An FPGA Implementation of a Long Short-Term Memory Neural Network

***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 an i7-3770k Desktop computer, proving the benefits of parallel computation for this kind of network.**

***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 network, in which the neuron outputs in a certain time step are also fed as inputs in the next time step, and since it possesses memory, the system can make sense of patterns within data sequences, unlike classical recursive neural networks. 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.

Hardware platforms could achieve a considerable speed-up over software implementations, which would prove useful for high data throughput systems, where the calculation overhead is critical and limits the performance. Furthermore, a hardware implementation can even benefit *offline* Deep Learning tasks by providing the results of a given experiment faster than in a regular CPU, yielding an increase in scientific productivity.

Hitherto, there is only one implementation, the one of [1], but its performance is undermined by the external memory access. Our implementation aims to make use of internal FPGA memory resources, and therefore achieve a higher throughput.

Summarizing, this article begins by explaining what are LSTM networks – and their mathematical details – in Section II, followed by a quick overview of their state of the art in Section III. Section IV outlines the proposed hardware architecture and its main constituent modules, and the performance and synthesis results are reported in Section V. Concluding remarks are given in Section VI.

II. LSTM Neural Networks

LSTM Networks were originally formulated in [2], but their formulation has been incrementally updated in [3] and [4], and the most recent version is [5]. One of the inital proposers of LSTM, Prof. Jürgen Schmidhüber, did a survey on the most common variations of the model [6], which is the reference for this short discussion.

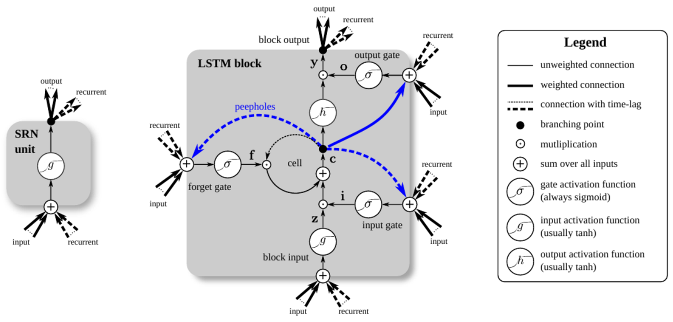

Fig. 1. A complete LSTM neuron, with all the features as described in [5]. Source: [6]

A single LSTM neuron is presented in Figure 1. As can be seen in the picture, we still have the recurrent connections from the regular RNNs, but now there are multiple entry points that control the flow of information through the network. Although omitted from the picture, all the gates are biased, as defined in Equations 1. The role and relevance of the main components can be summarized as follows.

- **Input Gate** – this is the input gate, where the relative importance of each feature of the input vector at time t, $\mathbf{x}^{(t)}$, and the output vector at the previous time step, $\mathbf{y}^{(t-1)}$, are weighed, producing an output $\mathbf{i}^{(t)}$.
- **Block Input Gate** – as the name implies, this gate controls the flow of information from the input gate to the memory cell. It also receives the input vector and the previous output set, producing an output $\mathbf{z}^{(t)}$. The **activation function of this gate can both** the logistic sigmoid, $\frac{\sigma(x)=1}{(1+e^{-x})}$, or the hyperbolic tangent, $\tanh(x)$ but the most common choice is the **hyperbolic tangent**.
- **Forget Gate** – its role is to control the contents of the Memory Cell, either to set or reset them, using the *Hadamard* vector multiplication of its output at time t, $\mathbf{c}^{(t-1)}$. The activation function of this gate is **always sigmoid**, and the resulting signal is $\mathbf{f}^{(t)}$.
- **Output Block Gate** – this gate has a role very similar to that of the Block Input Gate, but now it controls the information flow *out* of the LSTM neuron, namely the activated Memory Cell output. The control signal it produces is $\mathbf{o}^{(t)}$.
- **Memory Cell** – the cornerstone of the LSTM neuron. This is the memory element of the neuron, where the previous state is kept, and updated according to the

dynamics of the gates that connect to it. Also, this is where the peephole connections come from.

- **Output Activation** – the output of the Memory Cell goes through this activation function that, as the gate activation function, is the **hyperbolic tangent**.
- **Peepholes** – direct connections *from* the memory cell to the gates, that allow them to 'peep' at the states of the memory cell. They were added after the initial 1997 formulation, and their absence was proven to have a minimal performance impact [6]. For this reason, they were omitted in this architecture.

The operation of each set of gates of the layer is given by the following set of equations, where vectors are represented by bold, lower-case letters, and matrices are bold, upper-case letters

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

where $\odot$ is the Hadamard multiplication. The i-th element of the previous vectors in bold corresponds to the value of the i-th neuron in the layer, which is a very convenient and compact representation of the whole layer. Furthermore, if the layer has N LSTM neurons and M inputs (i.e. the size of the layer that precedes this), we see that the input weight matrices $\mathbf{W}_*$ have size $N \times M$, and the recurrent weight matrices $\mathbf{R}_*$ are square matrices of size $N \times N$, and that the bias weight vectors $\mathbf{b}_*$ and the vectors $\mathbf{y}^{(t)}$ through $\mathbf{c}^{(t)}$ have size N.

