C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs

Shuo Wang[1,+], Zhe Li[2,+], Caiwen Ding[2,+], Bo Yuan[3], Qinru Qiu[2], Yanzhi Wang[2] and Yun Liang[1,*]
[+]These authors contributed equally
[1]Center for Energy-Efficient Computing & Applications (CECA), School of EECS, Peking University, China
[2]Dept. of Electrical Engineering & Computer Science, Syracuse University, Syracuse, NY, USA
[3]Dept. of Electrical Engineering, City University of New York, NY, USA
[1]{shvowang,ericlyun}@pku.edu.cn, [2]{zli89,cading,qiqiu,ywang393}@syr.edu, [3]byuan@ccny.cuny.edu

ABSTRACT

Recently, significant accuracy improvement has been achieved for acoustic recognition systems by increasing the model size of Long Short-Term Memory (LSTM) networks. Unfortunately, the ever-increasing size of LSTM model leads to inefficient designs on FPGAs due to the limited on-chip resources. The previous work proposes to use a pruning based compression technique to reduce the model size and thus speedups the inference on FPGAs. However, the random nature of the pruning technique transforms the dense matrices of the model to highly unstructured sparse ones, which leads to unbalanced computation and irregular memory accesses and thus hurts the overall performance and energy efficiency.

In contrast, we propose to use a structured compression technique which could not only reduce the LSTM model size but also eliminate the irregularities of computation and memory accesses. This approach employs block-circulant instead of sparse matrices to compress weight matrices and reduces the storage requirement from $\mathcal{O}(k^2)$ to $\mathcal{O}(k)$. Fast Fourier Transform algorithm is utilized to further accelerate the inference by reducing the computational complexity from $\mathcal{O}(k^2)$ to $\mathcal{O}(k\log k)$. The datapath and activation functions are quantized as 16-bit to improve the resource utilization. More importantly, we propose a comprehensive framework called C-LSTM to automatically optimize and implement a wide range of LSTM variants on FPGAs. According to the experimental results, C-LSTM achieves up to 18.8X and 33.5X gains for performance and energy efficiency compared with the state-of-the-art LSTM implementation under the same experimental setup, and the accuracy degradation is very small.

KEYWORDS

FPGA; RNNs; LSTM; compression; block-circulant matrix; FFT

[*]Corresponding author.

ACM Reference Format:
Shuo Wang[1,+], Zhe Li[2,+], Caiwen Ding[2,+], Bo Yuan[3], Qinru Qiu[2], Yanzhi Wang[2] and Yun Liang[1,*]. 2018. C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs. In *FPGA'18: 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, February 25–27, 2018, Monterey, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3174243.3174253

1 INTRODUCTION

Recurrent neural networks (RNNs) represent an important class of neural networks that contain cycles to carry information across neurons while reading inputs. Long Short-Term Memory (LSTM), one of the most popular types of RNNs, achieves great success in the domains such as speech recognition, machine translation, scene analysis, etc. [25]. However, the significant recognition accuracy improvement comes at the cost of increased computational complexity of larger model size [9]. Therefore, customized hardware acceleration is increasingly important for LSTMs, as exemplified by recent works on employing GPUs [5, 17], FPGAs [13, 16] and ASICs [7] as accelerators to speedup LSTMs.

Among the numerous platforms, FPGA has emerged as a promising solution for hardware acceleration as it provides customized hardware performance with flexible reconfigurability. By creating dedicated pipelines, parallel processing units, customized bit width, and etc., application designers can accelerate many workloads by orders of magnitude using FPGAs [24]. More importantly, High-level Synthesis (HLS) has greatly lowered the programming hurdle of FPGAs and improved the productivity by raising the programming abstraction from tedious RTL to high-level languages such as C/C++ [4] and OpenCL [26].

While the benefits of FPGAs is clear, it is still challenging to design efficient designs for LSTMs on FPGAs mainly for two reasons. On one hand, the capacity of the FPGA on-chip memory (a few or tens of Mb on-chip memory) is usually not large enough to store all the weight matrices of a standard LSTM inference model (e.g. hundreds of Mb). Although the previous work ESE [13] proposes to use the parameter pruning based compression technique to compress the dense weight matrices in the LSTM model into sparse ones, the sparse matrices need extra storage and processing units to store and decode the indices of the non-zero data, respectively. The skewed distribution of the data is likely to cause unbalanced workloads among parallel compute units. Therefore, the benefits of unstructured model compression is diminished by the sparsity of weight matrices. On the other hand, the computational complexity among the operators of the LSTMs is highly skewed and the data dependencies between operator are complicated. So, it is difficult to evenly allocate computing resources under the FPGA resource constraints while guaranteeing the complex data dependencies.

In this work, we propose to compress the weight matrices in the LSTM inference model in a structured manner by using block-circulant matrix [22]. The circulant matrix is a square matrix, of which each row (column) vector is the circulant reformat of the row (column) vector. Any matrix could be transformed into a set of circulant submatrices aka block-circulant matrices. Therefore, by representing each block-circulant matrix with a vector, the storage requirement could be reduced from $\mathcal{O}(k^2)$ to $\mathcal{O}(k)$ if the block (vector) size is k. Since the compressed weight matrices are still dense, the block-circulant matrix based compression is amenable to hardware acceleration on FPGAs. In order to further speed up the computation of LSTMs, we propose to accelerate the most computation-intensive circulant convolution operator by applying Fast Fourier Transform (FFT) algorithm to reduce the computational complexity from $\mathcal{O}(k^2)$ to $\mathcal{O}(n\text{log}n)$.

After the model is compressed, we propose an automatic optimization and synthesis framework called C-LSTM to port efficient LSTM designs onto FPGAs. The framework is composed of model training and implementation flows. The former one is in charge of iteratively training the compressed LSTM model and exploring the trade-offs between compression ratio and prediction accuracy. As for the model implementation, it mainly consists of two parts which are (1) template generation and (2) automatic LSTM synthesis framework. For the former part, after analyzing a wide range of LSTM algorithms, we generalize a suite of LSTM primitive operators which is general enough to accommodate even the most complicated LSTM variant [25]. Then, a suite of highly optimized C/C++ templates of the primitive operators are manually generated by walking through a series of optimizations such as datapath and activation quantization, DFT-IDFT decoupling and etc. As for the latter part, the well-trained LSTM inference model is first analyzed and transformed into a directed acyclic dependency graph, where each node represents an operator and each edge indicates the associated data dependency between two operators. Secondly, we propose a specialized pipeline optimization algorithm considering both coarse-grained and fine-grained pipelining schemes to schedule the operators into appropriate stages. In the third step, we use an accurate performance and resource model to enable a fast design space exploration for optimal design parameters. Lastly, the scheduling results and optimization parameters are fed to code generator and backend toolchain as to implement the optimized LSTM accelerator design on FPGAs.

Overall, the contributions of this paper are listed as:

- We employ the block-circulant matrices based structured compression technique for LSTMs which largely reduces the computation complexity and memory footprint without incurring any computation and memory access irregularities. This method results in both compression and acceleration of the LSTM models.
- We develop a general LSTM optimization and synthesis framework C-LSTM to enable automatic and efficient implementations of a wide range of LSTM variants on FPGAs. The framework mainly consists of a suite of highly optimized C/C++ based templates of primitive operators and an automatic LSTM synthesis flow.
- We present efficient implementations of LSTMs which achieve up to 18.8X and 33.5X gains in performance and energy efficiency, respectively, compared with the state-of-the-art. The proposed implementations incur very small accuracy degradation.

