Learning to Skip Ineffectual Recurrent Computations in LSTMs

Arash Ardakani, Zhengyun Ji, and Warren J. Gross
McGill University, Montreal, Canada

***Abstract*—Long Short-Term Memory (LSTM) is a special class of recurrent neural network, which has shown remarkable successes in processing sequential data. The typical architecture of an LSTM involves a set of states and gates: the states retain information over arbitrary time intervals and the gates regulate the flow of information. Due to the recursive nature of LSTMs, they are computationally intensive to deploy on edge devices with limited hardware resources. To reduce the computational complexity of LSTMs, we first introduce a method that learns to retain only the important information in the states by pruning redundant information. We then show that our method can prune over 90% of information in the states without incurring any accuracy degradation over a set of temporal tasks. This observation suggests that a large fraction of the recurrent computations are ineffectual and can be avoided to speed up the process during the inference as they involve noncontributory multiplications/accumulations with zero-valued states. Finally, we introduce a custom hardware accelerator that can perform the recurrent computations using both sparse and dense states. Experimental measurements show that performing the computations using the sparse states speeds up the process and improves energy efficiency by up to 5.2× when compared to implementation results of the accelerator performing the computations using dense states.**

I. Introduction

Convolutional neural networks (CNNs) have surpassed human-level accuracy in different complex tasks that require learning hierarchical representation of spatial data [1]. CNN is a stack of multiple convolutional layers followed by a few fully-connected layers [2]. The computational complexity of CNNs is dominated by the multiply-accumulate operations of convolutional layers as they perform high dimensional convolutions while the majority of weights are usually found in fully-connected layers.

Recurrent neural networks (RNNs) have also shown remarkable success in processing variable-length sequences. As a result, they have been adopted in many applications performing temporal tasks such as language modeling [3], neural machine translation [4], automatic speech recognition [5], and image captioning [6]. Despite the high prediction accuracy of RNNs over short-length sequences, they fail to learn long-term dependencies due to the exploding gradient problem [7]. Long Short-Term Memories (LSTM) [8] is a popular class of RNNs, that was introduced in literature to mitigate the above issue. Similar to CNNs, LSTMs also suffer from high computational complexity due to the recursive nature of LSTMs.

The recurrent transitions in LSTM are performed as

$$\begin{pmatrix} f_t \\ i_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W_h h_{t-1} + W_x x_t + b, \tag{1}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot g_t, \tag{2}$$

$$h_t = o_t \odot \tanh(c_t), \tag{3}$$

where $W_x \in \mathbb{R}^{d_x \times 4d_h}$, $W_h \in \mathbb{R}^{d_h \times 4d_h}$ and $b \in \mathbb{R}^{4d_h}$ denote the recurrent weights and bias. The hidden states $h \in \mathbb{R}^{d_h}$ and $c \in \mathbb{R}^{d_h}$ retain the temporal state of the network. The gates f_t, i_t, o_t and g_t regulate the update of LSTM parameters. The logistic sigmoid function and element-wise product are denoted as σ and $\odot$, respectively. Eq. (1) involves the multiplication between the weight matrices (i.e., W_h and W_x) and the vectors x_t and h_{t-1} while Eq. (2) and Eq. (3) perform only element-wise multiplications. Therefore, the recurrent computations of LSTM are dominated by the vector-matrix multiplications (i.e., $W_h h_{t-1} + W_x x_t$). It is worth mentioning that LSTM architectures are usually built on high dimensional input (i.e., x_t) or state (i.e., h_{t-1}), increasing the number of operations performed in Eq. (1). Moreover, the recurrent computations of LSTM are executed sequentially, as the input at time t depends on the output at time $t-1$. Therefore, the high computational complexity of LSTMs makes them difficult to deploy on portable devices requiring real-time processes at the edge.

To reduce the computational complexity of CNNs, recent convolutional accelerators have exploited pruning [9], binarization [10], and intrinsic sparsity among activations [11], improving both processing time and energy efficiency. Exploiting pruning to speed up the recurrent computations of LSTMs has been also studied in [12]. However, no practical accelerator for recurrent computations was introduced in literature since the binarization algorithms developed for LSTMs are limited to certain temporal tasks such as word language modelling, and do not generalize well over different temporal tasks [13]. The convolutional accelerator introduced in [11] exploits the intrinsic sparsity among activations incurred by the rectified linear unit (ReLU) in order to avoid ineffectual multiplications/accumulations with zero-valued activations. Despite its remarkable performance, this technique has not been exploited in architectures accelerating recurrent computations yet, since LSTMs use sigmoid and tanh functions as their non-linear functions.

