IEICE TRANSACTIONS on Electronics

VOL. E103-C NO. 5
MAY 2020

A PUBLICATION OF THE ELECTRONICS SOCIETY
The Institute of Electronics, Information and Communication Engineers
Kikai-Shinko-Kaikan Bldg., 5-8, Shibakoen 3chome, Minato-ku, TOKYO, 105-0011 JAPAN

PAPER

A Highly Configurable 7.62GOP/s Hardware Implementation for LSTM

Yibo FAN[†a)], *Member*, Leilei HUANG[†], Kewei CHEN[†], *and* Xiaoyang ZENG[†], *Nonmembers*

SUMMARY The neural network has been one of the most useful techniques in the area of speech recognition, language translation and image analysis in recent years. Long Short-Term Memory (LSTM), a popular type of recurrent neural networks (RNNs), has been widely implemented on CPUs and GPUs. However, those software implementations offer a poor parallelism while the existing hardware implementations lack in configurability. In order to make up for this gap, a highly configurable 7.62 GOP/s hardware implementation for LSTM is proposed in this paper. To achieve the goal, the work flow is carefully arranged to make the design compact and high-throughput; the structure is carefully organized to make the design configurable; the data buffering and compression strategy is carefully chosen to lower the bandwidth without increasing the complexity of structure; the data type, logistic sigmoid (σ) function and hyperbolic tangent (tanh) function is carefully optimized to balance the hardware cost and accuracy. This work achieves a performance of 7.62 GOP/s @ 238 MHz on XCZU6EG FPGA, which takes only 3K look-up table (LUT). Compared with the implementation on Intel Xeon E5-2620 CPU @ 2.10GHz, this work achieves about 90× speedup for small networks and 25× speed-up for large ones. The consumption of resources is also much less than that of the state-of-the-art works.

key words: *Recurrent Neural Networks (RNN), Long Short-Term Memory (LSTM), hardware implementation*

1. Introduction

In recent years, deep neural networks (DNNs) have made a series of incredible achievements in many areas [1]–[3]. Taking pattern recognition as an example, DNN exceeds many excellent algorithms and now represents the new state of the art. As the name implies, neural networks work in a similar way as the human neurons. However, unlike human neural networks which could store information over time, DNNs could only work on the current input.

To address this issue, recurrent neural networks (RNNs) are designed later. This kind of neural networks would store previous outputs and serve them as assistant information for current predictions, which makes RNNs naturally suitable for solving problems related to sequences like speech recognition, language translation and image analysis. However, the information in RNNs could easily fall into vanishing or exploding [4].

To overcome this shortcoming, a novel type of RNNs, namely, long short-term memory (LSTM), is proposed. LSTM introduces an efficient gradient-based algorithm to ensure that the internal state remains constant during the recurrent process. Due to this advantage, LSTM has been widely refined and popularized by many researchers on software platforms. However, it is quite difficult for CPUs and GPUs to offer sufficient parallelism for the matrix calculation in LSTM. In consideration of this, hardware implementations are needed especially in embedded systems. Unfortunately, most of the existing hardware architectures for LSTM are non-configurable, which cannot support different sizes, different data range, precision or throughput. Moreover, the resource consumption of those designs is usually too much for the requirement of cost-sensitive applications like the Internet of Things (IoT).

In view of the above situation, a highly configurable 7.62 GOP/s hardware implementation for LSTM which takes only 3K look-up table is proposed in this paper. The rest of this paper would present it in a detailed way. To be more specific, Sect. 2 gives the background, motivations and contributions of this paper. Section 3 illustrates the proposed hardware implementation. Comparisons are made in Sect. 4. Finally, Sect. 5 concludes this paper.

2. Background, Motivations and Contributions

2.1 Background

2.1.1 Concept of Memory Cell

The concept of long short-term memory (LSTM) neural network was first put forward by S. Hochreiter and J. Schmidhuber [4] in 1997 to overcome the long-term memory dependency problem of conventional RNNs. To be more specific, in a standard RNN structure, each neuron passes the output information to a successor. As the propagation goes on, the information passing along the chain would easily fall into a situation either vanishing or exploding. Therefore, the output at the end of sequence usually fails to build a stable relationship with the input at the beginning of sequence.

To solve this problem, LSTM, a modified version of RNN, introduces the concept of "memory cell" to establish long-term memory dependency, which is composed of the cell state c_t, three gates which protect and control c_t and one gate which determines the output h_t. The cell state is like a conveyor belt, which runs straight down the entire chain with only some minor linear interactions. In this way, it is natural for the information to flow along the chain without

Manuscript received February 14, 2019.
Manuscript revised September 8, 2019.
Manuscript publicized November 27, 2019.

[†]The authors are with the State Kay Lab of ASIC & System, Fudan University, Shanghai, 200433 China.
a) E-mail: fanyibo@fudan.edu.cn (Corresponding author)

DOI: 10.1587/transele.2019ECP5008

being changed, which solves the problem of long-term dependency.

2.1.2 A Commonly-Adopted LSTM Model

There are several updated versions of LSTM models nowadays, such as those proposed by J. Schmidhuber et al. [5]–[7]. And as a summary article, J. Schmidhuber et al. [7] analyzed the role and utility of computational components of typical LSTM models. Among those models, the one described by Eqs. (1)–(6) is commonly adopted. This model has no peephole connections, thus the computation speed is relatively high and the resources consumption is relatively low. For those reasons, this model is also adopted by the proposed implementation.

$$f_t = W_{fx}x_t + W_{fh}h_{t-1} + b_f \tag{1}$$
$$i_t = W_{ix}x_t + W_{ih}h_{t-1} + b_i \tag{2}$$
$$g_t = W_{gx}x_t + W_{gh}h_{t-1} + b_g \tag{3}$$
$$o_t = W_{ox}x_t + W_{oh}h_{t-1} + b_o \tag{4}$$
$$c_t = \sigma(f_t) \odot c_{t-1} + \sigma(i_t) \odot \tanh(g_t) \tag{5}$$
$$h_t = \sigma(o_t) \odot \tanh(c_t) \tag{6}$$

In Eqs. (1)–(6),

i. σ stands for the logistic sigmoid function;
ii. tanh stands for the hyperbolic tangent function;
iii. $\odot$ stands for the element wise multiplication;
iv. f_t, i_t, g_t, and o_t stand for the input vectors of forget gate, input gate, candidate cell gate, and output gate at round t respectively;
v. x_t, c_t and h_t stand for the input vector, the cell state and the output vector of LSTM at round t;
vi. W_{**} stand for the weight matrices, where, the first symbol $*$ could be f, i, g and o; the second $*$ could be x and h;
vii. b_* stand for the bias vectors, where, the symbol $*$ could be f, i, g and o.

