Cloud Server Oriented FPGA Accelerator for Long Short-Term Memory Recurrent Neural Networks

Jiasheng Wang\textsuperscript{1,a}, Yu Zhou\textsuperscript{1,b}, Yuyang Sun\textsuperscript{1,c}, Keyang Li\textsuperscript{1,d} and Jun Liu\textsuperscript{1,e}

\textsuperscript{1}Center for Data Science, Beijing University of Posts and Telecommunications, Beijing, China, 100876

E-mail: \textsuperscript{a}wangjiasheng@bupt.edu.cn, \textsuperscript{b}zyicy@bupt.edu.cn, \textsuperscript{c}sunyuyang@bupt.edu.cn, \textsuperscript{d}2015302580280@whu.edu.cn, \textsuperscript{e}liujun@bupt.edu.cn

Abstract. Long Short-Term Memory network (LSTM), which is the most widely used and representative recurrent neural network architecture, plays an important role in language modeling, machine translation, image captioning, etc. However, due to its recurrent nature, general-purpose processors like CPUs and GPUs cannot achieve high parallelism, not to mention their high power consumption. FPGA accelerators can outperform them by flexibility, energy-efficiency and more delicate optimization in each phase of the algorithm. In this paper, we present a cloud-oriented FPGA accelerator for LSTM based on OpenCL. Different from previous works which are designed for embedded systems, our FPGA accelerator performs multiple time series predictions in parallel. We provide a general matrix optimization model to optimize the computation of LSTM in the cloud environments. The performance of our implementation beats both the CPU implementation and other previous hardware implementations. We present and analyze the performance results of our work.

1. Introduction

In recent years, the great progress of inference accuracy of deep learning makes it a hot topic in computer science. Deep Neural Networks (DNN) provides a practical method for computers to learn knowledges from massive labeled data. Recurrent neural network (RNN) is a class of Deep Neural Networks which processes sequential data and performs time series predictions. RNN has been proved to be successful in various time series prediction related applications like language modeling \cite{ref1} \cite{ref2}, machine translation \cite{ref3}, and image captioning \cite{ref4}–\cite{ref6}, etc. A number of RNN models are created to promote the performance of the traditional RNN. Among these models, Long Short-Term Memory network (LSTM) is the most commonly used recurrent neural network architecture. It implements a learned memory controller to improve the prediction accuracy of RNNs.

From model aspect, LSTM focus on how to improve the prediction accuracy while paying little attention to the throughput, respond time or the power consumption of the application. As the LSTM applications are becoming more and more mature, they are increasingly deployed in cloud environments. In this kind of situation, high throughput, short response time, and low power consumption are very important. However, owing to the recurrent nature, LSTM implementation on general-purpose processors like CPUs and GPUs cannot achieve high parallelism and needs high power consumption.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd
To solve this problem, a number of works have been done to utilize FPGA to accelerate LSTM [7] [8] [9]. FPGA accelerators can offer more opportunity and ability to optimize the applications delicately, given that they can perform the computation through the process of pipelining and the hardware structure can be customized to adapt to specific applications when being synthesized. In addition, FPGA accelerators are known to achieve low power consumption. However, most previous hardware implementations are only optimized on embedded platforms. As for the cloud environment, the host server usually receives multiple input streams in parallel. Traditional LSTM algorithms are largely composed of vector-based manipulations, which read vectors sequentially, reducing the overall throughput of the system.

In this paper, we design and implement a cloud server oriented FPGA accelerator of LSTM. The design and implementation are highly optimized for the LSTM applications in cloud environments. We use OpenCL to develop our accelerator. OpenCL is an open, cross-platform parallel programming language that can be used in CPU, GPU and FPGA developments, supported by many vendors [10]. Xilinx SDAccel SDK implements the OpenCL standard based on its high-level synthesis tools.