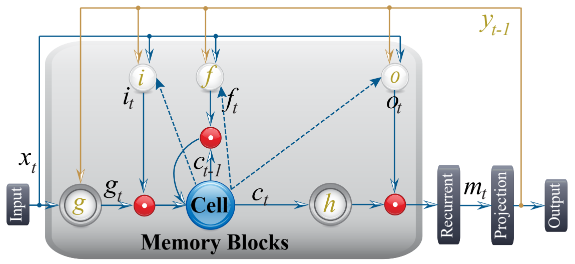


Figure 1: An LSTM based RNN architecture.

2 LSTM BACKGROUND

LSTM is a key component of the acoustic model in modern large-scale automatic speech recognition (ASR) systems [9, 25], and also the most computation and memory-intensive part. Due to the complicated and flexible data dependencies among gates, cells, and outputs, a lot of LSTM variants have been proposed. In this paper, we use a widely deployed variant called Google LSTM [25] as an example throughout this paper without loss of generality. The architecture details of the Google LSTM is shown in Figure 1. The LSTM accepts an input sequence $\mathbb{X} = (\mathbf{x}_1; \mathbf{x}_2; \mathbf{x}_3; ...; \mathbf{x}_T)$ (each of $\mathbf{x}_t$ is a vector corresponding to time t) with the output sequence from last step $\mathbb{Y}^{T-1} = (\mathbf{y}_0; \mathbf{y}_1; \mathbf{y}_2; ...; \mathbf{y}_{T-1})$ (each of $\mathbf{y}_t$ is a vector). The input of Google LSTM at time t depends on the output at $t-1$. The LSTM contains a special memory cell storing the temporal state of the network.It also contains three special multiplicative units which are input, output and forget gates. The output sequence $\mathbb{Y} = (\mathbf{y}_1; \mathbf{y}_2; \mathbf{y}_3; ...; \mathbf{y}_T)$ is computed by using the following equations iteratively from $t = 1$ to T:

$$\mathbf{i}_t = \sigma(\mathbf{W}_{ix}\mathbf{x}_t + \mathbf{W}_{ir}\mathbf{y}_{t-1} + \mathbf{W}_{ic}\mathbf{c}_{t-1} + \mathbf{b}_i), \tag{1a}$$
$$\mathbf{f}_t = \sigma(\mathbf{W}_{fx}\mathbf{x}_t + \mathbf{W}_{fr}\mathbf{y}_{t-1} + \mathbf{W}_{fc}\mathbf{c}_{t-1} + \mathbf{b}_f), \tag{1b}$$
$$\mathbf{g}_t = \sigma(\mathbf{W}_{cx}\mathbf{x}_t + \mathbf{W}_{cr}\mathbf{y}_{t-1} + \mathbf{b}_c), \tag{1c}$$
$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{g}_t \odot \mathbf{i}_t, \tag{1d}$$
$$\mathbf{o}_t = \sigma(\mathbf{W}_{ox}\mathbf{x}_t + \mathbf{W}_{or}\mathbf{y}_{t-1} + \mathbf{W}_{oc}\mathbf{c}_t + \mathbf{b}_o), \tag{1e}$$
$$\mathbf{m}_t = \mathbf{o}_t \odot \mathbf{h}(\mathbf{c}_t), \tag{1f}$$
$$\mathbf{y}_t = \mathbf{W}_{ym}\mathbf{m}_t, \tag{1g}$$

where symbols **i**, **f**, **o**, **c**, **m**, and **y** are respectively the input gate, forget gate, output gate, cell state, cell output, and a projected output; the $\odot$ operator denotes the element-wise multiplication, and the + operator denotes the element-wise addition. The **W** terms denote weight matrices (e.g. $\mathbf{W}_{ix}$ is the matrix of weights from the input vector $\mathbf{x}_t$ to the input gate), and the **b** terms denote bias vectors. Please note $\mathbf{W}_{ic}$, $\mathbf{W}_{fc}$, and $\mathbf{W}_{oc}$ are diagonal matrices for peephole connections, thus they are essentially a vector, and the matrix-vector multiplication like $\mathbf{W}_{ic}\mathbf{c}_{t-1}$ can be calculated by the $\odot$ operator. σ is the logistic activation function and h is a user-defined activation function. Here we use hyperbolic tangent (tanh) activation function as h. Overall, we have nine matrix-vector multiplications (excluding peephole connections which can be calculated by $\odot$). In one gate/cell, $\mathbf{W}_{*x}\mathbf{x}_t + \mathbf{W}_{*r}\mathbf{y}_{t-1}$ can be combined/fused in one matrix-vector multiplication by concatenating the matrix and vector as $\mathbf{W}_{*(xr)}[\mathbf{x}_t, \mathbf{y}_{t-1}]$.

3 STRUCTURED COMPRESSION

Deep neural networks (DNNs) bear a significant amount of redundancy [12] and thus model compression is a natural method to mitigate the computation and memory storage requirements for the hardware implementations on FPGAs. In this section, we propose to employ a structured compression technique to compress the weight matrices of LSTM model by using block-circulant matrices. We first introduce the block-circulant matrix and then integrate it with the inference and training algorithms of LSTMs. In the last, we explore the trade-offs between compression ratio and prediction error rate.

3.1 Block-Circulant Matrix

The circulant matrix is a square matrix whose each row (or column) vector is the circulant reformat of the row (or column) vectors [3, 22]. Any matrix could be transformed into a set of circulant submatrices (blocks) and we define the transformed matrix as a block-circulant matrix. For example, Figure 2 shows that the 8×4 weight matrix (on the left) is reformatted into a block-circulant matrix containing two 4×4 circulant matrices (on the right). Since each row vector of the circulant submatrix is a reformat of the first row vector, we could use a row vector to represent a circulant submatrix. Therefore, the first obvious benefit of the block-circulant matrix is that the number of parameters in each weight matrix is reduced by a factor of the block size $\mathcal{O}(k)$. As for the example in Figure 2, the 8×4 weight matrix (on the left) holding 32 parameters is reduced to two 4×4 circulant matrices (on the right) containing only 8 parameters, which easily leads to 4X model size reduction.

Intuitively, the model compression ratio is determined by the block size of the circulant submatrices: larger block size leads to higher compression ratio and vice versa. However, high compression ratio may degrade the prediction accuracy. Specifically, a larger block size should be selected to achieve a higher compression ratio but lower accuracy and the smaller block size provides higher accuracy but less compression ratio. The block size is 1 if no compression is utilized. It is necessary to note that block-circulant matrix based DNNs have been proved to asymptotically approach the original networks in accuracy with mathematical rigor [31]. Therefore, if the compression ratio is selected properly, the accuracy loss would be negligible. The trade-offs between compression ratio and predication accuracy are discussed in Section 3.3

3.2 Inference and Training Algorithms

The primary idea of introducing block-circulant matrix into LSTM model is to partition a $m \times n$ weight matrix $\mathbf{W}$ into $p \times q$ blocks, where $p = \frac{m}{k}$, $q = \frac{n}{k}$ and each block is a $k \times k$ circulant matrix. With bias and activation function omitted, the forward propagation