III. Related Work

LSTM Networks are nowadays one of the state-of-the-art algorithms in Deep Learning, and their performance is superior to that of other kinds of RNNs and Hidden Markov Models, both of which are generally used to address the same set of problems where LSTM are employed, namely predicting and producing classification decisions from time-series data. A very comprehensive description of applications can be found in one of the initial authors webpage dedicated to the subject [1].

The uses of LSTM Networks are plentiful: in **Handwriting Recognition** [7], where they surpass HMM-based models for optical character recognition [8]; in **Speech Recognition**, where, for instance, Graves et al. [9], in 2013, set a new record on the TIMT Phoneme Recognition Benchmark, and also in **Large Scale Acoustic Modelling of Speech** [10]. Other uses include **Handwriting Synthesis** [11], **Translation** [12], **Biomedical Applications** such as protein homology detection [13], **Music Analysis and Composition**, such as the transcription of piano music to MIDI [14] and automated composition [15] and improvisation [16], and lastly **Video and Image Analysis** as in [17]–[19].

Hitherto, there is but one actual implementation of an LSTM network in hardware, published recently (March 2016) by Chang et al. [1]. A 2 layer LSTM network (two layers in series) was implemented in FPGA, with 128 neurons each, which, for processing 1000 samples, yielded an execution time of 0.932 s. Assuming that both layers are equal, this yields an approximate execution time of 466 µs per incoming sample (dividing the total execution time by 2×1000), and if the computation time increases linearly with time, an 8 neuron layer would have an execution time of 29.13 µs. Therefore, an 8-neuron network of [1] would be able to perform around 34.3×10^3 forward propagations per second, while this work achieves 487×10^3 forward propagations per second, for that network size. This is because the work in [1] does not have a full level of parallelism when compared to the proposed design of Section IV, and it makes use of external memory to store the weights. On the other hand, the authors use it as a co-processor for the main CPU, and not as a standalone implementation.

IV. Proposed Architecture

The building blocks of the network in Section IV-E are presented in Sections IV-A through IV-D. The number representation system used for this network was a **signed fixed-point** Q6.11 system in two's complement. The are 7 integer bits (one of which is the sign bit), and 11 fractional bits, adding to a total of *18 bits*. The reason we use 18 bits, is to make full use of the DSP48E1 slices available in the FPGAs.

A. Activation Function Calculation

In order to evaluate the transcendental activation functions $\sigma(x)$ and $\tanh(x)$, **Polynomial Approximations** were used, as detailed in [20], since evaluating a polynomial does not have high memory usage requirements (as opposed to Table Methods, for instance) and, if the polynomial degree is sufficiently low, the number of multiplications needed is small enough to not pose a restriction both on resources (now DSP slices, and not memory) and on speed (number of clock cycles needed to output a result).

The error minimization strategy used to find the optimal polynomial was the **Least Maximum Approximation**, where the *maximum* error is *minimized*, making use of Remez's Algorithm, which produces a system of $n+2$ linear equations such as Equation 2

$$\begin{aligned}
p(x_i) - f(x_i) = (-1)^{n+1}\epsilon \Leftrightarrow\\
p_0 + p_1x_i + p_2x_i^2 + \cdots + p_nx_i^n - f(x_i) = (-1)^{n+1}\epsilon \quad (2)
\end{aligned}$$

where $i \in [0, n+1]$. Of course, both functions have horizontal asymptotes, which can be used as the value in the edges. Instead of performing the optimization in the single interval in between the chosen points from where the module evaluates the function as the value of the asymptote, the interval was further split in 4, in order to have lower degree polynomials – simpler to evaluate – at a reduced error cost. The algorithm was run using Python, and targeted second degree polynomials for each interval. The coefficients achieved for the Sigmoid and Hyperbolic Tangent functions are reported in Tables I

[1]http://people.idsia.ch/~juergen/rnn.html

TABLE I
POLYNOMIAL COEFFICIENTS FOR THE SIGMOID APPROXIMATION

p_0	p_1	p_2	Interval
0	0	0	$x < -6$
0.20323428	0.0717631	0.00642858	$-6 \leq x \leq -3$
0.50195831	0.27269294	0.04059181	$-3 \leq x \leq 0$
0.49805785	0.27266221	0.04058115	$0 \leq x \leq 3$
0.7967568	0.07175359	0.00642671	$3 \leq x \leq 6$
1	0	0	$x > 6$

TABLE II
POLYNOMIAL COEFFICIENTS FOR THE $\tanh(x)$ APPROXIMATION

p_0	p_1	p_2	Interval
-1	0	0	$x < -3$
-0.39814608	0.46527859	0.09007576	$-3 \leq x \leq -1$
0.0031444	1.08381219	0.31592922	$-1 \leq x \leq 0$
-0.00349517	1.08538355	-0.31676793	$0 \leq x \leq 1$
0.39878032	0.46509003	0.09013554	$1 \leq x \leq 3$
1	0	0	$x > 3$

and II. The maximum approximation errors were, respectively, 1.408×10^{-3} and 1.21×10^{-2}.

