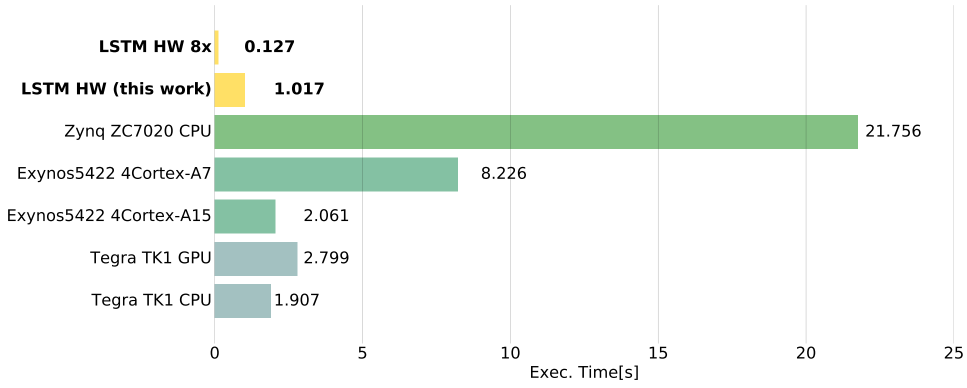

Figure 7: Execution time of feedforward LSTM character level language model on different embedded platforms (the lower the better).

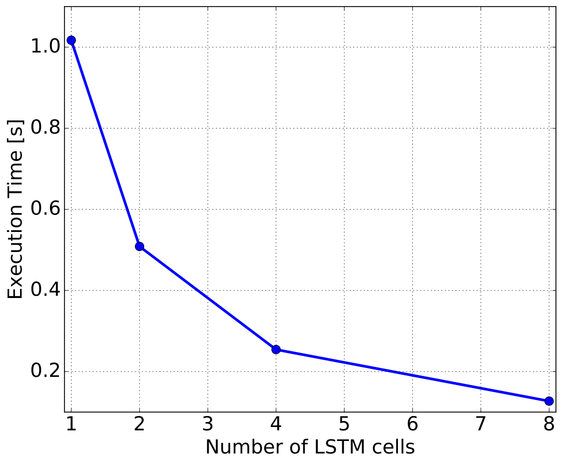

Figure 8: The execution time is projected to decrease with the increase of number of LSTM cells running in parallel. This can lead to significant performance improvement.

The GPU performance was slower because of the following reasons. The model is too small for getting benefit from GPU, since the software needs to do memory copies. This is confirmed by running the same Torch7 code on a MacBook PRO 2016. The CPU of the MacBook PRO 2016 executed the Torch7 code for character level language model in 0.304 s, whereas the MacBook PRO 2016's GPU executed the same test in 0.569 s.

The number of weights of some models can be very large. Even our small model used almost 530 KB of weights. Thus it makes sense to compress those weights into different number formats for a throughput versus accuracy trade off. The use of Q8.8 data format certainly introduces rounding

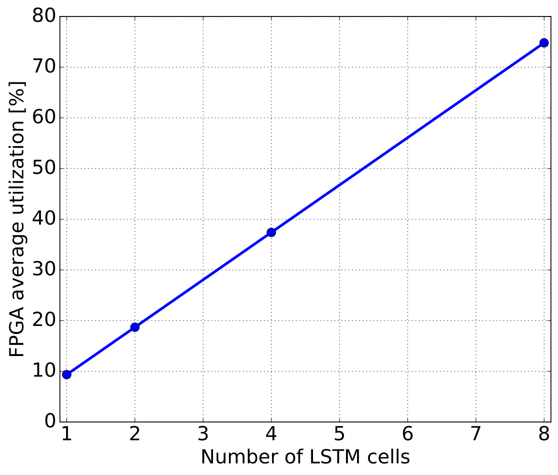

Figure 9: The average FPGA utilization percentage is projected to increase with number of LSTM cells. The prediction shows that it is possible to replicate 8 LSTM cells in the Zynq ZC7020.

errors. Then one may raise the question of how much these errors can propagate to the final output. Comparing the results from the Torch7 code with the LSTM module's output for same $\mathbf{x}_t$ sequence, the average percentage error for the $\mathbf{c}_t$ was 3.9% and for $\mathbf{h}_t$ was 2.8%. Those values are average error of all time steps. The best was 1.3% and the worse was 7.1%. The recurrent nature of LSTM did not accumulate the errors and on average it stabilized at a low percentage.

6 Conclusion

Recurrent Neural Networks have recently gained popularity due to the success from the use of Long Short Term Memory architecture in many applications, such as speech recognition, machine translation, scene analysis and image caption generation.

This work presented a hardware implementation of LSTM module. The hardware successfully produced Shakespeare-like text using a character level model. Furthermore, the implemented hardware showed to be significantly faster than other mobile platforms. This work can potentially evolve to a RNN co-processor for future devices, although further work needs to be done. The main future work is to optimize the design to allow parallel computation of the gates. This involves designing a parallel MAC unit configuration to perform the matrix-vector multiplication.

Acknowledgments

This work is supported by Office of Naval Research (ONR) grants 14PR02106-01 P00004 and MURI N000141010278 and National Council for the Improvement of Higher Education (CAPES) through Brazil scientific Mobility Program (BSMP). We would like to thank Vinayak Gokhale for the discussion on implementation and hardware architecture and also thank Alfredo Canziani, Aysegul Dundar and Jonghoon Jin for the support. We gratefully appreciate the support of NVIDIA Corporation with the donation of GPUs used for this research.

References

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

Bishop, Christopher M. Pattern recognition and machine learning. pp. 225–284, 2006.

Byeon, Wonmin, Liwicki, Marcus, and Breuel, Thomas M. 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.

Cho, Kyunghyun, Van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Fang, Hao, Gupta, Saurabh, Iandola, Forrest, Srivastava, Rupesh, Deng, Li, Dollár, Piotr, Gao, Jianfeng, He, Xiaodong, Mitchell, Margaret, Platt, John, et al. From captions to visual concepts and back. *arXiv preprint arXiv:1411.4952*, 2014.

Gers, Felix, Schmidhuber, Jürgen, et al. Recurrent nets that time and count. In *Neural Networks, 2000. IJCNN 2000, Proceedings of the IEEE-INNS-ENNS International Joint Conference on*, volume 3, pp. 189–194. IEEE, 2000.

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

Graves, Alex. Generating sequences with recurrent neural networks. *arXiv preprint arXiv:1308.0850*, 2013.

Graves, Alex and Schmidhuber, Jürgen. Framewise phoneme classification with bidirectional lstm and other neural network architectures. *Neural Networks*, 18(5):602–610, 2005.

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

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

Mao, Junhua, Xu, Wei, Yang, Yi, Wang, Jiang, and Yuille, Alan L. Explain images with multimodal recurrent neural networks. *arXiv preprint arXiv:1410.1090*, 2014.

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

Sak, Hasim, Senior, Andrew, and Beaufays, Françoise. 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.

Schmidhuber, Jürgen. Deep learning in neural networks: An overview. *Neural Networks*, 61:85–117, 2015.

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

Vinyals, Oriol, Toshev, Alexander, Bengio, Samy, and Erhan, Dumitru. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

Zaremba, Wojciech, Sutskever, Ilya, and Vinyals, Oriol. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014.