Our implementation transforms the vector manipulations into matrix operations, providing a general matrix optimization model that optimizes the matrix computation in cloud environments. We process multiple input streams in parallel, ensuring the high throughput of our system. Given that we predict multiple data sequences in a deep pipeline way instead of creating copies of processing hardware, increasing the number of input streams makes no impact on the resource usage. Our work achieves high performance in cloud environments. We measure the throughput by the number of predictions it can produce per second. The peak throughput of our design is 7573462.59 predictions per second, which means that the average execution time of one prediction per input stream is 132.04ns.

The rest of this paper is structured as follows. In Section II, we compare our implementation with related work. Section III discusses the details of our design and implementation. Section IV presents and analyzes the results of our work. In Section V, we conclude and point out future works.

2. Related Work
LSTM was first designed in [11], which has some updated variations [1] [12] [13]. The LSTM with peephole introduced by [14] is a famous variation that the cell memory influences the forget, input and output gates. As a result, the model peeps into the memory cell before deciding whether to memorize or forget. All those variations presented above have similar performance as shown in [15], which surveys on the most commonly used variations, discovering that the variations have few effects on overall prediction accuracy. Our work implements the LSTM version without peepholes [16].

LSTM network has been widely used in such various areas as language modeling, machine translation, image captioning, etc. Most software-implemented LSTM applications are paying too much attention on prediction accuracy, while ignoring the throughput and power consumption, which are vital in real production scenarios. Efforts have rarely been made to implement a hardware accelerator. In [7], Chang et al. implements a two layer LSTM network with 128 neurons each, which is regarded as the earliest actual hardware implementation for LSTM. However, it does not achieve high parallelism. Meanwhile, its performance is not pretty well compared to some latest implementations. Work in [8] optimizes the LSTM network both in computation and communication phase. The authors use pipelining and unrolling to improve the computation performance. Work in [9] focuses on a delicate optimization for LSTM. Then the authors compare their implementation with [7] directly, proving that their work in [9] has a significant speed-up over [7] and the software implementations.

Previous works like [7], [8] and [9] all aim at embedded systems.
However, we could hardly find any LSTM hardware implementations that are optimized for cloud environments. In cloud environments, host servers usually receive multiple input streams in parallel. We make some modifications on traditional LSTM algorithms and implement a cloud environment hardware system. Since our work is similar to [9], we compare our results to those in [9], proving that the on-chip computation performance of our work outperforms the software implementation and hardware implementation in [9].

3. Design and Implementation
In this section, we will first show the whole view of the system architecture of our FPGA accelerator. Secondly, we will demonstrate the data-flow between each component of our system, describing how they interact with each other. Then we will discuss our design in detail, providing a general matrix optimization model to analyze the core principles and methods of our optimization. Finally we will present and illustrate the specific implementation of our accelerator.

3.1. Architecture
Figure 1 shows the system architecture of our work. Since our accelerator is designed for cloud environments, our system architecture is quite different from that in embedded environments. It includes not only the FPGA board but also the host server, which is a universal server equipped with CPUs to receive requests from remote clients and send response to them. The host server controls the data transmission and schedules the computation-intensive tasks for the FPGA board. It transfers data with the FPGA board through PCIe bus. The DDR4 DRAMs persisted on the FPGA board receive input vectors from PCIe bus and save output vectors generated by the FPGA chip, functioning as a connector between FPGA chip and host server. The data transmissions between the DDR4 DRAMs and the FPGA chip are done on the AXI4 bus or the AXI4Lite bus. The three modules in the FPGA chip are our core computation components, which implement three different phases of the LSTM network. The on-chip modules are also connected to AXI4 bus or AXI4Lite bus. Since LSTM network is a recurrent neural network which needs multiple time steps to generate final output vectors, the computation of the three on-chip modules and the data transmission among them need to loop through several time steps. In each time step, output vectors from the previous module function as the input vectors of the next module. What’s more, The output vectors generated by the element-wise computation module from the previous time step are regarded as recurrent state, which are also fed to the matrix-matrix multiplication module as input data. The following sections will discuss the design principles and implementation details of three on-chip modules mentioned above.

