Optimizing Reconfigurable Recurrent Neural Networks

Zhiqiang Que*, Hiroki Nakahara†, Eriko Nurvitadhi‡
Hongxiang Fan*, Chenglong Zeng§, Jiuxi Meng*, Xinyu Niu§, Wayne Luk*
† Tokyo Institute of Technology, Japan, nakahara.h.ad@m.titech.ac.jp
‡ Intel Corporation, eriko.nurvitadhi@intel.com
§ Corerain Technologies Ltd., Shenzhen, China, {chenglong.zeng, xinyu.niu}@corerain.com
*Dept. of Computing, Imperial College London, UK, {z.que, h.fan17, jiuxi.meng16, w.luk}@imperial.ac.uk

***Abstract*—This paper proposes a novel latency-hiding hardware architecture based on column-wise matrix-vector multiplication to eliminate data dependency, improving the throughput of systems of RNN models. In addition, a flexible checkerboard tiling strategy is introduced to allow large weight matrices, while supporting element-based parallelism and vector-based parallelism. These optimizations improve the exploitation of the available parallelism to increase run-time hardware utilization and boost inference throughput. Furthermore, a quantization scheme with fine-tuning is proposed to achieve high accuracy. Evaluation results show that the proposed architecture can enhance performance and energy efficiency with little accuracy loss. It achieves 1.05 to 3.35 times better performance and 1.22 to 3.92 times better hardware utilization than a state-of-the-art FPGA-based LSTM design, which shows that our approach contributes to high performance FPGA-based LSTM systems.**

I. Introduction

Recurrent Neural Networks (RNNs) have been shown to have useful properties with many significant applications. Since they can record previous information to increase prediction accuracy, RNNs are applied to sequence processing problems such as speech recognition [1, 2], natural language processing [3] and video classification [4, 5]. Among the many RNN variants, the two most popular ones are Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU). FPGAs have been used to speed up the inference of RNNs [1, 6, 7, 8, 9], which offer benefits of low latency and low power consumption compared to CPUs or GPUs.

However, the data dependency in RNN computation makes systems stall until the required hidden vector returns from the full pipeline to start the next time-step calculation. Moreover, deep pipelining is often used to achieve a high operating frequency, which makes stall penalty worse since the system pipeline needs to be emptied. Besides, an inefficient tiling strategy also makes hardware resources idle. The design in [8] involves 6 matrix-vector multiplication (MVM) “tile engines”, each processing 400×400 matrices. The tile engine results are fed to an adder tree, so they are effectively processing a 400×2400 matrix in parallel. Any MVM that does not map to this dimension will leave some resources idle. Both of these issues result in low hardware utilization, as shown in Fig 1. The hardware utilization of Brainwave [8] is lower than 50% for all the LSTM models. The BrainWave-like Neural Processor Unit (NPU) [9] with a fine-grained zero-padding scheme also suffers from low hardware utilization, especially in LSTM models with medium sizes which are commonly used in many applications [10, 11, 12].

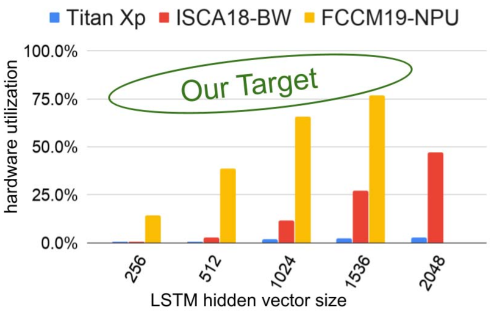


Fig. 1. Low hardware utilization of some existing LSTM Designs (ISCA18-BW [8] and FCCM19-NPU [9])

This paper proposes a novel latency-hiding hardware architecture and a flexible checkerboard tiling strategy for RNNs/LSTMs to improve hardware utilization and boost inference throughput. First, we propose column-wise matrix-vector multiplications (MVM) for RNN/LSTM gates operations, which can eliminate the data dependency. The column-wise block-striped decomposition of a matrix, as shown in Fig. 2, is an effective parallel method of MVM used in high-performance computing. However, most of the previous FPGA-based designs of RNNs focus on row-wise MVM. In our architecture, the calculation of the next time-step can start without waiting for the system pipeline to be emptied, which means that the system can be fully pipelined without stall since it only needs a partial input vector to start the computation. Besides, in the row-wise MVM, the vector needs to be replicated so that the dot products can be computed in parallel, while for the column-wise MVM, each element is replicated for each column and so can be more efficient using current FPGA architectures.

Moreover, we also propose a flexible checkerboard tiling strategy incorporating Element-based Parallelism (EP) and Vector-based Parallelism (VP) to boost inference throughput.

2576-2621/20/$31.00 ©2020 IEEE
DOI 10.1109/FCCM48280.2020.00011

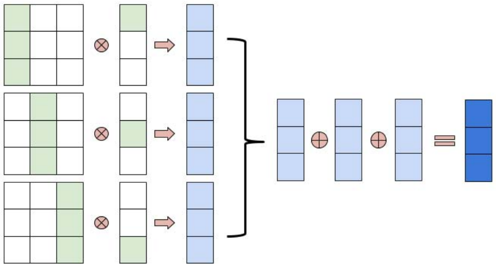


Fig. 2. Column-wise matrix-vector multiplication

To support EP and VP, we propose a hybrid hardware architecture that combines a multiplier-adder-tree and a multiply-accumulator. The architecture that deploys many parallel multipliers followed by a balanced adder-tree is commonly used in FPGA-based RNN/LSTM accelerators [8, 9, 13, 14, 15]. These designs are based on row-wise block-striped decomposition of MVM. For the column-wise MVM we propose an architecture deploying many parallel multipliers followed by accumulators since the partial result vectors of column-wise MVM are output from the pipeline cycle by cycle and then these partial result vectors are accumulated one by one, as shown in Fig. 2. Furthermore, a small balanced adder tree is placed between the multipliers and the accumulators, which balances the parallelism of EP and VP to increase throughput.

