Hardware Architecture of Bidirectional Long Short-Term Memory Neural Network for Optical Character Recognition

Vladimir Rybalkin and Norbert Wehn
Microelectronic Systems Design Research Group
University of Kaiserslautern, Germany
{rybalkin, wehn}@eit.uni-kl.de

Mohammad Reza Yousefi and Didier Stricker
Augmented Vision Department
German Research Center for Artificial Intelligence (DFKI)
Kaiserslautern, Germany
{yousefi, didier.stricker}@dfki.de

***Abstract*—Optical Character Recognition is conversion of printed or handwritten text images into machine-encoded text. It is a building block of many processes such as machine translation, text-to-speech conversion and text mining. Bidirectional Long Short-Term Memory Neural Networks have shown a superior performance in character recognition with respect to other types of neural networks. In this paper, to the best of our knowledge, we propose the first hardware architecture of Bidirectional Long Short-Term Memory Neural Network with Connectionist Temporal Classification for Optical Character Recognition. Based on the new architecture, we present an FPGA hardware accelerator that achieves 459 times higher throughput than state-of-the-art. Visual recognition is a typical task on mobile platforms that usually use two scenarios either the task runs locally on embedded processor or offloaded to a cloud to be run on high performance machine. We show that computationally intensive visual recognition task benefits from being migrated to our dedicated hardware accelerator and outperforms high-performance CPU in terms of runtime, while consuming less energy than low power systems with negligible loss of recognition accuracy.**

I. Introduction

Many variants of Artificial Neural Networks (ANNs) have been proposed throughout the years. A major source of distinction among ANN architectures is between those having purely acyclic connections, also known as feed-forward networks and those which include cyclic connections referred to as Recurrent Neural Networks (RNNs). The presence of cyclic connections allows for a memory-like functionality that can preserve impacts from the previous inputs to the network within its internal state, such that they can later influence the network output.

Bidirectional Recurrent Neural Networks (BRNNs) were proposed to take into account impact from both the past and the future status of the input signal by presenting the input signal forward and backward to two separate hidden layers both of which are connected to a common output layer. This way, the final network has access to both past and future context within the input signal that is helpful in applications such as character recognition that knowing the coming letters as well as those before improves the accuracy. There are, however, limitations in the standard architecture of RNNs, with the major one being the limited range of accessible context.

One of the most effective solutions to overcome this shortcoming was introduced by Long Short-Term Memory (LSTM) architecture [1], [2] that replaces plain nodes with memory blocks. LSTM memory blocks are composed of a number of gates that can control the store-access functions of the memory state and hence preserving the internal state over longer periods of time. LSTMs have proved successful over a range of sequence learning tasks. One such task that has previously shown superior performance of LSTM networks is character recognition [3], [4], [5], [6] and many others. Unlike conventional Optical Character Recognition (OCR) methods that rely on pre-segmenting the input image down to single characters, LSTM networks with the addition of a forward backward algorithm, known as Connectionist Temporal Classification (CTC) [7], are capable of recognizing textlines without requiring any pre-segmentation.

A direct mapping of Bidirectional Long Short-Term Memory (BLSTM) onto hardware gives very suboptimal results. The main challenge for efficient hardware implementation of RNNs is a high memory bandwidth required to transfer network weights from memory to computational units. RNNs have even more difficulty with memory bandwidth scalability than feed-forward Neural Networks (NNs) due to recurrent connections. Each neuron has to process additional N inputs, where N is a number of neurons in a layer. In case of LSTM memory cells, the requirements are even higher because of four input gates in contrast to a single one in a plain neuron of a standard RNN.

The BLSTM doubles the required bandwidth and requires a duplicate of a hidden layer that processes inputs in opposite direction. Further, the output layer processes outputs from hidden layers only from corresponding columns of an image that introduces additional latency and requires extra memory to store intermediate results. All of the above requires custom solutions. A high memory capacity is required to accommodate the network weights, resulting in a trade-off between different types of memory and bandwidth. One of the most efficient ways to mitigate the memory bandwidth requirement is to exploit inherent capabilities of NN to tolerate precision loss without diminishing recognition accuracy. The flexibility provided by FPGAs allows for architectures that can fully benefit form custom precision.