The Verilog model was described, where the correct coefficients are loaded according to the interval where the incoming operand is located. The evaluation of the polynomial was accomplished using the **Horner's Rule**, that is

$$p(x) = p_0 + p_1 x + p_2 x^2 = p_0 + x(p_1 + x p_2) \tag{3}$$

and, therefore, the calculation is simply the procedure of multiplying the operand by a value and adding a constant to the result, repeated two times. Again the internal machine state controls which values are multiplied and added according to the pipeline state. Due to the internal datapath pipeline, each module takes 5 clock cycles to output one result.

B. Matrix-vector Dot Product Calculation

From the Set of Equations 1, we see that the weight matrices $\mathbf{W}_*$ and $\mathbf{R}_*$ are multiplied by the input vector $\mathbf{x}$ and the layer output vector $\mathbf{y}$, respectively. This block implements matrix-vector multiplication to perform those calculations, and its description is parameterized in order to accommodate networks of various sizes, since $\mathbf{W}_*$ has size $N \times M$, and $\mathbf{R}_*$ has size $N \times N$, because x has length M (the number of inputs to the layer), while y has length N (the number of neurons in the layer). If a layer with different parameters is needed, either in terms of number of inputs or number of neurons, we only need to change the respective parameter before the synthesis stage, instead of having to redesign the whole block for that particular size.

The matrix-vector dot product of a matrix A of size $N \times M$ by a vector x of size M, if performed in a linear non-parallel way, can be described in terms of Algorithm 1

This operation has a computational complexity of $O(n^2)$. Each of the i-th components of the output vector y can be calculated **in parallel**, each one only requiring the corresponding i-th line from the matrix. Using this approach, matrix-vector multiplication can now be performed in **linear time**, which is one of the great advantages of custom-tailored hardware solutions.

Algorithm 1 Matrix-vector multiplication of a matrix

```
for $i = 1 : N$ do
    for $j = 1 : M$ do
        $y_i := y_i + A_{ij} \cdot x_j$
    end for
end for
```

Although this solution only requires one multiplication per row of the input matrix (i.e. N multiplications), if the row size is large, we may run out of resources in the FPGA; therefore, some sort of *resource multiplexing* strategy must be used to ensure the flexibility of the solution to accommodate networks of larger dimensions. The solution found for this design was to *share* the multiplication slice among the rows of the matrix: in a direct implementation of Algorithm 1, each multiplication slice is responsible for producing the i-th element of the output vector y (of size N), therefore the final result for the vector would be ready in M clock cycles (i.e. the number of columns); now, defining a parameter K_G, such that

$$K_G = \frac{\text{Number of rows}}{\text{Number of multipliers}}. \tag{4}$$

This parameters represent the number of rows that share the same multiplier. The same multiplier is responsible for producing several i-th elements of the output vector, in consecutive time slots of M clock cycles. For instance, in an 8×2 matrix scenario, with $K_G = 2$, we would have 4 multipliers, and the output vector elements y_0, y_2, y_4 and y_6 would be ready after $M = 2$ clock cycles, and the remaining – y_1, y_3, y_5 and y_7 – are ready after another two clock cycles, that is $2M = 4$ clock cycles after the calculations began.

Figures 2 and 3 depict a diagram of the memory access for the Matrix, and the row multiplication units within the module, respectively, where $K_G = 4$ and for the same matrix and vector sizes as before. Note that in this situation, we would only have 2 multipliers, and the module would be composed of two multiplication units, such as those in Figure 3, that work in parallel. They address a particular column using the signal `colSel`, which is used by the RAM module to output the corresponding column of the matrix (in regard to the input vector, obviously this signal selects only a single position), depicted in Figure 2. The dark shaded part of the memory is used by the first multiplier, and the light shaded one is used by the other, in parallel for a fixed `rowMux`. This signal is produced by the control unit of the module, and essentially operates the left multiplexer and right demultiplexer of Figure 3, that allows choosing the proper position of the weight column and writing to the correct output vector position, respectively. In this example, for `rowMux`=0, the control unit increments `colSel` from 0 to M, and thus evaluates y_0 and y_4. After this, `rowMux` is incremented to 1, and the process repeats until `rowMux` reaches $K_G - 1$. Therefore, we have the correct result vector in a time proportional to $K_G \times M$: a 3 clock cycle overhead is added to the previous estimate, since the memory

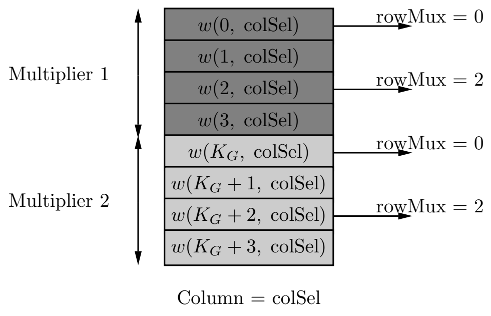

Fig. 2. A column of the matrix that serves as input to the module. The dark shaded part is for the first multiplier, and the light shaded is for the other, in parallel. The rowMux signal addresses the position within each shaded area.

only outputs the appropriate column in the *next clock cycle*, and this module is pipelined both at the input and at the output, in order to increase the maximum clock frequency.

