Journal of Signal Processing Systems (2020) 92:965–979
https://doi.org/10.1007/s11265-020-01549-8

Mapping Large LSTMs to FPGAs with Weight Reuse

Zhiqiang Que[1] · Yongxin Zhu[2] · Hongxiang Fan[1] · Jiuxi Meng[1] · Xinyu Niu[3] · Wayne Luk[1]

Received: 30 November 2019 / Revised: 23 March 2020 / Accepted: 6 May 2020 / Published online: 9 July 2020

Abstract

Long-Short Term Memory (LSTM) can retain memory and learn from data sequences. It gives state-of-the-art accuracy in many applications such as speech recognition, natural language processing and video classifications. Field-Programmable Gate Arrays (FPGAs) have been used to speed up the inference of LSTMs, but FPGA-based LSTM accelerators are limited by the size of on-chip memory and the bandwidth of external memory on FPGA boards. We propose a novel hardware architecture to overcome data dependency and a new blocking-batching strategy to reuse the LSTM weights fetched from external memory to optimize the performance of systems with size-limited on-chip memory for large machine learning models. Evaluation results show that our architecture can achieve 20.8 GOPS/W, which is among the highest for the FPGA-based LSTM designs storing weights in off-chip memory. Our design achieves 1.65 times higher performance-per-watt efficiency and 2.48 times higher performance-per-DSP efficiency when compared with the current state-of-the-art designs of LSTM using weights stored in off-chip memory. Compared with CPU and GPU implementations, our FPGA implementation is 23.7 and 1.3 times faster while consuming 208 and 19.2 times lower energy respectively, which shows that our approach enables large LSTM systems to be processed efficiently on FPGAs with high performance and low power consumption.

Keywords LSTM · FPGA · Hardware architecture

1 Introduction

Recurrent Neural Networks (RNNs) can remember past information so that they can improve the accuracy of future predictions, which makes them applicable to sequence processing problems such as speech recognition [7, 16], real-time control [33] and video classifications [34]. Developed to overcome the vanishing gradient problems that can be encountered when training traditional RNNs, Long Short-Term Memory (LSTM) networks have set accuracy records in multiple application domains. Although they have the benefits in accuracy, the typical four gates in an LSTM cell also result in a high computational cost during inference because each gate has its own weights and biases. In recent years, FPGAs have been used to speed up the inference of LSTMs [11, 15, 16, 31], which offer low latency and low power when compared to CPUs or GPUs.

Although FPGA-based LSTM accelerators have advantages in power consumption and latency, they are limited by the size of on-chip memory and the bandwidth of external memory on the FPGA boards. The situation is even worse when we consider a small embedded system which has a small on-chip memory and low memory bandwidth while requires low power consumption. In many previous FPGA based implementations [12, 24, 27, 29, 30], all the weights are stored in the on-chip memory, which not only is expensive but also limits the size of models that can be deployed. The size of weights for a typical single LSTM layer with both input and hidden vector sizes as 512

✉ Zhiqiang Que
z.que@imperial.ac.uk

Yongxin Zhu
zhuyongxin@sari.ac.cn

Hongxiang Fan
h.fan17@imperial.ac.uk

Jiuxi Meng
jiuxi.meng16@imperial.ac.uk

Xinyu Niu
xinyu.niu@corerain.com

Wayne Luk
w.luk@imperial.ac.uk

[1] Imperial College London, London, UK

[2] Shanghai Advanced Research Institute, Chinese Academy of Sciences, Beijing, China

[3] Corerain Technologies Ltd., Shenzhen, China

is around 33.6Mb when the data precision is 16-bit. This size becomes 134.3Mb when both the sizes of the vectors become 1024. However, the on-chip memory of an FPGA is limited. The Xilinx Zynq 7045 FPGA only has 19.2Mb while the Virtex 7 690T FPGA has 53Mb, which is insufficient to store a single LSTM layer with vectors of medium sizes, such as 1024, as shown in Fig. 1. Increasing the size of FPGA chip may help to store an entire LSTM layer in the on-chip memory. However, a typical neural network may have several LSTM layers or a large LSTM layer, which makes the weights hard to be stored entirely in the on-chip memory of an FPGA.

When an RNN model is so large that the weights must be stored in an external DRAM, the performance will be largely restricted by the transfer rate of the data. It is not efficient since the fetched weights are typically used only once for each LSTM inference. Besides, the existence of data dependency in RNNs makes systems stall in conventional designs as the systems need the new computed hidden units vector to start the calculation of next time-step.

This study aims to explore LSTM parallelism by designing a novel blocking-batching strategy and a stall-free hardware architecture to optimize the performance of FPGA-based LSTM models that are too large to store into the on-chip memory of FPGAs. To reduce external memory access to save power and decrease latency, we propose the a new blocking-batching strategy which splits the matrix of weights into multiple blocks while batching the input vectors so that calculations are able to be processed block by block with weight reuse. Batching the input vectors of activation for RNNs has been studied [2, 14, 29, 35] to improve the throughput, but little research has so far looked into the combination of blocking and batching targeting RNNs. Besides, we analyze the data dependency in the matrix-vector multiplication required by LSTM, and the stall-free hardware architecture is proposed. Some previous studies focus on weight pruning and data precision reduction to reduce the size of LSTM models to fit on-chip memory. These studies are orthogonal to our proposed strategy and architecture. With our method and novel hardware architecture, large LSTM systems can be processed efficiently on FPGAs.

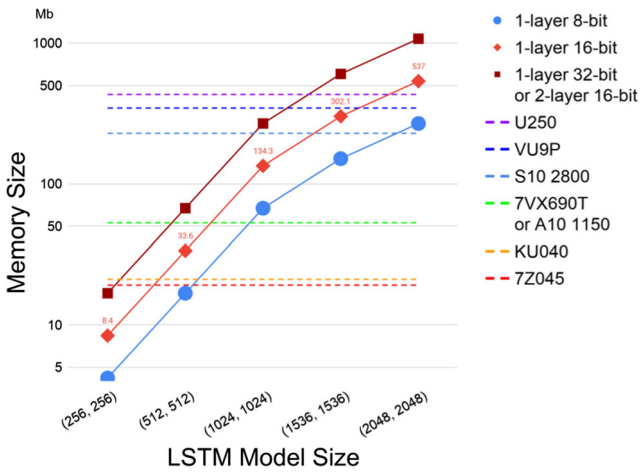


Figure 1 The sizes of weights for various LSTM layers and the on-chip memory sizes of various FPGAs

To the best of our knowledge, we are the first to propose and implement a Stall-free Blocking-batching Engine (SBE) with weight reuse for large LSTMs storing weights in the off-chip memory of FPGAs. Here are our contributions in this paper:

1. A new blocking-batching strategy reusing the LSTM weights to optimize the throughput of large LSTM systems on FPGAs.
2. A novel stall-free hardware architecture to reorder the multiplications to hide data dependency and stalls, which further improves the throughput of the system.
3. A performance model which enables a balance between performance, power and area for FPGA designs with an automation framework for our novel LSTM architecture to improve productivity of application developers.
4. Evaluations of our LSTM accelerator in different scenarios. The performance efficiency achieves 20.8 GOPS/W while the resource efficiency is 0.246 GOPS/DSP, which provides the leading published results of FPGA-based LSTM systems storing weights in off-chip memory.