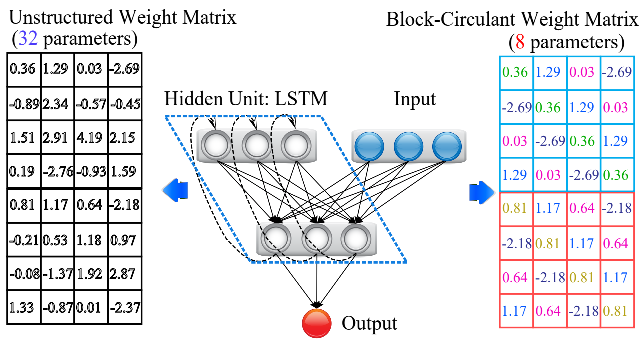


Figure 2: Block-circulant matrices for weight representation.

process of LSTM model in the inference phase is then given by:

$$\mathbf{a} = \mathbf{W}\mathbf{x} \iff \begin{bmatrix} \sum_{j=1}^{q} \mathbf{W}_{1j}\mathbf{x}_j \\ \sum_{j=1}^{q} \mathbf{W}_{2j}\mathbf{x}_j \\ \dots \\ \sum_{j=1}^{q} \mathbf{W}_{pj}\mathbf{x}_j \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1 \\ \mathbf{a}_2 \\ \dots \\ \mathbf{a}_p \end{bmatrix}, \tag{2}$$

where $\mathbf{a}_i$ is a column vector. Since each circulant matrix $\mathbf{W}_{ij}$ could be simplified as a vector $\mathbf{w}_{ij}$, i.e., $\mathbf{w}_{ij}$ is the first row vector of $\mathbf{W}_{ij}$, the structure of block-circulant matrix enables the use of Fast Fourier Transform (FFT) algorithm to speed up the circulant convolution $\sum_{j=1}^{q} \mathbf{W}_{ij}\mathbf{x}_j$. Therefore the Equation 2 can be performed as:

$$\mathbf{a}_i = \sum_{j=1}^{q} \mathcal{F}^{-1}[\mathcal{F}(\mathbf{w}_{ij}) \odot \mathcal{F}(\mathbf{x}_j)], \tag{3}$$

where $\mathcal{F}(\cdot)$ is the Discrete Fourier Transform (DFT) operator, $\mathcal{F}^{-1}(\cdot)$ is the inverse DFT (IDFT) operator, and $\odot$ is the element-wise multiply operator. Therefore, after applying FFT algorithm to the circulant convolution, the computational complexity of the LSTM inference model is reduced from $\mathcal{O}(pqk^2)$ to $\mathcal{O}(pqk \log k)$, meaning that the computational complexity of the LSTM inference model is reduced by a factor of $\mathcal{O}(\frac{k}{\log k})$.

The backward propagation process in the training phase can also be implemented using block-circulant matrices. Here we use a_{il} to denote the l-th output element in $\mathbf{a}_i$, and L to represent the loss function. Then by using the chain rule we can derive the backward propagation process as follows:

$$\frac{\partial L}{\partial \mathbf{w}_{ij}} = \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{w}_{ij}} = \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}, \tag{4}$$

$$\frac{\partial L}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \sum_{l=1}^{k} \frac{\partial L}{\partial a_{il}} \frac{\partial a_{il}}{\partial \mathbf{x}_j} = \sum_{i=1}^{p} \frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}. \tag{5}$$

where $\frac{\partial \mathbf{a}_i}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ are proved to be block-circulant matrices [31]. Thus, $\frac{\partial L}{\partial \mathbf{w}_{ij}}$ and $\frac{\partial L}{\partial \mathbf{a}_i} \frac{\partial \mathbf{a}_i}{\partial \mathbf{x}_j}$ can be calculated similarly as Equation (3) with the same computational complexity. The details of the training procedure for a fully-connected layer in DNNs are presented in [6, 27] and also applicable to the LSTM based RNNs.

Table 1: Comparison among different LSTM models.

Block Size	#Model Parameters	Computational Complexity	PER / PER Degradation (%)
1	8.01M	1	24.15 / 0.00
2	4.03M	0.50	24.09 / −0.06
4	2.04M	0.50	24.23 / 0.08
8	1.05M	0.39	24.57 / 0.32
16	0.55M	0.27	25.48 / 1.23

3.3 Compression and Accuracy Trade-offs

The block-circulant matrix based LSTM inference model enables a comprehensive tuning of model compression ratio by varying the block size k, thus leading to fine-grained trade-offs among the model size, computational complexity, and prediction accuracy. The proposed inference model of Google LSTM [25] is evaluated on the widely used TIMIT dataset [8]. Similar to [10], the audio data of TIMIT is preprocessed using a Fourier transform based filterbank with 50 coefficients (plus energy) distributed on a mel-scale, together with their first and second temporal derivatives. The number of features of the input speech and the architecture of Google LSTM used in this work is the same as ESE [13]. It is necessary to note that we use the widely adopted Phone Error Rate (PER) as the metric for the model prediction accuracy. The lower the PER value is, the higher the model prediction accuracy is and vice versa.

Table 1 presents the details of the trade-offs among three different metrics of Google LSTM using the block-circulant matrix based structured compression technique. We observe that the number of model parameters decreases linearly as the block size increases. Meanwhile, the PERs of different models do not have a severe degradation. For the block-circulant matrix based LSTM with block size of 2, the PER is even lower than non-compressed LSTM model whose block size is 1. For the LSTM models with block size of 8 and 16, we achieve 7.6X and 14.6X model size reduction and the computational complexity is reduced by factors of 2.6X and 3.7X while the PERs are only 0.32% and 1.23% higher than the non-compressed one, respectively. Therefore, we choose the compressed models of Google LSTM with block sizes of 8 and 16 to be further studied in this work.

4 FPGA ACCELERATION

In this section, we start by introducing a set of FPGA optimization techniques for circulant convolution operator and then apply quantizations to activation and element-wise operators. In the last, we propose an operator scheduling algorithm to generate the whole LSTM pipeline with the help of performance and resource models.

4.1 Circulant Convolution Optimization

Since the FFT based circulant convolution operator in the form of Equation 3 is the most computation-intensive operator in the LSTM inference model, we propose three techniques to further reduce the computational complexity by reducing the number of DFT and IDFT operator calls, and the redundant arithmetic operations of its complex number multiplication operators.

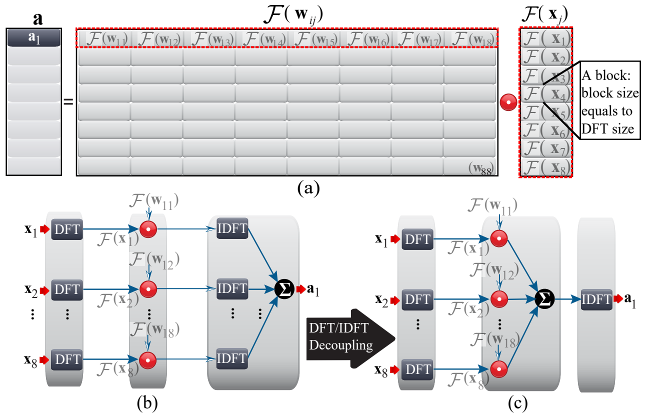

a
a1
=
F(w_ij)
F(w11) F(w12) F(w13) F(w14) F(w15) F(w16) F(w17) F(w18)
(w88)
F(x_j)
F(x1) F(x2) F(x3) F(x4) F(x5) F(x6) F(x7) F(x8)
A block: block size equals to DFT size
(a)
x1 DFT F(x1) F(w11) IDFT
x2 DFT F(x2) F(w12) IDFT
x8 DFT F(x8) F(w18) IDFT
Σ a1
(b)
DFT/IDFT Decoupling
(c)

Figure 3: An illustration of the (a) circulant convolution operator; (b) its original implementation; (c) and the optimized implementation.