Motivated by the above statements, this paper aims to first introduce a method that learns only the important information stored in to the state h_{t-1} as this state contributes the most in the recurrent computations (see Eq. (1)). Note that the input vector x_t is usually one-hot encoded except for certain tasks with high dimensional vocabulary sizes such as word language modelling. In this case, the embedding layer is used for these tasks to reduce the input dimension [3]. In either cases,

978-3-9819263-2-3/DATE19/©2019 EDAA

d_h is usually greater than d_x. After introducing the learning method, we show that a large fraction of information coming from previous time steps are unimportant and can be dropped. More precisely, the proposed learning algorithm can prune over 90% of values in the state vector h_{t-1} over different temporal tasks such as character language modeling, word language modeling and sequential image classification without incurring any performance degradation. We then introduce a custom hardware accelerator that can perform the recurrent computations on both sparse and dense representations of the state vector h_{t-1}. Finally, we show that performing the computations on the sparse state vector results in up to 5.2× speedup and energy improvement over the dense representation when both running on the proposed accelerator. To the best of our knowledge, this work is the first attempt to skip the ineffectual recurrent computations incurred by pruning the hidden state h_{t-1}.

II. Learning Ineffectual Recurrent Computations

During the past few years, it has been widely observed that deep neural networks contain numerous redundant operations that can be removed without incurring any accuracy degradation [12]. In [12], [9], it was shown that exploiting pruned weights can speed up the computations of LSTMs and CNNs by skipping the multiplications/accumulations with zero-valued weights. In [11], the intrinsic sparsity among the activations in CNNs was exploited to speed up the convolutional computations. In fact, CNNs commonly use the ReLU as their non-linear function, which dynamically clamps all negatively-valued activations to zero. Unlike CNNs, LSTMs use sigmoid and tanh units as their non-linear function. As a result, in order to use the latest technique to speed up the recurrent computations, we first need a training method to learn ineffectual computations among the activations (i.e., the state h_{t-1}) in LSTMs.

A. Pruning Method

The total number of operations required to perform Eq. (1) is equal to $2 \times (d_x \times 4d_h + d_h \times 4d_h) + 4d_h$ when considering each multiply-accumulate as two operations. In temporal tasks such as character-level language modeling, where the input vector x_t is one-hot encoded, its size (i.e., d_x) is limited to a few hundreds. In this case, the vector-matrix multiplication of $W_x x_t$ is implemented as a look-up table and its number of operations is equal to $4d_h$, similar to the number of operations required for biases. Eq. (2) and Eq. (3) also require $3d_h$ and d_h operations, respectively. Therefore, the computational core of LSTMs is dominated by the matrix-vector multiplication of $W_h h_{t-1} + W_x x_t$.

In order to reduce the computational complexity of LSTMs, we train LSTMs such that they learn to retain only the important information in their hidden states h_{t-1}. In fact, we train LSTMs with a sparsity constraint: all values below the threshold T are pruned away from the network, as shown in Fig. 1. During the training phase, we prune the state vector h_{t-1} when performing the feed-forward computations while its dense representation is used during the update process of the LSTM parameters. In fact, the feed-forward computations of LSTMs remains the same as the conventional computations except for Eq. (1) which can be mathematically formulated as

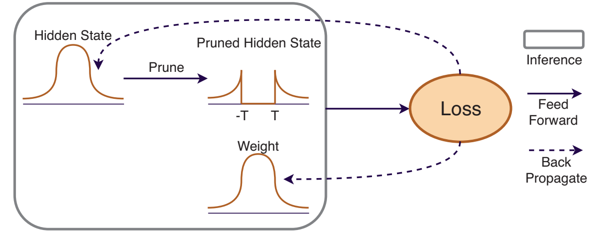

Fig. 1. The pruning procedure.

$$\begin{pmatrix} f_t \\ i_t \\ o_t \\ g_t \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W_h h_{t-1}^p + W_x x_t + b, \quad (4)$$

where h_{t-1}^p denotes the sparse state vector and is obtained by

$$h_{t-1}^p = \begin{cases} 0 & \text{if } |h_{t-1}| < T \\ h_{t-1} & \text{if } |h_{t-1}| \geq T \end{cases}. \quad (5)$$

