Efficient Weight Reuse for Large LSTMs

Zhiqiang Que*, Thomas Nugent*, Shuanglong Liu*, Li Tian‡, Xinyu Niu†, Yongxin Zhu‡, Wayne Luk*
*Dept. of Computing, School of Engineering, Imperial College London, UK
{*z.que, tn1115, s.liu13, w.luk*}*@imperial.ac.uk*
† Corerain Technologies Ltd., Shenzhen, China, *xinyu.niu@corerain.com*
‡ Shanghai Advanced Research Institute, Chinese Academy of Sciences, China, {*tianl, zhuyongxin*}*@sari.ac.cn*

***Abstract*—Long Short-Term Memory (LSTM) networks have been deployed in speech recognition, natural language processing and financial calculations in recent years, and are beginning to be used in systems where low latency and low power are required. To meet such requirements, we propose a stall-free hardware architecture by reorganising the order of operations in an LSTM system and develop a unique blocking-batching strategy to reuse the LSTM weights fetched from external memory to optimise the benefits of on-chip memory with a limited size for a large machine learning model. Evaluation results show that our architecture can achieve up to 20.8 GOPS/W, which would be among the highest for FPGA designs targeting LSTM systems with weights stored in off-chip memory. Comparing to the state-of-the-art design using off-chip memory to store the weights, we achieve 1.65 times higher performance-per-watt efficiency and 1.60 times higher performance-per-DSP efficiency. When compared with CPU and GPU implementation, our novel hardware architecture is 23.7 and 1.3 times faster while consuming 208 and 19.2 times lower energy respectively, which shows that our approach contributes to high performance and low power FPGA-based LSTM systems.**

I. Introduction

Recurrent Neural Networks (RNNs) can be utilised to process sequence data in many applications such as speech recognition [1], real-time control [2] and video classifications [3]. Among various types of RNN, Long Short-Term Memory (LSTM) networks can achieve high accuracy in many problems by using a memory cell that can remember long-term dependencies. A typical LSTM cell contains 4 gates, each with their own weights and biases which leads to a high computational cost during inference. This has typically been solved using multi-core CPUs and GPUs that can process large quantities of data.

FPGAs have been used to speed up the inference of LSTMs [4, 5, 6, 7], which offer benefits of low latency and low power when compared to CPUs or GPUs. Although FPGA-based LSTM accelerators have advantages in latency and power consumption, they are limited by the memory bandwidth of the FPGA board. The situation is even worse when we consider a small embedded system with low power and low memory bandwidth. An example of this is a monitoring camera system performing video processing, where large machine learning models have previously been infeasible due to high memory bandwidth and low latency requirements.

There has been previous work [7, 8, 9, 10] with FPGA based implementations such that all the weights are stored in the on-chip memory, but this is expensive and limits the size of models that can be deployed. When the RNN model is too large that the weights need to be stored on an external DRAM, it is not efficient because the fetched weights are typically used only once for each output computation.

In this work, we focus on LSTM models which are too large to store in on-chip memory of FPGA and we propose a novel blocking-batching strategy splitting the weight matrix into multiple blocks while batching the input activation vectors so that we can process the calculations block by block with weight reuse, which will reduce external memory access to save power and reduce latency. Batching the input activation vectors for RNN has been studied [9, 11, 12, 13] to increase the throughput, however few concern combining the blocking and batching for RNNs. In addition, we analyse the underlying data access pattern and dependency in the matrix-vector multiplication required by LSTM and a stall-free hardware architecture is proposed. With our method and new hardware architecture, large LSTM systems can be processed efficiently on FPGAs.

To the best of our knowledge, we are the first to propose and develop a weight reusing Stall-free Blocking-batching Engine (SBE) for large LSTMs whose weights are stored in the external memory of FPGAs.

Our contributions in this paper are as follows:

1) A new blocking-batching strategy to reuse the LSTM weights to optimise the throughput of large LSTM systems on FPGA devices with a performance analysis based on LSTM models which enables FPGA designs to balance between area, power and performance.
2) A novel stall-free hardware architecture to reorganise the multiplications involved in elimination of data dependency and stalls, thereby increasing the throughput of the system.
3) Evaluations of the proposed approach in different scenarios, showing improvements in speed and in energy efficiency.

Our performance efficiency can reach 20.8 GOPS/W while the resource efficiency is 0.246 GOPS/DSP, which would be the leading published results of LSTM systems whose weights are stored in off-chip memory, to the best of our knowledge. Comparing to the state-of-art design [13] storing weights in off-chip memory, we achieve 1.65 times higher performance-per-watt efficiency and 1.60 times higher performance-per-DSP efficiency.

II. Background

A. LSTM

LSTM is a class of Recurrent Neural Networks (RNN) which relies on a memory controller to learn long-term dependencies. It was initially proposed by Hochreiter and Schmidhuber [14]. Since then, there have been many modifications to the original LSTM cell for different applications, but the changes to the standard architecture are minimal and their effects on the overall prediction accuracy are negligible.

