ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA

Song Han[1,2], Junlong Kang[2], Huizi Mao[1,2], Yiming Hu[2,3], Xin Li[2], Yubin Li[2], Dongliang Xie[2]
Hong Luo[2], Song Yao[2], Yu Wang[2,3], Huazhong Yang[3] and William J. Dally[1,4]
[1] Stanford University, [2] DeePhi Tech, [3] Tsinghua University, [4] NVIDIA
[1] {songhan,dally}@stanford.edu, [2] song.yao@deephi.tech, [3] yu-wang@mail.tsinghua.edu.cn

ABSTRACT

Long Short-Term Memory (LSTM) is widely used in speech recognition. In order to achieve higher prediction accuracy, machine learning scientists have built increasingly larger models. Such large model is both computation intensive and memory intensive. Deploying such bulky model results in high power consumption and leads to a high total cost of ownership (TCO) of a data center.

To speedup the prediction and make it energy efficient, we first propose a *load-balance-aware pruning* method that can compress the LSTM model size by 20× (10× from pruning and 2× from quantization) with negligible loss of the prediction accuracy. The pruned model is friendly for parallel processing. Next, we propose a scheduler that encodes and partitions the compressed model to multiple PEs for parallelism and schedule the complicated LSTM data flow. Finally, we design the hardware architecture, named Efficient Speech Recognition Engine (ESE) that works directly on the sparse LSTM model.

Implemented on Xilinx XCKU060 FPGA running at 200MHz, ESE has a performance of 282 GOPS working directly on the sparse LSTM network, corresponding to 2.52 TOPS on the dense one, and processes a full LSTM for speech recognition with a power dissipation of 41 Watts. Evaluated on the LSTM for speech recognition benchmark, ESE is 43× and 3× faster than Core i7 5930k CPU and Pascal Titan X GPU implementations. It achieves 40× and 11.5× higher energy efficiency compared with the CPU and GPU respectively.

Keywords

Deep Learning; Speech Recognition; Model Compression; Hardware Acceleration; Software-Hardware Co-Design; FPGA

1. INTRODUCTION

Deep neural network is becoming the state-of-the-art method for speech recognition [6, 13]. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) are two popular types of recurrent neural networks (RNNs) used for speech recognition. In this work, we evaluated the most complex one: LSTM [14]. A similar methodology could be easily applied to other types of recurrent neural network.

FPGA '17, February 22 - 24, 2017, Monterey, CA, USA

ISBN 978-1-4503-4354-1/17/02…$15.00

DOI: http://dx.doi.org/10.1145/3020078.3021745

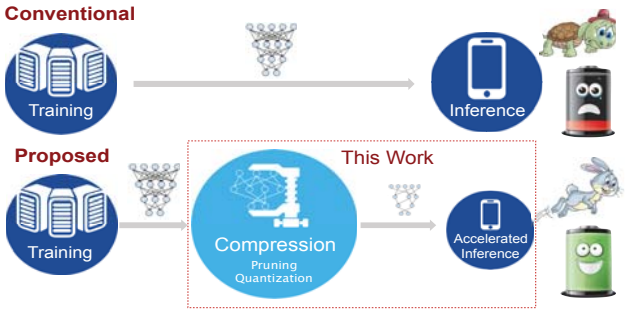


Figure 1: Proposed efficient DNN deployment flow: model compression+accelerated inference.

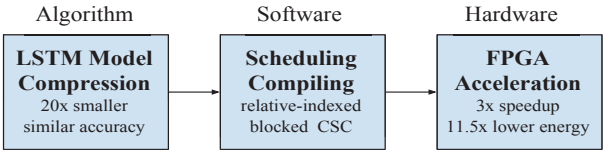


Figure 2: ESE optimizes LSTM computation across algorithm, software and hardware stack.

Despite the high prediction accuracy, LSTM is hard to deploy because of its high computation complexity and high memory footprint, leading to high power consumption. Memory reference consumes more than two orders of magnitude higher energy than ALU operations, thus our focus narrows down to optimizing the memory footprint.

To reduce memory footprint, we design a novel method to optimize across the algorithm, software and hardware stack: we first optimize the algorithm by compressing the LSTM model to 5% of it's original size (10% density and 2× narrower weights) while retaining similar accuracy, then we develop a software mapping strategy to represent the compressed model in a hardware-friendly way, finally we design specialized hardware to work directly on the compressed LSTM model.

The proposed flow for efficient deep learning inference is illustrated in Fig. 1. It shows a new paradigm for efficient deep learning inference, from Training=>Inference, to Training=>Compression=>Accelerated Inference, which

has advantage of inference speed and energy efficiency compared with conventional method. Using LSTM as a case study for the propose paradigm, the design flow is illustrated in Fig. 2.

The main contributions of this work are:

1. We present an effective model compression algorithm for LSTM, which is composed of pruning and quantization. We highlight our load balance-aware pruning and automatic flow for dynamic-precision data quantization.

2. The recurrent nature of RNN and LSTM produces complicated data dependency, which is more challenging than feedforward neural nets. We design a scheduler that can efficiently schedule the complex LSTM operations with memory reference overlapped with computation.

3. The irregular computation pattern after compression posed a challenge on hardware. We design a hardware architecture that can work directly on the sparse model. ESE achieves high efficiency by load balancing and partitioning both the computation and storage. ESE also supports processing multiple speech data concurrently.

4. We present an in-depth study of the LSTM and speech recognition system and did optimization across the algorithm, software, hardware boundary. We jointly analyze the trade-off between prediction accuracy and prediction latency.

2. BACKGROUND

Speech recognition is the process of converting speech signals to a sequence of words. As shown in Fig. 3, the speech recognition system contains the front-end and back-end, where front-end unit is used for extracting features from speech signals, and back-end processes the features and output the text. The back-end includes acoustic model (AM), language model (LM), and decoder. Here, Long Short-Term Memory (LSTM) recurrent neural network is used in the acoustic model.

The feature vectors extracted from front-end unit are processed by acoustic model, then the decoder uses both acoustic and language models to generate the sequence of words by maximum a posteriori probability (MAP) estimation, which can be described as

$$\hat{\mathbf{W}} = \arg\max_{\mathrm{W}} P(\mathbf{W}|\mathbf{X}) = \arg\max_{\mathrm{W}} \frac{P(\mathbf{X}|\mathbf{W})P(\mathbf{W})}{P(\mathbf{X})}$$

where for the given feature vector $\mathbf{X} = X_1 X_2 \dots X_n$, speech recognition is to find word sequence $\hat{\mathbf{W}} = W_1 W_2 \dots W_m$ with maximum posterior probability $P(\mathbf{W}|\mathbf{X})$. Because $\mathbf{X}$ is fixed, the above equation can be rewritten as

$$\hat{\mathbf{W}} = \arg\max_{\mathrm{W}} P(\mathbf{X}|\mathbf{W})P(\mathbf{W})$$

where $P(\mathbf{X}|\mathbf{W})$ and $P(\mathbf{W})$ are the probabilities computed by acoustic and language models respectively in Fig. 3 [20].