Attention should be paid on the flowing points.

i. the σ and tanh here are vector-based functions. In other words, they return the sigmoid or hyperbolic tangent value of each element of the input vectors.
ii. unlike other papers [9]–[12], [14]–[18], the $\mathrm{f_t}$, $\mathrm{i_t}$, $\mathrm{g_t}$, and $\mathrm{o_t}$ here stand for the input vectors instead of the output vectors of the related gates.
iii. Although the form of Eqs. (1)–(6) are quite simple, they perform a considerable amount of calculation.
For example, if one LSTM layer has H neurons and X inputs, then x_t would have a size of $\mathrm{X} \times 1$; h_{t-1} and b_f would have a size of $\mathrm{H} \times 1$; W_{ix} would have a size of $\mathrm{H} \times \mathrm{X}$; W_{fh} would have a size of $\mathrm{H} \times \mathrm{H}$.
In this way, Eq. (1) alone performs $\mathrm{H} \times (\mathrm{H} + \mathrm{W})$ times of multiplication and make a sum of $\mathrm{H} \times (\mathrm{H} + \mathrm{W} + 1)$ elements in one single round, which is a heavy burden for software platforms.

2.1.3 Related Papers

LSTMs, just like other neural networks, are conventionally implemented on software platforms. But as analyzed above, it faces greater challenges on the matrix-vector operations [8], let alone that the energy efficiency is rather too low because software platforms usually lack in parallelism.

In view of this, several hardware accelerators for LSTMs have been proposed to improve the performance.

A.X.M. Chang et al. [9] described a LSTM hardware implementation with 2 layers and 128 neurons based on FPGA. Later, they [10] presented three new hardware accelerators for LSTM to further improve the performance. Unfortunately, unlike software solutions, the layer size, namely, the neuron number and the input length, of those designs are non-configurable.

On the contrary, J.C. Ferreira et al. [11] proposed a size-configurable design, the parallelism of which is also high enough. However, the high parallelism is achieved at the cost of great design complexity.

In a similar way, Y. Guan et al.'s [12] design keeps the accuracy of results by adopting floating-point type data, which brings a large computational workload to the hardware system.

Other related papers also proposed remarkable algorithms or structures. For example, Y. Zhang et al. [13], [14] proposed a dense and sparse network and later an overlap of computation and data; Z. Wang et al. [15], [16] proposed a hybrid compression and later a mixed quantization scheme; J. Park el al. [17] and S. Han et al. [18] presents a sparse matrix format to improve the speed of LSTM accelerator.

2.2 Motivations and Contributions

Although several papers have proposed hardware solutions for LSTM, some defects still could be observed.

Firstly, unlike software ones, existing hardware solutions usually lack in configurability. As a mature design, the size of layer, the granularity of parallelism and the range of data should be configurable. Among the mentioned designs, only J.C. Ferreira et al. [11] and Y. Zhang et al. [13], [14] could support several layer sizes like 64 or 128 while other designs only support one certain size.

Secondly, the existing hardware solutions usually pay attention to the parallelism but overlook the bandwidth. Among the mentioned designs, only Z. Wang et al. [15] pays attention to the compression of weight matrices and S. Han et al. [18] chooses to store all weight matrices with on-chip memories.

Thirdly, float-point type, σ and tanh functions are costly to be implemented on hardware platform. As a result, the implementation of those computations may easily go to two different extremes. They are either over-simplified thus the performance would be too poor or over-complexed thus the hardware cost would be too much. For example, Y. Guan et al.'s [11] uses floating-point type data directly and

S. Han et al. [18] uses the look-up table, which both makes the final cost too much.

Considering the above situation, the proposed paper aims to put forward a hardware solution of LSTM to solve the above detects, which has the following contributions:

i. We carefully re-arranged the work flow to make the design compact and high-throughput;
(This design utilizes only 3K LUT but achieves a throughput of 7.62 GOP/s).
ii. We carefully organized the structure to make the design more configurable;
(The layer size, data range and parallelism could be configured on the fly, although the maximum layer size and parallelism must be pre-determined according to the system requirements.)
iii. We carefully chose the data buffering and compression strategy to lower the bandwidth without increasing the complexity of structure.
(Buffering strategy reduced 50% of the bandwidth.)
iv. We carefully optimized the data type, σ and tanh function to balance the hardware cost and accuracy.
(No obvious accuracy loss could be observed between the original version and the optimized version, but hardware cost is reduced to a large extend)

3. Hardware Implementation

3.1 Overview

Figure 1 gives the architecture of the proposed LSTM design, which shares the same notations with Eqs. (1)–(6). It can be easily seen from Fig. 1 that this design could be divided into two parts. One is the gating part responsible for the calculation of Eqs. (1)–(4). The data rate of this part is very high, so multiplier array and adder array are adopted to raise the parallelism. The other one is the network part responsible for the calculation of Eqs. (5)–(6). This part has a relatively low data rate, so only one instance of σ and tanh is used and shared to reduce the hardware cost.

The following sub-sections will give some discussions about the throughput, configurability, bandwidth and accuracy from the view of work flow, structure, buffering and compression strategy, data type and the implementation of σ and tanh function.

3.2 Work Flow and Throughput

3.2.1 Work Flow

The work flow of this design could be concluded into two phases, initialization phase and running phase. Again, the following notations share the same meaning with Eqs. (1)–(6).
In initialization phase:

i. All W_{**} are stored to off-chip memories;
ii. All b_* are stored to on-chip memories;
iii. All registers like the length of x_t, h_t are configured.

In running phase:

i. The design reads in the x_t of the current round and buffers them with on-chip memories.
ii. The design reads in the W_{**} in an interleaving way and send them to the multiplier and adder arrays together with x_t and h_t read from on-chip memories.
By word "interleaving", it means this design reads in the first row of W_{fx} and send with x_t; then it reads in the first row of W_{fh} and send with h_{t-1}; then it reads in the first row of W_{ix} and send with x_t; ...
In this way, the accumulator could get the result of f$_t$, i_t, g_t, o_t in the unit of element instead of vector. In other words, it could get the first elements of f$_t$, i_t, g_t, o_t; then the second elements of f_t, i_t, g_t, o_t; and then the third elements; ...
iii. Once the elements with the same order have been collected, the corresponding elements of c_t as well as h_t could be calculated by network part.
iv. Once the current round is finished, the ping-pong buffer for h_t is switched.

