Proc. of the International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME 2023)
19-20 July 2023, Tenerife, Canary Islands, Spain

Exploring the implementation of LSTM inference on FPGA

1st Manuel L. González
ITCL
Burgos, SPAIN
manuel.gonzalez@itcl.es

2st Randy Lozada
ITCL
Burgos, SPAIN
randy.lozada@itcl.es

3st Jorge Ruiz
ITCL
Burgos, SPAIN
jorge.ruiz@itcl.es

4st E.S. Skibinsky-Gitlin
ITCL
Burgos, SPAIN
erik.skibinsky@itcl.es

5th Ángel M. García-Vico
Dept. of Communication and Control Systems
UNED, Spain
amgarcia@scc.uned.es

6st Javier Sedano
ITCL
Burgos, SPAIN
javier.sedano@itcl.es

7st José R. Villar
University of Oviedo
Oviedo, SPAIN
villarjose@uniovi.es

***Abstract*—Long Short-Term Memory (LSTM) is a widely used deep neural network architecture in sequence modelling and temporal series forecasting. Although the main advantages that LSTM provides in terms of predictive capacity, it is well known in the literature that its performance in inference times and resource consumption is very high, so its application to low-resource environments is an issue. In this article, the performance of a High-Level Synthesis (HLS) implementation of the LSTM inference algorithm is analyzed. Two different LSTM implementations, decoupled-LSTM (d-LSTM) and coupled-LSTM (c-LSTM) are compared on the ZCU104 evaluation board, which contains a Xilinx® Zynq® UltraScale+™ MPSoC device, and their performance and resource usage are evaluated for different neural network models with varying complexity. The results show that d-LSTM performs better than c-LSTM in terms of inference time, but is more resource-intensive. Furthermore, the FPGA implementation of LSTM outperforms other common devices like GPUs and CPUs. The scalability of d-LSTM concerning sequence length is also explored. A positive linear behaviour has been observed in the two lighter models while in the heavier one, the growth is higher than linear. The study highlights the advantages of using FPGA for implementing LSTM inference algorithms, and the importance of exploiting parallelism through libraries like the Basic Linear Algebra Subprograms for optimizing performance.**

***Index Terms*—LSTM inference, FPGA, HLS**

I. Introduction

Recurrent Neural Networks (RNNs) have become crucial in numerous sequential tasks. Among RNN variants, the Long Short-Term Memory (LSTM) has gained popularity due to its capability to handle long-term dependencies and prevent gradients from vanishing or exploding [1]. LSTMs have been successfully used in various applications, such as image captioning, video-to-text, speech recognition, and machine translation, amongst others, [2]–[4]. Commonly, GPUs are used to speed up the inference of LSTM models, however, their high energy consumption, memory and computational demands pose significant challenges for their deployment on embedded systems. As a result, deploying these networks on power-constrained embedded platforms is particularly challenging. Field-Programmable Gate Arrays (FPGAs) are more effective and efficient in exploiting parallelism without batching, resulting in superior energy efficiency. Therefore, it is necessary to have custom hardware designed to accelerate LSTM inference in low-cost, low-power devices.

LSTM implementation on FPGAs has been widely studied. To minimize resource usage and increase execution speed, different approaches have been used to run LSTM neural networks on FPGAs. Several works use pruning techniques to reduce the computational cost of LSTM networks, including weight pruning, block-based sparsity, and bank-balanced sparsity. Works [5]–[7] utilize weight pruning, while [8] proposes bank-balanced sparsity as a novel sparsity pattern that can maintain model accuracy at a high sparsity level while still enabling an efficient FPGA implementation. Other techniques are explored in [9]–[11]. In [9], authors focus on processing-in-memory architectures using resistive random access memory (ReRAM) crossbars. The work proposes the use of ReRAM-based analogue approximate computing to conduct the LSTM-specific element-wise computation and dot-product computation to reduce resources and improve efficiency. In [10] authors propose a low resource utilization FPGA-based LSTM network architecture that achieves low-power and high-speed features through overlapping the timing of operations and pipelining the datapath. Finally, [11] proposes a latency-hiding hardware architecture based on column-wise matrix-vector multiplication to eliminate data dependency and improve the throughput of systems of RNN models.