In modern speech recognition system, LSTM architecture is often used in large-scale acoustic modeling and for computing acoustic output probabilities. In the speech recognition pipeline, LSTM is the most computation and memory intensive part. Thus we focus on accelerating the LSTM.

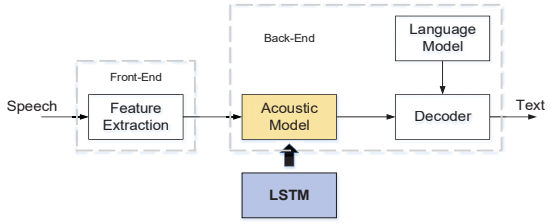


Figure 3: Speech recognition engine: LSTM is used in acoustic model. LSTM takes more than 90% time in the whole computation pipeline.

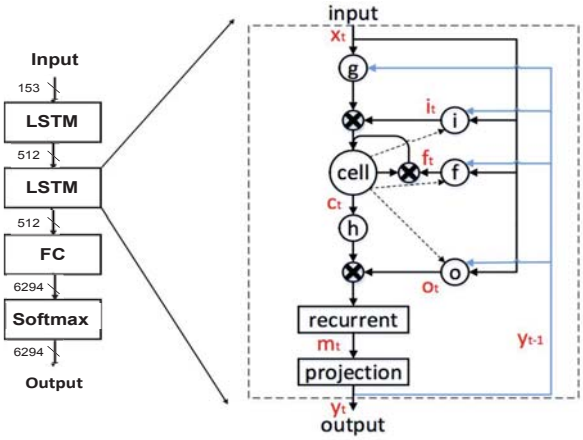


Figure 4: Data flow of the LSTM model.

The LSTM architecture is shown in Fig. 4, which is the same as the standard LSTM implementation [19]. LSTM is one type of RNN, where the input at time T depends on the output at $T-1$. Compared to the traditional RNN, LSTM contains special memory blocks in the recurrent hidden layer. The memory cells with self-connections in memory blocks can store the temporal state of the network. The memory blocks also contain special multiplicative units called gates: input gate, output gate and forget gate. As in Fig. 4, the input gate i controls the flow of input activations into the memory cell. The output gate o controls the output flow into the rest of the network. The forget gate f scales the internal state of the cell before adding it as input to the cell, which can adaptively forget the cell's memory.

An LSTM network accepts an input sequence $x = (x_1; \dots; x_T)$, and computes an output sequence $y = (y_1; \dots; y_T)$ by using the following equations iteratively from $t = 1$ to T:

$$i_t = \sigma(W_{ix}x_t + W_{ir}y_{t-1} + W_{ic}c_{t-1} + b_i) \quad (1)$$

$$f_t = \sigma(W_{fx}x_t + W_{fr}y_{t-1} + W_{fc}c_{t-1} + b_f) \quad (2)$$

$$g_t = \sigma(W_{cx}x_t + W_{cr}y_{t-1} + b_c) \quad (3)$$

$$c_t = f_t \odot c_{t-1} + g_t \odot i_t \quad (4)$$

$$o_t = \sigma(W_{ox}x_t + W_{or}y_{t-1} + W_{oc}c_t + b_o) \quad (5)$$

$$m_t = o_t \odot h(c_t) \quad (6)$$

$$y_t = W_{ym}m_t \quad (7)$$

Here the big O dot operator means element-wise multiplication, the W terms denote weight matrices (e.g. W_{ix} is the matrix of weights from the input to the input gate), and W_{ic}, W_{fc}, W_{oc} are diagonal weight matrices for peephole connections. The b terms denote bias vectors, while σ is the

logistic sigmoid function. The symbols i, f, o, c and m are respectively the input gate, forget gate, output gate, cell activation vectors and cell output activation vectors, and all of which are the same size. The symbols g and h are the cell input and cell output activation functions.

3. MODEL COMPRESSION

It has been widely observed that deep neural networks usually have a lot of redundancy [11, 12]. Getting rid of the redundancy won't hurt prediction accuracy. From the hardware perspective, model compression is critical for saving the computation as well as memory footprint, which means lower latency and better energy efficiency. We'll discuss two steps of model compression that consist of pruning and quantization in the next three subsections.

3.1 Pruning

In the pruning phase we first train the model to learn which weights are necessary, then prune away weights that are not contributing to the prediction accuracy; finally, we retrain the model given the sparsity constraint. The process is the same as [12]. In step two, the saliency of the weight is determined by the weight's absolute value: if the weight's absolute value is smaller than a threshold, then we prune it away. The pruning threshold is empirical: pruning too much will hurt the accuracy while pruning at the right level won't.

Our pruning experiments are performed on the Kaldi speech recognition toolkit [17]. The trade-off curve of the percentage of parameters pruned away and phone error rate (PER) is shown in Fig.6. The LSTM is evaluated on the TIMIT dataset [8]. Not until we prune away more than 93% parameters did the PER begins to increase dramatically. We further experiment on a proprietary dataset which is much larger: it has 1000 hours of training speech data, 100 hours of validation speech data, and 10 hours of test speech data, we find that we can prune away 90% of the parameters without hurting word error rate (WER), which aligns with our result on TIMIT dataset. In our later discussions, we use 10% density (90% sparsity).

3.2 Load Balance-Aware Pruning

On top of the basic deep compression method, we highlight our practical design consideration for hardware efficiency. To execute sparse matrix multiplication in parallel, we propose the load balance-aware pruning method, which is very critical for better load balancing and higher utilization on the hardware.

Pruning could lead to a potential problem of unbalanced non-zero weights distribution. The workload imbalance over PEs may cause a gap between the real performance and peak performance. This problem is further addressed in Section 4.

Load-balance-aware pruning is designed to solve this problem and obtain hardware-friendly sparse network, which produces the same sparsity ratio among all the submatrices. During pruning, we make efforts to avoid the scenario when the density of one submatrix is 5% while the other is 15%. Although the overall density is about 10%, the submatrix with a density of 5% has to wait for the other one with more computation, which leads to idle cycles. Load-balance-aware pruning assigns the same sparsity quota to submatrices, thus ensures an even distribution of non-zero weights.

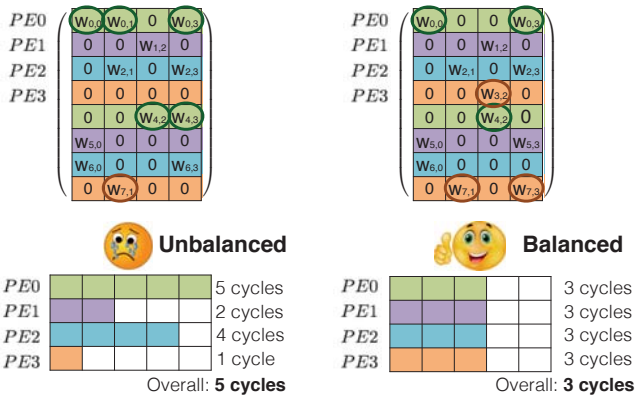


Figure 5: Load Balance Aware Pruning and its Benefit for Parallel Processing

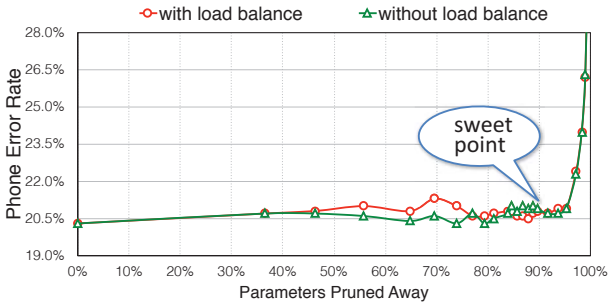


Figure 6: Accuracy curve of load-balance-aware pruning and original pruning.