3.2.2 Throughput

Unlike general LSTM designs, the work flow is re-arranged

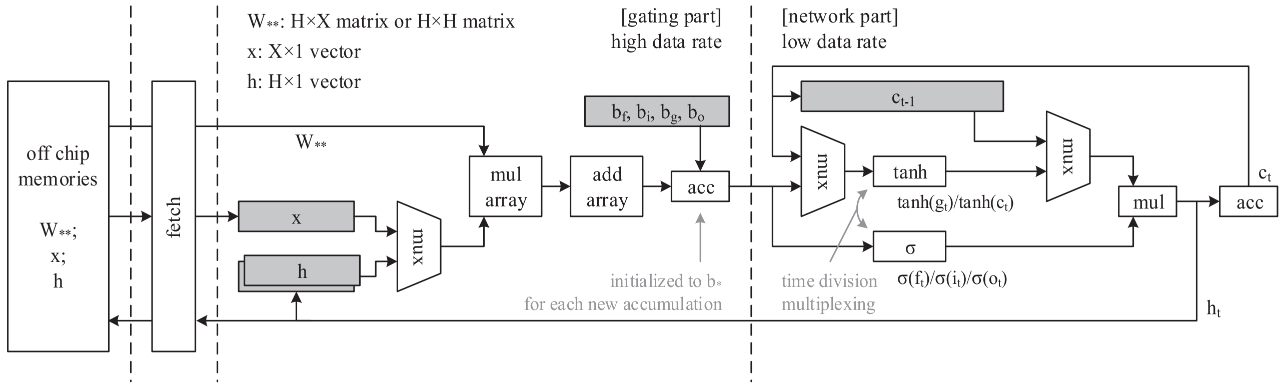


Fig. 1 Architecture of the proposed implementation

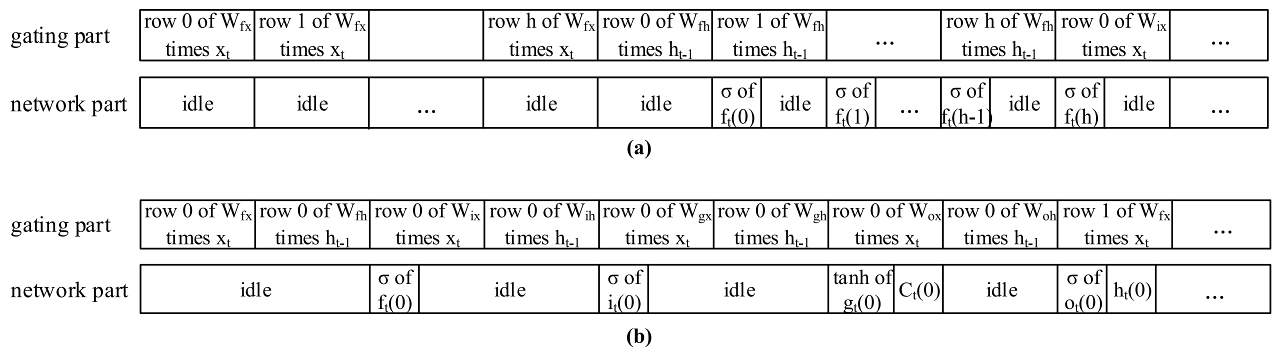


Fig. 2 (a) Space-time diagram of a non-interleaving implementation. (b) Space-time diagram of an interleaving implementation (the proposed one)

in an interleaving way so that the lifecycle of data f_t, i_t, g_t, o_t are greatly shortened. The difference of the interleaving and non-interleaving way is given in Fig. 2. Obviously, for the non-interleaving implementation, all elements of f_t are collected together, which is the same to i_t, g_t and o_t. As a result, those data have to be buffered before c_t and h_t are calculated out.

However, in the interleaving way, f_t, i_t, g_t and o_t with the same order are collected together. As a result, there is no need to store them to and fetch them from off-chip memories or buffer them with on-chip memories.

In addition, the calculation time of Eqs. (5)–(6) is covered by that of Eqs. (1)–(4). In this way, if bandwidth is not considered, the throughput would be directly determined by the calculation cycle of f_t, i_t, g_t, o_t, namely, the parallelism of multiplier and adder array. Thus, for LSTM layer with H neurons and X inputs, the cycle cost of one round is:

$$(X \times H + H \times H) \times 4/P \tag{7}$$

where, $X \times H$ is the cycle to calculate one $W_{*x}x_t$; $H \times H$ is the cycle to calculate one $W_{*h}h_t$; $\times 4$ is because four $W_{*x}x_t$ and $W_{*h}h_t$ in f_t, i_t, g_t, o_t need to be calculated; P is the parallelism.

3.3 Structure and Configurability

3.3.1 Layer Size

As mentioned before, the layer size could be configured on the fly. To support the configurability in layer size, the following items need to be adjusted.

i. the fetch module needs to know when to stop reading W_{**} and x_t;
ii. the accumulator needs to know when to start a new accumulation;
iii. the on-chip memory controller needs to know when to reset the address to 0 and switch the ping-pong buffer.

Thanks to the simplicity of the current structure, all of the above operations could be easily implemented with counters. However, since x_t, h_{t-1}, h_t and c_t are stored with on-chip memories, they will limit the maximum size of layer.

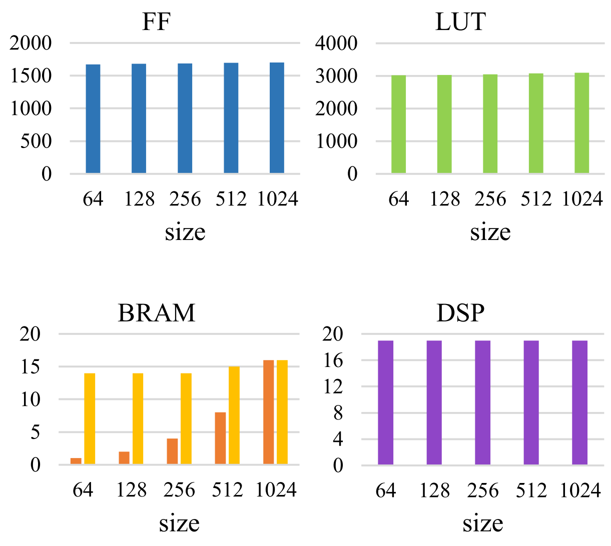


Fig. 3 Minimum resource to support different maximum layer size

Figure 3 gives the minimum resource to support different maximum layer size based on the synthesis result under FPGA XCZU6EG. (Parallelism is set to 32, which could be referred from Sect. 3.3.3). According to it, resources of flip flop (FF) and look-up table (LUT) increases slightly with the layer size while the digital signal process unit (DSP) keeps the same, because, for logic resource, the layer size would only have influence on the data width of counter. As to the memory resource, the increase of the logically-needed block memory (BRAM) is proportional to the increase of layer size, which is marked with darker yellow; however, the actually-used BRAM only increases under size 512, 1024, which is marked with lighter yellow, because the depth of most memories have not reached the depth of the BRAM.