In order to reduce the number of IDFT calls in the circulant convolution operator, we propose the DFT-IDFT decoupling technique. Since DFT and IDFT are linear operators [21], we could decouple the DFT and IDFT operators in Equation 3 and move the IDFT operator $\mathcal{F}^{-1}(\cdot)$ outside the accumulation operator $\sum$ as following,

$$\mathbf{a}_i = \mathcal{F}^{-1}\Big[\sum_{j=1}^{q} \mathcal{F}(\mathbf{w}_{ij}) \odot \mathcal{F}(\mathbf{x}_j)\Big], \tag{6}$$

where the number of IDFT operator calls for each circulant convolution operator is reduced from q to 1 and the numbers of the other operator calls are kept the same as before.

According to Equation 6, the number of DFT operator $\mathcal{F}(\cdot)$ calls in a circulant convolution operator is determined by q the number of weight vectors $\mathcal{F}(\mathbf{w}_{ij})$ and input vectors $\mathcal{F}(\mathbf{x}_j)$. Since the weight vectors $\mathbf{w}_{ij}$ are fixed when the training process is done, we could precalculate the $\mathcal{F}(\mathbf{w}_{ij})$ values and store them in the BRAM buffers of FPGAs and fetch the required values when needed instead of computing the associated DFT values at runtime. This method completely eliminates the DFT operator $\mathcal{F}(\cdot)$ calls for weight vectors and reduces the number of calls from $2qk$ to qk for each circulant convolution operator. The BRAM buffer size, however, would be doubled since the outputs of DFT values $\mathcal{F}(\mathbf{w}_{ij})$ are complex numbers whose both real and imaginary parts are needed to be stored. In order to alleviate the BRAM buffer overhead, we propose to exploit the complex conjugate symmetry property of DFT output values, where almost half of the conjugate complex numbers could be eliminated [21, 23]. Therefore, there is only negligible BRAM buffer overhead to store the DFT results of weight vectors $\mathcal{F}(\mathbf{w}_{ij})$.

The element-wise multiplication $\odot$ between two complex number vectors $\mathcal{F}(\mathbf{w}_{ij})$ and $\mathcal{F}(\mathbf{x}_j)$ requires $4k$ multiplications and $3k$ additions. Due to the complex conjugate symmetry property of DFT $\mathcal{F}(\cdot)$ results, about half of the multiplications and additions could be eliminated. Overall, Figure 3 illustrates the implementations of the original and optimized circulant convolution operators when the block size is 8.

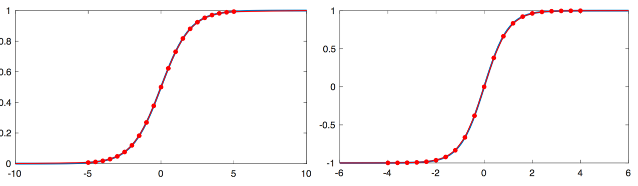

(a) sigmoid (b) tanh

Figure 4: Piece-wise linear activation functions.

4.2 Datapath and Activation Quantization

The LSTM model size could be further compressed without accuracy degradation if the datapath of LSTM implementation on FPGA is carefully quantized into shorter bitwidth. We design a bit-accurate software simulator to study the impact of the bitwidth of datapath on the prediction accuracy. We first analyze the numerical range of the trained weights in the LSTM, and then determine the bitwidth of integer and fractional parts to avoid data overflow and accuracy degradation. We observe that 16-bit fixed point is accurate enough for implementing the LSTM inference model on FPGAs.

In order to alleviate accuracy degradation problem caused by the data truncation and overflow problems in the architecture of the proposed circulant convolution operator. It is observed that the output data of IDFT are first divided by the block size (or IDFT input size) k, which is implemented as right shifting the numbers by $\log_2^k$ bits, and then output in the last stage of IDFT pipeline. However, the more bits are right shifted, the more fractional bits are truncated and thus degrading the overall accuracy. In order to deal with the accuracy loss caused by the data truncation, we propose to evenly distribute the shift operations inside the stages of the IDFT pipeline based on the observation that right shifting one bit at a time achieves better accuracy than right shifting multiple bits at once. As for the data overflow problem, it is most likely to occur in the accumulation stage of circulant convolution operator since multiple values are summed here. We propose to move the evenly distributed right shifting operations from stages of IDFT pipeline to the ones of DFT. Since the DFT is processed before accumulation operator and right shifting makes the number to be smaller and, it is less likely to cause overflow in accumulation stage.

The activation functions in LSTMs are all transcendental functions whose implementations on FPGA are very expensive with respect to resource utilization. In order to achieve a balance between accuracy and resource cost, we propose to utilize quantized piece-wise linear functions to approximate them. Figure 4 shows that the sigmoid and tanh functions are approximated using piece-wise linear functions with 22 segments. As we can see from the figure, the approximated and the original functions are almost the same and the error rate is less than 1%. Since the linear function could be represented in the slope-intercept form like $y = ax + b$, we only need to store the associated slope a and intercept b for each piece of linear function. In the real implementation, the computation complexity of activation functions only involves a simple comparison to index the associated pair of slope and intercept and one 16-bit fixed point multiplication followed by an addition. It is necessary to note that, according to our experimental results, the piece-wise linear approximation incurs negligible accuracy degradation for LSTMs.

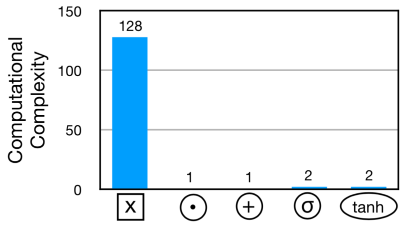


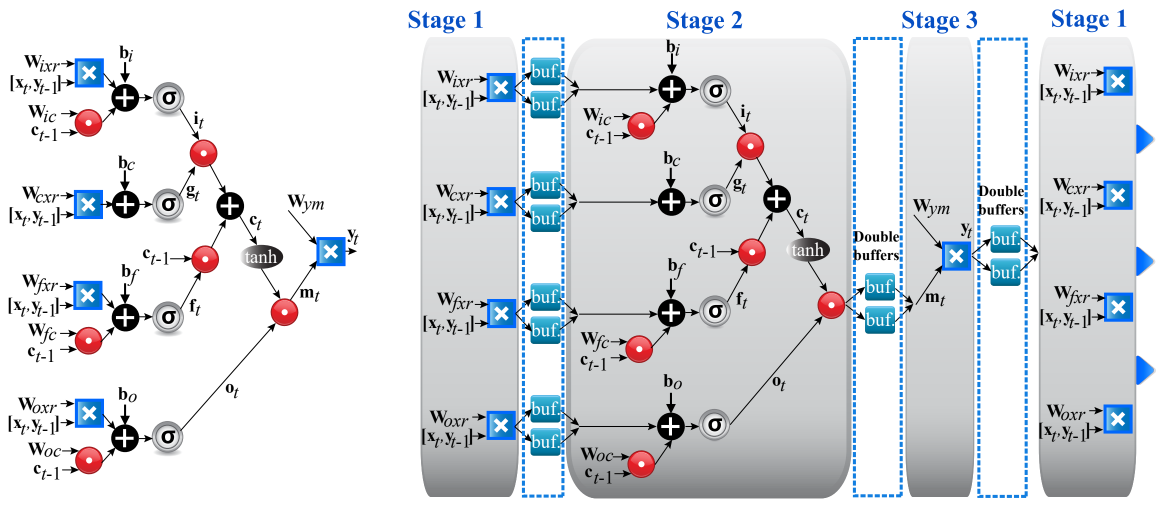
Figure 5: Computational complexity of LSTM operators.