As illustrated in Fig. 5, the matrix is divided into four colors, and each color belongs to a PE for parallel processing. With conventional pruning, PE_0 might have five non-zero weights while PE_3 may have only one. The total processing time is restricted to the longest one, which is five cycles. With load-balance-aware pruning, all PEs have three non-zero weights; thus only three cycles are necessary to carry out the operation. Both cases have the same non-zero weights in total, but load-balance-aware pruning needs fewer cycles. The difference of prediction accuracy with / without load-balance-aware pruning is very small, as shown in Fig. 6. There is some noise around 70% sparsity, so we put more experiments around 90% sparsity, which is the sweet point. We find the performance is very similar.

To show that load-balance-aware pruning still obtains comparable prediction accuracy, we compare it with original pruning on the TIMIT dataset. As demonstrated in Fig.6, the accuracy margin between two methods is within the variance of pruning process itself.

3.3 Weight and Activation Quantization

We further compressed the model by quantizing 32bit floating point weights into 12bit integer. We used linear quantization strategy on both the weights and activations.

In the weight quantization phase, the dynamic ranges of weights for all matrices in each LSTM layer are analyzed first, then the length of the fractional part is initialized to avoid data overflow.

The activation quantization phase aims to figure out the

Table 1: Weight Quantization under different Bits.

Weight Matrices[1]		Min	Max	Integer	Decimals		
					16bit	12bit	8bit
LSTM1	W_gifo_x[2]	-4.9285	5.7196	4	8	4	0
	W_gifo_r[2]	-0.6909	0.7140	1	11	7	3
	bias	-3.0143	2.1120	3	13	9	5
	W_ic	-0.6884	0.9584	1	15	11	7
	W_fc	-0.6597	0.7204	1	15	11	7
	W_oc	-1.5550	1.3325	2	14	10	6
	W_ym	-0.9373	0.8676	1	11	7	3
LSTM2	W_gifo_x	-1.0541	1.0413	2	10	6	2
	W_gifo_r	-0.6313	0.6400	1	11	7	3
	bias	-1.5833	1.8009	2	14	10	6
	W_ic	-0.9428	0.5158	1	15	11	7
	W_fc	-0.5762	0.6202	1	15	11	7
	W_oc	-1.0619	1.4650	2	14	10	6
	W_ym	-1.0947	1.0170	2	10	6	2

[1] Only weights in LSTM layers are qunantized.
[2] In Kaldi, Wcx, Wix, Wfx, Wox are saved together as W_gifo_x, and so does W_gifo_r mean.

Table 2: Activation Function Lookup Table.

Activation	Min	Max	sampling range	sampling points
Sigmoid Input	-51.32	59.16	-64-64	2048
Tanh Input	-104.7	107.4	-128-128	2048

optimal solution to the activation functions and the intermediate results. We build lookup tables and use linear interpolation for the activation functions, such as sigmoid and tanh, and analyze the dynamic range of their inputs to decide the sampling strategies. We also investigated how many bits are enough for the intermediate results of the matrices operations.

We explore different data quantization strategies of weights with real network trained under TIMIT corpus. The sparsity of LSTM layers after pruning and fine-tune procedure is about 88.8% (i.e. a density of 11.2%). Performing the weight and activation quantization, we can achieve 12bit quantization without any accuracy loss. The data quantization strategies are shown in Table.1,2 and Table.3. For the lookup tables of activation functions sigmoid and tanh, the sampling ranges are [-64, 64], [-128, 128] respectively, the sampling points are both 2048, and the outputs are 16bit with 15bit decimals. All the results are obtained under Kaldi framework.

For TIMIT, as shown in Table.4, the PER is 20.4% for the original network, and change to 20.7% after pruning and fine-tune procedure when 32-bit floating-point numbers are used. The PER remains as 20.7% without any accuracy loss under 16/12-bit quantization, and deteriorated to 84.5% while 8-bit quantization is employed.

4. ENCODING AND COMPILING

The LSTM computation includes sparse matrices multiplication, element-wise multiplication, and memory reference.

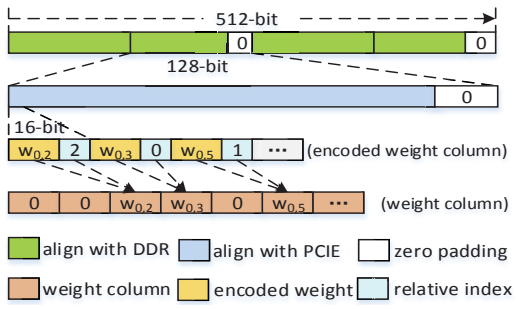


Figure 7: Encoding in CSC format and data align using zero-padding.

Table 3: Other Activation Quantization.

Activation	Min	Max	Width	Decimals
LSTM Input	-7.611	8.166	16	11
Intermediate Results	-107.8	109.4	16	8

Table 4: PER Before and After Compression.

Quantization Scheme	Phone Error Rate %
32bit floating original network	20.4%
32bit floating pruned network	20.7%
16bit fixed pruned network	20.7%
12bit fixed pruned network	**20.7%**
8bit fixed pruned network	84.5%

We design a data flow scheduler to make full use of the hardware accelerator.

Data is divided into n blocks by row where n is the number of PEs in one channels of our hardware accelerator. The first n rows are put in n different PEs. The $n+1$ row are put in the first PE again. This ensures that the first part of the matrix will be read in the first reading cycle and can be used in the next step computation immediately.

Because of the sparsity of pruned matrices. We only store the nonzero number in weight matrices to save redundant memory. We use relative row index and column pointer to help store the sparse matrix. The relative row index for each weight shows the relative position from the last nonzero weight. The column pointer indicates where the new column begins in the matrix. The accelerator will read the weight according to the column pointer.

Considering the byte-aligned bit width limitation of DDR, we use 16bit data to store the weight. The quantized weight and relative row index are put together(i.e. 12bit for quantized weight and 4bit for relative row index).

Fig.7 shows an example for the compressed sparse column (CSC) storage format and zero-padding method. We locate one column in weight matrix through a pointer, and calculate the absolute address of weights by accumulating relative indexes. In Fig.8, we demonstrate the computation pattern using a simple example. Given an input vector that has 6 elements $\{a_0,a_1,a_2,a_3,a_4,a_5\}$, and a weight matrix contains 8×6 elements. There are 2 PEs to calculate $a_3 \times w[3]$, where a_3 is the fourth element in the input vector and $w[3]$ represents the fourth column in the weight matrix.

5. HARDWARE IMPLEMENTATION

In this section, we first present challenges in hardware design and then propose the Efficient Speech Recognition Engine (ESE) accelerator system and detail how ESE accelerates the sparse LSTM.