C. Weight storage

The weights are stored in LUTRAM, and for that purpose, they are declared as a matrix of registers, where the access, both in terms of write and read operations, is made to each column. The write/read access can be performed at the same time, since the memory has separate communication ports for input/output, provided that the **addresses do not coincide**.

D. Gate Module

The Gate modules are responsible for producing the internal signal vectors for $\mathbf{z}^{(t)}$, $\mathbf{i}^{(t)}$, $\mathbf{f}^{(t)}$ and $\mathbf{o}^{(t)}$. This way, each Gate module needs to perform three tasks

1) Multiply matrix $\mathbf{W}_*$ by the input vector $\mathbf{x}^{(t)}$
2) Multiply matrix $\mathbf{R}_*$ by the previous layer output vector $\mathbf{y}^{(t-1)}$
3) Sum the bias vector $\mathbf{b}_*$ to the remaining matrix-vector dot product results.

Assuming that the network size N is always larger than the input size M, if we use the matrix-vector dot product units of Section IV-B, the multiplication in task 1 takes approximately $K_G \cdot M$ cycles and the one in task 2 takes $K_G \cdot N$ cycles. This way, tasks 1 and 2 can be performed in parallel, and we can use the extra time that task 2 takes, relative to task 1, to perform task 3, and sum the bias vector to the output of task 1, whose result is ready by that time. The module is triggered by a `beginCalc` input signal that activates the internal state machine, and outputs a `dataReady` signal that informs the network that the calculations have been concluded, and the value at the output is the final result. After validating and simulating this block, and taking into account the internal state-machine and that the internal datapath is pipelined, the exact number of clock cycles this module takes to produce an output is $6 + K_G \cdot N$.

E. 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. For that, we need four Gate Modules working in parallel, each one with its respective weight RAMs for $\mathbf{W}_*$ and $\mathbf{R}_*$, and followed by the respective activation function calculator (detailed in Section IV-A). There are three elementwise multiplications, two for producing signal $\mathbf{c}^{(t)}$ (which can be done in parallel and then summed elementwise) and one for $\mathbf{y}^{(t)}$ (which can be done only after applying the activation function $\mathbf{c}^{(t)}$).

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 $\mathbf{c}^{(t-1)}$. These ideas resulted in the LSTM network design of Figure 5, which is mathematically equivalent to Equations 1.

The two left multiplexers control the operands that are fed to the activation function modules, and the selecting signal is generated by the network's state machine, and its value is incremented after each complete usage of the '$\tanh$-σ-$(\cdot)$wise' structure: this is where the time multiplexing of the structure takes place. Since in state `Sel` $= 1$ the left operand of the elementwise multiplier (the one that preceded the flip-flops in the previous design) is the signal $\mathbf{c}^{(t-1)}$, another multiplexer was added before the elementwise multiplication, to select the $\mathbf{c}^{(t-1)}$ signal in that particular case, and the output from the $\tanh(\mathbf{x})$ block, otherwise.

The registers on the right hand side of Figure 5 are activated by signals generated within the network's state machine that enable the appropriate register, placing the output from the elementwise multiplier in the correct place. The first activated register is the middle one, which keeps the result from the elementwise mutiplication of the $\mathbf{z}^{(t)}$ and $\mathbf{i}^{(t)}$ vectors, then, after a full operation of the '$\tanh$-σ-$(\cdot)$wise' structure, the bottom register saves the other portion of the sum that evaluates to the $\mathbf{c}^{(t)}$ signal. Lastly, the top register saves the network output $\mathbf{y}^{(t)}$, which in the next incoming sample becomes $\mathbf{y}^{(t-1)}$ and is used by the Gate modules in this next batch of calculations.

Now, since there is only a single elementwise multiplier and only two activation function calculators, the total requirement for DSP slices is simply

$$4\frac{2N}{K_G} + 2N + N = N\left(\frac{8}{K_G} + 3\right), \tag{5}$$

where we see that we saved $5N$ multipliers, which for a large value of N can have a decisive impact. In terms of speed performance, although the Gate calculation time remains the same, now the '$\tanh$-σ-$(\cdot)$wise' structure runs for 3 consecutive times. After adjustments to the state machine,