3.3.2 Data Range

A fixed-point type with 16-bit width is adopted in this design to reduce the hardware cost of calculation. However, in order to meet the requirements of different scenarios, the data range could be configured on the fly, which is denoted by the Q value. For example, W_{fx} could be configured to

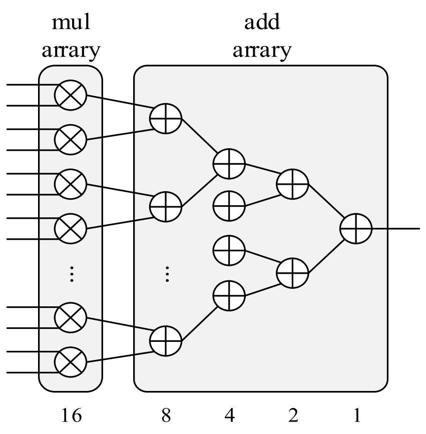


Fig. 4 Multiplier array and adder array

Q10.6, which means the highest 10 bits are integer part, while the lowest 6 bits are fractional part; x_t could be configured to Q4.12, which means the highest 4 bits are integer part, while the lowest 12 bits are fractional part; ... In this way, the hardware cost of the multiplier array and the adder array are still kept small because only fixed-point operations are involved; while the data range is extended by Q value to make this design adaptable to different scenarios. Extra cost is merely one configurable bit shifter in the accumulator. As to the accuracy of this optimization, it could be inferred from Table 3.

3.3.3 Parallelism

Unlike the layer size and data range, parallelism is not so meaningful to be configured on the fly, but it should be easily adjusted to meet the system requirements.

In this design, parallelism could be adjusted by

i. expanding or shrinking the data bus of W_{**};
ii. reshaping the on-chip memories for x_t and h_t;
iii. expanding or shrinking the size of multiplier and adder;

For example, if a parallelism of 16 is required, then the bus width of W_{**} should be expanded to 16 elements; the size of x_t and h_t should be reshaped to $16 \times (H/16)$ and $16 \times (X/16)$ instead of $1 \times H$ and $1 \times X$; the multiplier array and adder array should be expanded to handle 16 elements in one cycle, as described in Fig. 4.

Fortunately, the above adjustment could be easily controlled with parameter and generate block in Verilog HDL. In addition, although structures with low parallelism cannot be expanded to structure with higher parallelism on the fly, the converse of it is feasible, which should be meaningful under some low-power modes.

Of course, some tradeoff exists between the throughput and hardware cost under different parallelisms. As shown in Fig. 5, in addition to logic resource, memory resource also increases with the parallelism. The former one directly comes from the increase of adder array and multiplier array, while the later one comes from the change of memory shape. To be more specific, if the parallelism is 32, x_t requires a 32-in-width and (H/32)-in-depth memory instead of a 1-in-width and H-in-depth memory, which is similar to c_t, h_t and b_*. As a result, although the logically-needed memory keeps the same, the actually-used memory would be much larger, especially on FPGA platforms.

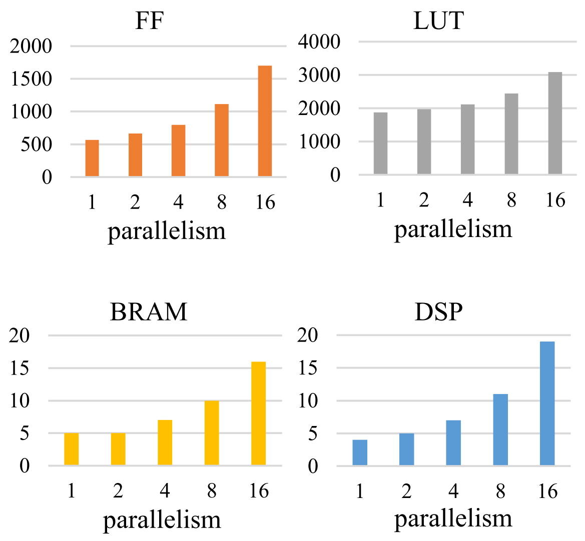


Fig. 5 Minimum resource to support different maximum parallelism

Table 1 Data size and bandwidth occupied by each data

Data	Size	Bandwidth (read)	Bandwidth (write)
W_{fx}, W_{ix}, W_{gx}, W_{ox}	$X \times H$	$X \times H$	\
W_{fh}, W_{ih}, W_{gh}, W_{oh}	$H \times H$	$H \times H$	\
x_t	X	$X \times H \times 4$	\
h_{t-1}	H	$H \times H \times 4$	\
b_f, b_i, b_g, b_o	H	H	\
f_t, i_t, g_t, o_t,	H	H	H
c_{t-1}	H	H	\
c_t, h_t	H	\	H

* Unit of size is element; unit of bandwidth is element/round
** The bandwidth listed here assumes that no on-chip memory is adopted.

3.4 Buffering and Compression Strategy and Bandwidth

3.4.1 Buffering Strategy

The data amount in LSTM design is quite large, which means the buffering strategy is very important to balance the occupation of bandwidth and the size of on-chip memories. To better analyze the situation, the detailed data amount and the corresponding bandwidth occupied are listed in Table 1.

Based on this table, the cost of the following two strategies could be easily estimated. If all of them are stored with off-chip memories, it requires a bandwidth of $4\times X\times H+4\times H\times H+X\times H\times 4+H\times H\times 4+4\times H+4\times(H+H)+H+2\times H$ element/round. If all of them are stored with on-chip memories, it requires a bandwidth of only $X + H$ element/round, which is used to read x_t and write h_t, but the on-chip memories are as large as $4 \times X \times H + 4 \times H \times H + X + 12 \times H$ elements.

