arXiv:1612.00694v1 [cs.CL] 1 Dec 2016

ESE: Efficient Speech Recognition Engine with Compressed 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 larger and larger models. Such large model is both computation intensive and memory intensive. Deploying such bulky model results in high power consumption given latency constraint and leads to high total cost of ownership (TCO) of a data center. In order 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 scheduler that encodes and partitions the compressed model to each PE 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 compressed model. Implemented on Xilinx XCKU060 FPGA running at 200MHz, ESE has a performance of 282 GOPS working directly on the compressed LSTM network, corresponding to 2.52 TOPS on the uncompressed 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.

1 Introduction

Deep neural network has surpassed the traditional acoustic model and become the state-of-the-art method for speech recognition [1, 2]. Long Short-Term Memory (LSTM) [3], Gated Recurrent Unit (GRU) [4] and vanilla recurrent neural networks (RNNs) are popular in speech recognition. In this work, we designed a hardware accelerator called ESE for the most complex one: the LSTM.

ESE takes the approach of EIE [5] one step further to address a more general problem of accelerating not only feed forward neural networks but also recurrent neural networks and LSTM. The recurrent nature of RNN produces complicated data dependency, which is more challenging than feed forward neural nets. To deal with this problem, we designed a data flow that can effectively schedule the complex RNN operations using multiple EIE cores.

Among all factors contribute to the monthly bill of a data center, power consumption is the major one. Since memory reference consumes more than two orders of magnitude higher energy than ALU operations, we focus on reducing the memory footprint.

In order to achieve this, we design a novel method to optimize across the algorithm, software and hardware. At algorithm level, ESE revisited pruning algorithm from the hardware efficiency

1st International Workshop on Efficient Methods for Deep Neural Networks at NIPS 2016, Barcelona, Spain.
Full paper to appear at FPGA 2017.

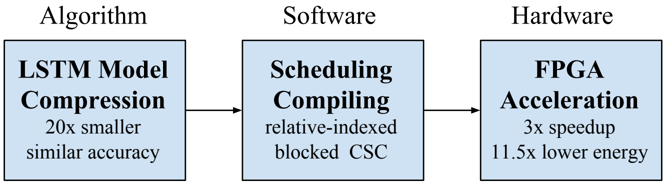
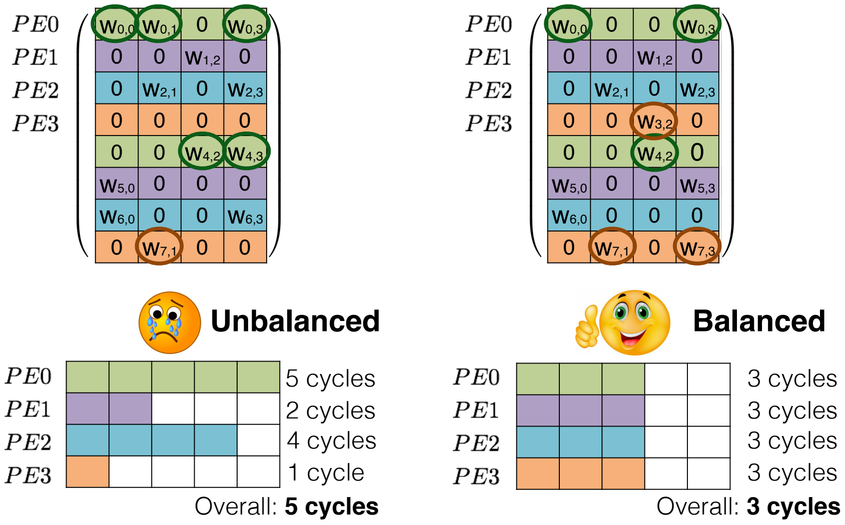


Figure 1: ESE optimizes LSTM computation across algorithm, software and hardware stack. ESE is 3× faster and 11.5× more energy efficient than Pascal Titan X GPU.

perspective, by introducing load-balance-aware pruning. Next we design a scheduler that can effectively schedule the compressed LSTM model using spMV as basic building block, with memory reference fully overlapped with computation. At hardware level, we design a new architecture that can works directly on the compressed model that could be efficiently mapped to FPGA. ESE achieves high efficiency by load balancing and partitioning both the computation and storage.