3.2. Matrix Optimization Model
The LSTM algorithm can be characterized by the following equations:
Figure 2 shows five matrix operators, which are \( \text{MAC} \) each element in two input matrices would perform a certain type of basic matrix operation. The main differences are matrix operators. That is to say, when performing matrix computations, we put forward a general matrix optimization model that optimizes all these matrix computations by using unrolling and pipelining manners. All of three on-chip modules presented in Figure 1 determine by the size of the weight matrix.

Implementation is largely based on matrix computations, including matrix-matrix multiplication, the matrix addition, the element-wise multiplication, the element-wise vector computation. However, the defective vector-based model mainly lies in its low throughput, because the LSTM network can only process one input stream at a time. Provided that in cloud environments we must handle multiple input streams, the traditional vector-based model is not longer suitable for our accelerator. Our implementation transforms the vector-based model to a matrix-based model, grouping multiple input vectors from various input streams into an input matrix. The LSTM implementation of our work can be characterized by the following equations:

\[
\begin{align*}
  i_t &= \sigma(W_{xi} \ast x_t + W_{hi} \ast h_{t-1} + b_i) \\
  f_t &= \sigma(W_{xf} \ast x_t + W_{hf} \ast h_{t-1} + b_f) \\
  \tilde{c}_t &= \tanh(W_{xc} \ast x_t + W_{hc} \ast h_{t-1} + b_c) \\
  o_t &= \sigma(W_{xo} \ast x_t + W_{ho} \ast h_{t-1} + b_o) \\
  c_t &= f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \\
  h_t &= o_t \odot \tanh(c_t)
\end{align*}
\]

where \( \odot \) is element-wise multiplication, \( \sigma \) is the \textit{sigmoid} function, \( x \) is the input vector, \( W \) is the corresponding weight matrix, \( b \) is the corresponding bias vector, \( c \) is memory cell activation, and \( h \) is the layer output vector. \( i, f, c, o \) represent the input gate, forget gate, candidate memory cell gate, and output gate respectively. The subscript \( t \) denotes the corresponding time step.

(1) (2) (3) (4) (5) (6) indicate that the traditional LSTM algorithm is based on vector manipulation. Previous LSTM hardware accelerators like [7] [8] [9] focus on optimization for this vector-based model, including the matrix-vector multiplication and some kinds of element-wise vector computation. However, the defective vector-based model mainly lies in its low throughput, because the LSTM network can only process one input stream at a time. Provided that in cloud environments we must handle multiple input streams, the traditional vector-based model is not longer suitable for our accelerator. Our implementation transforms the vector-based model to a matrix-based model, grouping multiple input vectors from various input streams into an input matrix. The LSTM implementation of our work can be characterized by the following equations:

\[
\begin{align*}
  I_t &= \sigma(X_t \ast W_{xi} + H_{t-1} \ast W_{hi} + B_i) \\
  F_t &= \sigma(X_t \ast W_{xf} + H_{t-1} \ast W_{hf} + B_f) \\
  \tilde{C}_t &= \tanh(X_t \ast W_{xc} + H_{t-1} \ast W_{hc} + B_c) \\
  O_t &= \sigma(X_t \ast W_{xo} + H_{t-1} \ast W_{ho} + B_o) \\
  C_t &= F_t \odot C_{t-1} + I_t \odot \tilde{C}_t \\
  H_t &= O_t \odot \tanh(C_t)
\end{align*}
\]

As (7) (8) (9) (10) (11) (12) show, after grouping multiple input vectors together, our implementation is largely based on matrix computations, including matrix-matrix multiplication, matrix addition, element-wise matrix multiplication, element-wise matrix \textit{sigmoid} function, and element-wise matrix \textit{tanh} function. In order to optimize the computation phase of our system, we put forward a general matrix optimization model that optimize all these matrix computations by using unrolling and pipelining manners. All of three on-chip modules presented in Figure 1 use this model to implement the LSTM network.