$W_{fx}(0,0)$	$W_{fx}(0,1)$	...	$W_{fx}(0,X-1)$
$W_{fh}(0,0)$	$W_{fh}(0,1)$	...	$W_{fh}(0,H-1)$
$W_{ix}(0,0)$	$W_{ix}(0,1)$	...	$W_{ix}(0,X-1)$
$W_{ih}(0,0)$	$W_{ih}(0,1)$	...	$W_{ih}(0,H-1)$
$W_{gx}(0,0)$	$W_{gx}(0,1)$	...	$W_{gx}(0,X-1)$
$W_{gh}(0,0)$	$W_{gh}(0,1)$	...	$W_{gh}(0,H-1)$
$W_{ox}(0,0)$	$W_{ox}(0,1)$	...	$W_{ox}(0,X-1)$
$W_{oh}(0,0)$	$W_{oh}(0,1)$	...	$W_{oh}(0,H-1)$
$W_{fx}(1,0)$	$W_{fx}(1,1)$	...	$W_{fx}(1,X-1)$
$W_{fh}(1,0)$	$W_{fh}(1,1)$	...	$W_{fh}(1,H-1)$
$W_{ix}(1,0)$	$W_{ix}(1,1)$	...	$W_{ix}(1,X-1)$
$W_{ih}(1,0)$	$W_{ih}(1,1)$	...	$W_{ih}(1,H-1)$
		...	
$W_{gx}(H-1,0)$	$W_{gx}(H-1,1)$	...	$W_{gx}(H-1,X-1)$
$W_{gh}(H-1,0)$	$W_{gh}(H-1,1)$	...	$W_{gh}(H-1,H-1)$
$W_{ox}(H-1,0)$	$W_{ox}(H-1,1)$	...	$W_{ox}(H-1,X-1)$
$W_{oh}(H-1,0)$	$W_{oh}(H-1,1)$	...	$W_{oh}(H-1,H-1)$

Fig. 6 W_{**} stored in the interleaving order

The above strategies go to two different extremes, one uses only off-chip memories, the other one uses only on-chip memories, both leading to an imbalance between on-chip memories and bandwidth. On the contrary, the buffering strategy adopted in the proposed design balances them in the following way.

i. x_t is buffered with on-chip memories, because it requires a large bandwidth to be transferred but a small space to be stored with. To be more specific, if x_t is buffered with one on-chip memory whose size is X elements, it could save a bandwidth of $X \times H \times 4$ element/round.
ii. h_{t-1} is similar to x_t, but in order to keep h_{t-1} for current round and collect h_t for next round, it is stored with ping-pong memories.
iii. thanks to the interleaving work flow, the lifecycle of f_t, i_t, g_t, o_t are greatly shortened as described in Sect. 3.2. In another word, neither bandwidth nor on-chip memory is needed for them;
iv. c_{t-1} is stored with just one piece of memory instead of ping-pong memories, because after any certain element of c_t is figured out, the corresponding element of c_{t-1} would be out of its lifecycle;
v. W_{**} is stored with off-chip memories since they are too costly to be stored with on-chip memories. To better utilize the burst transfer of bus and off-chip memories, they should be stored in the interleaving order as they are read, which is shown in Fig. 6. In this figure, $W_{**}(x, y)$ stands for the element in row x, column y.
vi. b_* is stored with on-chip memories to keep the simplicity of the data flow.

Table 2 Comparison of different buffering strategy

Strategy	On-chip memory	Bandwidth
a) use only off-chip memory	0	7687.03 Mbps
b) use only on-chip memory	128.2 Mb	0.94 Mbps
the proposed one	0.13 Mb	3840.94 Mbps

It is assumed that * this layer runs at a speed of 30 round/second and has a size of 1024×1024 inputs; ** data width of each element is 16 bits.

In this way, the buffering strategy adopted requires on-chip memories of $X + 2 \times H + H + 4 \times H$ elements and a bandwidth of $4 \times X \times H + 4 \times H \times H + X + H$ element per round. An example is given in Table 2 to help the comparison, according to which, the buffering strategy adopted saves 50% bandwidth at a cost of merely 0.13 Mb on-chip memories compared with strategy a).

3.4.2 Compression Strategy

To further improve the bandwidth, it is recommended to do some compression to W_{**}. Two strategies could be adopted. One is using some asymmetrical algorithms, because W_{**} are usually trained off-line. In other words, they are known values in general situation. By word "asymmetrical", it means the calculation cost of compression and de-compression should be different. Of course, for the current design, algorithms which are difficult to compress while easy to de-compress should be used. To be more specific, the interleaved W_{**} could be compressed as a whole, stored to off-chip memories, then de-compressed in fetch module. In other words, no changes in gating part or network part are needed. Due to this reason, this part is not integrated and designers should choose suitable compression algorithm according to the system requirements.

The other one is using sparse W_{**}. By word "sparse", it means only a few elements in W_{**} are kept and the rest are set to zero. In this way, only non-zero elements need to be calculated. To handle this kind of compression, the proposed design need to be re-structured in the following aspects.

i. each W_{**} and its index would be stored in pairs.
ii. memories for x_t and h_t would be increased according to the parallelism.
iii. One FIFO would be added between the gate part and network part.

For example, if the first row of the sparse W_{fx} has J non-zero elements whose indexes are a, b, c, ... and the first row of the sparse W_{fh} has K non-zero elements whose indexes are l, m, n, ..., it should be stored in the way described by Fig. 7. If the parallelism is 3, then 3 piece of $1 \times H$ SRAM are used to store h_{t-1}, 3 piece of $1 \times X$ SRAMs are used to store x_t. Indexes like a, b, c or l, m, n would be used as the address to access those SRAMs as described in Fig. 8.

As to the FIFO added between the gate part and network part, it is used to balance the throughput of gating

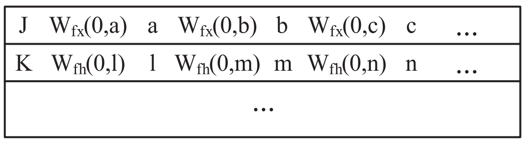

J W_{fx}(0,a) a W_{fx}(0,b) b W_{fx}(0,c) c …
K W_{fh}(0,l) l W_{fh}(0,m) m W_{fh}(0,n) n …
…

Fig. 7 Sparse W_{**} stored with column index

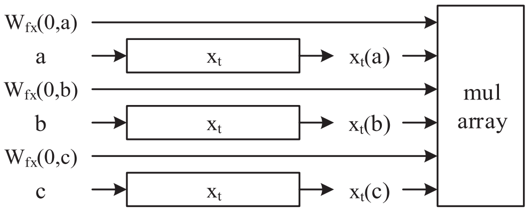


Fig. 8 Access of x_t

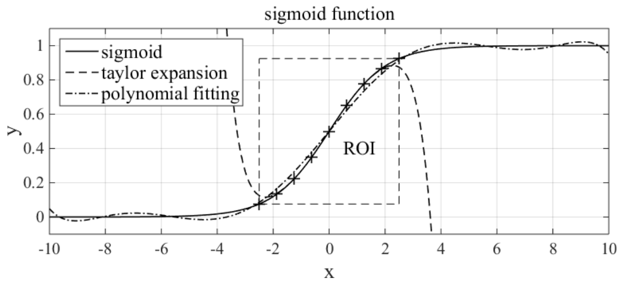


Fig. 9 Sigmoid function

part and network part because the former one is no longer regular.