4.3 Operator Scheduling

The recurrent nature of LSTM enforces strict data dependency among operators inside the LSTM module. In order to accommodate the complicated interactions of LSTM primitive operators, we propose a graph generator to transform the LSTM algorithm specification in the form of the equations like Equation 1 to a directed acyclic data dependency graph. Figure 6 (a) shows the generated LSTM directed operator graph from the LSTM descriptions, where each node is an LSTM primitive operator and the edge represents the data dependency between two operators. It is necessary to note that the generated graph is acyclic because we deliberately remove the feedback edges from cell output $\mathbf{c_t}$ to the LSTM module output $\mathbf{y_t}$. Since the backward edges are taken care of by the double-buffer mechanism, this practice would never harm the correctness and efficiency of the final LSTM accelerator design.

LSTMs exhibit a highly skewed distribution of computation complexity among the primitive operators. Figure 5 shows the normalized computational complexity of the five primitive operators of the Google LSTM [25] studied in this work. The computational complexity gap between the circulant convolution operator and element-wise multiply operator $\odot$ is as large as 128 times. So, if we want to pipeline these two operators we must either boost the parallelism of the former operator or make the latter operator wait (idle) for the former one. However, the reality is that the limited on-chip resources of FPGAs generally cannot sustain sufficient parallelism and the idle operators make the design inefficient. Therefore, pipelining a complex LSTM algorithm as a whole, such as the Google LSTM [25] shown in 6(a), is very inefficient on FPGAs.

In order to deal with this problem, we propose to break down the original single pipeline into several smaller coarse-grained pipelines and overlap their execution time by inserting double-buffers for each concatenated pipeline pair. For example, the original operator graph of Google LSTM [25] in 6(a) is divided into three stages in 6(b), where each stage will be implemented as a coarse-grained pipeline on FPGAs. The double-buffers added between stages are used to buffer the data produced/consumed by the previous/current stage. However, scheduling the operators to different stages in an efficient way is still a problem. We propose an operator scheduling algorithm shown in Algorithm 1 to tackle this problem. The algorithm takes the original operator graph $G = (V, E)$, operator weight set $W(V)$, and operator priority set $P(V)$ as input and outputs several operator subgraphs G_k. For original operator graph $G = (V, E)$, each vertex $v_i \in V$ represents an operator and the edge e_{ij} represents the data dependency between v_i and v_j. Each vertex v_i has a weight $W(w_i)$ which is the associated arithmetic computational complexity. The algorithm first traverses down the graph from the source vertex computing the priority of each vertex by



(a) Original operator graph **(b)** Scheduled operator graph

Figure 6: Illustration of operator scheduling on data dependency graph. The circle represents the element-wise operator, and the square represents the circulant convolution operator.

$$P(v_i) = \begin{cases} W(v_i) + \max\limits_{v_j \in Succ(v_i)} P(v_j), & v_i \neq v_{sink} \\ W(v_{sink}), & \text{otherwise} \end{cases} \tag{7}$$

Since $P(v_i)$ is accumulated with the maximum value of successors $P(v_j)$ as shown in Equation 7, priority set $P(V)$ is topologically ordered, which means that it is guaranteed that all predecessor operators are scheduled before scheduling a new operator [15]. After the prioritization, the algorithm selects the operator with the highest priority value and then determines the parallelism of the operator $N(v_j)$ and whether it should be added to the current or a new stage according to the resource utilization of FPGAs. Then, the operator subgraphs G_k and the operator parallelism set $N(V)$ are output by this algorithm, where each stage represents a corresponding LSTM execution stage that will be implemented as a coarse-grained pipeline on FPGAs. Since the overall throughput of this coarse-grained pipeline design is constrained by the slowest stage, we need to further determine the pipeline replication factor $R(G_k)$ for each stage. To fully utilize the resources of a certain FPGA chip, we also need to take into account of the resource utilization of each stage, and thus we propose to enumerate pipeline replication factor $R(G_k)$ to get the optimal setting with the help of our analytical performance and resource models which are presented in Section 4.4.

4.4 Performance and Resource Models

Since the throughput of the proposed coarse-grained pipeline design is constrained by the slowest stage, the analytical performance model is built as following,

$$FPS = \frac{Frequency}{\max\{T_1, T_2, ..., T_h, ...T_K\}}, \tag{8}$$

where FPS is the number of frames per second of C-LSTM accelerator, T_k represents the number of execution clock cycles of stage k,

Algorithm 1: Operator Scheduling Algorithm

```
Input: operator graph $G = (V, E)$, operator weight set $W(V)$, and priority set $P(V)$;
Output: operator subgraph of each stage $G_k = (V_k, E_k)$;
Traverse $G = (V, E)$ and compute priority set $P(V)$;
$k \leftarrow 0$, $N(V) \leftarrow \{1\}$;
foreach $v_i \in V$ in decreasing order of $P(v)$ do
    if $k = 0$ then
        $k \leftarrow k + 1$;
        $G_k \leftarrow v$; // add the operator to a new stage
    else
        foreach $N'(v_j) \in G_k$ do
            $N'(v_j) \leftarrow N(v_j) \cdot \lceil \frac{W(v_j)}{W(v_i)} \rceil$;
        end
        if resource constraints are satisfied then
            $G_j \leftarrow v$; // add the operator to current stage
            $N(V) \leftarrow N'(V)$; // update operator parallelisms
        else
            $k \leftarrow k + 1$;
            $G_k \leftarrow v_i$; // add the operator to a new stage
        end
    end
end
$K \leftarrow k$;
Enumerate $R(G_k)$ values to maximize throughput and fully utilize FPGA resource;
return $N(V)$, {$G_1, G_2, ..., G_K$}, and {$R(G_1), R(G_2), ..., (G_K)$};
```

and K is the total number of stages. T_k is calculated by considering the parallelism and input data size of each stage as following,

$$T_k = \lceil \max_{v_i \in G_k} \frac{Q(v_i)}{N(v_i)} / R(G_k) \rceil + D_k \tag{9}$$

where $Q(v_i)$ is the workload of operator v_i and D_k is the pipeline depth of stage k. It is necessary to note that, the compression ratio of the block-circulant matrices based technique is large enough to store the whole LSTM model on BRAMs of FPGAs, and for each frame, we only need to load very limited size of input data which makes computation time of LSTM to be overlapped with data loading.

The resource model of the highly optimized primitive operator templates is very straightforward because the linear model with respect to the associated operator parallelism $N(v_i)$ and stage parallelism $R(G_k)$ is accurate enough to guide the design space exploration for energy-efficient designs. The models are shown in the following,

$$DSP = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta DSP(v_i) \cdot N(v_i), \tag{10}$$

$$BRAM = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta BRAM(v_i) \cdot N(v_i), \tag{11}$$

$$LUT = \sum_{k=1}^{K} R(G_k) \cdot \sum_{v_i \in V,} \Delta LUT(v_i) \cdot N(v_i), \tag{12}$$

where $\Delta DSP(v_i)$, $\Delta BRAM(v_i)$, and $\Delta LUT(v_i)$ are obtained by profiling the resource consumption values for operator v_i on the FPGA using the manually optimized operator template.