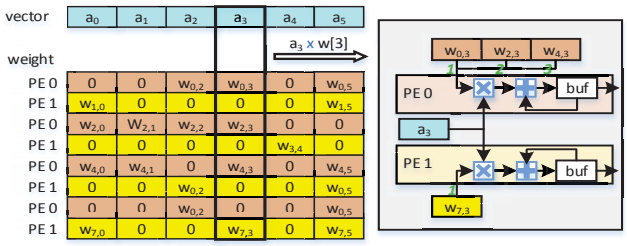


Figure 8: The computation pattern: non-zero weights in a column are assigned to 2 PEs, and every PE multiply-add their weights with the same element from the shared vector.

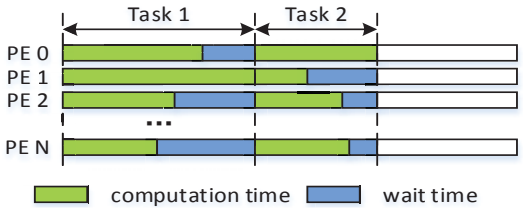


Figure 9: Imbalanced workload results in more waiting time.

5.1 Motivation

Although pruning and quantization can reduce memory footprint, some new challenges are introduced. General purpose processors cannot implement these challenges efficiently.

First, irregular computation is introduced by compression. After pruning, dense computation becomes sparse computation; After quantization, the weight and index are not byte-aligned. Instead, they must be grouped to be byte-aligned: we group the 4-bit pointer, and 12-bit weight into 2 bytes.

Second, load imbalance introduced by sparsity will reduce the hardware efficiency. In the sparse LSTM, a single element in the voice vector will be consumed by multiple PEs. As a result, operations of all PEs have to be synchronized. It will create a long waiting period if some PEs have fewer non-zero weights, as shown in Fig.9.

Moreover, general-purpose processors cannot fully exploit the parallelism in the compressed LSTM network. In the custom design, however, we have the freedom to take advantage of the parallelism both inter sparse SpMV operation and intra SpMV operation.

Many challenges exist in the specialized hardware accelerator design on FPGA. First, customized decoding circuits are needed to recover the original weight matrix. The index is relative, so accumulation is needed to recover the absolute index. We use only 4-bits to represent relative offset. If a real offset is more than 16, the largest offset that 4 bits can represent, a padding zero is introduced.

Second, data representation should be carefully designed. The data width of PCIE interface, external DDR3 memory interface, and data itself are not aligned. Moreover, the dynamic-precision quantization makes hardware computation on different data more complex and irregular. Bit shifts are necessary for different layers.

Third, a carefully designed scheduler/controller is needed. The LSTM network involves a complicated data flow and many different types of weights. Computations in the LSTM network have dependency with each other. Some computation can be executed concurrently, while other computation has to be executed sequentially. Moreover, the hardware design should support input vector sharing in the multi-channel system, which aims to perform multiple LSTM networks with different voice vectors concurrently. Therefore, a carefully designed scheduler is necessary for a highly pipelined design, which can overlap the data communication and computation.

5.2 System Overview

Fig.10 (a) shows the overview architecture of ESE system. It is a CPU+FPGA heterogeneous architecture to accelerate LSTM. The whole system can be divided into three parts: the hardware accelerator on a FPGA chip, the software program on CPU, and the external memory on the FPGA board.

Software part consists of a CPU and host memory. It communicates with FPGA via the PCI-Express bus. In the initialization procedure, it sends parameters and data structure information of the LSTM model to FPGA. It can transmit voice vectors and receive corresponding results if the hardware accelerator on FPGA is ready.

The external memory together with the FPGA chip on one development board stores all the parameters and voice vectors. The on-chip BRAM is limited while the amount of data in the LSTM model is large. The accelerator accesses the DRAM through memory controller (MEM Controller), which is built using the memory interface generator (MIG) IP.

On the FPGA chip, we put the ESE Accelerator, ESE Controller, PCIE Controller, MEM Controller, and On-chip Buffers. The ESE Accelerator consists of Processing Elements (PEs) which take charge of the majority of computation tasks in the LSTM model. PE is the basic computation unit for a slice of voice vectors with partial weight matrix. Each ESE channel implements the LSTM network for one voice vector sequence independently. On-chip buffers, including input buffer and output buffer, prepare data to be consumed by PEs and store the generated results. ESE Controller determines the behavior of other circuits on FPGA chip. It schedules PCIE/MEM Controller for data-fetch and the LSTM computation pipeline flow of ESE Accelerator. The accelerator reads parameters and voice vectors from and writes computation results to the DRAM memory. When MEM Controller is in the idle state, the accelerator can read results currently stored in the memory and feed them to the software part.

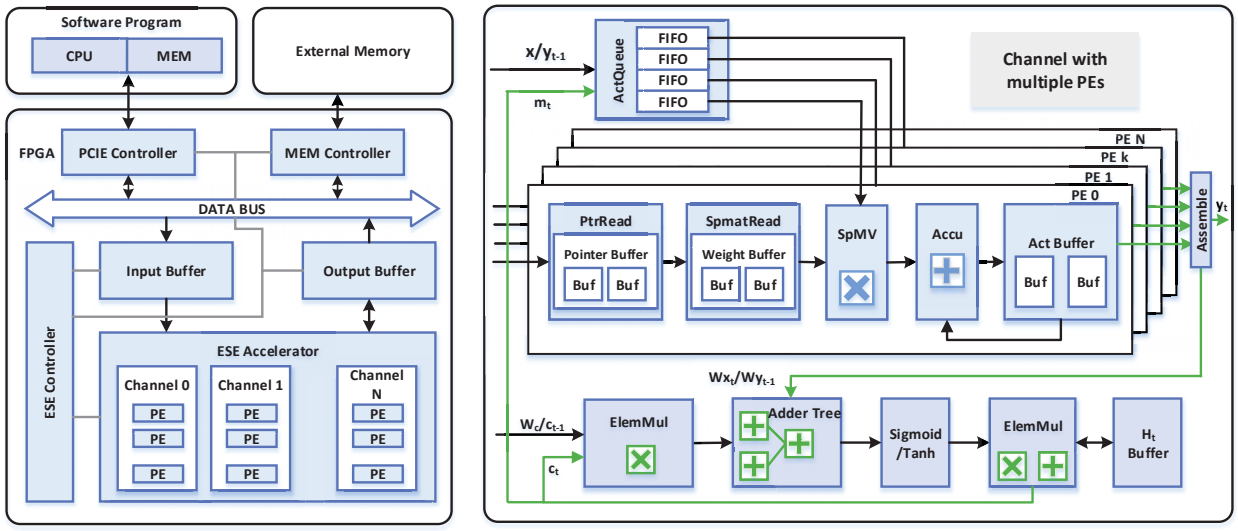
Table 5: Two types of LSTM operations: matrix-vector multiplication and element-wise multiplication.