We make the following contributions in this paper:

- A novel column-wise MVM for RNNs with latency hiding to increase the hardware utilization and system throughput.
- A flexible checkerboard tiling strategy incorporating EP and VP to exploit the available parallelism and further increase the hardware utilization and scalability.
- A comprehensive evaluation of the proposed method and hardware architecture.

II. Background and Preliminaries

A. RNNs

RNNs are artificial neural networks which have feedback connections and internal memory cells to record past information about long-term dependencies over an arbitrary time. They achieve high accuracy in many sequence processing problems such as text analysis, speech recognition and video classification.

LSTM was initially proposed in 1997 by Sepp Hochreiter and Jürgen Schmidhuber [16]. This study follows the standard LSTM cell [1, 8, 9, 12], as shown in Fig 3. The hidden state h_t is produced by the following equations:

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

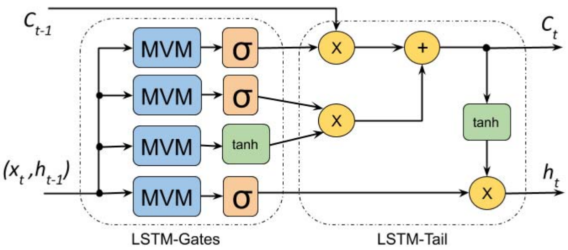


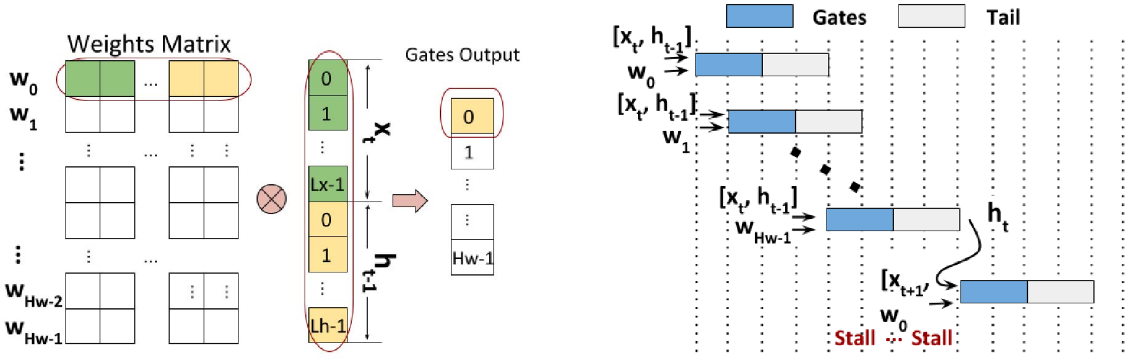
Fig. 3. Structure of an LSTM Cell

TABLE I
System Parameters

W	Weights matrix
w_n	All the weights of the row number n in W
w'_n	All the weights of the column number n in W
$W_i[n], W_f[n],$ $W_g[n], W_o[n]$	Row n in $i, f, g,$ or o gate weights matrices
Hw	Number of columns of weight matrix
Lw	Number of Rows of weight matrix
x_t	The input vector x at timestep t
h_t	The hidden vector h at timestep t
$x_t[j]$	The element j of input vector x at timestep t
$h_t[j]$	The element j of hidden vector h at timestep t
Lx	Number of elements in input vector x
Lh	Number of elements in hidden vector h
NPE	Number of processing elements
EP	Element-based Parallelism
VP	Vector-based Parallelism
TS	Timestep

Here, σ, $tanh$ and $\odot$ stand for the sigmoid function, the hyperbolic tangent function and element-wise multiplication respectively. i, f, g and o represent the input, forget, input modulation and 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 weight matrix for both input and hidden units. Bias is represented as b. The output c_t is the internal memory 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. The gates control the information flow inside the LSTM unit. The input gate decides what new information is to be written into the memory cell; the forget gate decides what old information is no longer needed and can be removed; the input modulation gate is used to modulate the information that the input gate will write into the memory Cell by adding non-linearity to the information; the output gate decides what the next hidden state should be.

Gated Recurrent Unit (GRU) is a variant of LSTM. It combines the forget and input gates into a single "update gate" and it has fewer parameters than the standard LSTM models. Our work focuses on the optimization of the standard LSTM and GRU but the proposed techniques can be applied to other RNN and LSTM variants.



(a) Row-wise matrix-vector multiplication

(b) Pipeline analysis

Fig. 4. Row-wise matrix-vector multiplication, showing the data dependency of LSTM

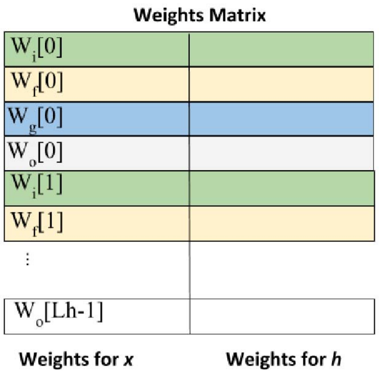


Fig. 5. The weights matrix, showing interlacing of $W_i[n]$, $W_f[n]$, $W_g[n]$ and $W_o[n]$

III. Design and Optimization Methodology

This section analyzes the data dependency problem and introduces several optimizations for RNN designs. We define the system parameters in Table I which are used for later calculations.