2 Model Compression by Load-Balance-Aware Pruning



$$\begin{array}{c} PE0 \\ PE1 \\ PE2 \\ PE3 \\ \\ \\ \\ \\ \end{array} \begin{pmatrix} w_{0,0} & w_{0,1} & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & 0 & 0 \\ 0 & 0 & w_{4,2} & w_{4,3} \\ w_{5,0} & 0 & 0 & 0 \\ w_{6,0} & 0 & 0 & w_{6,3} \\ 0 & w_{7,1} & 0 & 0 \end{pmatrix} \qquad \begin{array}{c} PE0 \\ PE1 \\ PE2 \\ PE3 \\ \\ \\ \\ \\ \end{array} \begin{pmatrix} w_{0,0} & 0 & 0 & w_{0,3} \\ 0 & 0 & w_{1,2} & 0 \\ 0 & w_{2,1} & 0 & w_{2,3} \\ 0 & 0 & w_{3,2} & 0 \\ 0 & 0 & w_{4,2} & 0 \\ w_{5,0} & 0 & 0 & w_{5,3} \\ w_{6,0} & 0 & 0 & 0 \\ 0 & w_{7,1} & 0 & w_{7,3} \end{pmatrix}$$

Unbalanced

$PE0$	5 cycles
$PE1$	2 cycles
$PE2$	4 cycles
$PE3$	1 cycle

Overall: **5 cycles**

Balanced

$PE0$	3 cycles
$PE1$	3 cycles
$PE2$	3 cycles
$PE3$	3 cycles

Overall: **3 cycles**

Figure 2: Load Balance Aware Pruning and its Benefit for Parallel Processing

Previous pruning methods removed the redundant connections based on the absolute value of the weights [6, 7], which lead to a potential problem of unbalanced non-zero weights distribution. In our hardware implementation, matrix is divided into different sub-matrices and assigned to the corresponding processing elements (PEs), so that the multiplication could be executed in parallel. However, only non-zero weights are stored and computed, thus PEs with less non-zero weights have to wait for those with more non-zero weights. The workload imbalance over PEs would result in a gap between the real performance and peak performance.

To solve this problem, we propose the **load-balance-aware pruning**, which produces the same compression rate among all the sub-matrices. In this way, the workload of each PE is roughly the same, and no waiting is needed any more. As shown in Fig. 2, the matrix is divided into four colors, and each color belongs to a PE for parallel processing. With conventional pruning, PE0 might have five non-zero weights while PE3 may have only three. The total processing time is the longest, which is 5 cycles. With load-balance-aware pruning, all PEs have three non-zero weights, thus only 3 cycles are needed to carry out the operation. Both cases have the same non-zero weights in total, but load-balance-aware pruning need fewer cycles. The difference of prediction accuracy with and without load-balance-aware pruning is very small, as shown in Fig. 3. There are some noise around 70%, and we put lots of experiments around 90%, which is the sweet point, and find the performance is very similar. We highlight this practical pruning strategy for hardware efficiency.

We evaluate a LSTM on the TIMIT dataset. As shown in Fig. 3, the sweet point sparsity is around 90%. On the sparsity point of 92.6%, the load-balance model achieves a Phone Error Rate (PER) of