This paper is an extension of [25]; compared with the previous work, we include details of LSTM gate weights matrix partition in Section 3.4 and a fine-tuning technique with data quantization in Section 3.5, as well as an extended performance model in Section 5.1 and new resource model in Section 5.2. We also include a new automatic hardware mapping framework for our LSTM architecture to improve the productivity of application developers in Section 5.3. In the evaluation section, the tuning of the blocking number is extended in Section 6.3 and more comparisons to previous work are added in Table 4. Finally, some additional publications are discussed in Section 2.

2 Previous Work

There has been much previous work on FPGA-based LSTM implementations whose weights are stored in on-chip memory. In contrast to convolutional neural networks (CNNs), whose architectures allow massive parallelism by the reuse of filter weights [8, 10], RNNs/LSTMs are harder to be accelerated on hardware due to their high complexity

and temporal dependency. In [11], an FPGA accelerator of LSTM is proposed for a learning problem of adding two 8-bit numbers using weights stored in on-chip memory. Rybalkin et al. [30] are the first to propose and design a BiLSTM hardware architecture for OCR with weights storing in on-chip memory. In their later work [29], FINN-L employs 1-8 bits as the quantized implementation which is able to surpass single-precision floating-point accuracy for a given dataset. However, the weights are still stored in on-chip memory. C-LSTM [32] is proposed to reduce LSTM inference weight matrices using the block circulant matrix. In [12], Microsoft proposes a Brainwave variant which is a single-threaded SIMD architecture for RNNs. Its idea is pinning the model weights into on-chip memory in order to achieve high memory read bandwidth to support high performance for RNNs. In [24], a Brainwave-like Neural Processing Unit (NPU) is implemented. They also propose TensorRAM for persistent data-intensive RNN sequence models. An E-RNN framework [20] is introduced to improve performance/energy efficiency under accuracy requirement with ADMM-based training for deriving block-circulant matrice-based RNN representation. In [26] a novel Timestep(TS)-buffer is introduced to avoid redundant calculations of LSTM gate operations to reduce system latency. In [27], the authors propose a novel latency-hiding hardware architecture based on column-wise matrix-vector multiplication to eliminate data dependency, improving the throughput of systems of LSTM/GRU models. These LSTM implementations store all the weights in on-chip memory of FPGAs. However, it is expensive and limits the size of models which are deployed since the on-chip memory is limited.

Many studies are focusing on weight pruning and model compression to reduce the size of weights so that the whole LSTM model can be stored in the on-chip memory to achieve good performance and efficiencies because of high memory bandwidth. In [16], the authors propose a pruning technique which compresses a large LSTM to fit the on-chip memory of an FPGA and improves inference efficiency. While in [13], DeltaRNN is proposed. It is based on the Delta Network algorithm which skips dispensable computations during inference of network. The authors in [3] propose Bank-Balanced Sparsity (BBS) which is able to both maintain model accuracy and enable an efficient FPGA accelerator implementation. These studies are orthogonal to our proposed strategy and architecture. We provide another approach which employs the blocking-batching technique with the stall-free hardware architecture to optimize the throughput of LSTM networks on FPGAs.

There have also been many previous studies of LSTM implementations storing weights in off-chip memory on FPGA devices. Chang et al. [4] presents a hardware implementation of LSTM using Xilinx Zynq 7020 FPGA. Both the weights and input data are quantized to 16-bit and stored in off-chip memory which is the performance bottleneck. The authors in [15] propose to use on-chip double buffers and develop a smart memory organization to overlap data transfers with computations. Later they propose an automated framework named FP-DNN [14] to map CNNs or RNNs on FPGA devices. LSTMs are processed using matrix multiplication kernel. However, they [14] do not explore the issues of data dependency in LSTMs.

In [2, 14, 29], the batching technique is proposed to increase the throughput of LSTM inferences. However, a large on-chip memory is still needed to store all the weights for efficient calculation without a proper blocking method. Otherwise, high bandwidth memory is required, like in the ASIC platform [2]. Besides, a framework which deploys an approximate computing scheme using small tiles, together with a novel hardware architecture for FPGA-based LSTMs, is presented in [28] which focuses on latency-critical applications. The work in [21–23] compares neural networks implemented on CPU, FPGA, GPU and ASIC and shows FPGA can provide superior performance/Watt over CPU and GPU.

In [36], Zhang et al. implement Long-term Recurrent Convolutional Network (LRCN) [6] on FPGA. However, the feature number, which is the length of LSTM input vector, has been reduced from 4096 to 256. This reduction prevents the system from working with large models and limits the effectiveness of RNN models. With our blocking-batching strategy and hardware architecture, small FPGAs are still able to run large RNN models efficiently. In particular, this work focuses on the FPGA-based acceleration of large RNN models whose weights are stored in the external memory of FPGA devices.

3 LSTM: Design and Optimization

This work mainly optimizes the Matrix-Vector Multiplications (MVM), which have complex data dependencies in LSTMs, for high throughput as most of the calculations of LSTM cells lie in the MVMs. The element-wise operations in the LSTM tail are able to run under the shadow of the matrix-vector multiplications in our pipelined design. In this section the basic of LSTM theory is first introduced. Then, an improved architecture is proposed to reorganize the multiplications to optimize the data dependency and remove stalls, which enhances the system throughput. Furthermore, a novel blocking-batching strategy is introduced which reuses the LSTM weights to increase the throughput of large LSTM systems on FPGA devices.

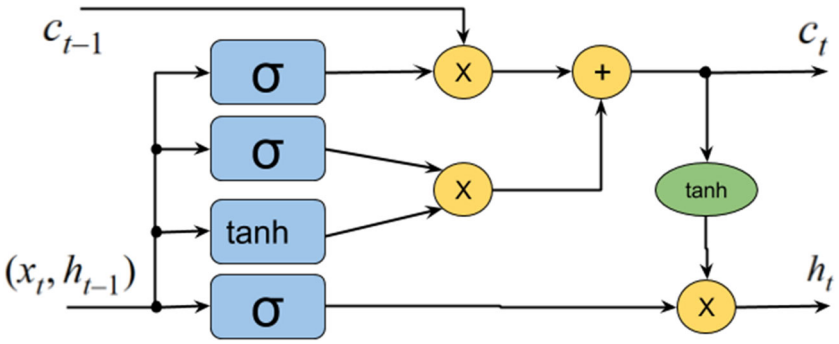


Figure 2 Structure of an LSTM Cell

3.1 LSTM Theory

The Long Short-Term Memory (LSTM) networks are based on Recurrent Neural Networks (RNN). It was initially proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [17]. The LSTM architecture, as shown in Fig. 2, relies on dedicated memory cells to store information about long-term dependencies over an arbitrary time period, which is well suited for processing time series data.

The standard LSTM as shown in [6, 15] is implemented in our work. The hidden state h_t is produced by the following equations, where $\odot$ is element-wise multiplication:

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

i, f, u and o represent the input gate, the forget gate, the input modulation gate and the output gate respectively. The input modulation gate is often considered as a sub-part of the input gate. The input vector and hidden vector are combined so that W represents the weights matrix for both input and hidden units. Bias is represented as b.