Target	SpMV Group	ElemMul Group
i_t	$W_{ix}x_t$, $W_{ir}y_{t-1}$	$W_{ic}c_{t-1}$
f_t	$W_{fx}x_t$, $W_{fr}y_{t-1}$	$W_{fc}c_{t-1}$
c_t	$W_{cx}x_t$, $W_{cr}y_{t-1}$	f_tc_{t-1}, i_tg_t
o_t	$W_{ox}x_t$, $W_{or}y_{t-1}$	$W_{oc}c_t$
m_t	N/A	o_th_t
y_t	$W_{ym}m_t$	N/A

5.3 ESE Controller (Scheduler)

The most expensive operations are sparse matrix vector multiplication (SpMV) and element-wise multiplication (ElemMul). We partition most operations, involved in the LSTM network described by equations (1) to (6), into the such two categories, as shown in Table 5.

LSTM is a complicated dataflow. We want to meet the data dependency and ensure more parallelism at the same time. Fig.11 shows the state machine in the ESE scheduler. It overlaps computation and memory reference. From state INITIAL to STATE_6, ESE accelerator completes the computation of a LSTM. The first three lines operations are fetching weights, pointers, and vectors/diagonal matrix/bias respectively to prepare for the next computation. Operations in the fourth line are matrix-vector multiplications, and that in the fifth line are element-wise multiplications (indigo blocks) or accumulations (orange blocks). Operations in the horizontal direction have to be executed sequentially, while those in the vertical direction can be executed concurrently. For example, we can calculate $W_{fr}y_{t-1}$ and i_t con-



(a) (b)

Figure 10: The Efficient Speech Recognition Engine (ESE) accelerator system: (a) the overall ESE system architecture; (b) one channel with multiple processing elements (PEs).

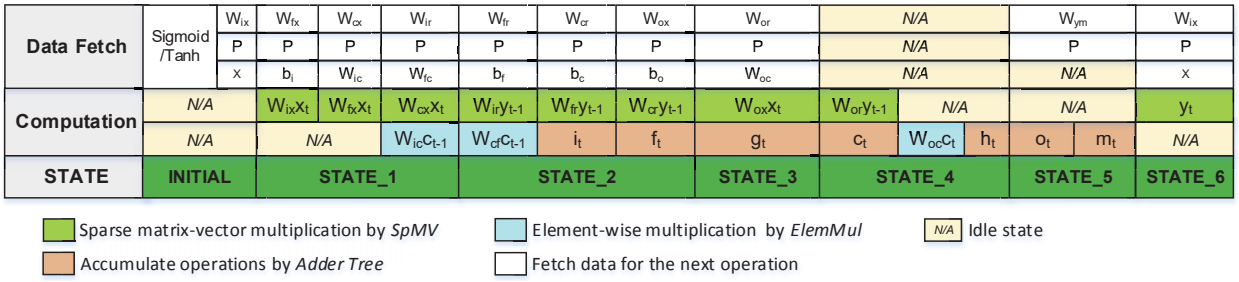


Figure 11: The state flow of ESE accelerator system: operations in the horizontal direction and vertical direction are executed sequentially and concurrently respectively.

currently, because the two operations are not dependent on each other in the LSTM network, and they can be executed by two independent computation units. $W_{ir}y_{t-1}/W_{ic}c_{t-1}$ and i_t have to be executed sequentially, because i_t is dependent on the former operations in LSTM network.

$W_{ix}x_t$ and $W_{fx}x_t$ are not dependent on each other in the LSTM network, but they cannot be calculated concurrently because they have resource conflict. Weights are stored in one external memory because even after compression the real world network cannot fit in the limited block RAM (4.25MB). Other parameters and input vector are stored in the other piece of DDR3 memory. Pointers are required for the same computations as weights, because we use pointers to look up weights in the compressed LSTM network. But pointers have small quantity and are accessed every time. Note that x, bias b, and diagonal matrix W_c are not accessed at the same time, and all these parameters have a relatively small quantity. Therefore, pointers, vectors, diagonal matrix and bias can be stored in the same external memory and can be prepared well during weight fetching period.

The latency of the element-wise operations and non-linear functions is not on the critical path. These operations are executed in parallel with the matrix-vector multiplication and weights-fetching.

5.4 ESE Channel Architecture

Fig.10 (b) shows the architecture of one ESE channel with multiple PEs. It is composed of Activation Queue (ActQueue), Sparse Matix-vector Multiplier (SpMV), Accumulator, Element-wise Multiplier (ElemMul), Adder Tree, Sigmoid/Tanh Units, and local buffers.

Activation Vector Queue (ActQueue). ActQueue consists of several FIFOs. Each FIFO stores some elements of the input voice vector a_j for each PE. ActQueue is shared by all the PEs in one channel, while each FIFO is owned by each PE independently.

ActQueue is used for decoupling the imbalanced workload among different PEs. Load imbalance arises when the number of multiply accumulation operations performed by every PE is different, due to the imbalanced sparsity. Those PEs with fewer computation tasks have to wait until the PE with the most computation tasks finishes. Thus if we have a FIFO, the fast PE can fetch a new element from the FIFO and won't need to be blocked by slow PEs. The data width of FIFO is 16-bit, and the depth is adjusted from 1 to 16 to investigate its effects on the latency, and the results are discussed in experiment section. These FIFOs are built on the distributed RAM on chip.

Sparse Matrix Read (SpmatRead). Pointer Read Unit (PtrRead) and Sparse Matrix Read (SpmatRead) manage the encoded weight matrix storage and output. The

start and end pointers p_j and p_{j+1} for column j determine the start location and length of elements in one encoded weight column that should be fetched for each element of a voice vector. SpmatRead uses pointers p_j and p_{j+1} to look up the non-zero elements in weight column j. Both PtrRead and SpmatRead consist of ping-pong buffers. Each buffer can store 512 16-bit values and is implemented with block rams. Each 16-bit data in SpmatRead buffers consists of a 4-bit index and a 12-bit weight.

Sparse Matrix-vector Multiplication (SpMV). Each element in the voice vector is multiplied by its corresponding weight column. Multiplication results in the same row of all new vectors are summed to generate an element in the result vector, which is a local reduce. In ESE, $SpMV$ multiplies an element from the input activation by a column of weight, and the current partial result is written into the partial result buffer $ActBuffer$. Accumulator $Accu$ sums the new output of $SpMV$ and previous data stored in Act Buffer. Multiplier instantiated in the design can perform 16bit$x$12bit functions.

Element-wise Multiplication (ElemMul). $ElemMul$ in Fig.10 (b) generates one vector by consuming two vectors. Each element in the output vector is the element-wise multiplication of two input vectors. There are 16 multipliers instantiated for element-wise multiplications per channel.

Adder Tree. $AdderTree$ performs summation by consuming the intermediate data produced by other units or bias data from input buffer.

Sigmoid/Tanh. They are the non-linear modules applied as activation functions to some intermediate summation results.