Maintaining the dense representation of the state vector during the update process allows the state values initially lied within the threshold to be updated. This technique was first introduced in [14] to binarize the weights, while we use it to prune the state vector in our work. During the update process, we need to compute the gradient on the hidden state h_0. Due to the discontinuity of the rectangular function used to obtain h_{t-1}^p at the threshold value, we approximate the derivatives on the hidden state h_0 by computing

$$\frac{\partial L}{\partial h_0} \approx \frac{\partial L}{\partial h_0^p}. \quad (6)$$

B. Experimental Results

In this section, we evaluate the performance of the proposed training algorithm that prunes away the noncontributory information of the hidden state vector h_{t-1} to the prediction accuracy on different temporal tasks: classification of handwritten digits on sequential MNIST [15] and both character-level and word-level language modeling tasks on Penn Treebank corpus [16]. Since the pruning threshold is empirical, we report the prediction accuracy of the above tasks for different sparsity degrees while using an 8-bit quantization for all weights and input/hidden vectors. Of course, we cannot always retain the prediction accuracy at the same level of the dense model for any sparsity degree, as pruning too much hurts the prediction accuracy.

1) Character-Level Language Modeling: The main goal of character-level language modeling is to predict the next character. For this task, the performance is measured by the bits per character (BPC) metric, where a low BPC is desirable. For this task, we use an LSTM layer with 1000 units (i.e., d_h) followed by a classifier. The model is then trained on the Penn Treebank dataset with a sequence length of 100. We minimize the cross entropy loss using stochastic gradient descent on mini-batches of size 64. The update rule is determined by ADAM with a learning rate of 0.002.

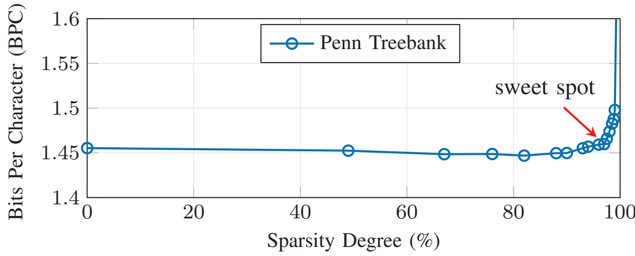

Fig. 2. Prediction accuracy of character-level language modeling on the test set of the Penn Treebank corpus for different sparsity degrees over a sequence length of 100.

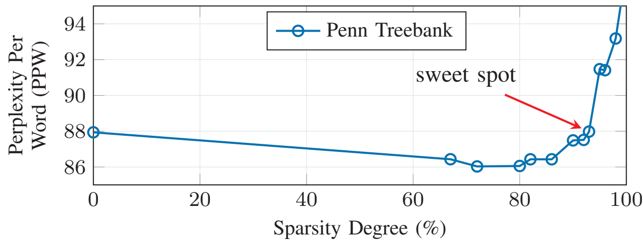

Fig. 3. Prediction accuracy of word-level language modeling on the test set of Penn Treebank for different sparsity degrees over a sequence length of 35.

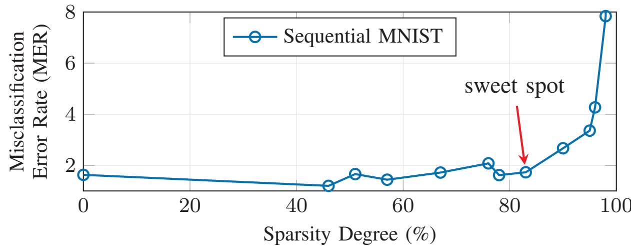

Fig. 4. Misclassification error rate of image classification task on the test set of MNIST.

For the data preparation, we use the same configuration as [3]: we split this dataset into 5017k, 393k and 442k training, validation and test characters, respectively. Penn Treebank corpus has a vocabulary size of 50. For the character-level language modeling task, the input vector is one-hot encoded.

Fig. 2 reports the prediction accuracy on the test set for different sparsity degrees in terms of BPC. The experimental results show that over 90% of information in the hidden states can be pruned away without incurring any degradation in the prediction accuracy. We can also see some improvements in the prediction accuracy in the pruned models. In fact, the pruning algorithm acts as a form of regularization that prevents the model from over-fitting. For implementation purposes, we focus on the model with the sparsity degree of 97% (i.e., the sweet spot), which results in no accuracy degradation compared to the dense model.