A. *Weights Matrix of LSTM Gates*

The four matrices of i, f, o, u gates of LSTMs share the same size. In our design, they are combined into one large matrix. 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. Since each gate matrix has the size of $Lh \times (Lx + Lh)$, the large combined matrix has the size of $(4 \times Lh) \times (Lx + Lh)$. Then we have the following equations:

$$Hw = 4 \times Lh \qquad (1)$$

$$Lw = Lh + Lx \qquad (2)$$

Besides, the weights of the four LSTM gates are interlaced in the matrix. For example, the first four rows of our weights matrix W are respectively the first row of the weights matrix for the input gate, forget gate, input modulation gate and output gate, as shown in Fig. 5. Therefore, the related elements in the result vector from four gates are adjacent and can be reduced via the element-wise operations in the LSTM-tail unit.

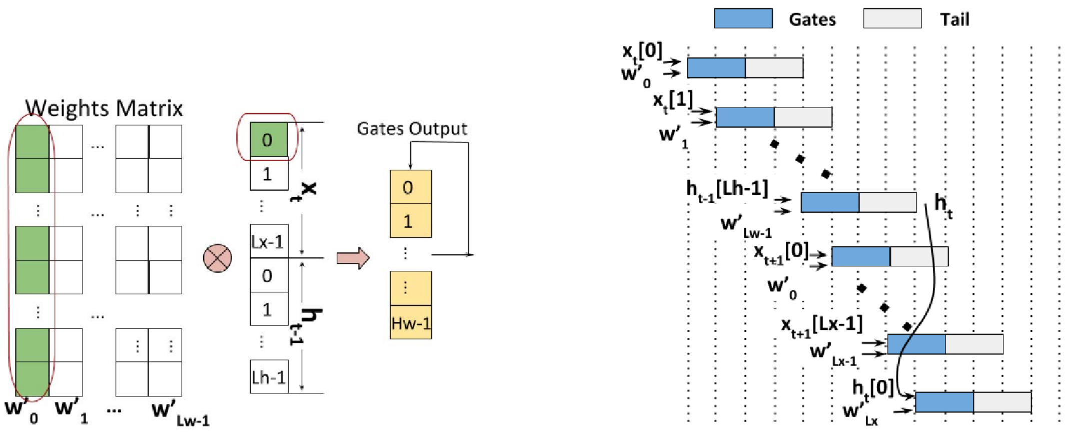
B. *Conventional design of MVM for RNNs and its problem*

The conventional implementation of matrix-vector multiplication (MVM) for RNNs is row-wise, and it involves the entire vector of (x_t, h_{t-1}) and an entire row of the weights at a time. However, this approach imposes additional stalling as the system needs to wait for newly computed hidden vector before starting the next time-step.

Data hazard exists because the whole new hidden vector h_t is required to start the new computation of x_{t+1} in the conventional design of MVM for RNN/LSTM. 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. 4. It indicates that the whole system pipeline needs to be emptied to get the new computed hidden vector h_t before the new matrix-vector operations can start. As [8] mentions, RNN programs have a critical loop-carry dependence on the h_t vector. If the full pipeline cannot return h_t to the vector register file in time to start the next time-step then the MVM unit will stall, as shown in Fig. 4b. Therefore, pipeline latency is important. On the other hand, deep pipelining is needed to achieve a high operating frequency for the design. This makes it difficult to obtain designs with the best trade-off.

C. *The Proposed column-wise MVM for RNNs/LSTMs*

We propose a new technique that can alleviate this problem by calculating the matrix-vector operations in a column-wise 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. 6a. To illustrate the idea, the latency of the system in the figure is shown as four, however, the real system latency can be much larger. In addition, only one element in the x_t vector is selected to do the calculation in this figure; however, the actual number of the



(a) Column-wise matrix-vector multiplication

(b) Pipeline analysis

Fig. 6. Column-wise matrix-vector multiplication with LSTM data dependency eliminated

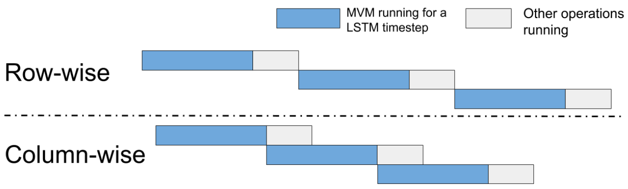


Fig. 7. Two example pipelines of a LSTM case with 3 timesteps utilizing row-wise MVM and column-wise MVM

involved elements of each cycle in the x_t vector depends on the parallelism of the system. The partial result vector comes from the small dot-product of the partial x_t vector and the corresponding weights. It is accumulated cycle by cycle to form the final result vector. 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 since it only needs a partial input vector, which means that the system can be fully pipelined without stall, as shown in Fig. 6b and Fig. 7. Each hidden vector can finish the computation in the shadow region of processing x_t before it is used.