Here we explain how ESE computes i_t. In the initial state, PE receives weight W_{ix}, pointers P and voice vector x. Then SpMV calculates $W_{ix}X_t$ in the first phase of STATE_1. $W_{ir}y_{t-1}$ and $W_{ic}c_{t-1}$ are generated by SpMV and ElemMul respectively in the first phase of STATE_2. In the second phase of STATE_2, Adder Tree accumulates these output and bias data from the input buffer and then the following non-linear activation function unit Sigmoid/Tanh produces intermediate data i_t. PE will fetch required parameters in the previous phase to overlap with the computation. The other LSTM network operations are similar. In Fig.11, either SpMV or ElemMul is in the idle state at some phases. This is because both matrix-vector multiplication and element-wise multiplication consume weight data, while PE cannot pre-fetch enough weight data for both computations in the period of one phase.

5.5 Memory System

In the hardware design, on-chip buffers are built upon a basic idea of double-buffering, in which double buffers are operated in a ping-pong manner to overlap data transfer with computation. We use two pieces of 4GB DDR3 DRAMs as the off-chip memory, named DDR_1 and DDR_2 in Fig.12. We design a memory controller (MEM Controller). Fig.12 shows the MEM Controller architecture. On the one hand, it receives instructions from ESE Controller and schedules the data flow among ESE accelerator, PCIE interface, and DDR3 interface. On the other hand, it rearranges received data into structures required by the destination interface. We take the data flow of result y as an example. Data y at the output port of PE is 16-bit wide, while the PCIE interface is 128-bit wide. In order to increase

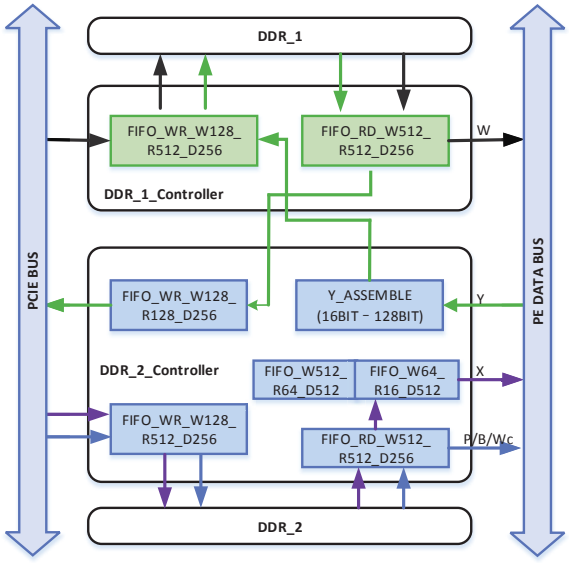


Figure 12: Memory management unit.

the data transmission speed, we assemble eight 16-bit data into one 128-bit value by Y_ASSEMBLE unit. Then it will be stored in DDR_1 temporarily and fed back to the software via PCIE interface when both PCIE and DDR_1 are in idle state. The behavior described above is shown as the green arrow line in Fig.12. Similarly, vector x is split into 32 16-bit values from a 512-bit value through asynchronous FIFOs. Moreover, asynchronous FIFOs, FIFO_WR_XX and FIFO_RD_XX, also play an important role of asynchronous clock domains isolation.

6. EXPERIMENTAL RESULTS

In this section, the performance of the hardware system is evaluated. First, we introduce the environment setup of our experiments. Then, hardware resource utilization and comprehensive experimental results are provided.

6.1 Experimental Setup

The proposed ESE hardware system is built on XCKU060 FPGA running at 200 MHz. Two external 4GB DDR3 DRAMs are used. Our host program is responsible for sending parameters and vectors into the programmable logic part, and collecting corresponding results.

We use TIMIT dataset to evaluate the performance of model compression. TIMIT is an acoustic-phonetic continuous speech corpus. It contains broadband recordings of 630 speakers of eight major dialects of American English, each reading ten phonetically rich sentences. We also use a proprietary, much larger speech recognition dataset which contains 1000 hours of training data, 100 hours of validation data and 10 hours of test data.

Our baseline software program runs on i7-5930k CPU and Pascal Titan X GPU. We use MKL BLAS / cuBLAS on CPU / GPU for dense matrix operation implementations, and MKL SPARSE / cuSPARSE on CPU / GPU for sparse matrix implementations.

6.2 Resource Utilization

Table 6 shows the resource utilization for our ESE design configured with 32 channels and each channel has 32 PEs on XCKU060 FPGA. The ESE accelerator design almost fully utilizes the FPGA's hardware resource.

Table 6: ESE Resource Utilization.

	LUT	LUTRAM[1]	FF	BRAM[1]	DSP
Avail.	331,680	146,880	663,360	1,080	2,760
Used	293,920	69,939	453,068	947	1,504
Utili.	88.6%	47.6%	68.3%	87.7%	54.5%

[1] LUTRAM is 64b each, BRAM is 36Kb each.

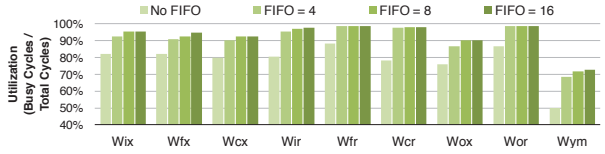


Figure 13: FIFO improves load balancing and decreases latency. The ALU utilization is more than 90% when FIFO depth is 8 for load balancing.

We configure each channel with 32 PEs, which is determined by balancing computation and data transfer. It is required that the speed of data transfer is no less than that of computation in order not to starve the DSP. As a result, we get equation 8. The expression to the left of the equal sign means that the amount of computations is divided by the computation speed. Multiplied by 2 in the numerator part means each data need multiplication and accumulation operations, and that in the denominator part indicates twice multiply-accumulate operations for 2 bytes (16-bit). ESE implements the multiply-accumulate operation in a pipeline manner. The expression to the right represents the cycles that ESE fetch the required amount of data from external memory. In our hardware implementation, both the frequencies of PE and memory interface controller are 200MHz. The width of external DRAM is 512-bit. Therefore, the proper number of PEs per channel is 32.

$$\frac{data\_size \times compress\_ratio \times 2}{PE\_num \times 2 \times freq\_PE} \geq \frac{data\_size \times compress\_ratio \times 16bit}{ddr\_width \times freq\_ddr} \quad (8)$$

FIFO Depth. ESE uses FIFO to decouple the PEs and solves load imbalance problem. Load imbalance here means the number of non-zero weight assigned to every PE is different. The FIFO for each PE reduces the waiting time for PEs with fewer computations. We adjust the cache depth to investigate its effect. The FIFO width is 16-bit, and its depth is set 1, 4, 8, 16 respectively. In Fig.13, when there's FIFO depth is one (no FIFO), the utilization, which is defined as busy cycle divided by total cycles, is low (80%) due to load imbalance. When the FIFO depth is 4, the utilization is above 90%. When FIFO depth is increased to 8 and 16, the utilization increased but has a marginal gain. Thus we chose the FIFO depth to be 8. Note that even when the FIFO depth is 8, the last matrix (Wym) still has low utilization. This is because that matrix has very few rows and each PE has few elements, and thus the FIFO cannot fully solve this problem for this matrix.

6.3 Accuracy, Speed, and Energy Efficiency

We evaluate the trade-off between accuracy and speedup of ESE in Fig.15. The speedup increases as more parameters get pruned away. The sparse model which is pruned to 10% achieved 6.2× speedup over the dense baseline model.

Table 7: Power consumption of different platforms.