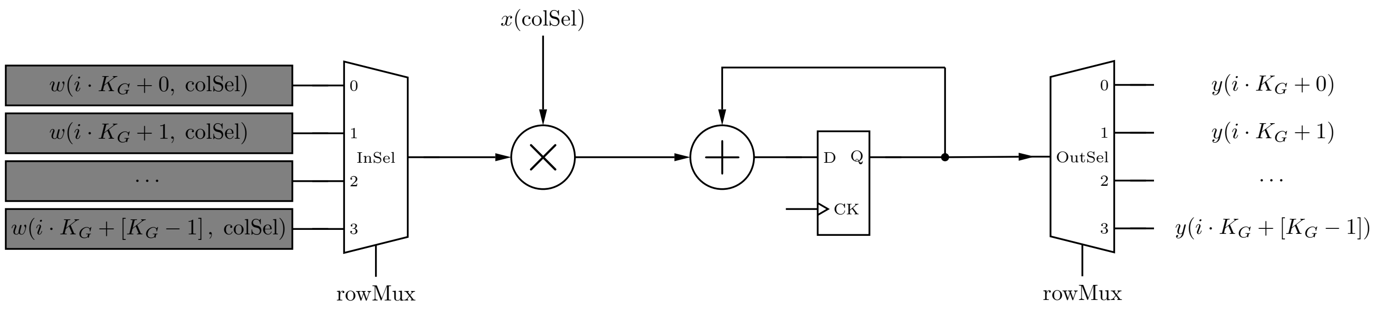
$x(\text{colSel})$
$w(i \cdot K_G + 0,\ \text{colSel})$
$w(i \cdot K_G + 1,\ \text{colSel})$
$\cdots$
$w(i \cdot K_G + [K_G - 1],\ \text{colSel})$
0
1
2
3
InSel
rowMux
D Q
CK
OutSel
rowMux
$y(i \cdot K_G + 0)$
$y(i \cdot K_G + 1)$
$\cdots$
$y(i \cdot K_G + [K_G - 1])$

Fig. 3. The i-th row multiplication unit of the Module, where rowMux and colSel are internal signals produced by the control unit of the module. The flip-flop accumulates the sum, and the output demux selects the appropriate memory position on where to store this value, within the slot attributed to this multiplication, from $i \cdot K_G$ to $i \cdot K_G + [K_G - 1]$

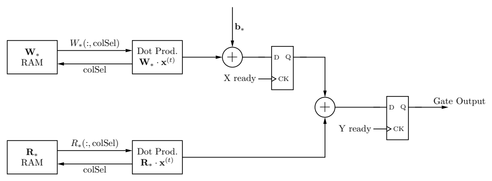

Fig. 4. Diagram of the hardware block that implements the Gate

and accounting for pipelining and synchronization within the datapath, the number of clock cycles needed after the gate module calculations is 27, so the total clock cycles needed to perform a complete forward propagation are

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

which is only 13 clock cycles more than a fully-parallel, naïve architecture. For instance, an $N = 32$ neuron network would require 320 DSP slices, while this new architecture use 160, at the expense of 13 more clock cycles.

V. 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. Python and Numpy were used rather than MATLAB, since the former has higher performance for the same level of code complexity. The model was trained using the SPSA method [21].

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. Even though this seems a rather simple problem, it accounts for all the essential issues at which this network excels: first, this is a *classification* problem in which the Machine Learning algorithm needs to output a prediction based on the input feature vector, and that prediction has to take into account the *history* of predictions and inputs so far, because the current bit is the sum of not only the bits of the two operands, but also the *carry* generated at the last few positions – this is where the memory cells of this special Recurrent Neural Network come into effect.

The software network was trained for 50 epochs. In each epoch, 5000 training sequences were presented, and then tested with 100 sequences, where the prediction error was evaluated for that epoch. The trained weights were loaded into the network and several addition problems were fed to the Network in the testbench, which yielded no errors.

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 the number of inputs set to $M = 2$. 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. This is because of the complexity of the sharing logic for the multipliers of the Matrix-vector Dot Product Module in IV-B. To synthesize the design for sizes $N \in \{64, 128\}$ a Virtex-7 VC707 board was used, which as a XC7VX485T (speed grade -2) FPGA core.

1) Maximum Frequency: The maximum clock frequency that does not cause timing violations of any sort is plotted in Figure 6. 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 $N = 32$, a network such as theses would use $32(8/2 + 3) = 224$ DSP slices, that exceeds the 220 slices available in the XC7Z020, so there is no synthesis data for that value (N.D.), as well.

It is clear that with 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 for higher values of K_G. 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 V-A is an $N = 8$ and $K_G = 2$ network clocked