As [17] mentions, the column-wise MVM only needs a partial input vector, but it produces the output vector later than row-wise since it waits for all the columns to be processed to get the final accumulated output vector. It seems that the succeeding hardware units that depend on the output vector (e.g., those that do activation functions and element-wise operations in the RNNs) would need to wait longer. While the row-wise MVM computes a subset of output vector completely before moving to the next subset. So, a subset of the final output vector is completed sooner than in a column-wise case. However, in the column-wise MVM, the succeeding units can get an entire output vector and not a subset. It does start the subsequent processing later than the one using row-wise MVM. However, increasing the number of succeeding units can help the system to finish the processing sooner than the row-wise case. Practically, we do not need to introduce many of these units since they will get only one input for a while.

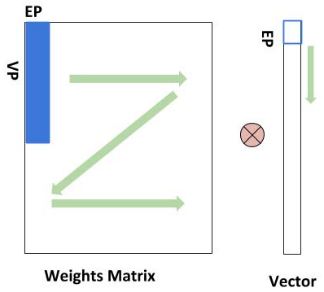


Fig. 8. The Element-based Parallelism (EP) and Vector-based Parallelism (VP) with a tile shaded in blue

When the x_t vector is small while the h_t vector is large, the system may still stall since the cycles of processing x_t vector can not fully cover the whole pipeline latency to get the h_t ready before it is needed. However, with the column-wise MVM, we can still process MVM of x_t and its corresponding weights when we are waiting for the h_t to be computed. While in the row-wise MVM, no new computation can process before h_t is computed. Moreover, when the input vector is short the LSTM models may not need a large hidden vector, otherwise it may bring overfitting easily.

D. Two Types of Parallelism and Tiling

To further exploit the available parallelism, we introduce Element-based Parallelism (EP) and Vector-based Parallelism (VP) in our design, as shown in Fig. 8. The weights matrix is tiled into small blocks with a size of (EP, VP). The proposed fine-grained tiling allows large weights to be processed sequentially. While within each tile, there is parallelism. In each cycle, our engine can process a tile of weights matrix and a sub-vector of $[x_t, h_{t-1}]$ with a size as EP.

EP and VP need to be carefully chosen so that the number of cycles to process the x_t vector, given by $\frac{Lx}{EP}$, is larger than the system latency to make sure the computation of hidden

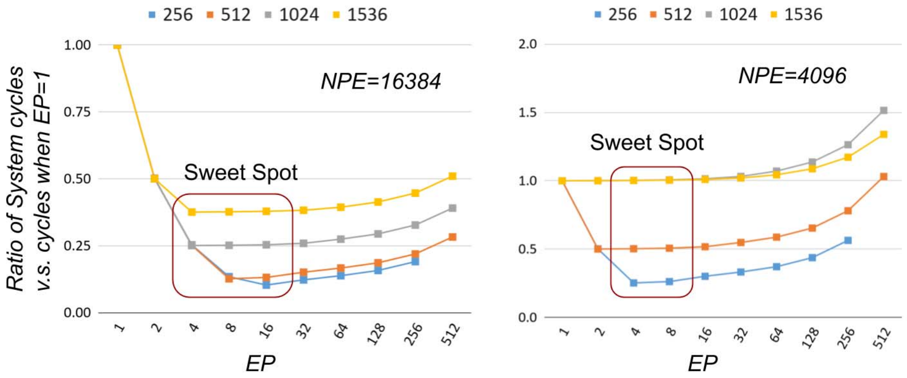


Fig. 9. System cycle number ratio depending on different EPs and different sizes of LSTM models

vectors can finish in the shadow of processing x_t vector. This number is small when the EP is large and it may still bring system stall. To increase system parallelism, VP is chosen to be as large as possible. However, the largest number of VP is Hw, which equals $4 \times Lh$, since there are only four gates in LSTM. GRU is smaller than LSTM since it has fewer gates than LSTM. In summary, the hardware utilization and system throughput can be improved via balancing EP and VP.

E. Design Space Exploration

When the previously discussed configurations are combined, we can characterize the hardware design space of a tiling block by (EP, VP) and the NPE, the number of processing elements. The effective performance varies with the number of PEs and tile size. To find out the optimal configuration parameters for our in-depth study, a cycle-accurate simulator is developed to conduct design space exploration. We propose a heuristic, greedy algorithm to explore design space. It starts with $EP = 1$ while the VP is given according to the system constraints shown in equations (3) and (4).

$$VP \leq Hw = 4 \times Lh \tag{3}$$

$$VP \leq \frac{NPE}{EP} \tag{4}$$

Practically, EP and VP should be as large as possible since when they increase the parallelism increases, which results in high throughput. However, when EP increases, the cycle number of processing the input vector (Lx) decreases so that the system may not have sufficient cycles to completely hide the processing of the hidden vector as we discussed in Section III-C. In our exploration, we set the VP as large as possible, which is $min(4 \times Lh, \frac{NPE}{EP})$. Fig. 9 presents the exploration results for different sizes of LSTM models, which are from 256 to 1536 with different colors, using our hardware design when the NPE is 4096 and 16384. The cycle number of processing determines the throughput of the system and the fewer, the better. As shown in Fig. 9, when EP is small, the processing cycle is large because the VP is constrained by equation (3) so that the effective PEs are less than NPE. From our result, the optimal configuration is a set of EP from 4 to 16. In these sweet spots, we can gain the highest parallelism which results in the highest system throughput.