Platform	CPU Dense	CPU Sparse	GPU Dense	GPU Sparse	ESE
Power	111W	38W	202W	136W	**41W**

Figure 14: Measured at the socket, the total power consumption of the machine with FPGA fully loaded is 132W. Without FPGA the idle machine consumes 91W. Subtracting the two, ESE consumes 41W.

Comparing the red and green line, we find that load-balance-aware pruning improves the speedup from 5.5× to 6.2×.

We measured power consumption of CPU, GPU and ESE. CPU power is measured by the `pcm-power` utility. GPU power is measured with `nvidia-smi` utility. We measure the power consumption of ESE by taking difference with / without the FPGA board installed. ESE takes 41 watts; CPU takes 111 watts(38 watts when using MKLSparse), GPU takes 202 watts (136 watts when using cuSparse).

The performance comparison of LSTM on ESE, CPU, and GPU is shown in Table 8. The CPU implementation used MKL BLAS and MKL SPBLAS for dense/sparse implementation, and the GPU implementation used cuBlas and cuSparse. We optimized the CPU/GPU speed by combining the four matrices of i, f, o, c gates that have no dependency into one large matrix. Both mklSparse and cuSparse implementation observed significant lower utilization of peak CPU/GPU performance for the interested matrix size (relatively small) and sparsity (around 10% non-zeros). We implement the whole LSTM on ESE. The model is pruned to 10% non-zeros. There are 11.2% non-zeros taking padding zeros into account. On ESE, the total throughput is 282 GOPS with the sparse LSTM, which corresponds to 2.52 TOPS on the dense LSTM. Processing the LSTM with 1024 hidden elements, ESE takes 82.7 us, CPU takes 6017.3/3569.9 us (dense/sparse), and GPU takes 240.2/287.4 us (dense/sparse). With batch=32, CPU sparse is faster than dense because CPU is good at serial processing, while GPU sparse is slower than dense because GPU is throughput oriented. With no batching, we observed both CPU and GPU are faster for the sparse LSTM because the saving of memory bandwidth is more salient.

Performance wise, ESE is 43× faster than CPU 3× faster than GPU. Considering both performance and power consumption, ESE is 197.0×/40.0× (dense/sparse) more energy efficient than CPU, and 14.3×/11.5× (dense/sparse) more energy efficient than GPU. Sparse LSTM makes both CPU and GPU more energy efficient as well, which shows the advantage of our pruning technique.

Table 8: Performance comparison of running LSTM on ESE, CPU and GPU

Plat.	ESE on FPGA (ours)									CPU		GPU	
Matrix	Matrix Size	Sparsity (%)[1]	Compres. Matrix (Bytes)[2]	Theoreti. Comput. Time (µs)	Real Comput. Time (µs)	Total Operat. (GOP)	Real Perform. (GOP/s)	Equ. Operat. (GOP)	Equ. Perform. (GOP/s)	Real Comput. Time (µs)		Real Comput. Time (µs)	
										Dense	Sparse	Dense	Sparse
W_{ix}	1024×153	11.7	36608	2.9	**5.36**	0.0012	218.6	0.010	1870.7	**1518.4**[3]	**670.4**	**34.2**	**58.0**
W_{fx}	1024×153	11.7	36544	2.9	**5.36**	0.0012	218.2	0.010	1870.7				
W_{cx}	1024×153	11.8	37120	2.9	**5.36**	0.0012	221.6	0.010	1870.7				
W_{ox}	1024×153	11.5	35968	2.8	**5.36**	0.0012	214.7	0.010	1870.7				
W_{ir}	1024×512	11.3	118720	9.3	**10.31**	0.0038	368.5	0.034	3254.6	**3225.0**[4]	**2288.0**	**81.3**	**166.0**
W_{fr}	1024×512	11.5	120832	9.4	**10.01**	0.0039	386.3	0.034	3352.1				
W_{cr}	1024×512	11.2	117760	9.2	**9.89**	0.0038	381.2	0.034	3394.5				
W_{or}	1024×512	11.5	120256	9.4	**10.04**	0.0038	383.5	0.034	3343.7				
W_{ym}	512×1024	10.0	104832	8.2	**15.66**	0.0034	214.2	0.034	2142.7	1273.9	611.5	124.8	63.4
Total	**3248128**	**11.2**	**728640**	**57.0**	**82.7**	**0.0233**	**282.2**	**0.208**	**2515.7**	**6017.3**	**3569.9**	**240.3**	**287.4**

[1] Pruned with 10% sparsity, but padding zeros incurred about 1% more non-zero weights.
[2] Sparse matrix index is included, and weight takes 12 bits, index takes 4 bits => 2 Bytes per weight in total.
[3] Concatenating W_{ix}, W_{fx}, W_{cx} and W_{ox} into one large matrix $W_{\text{ifoc\_x}}$, whose size is 4096×153.
[4] Concatenating W_{ir}, W_{fr}, W_{cr} and W_{or} as one large matrix $W_{\text{ifoc\_r}}$, whose size is 4096×512. These matrices don't have dependency and combining matrices can achieve 2× speedup on GPU due to better utilization.

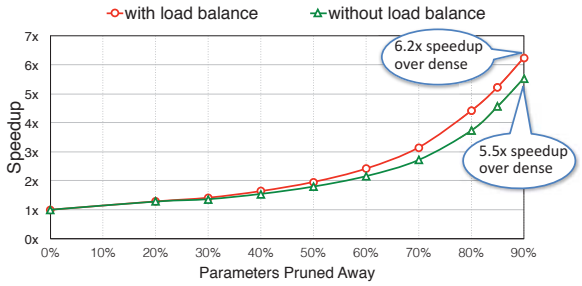


Figure 15: Computation latency decreases as the sparsity increases. Running the sparse model is 4.2× faster over the dense model, both run on ESE. Load balance aware pruning helps speedup.

7. RELATED WORK

Deep Compression Deep Compression [11] is a method that can compress convolutional neural network models by 35x-59x without hurting the accuracy. It comprises of pruning, weight sharing and Huffman coding. However, the compression rate is targeting CNN and image recognition. In this work we target LSTM and speech recognition. The method also differs from the previously proposed 'Deep Compression' in that we catered specially for FPGA design. During pruning, we enforce each row have the same amount of weight to enforce hardware load balancing. During quantization, we use linear quantization instead of non-linear quantization, which made it possible to directly use the integer ALU. We also eliminate the Huffman Coding step which introduced extra decoding overhead but marginal gain.

CNN Accelerators Many custom accelerators have been proposed for CNNs. DianNao [2] implements an array of multiply-add units to map large DNN onto its core architecture. Due to limited SRAM resource, the off-chip DRAM traffic dominates the energy consumption. DaDianNao [3] and ShiDianNao [5] eliminate the DRAM access by having all weights on-chip (eDRAM or SRAM). However, these DianNao-series architectures are CNN proposed to accelerate CNN, and the weights are uncompressed and stored in the dense format. In this work, we target LSTM neural network and speech recognition, and data compression is also supported in our ESE architecture. Our work in this paper is also distinguished itself with Angel-Eye architecture, which also has the compression, compilation and acceleration, but it is for CNN and image recognition tasks [9, 18].