The gates control the information flow inside the LSTM unit. The input gate decides which elements will enter the memory cell; the forget gate decides which elements are no longer remembered; the input modulation gate decides if the memory cell needs an update; the output gate decides which elements from the memory cell are output. The output c_t is the internal cell state and h_t is the output of the cell, also called the hidden state, which is passed to the next time-step or next layer. Our work focuses on the optimization of the standard LSTM, but the proposed techniques can be applied to other RNN and LSTM variants.

3.2 Overcoming Data Dependency

The conventional matrix-vector multiplication implementation involves the entire vector of (x_t, h_{t-1}) and a whole row of the weights matrix at a time. However, additional stalling is introduced since the system needs to wait for the new computed hidden units vector before it starts the next time-step calculation. 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. 3, where W_x and W_h represent the weights for the input vector and the weights for the hidden vector respectively. This implies the whole system pipeline needs to be emptied to get the new computed hidden units vector so that the new calculation of matrix-vector multiplications can start.

To alleviate this problem, we propose a new technique which calculates the matrix-vector multiplications in a unique manner. In the beginning, only a few elements from the x_t vector are involved while h_{t-1} is not touched, but all the corresponding columns within the weights matrix are involved to run the calculation, as shown in Fig. 4. The number of elements involved in the x_t vector each cycle depends on the parallelism of the system. This number may be just one. In this way, the calculation of the new inference which involves (x_{t+1}, h_t) can start without waiting for the system pipeline to be emptied to get the h_t. It means that the system can be fully pipelined without stall and each

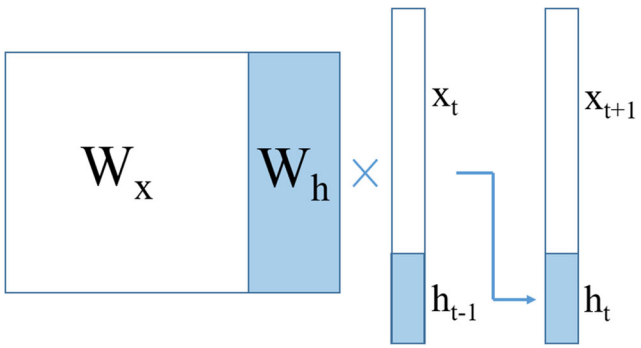


Figure 3 Matrix-vector multiplication, showing the data dependency.

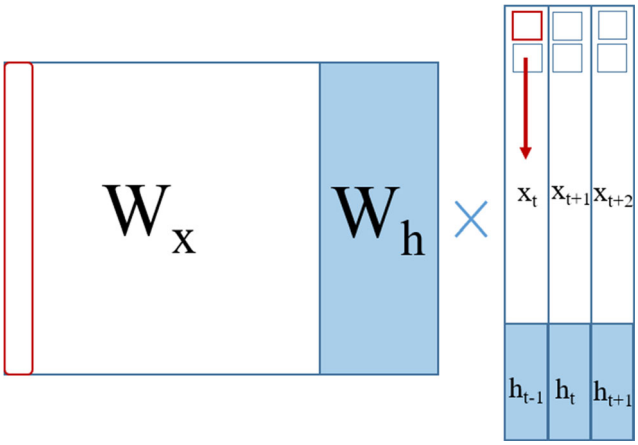


Figure 4 New matrix-vector multiplication method using columns

hidden vector can finish the computation in the shadow of processing x_t vector before it is touched.

3.3 Novel Blocking-Batching Strategy

Many existing FPGA based LSTM designs share the same problem where all the weights need to be stored internally because of the long latency to the external memory. This approach is impractical for large machine learning models or small FPGAs. Even after weights pruning and model compression, the designs may still suffer from insufficient on-chip memory because of large compressed models.

To solve this problem, we propose a novel technique which splits the weights matrix into multiple blocks while batching the input activation vectors so that we can process the calculations block by block and reuse the weights. There has been much work about improving the throughput by batching input activation vectors for RNNs. However, little research looks into the combination of blocking and batching targeting RNNs. This technique can be used for a general LSTM model or incorporated with the technique proposed in Section 3.2. It may be similar to classic Block Matrix Multiplications (BMM), however our proposal also considers the data dependency within LSTMs. With multiple vectors organized in a batch, the system can now reuse the same weights for the new matrix-vector multiplications in the next time-step. Since external memory accesses are expensive we design to decrease the number of loads from external memory. Our approach can reuse the weights for several input vectors before reloading new weights from external memory. This approach is especially useful for embedded systems where FPGA size and memory resources are both limited.

Typically the transfer time of the weights is much larger than the computation time in LSTM inferences. 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 external 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 so that the memory latency can be hidden. In this way, we convert memory-bound applications to compute-bound ones and increase performance. Besides, a double buffering architecture is proposed to store two blocks on chip. Whilst calculating one block, the other block can be transferred to maximize performance and efficiency by reducing the time of stalling.

3.4 Weights Matrix Partition

In our design, the matrix of weights contains gate-weights from all kinds of gates in LSTM and these weights are interlaced within the matrix. For example, the first four rows of our weights matrix are respectively the first row of the weights matrix for input gate, input modulation gate, forget gate and output gate, as shown in Fig. 5. Thus, in one time-step for the LSTM algorithm, we only need to focus on optimizations of one large matrix multiplying one vector for the whole LSTM cell instead of four small matrices multiplying one vector which is decentralized. Besides, the weights matrix is sliced along the column and the number of rows is the same as the number of rows in the weights matrix. Thus, each block contains weights from all the LSTM gates for increasing design parallelism.