2) Word-Level Language Modeling: For the word-level task, we use Penn Treebank corpus with a 10K vocabulary size. Similar to [3], we split the dataset into 929K training, 73K validation and 82K test tokens. We use one LSTM layer of size 300 followed by a classifier layer to predict the next word. The performance is evaluated on perpelexity per word (PPW). Similar to character-level tasks, the models with lower PPW are better. We also use an embedding layer of size 300 to reduce the dimension of the input vector [3]. Therefore, the input vector x_t contains real values as opposed to the character-level modeling task in which the input vector is one-hot encoded.

We train the model with the word sequence of 35 while applying the dropout probability of 0.5 on the non-recurrent connections similar to [17]. We train the network using the learning rate of 1 and the learning decay factor of 1.2. We also clip the norm of the gradients to 5. Fig. 3 shows the predication accuracy of the pruned models performing the word-level task on the test set of Penn Treebank. From the experimental results, we observe that over 90% of information in the hidden state can be pruned away without affecting the prediction accuracy.

3) Image Classification: We perform an image classification task on MNIST dataset containing 60000 gray-scale images (50000 for training and 10000 for testing), falling into 10 classes. For this task, we process the pixels in scanline order: each image pixel is processed at each time step similar to [15]. To this end, we use an LSTM layer of size 100 and a softmax classifier layer. We also adopt ADAM step rule with the learning rate of 0.001. The misclassification error rate of the models with pruned states is illustrated in Fig. 4. Similar to the language modeling tasks, the MNIST classification task can be performed with the models exploiting over 80% pruned hidden state without affecting the misclassification error rate.

III. Hardware Implementation

So far, we have shown that over 90% of the hidden vector h_{t-1} can be pruned away without any accuracy degradation over a set of temporal tasks. This observation suggests that over 10× speedup is expected when performing the matrix-vector multiplication $W_h h_{t-1}$. However, exploiting the sparse hidden vector in specialized hardware to speed up the recurrent computations is nontrivial and introduces new challenges. The first challenge is to design a dataflow to efficiently perform the recurrent computations, as the LSTM network involves two different types of computations: the matrix-vector multiplication (i.e., Eq.(1)) and the Hadamard product (i.e., Eq. (2) and Eq. (3)). Secondly, the bandwidth of the off-chip memory is limited, making the fully exploitation of the parallelism difficult. Finally, a customized decoding scheme is required to denote the indices to the unpruned information. In this section, we propose a dataflow for recurrent computations and its architecture, which efficiently exploit the pruned hidden vector to skip the ineffectual computations under a bandwidth limited interface with the off-chip memory.

A. Recurrent Computation Dataflow

As discussed in Section II-A, the main computational core of LSTM involves two vector-matrix multiplications. A typical approach to implement a vector-matrix multiplications is a semi-parallel implementation, in which a certain number of processing elements (PEs) are instantiated in parallel to perform multiply-accumulate operations. In this approach, all the PEs share the same input at each clock cycle and the computation is performed serially. Fig. 5(a) illustrates an example performing a vector-matrix multiplication on an input vector of size 6 and a weight matrix of size 4×6. In this example, 4 PEs are