4.5 Putting It All Together

The final hardware architecture of the Google LSTM algorithm [25] is shown in Figure 7. This design mainly consists of three coarse-grained pipeline stages corresponding to the operator scheduling result shown in Figure 6(b). At Stage 1, the input vectors $\mathbf{x}_t$ and the prestored DFT values of weight matrices $\mathbf{W}$ are convolved using the circulant convolution operator whose output is written into the double-buffer. Since all the DFT values of weight matrices are compressed small enough, they could be stored in on-chip BRAM buffers instead of off-chip DDR memory. The performance of the circulant convolution operator is thus no longer bottlenecked by off-chip memory bandwidth and the parallel compute units could be fully exploited on FPGAs. In stage 2, the input data are first read from double-buffer of the previous stage and then processed by a series of element-wise operators including addition, multiplication and activation functions in the LSTM cell module. The output of Stage 2 is also written to double-buffer for the next stage. As for Stage 3, the results of the prior stage are fetched from double-buffer and are then projected to output using the circulant convolution operator. In the last, the projected output will be forwarded to Stage 1 for the next iteration.

5 C-LSTM FRAMEWORK

In order to embrace a wide range of LSTM architectures, we propose a comprehensive framework called C-LSTM to assist the LSTM model training using the block-circulant matrix based structured compression and enable an automatic flow to generate efficient LSTM inference designs on FPGAs. As shown in Figure 8, the C-LSTM framework is mainly composed of two parts which are LSTM model training and its implementation on FPGAs. The details of the C-LSTM framework are explained in the following sections.

5.1 Model Training

The model training, which is shown on the left side of Figure 8, accepts the LSTM architecture specifications in the form of Equation 1 as input. Then, the block-circulant matrix based structured compression is applied to the weight matrices of the model. In the following, TensorFlow [1] is used as the training framework to iteratively train the LSTM model. The trade-offs between compression ratio and prediction accuracy is explored in this procedure. In the

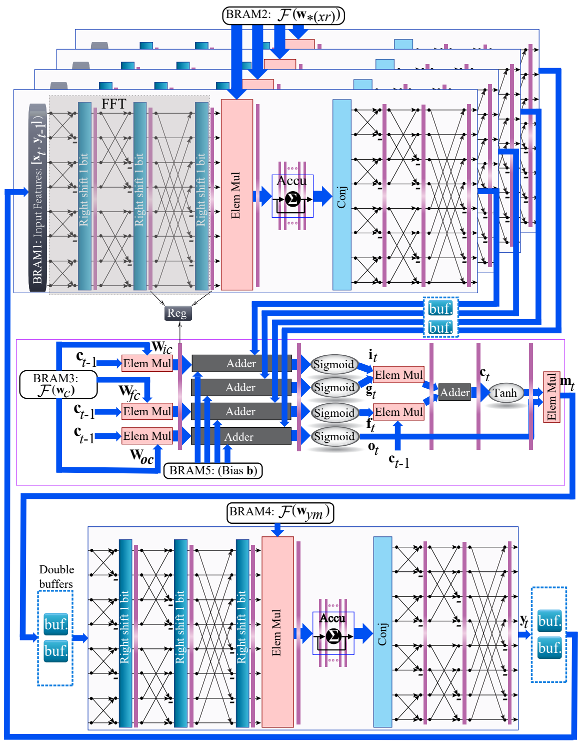


Figure 7: The proposed Google LSTM architecture.

last, the LSTM inference model is configured with the well-trained weight matrices and sent to model implementation flow for further acceleration on FPGAs.

5.2 Model Implementation

The model implementation of the C-LSTM framework is shown in the right side of Figure 8, It mainly consists of two parts which are operator templates generation (upper part) and automatic synthesis framework (lower part). Since the number of primitive operators of LSTMs is limited, we propose to manually write the template for each primitive operator. As for the LSTM algorithms studied in this work, we define hyperbolic tangent *tanh*, sigmoid σ, element-wise vector addition, element-wise vector multiplication, and circulant convolution as primitive operators. The optimization techniques presented in Section 4 are all applied to these operators. It is necessary to note that, the proposed primitive operator templates are general enough to implement almost any kind of LSTM variant to best of our knowledge.

The automatic synthesis framework is fed with the well-trained inference model provided by the model training flow. Then a directed acyclic data dependency graph is generated to represent the computation flow of LSTM. The operators in the graph are scheduled to compose a multi-stage coarse-grained pipeline as to maximize the performance under certain resource constraints with the help of analytical performance and resource models. The scheduling result is then given to the code generator. The code generator takes the operator scheduling result as input and generates the final C/C++ based code automatically by integrating the associated primitive operator templates together. Since the interface of

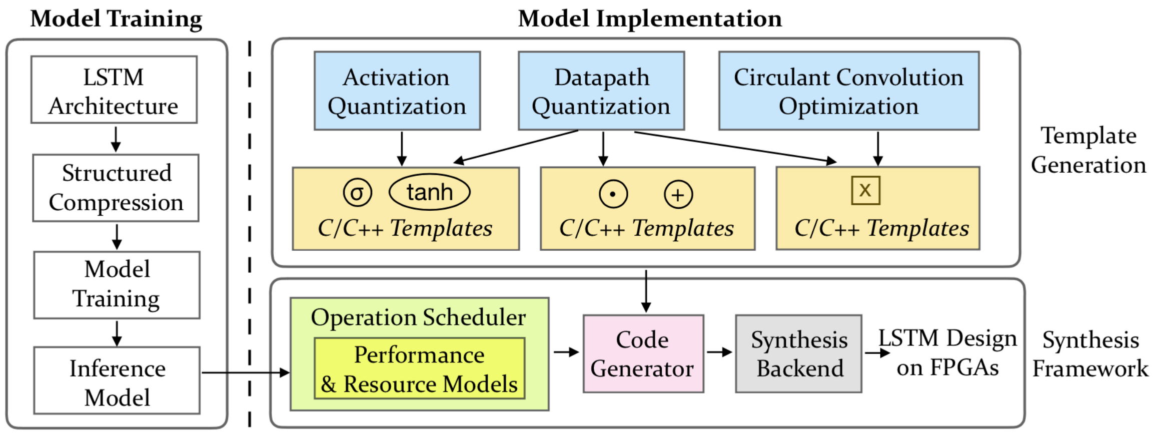


Figure 8: C-LSTM framework overview.

Table 2: Comparison of FPGA platforms

FPGA	DSP	BRAM	LUT	FF	Process
XCKU060	2,760	1,080	331,680	663,360	20nm
Virtex-7(690t)	3,600	1,470	859,200	429,600	28nm

each template is well defined and the tunable parameters are expressed using C/C++ marcos, the code generation is very efficient. The synthesis backend which is an off-the-shelf commercial HLS tool, accepts the C/C++ code as input and outputs the optimized LSTM hardware implementation on FPGAs. It is necessary to note that each commercial HLS toolchain requires specific coding style to achieve the best performance, and thus the templates of the primitive operators should be tailored accordingly [29].

6 EXPERIMENT EVALUATION

6.1 Experiment Setup

The proposed techniques for LSTMs are evaluated on two platforms: Xilinx KU060 and Alpha Data's ADM-7V3. The Xilinx KU060 platform consists of a Xilinx XCKU060 FPGA and two 4GB DDR3 memory. The ADM-7V3 board consists of a Xilinx Virtex-7 (690t) FPGA and a 16GB DDR3 memory. The comparison of the FPGA on-chip resources of the two platforms is presented in Table 2. The ADM-7V3 FPGA board is connected to the host via PCI-e 3.0 X8 interface, and the host machine is a server with Intel Core i7-4790 CPU. Xilinx SDx 2017.1 is used as the commercial synthesis backend to synthesize the C/C++ based LSTM design onto FPGAs. The proposed FPGA implementations of LSTMs are operating at 200MHz on both platforms.