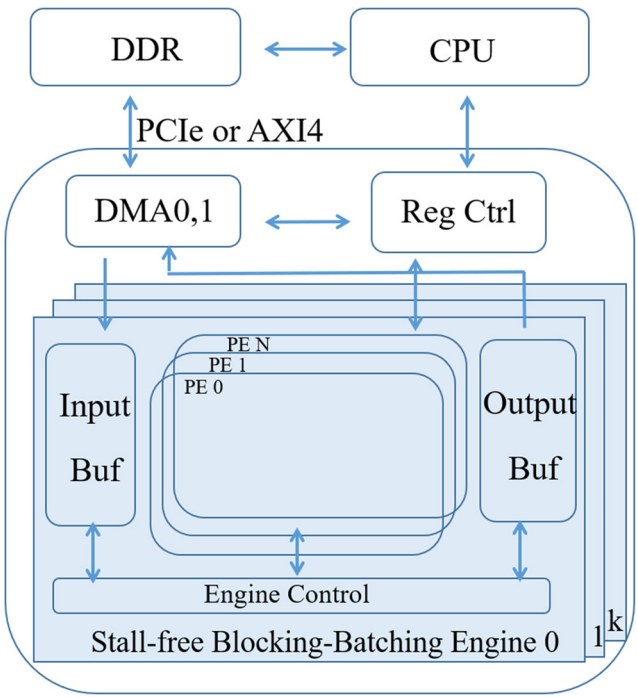


Figure 6 The system overview

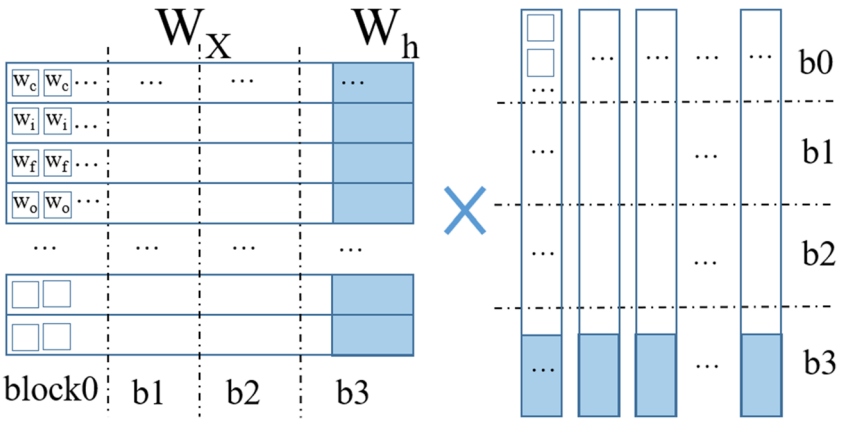


Figure 5 Blocking of the weights matrix and input activation vectors

3.5 Data Quantization and Fine Tuning

Numerous prior efforts [16, 29, 30] have shown that LSTMs are robust to low bit-width quantization. Instead of using double or single precision floating-point representations, fixed-point precision can be used in FPGA-based LSTM accelerators to achieve higher performance and power efficiency. To keep the accuracy and avoid data overflow, we use partial quantization [8] to extend the bit-width of intermediate data. In this work, a 16-bit fixed-point data format is proposed to implement the multipliers in the LSTM gates while the accumulators after the multipliers are 32-bit. The multipliers and adders in the element-wise operations and batch normalization are both 32-bit.

Quantization-aware fine-tuning [18] is applied to our quantized LSTM to recover accuracy. The gradient, weight, activation tensors are stored in floating-point format. To emulate the quantization error, all the operations are performed in fixed point arithmetic. Therefore, the conversation between floating-point data and fixed-point data is applied before and after each operation to match the data format. With the help of quantization-aware fine-tuning, we evaluate the performance and power efficiency of the proposed LSTM accelerator using real-time video activity recognition on UCF101 dataset [19], and there is no accuracy loss.

4 System architecture and Implementation

4.1 System Overview

Figure 6 shows the overall system on an FPGA board while the Stall-free Blocking-batching Engine (SBE) is illustrated in Fig. 7. This system is composed of SBE units, DMA units, a CPU and a DDR3 DRAM as the external memory. All the LSTM weights and input activation vectors which are image features extracted from CNN layers are stored in the external memory. The control commands are transferred by the Reg Ctrl unit using the AXI4-lite bus. The DMA units manage the data communication and they are connected to the PCIe bus or AXI4 bus. The CPU is used to send configurable parameters to the SBE and it controls the transmitting of the weights and receiving the results after the hardware finishes processing.

4.2 SBE Architecture

Figure 7 shows the details of the SBE architecture. As mentioned, we only transfer one block 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. Besides, the partial batch_size input x vectors of activation are also stored in a double buffer. These buffers work together to enable parallel operation of data communication and LSTM inference computation with a carefully chosen block and batch size.

PEs — The LSTM gate operations (matrix-vector multiplications) are performed in the Processing Element (PE) units. One element is chosen from the partial input vector and multiplied by all the corresponding weights. The partial results of one activation are accumulated via the Inter-Block link as shown in Fig. 7, and is finally stored into the small Blocking-Batching(BB)-FIFO to be read in the computation of the next block. Each partial activation vector in the batch generates one result and is stored in the BB-FIFO. Therefore, the depth of the BB-FIFO is equal to the batch_size.

PPs — The other operations after the matrix-vector multiplications in the LSTM cell are processed in the Post Processing (PP) units. Their parallelism is configurable to reduce the latency and improve the performance depending on the available FPGA resources. The batch normalization (BN) [5] unit, which is optional and can be turned off via the controller, performs batch normalization on the results from

the matrix-vector multiplications. The Sigmoid/Tanh are the non-linear modules which apply the activation functions. We implement these activation functions using piece-wise linear approximation [1], which is shown to have little impact on accuracy during LSTM-RNN inference [15]. The results will be placed in the output buffer, waiting to be transferred via DMAs.

4.3 Consolidating after Blocking

The main issue of blocking is how the system consolidates the partial results. In our architecture, BB-FIFO in the PEs is utilized to consolidate the block computations. When a new block starts computation, the entry in the BB-FIFO is read via the Exter-Block link and used as an initial value for the accumulator. After the new block finishes computation, the partial results of the new block are 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 from the current batch will be passed to the LSTM Interconnection unit as shown in Fig. 7, where they will be reshaped for later processing.

Table 1 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	$L_x/(L_x + L_h)$
B	Batch size
P [1]	$Performance/(2 \times frequency)$

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

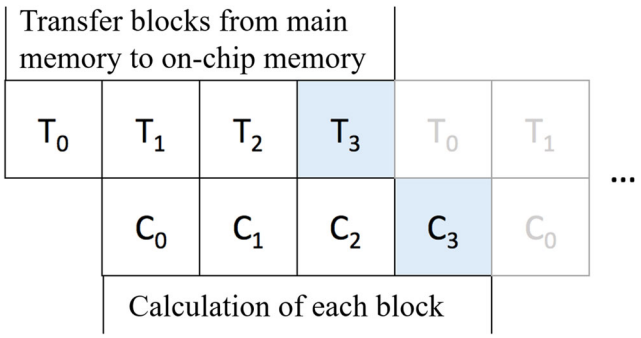


Figure 8 Timing diagram for Case 1

5 Performance Model and Hardware Mapping Framework

5.1 Performance Model

Generally, there are three cases which involve the proposed 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 1 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.

Calculation Time $\geq$ 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 when the hidden unit weights can be stored in one block, 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 without stalling.

The ideal timing diagram for this case is shown in Fig. 8, where there is no idle time. T_0 is transfer time for Block0 while C_0 is computation time for Block0. As shown in Fig. 8, C_0 can start when T_0 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. 4; by the time we reach the h elements they will be ready. If the hidden input vector occupies a large 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

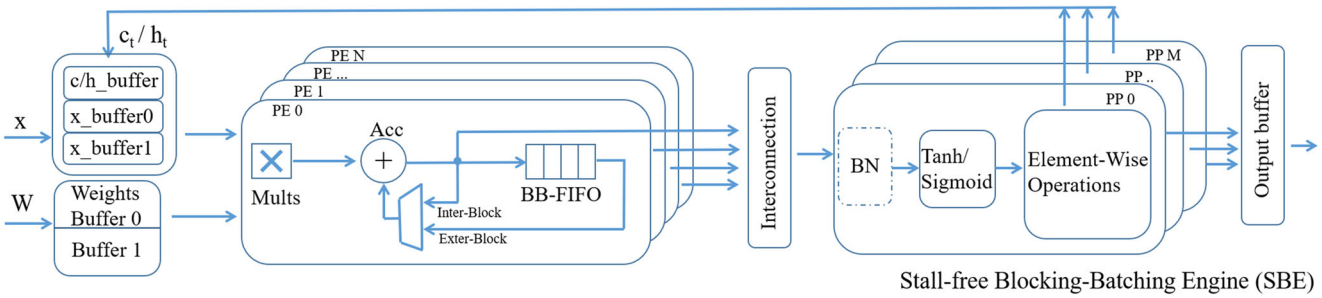


Figure 7 The Details of Stall-free Blocking-batching Engine (SBE) Architecture

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. 9.