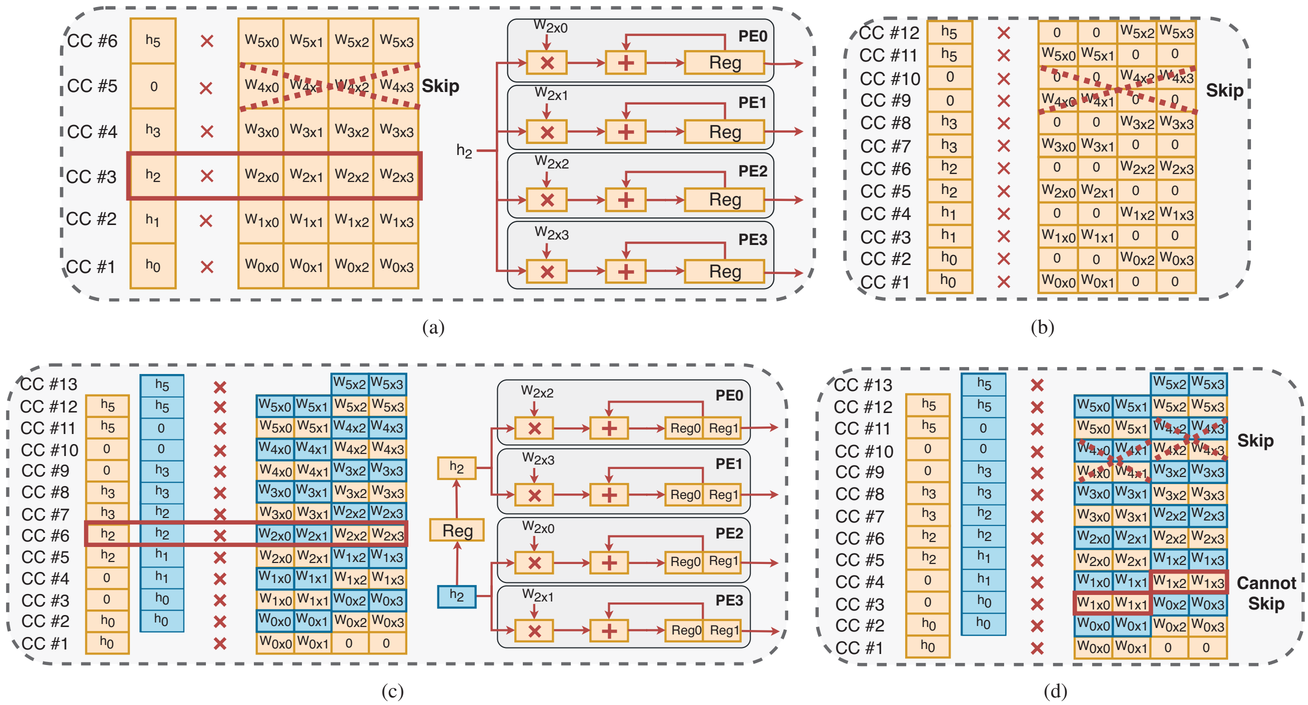

Fig. 5. A vector-matrix multiplication: (a) under an unlimited data bandwidth with a batch size of 1, (b) under a limited data bandwidth with a batch size of 1 and (c) under a limited data bandwidth with a batch size of 2. (d) Skipping is allowed only when both batches contain zero-valued inputs at the same position.

required in parallel to perform the computations serially within 6 clock cycles. Clearly, the computation of the zero-valued elements of the input vector can be skipped to speed up the process. However, in a larger scale, this approach requires high memory bandwidth to provide weights for the PEs in parallel. Let us consider that the memory bandwidth provides only a single input element and two weight values at each clock cycle as an example. In this case, 12 clock cycles are required to perform the computations when using 4 PEs in parallel. In fact, the latency is doubled for the bandwidth limited scenario while the utilization factor of PEs are reduced to 50% as shown in Fig. 5(b).

To increase the throughput and utilization factor of the bandwidth limited scenario, we use a higher batch size. In this case, we need to use more memory elements to store the partial values for each batch. For instance, we use a batch size of 2 to fill the empty pipeline stages in Fig. 5(b). More precisely, the first input element of the first batch and the first two weights for the first two PEs are read. The weights for the PEs are then stored into the registers to be used for the first input element of the next batch. Meanwhile, the first two PEs perform the multiply-accumulate operations between the weights and the first input element of the first batch. The partial result is stored into the scratch memory cells. In the second clock cycle, the first input element of the second batch along with the weights for the second two PEs are read from the off-chip memory. Meanwhile, the first element of the first batch is passed to the second two PEs through the pipeline stages. Therefore, the computations of the second two PEs are performed with the first element of the first batch and the computations of the first two PEs with the first element of the second batch. After 2 clock cycles for this example, all the PEs process the multiply-accumulate operations in parallel and the pipeline stages are full, resulting in a high utilization factor as shown in Fig. 5(c).

While using a higher batch size results in a higher utilization factor and throughput, each positional element of all the batches has to be zero in order to skip the computations of its corresponding clock cycles. Let us consider the previous example again. If the second input element of the first batch is zero while the second input element of the second batch is non-zero, we cannot skip its corresponding clock cycles since both batches share the same weights as depicted in Fig. 5(d). Therefore, we can only skip those computations in which all the input elements of the all batches are zero (see Fig. 5(d)).