We notice that these different types of matrix computations have similar structures, while the main differences are matrix operators. That is to say, when performing matrix computations, each element in two input matrices would perform a certain type of basic matrix operation. Figure 2 shows five matrix operators, which are \textit{MAC}, +, *, \textit{sigmoid} and \textit{tanh}, referring to the matrix-matrix multiplication, the matrix addition, the element-wise multiplication, the element-wise matrix sigmoid function and the element-wise matrix \textit{tanh} function respectively.

Figure 3 can be considered as the expanded view of Figure 2, presenting the details of our matrix optimization model. \textit{Input Stream Count} denotes the number of input streams which need to be processed, while \textit{LSTM Size} denotes the scale of the LSTM model, which is determined by the size of the weight matrix.
We assume that the size of the input vector is less than the LSTM Size. In Figure 3, Matrix A represents the input matrix in on-chip computation modules, while Matrix B represents the weight matrix in the matrix-matrix multiplication module or the input matrix in the other two modules. When performing matrix computation, we always unroll the loop of iterating the columns of input matrix or the rows of the weight matrix. The PE (Processing Element) in Figure 3 shows the unrolling logic. The number of PEs is LSTM Size. Every PE fetches one value from the column of the input matrix or the corresponding row of the weight matrix to its inside BRAM, where matrix computation with specific operator are processed parallel. If the matrix B is the weight matrix in the matrix-matrix multiplication module, the previous PE also has to write the output value to the next PE, satisfying the need of MAC operator.

All PEs are running in a deep pipeline way. Different PEs are processing different input streams simultaneously, and even a single PE is processing multiple input streams. For instance, it can read one value of input stream 3 from Matrix A, while reading one value of input stream 2 from Matrix B and performing the computation of the matrix operator of input stream 1 in parallel. In conclusion, all PEs are processing different phases of the computation of different
input streams. This pipeline mechanism keeps the hardware working all the time and improves performance significantly.

In order to maximize the pipeline performance, we must ensure the PEs carry no dependencies. Considering that we have $LSTM\, Size$ PEs, which could access at most $LSTM\, Size$ columns of the input matrix and at most $LSTM\, Size$ rows of the weight matrix in parallel, we store the matrices in LUTRAM. The input matrix is partitioned by columns while the weight matrix is partitioned by rows, constructing N-ports memory which can provide N parallel accesses. After the partition, the PEs carry no dependencies and the initiation interval of the pipeline is 1.

The following subsections in this chapter will present the implementation details of the three on-chip computation modules, based on the matrix optimization model.

### 3.3. Matrix-Matrix Multiplication

**Algorithm 1** Matrix-matrix multiplication

**Input:**
- $A = [a_{ij}]_{1 \leq i \leq MAX\,STREAMS, 1 \leq j \leq MAX\,LSTM\,SIZE}$ → The input matrix
- $W = [w_{ij}]_{1 \leq i,j \leq MAX\,LSTM\,SIZE}$ → The weight matrix

**Output:**
- $B = [b_{ij}]_{1 \leq i \leq MAX\,STREAMS, 1 \leq j \leq MAX\,LSTM\,SIZE}$ → The output matrix

1: for $j = 1$ to real$lstm\,size$ do
2: \hspace{1em} #pragma pipeline
3: for $i = 1$ to stream$\,count$ do
4: \hspace{2em} Initialize Temp to zero
5: \hspace{2em} #pragma unroll by factor of MAX$LSTM\,SIZE$
6: \hspace{2em} for $k = 1$ to MAX$LSTM\,SIZE$ do
7: \hspace{3em} Temp = $a_{i,k} \ast w_{k,j}$ + Temp
8: \hspace{2em} end for
9: \hspace{1em} $b_{i,j} = Temp$
10: end for
11: end for