We measure the latency of our C-LSTM designs on KU060 platform using the number of clock cycles times the clock period (5ns) reported by Xilinx SDx tools. To make a fair comparison with ESE [13], the latency of ESE reported in Table 3 is its theoretical time. Since we do not have the KU060 platform, we cannot give out an accurate estimation and the associated power and energy efficiency results are left blank. As for the ADM-7V3 platform, the execution time of C-LSTM designs are obtained by using Xilinx SDx runtime profiler, and the power is profiled using the TI Fusion Power device through the associated interface on ADM-7V3 with a sampling rate of 100Hz.

Besides the LSTM based RNN architecture used in [13, 25], we also evaluated the performance on a smaller LSTM model [20], where the input feature is a 39-dimension vector (12 filterbank coefficients plus energy and its first/second temporal derivatives), and the gate/cell layers' dimension is 512. In this small model, the peephole connection and projection layer are not employed. The model contains two stacked LSTM as well. However, we used bidirectional architecture [2, 10] to get a better PER.

In order to make a convincing validation for the superiority of the proposed C-LSTM optimization framework, we compare our design with the state-of-the-art LSTM design ESE [13].The same dataset, LSTM algorithm, and FPGA platforms are used in the associated experiments as ESE to make a fair comparison.

6.2 Experimental Results of Google LSTM

With the compression technique of C-LSTM, we are able to store all the weights matrices and the projection matrix in BRAM, after performing compression on the baseline. The baseline has the same structure as the baseline in ESE.

According to the results of latency and FPS in Table 3, we achieve 3.6X and 4.3X latency reduction and 11X and 13X performance speedup for FFT8 and FFT16 based compression techniques compared with ESE on the platform of KU060. It is necessary to note that the gap between latency reduction and performance speedup stems from the coarse-grained architecture of the proposed LSTM accelerator. And thus the latency of our proposed C-LSTM accelerator for Google LSTM algorithm is the latency of one stage multiplied by 3, because each input frame needs to go through three coarse-grained pipelines. However, after three frames have been processed, the following frame could be processed at every one stage of latency.

As we can see from Table 2, the resource of the FPGA chip Virtex-7 of the ADM-7V3 platform is 30% higher than the FPGA XCKU060 of KU060 platform. Therefore, to make a fair comparison, we use the total resource of KU060 as the resource consumption bound for the ADM-7v3 platform. Compared with ESE, we achieve 10.2X and 18.8X performance speedups and 19.1X and 33.5X energy efficiency

Table 3: Detailed comparison for different LSTM designs.

	ESE [13]	C-LSTM FFT8 (Block size: 8)		C-LSTM FFT16 (Block size: 16)		C-LSTM FFT8 (Block size: 8)		C-LSTM FFT16 (Block size: 16)	
LSTM Algorithm	Google LSTM [25]					Small LSTM [20]			
Weight Matrix Size (#Parameters of LSTM)	0.73M	0.41M		0.20M		0.28M		0.14M	
Quantization	12bit fixed	16bit fixed		16bit fixed		16bit fixed		16bit fixed	
Matrix Compression Ratio	4.5 : 1[1]	7.9 : 1		15.9 : 1		7.9 : 1		15.9 : 1	
Platform	KU060	KU060	7V3	KU060	7V3	KU060	7V3	KU060	7V3
DSP (%)	54.5	96.5	74.3	98.0	77.4	77.6	60.5	84.9	65.2
BRAM (%)	87.7	87.6	65.7	89.1	63.3	83.3	66.9	87.2	64.1
LUT (%)	88.6	75.2	58.7	72.8	55.3	92.5	67.6	93.6	72.3
FF (%)	68.3	58.9	46.5	63.4	48.1	61.2	49.0	70.7	54.6
Frequency (MHz)	200								
PER Degradation	0.30%	0.32%		1.23%		0.29%		1.16%	
Latency (μs)	57.0	15.4	16.7	8.1	9.1	8.9	9.8	4.8	5.4
Frames per Second (FPS)	17,544	195,313	179,687	371,095	330,275	337,838	307,432	628,379	559,257
Power (W)	41	-	22	-	23	-	21	-	22
Energy Efficiency (FPS/W)	428	-	8,168	-	14,359	-	14,640	-	25,420

[1] This estimation considers both weights and indices (there is at least one index per weight after compression in ESE). However, this is a pessimistic estimation for ESE because indices can use fewer bits for representation than weights;

gains using FFT8 and FFT16, respectively. Since the power consumption of C-LSTM is only half of the ESE, the energy efficiency gain is higher than performance. It is necessary to note that as shown in Table 2, the manufacturing process of XCKU060 FPGA is 20nm while the process of Virtex-7 is 28nm, which means the energy efficiency gain reported here is pessimistic.

Although the promising performance and energy gains are achieved by C-LSTM, the resource utilization for LUT, FF, and BRAM are less than ESE, and more important, the relative PER degradation is very small, which are 0.32% and 1.23% using FFT8 and FFT16, respectively. After detailed analysis, we summarize the fundamental reasons for the high performance and power gains in three aspects. First, the structured compression used in this work eliminates the irregular computation and memory accesses which not only makes the design more regular but also exposes more parallelism. This could be verified in that the DSP resource consumption of the proposed method is much more than ESE. Secondly, the whole model (weights matrices and the projection matrix) could be stored on-chip without fetching data from off-chip DRAM, making the LSTM not bounded by memory. Lastly, the more efficient implementation of LSTM on FPGAs contributes to the high efficiency. For example, we use the 22-segment piece-wise linear function to approximate the activation functions while ESE employs look-up tables which break the activation down into 2048 segments and consume more resources. Moreover, we propose to employ FFT based block-circulant matrix multiplication while ESE uses sparse matrix multiplication which needs to store extra indices for sparse matrices and thus prevents from storing the whole model on-chip.

6.3 Experimental Results of Small LSTM

In order to validate that proposed C-LSTM is not only appropriate for Google LSTM model, we also implement a Small LSTM [20] model on both FPGA platforms.

In KU060 platform, the FFT8 and FFT16 designs could achieve 19.3X and 35.9X performance speedup compared with ESE, respectively. In the ADM-7V3 platform, the performance speedups are 17.5X and 31.9X and the energy efficiency gains are 34.2X and 59.4X compared with ESE, respectively. For both platforms, the PER degradation is 0.29% and 1.16% for FFT8 and FFT16, respectively.

7 RELATED WORK

Recently, FPGA has emerged as a promising hardware acceleration platform for DNNs as it provides high performance, low power and reconfigurability. A lot of FPGA based accelerators have been proposed for convolutional neural networks (CNNs) to overcome the computing and energy efficiency challenges. [28] proposes to utilize systolic array based convolution architecture to achieve better frequency and thus performance for CNNs on FPGAs. [18] employs the Winograd algorithm to reduce the multiplication operators as to save DSP resources and accelerate matrix multiplication in CNNs. [30] proposes to take advantage of the heterogeneous algorithms to maximize the resource utilization for convolutional layers on FPGAs. Some studies also propose to transform the CNN models to frequency domains and then exploit FFT algorithms for further acceleration [14]. The FFT based acceleration scheme used in the CNN model is completely different from this work, in which we target on a totally different LSTM based RNN model and the FFT algorithm is applied to the circulant convolution operators instead of the convolution layers of CNNs.

