CompressedCache: Enabling Storage Compression on Neuromorphic Processor for Liquid State Machine

Zhijie Yang, Rui Gong, Lianhua Qu, Ziyang Kang, Li Luo, Lei Wang, and Weixia Xu

College of Computer Science and Technology, National University of Defense Technology, Changsha, Hunan, P.R.China
yangzhijie@nudt.edu.cn, Leiwang@nudt.edu.cn

Abstract. Spiking Neural Network (SNN) based neuromorphic processors have gained momentum due to their high energy efficiency. As a kind of SNN, Liquid State Machine (LSM) shows potential in domains such as image recognition and speech recognition, and it is simpler to train than other SNNs. In neuromorphic processors, weights and synapses are stored on-chip to reduce the energy cost of data movement. However, the storage of them is redundant if the deployed network on the neuromorphic processor is LSM which is a sparse SNN. By exploiting the sparsity of LSM, adopting storage compression can reduce the power consumption of the processor or enable a single chip to deal with more complex tasks with more logic neurons. In this work, we propose a lossy storage compression method, Compressed Sparse Set Associative Cache (CSSAC) which makes use of the sparsity and the robustness of LSM. We apply CSSAC on an LSM-oriented neuromorphic processor to demonstrate how the hardware design supports CSSAC to enable storage compression and complete LSM computation. CSSAC does not introduce much metadata overhead to ensure the compression effect, nor does it decrease the accuracy of LSM or the performance of the processor. Experimental results show that in our implementation, CSSAC can, at best, result in 14%-55% reduction in on-chip storage and 5%-46% reduction in power consumption of the processor under different weight data widths on MNIST, NMNIST, DVS128 Gesture datasets.

Keywords: spiking neural network, neuromorphic processor, liquid state machine, storage compression

1 Introduction

SNNs [1–3] and neuromorphic processors [4–6] have attracted much attention and developed rapidly due to the characteristics of modeling the behavior of neurons in the brain and high energy efficiency. As a kind of SNN, LSM [1] has shown great potential in the fields of image recognition and speech recognition [7]. Because it is natural to use LSM to recognize the spike trains produced by various new sensors, such as Dynamic Vision Sensor (DVS) [8] and Dynamic Audio Sensor (DAS). Besides, compared with other SNNs, the training of LSM is simpler because only the readout layer of it, which is generally a single-layer fully connected layer, needs to be trained. Moreover, different readout layers can share the same versatile reservoir layer which is responsible for data pre-processing, to deal with multiple tasks.

In neuromorphic processors like TrueNorth [4] and Loihi [5], all synapses and weights are reserved on-chip storage to support the deployment of different kinds of SNNs with dense or sparse connectivities. However, the storage reserved for weights

and synapses is redundant if the deployed network on the neuromorphic processor is LSM which is sparse. It limits the number of logic neurons of LSM that a single chip with the fixed area can support, which influences the processing ability and the best accuracy of LSM. Besides, in the neuromorphic processor such as TrueNorth, the power consumption used for storage is several times that of computing and communication. Thus, compressing the storage of weights and synapses can reduce power consumption or increase the number of logic neurons on a single chip without adding more storage, which is vital to enabling a single chip to handle more complex tasks and the construction of multi-core neuromorphic processor to simulate larger-scale biological neural networks in the next generation.

In this work, we propose CSSAC, a lossy storage compression method for LSM-oriented neuromorphic processor. It makes use of the sparsity and the robustness of LSM to enable storage compression without incurring accuracy loss of LSM. To demonstrate how the hardware design supports CSSAC to enable storage compression and LSM computation, we design a hardware neuron for LSM-oriented neuromorphic processor using CSSAC storage organization. Thus, CSSAC can be used in hardware neurons of similar neuromorphic processors deployed sparse SNNs like LSM. To sum up, our main contributions are as follows:

- We design a lossy storage compression method, CSSAC, for LSM-oriented neuromorphic processor. It makes use of the sparsity and the robustness of LSM to enable storage compression on weights and synapses of LSM and does not incur LSM accuracy loss, large metadata overhead nor processor performance degradation.
- We design a hardware neuron for LSM-oriented neuromorphic processor using CSSAC storage organization, demonstrating how the hardware design supports CSSAC to reduce the storage and power consumption and to complete LSM computation.

The experimental results show that, in our implementation, CSSAC can, at best, result in 14%-55%, reduction in storage and 5%-46%, reduction in power consumption under different weight data widths on MNIST, NMNIST [9], DVS128 Gesture [10] datasets when compared with the uncompressed implementation.

2 Background and Motivation

In this section, we will introduce the basic structure of LSM and compressed sparse storage methods. Then we will introduce our motivation, the sparsity of LSM and the robustness of LSM.

2.1 Liquid State Machine

As shown in Fig. 1, LSM consists of the following three components. The input layer, which is a feed-forward neural network for receiving external spike trains produced by new sensors such as DVS and DAS. The reservoir layer, which is a spiking recurrent neural network composed of multiple neurons for data pre-processing. The readout layer, which can be many kinds of structures such as Support Vector Machines (SVM), fully connected neural networks, and linear regression models, etc. with plastic weights for classification.