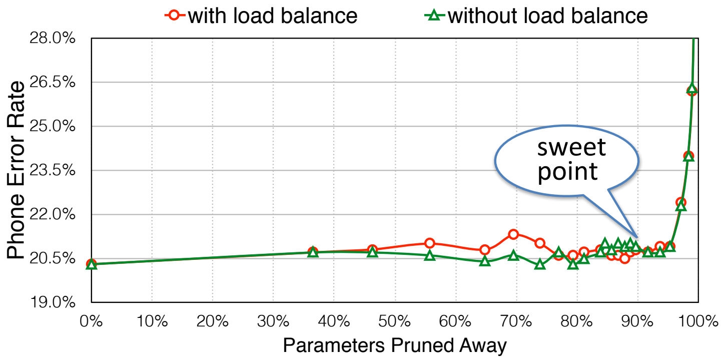
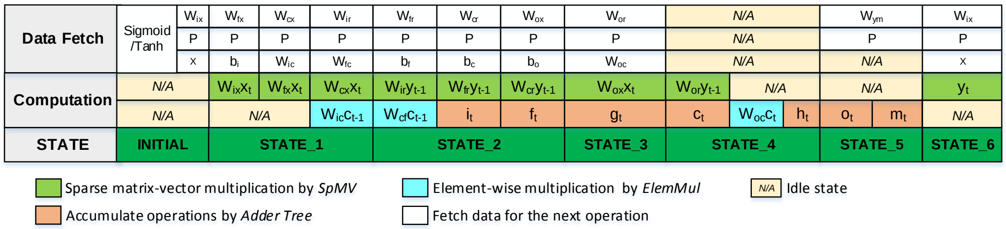


Figure 3: Phone Error Rate Before and After Compression (Pruning + Quantization).



Data Fetch	Sigmoid /Tanh	W_{ix}	W_{fx}	W_{cx}	W_{ir}	W_{fr}	W_{cr}	W_{ox}	W_{or}	*N/A*			W_{ym}		W_{ix}
		P	P	P	P	P	P	P	P	*N/A*			P		P
		x	b_i	W_{ic}	W_{fc}	b_f	b_c	b_o	W_{oc}	*N/A*			*N/A*		x
Computation	*N/A*		$W_{ix}x_t$	$W_{fx}x_t$	$W_{cx}x_t$	$W_{ir}y_{t-1}$	$W_{fr}y_{t-1}$	$W_{cr}y_{t-1}$	$W_{ox}x_t$	$W_{or}y_{t-1}$	*N/A*		*N/A*		y_t
	N/A		*N/A*		$W_{ic}c_{t-1}$	$W_{cf}c_{t-1}$	i_t	f_t	g_t	c_t	$W_{oc}c_t$	h_t	o_t	m_t	*N/A*
STATE	INITIAL		STATE_1			STATE_2			STATE_3	STATE_4			STATE_5		STATE_6

🟩 Sparse matrix-vector multiplication by *SpMV*
🟦 Element-wise multiplication by *ElemMul*
N/A Idle state
🟧 Accumulate operations by *Adder Tree*
□ Fetch data for the next operation

Figure 4: The data flow of ESE accelerator system: operations in the horizontal direction and vertical direction are executed sequentially and concurrently respectively.

20.7%, which is only 0.4% higher than the original model. 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 92% of the parameters without hurting Word Error Rate (WER), which aligns with our result on TIMIT dataset. In our later discussions we will use a conservative density of 10%(90% sparse).

We compress the model by quantizing 32-bit floating point weights into 12-bit integer. Plus the 4bits for sparse index, each weight is still aligned at 16bit. Activations are quantized to 16-bit integer. We use linear quantization strategy on both the weights and activations.

3 Computation Scheduling

Compressed LSTM model is highly irregular, and thus accelerators on dense LSTMs cannot effectively take advantage of sparsity [8, 9]. LSTM is a complicated dataflow, we want to have more parallelism and meet the data dependency at the same time, but previous spMV accelerator [10, 11, 12] or sparse DNN accelerator [5] cannot achieve such scheduling.

We propose an ESE scheduler, shown in Fig.4, in which computation and data-fetching are fully overlapped. Operations in the first three lines fetch the pointers, weights and biases from memory to prepare for computation. Operations in the fourth line are matrix-vector multiplications. And operations 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 are able be executed concurrently. For example, we can calculate $W_{fr}y_{t-1}$ and i_t concurrently, because the two operations are not dependent on each other in the LSTM network, and they can be executed by two independent computation units in hardware system. $W_{ir}y_{t-1}/W_{ic}c_{t-1}$ and i_t have to be executed sequentially, because i_t is dependent on the former operations in LSTM network.