In this study, High-Level Synthesis (HLS) will be used to assess various implementations of LSTM inference algorithms on FPGA. The evaluation criteria will include resource consumption, execution time, and comparison to CPU and GPU baselines. Additionally, the scalability of the optimal solution will be investigated. The following sections will be organized as follows: Section II will present the different implementations of the LSTM inference algorithm. Section III will describe the implementation of the LSTM algorithm using HLS for execution on FPGA. Finally, Section V will present comparative results in terms of resource consumption and execution time. Finally, the study will conclude with insights and potential areas for future investigation in Section VI.

II. LSTM Inference Algorithm

LSTM is a type of neural network designed to overcome the vanishing gradient problem commonly encountered in traditional RNNs. This problem occurs when gradients become very small or zero during back-propagation, making it difficult for the network to learn long-term dependencies in the input sequence $\{x_t\}_{t=1,\ldots,T}$. LSTM addresses this issue by introducing memory cells and gates that control the flow of information through the network [1]. The three main components of an LSTM network are:

- **Memory cell**, c_t: This is the core component of the LSTM responsible for storing long-term information.
- **Hidden state**, h_t: This is the output of the LSTM used to make predictions or generate new data. The hidden state is computed based on the current input, x_t, the previous hidden state, h_{t-1}, and the current memory cell state, c_t.
- **Gates**: There are three types of gates in LSTM that control the flow of information in and out of the memory cell. These gates modulate the previous hidden state, h_{t-1} and the current input data x_t using a sigmoid function. Their output is a number between 0 and 1 for each element in the memory cell. These three gates are:
  1) **Forget gate**, f_t: This gate determines which information to forget from the memory cell. A value of 0 means the information should be forgotten, and a value of 1 means the information should be retained.
  2) **Input gate**, i_t: This gate determines which new information to add to the memory cell. A value of 0 means that the information should not be added, and a value of 1 means that the information should be added.
  3) **Output gate**, o_t: This gate determines which information to output from the memory cell. The output of this gate is then multiplied by the output of the memory cell to produce the final output.

During the forward pass of the LSTM algorithm, the input sequence is passed through the LSTM layer one element at a time. At each time step, the input is fed into the three gates, and their results are used to update the memory cell state. The updated memory cell state is then passed through the output gate to generate the hidden state, which is the output of the LSTM layer at that time step. The LSTM inference algorithm is commonly implemented as follows:

$$
\begin{aligned}
i_t &= sigmoid\left(W^{(i)} \cdot x_t + U^{(i)} \cdot h_{t-1} + B^{(i)}\right)\\
f_t &= sigmoid\left(W^{(f)} \cdot x_t + U^{(f)} \cdot h_{t-1} + B^{(f)}\right)\\
\tilde{c}_t &= tanh\left(W^{(c)} \cdot x_t + U^{(c)} \cdot h_{t-1} + B^{(c)}\right)\\
o_t &= sigmoid\left(W^{(o)} \cdot x_t + U^{(o)} \cdot h_{t-1} + B^{(o)}\right)\\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t\\
h_t &= o_t \circ tanh\left(c_t\right)
\end{aligned}
$$

Where $W^{(\mu)} \in \mathbb{R}^{N_{features} \times N_{cells}}$ with $\mu = i, f, c, o$ are the weights for each gate or memory cell. $U^{(\mu)} \in \mathbb{R}^{N_{cells} \times N_{cells}}$ are the hidden weights and $B^{(\mu)} \in \mathbb{R}^{N_{cells}}$ are the biases. The operation $\circ$ is the Hadamard product, also known as the element-wise product.