According to the (7) (8) (9) (10) in Section 3.2, we can see that matrix-matrix computation is the first task of the LSTM network. The pseudocode of the matrix-matrix multiplication is presented in Algorithm 1. The input matrix $A$ can be the layer input matrix $X_t$ or the recurrent state matrix $H_{t-1}$ from the previous time step and the weight matrix can be any of the $W_{xi}, W_{xf}, W_{xc}, W_{xo}, W_{hi}, W_{hf}, W_{hc}, W_{ho}$. We define $MAX\,STREAMS$ as an upper bound of the number of rows of the input matrix and the output matrix, deciding the maximum input streams we can process together. We also define $MAX\,LSTM\,SIZE$ as an upper bound of the number of rows of the weight matrix and columns of the input matrix, the weight matrix and the output matrix. LSTM applications can use this model if weight scale is no more than $MAX\,LSTM\,SIZE$. We assume that the size of input vector are less than the LSTM weight size. The matrix-matrix multiplication is performed by three nested loops. real$lstm\,size$ denotes the actual weight size of the LSTM layer and stream$\,count$ denotes the number of input streams. The innermost loop generate one element of the output matrix and the rest two outer loop traverse the output matrix in row-major order.

We completely unroll the innermost loop and pipeline the second loop to achieve high parallelism. We can see that Algorithm 1 is based on the matrix optimization model and the matrix operator is $MAC$. The two inner loops implement the matrix optimization model while
the outermost loop performs the calculation sequentially. In this way, we can predict multiple data sequences in parallel and achieve high throughput.

3.4. Gate Module
After the matrix-matrix multiplication, we can see that the rest of work in (7) (8) (9) (10) are matrix addition, element-wise sigmoid function and element-wise tanh function. Given that we construct the data in the form of matrices rather than vectors, we can keep benefiting from the deep pipeline way. Provided that the sigmoid function and the tanh function are non-linear which are hard to be implemented in FPGA, we use the Polynomial Approximations [9] to implement approximate linear functions. We adopt the same polynomial coefficients which are also used in [9]. The maximum approximation are 1.408X10^-3 for the sigmoid function and 1.21X10^-2 for the tanh function.

Algorithm 2 Gate Module

Input:
\[ A = [a_{ij}]_{1 \leq i \leq \text{MAX}\_\text{STREAMS}},1 \leq j \leq \text{MAX}\_\text{LSTM}\_\text{SIZE} \rightarrow \text{The input matrix 1} \]
\[ B = [b_{ij}]_{1 \leq i \leq \text{MAX}\_\text{STREAMS}},1 \leq j \leq \text{MAX}\_\text{LSTM}\_\text{SIZE} \rightarrow \text{The input matrix 2} \]
\[ \text{bias} = [v_{i}]_{1 \leq i \leq \text{MAX}\_\text{LSTM}\_\text{SIZE}} \rightarrow \text{The bias vector} \]

Output:
\[ C = [c_{ij}]_{1 \leq i \leq \text{MAX}\_\text{STREAMS}},1 \leq j \leq \text{MAX}\_\text{LSTM}\_\text{SIZE} \rightarrow \text{The output matrix} \]

1: #pragma pipeline
2: for i = 1 to stream_count do
3:   #pragma unroll by factor of MAX\_LSTM\_SIZE
4:     for j = 1 to MAX\_LSTM\_SIZE do
5:       Temp = a_{i,j} + b_{i,j} + v_{j}
6:       c_{i,j} = approximate\_linear\_function(Temp)
7:   end for
8: end for

Algorithm 2 presents the details of matrix addition and element-wise activation functions. The \( A \) matrix is the output matrix from \( X_t * W_{xi}, X_t * W_{xf}, X_t * W_{xc}, \) or \( X_t * W_{xi}, \) the \( B \) matrix is the output matrix from \( H_{t-1} * W_{hi}, H_{t-1} * W_{hf}, H_{t-1} * W_{hc}, \) or \( H_{t-1} * W_{ho}, \) bias can be any weight bias vector of \( b_{i}, b_{j}, b_{c}, b_{o}, \) and the \( C \) matrix can be one of \( I_{t}, F_{t}, C_{t}, O_{t} \) respectively. We can see that Algorithm 2 is based on the matrix optimization model, using +, sigmoid and tanh as the matrix operators.

