FPGA Implementation of a LSTM Neural Network

José Fonseca*, João Canas Ferreira*, Ivo Timóteo†
FEUP*, University of Cambridge†

***Abstract*—Our work proposes a hardware architecture for a Long Short-Term Memory (LSTM) Neural Network, aiming to outperform software implementations, by exploiting its inherent parallelism. The main design decisions are presented, along with the proposed network architecture. A description of the main building blocks of the network is also presented. The network is synthesized for various sizes and platforms, and the performance results are presented and analyzed. Our synthesized network achieves a 251 times speed-up over a custom-built software network, running on a Desktop computer, and a 14 times speed up over the current state of the art.**

***Keywords*—Neural Networks, Long Short-Term, FPGA, Reconfigurable Hardware, Machine Learning**

I. Introduction

NEURAL Networks are one of the most commonly used techniques in Deep Learning. This particular type of network, a Long Short-Term Memory (LSTM) Network, is a recursive neural network, with memory. These algorithms have been profusely implemented in software, and their practical applications are plentiful. However, the benefits of the inherent parallelism offered by a dedicated hardware platform are not exploited, and there are relatively few implementations of Machine Learning algorithms in these kind of platforms.

II. LSTM Neural Networks

LSTM Networks were originally formulated in [1], and their operation is detailed in Equations 1.

$$\begin{aligned}
\mathbf{z}^{(t)} &= g(\mathbf{W}_z\mathbf{x}^{(t)} + \mathbf{R}_z\mathbf{y}^{(t-1)} + \mathbf{b}_z) \\
\mathbf{i}^{(t)} &= \sigma(\mathbf{W}_i\mathbf{x}^{(t)} + \mathbf{R}_i\mathbf{y}^{(t-1)} + \mathbf{p}_i \odot \mathbf{c}^{(t-1)} + \mathbf{b}_i) \\
\mathbf{f}^{(t)} &= \sigma(\mathbf{W}_f\mathbf{x}^{(t)} + \mathbf{R}_f\mathbf{y}^{(t-1)} + \mathbf{p}_f \odot \mathbf{c}^{(t-1)} + \mathbf{b}_f) \\
\mathbf{o}^{(t)} &= \sigma(\mathbf{W}_o\mathbf{x}^{(t)} + \mathbf{R}_o\mathbf{y}^{(t-1)} + \mathbf{p}_o \odot \mathbf{c}^{(t)} + \mathbf{b}_o) \\
\mathbf{c}^{(t)} &= \mathbf{i}^{(t)} \odot \mathbf{z}^{(t)} + \mathbf{f}^{(t)} \odot \mathbf{c}^{(t-1)} \\
\mathbf{y}^{(t)} &= \mathbf{o}^{(t)} \odot h(\mathbf{z}^{(t)}),
\end{aligned} \tag{1}$$

where $\odot$ is the Hadamard multiplication. A layer has N LSTM neurons and M inputs (i.e. the size of the layer that precedes this).

Hitherto, there is but one actual implementation of an LSTM network in hardware, published recently (March 2016) by Chang et al. [2].

III. Proposed Architecture

A. Network Architecture

Equations 1 suggest that the signals $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$ do not depend on each other – they operate only on the current input vector $\mathbf{x}^{(t)}$ and the previous layer output $\mathbf{y}^{(t-1)}$ – and therefore can be calculated in parallel. Furthermore, we can avoid a naive translation of the Equations 1, which would replicate unnecessary resources (such as elementwise multipliers and activation function calculators) and require more area to save a negligible number of clock cycles, by noting that one of the operands is the output of a $\tanh(\mathbf{x})$ block and the other of a $\sigma(\mathbf{x})$, and they are then multiplied (elementwise, of course) together. Instead of replicating these '$\tanh$-σ-$(\cdot)$wise' structures, we use a *single one* and choose the operand according to the state that the network is currently in. The issue about the elementwise multiplier for $\mathbf{c}^{(t)}$, which does not use the $\tanh$ activation function, can be solved by adding another multiplexer that chooses between the output of the $\tanh(\mathbf{x})$ module or the signal $\mathbf{c}^{(t-1)}$.

The total requirement for DSP slices is

$$4\frac{2N}{K_G} + 2N + N = N\left(\frac{8}{K_G} + 3\right). \tag{2}$$

On the other hand, the number of clock cycles needed to output a result is simply

$$(N \cdot K_G + 6) + 27 = 33 + N \cdot K_G. \tag{3}$$

IV. Results

A. Validation

The functionality of the network was verified against a Python model of an LSTM network that was developed as a reference, both for the forward propagation of the network, as well as for the training algorithm.

The learning problem presented to both the software and hardware network is the **addition** of two binary numbers of 8 bits. The i-th bit of each number is fed to the network as a vector, and the network outputs its prediction of the correct value of the i-th bit of the *result*. After the whole number is processed, the memory cells of the LSTM network are reset and a new addition task can be presented to the network.