This implementation is named decoupled LSTM inference (d-LSTM). It is decoupled because exists a tuple of weights $\left\{W^{(\mu)}, U^{(\mu)}, B^{(\mu)}\right\}$ for each gate or memory cell that can be calculated independently. Since a decoupled version of the LSTM inference exists, the coupled version can also be implemented. Coupled LSTM inference (c-LSTM) is defined as follows:

$$
\begin{aligned}
y_t &= W \cdot x_t + U \cdot h_{t-1} + B\\
i_t &= sigmoid\left(y_t[0 : N_{cells}]\right)\\
f_t &= sigmoid\left(y_t[N_{cells} : 2N_{cells}]\right)\\
\tilde{c}_t &= tanh\left(y_t[2N_{cells} : 3N_{cells}]\right)\\
o_t &= sigmoid\left(y_t[3N_{cells} : 4N_{cells}]\right)\\
c_t &= f_t \circ c_{t-1} + i_t \circ \tilde{c}_t\\
h_t &= o_t \circ tanh\left(c_t\right)
\end{aligned}
$$

The c-LSTM inference utilizes an intermediate vector $y_t \in \mathbb{R}^{4N_{cells}}$ for calculating gates and memory cells. To achieve this, non-overlapping slides of y_t with dimension N_{cells} are taken. This is denoted in expressions for i_t, f_t, $\tilde{c}_t$ and o_t with $y_t[s_{init} : s_{final}]$ where s_{init} is the initial element of the slide and s_{final} the final. The implementation involves three tensors: W, U, and B. $W^{(\mu)} \in \mathbb{R}^{N_{features} \times 4N_{cells}}$, $U^{(\mu)} \in \mathbb{R}^{N_{cells} \times 4N_{cells}}$ and $B^{(\mu)} \in \mathbb{R}^{4N_{cells}}$. Tensors has no superscripts because they are composed by concatenating $W^{(\mu)}$, $U^{(\mu)}$, and $B^{(\mu)}$ for each μ.

In d-LSTM, each gate and the memory cell are calculated independently using a separate set of weights, which allows for parallel computation. This means that the LSTM cells can be calculated in parallel, which may result in faster inference times. The c-LSTM algorithm utilizes an intermediate vector y_t to calculate the gates and memory cells, which allows for efficient computation of all gates and cells in a single matrix multiplication (MatMul). This approach can further reduce the overall computational cost using efficient hardware implementation of MatMul. In terms of memory storage, two implementations use the same number of parameters.

III. HLS Implementation of LSTM on FPGA

Currently, FPGA technology is becoming increasingly important in the development of applications where specific system performance must be maintained. As a result, it is increasingly common to find Machine Learning solutions implemented with FPGA. The leading manufacturer of FPGAs is AMD® Xilinx®, which has evolved these devices into Zynq® architectures [12]. These consist of a combination of a processing system (PS) composed of several dedicated ARM® processors such as APU (Application Processing Unit), RPU (Real-Time Processing Unit), and GPU (Graphic Processing Unit), external memory (DDR), interconnection peripherals, and programmable logic (PL) or FPGA, all on the same chip. The advantage of using these devices, besides their low power consumption and high data rate is the ability to exploit the reconfigurability of the PL. The PL of an FPGA is composed of various resources that include configurable logic blocks (CLB), interconnect resources, digital signal processing (DSP) and block random-access memory (BRAM). Each CLB consists of multiple Slices. Slices can be configured to implement combinatorial and sequential logic circuits using flip-flops (FF) and lookup tables (LUT).

Conventional hardware description languages (HDL) such as Verilog or Very High-Speed Integrated Circuit HDL (VHDL) are not ideal for implementing complex algorithms, especially in fields like Machine Learning. High-Level Synthesis (HLS) tools offer a solution to this problem by allowing algorithm description in high-level programming languages such as C/C++, which can then be synthesized into a low-level HDL language for programming the PL. However, non-synthesizable data and logic structures in high-level code must be analyzed and rewritten to be implemented on PL. Also, exploiting FPGA architecture parallelism requires using programming paradigms like Producer-Consumer, Streaming Data, and Pipeline [13].

Xilinx® provides the Vitis™ development platform for programming HLS modules. Utilizing HLS in this environment comes with an additional benefit: the ability to employ libraries called Vitis Accelerated Libraries [14]. One of these libraries is Vitis Basic Linear Algebra Subroutines (BLAS), an FPGA-enhanced version of the standard BLAS library [15]. This library delivers a considerable advantage by enabling function optimization for performance through the adjustment of parallelism levels in its operations. To activate this feature, the $LogParEntries$ parameter of the methods must be non-zero. This parameter is calculated as $\log_2(ParEntries)$, where $ParEntries$ represents the number of entries processed concurrently. Note that $ParEntries$ must be a power of 2 to fully benefit from this capability. Certain BLAS methods are valuable for the development of a highly paralleled LSTM Cell, which includes:

- **GEMV**: perform a generalized MatMul plus another vector. The general formula is:

$$GEMV(\alpha, \beta, M, x, y) = \alpha M \cdot x + \beta y$$

Where α, β are constants, M is a two-dimensional matrix and x, y are vectors. Particularly with $\alpha = 1$ and $\beta = 1$, it can be used to compute MatMul plus Bias on both LSTM implementations. For this special case, GEMV will be denoted as $GEMV(M, x, y)$.