The elaborate structure of LSTM memory cells including 5 activation functions and multiple multiplicative units implementing gates yield high implementation complexity resulting in high energy consumption and long runtime for software implementations when running on standard computing platforms like CPUs. The conventional spatial parallelization is very resource demanding. More efficient functional parallelization, i.e. pipelining can largely increase throughput at a cost of lower area and energy consumption. However, RNNs have

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

constrained potential for functional parallelization due to the recurrent nature in contrast to feed-forward NNs. Because of data dependencies the next input cannot be processed until the processing of the preceding input is not complete. Consequently, the pipeline is part time idle. The FPGAs allow for customized pipelined architectures that can overcome the problem. NNs relaying on trained network parameters can require to reprogramme the weights as the result machine learning benefits from reconfigurability of FPGAs.

In summary, multiple trade-offs between different types of memories and bandwidth, between custom precision and accuracy, between different parallelization schemes, area and energy consumption make custom hardware architecture the top choice especially for embedded systems that have constrained energy budget.

We extend the previous research in the area by proposing the first hardware architecture of BLSTM with CTC. In the new architecture, we rearrange memory access patterns that allows us to avoid a duplication of a hidden layer, and also to reduce the memory required for intermediate results and processing time. Additionally, the proposed architecture allows for implementation of uni-directional LSTM that extends the application area. Based on the new architecture, we present an FPGA hardware accelerator designed for a highly accurate character recognition of old German text (Fraktur) [5]. The network is composed of one hidden layer with 100 hidden nodes in each of the forward and backward directions of our BLSTM network (200 hidden nodes altogether). We show feasibility of implementing a medium size LSTM network making use of only on-chip memory that minimizes power consumption and brings higher performance as we are not limited with off-chip memory bandwidth. The new hardware architecture makes use of 5-bit fixed-point numbers to represent weights vs. minimum 16-bits in the previous works without impairing accuracy. The reference recognition accuracy is 98.2337% using single-precision floating-point format. The hardware accelerator achieves 97.5120% recognition accuracy, while outperforming high-performance CPU in terms of runtime and consuming less energy than low power systems. The experiments show that the proposed architecture achieves a throughput of 152.16 Giga-Operation Per Second (GOPS) at 166 MHZ that is the highest throughput compared to the state-of-the-art [8].

The paper is structured as follows. In Section II, we review the LSTM algorithm. In Section III, we review the previous publications of FPGA-based hardware architectures of RNNs. The novel architecture is described in Section IV. In Section V, we introduce software reference implementations. Finally, Section VI presents a comparison to software reference implementations in term of runtime, energy consumption and accuracy of recognition. Section VII concludes the paper.

II. LSTM Theory

For the sake of clarity we review the basic LSTM algorithm, earlier presented in [3]. The LSTM architecture is composed of a number of recurrently connected "memory cells". Each memory cell is composed of three multiplicative gating connections, namely input, forget, and output gates ($\boldsymbol{i}$, $\boldsymbol{f}$, and $\boldsymbol{o}$, respectively, in Eq. 1); the function of each gate can be interpreted as write, reset, and read operations, respectively, with respect to the cell internal state ($\boldsymbol{c}$). The gate units in a memory cell facilitate the preservation and access of the cell internal state over long periods of time. The peephole connections ($\boldsymbol{p}$) are supposed to inform the gates of the cell about its internal state. There are recurrent connections from the cell output ($\boldsymbol{y}$) to the cell input ($\boldsymbol{I}$) and the three gates. Eq. 1 summarizes formulas for LSTM network forward pass. Rectangular input weight matrices are shown by $\boldsymbol{W}$, and square weight matrices by $\boldsymbol{R}$. $\boldsymbol{x}$ is the input vector, $\boldsymbol{b}$ refers to bias vectors, and t denotes the time (and so $t-1$ refers to the previous timestep). Activation functions are point-wise non-linear functions, that is *logistic sigmoid* ($\frac{1}{1+e^{-x}}$) for the gates (σ) and *hyperbolic tangent* for input to and output from the node (l and h). Point-wise vector multiplication is shown by $\odot$ (equations adapted from [9]):