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

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

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. There are indeed some drawbacks, such as increasing the block size, which are covered in cases 2 and 3.

Case 2 — In this case when the hidden unit weights can be stored in two blocks, we must wait until both of the last blocks to be cached in the on-chip memory before starting computation, because the next hidden vector in the batch has a dependency on the previous one. Figure 10 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 doubling the *time to process one sub-vector*, it will be negligible 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 the time for processing all the M_{op}, but the hidden weights also have the added processing time which takes up 2 blocks out of all N_b blocks.

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

Case 3 — In the most complex case when the hidden unit weights can be stored in more than two blocks, 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 take longer to transfer, then

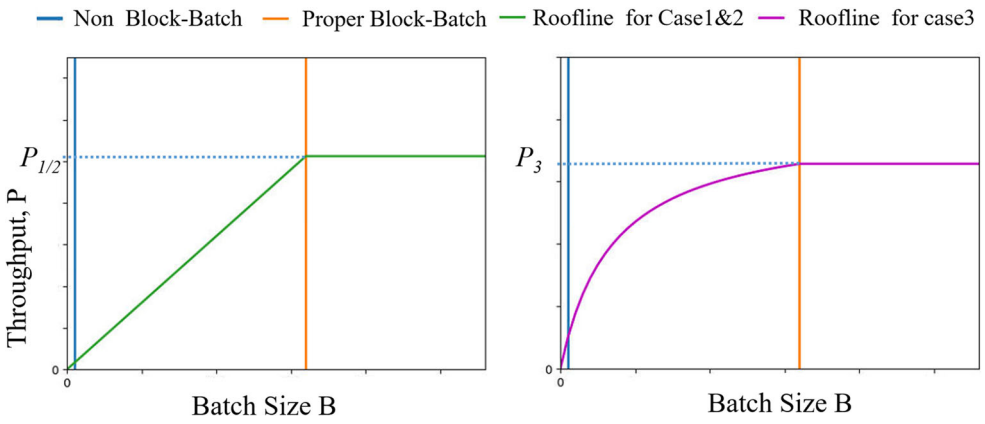


Figure 9 Roofline performance model for Case 1&2 (left) and Case 3 (right).

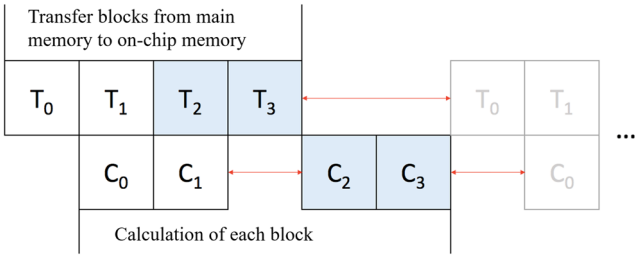


Figure 10 Timing diagram for Case 2.

$\frac{\alpha M_{op}}{N_t}$ 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 need more time to transfer since each calculation is only one sub-vector from the batch, yet all the weights need to be transferred each time, as shown in Fig. 11 The final roofline model is shown in Fig. 9.

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

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

Although this case seems to offer poor performance because of the limitation of memory bandwidth, we should note 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 memories on the target FPGA, then this case will be become Case 1 which is compute-bound with high performance.

5.2 Resource Modelling

FPGA-based LSTM accelerators are constrained by two types of resources: one is the logic resources such as LUTs and DSPs, the other is the memory resource i.e. the BRAMs. Based on [9, 37], DSPs are the limiting resource for the computation engines. Therefore, only DSP usage is considered in this category. In our design, fixed-point adders are implemented using LUTs in order to save DSPs since the adders consume much fewer LUTs compared to those of multipliers and considering the available LUTs are far more than DSPs in an FPGA. Let D_{mul} represent the number of DSP usage of one multiplier, so the total number of required DSPs is $N_{pe} * D_{mul}$.

The memory resources are mainly occupied by the dual buffers and BB-FIFOs and their usage is given by:

$$BRAM_{Num} = \frac{(L_x + L_h) * (4L_h + B) * DW * 2/N_b + B * N_{pe} * DW}{BRAM_{size}} \tag{7}$$

Practically, BB-FIFO can be implemented using LUTRAM in order to save BRAMs when the entry of each BB-FIFO is small. If so, the term of $\frac{B*N_{pe}*DW}{BRAM_{size}}$ in equation (7) can be omitted.

5.3 Framework of Automatic hardware Mapping

This section presents Blocking-Batching(BB)-LSTM, an end-to-end LSTM framework to enable effective deployment of applications on FPGA by replacing standard LSTM layers with an efficient configurable LSTM template written in Verilog. It optimizes parameters of the proposed SBE hardware based on users' constraints to enable design space exploration to meet design requirements. In addition, it automatically reads model configuration files and coefficient data files to map a high-level model into an FPGA-based accelerator for inference to reduce the development effort.

The overall BB-LSTM framework is shown in Fig. 12. After users create a network model using the Keras API in TensorFlow with a satisfactory accuracy, the network configuration and coefficients can be exported as a JSON file

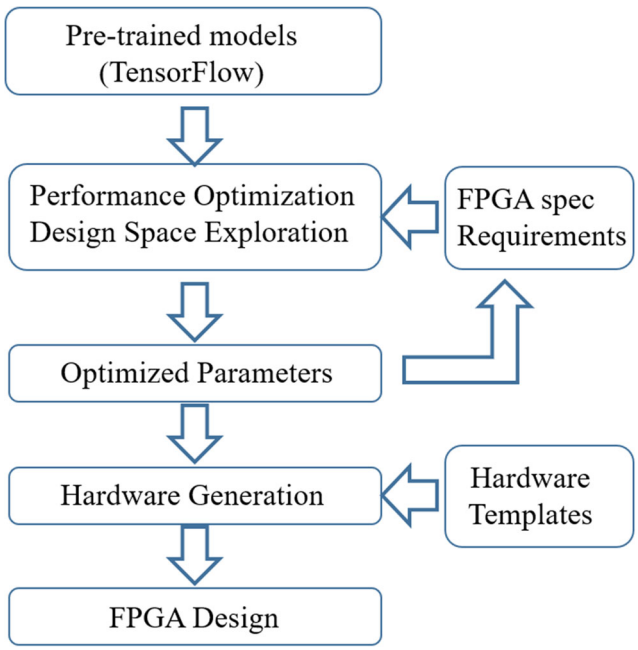


Figure 12 BB-LSTM framework.

and an HDF5 file respectively. BB-LSTM reads both files about this model and the parameters of the targeted FPGA platform with the user requirements for performance or area. Then it evaluates the performance of the combinations based on the above LSTM performance model described in Section 5.1 and the optimized parameters including the best batching size, blocking number, LSTM case values are generated. These optimized parameters will be compared with the input to check if the available resources from the targeted FPGA platform meet the requirements. If acceptable, the parameters will be used in the hardware generation to produce the FPGA designs based on the hardware RTL templates.