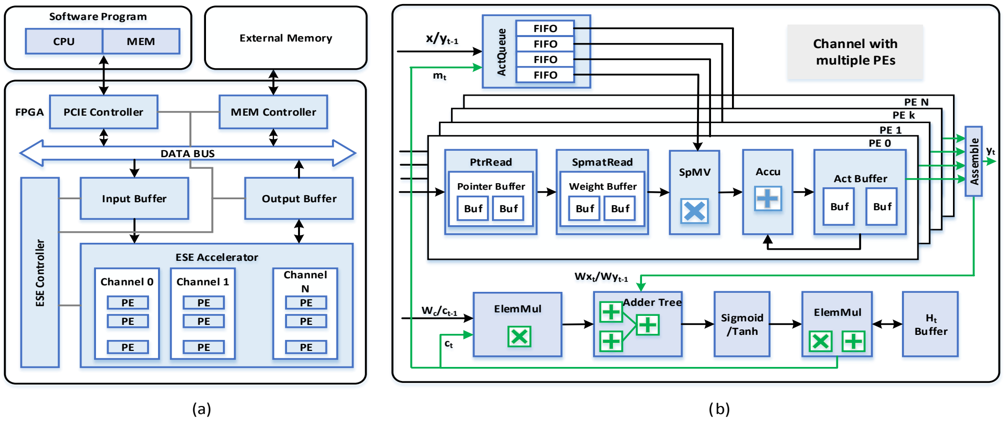


Figure 5: The Efficient Speech Recognition Engine accelerator system: (a) the overall ESE system architecture; (b) one channel with multiple processing elements (PEs).

4 Hardware Architecture

Fig.5(a) shows the overview architecture of ESE system. It is a CPU+FPGA heterogeneous architecture to accelerate LSTM networks. Fig.5(b) shows the architecture of one channel with multiple PEs. It is composed of several major components:

Activation Vector Queue (ActQueue). ActQueue consists of several FIFOs. Each FIFO stores 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's fuction is to decouple the load imbalance across different PEs.

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.

Sparse Matrix-vector Multiplication (SpMV). SpMV unit multiplies the activation by a column on weight, and the current partial result is written into partial result ActBuffer. Accumulator sums the new output of SpMV and previous data stored in Act Buffer. Multiplier instantiated in the design can perform 16bit×12bit functions.

Element-wise Multiplication (ElemMul). ElemMul in Fig.5(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 in each channel.

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

Sigmoid/Tanh. These non-linear modules are implemented with quantized look-up table.

Constrained by FPGA resource, ESE cannot buffer all the weights on-chip, thus hiding latency is important. ESE adopts double buffering to overlap the time of data transfer and computation.

5 Experimental Results

We implemented ESE on Xilinx XCKU060 FPGA. ESE is clocked at 200MHz, and there is a large room for improvement. We evaluate the speedup and energy efficiency of ESE and compared with CPU and GPU. Our baseline LSTM runs on Intel Core i7-5930K CPU and Pascal Titan X GPU. We use MKL BLAS/cuBLAS for dense LSTM, and MKL SPARSE/cuSPARSE for sparse LSTM.

The experimental results of LSTM on ESE, CPU, and GPU are shown in Table 1. The model is pruned to 10% non-zeros. There are fewer than 12.2% non-zeros even 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. On CPU and GPU, we combine some matrices together to improve the performance. Processing the LSTM with 1024 hidden elements, ESE takes 82.7 us, CPU takes