This work will focus on the optimisation of the standard LSTM but the proposed techniques can be applied to other RNN and LSTM variants. We follow the implementation of LSTM as used in [5, 15] and the equations are shown below, where $\odot$ is an element-wise multiplication:

$$
\begin{aligned}
i_t &= \sigma(W_i[x_t, h_{t-1}] + b_i) \\
f_t &= \sigma(W_f[x_t, h_{t-1}] + b_f) \\
u_t &= \sigma(W_u[x_t, h_{t-1}] + b_u) \\
o_t &= \sigma(W_o[x_t, h_{t-1}] + b_o) \\
c_t &= f_t \odot c_{t-1} + i_t \odot u_t \\
h_t &= o_t \odot tanh(c_t)
\end{aligned}
$$

i, f, u and o represent the input, forget, update and output gate respectively. We combine the input vector and hidden vector so that W represents the single weight matrix for both input and hidden elements and b represents the bias.

The input gate controls which elements are entered into the memory cell; the forget gate controls which elements are no longer remembered; the update gate controls which elements in the memory cell are updated; the output gate controls which elements from the memory cell are output. The output c_t is the cell state and h_t is the output of the cell, also called the hidden state, which is passed to the next layer. In an LSTM network, the output from the last layer is usually fed into the h_{t-1} input of the cell of the next timestep, and the same input vector x_t is used across layers in the same timestep.

B. Related Work

There has been much previous work on FPGA based LSTM implementations using on-chip memory to store all the weights. Ferreira et al. proposed an FPGA accelerator of LSTM in [7] for a learning problem of adding two 8-bit numbers with weights stored in on-chip memory. Rybalkin et al. [8] presented the first hardware architecture designed for BiLSTM for OCR. The architecture was implemented with 5-bit fixed-point numbers for weights and activations which were stored in on-chip memory. In addition, their later work [9] used 1-8 bits as the quantized implementation which can surpass a single-precision floating-point accuracy for a given dataset. The weights were still stored in on-chip memory. In [16] C-LSTM was proposed to reduce LSTM inference weight matrices using block circulant matrix and apply FFT algorithm to accelerate computation intensive block circulant convolution. Brainwave [10] proposed a single threaded SIMD architecture for CNN/RNN. Its high performance was through pinning neural networks - the idea that model weights could be pinned onto on-chip memory in order to achieve the necessary high memory read bandwidth. These FPGA based implementations stored all the weights in the on-chip memory, however this is expensive and limits the size of models that can be deployed.

Many works are focusing on model compression and weight pruning to reduce the weights size so that LSTM models can be stored in the on-chip memory to achieving good performance and efficiencies. [4] employed a pruning technique that compressed a large LSTM to fit the on-chip memory of an FPGA and improved inference efficiency. While in [17], Gao C. et al. proposed the DeltaRNN which was based on the Delta Network (DN) algorithm that skips dispensable computations during network inference The work in [18] proposed Bank-Balanced Sparsity (BBS) that could both maintain model accuracy and enable an efficient FPGA accelerator implementation. Such work is orthogonal to our proposed strategy. We provide another approach using the blocking-batching technique with a stall-free hardware architecture to optimise the throughput of LSTM systems on FPGA devices.

There has also been much previous work using off-chip memory for LSTM on FPGAs. In [19], Chang et al. presented an FPGA-based hardware implementation of LSTM on the Xilinx Zynq 7020 with 16-bit quantization for weights and input data. Both of the weights and input data were stored in off-chip memory which had been identified as the performance bottleneck. Guan et al. [5] proposed a smart memory organisation with on-chip double buffers to overlap computations with data transfers. And later in [13] they proposed an automated framework for mapping CNN or RNN on FPGAs. The matrix multiplication kernel was proposed to process LSTMs. However, [13] did not explore data dependence issue of LSTMs.

In [9, 12, 13], the batching technique is introduced to improve performance of LSTM inference, however without a proper blocking method they still need a large on-chip memory to store all the weights for efficient calculation. Otherwise high memory bandwidth memory is needed, like in the ASIC platform [12]. In addition, a framework that comprised of an approximate computing scheme in small tiled manner, together with a novel FPGA-based architecture for LSTMs, was presented in [20] with a focus on latency-critical applications.

Zhang et al. [21] implemented LRCN (Long-term Recurrent Convolutional Network) [15] using FPGA but the feature number, which was the size of input vectors for LSTM, had been reduced to 256 from 4096. This limits the effectiveness of the recurrent model and prevents the system from working with larger models. With our blocking-batching strategy, small FPGAs can still process a large RNN model efficiently. This is exactly the motivation of this work which focuses on the acceleration of large LSTM models stored in the off-chip memory of FPGA.

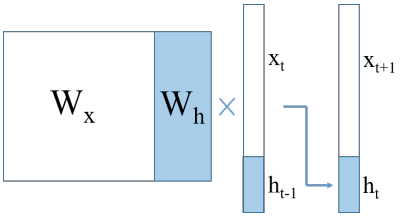

Fig. 1. Matrix-vector multiplication, showing the data dependency

III. Design and Optimization

Since most of the calculations within LSTM cells lie in the matrix-vector multiplication with complex data dependencies, this work will mainly focus on optimising this operation for high throughput. The element-wise operations in the LSTM tail can work under the shadow of the matrix-vector multiplications. In this section, an improved architecture is first presented to reorganise the multiplications to optimise data dependency and reduce stalls, thereby increasing the throughput of the system. Then a new blocking-batching strategy of reusing the LSTM weights to enhance the throughput of large LSTM systems is described via 3 scenarios. In addition, the optimised values of settings such as the best blocking number and batch sizes are introduced.