6 Evaluation

6.1 Experimental Setup

LSTM has many variants that target different applications. In our work, we choose the LRCN [6] which covers video activity recognition to demonstrate our approach. Generally, the LRCN is implemented using a CNN to extract a fixed-length feature vector which is passed to a recurrent sequence learning component, such as an LSTM. In this work, we choose the features of each video frame from the average pool layer of an Inception-v3 model which has been pre-trained on the ImageNet data-set. One additional Fully Connected (FC) layer is used to produce 1792 features which are 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 [6].

To demonstrate 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 an XC7Z045 FPGA and a dual ARM Cortex-A9 processor. 1 GB DDR3 RAM is included in the platform as off-chip memory. There is only 19.2 Mb on-chip memory on the XC7Z045 FPGA while the weights in this LSTM model are more than 32Mb which are too large to store in the on-chip memory of this FPGA device. Thus, the weights have to be stored in the off-chip memory and reused in a smart way. We also implement the LSTM-512 model which has 512 hidden units using the Virtex 7 VX690T FPGA.

6.2 Resource Utilization

The resource utilization is shown in Table 2 for our SBE design on the Zynq 7045 FPGA. The number of PEs, N_{pe}, is set to 1024 targeting LSTM-256 while the batch size is tuned to 64. N_t is 16 when the DMA data bus is 256-bit and the LSTM datapath is 16-bit. If the DMA data bus is 512-bit then the proper batch size is 32 and this does not change the resource utilization of BB-FIFO because each LUTRAM is 64 bits. N_t needs scaling if the DMA data bus works under a different frequency with computation engines. The number of PPs is 4. For our system on Zynq, almost all the FPGA's hardware resources are used. A few multiplication units are implemented using LUTs because there are only 900 DSP elements in our system. Note that there are 2048 PEs for 2048 for LSTM-512 targeting Virtex 7 VX690T FPGA because this device has an abundance of DSPs.

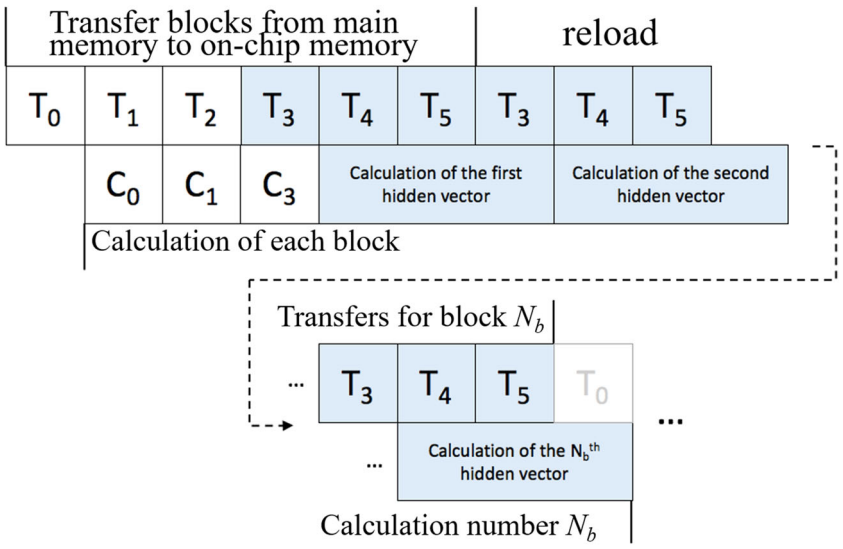


Figure 11 Timing diagram for Case 3.

Table 2 Resource utilization.

		LUT	LR[1]	FF	BRAM	DSP
Z[2]	Avail.	219k	70k	437k	545	900
	Used	166k	49k	150k	517.5	900
	Utili.	75.8%	69.9%	34.4%	94.9%	100%
V[3]	Avail.	433k	174k	866k	1470	3600
	Used	204k	71k	222k	1070	2060
	Utili.	47%	41%	25.6%	72.8%	57%

[1]LUTRAM
[2]Zynq 7045
[3]Virtex 7 690T

6.3 Design Parameter Tuning

The idea is to tune the design parameters mentioned above to achieve the optimal performance for the proposed LSTM accelerator.

Batch size tuning The best batch size is determined by balancing the computation time and communication time from the off-chip memory 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. 9. 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.

Blocking number tuning Since only two blocks are stored on the FPGA, when the blocking number increases, the block size decreases, and then the required on-chip memory decreases for a given LSTM model. This means a large LSTM system can be processed efficiently even with a small FPGA. However, as we discussed in Section 5.1 the blocking number cannot be too large for a given system as performance can be reduced as shown in Case 3. The LRCN performance with different blocking numbers on the Xilinx ZC706 platform is shown in Fig. 13. 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. 13, 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 the amount of on-chip memory will be required. Furthermore, if users can tolerate reduced performance then they can choose a smaller and cheaper FPGA as shown in Fig. 13.

6.4 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 with 1024 frames in total. Compared with the LRCN on CPU and GPU, our Zynq FPGA design achieves the same

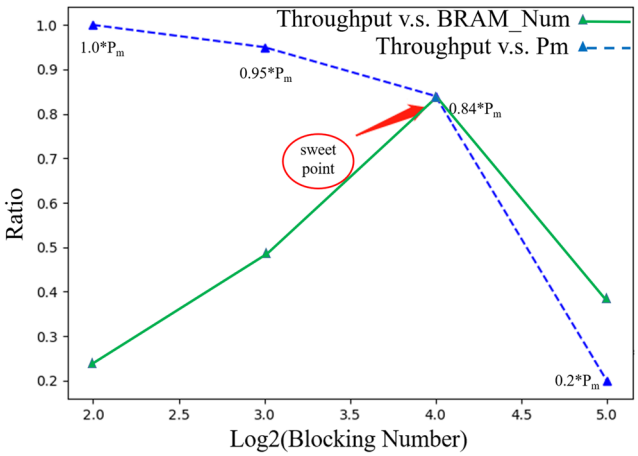


Figure 13 Throughput vs Blocking Number on ZYNQ 7045.

Table 3 Performance comparison of the FPGA design versus CPU and GPU.

	CPU	GPU	This Paper	This Paper
Platform	Xeon E5-2665	TITAN X Pascal	Virtex 7 VX690T	Zynq 7Z045
Frequency	2.4 GHz	1.6 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, u gates

[2]Each sample/video has 32 frames

accuracy. Besides, our design is 23.7 and 1.3 times faster and consumes 208 and 19.2 times less power respectively as shown in Table 3.

Parameterizable performance scaling for various LSTM sizes and batch sizes is demonstrated and shown in Fig. 14. 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. 15. In addition, Fig. 13 shows the performance scaling for different blocking numbers. These results show the customizability of our architecture for various scenarios.