EIE Accelerator The EIE architecture proposed by Han et al. [10] can performs inference on compressed network model and accelerates the resulting sparse matrix-vector multiplication with weight sharing. With only 600mW power consumption, EIE can achieve 102 GOPS processing power on a compressed network corresponding to 3 TOPS/s on an uncompressed network, which is 24000× and 3400× more energy efficient than a CPU and GPU respectively. But EIE is also not designed for LSTM and speech recognition, ESE in this paper is targeted for LSTM and ESE has many considerations for FPGA while EIE is for ASIC, which leads different design optimization. Besides, EIE use codebook-based quantization, but ESE use direct quantization.

Sparse Matrix-Vector Multiplication Accelerators To pursue a better computational efficiency on machine learning and deep learning, several recent works focus on using FPGA as an accelerator for Sparse Matrix-Vector Multiplication (SpMV). Zhuo et al. [21] proposed an FPGA-based design on Virtex-II Pro for SpMV. Their design outperforms general-purpose processors, but the performance is limited by memory bandwidth. Fowers et al. [7] proposed a novel sparse matrix encoding and an FPGA-optimized architecture for SPMV. With lower bandwidth, it achieves 2.6× and 2.3× higher power efficiency over CPU and GPU respectively while having lower performance due to lower memory bandwidth. Dorrance et al. [4] proposed a scalable SMVM kernel on Virtex-5 FPGA. It outperforms CPU and GPU counterparts with >300× computational efficiency and has 38-50× improvement in energy efficiency. For compressed deep networks, previously proposed SpMV accelerators can only exploit the static weight sparsity. In this paper, we use the relative indexed compressed sparse column (CSC) format for data storing, and we develop a scheduler which can map a complicate LSTM network on ESE accelerator.

GRU on FPGA Nurvitadhi et al presented a hardware accelerator for Gated Recurrent Network (GRU) on Stratix V and Arria 10 FPGAs [16]. This work shows that FPGA can provide superior performance/Watt over CPU and GPU. In our work, we present a FPGA accelerator for LSTM network. It also demonstrates a higher efficiency FPGA comparing with CPU and GPU. Different from theirs, our ESE is especially designed for sparse LSTM model. It can achieve more benefits but also introduces a more difficult hardware design.

LSTM on FPGA In order to explore the parallelism for RNN/LSTM, Chang presented a hardware implementation of LSTM network on Zynq 7020 FPGA from Xilinx with

2 layers and 128 hidden units in hardware [1]. The implementation is 21 times faster than the ARM Cortex-A9 CPU embedded on the Zynq 7020 FPGA. Lee accelerated RNNs using massively parallel processing elements (PEs) for low latency and high throughput on FPGA [15]. These implementations did not support sparse LSTM network, while our ESE can achieve more speed up by supporting sparse LSTM.

8. CONCLUSION

In this paper, we present Efficient Speech Recognition Engine (ESE) that works directly on compressed sparse LSTM model. ESE is optimized across the algorithm-software-hardware boundary: we first propose a method to compress the LSTM model by 20× without sacrificing the prediction accuracy, which greatly saves the memory bandwidth of FPGA implementation. Then we design a scheduler that can map the complex LSTM operations on FPGA and achieve parallelism. Finally we propose a hardware architecture that efficiently deals with the irregularity caused by compression. Working directly on the compressed model enables ESE to achieve 282 GOPS (equivalent to 2.52 TOPS for dense LSTM) on Xilinx XCKU060 FPGA board. ESE outperforms Core i7 CPU and Pascal Titan X GPU by factors of 43× and 3× on speed, and it is 40× and 11.5× more energy efficient than the CPU and GPU respectively.

9. ACKNOWLEDGMENT

This work was supported by National Natural Science Foundation of China (No.61373026, 61622403, 61261160501).

We would like to thank Wei Chen, Zhongliang Liu, Guanzhe Huang, Yong Liu, Yanfeng Wang, Xiaochuan Wang and other researchers from Sogou for their suggestions and providing real-world speech data for model compression performance test.

10. REFERENCES

[1] A. X. M. Chang, B. Martini, and E. Culurciello. Recurrent neural networks hardware implementation on FPGA. *CoRR*, abs/1511.05552, 2015.

[2] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam. Diannao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *ASPLOS*, 2014.

[3] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. Dadiannao: A machine-learning supercomputer. In *MICRO*, December 2014.

[4] R. Dorrance, F. Ren, et al. A scalable sparse matrix-vector multiplication kernel for energy-efficient sparse-blas on FPGAs. In *FPGA*, 2014.

[5] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam. Shidiannao: shifting vision processing closer to the sensor. In *ISCA*, pages 92–104. ACM, 2015.

[6] D. A. et al. Deep speech 2: End-to-end speech recognition in english and mandarin. *arXiv, preprint arXiv:1512.02595*, 2015.

[7] J. Fowers, K. Ovtcharov, K. Strauss, et al. A high memory bandwidth fpga accelerator for sparse matrixvector multiplication. In *FCCM*, 2014.

[8] J. S. Garofolo, L. F. Lamel, W. M. Fisher, J. G. Fiscus, and D. S. Pallett. Darpa timit acoustic-phonetic continous speech corpus cd-rom. nist speech disc 1-1.1. *NASA STI/Recon technical report n*, 93, 1993.

[9] K. Guo, L. Sui, et al. Angel-eye: A complete design flow for mapping cnn onto customized hardware. In *ISVLSI*, 2016.

[10] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. Eie: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*, 2016.

[11] S. Han, H. Mao, and W. J. Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*, 2016.

[12] S. Han, J. Pool, J. Tran, and W. J. Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of Advances in Neural Information Processing Systems*, 2015.

[13] A. Hannun, C. Case, J. Casper, B. Catanzaro, G. Diamos, E. Elsen, R. Prenger, S. Satheesh, S. Sengupta, A. Coates, and A. Ng. Deep speech: Scaling up end-to-end speech recognition. *arXiv, preprint arXiv:1412.5567*, 2014.

[14] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 1997.

[15] M. Lee, K. Hwang, J. Park, S. Choi, S. Shin, and W. Sung. Fpga-based low-power speech recognition with recurrent neural networks. *arXiv preprint arXiv:1610.00552*, 2016.

[16] E. Nurvitadhi, J. Sim, D. Sheffield, A. Mishra, S. Krishnan, and D. Marr. Accelerating recurrent neural networks in analytics servers: Comparison of fpga, cpu, gpu, and asic. In *Field Programmable Logic and Applications (FPL), 2016 26th International Conference on*, pages 1–4. EPFL, 2016.

[17] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The Kaldi speech recognition toolkit. In *IEEE 2011 workshop on automatic speech recognition and understanding*, 2011.

[18] J. Qiu, J. Wang, et al. Going deeper with embedded FPGA platform for convolutional neural network. In *FPGA*, 2016.

[19] H. Sak et al. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, pages 338–342, 2014.

[20] L. D. Xuedong Huang. *An Overview of Modern Speech Recognition*, pages 339–366. Chapman & Hall/CRC, January 2010.

[21] L. Zhuo and V. K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *FPGA*, 2005.