A. Overcoming Data Dependency

The traditional implementation of the matrix-vector multiplication involves the entire vector of (x_t, h_{t-1}) and a whole row of the weights at a time. However, this approach imposes additional stalling as the system needs to wait for new computed hidden units vector before starting the next time-step. This is mainly due to the data dependency between the output from the current time-step and the vector for the next time-step as shown in Fig. 1, where W_x and W_h represent the weights for the input vector and the weights for the hidden vector respectively. That implies that the whole system pipeline needs to be emptied to get the new computed hidden units before the new matrix-vector operations can start.

We propose a new technique that can alleviate this problem by calculating the matrix-vector operations in a different manner. At the beginning, only a few elements from the x_t vector are used while h_{t-1} is not touched, but all the elements in the corresponding columns of the weights matrix are used to do the operations, as shown in Fig. 2. The number of the involved elements in the x_t vector each cycle depends on the parallelism of system. In this way, the calculation of the new inference of (x_{t+1}, h_t) can start without waiting for the system pipeline to be emptied to get the h_t, which means that the system can be fully pipelined without stall. Each hidden vector can finish the computation in the shadow of processing x_t before it is used.

B. New Blocking-Batching Strategy

Many LSTM designs on FPGA share the same problem where all the weights need to be stored on-chip because of the slow latency to the off-chip memory. This approach is inapplicable for large machine learning models or a small

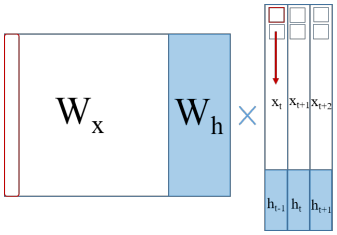

Fig. 2. New matrix-vector multiplication method using columns

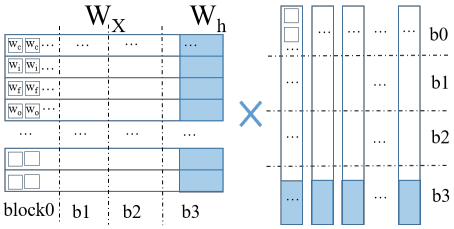

Fig. 3. Blocking of the weights matrix and input activation vectors

FPGA. Even after model compression and weights pruning, the designs can still suffer from insufficient on-chip memory for large compressed model. To solve this problem, we propose splitting the weight matrix into multiple blocks while batching the input activations vectors so that we can process the calculations block by block with weight reuse. This technique can be used for general LSTM model, or incorporated with the technique proposed in Section III-A. It seems similar to classic block matrix-matrix multiplications, however our proposal also considers the data dependence in LSTMs. With multiple vectors organised in a batch, the system can now reuse the same weights for the next matrix-vector operation without the full input vector being ready. Since memory accesses are expensive we want to reduce the number of loads from memory. This method can reuse the weights on more input vectors before reloading new weights from memory. This approach is especially useful in an embedded system where FPGA size and memory resources are both limited.

In our approach, the matrix includes parameters from all kinds of gates. In addition, the gate weights are interlaced in the matrix. Furthermore, we slice the weights matrix along the column, so the number of columns in each block is 1/(Blocking_Number of the original number of columns in the weights matrix), while the number of rows is the same as the original number of rows, as shown in Fig.3. So each block includes parameters from all kinds of gates.

Typically the transfer time of the weights is much larger than the computation time. By processing multiple time-steps of the input vector in a batch, we can use the weights multiple times before reloading, which reduces the number of memory accesses. Assuming the number of processing elements is fixed, increasing the batch size will also increase the computation time. We can find a batch size such that the computation time is equal to or larger than the transfer time to hide memory latency. In this way, we convert memory-bound applications to compute-bound ones and improve the

TABLE I
BLOCKING-BATCHING PARAMETERS

M_{op}	Number of matrix operations
N_{pe}	Number of processing elements
N_t	Number of elements transferred each cycle
N_b	Number of blocks
L_x, L_h	Number of elements in x and h vectors respectively
α	$\frac{L_x}{L_x+L_h}$
B	Batch size
P [1]	$\frac{Performance}{2*frequency}$

[1] performance is in terms of throughput while 2 means each data need both multiplication and accumulation operations.

performance.

In addition, we make use of a double buffering architecture which stores two blocks on-chip. Whilst calculating one block we can transfer the other block to maximise efficiency by reducing the stalling time.

C. 3 Scenarios

There are 3 cases in this blocking-batching strategy:

1) The hidden unit weights can be stored in one block
2) The hidden unit weights can be stored in two blocks
3) The hidden unit weights can be stored in more than two blocks

We define a few parameters, as shown in Table I for later calculations. Ideally we would like the calculation time for each block to be equal to the transfer time, but in reality usually one is significantly longer than the other. Let us assume the calculation time for one block is longer than the transfer time for one block.

$$\text{Calculation Time} \geq \text{Transfer Time}$$
$$\frac{M_{op}B}{N_b N_{pe}} \geq \frac{M_{op}}{N_b N_t} \implies B \geq \frac{N_{pe}}{N_t} \tag{1}$$

This gives us the constraint $B \geq \frac{N_{pe}}{N_t}$ when the calculation time is greater than or equal to the transfer time. Similarly we can derive the constraint $B < \frac{N_{pe}}{N_t}$ when calculation time is less than the transfer time.