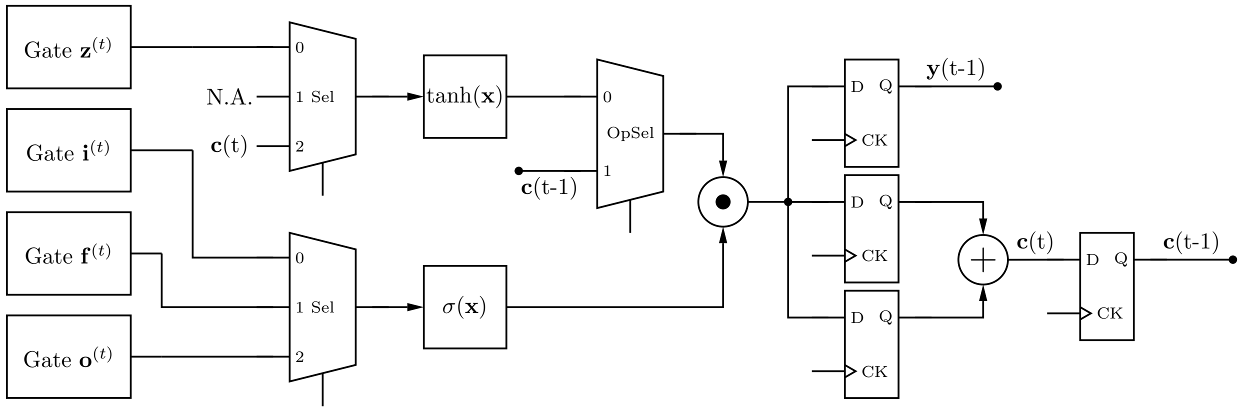

Fig. 5. Block diagram of the hardware LSTM Neural Network

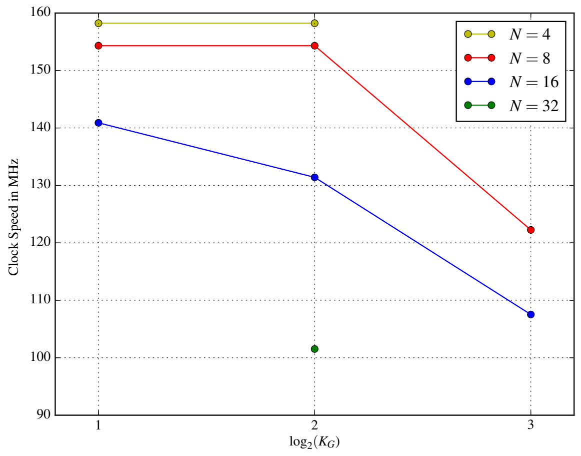

Fig. 6. The maximum achievable clock frequency for feasible network sizes in the XC7Z020

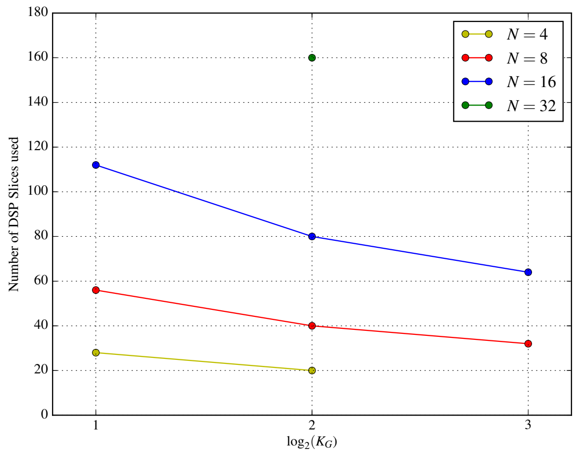

Fig. 7. The number of DSP slices used for several Network Sizes N and resource sharing level K_G

TABLE III
MAXIMUM CLOCK FREQUENCIES FOR THE VC707

	Frequency
$N = 64$	140.854 MHz
$N = 128$	140.854 MHz

TABLE IV
LUT USAGE FOR DIFFERENT N AND K_G IN THE XC7Z020

	$K_G = 2$	$K_G = 4$	$K_G = 8$
$N = 4$	6.87%	6.04%	N.A.
$N = 8$	14.64%	13.03%	14.11%
$N = 16$	28.97%	27.72%	29.85%
$N = 32$	N.A.	91.09%	N.D.

at 154.321 MHz, which yields a clock period of 6.48 ns. The maximum clock achievable for the VC707 networks is reported in Table III

2) DSP Slices Usage: The estimates made in Equation 5 were shown to be accurate, as Figure 7 confirms. The reference network design with $N = 8$ uses 56 DSP slices, which corresponds to 25.45% of the total number of DSP slices available.

The DSP usage for the VC707 was also accurate.

3) Other Resources Usage: The LUT, LUTRAM and Flip-Flop usages are discussed here. In Table IV, the usage of LUTs is reported. We can see that although there is not a clear trend on how the LUT usage varies with increasing K_G, it is clear, and expectable, that the LUT usage increases with the size of the network by an approximate factor of 2, from $N = 2$ to $N = 16$. As for $N = 32$, the usage does not follow this apparent trend, and rises sharply to 91%. For $K_G = 8$ the usage surpasses the maximum amount of LUTs available in the XC7Z020.

As for LUTs, the FF usage also scales according to a $2\times$ factor. In terms of LUTRAM, used to store the network weights, the amount used increases by 2 with increasing values of N, as before, and **does not** depend on K_G. This is because the amount of weights only depends on the network sizes M and N, and not on K_G. Furthermore, unlike LUTs, it scales well with increasing network sizes, and does not pose a limitation on the network size. The usage results for the XC7Z020 are reported in Table V; the results for the VC707 are in Table VI.