3.5. Element-wise Computation
The rest work of our accelerator is implementing the last two Equations (11) and (12), which mainly consist of element-wise multiplication, matrix addition and element-wise tanh function. The optimization methods are similar to Algorithm 2 and the only difference is the matrix operators. This module is also based on the matrix optimization model, using the *, +, and tanh as matrix operators.

4. Results
We first introduce our experiment environment, including the hardware and software platforms as well as experimental methods. Then we analyze the results of different-sized synthesized networks and various numbers of input streams.
4.1. Experimental Environment
We implement an OpenCL-based FPGA accelerator with the assistance of a high-level synthesis design tool, Xilinx SDAccel v2017.1. Our hardware system consists of a Xilinx Kintex UltraScale KCU1500 Acceleration Development Board and a general host server with Intel(R) Pentium(R) CPU G2030. The host server connects with FPGA board using PCIe Gen2(5 GT/s) with 8 lanes.

We use Keras with TensorFlow backend to train and test the software LSTM network. The same weight parameters and input vectors are used as the input of our hardware implementation. The output vectors of the hardware implementation are compared with the software LSTM network, ensuring that two implementations have the same results within a certain error range because of the non-linear activation functions explained in 3.4.

We test our system with different-sized synthesized networks and various numbers of input streams to examine the performance. We define the size of synthesized network as $N$, representing for the scale of the weight matrix, which can be 4, 8, 16 and 32. The number of input stream is defined as $I \in \{4, 8, 32, 64, 128, 512, 1024\}$. The different results of various $N$ and $I$ are presented and analyzed in the next subsection. We run our application 1000 times for each combination of $N$ and $I$.

4.2. Performance
To show the benefits of our implementation, we compare the results to a similar FPGA accelerator as presented in [9]. The [9] focuses on on-chip computation performance. The [9] also implement a software LSTM layer in python. We compare the average computation time per input stream of our accelerator with the performance of hardware implementation and python implementation in [9]. The comparison result is reported in Table 1. The results prove that the performance of our implementation is better than the [9] in all of the different-sized synthesized networks.

The results presented in Table 1 are our best results when $I = 1024$.

| $N$  | Ours  | Refs.[9] hardware | Refs.[9] python |
|------|-------|-------------------|-----------------|
| 4    | 132.04 $ns$ | 259.12 $ns$       | 65 $\mu s$      |
| 8    | 265.77 $ns$ | 317.52 $ns$       | 72 $\mu s$      |
| 16   | 335.60 $ns$ | 461.336 $ns$      | 96 $\mu s$      |
| 32   | 1.33 $\mu s$ | 1.586 $\mu s$    | 185 $\mu s$     |
Table 2. Clock Frequency

| N   | Ours  | Refs.[9] |
|-----|-------|----------|
| 4   | 300MHz| 158MHz   |
| 8   | 280MHz| 154MHz   |
| 16  | 250MHz| 140MHz   |
| 32  | 190MHz| 102MHz   |

Figure 4 demonstrates that the throughput improves when $I$ grows. Especially when $I$ is small, the throughput grows linearly. It is because of the high parallelism that our matrix model performs better.

As Section 3.2 explains, we process multiple input streams in a deep pipeline way, predicting multiple data sequences in parallel. When $I$ is small, the pipeline queue is always empty, leading to the hardware idleness. For example, if one PE can process four input streams in parallel and there are totally 16 PEs ($N$ is 16), we must process at least 64 input streams together to make the pipeline full, keeping the hardware busy all the time. We achieve the highest throughput when $I = 1024$, and when the $I$ keep growing, the impact on throughput is so little that it can be ignored.

Table 2 presents different clock frequencies when $N$ changes. When $N$ grows, the clock frequency decreases to avoid timing violations and guarantee the accelerator functions properly. We can see that the clock frequencies of our implementation outperform the previous implementation [9]. We do not present the results of different $I$ because $I$ hardly influences the clock compared to $N$, and we can ignore it. In fact, as for each $N$, we synthesize the network which can process at most 1024 input streams, then we can change $I$ from 4 to 1024 to test the performance.