$$
\begin{aligned}
\boldsymbol{I}^t &= l(\boldsymbol{W}_I \boldsymbol{x}^t + \boldsymbol{R}_I \boldsymbol{y}^{t-1} + \boldsymbol{b}_I) \\
\boldsymbol{i}^t &= \sigma(\boldsymbol{W}_i \boldsymbol{x}^t + \boldsymbol{R}_i \boldsymbol{y}^{t-1} + \boldsymbol{p}_i \odot \boldsymbol{c}^{t-1} + \boldsymbol{b}_i) \\
\boldsymbol{f}^t &= \sigma(\boldsymbol{W}_f \boldsymbol{x}^t + \boldsymbol{R}_f \boldsymbol{y}^{t-1} + \boldsymbol{p}_f \odot \boldsymbol{c}^{t-1} + \boldsymbol{b}_f) \\
\boldsymbol{c}^t &= \boldsymbol{i}^t \odot \boldsymbol{I}^t + \boldsymbol{f}^t \odot \boldsymbol{c}^{t-1} \\
\boldsymbol{o}^t &= \sigma(\boldsymbol{W}_o \boldsymbol{x}^t + \boldsymbol{R}_o \boldsymbol{y}^{t-1} + \boldsymbol{p}_o \odot \boldsymbol{c}^t + \boldsymbol{b}_o) \\
\boldsymbol{y}^t &= \boldsymbol{o}^t \odot h(\boldsymbol{c}^t)
\end{aligned}
\tag{1}
$$

CTC is the output layer used with LSTM networks designed for sequence labelling tasks. Using the CTC layer, LSTM networks can perform the transcription task without requiring pre-segmented input, as it is trained to predict the conditional probabilities of the possible output labellings, given input sequences. CTC layer has K units, $K-1$ being the number of elements in the alphabet of labels. The outputs are normalized using softmax activation function at each timestep. The softmax activation function ensures that network outputs are all in the range between zero and one, and they sum to one at each timestep. This means that the outputs can be interpreted as the probabilities of the characters at a given timestep (column in our case). The first $K-1$ outputs provide predictions over the probabilities of their corresponding labels for the input timestep, and the remaining one unit estimates the probability of a blank or no label. Summation over the probabilities that correspond to a particular output label yields its estimated probability (see [7] for more details and related equations).

III. Related Works

To the best of our knowledge, only a very limited number of publications on FPGA-based hardware architectures of both recognition and learning phases of RNNs exist.

A recursive learning scheme for RNNs based on hardware-aware Simultaneous Perturbation Method was proposed in [10]. They implemented a serial processing of a small size RNN of 32 neurons with on-chip learning scheme on FPGA using a single precision floating-point format.
In [11], the authors presented a transformation strategy which leads to replacing the original LSTM learning algorithm with Simultaneous Perturbation Stochastic Approximation that yields competitive accuracy to the original algorithm, while enabling efficient hardware implementation of LSTM with on-chip learning.
An FPGA acceleration framework for RNN based language model (RNNLM) was proposed in [12]. They focused on accelerating the training procedure for RNNLM and presented an FPGA implementation of a standard RNN. In their work, they

used Back-Propagation Through Time (BPTT) algorithm that was considered to be prohibitively complex at the time [10]. The main idea was to unfold the RNN to a finite feed-forward structure with a fixed number of steps and compute them in parallel. The authors balanced the workload by conducting the calculation of the hidden layer in serial while parallelizing the computation of the output layer. The weights and intermediate data were stored in off-chip memory. They have shown that a mixed-precision scheme with 16 bits for output layer weight matrix and 64 bits for the rest of training parameters does not compromise the quality of the training comparatively with floating-point design.
We are aware only about a single published hardware architecture and its implementation of LSTM. In [8] authors presented an FPGA-based hardware implementation of pre-trained LSTM with 2 layers and 128 hidden units. It was tested using a character level language model. In the design, they used 16-bit quantization for weights and input data both of which were stored in off-chip memory. They used only partial parallelization that implies time multiplexing of the computational units to compute next layer or next timestep output. Intermediate results were sent back to off-chip memory in order to be read for the next stage of computation.

This work presents a different architecture from [8], in the sense that it was designed for BLSTM. In the current work, we also propose architecture of CTC for OCR that has not been presented in the previous publications. The proposed architecture processes $\boldsymbol{i}$, $\boldsymbol{f}$, $\boldsymbol{o}$, $\boldsymbol{I}$ (Eq. 1) in parallel rather than in sequential stages. All intermediate results and parameters are stored in on-chip memory but input data in DRAM without affecting throughput. The architecture is implemented with 5-bit fixed-point numbers for weights in contrast to minimum 16 bits in the previous publications. In our design, all blocks are synchronized implicitly as they are implemented with AXI4-Stream interface with blocking reads and writes that simplifies design, integration and debugging.