Table 1: 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	18304	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	18272	2.9	**5.36**	0.0012	218.2	0.010	1870.7				
W_{cx}	1024×153	11.8	18560	2.9	**5.36**	0.0012	221.6	0.010	1870.7				
W_{ox}	1024×153	11.5	17984	2.8	**5.36**	0.0012	214.7	0.010	1870.7				
W_{ir}	1024×512	11.3	59360	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	60416	9.4	**10.01**	0.0039	386.3	0.034	3352.1				
W_{cr}	1024×512	11.2	58880	9.2	**9.89**	0.0038	381.2	0.034	3394.5				
W_{or}	1024×512	11.5	60128	9.4	**10.04**	0.0038	383.5	0.034	3343.7				
W_{ym}	512×1024	10.0	52416	8.2	**15.66**	0.0034	214.2	0.034	2142.7	1273.9	611.5	124.8	63.4
Total	**3248128**	**11.2**	**364320**	**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.

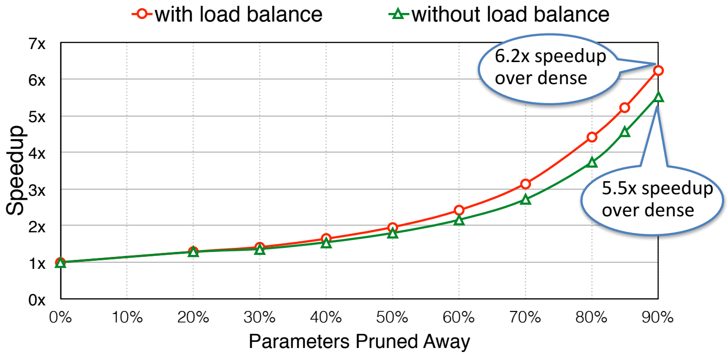


Figure 6: Sparse LSTM model running on ESE can be 6.2× faster than the dense model.

6017.3/3569.9 us (dense/sparse), and GPU takes 240.2/287.4 us (dense/sparse). ESE is 43× faster than CPU 3× faster than GPU.

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). 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. Though sparse LSTM makes GPU more energy efficient, it is still one magnitude lower than ESE.

We analyze the trade off between sparsity and speedup in Fig. 6. The speedup increases as more parameters get pruned away. The sparse model pruned to 10% achieved 6.2× speedup over the baseline dense model on ESE. Besides, load-balance-aware pruning makes the performance 11% higher, as shown in the red and green line,

6 Conclusion

We present Efficient Speech Recognition Engine (ESE) that works directly on compressed LSTM model. ESE is optimized across the algorithm-software-hardware boundary: we first propose a method to compress the LSTM model by 12× 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.

Acknowledgment

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.

References

[1] Awni Hannun, Carl Case, Jared Casper, Bryan Catanzaro, Greg Diamos, Erich Elsen, Ryan Prenger, Sanjeev Satheesh, Shubho Sengupta, Adam Coates, and Andrew Ng. Deep speech: Scaling up end-to-end speech recognition. *arXiv, preprint arXiv:1412.5567*, 2014.

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

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

[4] Kyunghyun Cho, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. On the properties of neural machine translation: Encoder-decoder approaches. *arXiv preprint arXiv:1409.1259*, 2014.

[5] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: efficient inference engine on compressed deep neural network. *arXiv preprint arXiv:1602.01528*, 2016.

[6] Song Han, Jeff Pool, John Tran, and William J Dally. Learning both weights and connections for efficient neural networks. In *Proceedings of Advances in Neural Information Processing Systems*, 2015.

[7] Song Han, Huizi Mao, and William J Dally. Deep Compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *ICLR*, 2016.

[8] Eriko Nurvitadhi, Jaewoong Sim, David Sheffield, Asit Mishra, Srivatsan Krishnan, and Debbie Marr. Accelerating recurrent neural networks in analytics server: Comparison of fpga, cpu, gpu, and asic. In *Field Programmable Logic (FPL), 2016 International Conference on*. IEEE, 2016.

[9] Andre Xian Ming Chang, Berin Martini, and Eugenio Culurciello. Recurrent neural networks hardware implementation on FPGA. *CoRR*, abs/1511.05552, 2015.

[10] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on fpgas. In *FPGA*, 2005.

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

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