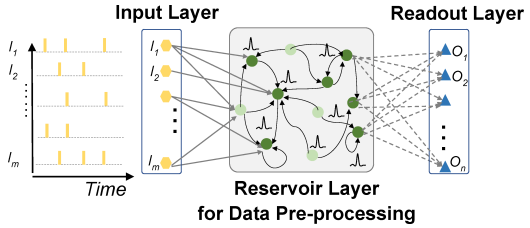


Fig. 1. The typical structure of LSM. The reservoir layer is a spiking recurrent neural network for data pre-processing with implastic weights. The weights in the readout layer are plastic.

As the core of LSM, the reservoir layer is the most time-consuming and has the most complex structure, among the three layers of LSM. As shown in Fig. 1, there are two different types of neurons in the reservoir layer. Ones are the excitatory neurons (the dark green ones) and others are the inhibitory neurons (the light green ones). So there are four kinds of synapses in the reservoir layer according to the difference between the starting and ending neurons. They are excitatory-to-excitatory synapse, excitatory-to-inhibitory synapse, inhibitory-to-excitatory synapse, and inhibitory-to-inhibitory synapse.

The topology of the reservoir layer is generated according to different connection probabilities of the four connections in network initialization. The weights of the reservoir layer will remain unchanged during both inference and training after the initialization. While only the weights of the readout layer are plastic and can be trained to do classifications of different tasks. This feature makes LSM training simpler and less time-consuming than other SNNs.

2.2 Compressed Sparse Storage Formats

Compressed sparse storage formats are used to exploit the sparsity of neural networks to enable storage compression. The main idea of it is that only the non-zero entries will be stored, together with their indices which uniquely identify each entry. The existing sparse storage formats includes Compressed Sparse Row (CSR) [11], Compressed Sparse Fiber (CSF) [12], co-ordinate [13] and their variants [14]. In these formats, the non-zero entries are stored contiguously in the memory along with their indices. For example, each entry in CSR is $(dw + iw)$ long, where dw and iw are the bit widths of the non-zero elements and their indices, respectively. The bit width of the index is determined by the number of entries before compression. Therefore, when the network sparsity is high enough, using these compressed sparse storage methods can bring benefits. However, when the network sparsity is not high enough, the metadata overhead brought by them will make the effect worse or even counterproductive.

2.3 Motivation

In this section, we will introduce the sparsity and robustness of LSM which provides us a chance to enable lossy storage compression on LSM-oriented neuromorphic processor. Then we will introduce our motivation.

The Sparsity of LSM Lsm is spare both in space and time. The sparsity in space of LSM is the key to storage compression. During the initialization of the reservoir layer, the four types of synapses are randomly generated with different connecting probabilities. Through a large number of experiments, we find the optimal connection probabilities that can make the LSM network reach the highest accuracy under different datasets.

With the optimal connection probabilities, averagely, the total connection probability is about 34.7%. In other words, 65.3% of all weights are zero. Thus these weights can be compressed and their storage can be saved by using the compressed sparse storage method.

The Robustness of LSM We observe that if we randomly replace a certain proportion of the non-zero weights with a non-zero value in the reservoir layer, the LSM accuracy will not decrease after the replacement. So LSM has a certain degree of robustness.

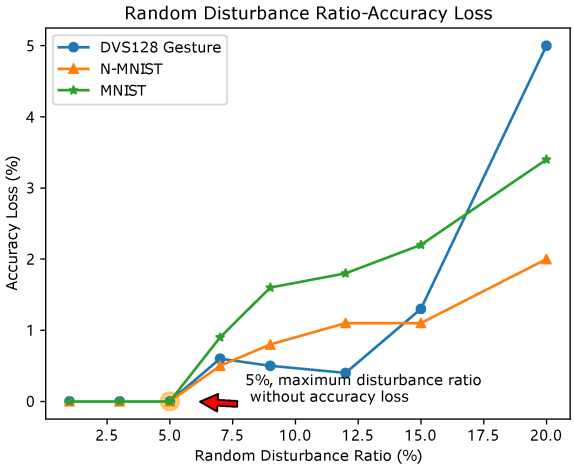


Fig. 2. Relationship between random disturbance ratio on non-zero weights and accuracy loss under different datasets.

We conduct an experiment to further quantify the robustness of LSM. First, we generate and initialize an LSM network. Then we train the weights of its readout layer until the accuracy of the network is converged. After that, we keep the weight of the readout layer unchanged, randomly replace a certain percentage of the non-zero weights with a non-zero value in the reservoir layer. We define this kind of operation as the random disturbance. After the disturbance, we observe the relationship between the random disturbance ratio and accuracy loss. As shown in Fig. 2, when the random disturbance ratio is less than or equal to 5% on the MNIST, NMNIST, and DVS128 Gesture datasets, the LSM accuracy will not lose.

In summary, the static sparsity of LSM provides us a chance to enable storage compression on LSM-oriented neuromorphic processor. The robustness of LSM allows us to change a small part of non-zero weights without incurring the accuracy loss of LSM. These two features of LSM create the foundation for the lossy sparse compression method on LSM-oriented neuromorphic processor to reduce the power consumption or increase supported logic neurons with the fixed chip area.