Some existing FPGA-based designs of LSTM accelerators are compared with ours in Table 4 to illustrate the benefits of our proposed approach. For a fair comparison, we only show the previous work involving detailed implementation of the LSTM system storing the weights in the external memory of FPGA. The table lists the FPGA chips, model storage, precision, run-time frequency, throughput, power efficiency and resource efficiency. It contains a range of designs across this parameter space for comparison. Because of our novel architecture which can reuse the fetched weights and reduce off-chip memory access, our design achieves power efficiency of 20.84 GOPS/W and resource efficiency of 0.246 GOPS/DSP which are the highest with respect to state-of-the-art FPGA implementations of dense LSTM models with weights stored in off-chip memory. With a similar number of DSP resources to [14], our system using Virtex 7 achieves 356 GOPS which is the highest performance among all the FPGA implementations of LSTMs storing weights the off-chip memory. Because of routing congestions, our Virtex 7 design only runs at 125MHz. We believe that our implementation can achieve higher operating frequencies with further low-level optimizations. However, we leave that for future work since it has a very limited impact on the conclusions we draw from our study in this paper.

Note that our comparison does not cover recent approaches [3, 13, 29] about LSTM acceleration adopting model compression and weight pruning to make the best use of on-chip memory. Such techniques are orthogonal to our proposed approach. Future work will explore how these techniques can further enhance the effectiveness of the proposed approach.

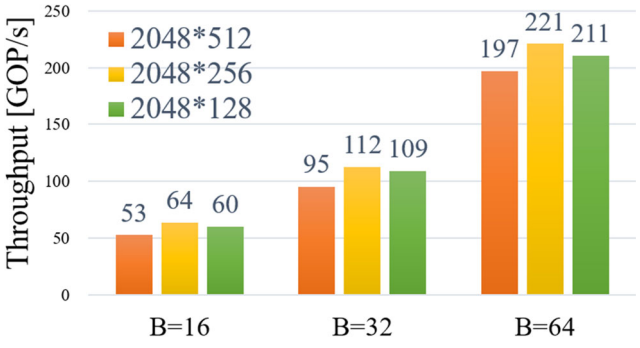


Figure 14 Throughput depending on batch size on ZYNQ 7045.

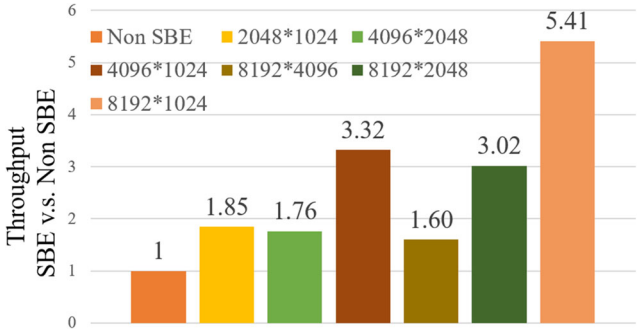


Figure 15 Throughput of our design v.s non SBE design for very large LSTM systems on ZYNQ 7045.

Table 4 Comparison with previous implementations of Dense LSTM models storing weights on off-chip memory.

FPGA	Chang [4] Zynq 7Z020	Guan [15] Virtex 7 VX485T	ESE [16] Kintex KU060	FP-DNN [14] Stratix V GSMD5	This Paper Virtex 7 VX690T	This Paper Zynq 7Z045
Model Storage				off-chip		
Prec. (bits)	16	32[a]	12	16 32[a]	16	16
DSP Number	220	2800	2760	3180[b]	3600	900
Freq. (Mhz)	142	150	200	150	125	142
Perf. (GOPS)	0.47	7.26	282[c]	316 86[a]	356	221
Power Effi. (GOPS/W)	0.268	0.37	6.87	12.63 3.44[a]	13.48	20.84
Resource Effi.[d] (GOPS/DSP)	0.002	0.003	0.102	0.099 0.027[a]	0.099	0.246

[a]Floating point

[b]One Intel FPGA DSP includes two 18*18 multipliers

[c]Dense Model

[d]To make a fair comparison, the total number of DSP in device is used to calculate GOPS/DSP when evaluating LSTM accelerator

7 Conclusions and Future Work

In this paper, we propose a Stall-free Blocking-batching Engine (SBE) architecture with a framework of automatic hardware mapping for optimizing the inference design of large LSTM models on FPGAs. The proposed accelerator is implemented using Zynq and Virtex-7 FPGAs and achieves excellent performance and efficiency, which shows the effectiveness of our approach. Further research includes combining the proposed SBE with pruning methods to allow large, sparse models to run on small embedded FPGAs.

Acknowledgments The support of the United Kingdom EPSRC (grant numbers EP/L00058X/1, EP/P010040/1, EP/N031768/1, EP/L016796/1 and EP/S030069/1), Natural Science Foundation of China (U1831118), Shanghai Municipal Commission of Science and Technology (19511131202) and Pudong Science and Technology Development Fund (PKX2019-D02), Xilinx and Corerain Technologies is gratefully acknowledged.

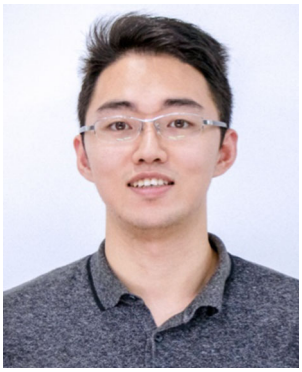
References

1. Amin, H., et al. (1997). Piecewise linear approximation applied to nonlinear function of a neural network. IEEE Proceedings-Circuits, Devices and Systems, 144.
2. Ardakani, A., Ji, Z., Gross, W.J. (2018). Learning to skip ineffectual recurrent computations in LSTMs. arXiv:1811.10396.
3. Cao, S., et al. (2019). 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*.
4. Chang, A.X.M., Martini, B., Culurciello, E. (2015). Recurrent neural networks hardware implementation on FPGA. arXiv:1511.05552.
5. Cooijmans, T., Ballas, N., Laurent, C., Gülçehre, Ç., Courville, A. (2016). Recurrent batch normalization. arXiv:1603.09025.
6. Donahue, J., et al. (2015). Long-term Recurrent Convolutional Networks for Visual Recognition and Description. In *Proceedings of the IEEE conference on computer vision and pattern recognition*.
7. Eyben, F., et al. (2009). From speech to letters-using a novel neural network architecture for grapheme based ASR. In *Automatic speech recognition & understanding, 2009. ASRU 2009. IEEE workshop*.
8. Fan, H. et al. (2018). A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA. International conference on field-programmable technology (FPT). IEEE.
9. Fan, H., et al. (2018). Reconfigurable Acceleration of 3d-CNNs for Human Action Recognition with Block Floating-Point Representation. In *28th international conference on field programmable logic and applications (FPL). IEEE*.
10. Fan, H., et al. (2019). F-E3D: FPGA-based Acceleration of an Efficient 3D Convolutional Neural Network for Human Action Recognition. In *IEEE 30Th international conference on application-specific systems, architectures and processors (ASAP). IEEE*.