- **AXPY**: perform a generalized summation of two vectors:

$$AXPY(\alpha, x, y) = \alpha x + y$$

Particularly with $\alpha = 1$, it can be used to compute vector addition in the LSTM cell state. For this special case, AXPY will be denoted as $AXPY(x, y)$.

- **AXPY***: is a modified version of AXPY to perform a generalized element-wise product of two vectors:

$$AXPY^*(\alpha, x, y) = \alpha x \circ y$$

Particularly with $\alpha = 1$, it can be used to compute Hadamard product in LSTM cell state and hidden vector. For this special case, AXPY* will be denoted as $AXPY^*(x, y)$.

Using these BLAS methods, LSTM inference implementation can be rewritten to be implemented in HLS. In the case of d-LSTM:

$$\begin{aligned}
i_t &= sigmoid\left(GEMV\left(W^{(i)}, x_t, GEMV\left(U^{(i)}, h_{t-1}, B^{(i)}\right)\right)\right)\\
f_t &= sigmoid\left(GEMV\left(W^{(f)}, x_t, GEMV\left(U^{(f)}, h_{t-1}, B^{(f)}\right)\right)\right)\\
\tilde{c}_t &= tanh\left(GEMV\left(W^{(c)}, x_t, GEMV\left(U^{(c)}, h_{t-1}, B^{(c)}\right)\right)\right)\\
o_t &= sigmoid\left(GEMV\left(W^{(o)}, x_t, GEMV\left(U^{(o)}, h_{t-1}, B^{(o)}\right)\right)\right)\\
c_t &= AXPY\left(AXPY^*\left(f_t, c_{t-1}\right), AXPY^*\left(i_t, \tilde{c}_t\right)\right)\\
h_t &= AXPY^*\left(o_t, tanh\left(c_t\right)\right)
\end{aligned}$$

And for c-LSTM:

$$\begin{aligned}
y_t &= GEMV\left(W, x_t, GEMV\left(U, h_{t-1}, B\right)\right)\\
i_t &= sigmoid\left(y_t[0 : N_{cells}]\right)\\
f_t &= sigmoid\left(y_t[N_{cells} : 2N_{cells}]\right)\\
\tilde{c}_t &= tanh\left(y_t[2N_{cells} : 3N_{cells}]\right)\\
o_t &= sigmoid\left(y_t[3N_{cells} : 4N_{cells}]\right)\\
c_t &= AXPY\left(AXPY^*\left(f_t, c_{t-1}\right), AXPY^*\left(i_t, \tilde{c}_t\right)\right)\\
h_t &= AXPY^*\left(o_t, tanh\left(c_t\right)\right)
\end{aligned}$$

Where $sigmoid$ and $tanh$ are HLS functions of the Vitis Math library. GEMV, AXPY and AXPY* methods could be paralleled with a power of two $ParEntries$. For a full parallelization, this parameter must be equal to the size entry vector x for GEMV, AXPY and AXPY*. This means in LSTM inference that both $N_{features}$ and N_{cells} must be power-of-two numbers. Mostly, N_{cells} can be chosen in order to satisfy this requirement. For $N_{features}$ is more difficult because it depends on the data available to feed the model. The Neural Networks proposed in Section IV will fulfil these parallelism conditions. The main difference between d-LSTM and c-LSTM is the number of GEMV operations. d-LSTM has 8 GEMVs and c-LSTM has 2 GEMVs that are four times larger than d-LSTM ones. It is important to notice that if GEMV in d-LSTM has a power-of-two $ParEntries$ the corresponding c-LSTM will have it too. These differences will be tested during the experimentation and results will be exposed in Section V.