B. *Zero-State-Skipping Accelerator*

Since this paper targets portable devices at the edge with limited hardware resources and memory bandwidth, we use the aforementioned scheme to exploit the sparse hidden vector in order to accelerator the recurrent computations. Fig. 6 shows the detailed architecture of the proposed accelerator for LSTM networks. To this end, we exploit four tiles: each tile contains 48 PEs performing multiply-accumulate operations. In fact, each tile is responsible to obtain the values of one gate by computing Eq. (4). Therefore, the first three tiles and the fourth tile are equipped with sigmoid and tanh functions, respectively. We also adopt LPDDR4 as the off-chip memory, which easily provides a bandwidth of 51.2 Gbps [18]. More precisely, this data bandwidth provides 24 8-bit weights and a single 8-bit input element for 24 PEs at a nominal frequency of 200 MHz. As a result, we associate each PE with a 16×12-bit scratch

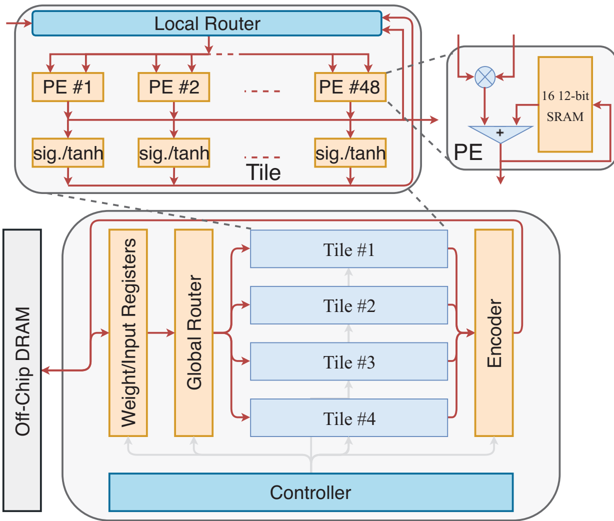

Fig. 6. The proposed LSTM accelerator.

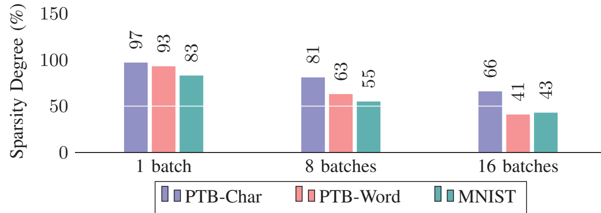

Fig. 7. Sparsity degree of the hidden state vector over different batch sizes.

memory to store the partial products for 16 batch sizes. For the Hadamard products of Eq. (2), the first tile performs the element-wise product of the first term (i.e., $f_t \odot c_{t-1}$) by reading the values of c_{t-1} from the off-chip memory. Meanwhile, the computations for the second term ($i_t \odot g_t$) is performed in the second tile. Then, the results of the both terms are passed to the fourth tile to perform the addition and obtain tanh(c_t). It is worth mentioning that all the PEs can take inputs from either the off-chip memory or the scratch memory of the same tile or others through the global and local routers (see Fig. 6). Finally, the computations of Eq. (3) are performed in the third tile to obtain h_t. The obtained results are then passed to an encoder that keeps track of zero-valued elements using a counter. More precisely, the encoder counts up if the current input value of all the batches is zero. Afterwards, the obtained offset is stored along with the hidden state vector into the off-chip memory. During the recurrent computations of the next time step, the offset is only used to read the weights that correspond to the non-zero values. Therefore, no decoder is required in this scheme. The weight/input registers in Fig. 6 are also used to provide the pipeline stages for the batches.

C. Methodology

We implemented the proposed accelerator in Verilog and synthesized via Cadence Genus Synthesis Solution using TSMC 65nm GP CMOS technology. We also used Artisan dual-port memory compiler to implement the scratch memories for PEs.

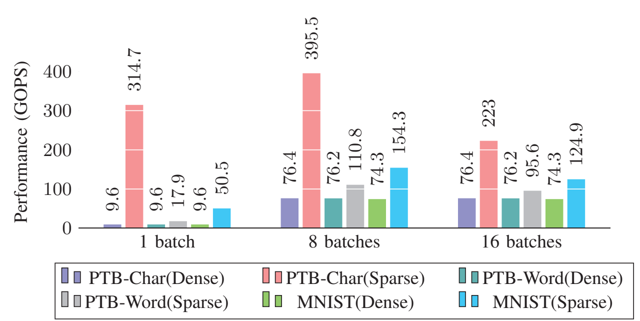

Fig. 8. Performance of the proposed accelerator over dense and sparse models.