Case 1 — In this case, the performance is almost dictated by having to store all weights in the on-chip memory. If the maximum performance is P_m, then this case can achieve P_m. This is due to the novel stall-free blocking-batching architecture that ensures we are always calculating and there is no stalling.

The ideal timing diagram for this case is shown in Fig. 4, where there is no idle time. T0 is transfer time for Block0 while C0 is computation time for Block0. As shown in Fig.4, C0 can start when T0 has finished. In practice, we find that there are some special cases where we must stall the pipeline to wait for the final block to finish calculating. Normally we can ignore the system latency because we can start processing the x part of the final block before we reach the h elements, as illustrated in Fig. 2; by the time we reach the h elements they will be ready. If the hidden input vector occupies a large

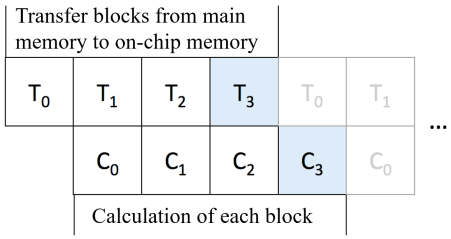

Fig. 4. Timing diagram for case 1

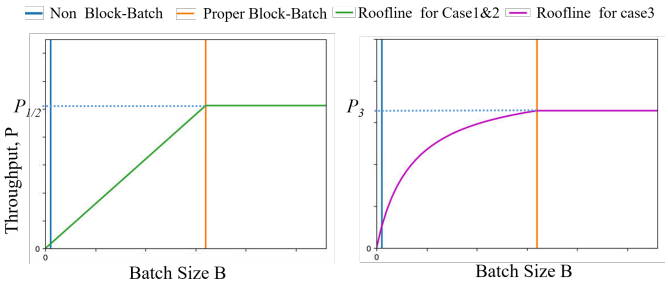

Fig. 5. Roofline performance model for case 1&2 (left) and case 3 (right)

amount of the block, then we will have to wait for the system pipeline to finish processing the last vector, which will cause stalls. We find that these stalls cause the calculation of the final block to take about 10% longer time. The calculations below consider the simple case when there are no stalls.

We can calculate the effect on performance by considering the total number of operations that must be done against the time spent. The performance depends on the time we spend transferring each block versus the calculation time of each block, as shown in the following equations and Fig. 5.

$$P = \frac{M_{op}B}{\frac{M_{op}B}{N_{pe}}} = N_{pe} \qquad \text{when } B \geq \frac{N_{pe}}{N_t} \tag{2}$$
$$P = \frac{M_{op}B}{\frac{M_{op}}{N_t}} = BN_t \qquad \text{when } B < \frac{N_{pe}}{N_t} \tag{3}$$

The blocking number, N_b, can be increased to reduce the on-chip memory needed. Due to storing two blocks on-chip, we only need $\frac{2}{N_b}$ as much memory as storing all weights on-chip. This means we can process a model many times larger, or process the same model using a fraction of the on-chip memory. Of course there are some drawbacks to increasing the block size, which are covered in cases 2 and 3.

Case 2 — In this case we must wait for both of the last blocks to be in the on-chip memory before starting computation, because the next hidden vector in the batch has a dependency on the previous. Fig. 6 shows the timing diagram for this case, where the red arrows indicate the extra time we must wait.

In theory there is a small overlap at the beginning where we can begin to compute the first sub-vector, and also at the end when we can start transferring while working on the last sub-vector in the last vector of the batch. Since this is equal to double the *time to process one sub-vector*, it will be negligible

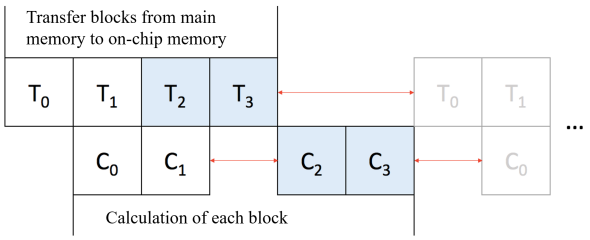

Fig. 6. Timing diagram for case 2

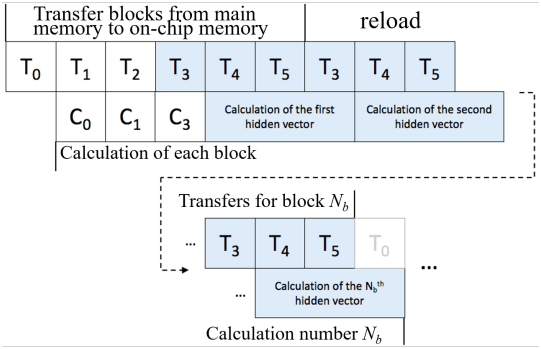

Fig. 7. Timing diagram for case 3

compared to the total time and we shall leave this out of our approximations.