IV. Architecture

In this chapter we address the challenges highlighted in the Section I. First, we follow an overview of the complete system.

A. The System Architecture Overview

The hardware accelerator is implemented on Xilinx Zynq-7000 XC7Z045 SoC. The software part is running on Processing System (PC) and hardware part is implemented in Programmable Logic (PL). The software part plays an auxiliary role. It reads all images required for the experiment from SD card to DRAM and iteratively configures hardware blocks for all images. As soon as the last image has been processed, the software computes a recognition accuracy of the results based on Levenshtein distance [13] with respect to a reference output.

The NN implemented in the paper, see Fig. 1 includes an input layer that receives images column by column. In the testing set of T text lines, each is represented with a gray-scale image with P pixels height and variable length up to C columns. The BLSTM is composed of a Forward Hidden Layer (FHL) that includes n_i^f LSTM cells processing images from left to right and a Backward Hidden Layer (BHL) that includes n_i^b LSTM cells processing images from right to left each with a distinct set of weights, where $(i \in 0 \dots N-1)$. Each LSTM

TABLE I: General information

Number of images in the testing set, T	3401
Max number of columns per image, C	732
Number of pixels per column, P	25
Number of LSTM memory cells in a hidden layer, N	100
Number of gates including input to a memory cells, G	4
Number of inputs per gate, S^H	126
Number of units in the output layer, K	110
Number of inputs per output unit, S^O	201

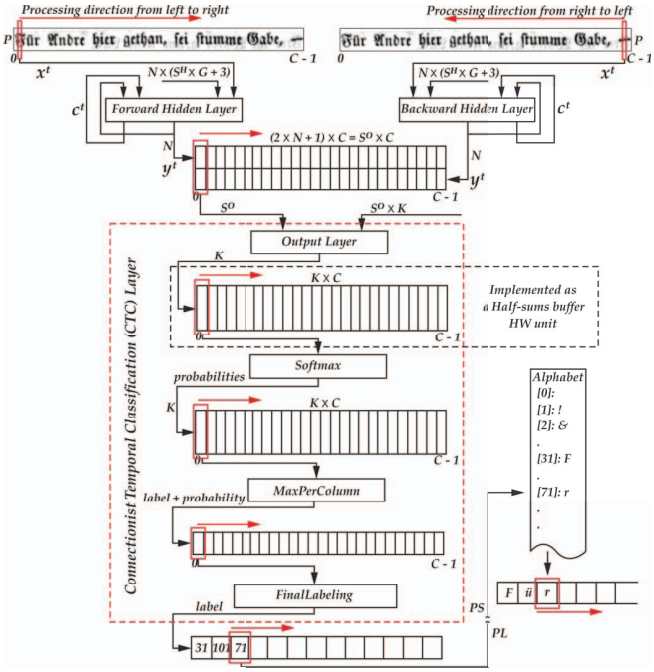

Fig. 1: The design overview.

gate receives S^H source values that include single bias value, P pixels and N output values received recurrently from the previous time step. The output layer includes n_i^o units with $(i \in 0 \dots K-1)$, where each receives a single bias value and $2N$ values that are concatenated outputs of the hidden layers from corresponding columns of the image. For classification purposes, we make use of CTC layer.

B. The parallelization schemes

A RNN can be mapped to a number of different architectures with different complexity and throughput. The highest throughput can be achieved in the case of fully unrolled RNN over all timesteps (columns in the case of OCR), see Fig. 2 (a). $\boldsymbol{x}^t$ denotes an input vector to each memory cell at time t. In terms of hardware architecture, it is identical to applying spatial parallelization to the complete NN and functional prallelization, i.e. pipelining to each instance. This results in a throughput of one image per clock cycle at a cost of prohibitive resource consumption and memory bandwidth. Less resource consuming architecture will imply to instantiate all neurons over a single timestep with pipelining, illustrated in Fig. 2 (b). It results in a throughput of a single image column per clock cycle that will still require prohibitive resources even in the case of moderate size RNN. In the proposed architecture we instantiate only a single LSTM memory cell, see Fig. 2 (c) with all multiplicative units unrolled. All inputs corresponding to a single cell in one timestep are processed in parallel, while

all memory cells are processed in a sequence. This approach results in minimal resource consumption, while providing a throughput of a single neuron function per clock cycle after applying pipelining.