3.5 σ and *tanh* Function, Data Type and Accuracy

3.5.1 σ and tanh Function

σ and *tanh* function are costly to be directly implemented on hardware platform. Taking σ function as the example, it contains exponent calculations as shown in Eq. (8).

$$\sigma(x) = 1/(1 + \exp(-x)) \tag{8}$$

Thus two different methods could be used to implement this function directly on hardware platform. One is by calculation, both Taylor expansion and polynomial fitting with an order of 8 have been tried to substitute this function. However, as shown in Fig. 9, Taylor expansion diverges quickly when leaving the expansion point; while the error of polynomial fitting is rather too high.

The other one is by look-up table. However, as mentioned above, a fixed-point type with 16-bit data width is used in this design. In other words, the look-up table would be as large as 1Mb, which is too costly.

In this design, to balance the hardware cost and accuracy, σ function is divided into 2 regions as described in Fig. 9. Region in the dashed box is called the region of interest (ROI). ROI would be further divided into several sub-regions and each sub-region would be fitted with an individual second-order function. As to the other regions, σ function returns with 0 or 1 directly. Attention should be paid on the fact that since the σ function is centrosymmetric, only half of it needs to be implemented.

An example of such an implementation is given as follows. In this example, it is assumed that the input data has a type of Q10.6, which covers a range of $[-512, 512 - 2^{-6}]$. ROI is set to $(-8, 8)$, then for the input data in the range of $[8, 512 - 1/2^6]$, it returns with 1; for the input data in the range of $[-512, -8]$, it returns with 0.

To simplify the structure, each sub-region is equal-length and the length is a power of 2 times the value of LSB, namely, 2^{-6}. In this way, if the input data is in ROI, the higher bits could be directly used as the address to read the fitting coefficients; while the lower bits could be directly used as the input of the fitting function. For example, if each sub-region has a length of 1, bit [8:6] would be used as the address; while bit [5:0] would be used as the input. Figure 10 has described the above architecture in a more visual way.

Since *tanh* function could be implemented in the same way and the data rate in network part is relatively low as suggested by Fig. 2, these two functions are reused in the time-division-multiplexing (TDM) way.

Also as illustrated by Fig. 10, the second-order function fitted out is implemented by the iteration of a one-order function engine. In other words, this implementation could be configured on the fly to realize higher-order fittings.

In addition, both ROI and the number/length of sub-regions are configurable to be adapted to different Q values. Since each sub-region is equal-length and the length is a power of 2 times the value of LSB, this could be easily realized by two selectors. One is used to extract the bits to be served as address of fitting coefficient; the other is used to extract the bits to be served as inputs of fitting function.

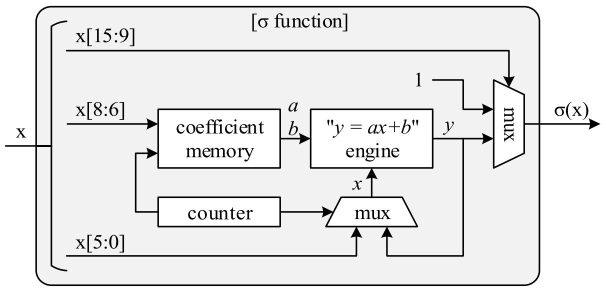


Fig. 10 Implementation of Sigmoid function (x ≥ 0)

Table 3 Accuracy of the proposed design

Size	Test Amount	Floating Ver.	Optimized Ver.
96	6345	97.57%	97.56%
256	4167	95.68%	95.66%
512	10378	98.08%	98.08%
1024	19815	96.09%	96.09%

3.5.2 Data Type and Accuracy

As mentioned in Sect. 3.3.2, a fixed-point type with 16-bit data width is adopted in this design to reduce the hardware cost of calculation; while the Q value is adopted to expand the data range.

A floating-type LSTM with no optimization in speed is used to evaluate the performance of the data type, σ and *tanh* function. According to Table 3, no obvious accuracy loss could be detected for the proposed optimization.

4. Comparison

4.1 Frequency, Hardware Cost and Power

Thanks to the optimization in work flow, structure and data type, this design achieves the highest working frequency, the smallest hardware cost and the lowest power among those designs listed in Table 4, where, the best results of each comparison item is highlighted with bold italic.

To be more specific, based on the function partition, the proposed design clearly divide the LSTM into two different parts, the gating part whose calculation is simple but data rate is high and the network part whose data rate is low but the calculation is complex. In this way, the calculation in the gating part, namely, matrix operation, only involves adding and multiplying; while the calculation in the networking part, namely, the σ and *tanh* function, is also simplified into iterations of multiplying and adding. Moreover, those two parts all benefit from the Q value adopted, which involves only fixed-type calculations but keep the data range large. As a result, compared with other designs, this work's frequency could be optimized to a great extent.

In addition to the simplicity of calculation and the data type with Q value, the hardware cost and power also benefit from (1) the interleaving work flow which saves the on-chip memory for f_t, i_t, g_t and o_t; (2) the ROI region (and centrosymmetric) adopted for σ and *tanh* function which avoids the use of the large lookup table or complex calculation; (3) the TDM reuse of the σ and *tanh* which further reduced the cost and power.

4.2 Throughput and Normalized Throughput

In this paper, throughput is compared using OP/s, which

Table 4 Comparison in frequency, hardware cost, power, throughput, time and error rate

Work	Frequency (F)	Hardware Cost (C)	Power (P)	Through put (Th)	Th/C (MOP/s/LUT)	Th/P (MOP/s/W)	Targeted Network Size (S)	Time (Ti)	Ti/S (us/element)	Error Rate *
[9]	142 MHz @ ZC7020	12960 FF 7201 LUT ***16 BRAM*** 50 DSP48	1.942 W	264.3 MOP/s	0.0367	136.09	2×128×65	0.932 s	56.01	***C 3.9%*** ***H 2.8%***
[10]	142 MHz @ ZC7020	61834 25567 16093 13598 LUT	/ / 1.8 / W	/ / 193.19 / MOP/s	/ / 0.012 /	454.28 400.98 107.33 145.90	2×128×50	/	/	***C 3.9%*** ***H 2.8%***
[11]	140 MHz @ ZC7020	14215 FF 15423 LUT 80 DSP	1.52 W	4.53 MOP/s	0.29	2.98	2×16	2.052 us	0.064	/
[12]	150 MHz @ VC707	182646 FF 198280 LUT 1072 BRAM 1176 DSP	19.63 W	7.26 GOP/s	0.036	375	61×123	0.39 s	5198	/
[18]	200 MHz @ XCKU 060	453068 FF 293920 LUT 947 BRAM 1504 DSP	41 W	***282 GOP/s***	0.96	6878	1024×153 +1024×512	82.7 us	***0.00012***	20.7%
[19]	190 MHz @ KCU200	41% FF 89% LUT 56% BRAM 55% DSP	7.0 W	/	/	/	1×1024	***1.33 us***	*0.00130*	/
[20]	120 MHz @ 10AX115 N2F45E1 SG	8610 FF 4043 LUT 16 BRAM 45 DSP	1.847 W	/	/	/	/	/	/	/
this work	***238 MHz*** @ XCZU 6EG	***1703 FF*** ***3092 LUT*** ***16 BRAM*** ***19 DSP***	0.885 W	7.62 GOP/s	***2.50***	***8610.2***	1024×1024	1.1 ms	***0.00105***	***3.9%***