The performance calculation is done in a similar way when $B = \frac{N_{pe}}{N_t}$; we consider that each matrix element must be transferred and the transfer time is equal to time for processing all the M_{op}, but the hidden weights also have the added processing time which takes up 2 blocks in all N_b blocks.

$$P = \frac{M_{op}B}{\frac{M_{op}B}{N_{pe}} + \frac{2M_{op}B}{N_b N_{pe}}} = \frac{N_{pe}N_b}{N_b + 2} \tag{4}$$

Case 3 — In the most complex case we have multiple blocks due to a large LSTM model and/or a small FPGA. In this case the hidden vector will be split across more than two blocks, so we cannot store it all on-chip at the same time.

Due to the data dependency between sub-vectors we must reload the last few blocks where the hidden vector is. This must be reloaded N_b number of times to finish each vector in the batch.

The performance calculation is more complex but follows the same pattern as before. We consider each case, when the x_t input and weights takes longer to transfer, then $\frac{\alpha M_{op}}{N_t}$ is larger than $\frac{\alpha M_{op}B}{N_{pe}}$ as shown in equation (6), or when the calculation takes longer than $\frac{\alpha M_{op}B}{N_{pe}}$ as shown in equation (5). Conversely, the hidden input and weights always spend more time transferring since each calculation is only one sub-vector from the batch, yet all the weights need to be transferred each time. The final roofline model is shown in Fig. 5.

$$p = \frac{M_{op}B}{\frac{\alpha M_{op}B}{N_{pe}} + \frac{(1-\alpha)M_{op}B}{N_t}} = \frac{N_{pe}N_t}{\alpha N_t + (1-\alpha)N_{pe}},\ B \geq \frac{N_{pe}}{N_t} \tag{5}$$

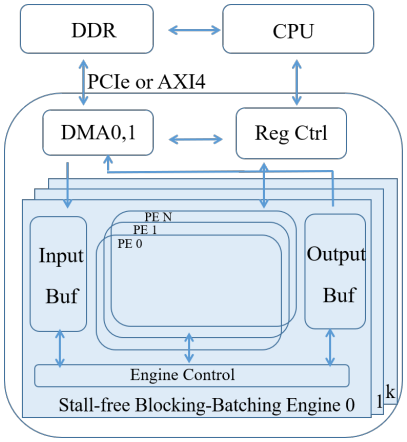

Fig. 8. The entire system

$$p = \frac{M_{op}B}{\frac{\alpha M_{op}}{N_t} + \frac{(1-\alpha)M_{op}B}{N_t}} = \frac{BN_t}{\alpha + (1-\alpha)B},\ B \leq \frac{N_{pe}}{N_t} \tag{6}$$

Although this case seems to offer poor performance because of the limitation of memory bandwidth, we should remember that this is similar to the standard method without processing using columns. We would need to load each block into memory B times and each sub-vector would be processed individually. With our new architecture, we re-use the weights as much as possible for the independent part of the vector, and only need to reload the weights for the dependent part of the vector with the hidden weights. Furthermore, if there are more on-chip memory on the target FPGA then this case will be changed to case 1 which becomes compute-bound with high performance.

IV. System Architecture

A. System Overview

Fig. 8 shows the overall system on a FPGA board while Fig. 9 shows the architecture of the Stall-free Blocking-batching Engine (SBE). This system consists of SBE units, with a CPU and DDR3 DRAM as the off-chip memory. All the weights and input activations (CNN-extracted features) are stored in the off-chip memory. The Reg Ctrl unit, which is connected to the AXI4-lite bus, is used to transfer the control commands while data communication is managed by the DMA units which are connected to the PCIe bus or AXI4 bus. The CPU is used to send configurable parameters to the SBE and control the transmitting of the weights and receiving the results when the hardware finishes processing, which is all done via the Reg Ctrl unit.

B. SBE Architecture

The details of the SBE architecture is shown in Fig. 9. As mentioned, only one block is transferred from the off-chip memory to the FPGA on-chip memory in each iteration of computation. The partial weights will be stored in buffer0 and buffer1 which work as a double buffer. In addition the partial batch_size activations of the input x vectors are also stored in a double buffer. With a carefully chosen batch size, these

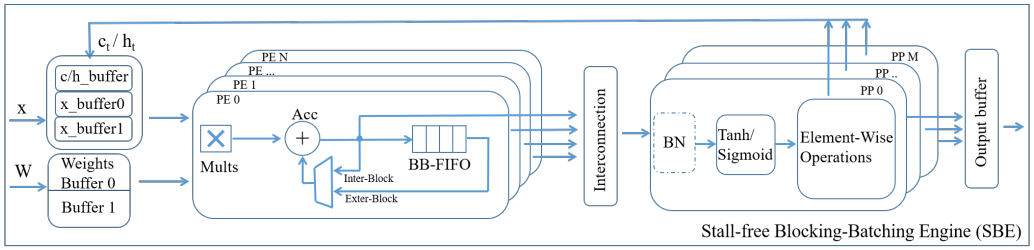

Fig. 9. Stall-free Blocking-batching Engine (SBE) Architecture Details

buffers work to overlap the time of data communication with LSTM inference computation.

The processing elements (PE) perform the matrix-vector operations that work as the LSTM gates. They multiply one element from the partial input vector by all the corresponding weights. The partial result of one partial activation will be accumulated via the inter-block linking and finally stored into the small Blocking-Batching(BB)-FIFO to be used in the next block. Each partial activation in the batch will generate one result and will be stored in the BB-FIFO. Therefore, the depth of the BB-FIFO is equal to the batch_size.