There are also a lot of works on implementing RNN accelerators for FPGAs [11, 16, 19]. [19] designs an accelerator for the gated recurrent network (GRU) which embodies a different architecture from the LSTM based RNNs. [11] and [16] focus on LSTM based RNNs but none of these works utilize compression techniques to reduce the model size. The most relevant study to this work is ESE [13], which proposes a software and hardware co-design framework to accelerate compressed sparse LSTM model obtained by parameter pruning [12]. The performance and energy efficiency gains achieved by ESE is very promising compared with CPU and GPU based implementations. However, due to the irregular computation and memory accesses caused by the sparse weight matrices of the compressed model, the computing power of the FPGA is not fully exerted by ESE. In order to deal with this problem, this work proposes to employ a structured compression technique as to completely eliminate the irregularities of computation and memory accesses. Moreover, a suite of highly efficient optimization techniques is enabled by an automatic synthesis framework to generate LSTM accelerators with much higher performance and energy efficiency under the same conditions.

8 CONCLUSION

In this paper, we propose to employ a structured compression technique using block-circulant matrices to compress the LSTM model small enough to be fitted on BRAMs of FPGA. Besides the reduced model size, the irregular computation and memory accesses have been completely eliminated by the regular structure of the block-circulant matrices. Moreover, an efficient FFT based fast circulant convolution is applied to accelerate the LSTM computation by reducing both the computational and storage complexities. In order to accommodate a wide range of LSTM variants, we also propose an automatic optimization and synthesis framework. Overall, compared with the state-of-the-art LSTM implementation, the proposed C-LSTM designs generated by our framework achieve up to 18.8X and 33.5X gains for performance and energy efficiency with small accuracy degradation, respectively.

ACKNOWLEDGMENTS

This work is supported by Beijing Natural Science Foundation (No. L172004) and National Science Foundation under grants CNS #1704662 and CNS #1739748. We thank all the anonymous reviewers for their feedback.

REFERENCES

[1] Martín Abadi et al. 2016. Tensorflow: large-scale machine learning on heterogeneous distributed systems. *Arxiv preprint arxiv:1603.04467*.
[2] 2014. *Automatic speech recognition: A deep learning approach, author=Yu, Dong and Deng, Li*. Springer.
[3] Yu Cheng, Felix X Yu, Rogerio S Feris, Sanjiv Kumar, Alok Choudhary, and Shi-Fu Chang. 2015. An exploration of parameter redundancy in deep networks with circulant projections. In *ICCV*.
[4] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. 2011. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*.
[5] Zheng Cui, Yun Liang, Kyle Rupnow, and Deming Chen. 2012. An accurate gpu performance model for effective control flow divergence optimization. In *IPDPS*.
[6] Caiwen Ding et al. 2017. Circnn: accelerating and compressing deep neural networks using block-circulant weight matrices. In *MICRO*.
[7] Steven K Esser et al. 2016. Convolutional networks for fast, energy-efficient neuromorphic computing. *Proceedings of the national academy of sciences*.
[8] John S Garofolo, Lori F Lamel, William M Fisher, Jonathon G Fiscus, and David S Pallett. 1993. DARPA TIMIT acoustic-phonetic continous speech corpus CD-ROM. NIST speech disc 1-1.1. *Nasa sti/recon technical report n*, 93.
[9] Felix A Gers and Jürgen Schmidhuber. 2000. Recurrent nets that time and count. In *Proceedings of the IEEE-INNS-ENNS International Joint Conference on Neural Networks*.
[10] Alex Graves, Navdeep Jaitly, and Abdel-rahman Mohamed. 2013. Hybrid speech recognition with deep bidirectional LSTM. In *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*.
[11] Yijin Guan, Zhihang Yuan, Guangyu Sun, and Jason Cong. 2017. Fpga-based accelerator for long short-term memory recurrent neural networks. In *ASP-DAC*.
[12] Song Han, Huizi Mao, and William J Dally. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *Arxiv preprint arxiv:1510.00149*.
[13] Song Han et al. 2017. ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA. In *FPGA*.
[14] Jong Hwan Ko, Burhan Mudassar, Taesik Na, and Saibal Mukhopadhyay. 2017. Design of an Energy-Efficient Accelerator for Training of Convolutional Neural Networks Using Frequency-Domain Computation. In *DAC*.
[15] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. 2015. Orchestrating multiple data-parallel kernels on multiple devices. In *PACT*.
[16] Sicheng Li, Chunpeng Wu, Hai Li, Boxun Li, Yu Wang, and Qinru Qiu. 2015. Fpga acceleration of recurrent neural network based language model. In *FCCM*.
[17] Yun Liang, Huynh Phung Huynh, Kyle Rupnow, Rick Siow Mong Goh, and Deming Chen. 2015. Efficient gpu spatial-temporal multitasking. *TPDS*, 26, 3.
[18] Liqiang Lu, Yun Liang, Qingcheng Xiao, and Shengen Yan. [n. d.] Evaluating fast algorithms for convolutional neural networks on fpgas. In *FCCM*.
[19] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. 2016. Accelerating recurrent neural networks in analytics servers: comparison of fpga, cpu, gpu, and asic. In *FPL*.
[20] Christopher Olah. [n. d.] http://colah.github.io/posts/2015-08-Understanding-LSTMs. ().
[21] Alan V Oppenheim. 1999. *Discrete-time signal processing*. Pearson Education India.
[22] Victor Pan. 2012. *Structured matrices and polynomials: unified superfast algorithms*. Springer Science & Business Media.
[23] Anamitra Bardhan Roy, Debasmita Dey, Bidisha Mohanty, and Devmalya Banerjee. 2012. Comparison of FFT, DCT, DWT, WHT compression techniques on electrocardiogram and photoplethysmography signals. In *IJCA*.
[24] Kyle Rupnow, Yun Liang, Yinan Li, Dongbo Min, Minh Do, and Deming Chen. 2011. High level synthesis of stereo matching: productivity, performance, and software constraints. In *FPT*.
[25] Haşim Sak, Andrew Senior, and Françoise Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *Fifteenth annual conference of the international speech communication association*.
[26] Shuo Wang, Yun Liang, and Wei Zhang. 2017. FlexCL: An Analytical Performance Model for OpenCL Workloads on Flexible FPGAs. In *DAC*.
[27] Yanzhi Wang et al. 2018. Towards ultra-high performance and energy efficiency of deep learning systems: an algorithm-hardware co-optimization framework. In *AAAI*.
[28] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. 2017. Automated Systolic Array Architecture Synthesis for High Throughput CNN Inference on FPGAs. In *DAC*.
[29] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. 2017. Energy Efficient Scientific Computing on FPGAs Using OpenCL. In *FPGA*.
[30] Qingcheng Xiao, Yun Liang, Liqiang Lu, Shengen Yan, and Yu-Wing Tai. 2017. Exploring Heterogeneous Algorithms for Accelerating Deep Convolutional Neural Networks on FPGAs. In *DAC*.
[31] Liang Zhao, Siyu Liao, Yanzhi Wang, Jian Tang, and Bo Yuan. 2017. Theoretical Properties for Neural Networks with Weight Matrices of Low Displacement Rank. *Arxiv preprint arxiv:1703.00144*.