* in column "Error Rate", H stands for output vector, namely h_t, C stands for cell state, namely c_t

stands for the (multiply) operation per second. For the proposed structure, if a parallelism of 32 is adopted, the throughput could be easily estimated by 32 (OP) times 238 (MHz), namely 7.62 GOP/s. This throughput is the highest one among those designs listed in Table 4 except S. Han et al. [18], and the throughput achieved by S. Han et al. [18] comes from a sparse W_{**}, which is not recommended from this paper's point of view.

First, although the accuracy loss is just 0.3% according to S. Han et al.'s paper [18], it may not be so convincing, because the original error rate which uses dense W_{**} is as high as 20.4%. In other words, the accuracy loss of sparse W_{**} for high accuracy vectors are remained to be checked.

Second, the control of data flow is quite trivial because the number of non-zero elements in each row is irregular. In other words, the design is too closely coupled with the feature of W_{**}, which may lower the configurability in parallelism and layer size. Nevertheless, the proposed design could be also restructured to handle sparse W_{**} as described in Sect. 3.4.2.

In order to give a fairer comparison, the throughput is normalized to hardware cost and power separately. Figure 11 (a) and column "Th/C" in Table 4 give the throughput over hardware cost, namely, the throughput offered by each LUT; Fig. 11 (b) and column "Th/P" in Table 4 give the throughput over power, namely, the throughput offered by each watt. According to them, the proposed design offers the highest value for both normalizations, which benefits a lot from the high parallelism & throughput and the low cost & power.

4.3 Time and Normalized Time

Although, J. Liu et al. [10] achieves the shortest time among those designs, it targets to a small network. As a result, a normalized time is given to help the comparison. Figure 11 (c)–(d) and Column "Ti/S" in Table 4 gives the time over network size, namely, the time to process each element. According to them, S. Han et al. [18] achieves the shortest time at a great cost due to the sparse W_{**} adopted. But it must be pointed out again that this may be suitable for low accuracy situations only and the configurability may be limited. While for dense W_{**}, the proposed design offers the shortest time.

4.4 Configurability

Among those designs, only the proposed one could be configured in not only the size and data range but also the parallelism and σ and tanh function, which is listed in Table 5. Again the best results of each comparison item is highlighted with bold italic.

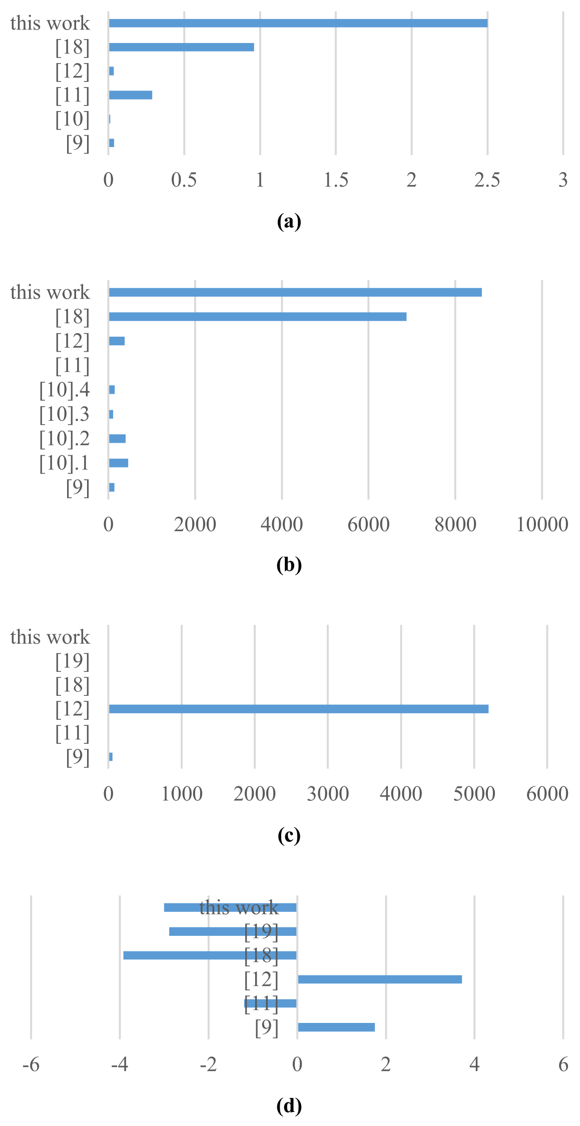


Fig. 11 (a) Throughput vs power (MOP/s/W). (b) Throughput vs cost (MOP/s/LUT). (c) Time vs size (us/element). (d) Time vs size in log10 (us/element in log10)

Table 5 Comparison in configurability

Work	Size	Data range	Parallelism	σ and tanh function
[9]	N (128×65)	N (Q8.8)	N	N (y = ax+b)
[10]	N (128×50)	N (Q8.8)	***Y***	/
[11]	Y (4/8/16/32/64/128)	N (Q6.11)	***Y***	N (y = ax^2+bx+c)
[12]	N (61×123)	N	N	N (y = ax+b)
[18]	N (1024×153) (1024×512)	N (16bit)	N	N (look-up table)
[13]	Y (32/64/128)	N	N	/
[14]	Y (64/128)	N	N	N (y = ax+b)
this work	***Y (arbitrary)***	***Y (Qx.16-x)***	***Y***	***Y (ROI) (sub-section) (order)***

i. Layer size: Instead of a fixed or discrete layer sizes, the proposed one supports arbitrary ones, and it could be configured on the fly. (Of course, the maximum layer size is limited by the on-chip memories for x_t and h_t.)
ii. Data type: Although a 16-bit fixed-point data type is adopted in this design, the Q value is adopted to change data range of W_{**}, x_t, h_t, … on the fly.
iii. Parallelism: Although the highest parallelism and the corresponding hardware cost is pre-determined by system requirements, it could be configured to a lower one on the fly, which is meaningful under some low-power mode.
vi. σ and tanh function: the ROI, the number and length of sub-sections, even the order of fitting functions could be configured on the fly.