Consolidating of the block computations is done via the BB-FIFO in the PE units. When the new block computation begins, the value in the BB-FIFO will be read via the Exter-Block link and used as the initial value for the accumulator. After the new block computation, the partial results of the new block will be accumulated into the former partial results and finally stored into the BB-FIFOs. When all the blocks are processed, the final result across all blocks for the batch will be generated on the LSTM interconnection unit, where they will be reshaped for later processing.

The post processing (PP) units are used to perform other functions after the matrix-vector multiplications in the LSTM cell and they work under the shadow of the PEs. Their parallelism is configurable to improve the performance and reduce the latency depending on the FPGA resources available. The batch normalization (BN) [22] unit, which is optional and can be turned off via the controller, performs the batch normalisation on the results of the matrix-vector multiplications. The Sigmoid/Tanh are the non-linear modules which apply the activation functions. We implement these activation functions using a piece-wise linear approximation [23], which is shown to have little impact on accuracy during LSTM-RNN inference [5]. The outputs will be buffered in the output buffer while waiting to be transferred via DMA.

V. Evaluation

A. Experimental Setup

Many variants of LSTM have been proposed which are suitable for different tasks. In this work, the LRCN [15] for video activity recognition is used to demonstrate our approach. Typically, the LRCN is implemented using a CNN to extract a fixed-length vector of features which are then passed into a recurrent sequence learning module, such as an LSTM. In this work, the features of each frame in the video come from the average pool layer of the Inception-v3 which has been pre-trained on the ImageNet dataset. An additional Fully Connected layer is applied to transfer the features number to 1792 and then fed to our LSTM system. We retrain the LSTM network to get the top-1 accuracy of 72.97% and top-5 accuracy of 89.61% which are higher than the accuracy of 67.37% in the original LRCN design [15].

TABLE II
Resource Utilization

		LUT	LUTRAM	FF	BRAM	DSP
Zynq 7045	Avail.	218600	70400	437200	545	900
	Used	165668	49224	150451	517.5	900
	Utili.	75.8%	69.9%	34.4%	94.9%	100%
Virtex7 690T	Avail.	433200	174200	866400	1470	3600
	Used	203549	71478	221576	1070	2060
	Utili.	47%	41%	25.6%	72.8%	57%

To recognise the performance and limitations of the proposed LSTM hardware acceleration, we implement the hardware system for the LSTM part in LRCN for the RGB model, where the LSTM-256 model has 256 hidden units. Each LSTM-256 gate weights matrix is 2048*256 and there are four gates. The target platform is Xilinx ZC706, which consists of a XC7Z045 FPGA and dual ARM Cortex-A9 processor. 1 GB DDR3 RAM is installed on the platform as the off-chip memory. The on-chip memory of the XC7Z045 is 19.2Mb while the weights in this LSTM model are more than 32Mb which are too large to store in the on-chip memory of the FPGA. We also implement the LSTM-512 model which has 512 hidden units using the Virtex 7 VX690T FPGA.

B. Resource Utilisation

Table II shows the resource utilisation for our stall-free BPE design on the Zynq 7045 FPGA. The number of PEs, N_{pe}, is configured to 1024 targeting LSTM-256 while the batch size is 64. N_t is 16 when the DMA data bus is 256-bit with a 16-bit LSTM datapath. If the DMA data bus is 512-bit then the proper batch size is 32. N_t needs scaling if DMA data bus works under a different frequency with computation engines. For our system on Zynq, almost all the FPGA's hardware resources are utilised. A few multiplication units are implemented using LUT because there are only 900 DSP elements in our system. Note that the number of PEs, N_{pe}, is configured to 2048 for LSTM-512 targeting Virtex 7 VX690T FPGA because this device has an abundance of DSPs.

The best batch size. The best batch size is determined by balancing the computation time and communication time

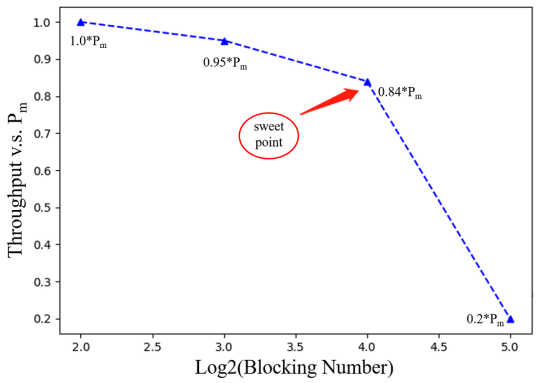

Fig. 10. Throughput vs Blocking Number on ZYNQ 7045

from the off-chip to on-chip memory. For case 1 and 2, the best batch size on Zynq can be easily calculated from equation (1), which shows that B = N_{pe}/N_t = 64. However, for case 3, the performance equations (5) and (6) are complex, but we can still get 64 as the proper batch size, as illustrated in Fig. 5. The performance is not related to B when $B \geq \frac{N_{pe}}{N_t}$ as shown in equations (2) and (5), which means increasing the batch size does not increase performance beyond a certain point, but only wastes the on-chip memory.