TABLE V
Flip-flop and LUTRAM usage for the XC7Z020

	LUTRAM	FF
$N = 4$	0.18%	3.39%
$N = 8$	4.41%	6.7%
$N = 16$	8.83%	13.36%
$N = 32$	17.66%	26.45%

TABLE VI
Flip-flop, LUTRAM and LUT usage for the VC707

	LUTRAM	FF	LUT
$N = 64$	24.22%	9.14%	24.22%
$N = 128$	14.09%	18.3%	41.82%

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**. The metric in [1], as discussed in Section III, is not very informative, since although a system can perform many calculations per second, if those operations are redundant, that metric has no relevant information regarding **how fast** the system can **perform the actual task it was meant to do**, which in this case is a complete Forward Propagation through the network. The prediction time is the time elapsed from the moment a new input vector is applied *to* the moment the LSTM Network produces an output vector. Hence, we multiply the number of clock cycles yielded by Equation 6 by the equivalent clock period from the synthesis clock report of Figure 6. Nevertheless, a comparison on the number of Millions of operations both systems do will be presented further ahead. This result is reported in Table VII, where the calculation time of the Python module's Forward Propagation function is also included. This time was measured using the **timeit**[2] module, which allows the evaluation of the execution time of small pieces of code as well as complete functions with arguments. The Python code was run on a Linux System, powered by an i7-3770k Intel Processor, running at 4.2GHz, with 8GB of RAM.

The performance increase is impressive, even for the slowest of the designs (the $N = 32$ network). The hardware network is, at best, $\times 251$ faster than the software counterpart, and at worst $\times 117$ faster. Also, it is noticeable that increasing the level of resource sharing increases the computation time, since the level of parallelism is lower.

Since the previous values are the time needed for a *single* forward propagation, to know how many forward propagations

[2]https://docs.python.org/3.5/library/timeit.html

TABLE VII
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	$\times 117$

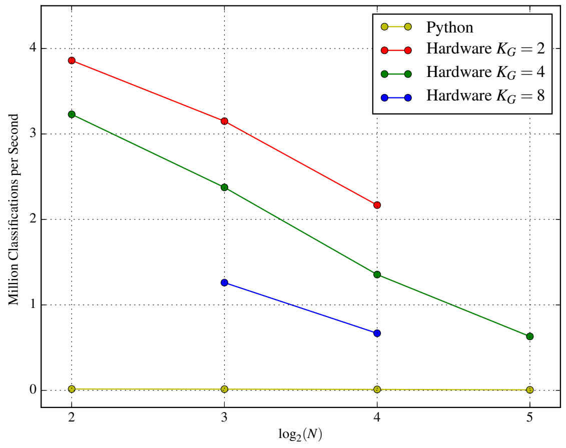

Fig. 8. Millions of classifications per second of each design according to the network size N. The comparison is between the software Python model and 3 networks of different levels of resource sharing K_G.

we can perform *per second*, we only need to invert the previous values. These values are presented in Figure 8. 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, which is a very significant result that proves the relevance of this implementation.

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 [1], for $N = 128$, takes an estimated 29.13 µs (see Section III), our design yields an improvement of $14\times$ over it. In terms of millions of operations per second (Mops), for an $N = 128$ network as the one of [1], our work achieves 4534.8 Mops per second while theirs only achieves 264.4 Mops per second. This is because, as stated in [1], a network of this size performs 132.1×10^3 Mops, and since our work outputs a sample every $\frac{1}{29.13\,\mu s}$, we have that 4534.8 Mops per sec. $= \frac{132.1 \times 10^3 \text{ Mops}}{29.13\,\mu s}$.

1) Power Consumption: Another important metric of the performance of a design is its **power consumption**. These power consumption estimates are post Place&Route, and have a **medium** confidence level. All XC7Z020 designs yielded a constant baseline value for the *static* power consumption of around 120 mW, and the power consumption reported in Figure 9 refers to the *total* power consumptions, i.e. both the baseline static power and the dynamic power consumption. It is clear that the smaller the network is, the less power is consumed, as one would expect. Furthermore, an increasing level of resource sharing yields a substantially lower power consumption figure: this is because less DSP slices are used as K_G increases. Of course, even though the power consumption is lower in that case, that comes at the expense of a lower clock frequency and more clock cycles elapsed per forward propagation.

For the networks synthesized on the VC707, the power consumption figures are reported in Table VIII. Since the

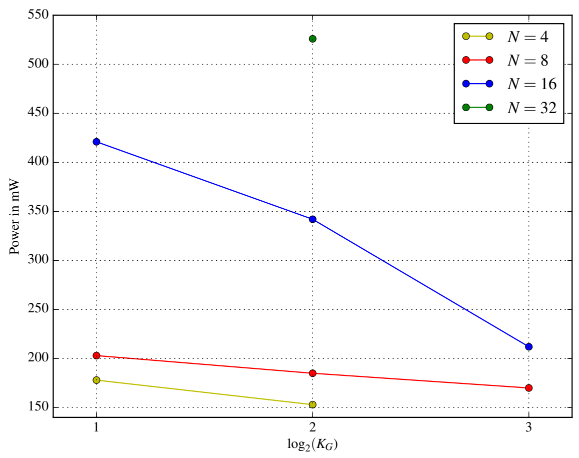

Fig. 9. Power Consumption estimates for several Network Sizes N and resource sharing level K_G

TABLE VIII
POWER CONSUMPTION ESTIMATES FOR SEVERAL NETWORK SIZES N IN THE VC707

	Power
$N = 64$	1.34 W
$N = 128$	1.509 W

device is bigger, and since the network size is larger, the power consumption is higher than before, but still within reasonable levels given the complexity of the system.