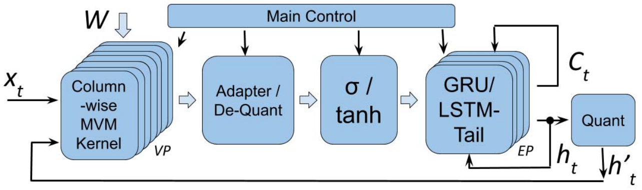


Fig. 10. System overview

IV. Hardware Architecture

The proposed hardware architecture for neural network implementation is presented in this section.

A. System details

Fig. 10 shows the overall system while Fig. 11 and Fig. 12 show the details of a computational kernel unit and an LSTM-tail unit respectively. There are VP kernels and each kernel has EP Processing Elements (PEs), so the effective number of PEs is $VP \times EP$. The VP and EP values are determined via the design space exploration in Section III-E. The Adapter is used to convert the parallelism between kernels and tails. Then, De-Quantization (De-Quant) is applied to convert quantized values into fixed-point values. The Sigmoid (σ) and hyperbolic tangent $(tanh)$ functions are implemented using lookup tables of size 2048 [2, 9]. The LSTM-tail unit as shown in Fig. 12 and GRU-tail unit mainly perform the element-wise operations. Several FIFOs in these tail units are utilized to synchronize the data and they are not shown in the figure. The output hidden vector (h_t) needs the quantization (Quant) before it can be used in the MVM kernels, so a Quant unit is utilized after the final output of LSTM-tail units as shown in Fig. 10.

Generally, the row-wise MVM is based on an architecture with parallel multipliers followed by a balanced adder tree. Accumulating the products of these multiplications is usually achieved using a balanced adder tree structure so that a number

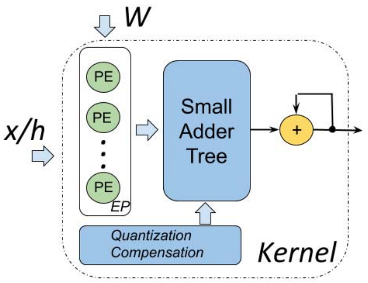


Fig. 11. The kernel unit

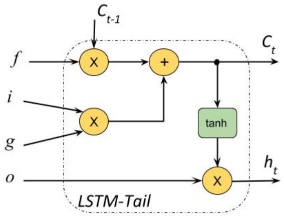


Fig. 12. The LSTM-tail unit

of related additions can be scheduled in parallel and the latency of the system can be minimized. The column-wise MVM is based on the architecture of parallel multipliers followed by parallel accumulators, since the elements in the partial result vector are not related. To support element-based parallelism, a small balanced adder tree is placed between the multipliers and the accumulators, as shown in Fig. 11. This adder tree can help to balance EP and VP to improve parallelism. Furthermore, each kernel has a component of quantization compensation to efficiently handle zero-points in quantized MVM operations [18].

B. FPGA DSP sharing for 8-bit multiplications

The DSP blocks in modern FPGAs, which are highly configurable, are often underutilized when implementing 8-bit RNN systems. [19] showed methods to extract two INT8 multipliers from Xilinx DSP48E2 Blocks which contain a 27x18 multiplier. [20] illustrated a method to pack 2 INT8 multiplications into one INT18 multiplier with extra ALMs. Both these methods require two multiplications to share one input operand. In our column-wise architecture, the computation order of MVM is different from the one in row-wise MVM. With our column-wise MVM using in RNN designs, one column of the weights matrix naturally shares the same element of the input vector and conducts the multiplications at the same time. Thus, these multiplications share one input operand, which helps us to pack four INT8 multiplications into one DSP blocks on Intel FPGAs [20] to reduce the hardware resources.

V. Quantization and Fine Tuning

A. Data Quantization

Numerous prior efforts [2, 8, 21, 22, 23] have shown that RNNs/LSTMs are robust to low bit-width quantization. Instead of using double or single precision floating-point representation, fixed-point representation can be used in FPGA-based RNN accelerators to achieve high performance and high power efficiency. In this work, we convert the input activations, the hidden units and the weights to 8-bit integers. We perform all arithmetic operations in fixed point and check that there is no significant accuracy degradation after fine-tuning is applied. To quantize and dequantize a real value r, we use the following mapping [18]:

$$r = S(q - z) \tag{5}$$

where scale S and z are our quantization parameters. The S is an positive real number given by $(r_{max} - r_{min})/(q_{max} - q_{min})$. Note that r_{max} and r_{min} are maximum and minimum values of a real value respectively; q_{min} and q_{max} represent the range of an 8-bit integer (0 and 255 in our implementation). The parameter S scales an RNN/LSTM network and z denotes a zero point. It is important to note that S is a floating-point number whereas the zero-point is of the same type as quantized values (q) which is an 8-bit integer. However, modern implementations often bypass this floating-point multiplication by approximation techniques shown to have a negligible effect on the accuracy of the net.

TABLE II
Accuracy Discussion

	LRCN Orig.[12] (Inception-v3+LSTM)	FPL'17 [27]	This work
Precision	Float 32-bit	Fixed 12-bit	Fixed 8-bit
Accuracy	70.36%	42.0%	70.10%

To maintain accuracy and avoid data overflow, we propose partial quantization [24, 25] to extend the bit-width of intermediate data. In this work, an 8-bit fixed-point data format is proposed to implement the multipliers in the LSTM gates. All the 16-bit products of the multiplications are passed to the adder tree to keep accuracy. The accumulators are 32-bit. The multipliers and adders for the element-wise operations in the LSTM-tail are all 16-bit fixed-point.