5. Conclusion

This paper proposed a highly configurable 7.62 GOP/s hardware implementation for LSTM. The work flow is rearranged to make the design compact and high-throughput; the structure is organized to make the design configurable; the data buffering and compression strategy is chosen to lower the bandwidth without increasing the complexity of structure; the data type, σ and tanh function are optimized to balance the hardware cost and accuracy. This work achieves 7.62 GOP/s @ 238 MHz on XCZU6EG FPGA, which takes only 3K LUT.

Acknowledgments

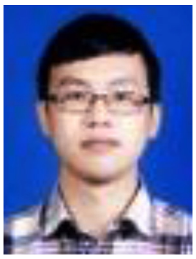
This work was supported in part by the National Natural Science Foundation of China under Grant 61674041, in part by the STCSM under Grant 16XD1400300, in part by the State Key Lab of ASIC & System under Grant 2015MS006.

References

[1] M. Sundermeyer, R. Schlüter, and H. Ney, "LSTM neural networks for language modeling," Thirteenth Annual Conference of the International Speech Communication Association, 2012.
[2] A. Graves, N. Jaitly, and A.-R. Mohamed, "Hybrid speech recognition with deep bidirectional lstm," 2013 IEEE Workshop on Automatic Speech Recognition and Understanding (ASRU), IEEE, pp.273–278, 2013.
[3] J. Cheng, P.-S. Wang, G. Li, Q.-H. Hu, and H.-Q. Lu, "Recent advances in efficient computation of deep convolutional neural networks," Frontiers of Information Technology & Electronic Engineering, vol.19, no.1, pp.64–77, 2018.
[4] S. Hochreiter and J. Schmidhuber, "Long short-term memory," Neural Computation, vol.9, no.8, pp.1735–1780, 1997.
[5] F.A. Gers and J. Schmidhuber, "Recurrent nets that time and count," Proc. IEEE-INNS-ENNS International Joint Conference on Neural Network (IJCNN), vol.III, pp.189–194, IEEE, 2000.
[6] F.A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," Proc. International Conference on Artificial Neural Networks (ICANN), vol.II, pp.850–855, 1999.
[7] K. Greff, R.K. Srivastava, J. Koutnik, B.R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," IEEE Trans. Neural Netw. Learn. Syst., vol.28, no.10, pp.2222–2232, 2017.
[8] J.T. Lo and D. Bassu, "Adaptive multilayer perceptrons with long- and short-term memories," IEEE Trans. Neural Netw., vol.13, no.1, pp.22–33, Jan. 2002.
[9] A.X.M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," Computer Science, 2015.
[10] A.X.M. Chang and E. Culurciello, "Hardware accelerators for recurrent neural networks on fpga," Proc. IEEE International Symposium on Circuits and Systems (ISCAS), IEEE, pp.1–4, 2017.
[11] J.C. Ferreira and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," Proc. IEEE International Conference on Reconfigurable Computing and FPGAs (ReConFig), pp.1–8, 2017.
[12] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "Fpga-based accelerator for long short-term memory recurrent neural networks," Proc. 22nd Asia and South Pacific Design Automation Conference (ASPDAC), IEEE, pp.629–634, 2017.
[13] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li, and X. Zhou, "A Power-Efficient Accelerator Based on FPGAs for LSTM Network," 2017 IEEE International Conference on Cluster Computing (CLUSTER), Honolulu, HI, pp.629–630, 2017.
[14] Y. Zhang, C. Wang, L. Gong, Y. Lu, F. Sun, C. Xu, X. Li, and X. Zhou, "Implementation and Optimization of the Accelerator Based on FPGA Hardware for LSTM Network," 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), Guangzhou, pp.614–621, 2017.
[15] Z. Wang, J. Lin, and Z. Wang, "Hardware-Oriented Compression of Long Short-Term Memory for Efficient Inference," IEEE Signal Process. Lett., vol.25, no.7, pp.984–988, July 2018.
[16] Z. Wang, J. Lin, and Z. Wang, "Accelerating Recurrent Neural Networks: A Memory-Efficient Approach," IEEE Trans. Very Large Scale Integr. (VLSI) Syst., vol.25, no.10, pp.2763–2775, Oct. 2017.
[17] J. Park, J. Kung, W. Yi, and J. Kim, "Maximizing system performance by balancing computation loads in LSTM accelerators," 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, pp.7–12, 2018.
[18] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, and Y. Wang, "Ese: Efficient speech recognition engine with sparse LSTM on FPGA," Proc. International Symposium on Field-Programmable Gate Arrays (FPGA), ACM, pp.75–84, 2017.
[19] J. Liu, J. Wang, Y. Zhou, and F. Liu, "A cloud server oriented FPGA accelerator for LSTM recurrent neural network," IEEE Access, vol.7, pp.122408–122418, 2019.
[20] K. Khalil, O. Eldash, A. Kumar, and M. Bayoumi, "Economic LSTM approach for recurrent neural networks," IEEE Trans. Circuits Syst. II, Exp. Briefs, vol.66, no.11, pp.1885–1889, Nov. 2019.

Yibo Fan received the B.E. degree in electronics and engineering from Zhejiang University, China in 2003, M.S. degree in microelectronics from Fudan University, China in 2006, and Ph.D. degree in engineering from Waseda University, Japan in 2009. From 2009 to 2010, he worked as an assistant professor in Shanghai Jiaotong University, and currently, he is the assistant professor in the college of microelectronics of Fudan University. His research interesting includes image processing, video coding and associated VLSI architecture.

Leilei Huang received the B.S. and M.S. degree in Microelectronics and Solid Electronics from Fudan University, Shanghai, China, in 2014 and 2017. He is currently pursuing toward the Ph.D. degree in Microelectronics and Solid Electronics from Fudan University. His research interests include VLSI design, algorithms and corresponding VLSI architectures for multimedia signal processing.



Kewei Chen received the B.S. degree in microelectronics from Fudan University, Shanghai, China, in 2017. He is currently working towards the M.S. degree in microelectronics at Fudan University, Shanghai, China. His research interests include image processing algorithms and VLSI architectures for multimedia signal processing.

Xiaoyang Zeng received the B.S. degree from Xiangtan University, Xiangtan, China, in 1992, and the Ph.D. degree from Changchun Institute of Optics, Fine Mechanics, and Physics, Chinese Academy of Sciences, Changchun, China, in 2001. From 2001 to 2003, he was a Postdoctoral Researcher with Fudan University, Shanghai, China. Then, he joined the State Key Lab of ASIC and System, Fudan University, as an Associate Professor, where he is currently a Full Professor and the Director. His research interests include information security chip design, system-on-chip platforms, and VLSI implementation of digital signal processing and communication system.