All these estimates were provided by Vivado, and have a **medium** confidence level.

VI. 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. [1]. 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.

References

[1] 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

[2] 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

[3] F. Gers, J. Schmidhuber, and F. Cummins, “Learning to forget: continual prediction with lstm,” *Neural Computation*, vol. 12, no. 10, pp. 2451 – 71, 2000. [Online]. Available: http://dx.doi.org/10.1162/089976600300015015

[4] F. Gers and J. Schmidhuber, “Recurrent nets that time and count,” *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks. IJCNN 2000. Neural Computing: New Challenges and Perspectives for the New Millennium*, vol. vol.3, pp. 189 – 94, 2000. [Online]. Available: http://dx.doi.org/10.1109/IJCNN.2000.861302

[5] A. Graves and J. Schmidhuber, “Framewise phoneme classification with bidirectional lstm and other neural network architectures,” *Neural Networks*, pp. 5–6, 2005.

[6] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, “LSTM: A search space odyssey,” *CoRR*, vol. abs/1503.04069, 2015. [Online]. Available: http://arxiv.org/abs/1503.04069

[7] E. Grosicki and H. El Abed, “Icdar 2009 handwriting recognition competition,” July 2009, pp. 1398–1402.

[8] T. M. Breuel, A. Ul-Hasan, M. A. Al-Azawi, and F. Shafait, “High-performance ocr for printed english and fraktur using lstm networks,” in *Proceedings of the 2013 12th International Conference on Document Analysis and Recognition*, ser. ICDAR ’13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 683–687. [Online]. Available: http://dx.doi.org/10.1109/ICDAR.2013.140

[9] A. Graves, A.-R. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” in *2013 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, May 2013, pp. 6645–6649.

[10] H. Sak, A. W. Senior, and F. Beaufays, “Long short-term memory recurrent neural network architectures for large scale acoustic modeling,” pp. 338–342, 2014. [Online]. Available: http://www.isca-speech.org/archive/interspeech_2014/i14_0338.html

[11] A. Graves, “Generating sequences with recurrent neural networks,” *CoRR*, vol. abs/1308.0850, 2013. [Online]. Available: http://arxiv.org/abs/1308.0850

[12] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” vol. 4, no. January, Montreal, QC, Canada, 2014, pp. 3104 – 3112.

[13] S. Hochreiter, M. Heusel, and K. Obermayer, “Fast model-based protein homology detection without alignment,” *BIOINFORMATICS*, vol. 23, no. 14, pp. 1728–1736, JUL 15 2007.

[14] S. Bock and M. Schedl, “Polyphonic piano note transcription with recurrent neural networks,” *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings*, pp. 121 – 124, 2012. [Online]. Available: http://dx.doi.org/10.1109/ICASSP.2012.6287832

[15] A. E. Coca, D. C. Correa, and L. Zhao, “Computer-aided music composition with lstm neural network and chaotic inspiration,” Dallas, TX, United states, 2013. [Online]. Available: http://dx.doi.org/10.1109/IJCNN.2013.6706747

[16] D. Eck and J. Schmidhuber, “Finding temporal structure in music: Blues improvisation with LSTM recurrent networks,” *Neural Networks for Signal Processing - Proceedings of the IEEE Workshop*, vol. 2002-January, pp. 747 – 756, 2002. [Online]. Available: http://dx.doi.org/10.1109/NNSP.2002.1030094

[17] O. Vinyals, A. Toshev, S. Bengio, and D. Erhan, “Show and tell: A neural image caption generator,” *CoRR*, vol. abs/1411.4555, 2014. [Online]. Available: http://arxiv.org/abs/1411.4555

[18] S. Venugopalan, H. Xu, J. Donahue, M. Rohrbach, R. J. Mooney, and K. Saenko, “Translating videos to natural language using deep recurrent neural networks,” *CoRR*, vol. abs/1412.4729, 2014. [Online]. Available: http://arxiv.org/abs/1412.4729

[19] J. Donahue, L. A. Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, “Long-term recurrent convolutional networks for visual recognition and description,” *CoRR*, vol. abs/1411.4389, 2014. [Online]. Available: http://arxiv.org/abs/1411.4389

[20] J.-M. Muller, *Elementary Functions: Algorithms and Implementation*, 2nd ed. Birkhauser, 2005.

[21] J. Spall, “Multivariate stochastic approximation using a simultaneous perturbation gradient approximation,” *IEEE Transactions on Automatic Control*, vol. 37, no. 3, pp. 332–341, Mar 1992.