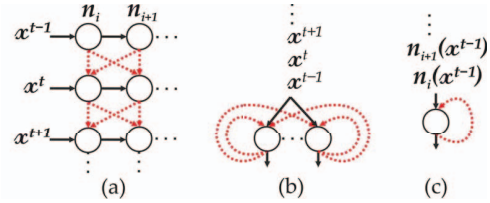

Fig. 2: The parallelization approaches.

C. The complexity reduction through the precision reduction

Multiplicative units of the hidden and output layers perform operations with the network weights that are responsible for the largest part of required memory bandwidth. Furthermore, the units are responsible for the highest resource consumption. Thus, the matrix-vector multiplication units become top priority for optimization. In contrast to the previous hardware implementations of RNNs that use at least 16-bit fixed-point numbers for weights representation, we compress the precision of the network parameters even further and use 5-bit fixed-point numbers for weights and inputs, and up to 16 bits for internal state and intermediate results, with exception of softmax function, where we used up to 32 bits. In Section VI, we show a negligible accuracy reduction in comparison to single-precision floating-point software implementation.
All 5 activation functions of LSTM memory cell are implemented using LUTs with at maximum of 256 8-bit entries. The softmax function uses exponent that is implemented with a LUT including 256 24-bit entries.

D. The hidden layer architecture

For ease of understanding, we introduce sets of outputs ${}_jY^f$, ${}_jY^b$, ${}_jY^o$ from FHL, BHL and output layer respectively with $(j \in 0 \dots C-1)$ corresponding to a column. As already stated, spatial parallelization of BLSTM suffers from high resource utilization. Besides, RNNs have constrained benefits from functional parallelization. In the chosen approach from Fig. 2 (c), the pipeline is part time idle between the time we feed input corresponding to n_{N-1} memory cell (neuron) to the pipeline and time it is processed.

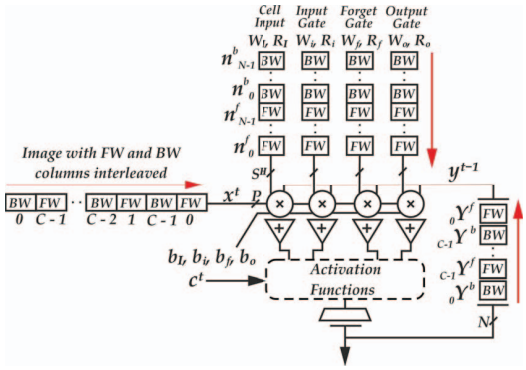

Fig. 3: The hidden layer architecture.

We rearrange memory access patterns and propose an architecture that processes image with forward and backward columns interleaved, see Fig. 3. The datapath is a fully pipelined instance of a single LSTM neuron with all multiplicative units fully unrolled. Every clock cycle it receives weights corresponding to a distinct neuron that allows us to achieve a throughput of one neuron function per clock cycle. The neurons processing images in forward and backward directions handled interchangeably. Beginning from the first column, the computation of the n^f_{N-1} neuron is followed by the n^b_0 neuron. And as soon the n^b_{N-1} is fed to the pipeline, the n^f_0 neuron provided with data from the previous time step and a new image column can be fed to the datapath, and this way we avoid idling pipeline. At the same time the throughput is preserved to a single neuron per clock cycle and we need only a half of memory bandwidth with respect to duplicated datapath; however, we cannot avoid a doubling of weights' memory. The proposed architecture turns the drawback of the RNNs into advantage, i.e. backward processing comes in low cost. There is only a latency penalty of $N-D$ clock cycles, if $N > D$ as the depth of the pipeline cannot hide the complete processing latency of BHL. The same approach can be applied to uni-directional RNN that will allow for processing separate samples, e.g. images at the same time.