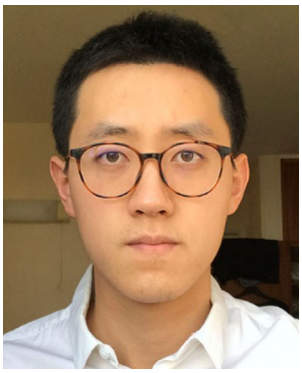
11. Ferreira, J.C., et al. (2016). An FPGA implementation of a long short-term memory neural network. In *Reconfigurable computing and FPGAs (reconfig). IEEE.*
12. Fowers, J., et al. (2018). A configurable Cloud-Scale DNN processor for Real-Time AI. In *Proceedings of the 45th Annual International Symposium on Computer Architecture*.
13. Gao, C., et al. (2018). DeltaRNN: A power-efficient recurrent neural network accelerator. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM*.
14. Guan, Y., et al. (2017). 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.*
15. Guan, Y., et al. (2017). FPGA-based Accelerator for Long Short-Term Memory Recurrent Neural Networks. In *22Nd asia and south pacific design automation conference (ASP-DAC). IEEE*.
16. Han, S., et al. (2017). ESE: Efficient speech recognition engine with sparse LSTM on FPGA. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*.
17. Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term memory. *Neural computation*, *9*(8), 1735–1780.
18. Jacob, B., et al. (2018). Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 2704–2713*.
19. Khurram Soomro, A.R.Z., & Shah, M. (2012). UCF101: A dataset of 101 human action classes from videos in the wild CRCV-TR-12-01.
20. Li, Z., et al. (2019). E-RNN: Design optimization for efficient recurrent neural networks in FPGAs. In *International symposium on high performance computer architecture (HPCA). IEEE.*
21. Nurvitadhi, E., et al. (2016). Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC. In *International conference on field-programmable technology (FPT). IEEE.*
22. Nurvitadhi, E., et al. (2016). Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC. In *26th international conference on field programmable logic and applications (FPL). IEEE.*
23. Nurvitadhi, E., et al. (2017). Can FPGAs beat GPUs in accelerating next-generation deep neural networks? In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays ACM*.
24. Nurvitadhi, E., et al. (2019). Why Compete When You Can Work together: FPGA-ASIC Integration for Persistent RNNs. In *27th annual international symposium on field-programmable custom computing machines (FCCM). IEEE.*
25. Que, Z., et al. (2019). Efficient Weight Reuse for Large LSTMs. In *30th international conference on application-specific systems, architectures and processors (ASAP). IEEE.*
26. Que, Z., et al. (2019). Real-time Anomaly Detection for Flight Testing using AutoEncoder and LSTM. In *International conference on field-programmable technology (FPT). IEEE.*
27. Que, Z., et al. (2020). Optimizing reconfigurable recurrent neural networks. In *Field-programmable custom computing machines (FCCM). IEEE.*
28. Rizakis, M., et al. (2018). Approximate FPGA-based LSTMs under computation time constraints. arXiv:1801.02190.
29. Rybalkin, V., Pappalardo, A., Ghaffar, M.M., Gambardella, G., Wehn, N., Blott, M. (2018). FINN-L: Library extensions and design trade-off analysis for variable precision LSTM networks on FPGAs. In *28th international conference on field programmable logic and applications (FPL). IEEE.*
30. Rybalkin, V. et al. (2017). 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*.
31. Sun, Z., et al. (2018). FPGA Acceleration of LSTM based on data for test flight. In *International conference on smart cloud (smartcloud). IEEE.*
32. Wang, S., et al. (2018). C-LSTM: Enabling efficient LSTM using structured compression techniques on FPGAs. In *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays. ACM*.
33. Xiong, P., et al. (2018). Application of transfer learning in continuous time series for anomaly detection in commercial aircraft flight data. In *International conference on smart cloud (smartcloud). IEEE.*
34. Yue-Hei Ng, J., et al. (2015). Beyond short snippets: Deep networks for video classification. In *Proceedings of the conference on computer vision and pattern recognition. IEEE.*
35. Zhang, M., et al. (2018). DeepCPU: Serving RNN-based deep learning models 10x faster. In: 2018 {USENIX} Annual technical conference ({USENIX}{ATC} 18).
36. Zhang, X., et al. (2017). High-performance video content recognition with long-term recurrent convolutional network for FPGA. In *Field programmable logic and applications (FPL). IEEE.*
37. Zhao, R., et al. (2017). Optimizing CNN-based object detection algorithms on embedded FPGA platforms. In *International symposium on applied reconfigurable computing. Springer*.

Zhiqiang Que is a research assistant pursuing the Ph.D. degree in the department of Computing, Imperial College London, UK. He received his B.S in Microelectronics and M.S in CS from Shanghai Jiao Tong University in 2008 and 2011 respectively. From 2011 to 2016, he worked on microachitecture design of ARM-compliant CPUs with the Marvell semiconductor Ltd., Shanghai. His research interests include computer architectures, embedded systems, high-performance computing and computer-aided design (CAD) tools for hardware design optimization.

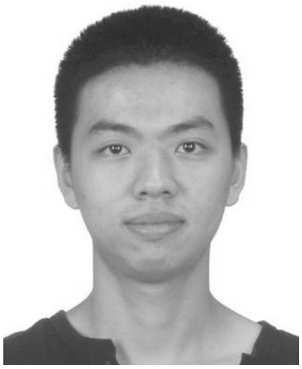
Yongxin Zhu received his Ph.D. in Computer Science from National University of Singapore, in 2001. He was with the National University of Singapore as a Research Fellow from 2002 to 2005, and with the School of Microelectronics, Shanghai Jiao Tong University as an Associate Professor from 2006 to 2017. In 2017, he joined the Shanghai Advanced Research Institute, Chinese Academy of Sciences, as a Full Professor. He is also an Adjunct Professor with Shanghai Jiao Tong University and University of Chinese Academy of Sciences. He has authored more than 150 English papers and 50 Chinese papers. His research interests include computer architectures, system-level IC design, big data processing and block chain. Dr. Zhu is a senior member of IEEE, a distinguished member of China Computer Federation. He has served more than 40 conferences and journals as an Editor, Program Chair, Publicity Chair and TPC Member.



Hongxiang Fan received master's degree in advance computing from Imperial College London in 2018. He is currently pursuing the Ph.D. degree in computer science with Imperial College London, United Kingdom. His current research interests include machine learning and high-performance computing.



Jiuxi Meng received the B.Eng. degree in electronic engineering from the University of Southampton in 2016 and master's degree in analog and digital circuit design from Imperial College London in 2017. He is currently pursuing the Ph.D. degree in the department of computing in Imperial College London. His research interests include field programmable gate array system design, data center architecture and cloud computing.



Xinyu Niu received the Degree from Fudan University, Shanghai, China, and the M.Sc. and Ph.D. degrees from Imperial College London, London, U.K. His current research interests include developing applications and tools for reconfigurable computing that involves runtime reconfiguration.

Wayne Luk received the M.A., M.Sc., and D.Phil. Degrees in Engineering and Computing Science from the University of Oxford, Oxford, U.K. He is a Professor of Computer Engineering with Imperial College London, London, U.K. He was a Visiting Professor with Stanford University, Stanford, CA, USA. His current research interests include theory and practice of customizing hardware and software for specific application domains, such as multimedia, networking, and finance.