B. Synthesis

The proposed network was first synthesized for a Xilinx XC7Z020 SoC, for sizes $N \in \{4, 8, 16, 32\}$, varying the resource sharing parameter K_G, while keeping $M = 2$. For a network size of 32 and $K_G = 8$, the LUT usage exceeded the LUT resources available in the FPGA, so only lower values of K_G were successfully synthesized. To synthesize the design for sizes $N \in \{64, 128\}$ a Virtex-7 VC707 board was used, which has a XC7VX485T (speed grade -2) FPGA core.

TABLE I
TOTAL PROCESSING TIME FOR A SINGLE FORWARD PROPAGATION ON THE XC7Z020

	$K_G = 8$	$K_G = 4$	$K_G = 2$	Python	Speed-up
$N = 4$	N.A.	309.68 ns	259.12 ns	65 μs	$\times$251
$N = 8$	793.46 ns	421.12 ns	317.52 ns	72 μs	$\times$228
$N = 16$	1.497 μs	738.19 ns	461.336 ns	96 μs	$\times$208
$N = 32$	N.D.	1.586 μs	N.A.	185 μs	$\times$117

1) Maximum Frequency: For $N = 4$, since there are only 4 rows to be multiplied, the maximum value of K_G is 4, and hence no synthesis was performed for $K_G = 8$ (N.A.); also, since $K_G = 2$ and $N = 32$, it would use $32(8/2 + 3) = 224$ DSP slices, and that exceeds the 220 slices available in the XC7Z020, so there is no synthesis data (N.D.), as well.

Increasing K_G, the maximum clock speed decreases, and that decrease is steeper for larger values of K_G. This means that there is a critical path in the Matrix-Vector multiplication unit, whose multiplexer becomes increasingly complex. On the other hand, when K_G is the same, smaller networks are faster than larger networks. The fastest design is an $N = 4$ and $K_G = 2$ network, with a clock frequency of 158.228 MHz, and the slowest one is an $N = 32$ and $K_G = 4$ network, clocked at 101.523 MHz. The reference design used for validation in Section IV-A is an $N = 8$ and $K_G = 2$ network clocked at 154.321 MHz, which yields a clock period of 6.48 ns. The maximum clock frequency for the VC707 was 140.854 MHz for both $N = 64$ and $N = 128$.

C. Performance

To evaluate the throughput of the system, a metric was defined based on how many predictions it can produce per second (i.e. produce a new result bit in the output sequence), in **millions**. Hence, we multiply the number of clock cycles yielded by Equation 3 by the equivalent clock period from the synthesis clock report for each network. The Python code was run on an i7-3770k Intel Processor, running at 4.2GHz, with 8GB of RAM.

The hardware network is, at best, $\times$251 faster than the software counterpart, and at worst $\times$117 faster. Inverting the values in Table I, we know how many forward propagations the network can produce, per second. These values are presented in Figure 1. While the $N = 8$ and $K_G = 2$ network is able to perform around 3.15 million predictions per second, the Python model can only output around 14 thousand predictions.

As for the larger-sized networks synthesized in the VC707, the results are also very promising. For network sizes of $N = 64$ and $N = 128$, a complete forward propagation takes 1.14 μs and 2.052 μs respectively, and for both the maximum clock frequency achievable was 140.845 MHz. Since the design [2], for $N = 128$, takes an estimated 29.13 µs, our design yields an improvement of 14$\times$ over it.

V. CONCLUSION

The LSTM Hardware architecture presented surpassed the performance of the custom-built software implementation by 251$\times$, at best, and also the only current hardware implementation by 14$\times$, and solely making use of internal FPGA resources, achieving a higher level of parallelism. The higher levels of parallelism of this work are achieved at the cost of increasing design complexity, which limits its scalability to higher sized networks, unlike the implementation of Chang et al. [2]. On the other hand, the HDL description of this work is parameterized, and is thus very flexible for networks of any size, not requiring a redesign of the system every time a differently sized network is required. Furthermore, making use of internal memory makes it suitable for including an on-chip learning system that can perform training on the network weights.

Given these results, this architecture advances the current state of the art in LSTM Neural Networks hardware implementations, providing the most efficient implementation to date.

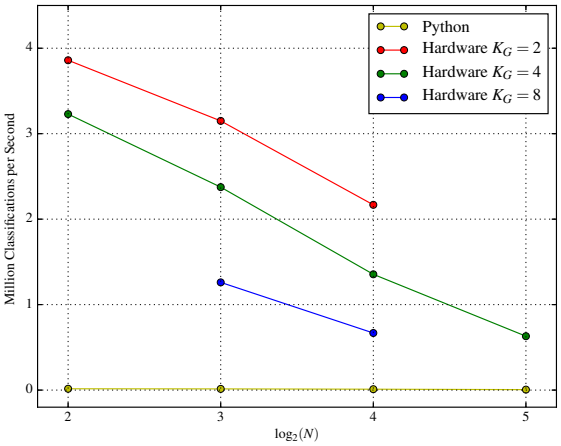

Fig. 1. Millions of classifications per second of each design according to the network size N. The comparison is between the software Python model and 3 networks of different levels of resource sharing K_G.

REFERENCES

[1] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735 – 80, 1997. [Online]. Available: http://dx.doi.org/10.1162/neco.1997.9.8.1735

[2] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *CoRR*, vol. abs/1511.05552, 2015. [Online]. Available: http://arxiv.org/abs/1511.05552