IV. Materials and Methods

In this section, different hardware and LSTM neural networks will be exposed to perform the experiments. The hardware used to test the LSTM implementation was the ZCU104 evaluation board which contains a Xilinx® Zynq® UltraScale+™ MPSoC device. In particular, this device has a PL with 460.8k FF, 230.4k LUT, 408 BRAM and 1 728 DSP. Both d-LSTM and c-LSTM will be implemented in ZCU104 and a performance-resource comparison will be exposed in Section V. To demonstrate the advantage of FPGA implementation of LSTM in terms of performance, experimentation over GPUs and CPU will be exposed. The GPUs used were NVIDIA A30 and NVIDIA GeForce GTX 1650. The CPU used was Intel i7-9750H. On these commonly used devices, only d-LSTM will be executed because is the implementation available on the TensorFlow package.

To perform the experimentation over different LSTM inferences different Neural Networks will be executed. Will be three LSTM-Neural Networks with increasing complexity. Their specifications can be shown in Table I. Notice that the main difference between them is the increasing number of LSTM layers with a different number of cells. The topology and input shape of LSTM layers is chosen to be a power of two in order to exploit the BLAS parallelism.

TABLE I
LSTM Neural Networks description

Specifications	***LSTM 1***	***LSTM 2***	***LSTM 3***
LSTM Layers	2	3	4
Dense Layers	2	2	2
LSTM Topology	$[32, 64]$	$[32, 64, 128]$	$[32, 64, 128, 32]$
Dense Topology	$[32, 1]$	$[32, 1]$	$[32, 1]$
Input shape	4	4	4
Sequence length	16	16	16
Parameters	31 681	132 545	150 081

V. Results and Discussion

In this section, performance and resource comparison between the inference of d-LSTM and c-LSTM in the ZCU104 evaluation board will be exposed. Also, a latency comparison between the FPGA implementation of LSTM and another device like GPU and CPU will be shown. Finally, scalability in sequence length will be explored for the best implementation. The inference latency for d-LSTM and c-LSTM in ZCU104 can be shown in Table II. It can be seen that for all LSTM examples in Table I d-LSTM performs better than c-LSTM.

In Table III resources of d-LSTM and c-LSTM are detailed. Higher resource usage implies higher energy consumption and smaller LSTM-Neural Networks implementations. These results show that d-LSTM is more resource-hungry than c-LSTM. d-LSTM uses more DSP, BRAM and FF than c-LSTM for all implementations but c-LSTM uses more LUT than d-LSTM. BRAM and LUT resources could be used as memory storage in FPGA. As both implementations have the same number of parameters to be stored in memory, c-LSTM is using more BRAM and d-LSTM is using more LUT. To compare the HLS implementation of LSTM in ZCU104 concerning other devices, inference latency of FPGA, GPU and CPU technologies are plotted in Fig. 1. These results show that HLS implementation of LSTM in ZCU104 is on average 12.7 times faster than in GPU A30, 19.5 times faster than GPU 1950 and 14.5 times faster than CPU i7-9750H.

TABLE II
LSTM inference latency in ZCU104

	d-LSTM **(ms)**	***c-LSTM*** **(ms)**
LSTM 1	**0.562**	1.399
LSTM 2	**1.402**	2.630
LSTM 3	**2.641**	6.474

TABLE III
LSTM resource usage in ZCU104

	LSTM	DSP	BRAM	LUT	FF
LSTM 1	d-LSTM	1 070	105	**121 592**	212 645
	c-LSTM	**776**	**76**	149 811	**80 013**
LSTM 2	d-LSTM	1 329	164	**157 644**	265 305
	c-LSTM	**1 063**	**123**	197 512	**107 450**
LSTM 3	d-LSTM	1 413	391	**228 184**	373 723
	c-LSTM	**1 253**	**179**	230 242	**128 190**

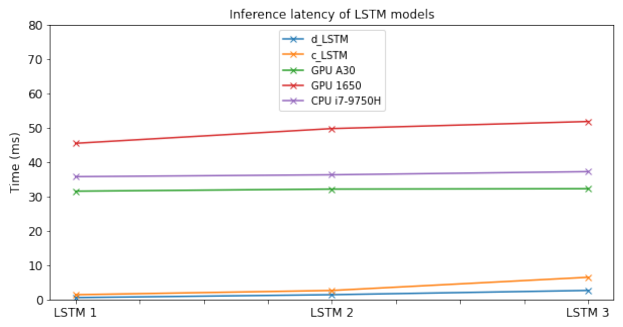

Fig. 1. Inference latency in milliseconds of d-LSTM, c-LSTM executed in ZCU104 and d-LSTM executed on GPU A30, GPU 1650 and CPU i7-9750H. The comparison between devices has been done for the three LSTM models detailed in Table I.