4.3. Resource Utilization and Power Consumption

In Table 3, we can see that BRAM, DSP, FF and LUT all scale up to a 2X factor when $N$ changes from 4 to 32. Our accelerator needs LUT most because of the memory partition as explained in Section 3.2. In order to achieve high performance, our resource utilization is pretty high and we cannot synthesize the network when $N >= 64$. Given that we process multiple input streams in a deep pipeline way instead of unrolling, the increment of $I$ would not consume hardware resource, so that our accelerator can process plenty of input streams together. Table 3 also presents the power consumption of each $N$. As one would expect, the smaller the network is, the less power is consumed.

5. Conclusion and Future Work

We have presented an OpenCL-based FPGA accelerator for long short-term memory recurrent neural networks. We optimize our work in cloud environments, processing multiple input streams in parallel. We provide a general matrix optimization model, transforming the vector-based manipulations to matrix-based operations, which improves the throughput. When the number of input streams is big enough, we obtain high throughput while still retaining the same resource consumption. Additionally, it is also worthwhile to mention that our work still have potential for further research and improvements. First, we only focus on the on-chip computation performance in this implementation while the communication phase need more optimization. Second, we used 32 bit floating point operations instead of fixed point operations, which would exert negative impact on performance and resource consumption. Third, we will add matrix tiling to make our work more scalable.
Table 3. Resource Utilization and Power Consumption

|     | BRAM | DSP | FF | LUT | POWER |
|-----|------|-----|----|-----|-------|
| N=4 | 3%   | 7%  | 9% | 20% | 3.6W  |
| N=8 | 6%   | 14% | 15%| 28% | 4.3W  |
| N=16| 11%  | 28% | 27%| 48% | 5.5W  |
| N=32| 56%  | 55% | 41%| 89% | 7.0W  |

References

[1] H. Sak, A. Senior, and F. Beaufays, “Long short-term memory recurrent neural network architectures for large scale acoustic modeling,” in Proc. of the Annual Conf. of INTERSPEECH, 2014.

[2] 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.

[3] 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.

[4] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” CVPR, 2015.

[5] J. Mao, W. Xu, Y. Yang, J. Wang, and A. L. Yuille, “Explain images with multimodal recurrent neural networks,” arXiv preprint arXiv:1410.1090, 2014.

[6] H. Fang, S. Gupta, F. Iandola, R. Srivastava, L. Deng, P. Doll’ar, J. Gao, X. He, M. Mitchell, J. Platt, et al., “From captions to visual concepts and back,” arXiv preprint arXiv:1411.4952, 2014.

[7] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello, “Recurrent neural networks hardware implementation on FPGA,” arXiv preprint arXiv:1511.05552, 2015.

[8] Yijin Guan, Zhihang Yuan, Guangyu Sun, Jason Cong, “FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks,” 2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC), 2017.

[9] Joao Canas Ferreira, Jose Fonseca, “An FPGA implementation of a long short-term memory neural network,” 2016 International Conference on ReConFigurable Computing and FPGAs (ReConFig), 2016.

[10] Khronos, “The open standard for parallel programming of heterogeneous systems”, 2015.

[11] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” Neural Computation, vol. 9, no. 8, pp. 1735-1780, 1997.

[12] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” EMNLP, 2014.

[13] S. Otte, M. Liwicki, and A. Zell, “Dynamic cortex memory: Enhancing recurrent neural networks for gradient-based sequence learning,” in Artificial Neural Networks and Machine Learning-ICANN 2014, Springer, 2014, pp. 1-8.

[14] F. Gers, J. Schmidhuber, et al., “Recurrent nets that time and count,” in Neural Networks, 2000. IJCNN 2000. Proceedings of the IEEE-INNS-ENNS International Joint Conference on, vol. 3, IEEE, 2000, pp. 189-194.

[15] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “Lstm: A search space odyssey,” arXiv preprint arXiv:1503.04069, 2015.

[16] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional lstm and other neural network architectures,” Neural Networks, 2005, 18(5-6): 602-610.