E. The output layer architecture

For the sake of clarity, we note that the output layer processes the concatenated outputs from FHL and BHL only from corresponding (the same) columns of the image that can require to wait $2 \times N \times C + D$ clock cycles before all outputs from the hidden layers are available, and $2 \times N \times C$ memory entries to store the outputs. The rearranged memory access pattern that results in handling forward and backward columns in interleaved manner allows for processing outputs from the hidden layers as soon as the results of the centric columns are available that will half the waiting time and memory but will still require to instantiate $2N$ multiplicative units in order to process the concatenated outputs from the hidden layers.

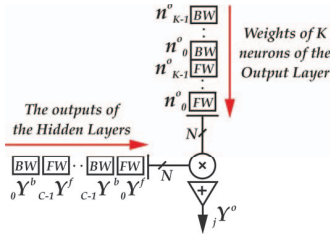

Fig. 4: The output layer architecture.

We propose to start processing as soon as the first N outputs from FHL are available that avoids implementing large buffer. Furthermore, we instantiate only a single neuron of the output layer with N multiplicative units and reduction tree, see Fig. 4. Every K clock cycles the unit consumes N outputs from hidden layer and every cycle a new set of weights corresponding to a class. In the proposed architecture, we reduce the time and the number of multiplicative units into half.

F. The half-sums buffer architecture

The ${}_jY^o$ outputs corresponding to the FHL and BHL from the same columns have to be still summed up that requires

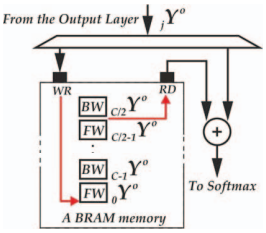

Fig. 5: The half-sums buffer architecture.

buffering of the half-sums. Without proposed memory access pattern, the algorithm would require storing of $2 \times K \times C$ values. In contrast, in the proposed architecture, see Fig. 5, we reduce the required memory into half. As soon as the last value corresponding to the centric column from BHL has been written to the memory, we stop writing to the memory and start reading the values and summing them with the new outputs from the output layer that will correspond to FHL values from the centric column. The proposed memory pattern results that first writes and later reads of the values from corresponding columns are also interleaved. The red arrows in Fig. 5 show the first values to be written and the first ones to be read.

G. The Softmax, MaxPerColumn and FinalLabelling cores

See Fig. 1, the $Softmax$ hardware unit normalizes outputs from output layer using softmax activation function. The following $MaxPerColumn$ unit finds a label with the highest probability per column and forwards its index and corresponding probability to the next block. The $FinalLabeling$ block processes labels from left to right. If the label corresponds to class zero, we say it's a blank. Otherwise, we continue processing until next blank and find a maximum probability in the block between the blanks. The label corresponding to the maximum probability points to the character that is considered to be represented between the blanks.

H. The interfacing

All weights and parameters are stored in on-chip memory. The implemented design running at 166 MHz results in $(S^H \times G + 3 + S^O/2) \times 5bits \times 166MHz = 62GB/s$ of required memory bandwidth only for network weights in order to achieve a throughput of a single neuron per clock cycle. The calculated bandwidth is prohibitive in the case of embedded system coupling with off-chip memory even in the case of small and medium size NNs. The modern FPGA chips have an adequate number of on-chip memory resources. However, the number of evaluable memory ports can become a limiting factor as the architecture makes use of hundreds of values in parallel, namely $(S^H \times G + 3 + S^O/2)$. The 5-bit quantization per weight and concatenation in blocks of 7 reduces the burden on the number of ports. All images are stored in DRAM. Each pixel has 5-bit representation that results in 26 MB/s. The images and output labels are read/written using custom Direct Memory Access (DMA) blocks with AXI4 memory-mapped interface for reading/writing and AXI4-Stream interface for writing/reading.

V. Reference Implementations

For purposes of comparison, we implemented software references running on platforms with the following processors, see Table II. In the case of real time, i.e. on-line OCR, images appear one at a time, thus a system optimized for a single image processing is more suitable than a batch processing system optimized for off-line processing. We took into account both scenarios and implemented two parallelization schemes. In the case of the first approach, parallelization was applied on a level of neurons, meaning that all computational units (threads) process separate neuron functions at each point of time. In the latest approach, all computational units (threads) process separate images at each point of time that is identical to processing them in batches with a batch size equal to a number of available threads. For a fair comparison with respect to accuracy/complexity, we implemented the network with single-precision floating-point/integer weights. For optimization we used OpenMP API for parallelization; SSE4.2/AVX2 and NEON intrinsics for vectorization for Intel and ARM respectively. All software implementations have been complied with gcc 4.9.2 -O3. In the case of optimized integer implementation, although 8 bits would be enough, we switched to 16 bits per weights and inputs because Intel SSE/AVX instruction sets do not support multiplication on a vector of 16/32 packed signed 8-bit operands [14] that is a constraint in the case of RNN, where both weights and recurrent path values can be negative.