The proper blocking number. For a given LSTM model, when the blocking number increases, the block size decreases, and then the required on-chip memory decreases, because we will only store two blocks on the FPGA. This means that we can process a large LSTM system efficiently even with a small FPGA. However, for a given system, the blocking number cannot be too large because performance can be reduced as shown in case 3. The performance of the LRCN with different blocking numbers on the Xilinx ZC706 platform is shown in Fig. 10. P_m is the ideal performance when all the weights are stored in the on-chip memory without external DRAM accesses. It is the highest performance that the system can achieve. From Fig. 10, the proper blocking number is 16, which is the sweet point with only 1/8 on-chip memory required compared to previous research which put all the weights in the on-chip memory. It is the best trade-off between on-chip memory size/usage (or FPGA device) and performance. For a given application and performance requirement, the proper blocking number and blocking size will help us to choose the proper FPGA device. We do not need to select a large and expensive FPGA with large on-chip memory before the blocking-batching strategy is applied. When the blocking number decreases from 16 to 8, the performance can still be boosted by about 10%. However, a larger and more expensive FPGA with double on-chip memory will be required. Furthermore, if the user can bear with a reduced performance then they can choose a smaller and cheaper FPGA as shown in Fig. 10.

C. Performance and Efficiency Comparison

To compare the performance of the proposed design on FPGA with other platforms, we implement the LRCN on Intel Xeon E5-2665 CPU and NVIDIA X Pascal GPU based on Tensorflow(r1.12) framework. The CuDNN libraries are used for optimizing the GPU solution. Both CPU and GPU implementations run with batch size set to 32 samples, which are 1024 frames in total. Compared with the LRCN on CPU and GPU, our Zynq FPGA design is 23.7 and 1.3 times faster and consumes 208 and 19.2 times less power respectively as shown in Table III.

TABLE III
PERFORMANCE COMPARISON OF THE FPGA DESIGN VERSUS CPU AND GPU

	CPU	GPU	This Paper	This Paper
Platform	Intel Xeon E5-2665	TITAN X Pascal	Virtex 7 VX690T	Zynq 7Z045
Frequency	2.4 GHz	1.62 GHz	125 Mhz	142 MHz
Technology	22 nm	16 nm	28 nm	28 nm
Power(W)	93	159	26.5	10.6
Precision	32 bit float		16 bit fixed	
Model Size per Frame[1]	8192[1]* 256			
Time per Sample[2](ms)	14.45	0.78	0.38	0.61
Energy per Sample[2](mJ)	1343	124.02	10.05	6.47

[1] Combing the four matrices of i, f, o, c gates.

[2] Each sample/video has 32 frames.

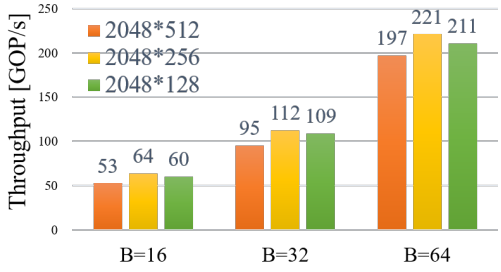

Fig. 11. Throughput depending on Batch Size on ZYNQ 7045

We have demonstrated parameterisable performance scaling for different LSTM sizes and batch size approaches, see Fig.11. With very large LSTM models, our design can achieve 1.60-5.41 times higher performance than the ones without SBE, as shown in Fig.12. In addition, the performance scaling for different blocking number is shown in Fig.10. The results show flexible customizability of the architecture for different scenarios.

To illustrate the benefits of our proposed approach, some

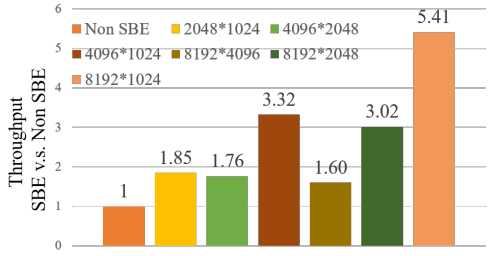

Fig. 12. Throughput of our design v.s Non SBE Design for Very Large LSTM Systems on ZYNQ 7045

TABLE IV
COMPARISON WITH PREVIOUS IMPLEMENTATIONS OF DENSE LSTM MODELS USING OFF-CHIP MEMORY

	2017 [5]	ESE [4]	FP-DNN [13]	This Paper	This Paper
FPGA	Virtex7 VX485T	Kintex KU060	StratixV GSMD5	Virtex7 VX690T	Zynq 7Z045
Model Storage	off-chip				
Prec. (bits)	32[a]	12	16 32[a]	16	16
No. of Used DSP	1176	1504	2072[b]	2060	900
Freq. (Mhz)	150	200	150	125	142
Perf. (GOPS)	7.26	282	316 86[a]	356	221
Power Effi. (GOPS/W)	0.37	6.87	12.63 3.44[a]	13.48	20.84
Resource Effi.[c] (GOPS/DSP)	0.006	0.188	0.153 0.042[a]	0.173	0.246

[a] Floating point
[b] One Intel FPGA DSP includes two 18*18 multipliers
[c] To make a fair comparison, the number of used DSPs is used to calculate GOPS/DSP when evaluating LSTM accelerator

existing FPGA-based LSTM-RNN accelerator designs are compared with ours in Table IV. For a fair comparison, We only show the previous work with detailed implementation of the LSTM system storing the weights in external memory of FPGA. We list the FPGA chips, model storage, precision, run-time frequency, throughput, power efficiency and resource efficiency. The table contains a range of designs across this parameter space for comparison. Our design achieves power efficiency as 20.84 GOPS/W and resource efficiency as 0.246 GOPS/DSP which are the highest with respect to state-of-the-art implementations on FPGAs operating on a dense LSTM model with weights stored in off-chip memory. With a similar number of DSP resources to [13], our system using Virtex 7 achieves 356 GOPS which is the highest performance among all the FPGA implementations of LSTMs storing weights in the off-chip memory. Because of routing congestions, our Virtex 7 design only runs at 125Mhz.