B. Fine Tuning

Quantization-aware fine-tuning [18] is applied to our quantized RNN/LSTM to recover accuracy. The gradient, weight, activation tensors are stored in floating-point. To emulate quantization error, all the operations are performed in a fixed-point manner. Therefore, the conversion 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 for the UCF101 dataset [26] with little accuracy loss. This is quantified in Table II.

VI. Evaluation and Analysis

This section presents hardware implementation results across two generations of Intel FPGAs that demonstrate the scalability of the proposed optimizations for RNNs.

TABLE III
RESOURCE UTILIZATION

	ALMs	M20K	DSP	Freq.
Arria 10 (1150)	186534 (44%)	1178 (43%)	1176 (77%)	259Mhz
Stratix 10 (2800)	487232 (52%)	10061 (86%)	4368 (76%)	260Mhz

TABLE IV
PERFORMANCE COMPARISON OF BW, FCCM19-NPU AND OUR DESIGN

benchmark		BW[8]	FCCM19 -NPU[9]	This Work
GRU	latency (ms)	0.013	0.00145	0.00058
h=512	HW Utilization	0.5%	21.7%	64.1%
TS=1	Perf.(TOPS)	0.25	2.17	5.46
GRU	latency (ms)	3.792	3.139	2.59
h=1024	HW Utilization	10.4%	60.2%	85.5%
TS=1500	Perf.(TOPS)	4.98	6.01	7.28
GRU	latency (ms)	0.951	1.454	1.36
h=1536	HW Utilization	23.3%	73.2%	91.4%
TS=375	Perf.(TOPS)	11.17	7.30	7.79
LSTM	latency (ms)	0.425	0.110	0.033
h=256	HW Utilization	0.8%	14.3%	56.1%
TS=150	Perf.(TOPS)	0.37	1.43	4.79
LSTM	latency (ms)	0.077	0.027	0.014
h=512	HW Utilization	2.8%	38.8%	85.9%
TS=25	Perf.(TOPS)	1.37	3.89	7.33
LSTM	latency (ms)	0.074	0.064	0.054
h=1024	HW Utilization	2.8%	65.7%	90.7%
TS=25	Perf.(TOPS)	5.68	6.56	7.73
LSTM	latency (ms)	0.145	0.246	0.236
h=1536	HW Utilization	27.1%	76.9%	94.1%
TS=50	Perf.(TOPS)	13.01	7.67	8.02

A. Experimental Setup

For our study, we choose our benchmark workloads from the DeepBench suite [8, 9] for a fair and direct comparison. It is a set of micro-benchmarks containing representative layers from popular DNN models such as DeepSpeech [28]. Two generations of Intel FPGAs, an Arria 10 1150 (A10) and Stratix 10 2800 (S10) are evaluated and compared with previous work. Both run persistent LSTM/GRU models of inference designed using Verilog RTL. Quartus Prime 18.1 is used to target both A10 and S10.

B. Resource Utilization

Table III shows the resource utilization of our designs with two configurations on FPGAs. We implement the configuration of (EP, VP) as (16, 1024) using a Stratix 10 FPGA which includes 16384 effective 8-bit multipliers in the MVM kernels. A small version with the configuration of (EP, VP) as (4, 1024) is implemented using an Arria 10 FPGA which has 4096 8-bit multipliers in the MVM kernels. Although we achieve a similar frequency to which reported in the original BW paper [8] and Intel-NPU [9], we believe that further low-level optimizations can be applied to our implementation to achieve even higher operating frequencies. We leave that for future work since it has a limited impact on the conclusions we draw from our study in this paper.

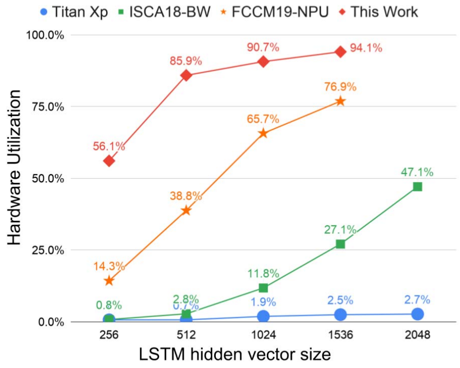


Fig. 13. Hardware utilization of LSTMs

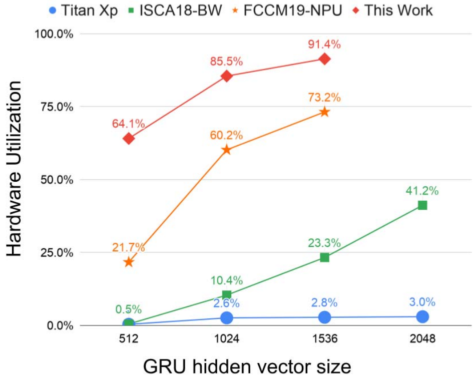


Fig. 14. Hardware utilization of GRUs

C. Case Study

RNNs have many variants that target different applications. In our work, we choose the LRCN [12] as a case study which is applied for video activity recognition. Generally, the LRCN is implemented using a CNN to extract a fixed-length vector of features which are then passed to a recurrent sequence learning component, such as an LSTM. 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 dataset. The LSTM part of the LRCN RGB model which has 256 hidden units is implemented. Quantization-aware fine-tuning [18] is applied to our quantized LSTM to recover accuracy as discussed in Section V-B with little accuracy loss as shown in Table II.