TABLE II: The reference platforms

Processor	TDP, [W]	Cores (Threads)	Optimizations
Intel Xeon E5-2670 v3 @ 3.1 GHz (turbo)	120	12(24)	AVX2
Intel Core i7-4790T @ 3.9 GHz (turbo)	45	4(8)	
Intel Atom C2758 @ 2.4 GHz	20	8(8)	SSE4.2
ARM Cortex-A53 @ 1.2 GHz	<1	4(4)	NEON
ARM Cortex-A9 @ 800 MHz	<1	2(2)	intrinsics

VI. Results

We make use of custom IP blocks designed using Xilinx Vivado High-Level Synthesis (v2015.3). The system integration step was performed using Xilinx Vivado Design Suite (v2015.3). The resource utilization of a single instance of the complete hardware accelerator shows that none of the main resources exceed 15%, see Table III.

TABLE III: Resource utilization

Resources	Single instance	On-line	Off-line
LUT	32815 (15%)	161574 (74%)	190036 (87%)
FF	14532 (3%)	51213 (12%)	78516 (18%)
BRAM_36K	83 (15%)	339 (62%)	498 (91%)
DSP	33 (4%)	195 (22%)	198 (22%)
Slice	10419 (19%)	47160 (86%)	53756 (98%)

We can put six instances of the complete design that allows for processing six separate images at each point of time that is beneficial for off-line batch processing. For on-line, i.e. without batch processing, we implemented a configuration that uses six instances of the main blocks except DMA and $FinalLabeling$. This scheme lets us to process a single image in shorter time as each instance iterates only $2N/6$ iterations over each column. All three configurations are running on Xilinx Zynq-7000 XC7Z045 SoC at 142 MHz. In Table IV, we are comparing FPGA configurations with respect to accuracy, runtime and energy required to process the complete dataset. In case of energy measurements, we compare energy consumed for computation only: $Energy = (Pcomp - Pidle) \times Runtime$, where $Pcomp$ - power consumption of the complete system during computation, $Pidle$ - power consumption of the complete system without workload or FPGA being configured.

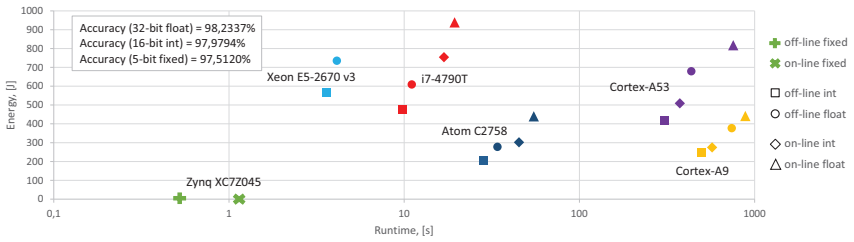

Fig. 6: The comparison between FPGA and software implementations.

TABLE IV: Comparison between FPGA configurations

Configuration	Runtime, [sec]	Energy, [J]	Accuracy, [%]	Throughput, [GOPS]
[8] @142MHz	-	-	-	0.284
*Single instance @142MHz	2.764	3.87	97.5120	130,40
*Single instance @166MHz	2.371	4.03	97.5120	152,16
*On-line @142MHz	1.170	8.15	97.6487	308,05
*Off-line @142MHz	0.520	6.45	97.5120	693,12

* this paper

In the state-of-the-art paper [8], the authors used a total number of multiplications and additions to express the computational complexity, although this metric does not take into account a difference in implementation complexity. For the purposes of direct comparison, we run our design at 142 MHz, we take into account only complexity of LSTM and neglect the rest of the network. According to Eq. (1), the complexity of a single LSTM layer is $N \times (2 \times S^H \times G + 10)$, where two comes from multiplication and addition counted as separate operations. The computed number of operations has to be repeated for each column of each image that is 1770216 columns in the complete testing dataset. As the proposed BLSTM architecture allows for implementation of uni-directional LSTM, we double the number of operations that results in 130,40 GOPS. The achieved throughput is 459 times higher than state-of-the-art, see Table IV.