The HLS implementation of d-LSTM has better performance than c-LSTM by using more FPGA resources. The comparison has been done with a constant sequence length of 16 timesteps. To check the scalability of the best-in-performance implementation concerning the sequence length, execution with 8, 16 and 32 timesteps have been performed on ZCU104. The d-LSTM inference of models in Table I for 8, 16 and 32 timesteps are plotted in Fig. 2. The resource usage remains the same for each model independent of the number of time steps. For LSTM 1 $R^2 = 0.986$, $R^2 = 0.999$ for LSTM 2 and $R^2 = 0.943$ for LSTM 3. This means that for lower-complexity models the behaviour of d-LSTM inference latency is linear with the number of time steps. This is not the case with the higher complexity model and further research is needed to explain this behaviour.

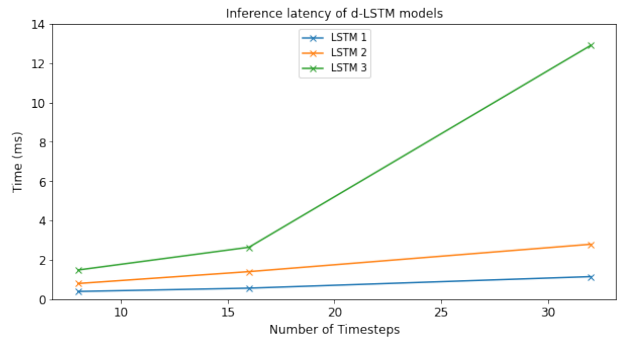

Fig. 2. Inference latency in milliseconds of d-LSTM executed in ZCU104 with 8, 16 and 32 timesteps for the three LSTM models detailed in Table I.

VI. Conclusion and Future Work

This article explores two LSTM inference implementations: d-LSTM and c-LSTM. A comparison in terms of performance and resource utilization on FPGA is realized. The HLS implementation of LSTM inference in FPGA is compared with other commonly used devices such as GPU and CPU. Three LSTM-Neural Networks with increasing complexity are used to perform the experiments. The FPGA used for testing was Zynq UltraScale+ MPSoC ZCU104 dev-board, while NVIDIA A30 and NVIDIA GeForce GTX 1650 GPUs and Intel i7-9750H CPU were used for comparison. The results show that d-LSTM outperforms c-LSTM for all tested models in terms of inference latency. However, d-LSTM is more resource-hungry than c-LSTM, with the former using more DSP, BRAM and FF and the latter using more LUT. The HLS implementation of d-LSTM on ZCU104 is on average 12.7 times faster than in GPU A30, 19.5 times faster than GPU 1950 and 14.5 times faster than CPU i7-9750H. Finally, for d-LSTM, the inference latency has a linear behaviour by the number of timesteps for lower complexity models and a non-linear behaviour for the highest complexity model.

Overall, the study suggests that the HLS implementation of d-LSTM on FPGA is an efficient option for LSTM inference and can outperform other commonly used devices. It highlights the importance of considering both performance and resource utilization when selecting an LSTM implementation. Future work arising from this work is oriented towards understanding the nonlinear behaviour of latency in d-LSTM inference for high-complexity models.

Acknowledgment

This work has been founded by: the Ministry of Science and Innovation under CERVERA Excellence Network CER-20211003 (IBERUS), Missions Science and Innovation MIG-20211008 (INMERBOT), CDTI (Centro para el Desarrollo Tecnológico Industrial) under project CER-20211022, ICE (Junta de Castilla y León) under project CCTT3/20/BU/0002, the Spanish Ministry of Economics and Industry under the grant PID2020-112726RB-I00, the Spanish Ministry of Science, Innovation and Universities with code PID2019-107793GB-I00 and TED2021-131983B-I00, the Principado de Asturias under the grant SV-PA-21-AYUD/2021/50994 and the FEDER funds, ref. 1380734.