D. Performance and Efficiency Comparison

To illustrate the benefits of our proposed approach, some existing FPGA-based LSTM/GRU accelerator designs are compared with ours in Table IV and Table V. The DeepBench published results [8] on a modern NVIDIA Titan Xp GPU

TABLE V
COMPARISON WITH PREVIOUS IMPLEMENTATIONS OF LSTM

	2016[29]	2017[1]	2017[23]	ESE[2]	FP-DNN[14]	FINN-L[15]	BW[8]	FCCM19-NPU[9]	This work
FPGA	Virtex7 VX485T	Virtex7 VX485T	Zynq Z7045	Kintex KU060	StratixV GSMD5	Zynq ZU7EV	Stratix10 GX2800	Stratix10 GX2800	Stratix10 GX2800
Model Storage	on-chip	off-chip	on-chip	off-chip	off-chip	on-chip	on-chip	on-chip	on-chip
Precision (bits)	18 fixed	Float32	5 fixed	12 fixed	16 fixed	1-8 fixed	BFP8	8 fixed	8 fixed
DSP Used	-	1176	-	1504	1036	-	5245 (91%)	4880 (85%)	4368 (76%)
Frequency (MHz)	141	150	142	200	150	266	250	260	260
Performance (GOPS)	4.56	7.26	693	282	316	1833	370[a] 22620	1431[a] 7980	4790[a] 8015
Power Efficiency (GOPS/W)	-	0.37	55.88	6.87	12.63	-	180	118	129
LSTM Hardware Utilization	-	-	-	-	-	-	0.8%[a] 47.1%	14.3%[a] 76.9%	56.1%[a] 94.1%

[a] When targeting a small LSTM model (h=256).

is also included. For a fair comparison, we only show the previous work with a detailed implementation of the LSTM system. We show the latency, hardware (HW) utilization, throughput, FPGA chips, model storage, precision, run-time frequency, average throughput and power efficiency. Hardware utilization is the percentage of run time during which the hardware is not idle. With a similar number of DSP resources to [9], our design achieves 94.1% hardware utilization which is the highest with respect to state-of-the-art implementations on FPGAs, as shown in Fig. 13 and Fig. 14. Overall, our design provides over 1.05 to 3.35 times higher performance and 1.22 to 3.92 times higher hardware utilization than the state-of-the-art design [9] respectively, as shown in Table IV. The results show flexible customizability of our architecture for different scenarios.

VII. RELATED WORK

There has been much previous work on FPGA based RNN/LSTM implementations as shown in Table V. Rybalkin et al. [23] are the first to propose and design a 5-bit fixed-point BiLSTM hardware architecture for OCR. In their later work [15], FINN-L employs 1-8 bits as the quantized implementation which surpasses a single-precision floating-point accuracy for a given dataset. Guan et al. [1] propose a smart memory organization with on-chip double buffers to overlap computations with data transfers. An automated framework is proposed [14] for mapping CNNs and RNNs on FPGAs. [30] proposes the cross-kernel optimization within RNN cells targeting Plasticine [31], a coarse-grained reconfigurable architecture (CGRA). [8] proposes a Brainwave variant which is a single-threaded SIMD architecture for persistent RNNs. [9] introduces a Brainwave-like neural processing unit (NPU) for RNNs. They also propose TensorRAM for large persistent data-intensive RNN sequence models. All of these RNN designs are based on row-wise MVM and suffer from data dependency. Deploying the proposed latency-hiding hardware architecture involving column-wise MVM and the proposed flexible checkerboard tiling strategy, our design can achieve high throughput and hardware utilization. For the commonly used INT8 precision, we achieve a throughput of 8015 GOPS which is the highest with respect to state-of-the-art INT8 FPGA-based RNN designs. The only prior work that provides a higher throughput is [8] using 8-bit block floating-point. However when targeting small LSTM model, our throughput is 12.95 times higher than [8] and 3.35 times higher than [9]. Furthermore, we achieve the highest hardware utilization among all these designs across various LSTM models.

In addition, in [32, 15, 14], a batching technique is introduced to improve the performance and utilization of LSTM inference. Since our design executes a single input at a time, increasing batch size does not affect the utilization.

[9] also provides an INT4 design which achieves higher performance than an INT8 design using the same FPGA device. Some designs use binarised datapath [15, 33, 34]. Utilizing low precision is orthogonal to our proposed approach which transforms computation to eliminate data dependency. The technique of low precision is complementary to our approach to achieve even higher performance and efficiency. Since useful inference results may not be possible when bit-width is too small, we target INT8 using linear quantization [18] which is used in many DNN-based applications.

VIII. CONCLUSIONS AND FUTURE WORK

This paper proposes a novel latency-hiding hardware architecture based on column-wise MVM and a flexible checkerboard tiling strategy for RNNs/LSTMs to improve hardware utilization and boost inference throughput. We have implemented the proposed accelerator on Arria 10 and Stratix 10 FPGAs with superior performance and efficiency which show the effectiveness of our approach. Further research includes combining our method with in-memory computing and the automation of the proposed approach to enable rapid development of efficient RNN/LSTM designs.

ACKNOWLEDGEMENTS

The support of the United Kingdom EPSRC (grant numbers EP/L016796/1, EP/N031768/1, EP/P010040/1, and EP/S030069/1), Corerain and Intel are gratefully acknowledged.