In Fig. 6, we present comparison between the last two FPGA configurations from Table IV and software reference implementations from Section V. For Xeon E5-2670 v3 we omit the results for no batch, i.e. on-line processing due to high runtime that is a reason of high inter-thread synchronization cost in the case of fine-grained parallelization. We show that in the both scenarios, i.e. on-line and off-line OCR, our dedicated hardware accelerator outperforms high-performance CPU in terms of runtime, while consuming less energy than low power systems with $<1\%$ of accuracy reduction with respect to single-precision floating-point implementation. The accuracy of integer software implementations is 97.9794%.

VII. Conclusion

In this paper, we proposed the first hardware architecture of BLSTM neural network with CTC layer for OCR. Based on the new architecture, we presented an FPGA hardware accelerator that achieves 459 times higher throughput than state-of-the-art. We presented two configurations of the accelerator, one optimized for off-line OCR and another for on-line, i.e. no batch OCR scenario. We performed rigorous comparison with highly optimized single-precision floating-point and integer software implementations running on various platforms. We demonstrated that visual recognition task benefits from being migrated to our dedicated hardware accelerators in both scenarios and outperforms high-performance CPU in terms of runtime, while consuming less energy than low power systems with $<1\%$ of accuracy reduction with respect to single-precision floating-point implementation.

References

[1] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, nov 1997.

[2] A. Graves and J. Schmidhuber, "Framewise phoneme classification with bidirectional LSTM and other neural network architectures." *Neural networks : the official journal of the International Neural Network Society*, vol. 18, no. 5-6, pp. 602–10, 2004.

[3] A. Graves, "Supervised Sequence Labelling with Recurrent Neural Networks," Ph.D. dissertation, Berlin, Heidelberg, 2008.

[4] T. M. Breuel, A. Ul-hasan, M. A. Azawi, and F. Shafait, "High-Performance OCR for Printed English and Fraktur using LSTM Networks," in *International Conference on Document Analysis and Recognition, 2013.*, 2013, p. In Press.

[5] M. R. Yousefi, M. R. Soheili, T. M. Breuel, E. Kabir, and D. Stricker, "Binarization-free ocr for historical documents using lstm networks," in *Document Analysis and Recognition (ICDAR), 2015 13th International Conference on*. IEEE, 2015, pp. 1121–1125.

[6] M. R. Yousefi, M. R. Soheili, T. M. Breuel, and D. Stricker, "A comparison of 1d and 2d lstm architectures for the recognition of handwritten arabic," in *SPIE/IS&T Electronic Imaging*. International Society for Optics and Photonics, 2015, pp. 94 020H–94 020H.

[7] A. Graves, S. Fernández, F. Gomez, and J. Schmidhuber, "Connectionist temporal classification: labelling unsegmented sequence data with recurrent neural networks," in *Proceedings of the 23rd international conference on Machine learning*, ser. ICML '06. New York, NY, USA: ACM, 2006, pp. 369–376.

[8] 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

[9] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *arXiv*, p. 10, 2015.

[10] Y. Maeda and M. Wakamura, "Simultaneous perturbation learning rule for recurrent neural networks and its fpga implementation," *IEEE Transactions on Neural Networks*, vol. 16, no. 6, pp. 1664–1672, 2005.

[11] R. Tavcar, J. Dedic, D. Bokal, and A. Zemva, "Transforming the lstm training algorithm for efficient fpga-based adaptive control of nonlinear dynamic systems," *Informacije MIDEM, Journal of Microelectronics, Electronic Components and Materials*, vol. 43, no. 2, pp. 131–138, 2013.

[12] S. Li, C. Wu, H. Li, B. Li, Y. Wang, and Q. Qiu, "Fpga acceleration of recurrent neural network based language model," in *Field-Programmable Custom Computing Machines (FCCM), 2015 IEEE 23rd Annual International Symposium on*. IEEE, 2015, pp. 111–118.

[13] R. a. Wagner and M. J. Fischer, "The String-to-String Correction Problem," *Journal of the ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[14] "Intel intrinsics guide." [Online]. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/