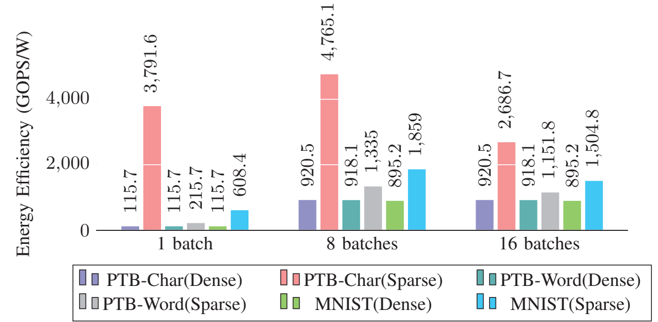

Fig. 9. Energy efficiency of the proposed accelerator over dense and sparse models.

The proposed accelerator occupies a silicon area of 1.1 mm^2 and yields a peak performance of 76.8 Gops and performance efficiency of 925.3 Gops/W over dense models at a nominal frequency of 200 MHz. For evaluation purposed, we perform the recurrent computations on the proposed accelerator at its nominal frequency over a set of temporal tasks when using both dense and sparse representations for the hidden state vector h_{t-1}.

D. Implementation Results

In Section II-B, we showed that the accuracy curves of different temporal tasks for different sparsity degree when using a batch size of one. However, due to the limited bandwidth of the off-chip memory, we use higher batch sizes to increase the utilization factor of PEs and improve the throughput of the accelerator (see Section III-A). To exploit the sparsity among information of the hidden state in order to speed up the recurrent computations, all the elements of all the batches must be zero. For instance, all the third elements of all the batches must be zero to skip their ineffectual computations. As a result, such a constraint incurs a sparsity degradation. Fig. 7 shows the sparsity degree of the models with the most sparse hidden state for different batch sizes while incurring no accuracy degradation (i.e., the sweet spots). The run-time energy efficiency and performance of the proposed accelerator for both dense and sparse models and different batch sizes are reported in Fig. 8 and Fig. 9, respectively. Performing the recurrent computations using the sparse hidden state results in up to 5.2× speedup and energy efficiency compared to the most energy-efficient dense model.

Fig. 10. Peak performance of the state-of-the-art accelerators.

IV. Related Work

During the past few years, many works have focused on designing custom accelerators for CNNs. For instance, DianNao exploits straight-forward parallelism by adopting an array of multiply-accumulate units to accelerate the computations of CNNs [19]. However, such an approach requires numerous memory accesses to the off-chip memory, which dominates its power consumption. DaDianNao was introduced in [20] to eliminate the memory accesses to the off-chip memory by storing the weights on-chip. Cnvlutin [11] relies on DaDianNao architecture and exploits the intrinsic sparsity among activations incurred by using the ReLU as the non-linear function to speedup the convolutional process. Cnvlutin showed that skipping the ineffectual computations with zero-valued activations can improve the performance by up to 1.55×. In [9], a custom accelerator was proposed to perform vector-matrix multiplications of fully-connected networks using sparse and compressed weights. While all the aforementioned architectures have proven to be effective, they accelerate only either CNNs or fully-connected networks, not LSTMs [12].

In [12], a custom architecture, called ESE, was introduced to accelerate the computations of LSTMs. ESE exploits the sparsity among the weight matrices to speed up the recurrent computations. More precisely, it was shown that performing the recurrent computations on the model with sparse weights is 4.2× faster than the model with dense weights when running both models on ESE. In [21], a new sparse matrix format and its accelerator called CBSR were introduced to improve over ESE architecture. This work showed that using the new sparse format improves the performance by 25%∼30% over ESE. The two above works exploited the sparsity among the weights in LSTMs to speed up the recurrent computations. However, our work in this paper takes a completely different approach by first learning the ineffectual information in the hidden state and then exploiting the sparsity among them to accelerate the recurrent computations while using dense weights. Fig. 10 compares the proposed accelerator in this paper with ESE and CBSR. In fact, our work outperforms both ESE and CBSR in terms of performance by factors of 1.9× and 1.5× respectively. It is worth mentioning that we have used the improvement factor of CBSR over ESE to estimate the performance of CBSR architecture in Fig. 10. Moreover, ESE yields a peak energy efficiency of 61.5 GOPS/W on a Xilinx FPGA while our accelerator results in a peak energy efficiency of 4.8 TOPS/W on an ASIC platform. Therefore, a direct comparison in terms of energy efficiency does not construct a fair comparison.