References

[1] Y. Guan, Z. Yuan, G. Sun, and J. Cong, "FPGA-based accelerator for long short-term memory recurrent neural networks," in *Design Automation Conference (ASP-DAC), 2017 22nd Asia and South Pacific*. IEEE, 2017, pp. 629–634.

[2] S. Han *et al.*, "ESE: Efficient speech recognition engine with sparse LSTM on FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2017, pp. 75–84.

[3] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[4] J. Yue-Hei Ng, M. Hausknecht, S. Vijayanarasimhan, O. Vinyals, R. Monga, and G. Toderici, "Beyond short snippets: Deep networks for video classification," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 4694–4702.

[5] Z. Que, T. Nugent, S. Liu, L. Tian, X. Niu, Y. Zhu, and W. Luk, "Efficient Weight Reuse for Large LSTMs," in *2019 IEEE 30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, vol. 2160. IEEE, 2019, pp. 17–24.

[6] Z. Sun and et al., "FPGA acceleration of LSTM based on data for test flight," in *IEEE International Conference on Smart Cloud (SmartCloud)*, 2018, pp. 1–6.

[7] Z. Que *et al.*, "Real-time Anomaly Detection for Flight Testing using AutoEncoder and LSTM," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2019.

[8] 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.

[9] E. Nurvitadhi *et al.*, "Why Compete When You Can Work Together: FPGA-ASIC Integration for Persistent RNNs," in *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2019, pp. 199–207.

[10] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv preprint arXiv:1409.2329*, 2014.

[11] A. Graves, A.-r. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," in *IEEE international conference on acoustics, speech and signal processing*. IEEE, 2013, pp. 6645–6649.

[12] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 2625–2634.

[13] E. Nurvitadhi *et al.*, "Accelerating recurrent neural networks in analytics servers: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2016, pp. 1–4.

[14] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *Field-Programmable Custom Computing Machines (FCCM), 2017 IEEE 25th Annual International Symposium on*. IEEE, 2017, pp. 152–159.

[15] 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 *28th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2018.

[16] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[17] E. Nurvitadhi, A. Mishra, Y. Wang, G. Venkatesh, and D. Marr, "Hardware accelerator for analytics of sparse data," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*. EDA Consortium, 2016, pp. 1616–1621.

[18] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[19] "Deep Learning with INT8 Optimization on Xilinx Devices," 2017. [Online]. Available: https://www.xilinx.com/support/documentation/white papers/ wp486-deep-learning-int8.pdf

[20] M. Langhammer, B. Pasca, G. Baeckler, and S. Gribok, "Extracting INT8 Multipliers from INT18 Multipliers," in *International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2019.

[21] C. Gao, D. Neil, E. Ceolini, S.-C. Liu, and T. Delbruck, "Deltarnn: A power-efficient recurrent neural network accelerator," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 21–30.

[22] S. Cao, C. Zhang, Z. Yao, W. Xiao, L. Nie, D. Zhan, Y. Liu, M. Wu, and L. Zhang, "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, pp. 63–72.

[23] V. Rybalkin, N. Wehn, M. R. Yousefi, and D. Stricker, "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, pp. 1394–1399.

[24] H. Fan, S. Liu, M. Ferianc, H.-C. Ng, Z. Que, S. Liu, X. Niu, and W. Luk, "A Real-Time Object Detection Accelerator with Compressed SSDLite on FPGA," in *International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 14–21.

[25] H. Fan *et al.*, "F-E3D: FPGA-based Acceleration of an Efficient 3D Convolutional Neural Network for Human Action Recognition," in *30th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2019.

[26] A. R. Z. Khurram Soomro and M. Shah, "UCF101: A Dataset of 101 Human Action Classes From Videos in The Wild," *CRCV-TR-12-01*, November, 2012.

[27] X. Zhang, X. Liu, A. Ramachandran, C. Zhuge, S. Tang, P. Ouyang, Z. Cheng, K. Rupnow, and D. Chen, "High-performance video content recognition with long-term recurrent convolutional network for FPGA," in *Field Programmable Logic and Applications (FPL), 27th International Conference on*. IEEE, 2017, pp. 1–4.

[28] A. Hannun *et al.*, "Deep speech: Scaling up end-to-end speech recognition," *arXiv preprint arXiv:1412.5567*, 2014.

[29] J. C. Ferreira and J. Fonseca, "An FPGA implementation of a long short-term memory neural network," in *International Conference on ReConFigurable Computing and FPGAs (ReConFig),*. IEEE, 2016, pp. 1–8.

[30] T. Zhao, Y. Zhang, and K. Olukotun, "Serving recurrent neural networks efficiently with a spatial accelerator," *arXiv preprint arXiv:1909.13654*, 2019.

[31] R. Prabhakar *et al.*, "Plasticine: A reconfigurable architecture for parallel patterns," in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 2017.

[32] A. Ardakani, Z. Ji, and W. J. Gross, "Learning to skip ineffectual recurrent computations in lstms," *arXiv preprint arXiv:1811.10396*, 2018.

[33] H. Nakahara, H. Yonekawa, T. Fujii, and S. Sato, "A lightweight Yolov2: A binarized CNN with a parallel support vector regression for an FPGA," in *Proceedings of the ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2018, pp. 31–40.

[34] E. Nurvitadhi, D. Sheffield, J. Sim, A. Mishra, G. Venkatesh, and D. Marr, "Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU, and ASIC," in *2016 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2016, pp. 77–84.