References

[1] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: https://direct.mit.edu/neco/article/9/8/1735-1780/6109

[2] N. Srivastava, E. Mansimov, and R. Salakhutdinov, "Unsupervised Learning of Video Representations using LSTMs," Jan. 2016, arXiv:1502.04681 [cs]. [Online]. Available: http://arxiv.org/abs/1502.04681

[3] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H. Yang, and W. J. Dally, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," Feb. 2017, arXiv:1612.00694 [cs]. [Online]. Available: http://arxiv.org/abs/1612.00694

[4] N. Kalchbrenner, E. Elsen, K. Simonyan, S. Noury, N. Casagrande, E. Lockhart, F. Stimberg, A. v. d. Oord, S. Dieleman, and K. Kavukcuoglu, "Efficient Neural Audio Synthesis," Jun. 2018, arXiv:1802.08435 [cs, eess]. [Online]. Available: http://arxiv.org/abs/1802.08435

[5] S. Guo, C. Fang, J. Lin, and Z. Wang, "A Configurable FPGA Accelerator of Bi-LSTM Inference with Structured Sparsity," in *2020 IEEE 33rd International System-on-Chip Conference (SOCC)*. Las Vegas, NV, USA: IEEE, Sep. 2020, pp. 174–179. [Online]. Available: https://ieeexplore.ieee.org/document/9524784/

[6] T. Mealey and T. M. Taha, "Accelerating Inference In Long Short-Term Memory Neural Networks," in *NAECON 2018 - IEEE National Aerospace and Electronics Conference*. Dayton, OH, USA: IEEE, Jul. 2018, pp. 382–390. [Online]. Available: https://ieeexplore.ieee.org/document/8556674/

[7] M. Ferianc, Z. Que, H. Fan, W. Luk, and M. Rodrigues, "Optimizing Bayesian Recurrent Neural Networks on an FPGA-based Accelerator," in *2021 International Conference on Field-Programmable Technology (ICFPT)*. Auckland, New Zealand: IEEE, Dec. 2021, pp. 1–10. [Online]. Available: https://ieeexplore.ieee.org/document/9609847/

[8] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Seaside CA USA: ACM, Feb. 2019, pp. 63–72. [Online]. Available: https://dl.acm.org/doi/10.1145/3289602.3293898

[9] J. Han, H. Liu, M. Wang, Z. Li, and Y. Zhang, "ERA-LSTM: An Efficient ReRAM-Based Architecture for Long Short-Term Memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 6, pp. 1328–1342, Jun. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8944023/

[10] E. Bank-Tavakoli, S. A. Ghasemzadeh, M. Kamal, A. Afzali-Kusha, and M. Pedram, "POLAR: A Pipelined/Overlapped FPGA-Based LSTM Accelerator," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 3, pp. 838–842, Mar. 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8889770/

[11] Z. Que, H. Nakahara, E. Nurvitadhi, H. Fan, C. Zeng, J. Meng, X. Niu, and W. Luk, "Optimizing Reconfigurable Recurrent Neural Networks," in *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. Fayetteville, AR, USA: IEEE, May 2020, pp. 10–18. [Online]. Available: https://ieeexplore.ieee.org/document/9114854/

[12] L. H. Crockett, R. A. Elliot, M. A. Enderwitz, and R. W. Stewart, *The Zynq Book: Embedded Processing with the Arm Cortex-A9 on the Xilinx Zynq-7000 All Programmable Soc*. Glasgow, GBR: Strathclyde Academic Media, 2014.

[13] X. Inc., "Vitis high-level synthesis user guide," *Xilinx Technical Documentation*, vol. Ug1399, pp. 1–657, 2020.

[14] Xilinx, "Vitis libraries," 2022. [Online]. Available: https://xilinx.github.io/Vitis_Libraries

[15] ——, "Vitis blas library," 2022. [Online]. Available: https://xilinx.github.io/Vitis_Libraries/blas/2022.1/index.html