V. Conclusion

In this paper, we first introduced a new training scheme that learns to retain only the important information in the hidden state h_{t-1} of LSTMs. We then showed that the proposed method can prune away over 90% of the hidden state values without incurring any degradation on the prediction accuracy over a set of temporal tasks. We also introduced a new accelerator that performs the recurrent computations on both dense and sparse representations of the hidden state vector. Finally, we showed that performing the recurrent computations on the sparse models results in up to 5.2× speedup and energy efficiency over their dense counterparts.

References

[1] Y. Lecun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, 5 2015.

[2] Y. LeCun, B. Boser *et al.*, “Backpropagation Applied to Handwritten Zip Code Recognition,” *Neural Comput.*, vol. 1, no. 4, pp. 541–551, Dec. 1989.

[3] T. Mikolov, M. Karafiát *et al.*, “Recurrent neural network based language model,” in *Eleventh Annual Conf. of the International Speech Communication Association*, 2010.

[4] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in *NIPS ’14*. Cambridge, MA, USA: MIT Press, 2014, pp. 3104–3112.

[5] A. Graves, A. rahman Mohamed, and G. E. Hinton, “Speech Recognition with Deep Recurrent Neural Networks,” *CoRR*, vol. abs/1303.5778, 2013.

[6] O. Vinyals, A. Toshev *et al.*, “Show and Tell: A Neural Image Caption Generator,” 2014, cite arxiv:1411.4555.

[7] R. Pascanu, T. Mikolov, and Y. Bengio, “On the Difficulty of Training Recurrent Neural Networks,” in *Proceedings of the 30th International Conference on Machine Learning - Volume 28*, ser. ICML’13. JMLR.org, 2013, pp. III–1310–III–1318.

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

[9] S. Han, X. Liu *et al.*, “EIE: Efficient Inference Engine on Compressed Deep Neural Network,” in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 243–254.

[10] A. Ardakani, C. Condo, and W. J. Gross, “A Convolutional Accelerator for Neural Networks With Binary Weights,” *IEEE International Symposium on Circuits & Systems (ISCAS)*, 2018.

[11] J. Albericio, P. Judd *et al.*, “Cnvlutin: Ineffectual-neuron-free Deep Neural Network Computing,” in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA ’16. IEEE Press, 2016, pp. 1–13.

[12] S. Han, J. Kang *et al.*, “ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA,” in *Proceedings of the 2017 ACM/SIGDA International Symposium on FPGA*, ser. FPGA ’17. New York, NY, USA: ACM, 2017, pp. 75–84.

[13] C. Xu, J. Yao *et al.*, “Alternating Multi-bit Quantization for Recurrent Neural Networks,” in *ICLR ’18*, 2018.

[14] M. Courbariaux, Y. Bengio, and J. David, “BinaryConnect: Training Deep Neural Networks with binary weights during propagations,” *CoRR*, vol. abs/1511.00363, 2015.

[15] Q. V. Le, N. Jaitly, and G. E. Hinton, “A simple way to initialize recurrent networks of rectified linear units,” *CoRR*, vol. abs/1504.00941, 2015.

[16] M. P. Marcus, M. A. Marcinkiewicz, and B. Santorini, “Building a Large Annotated Corpus of English: The Penn Treebank,” *Comput. Linguist.*, vol. 19, no. 2, pp. 313–330, Jun. 1993.

[17] W. Zaremba, I. Sutskever, and O. Vinyals, “Recurrent Neural Network Regularization,” *CoRR*, vol. abs/1409.2329, 2014.

[18] Micron Technology Inc., “DDR4 SDRAM for Automotive.” [Online]. Available: http://www.micron.com

[19] T. Chen, Z. Du *et al.*, “DianNao: A Small-footprint High-throughput Accelerator for Ubiquitous Machine-learning,” in *Proceedings of the 19th ASPLOS*. New York, NY, USA: ACM, 2014, pp. 269–284.

[20] Y. Chen, T. Luo *et al.*, “Dadiannao: A machine-learning supercomputer,” in *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec 2014, pp. 609–622.

[21] J. Park, J. Kung *et al.*, “Maximizing system performance by balancing computation loads in lstm accelerators,” in *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2018, pp. 7–12.