With our weights reusing SBE, small FPGAs can still process a large RNN model efficiently. Note that our comparison does not cover recent approaches [9, 17, 18] about LSTM acceleration using model compression and weight pruning to fit in on-chip memory. Such techniques are orthogonal to our proposed approach. Since useful inference results may not be possible when the FPGA has insufficient memory to store an accurate compressed model, it can still suffer from insufficient on-chip memory of FPGAs for large compressed models. Our technique complements these approaches for improving efficiency. Future work will explore pruning methods to allow large, sparse models to run on FPGAs.

VI. CONCLUSIONS AND FUTURE WORK

This paper proposes a Stall-free Blocking-batching Engine (SBE) architecture to accelerate the inference of LSTM-RNN systems on FPGAs. We focus on deploying large machine learning models on FPGAs, where resources are at a premium. We have implemented the proposed accelerator on Zynq and Virtex-7 FPGAs with excellent performance and efficiency which shows the effectiveness of our approach. Further research could look into combining the proposed SBE with pruning methods to allow large, sparse models to run on small embedded FPGAs, and the complete automation of the resulting approach to enable rapid development of efficient LSTM designs.

ACKNOWLEDGEMENT

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1 and EP/L00058X/1), Natural Science Foundation (No. 61201059) & Youth Fund (No.61704179) of China, Corerain and Xilinx are gratefully acknowledged.

REFERENCES

[1] F. Eyben *et al.*, "From speech to letters-using a novel neural network architecture for grapheme based ASR," in *Automatic Speech Recognition & Understanding, 2009. ASRU 2009. IEEE Workshop*, 2009.
[2] P. Xiong *et al.*, "Application of transfer learning in continuous time series for anomaly detection in commercial aircraft flight data," in *IEEE International Conference on Smart Cloud (SmartCloud)*, 2018.
[3] J. Yue-Hei Ng *et al.*, "Beyond short snippets: Deep networks for video classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.
[4] S. Han *et al.*, "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.
[5] Y. Guan *et al.*, "FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks," in *Design Automation Conference (ASP-DAC), 2017*.
[6] Z. Sun *et al.*, "FPGA acceleration of LSTM based on data for test flight," in *IEEE International Conference on Smart Cloud (SmartCloud)*, 2018.
[7] J. C. Ferreira *et al.*, "An FPGA implementation of a long short-term memory neural network," in *ReConFigurable Computing and FPGAs (ReConFig)*, 2016.
[8] V. Rybalkin *et al.*, "Hardware architecture of bidirectional long short-term memory neural network for optical character recognition," in *Proceedings of the Conference on Design, Automation & Test in Europe*, 2017.
[9] V. Rybalkin, A. Pappalardo, M. M. Ghaffar, G. Gambardella, N. Wehn, and M. Blott, "FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs," in *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE.
[10] J. Fowers *et al.*, "A Configurable Cloud-Scale DNN Processor for Real-Time AI," in *Proceedings of the 45th Annual International Symposium on Computer Architecture*, 2018.
[11] M. Zhang *et al.*, "DeepCPU: Serving RNN-based deep learning models 10x faster," in *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*.
[12] A. Ardakani, Z. Ji, and W. J. Gross, "Learning to skip ineffectual recurrent computations in LSTMs," *arXiv preprint arXiv:1811.10396*, 2018.
[13] Y. Guan *et al.*, "FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates," in *Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017.
[14] S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[15] J. Donahue *et al.*, "Long-term Recurrent Convolutional Networks for Visual Recognition and Description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015.
[16] S. Wang *et al.*, "C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM.
[17] C. Gao *et al.*, "DeltaRNN: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018.
[18] S. Cao *et al.*, "Efficient and Effective Sparse LSTM on FPGA with Bank-Balanced Sparsity," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019.
[19] A. X. M. Chang, B. Martini, and E. Culurciello, "Recurrent neural networks hardware implementation on FPGA," *arXiv preprint arXiv:1511.05552*, 2015.
[20] M. Rizakis *et al.*, "Approximate FPGA-based LSTMs under computation time constraints," *arXiv preprint arXiv:1801.02190*, 2018.
[21] X. Zhang *et al.*, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *Field Programmable Logic and Applications (FPL)*. IEEE, 2017.
[22] T. Cooijmans, N. Ballas, C. Laurent, Ç. Gülçehre, and A. Courville, "Recurrent batch normalization," *arXiv preprint arXiv:1603.09025*, 2016.
[23] H. Amin *et al.*, "Piecewise linear approximation applied to nonlinear function of a neural network," *IEEE Proceedings-Circuits, Devices and Systems*, vol. 144, 1997.