3 Compressed Sparse Set Associative Cache

In this section, we will present the details of CSSAC and analyzes the metadata overhead it brings. The discussion about the sensitivity of the parameters is in Section 6.

The main idea of the CSSAC approach is to compress the storage by storing only the non-zero weights on the chip and organizing the weight memory as a read-only set-

associative cache. During the processor initialization, All non-zero weights are transferred to the on-chip weight memory to be stored. Weights that have nowhere to be stored due to the storage limit will be discarded. But their synapses information is preserved on the chip. Thus discarded weights can be replaced by other weights stored on chip when needed.

3.1 Details of the Method

In the CSSAC method, we first group all weights including zero ones and non-zero ones of a neuron into equal groups. Then we generate tags based on higher bits of their addresses for them. With the tag, a weight can be distinguished from other weights within the same group.

All non-zero weights and their tags will then be transferred to the belonging set in memory on-chip. But some of them will have no place to store because we reduce entries in each set in on-chip memory by the same amount. The number of reduced entries in each set is determined by the compression ratio under the random disturbance ratio without accuracy loss. Thus, these weights will be discarded. Note that some storage units may be empty. The on-chip memory is organized as a read-only set-associative cache after the above initialization. Each stored entry is $(dw + tw)$ long, where dw and tw are the bit widths of the weights and their tags respectively. tw is calculated as $tw = \lceil log2(N) \rceil$, where N is the number of entries in a set before the compression. Besides, all of the synapses information is stored in an on-chip adjacent vector in which each bit indicates whether a synapse exists or not, to distinguish weights that are discarded in compression from zero weights. Thus, if the discarded weights are needed, their existence can be known through synapses information and they can be replaced by the weights stored in the same set to be used.

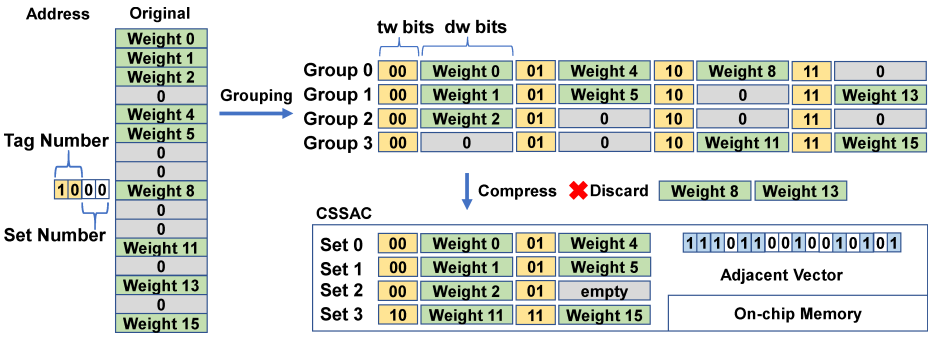


Fig. 3. Compressed sparse set-associative cache.

During the process of computation, a request for weight will lead to simultaneous accesses to both the weight memory and the adjacent vector. By the modulus operation between the requested weight index and the number of sets, the corresponding set number is generated. All tags in the target set are read out and compared with the tag of request weight. If the tag of the requested weight hits one of the tags in the set, the corresponding weight is read out to be used. If not and the bit corresponding to the requested weight in the adjacent vector is "1", the first item in the same set is read out to replace the requested one.

As shown in Fig. 3, we give an example of the CSSAC. Assume that a neuron has 16 synapses and weights. Only 9 of synapses are existent, i.e. the weights of them are non-zero, due to the static sparsity. Firstly, weights are grouped into 4 groups equally based on the modulo operation of their indexes. Each set has four entries. So the bit width of the tag is 2. The entries of each set are reduced to 2 in on-chip memory. During the process of transferring all non-zero weights and their tags to on-chip memory, $weight$ 8 and $weight$ 13 are discarded because of the limitation of storage space. Besides, the adjacent vector is also set to store all synapses information without compressing.

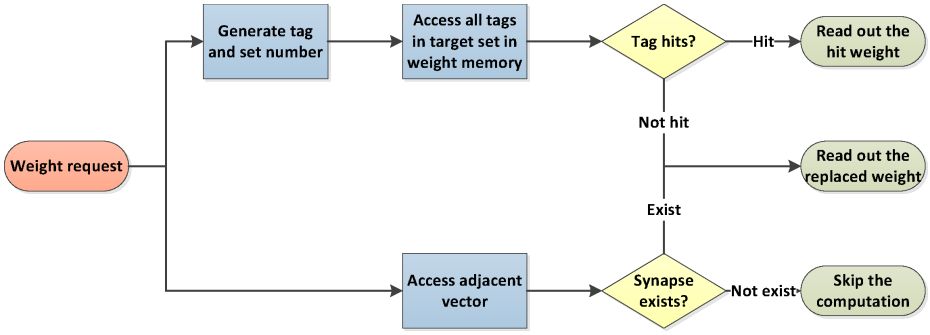


Fig. 4. The process of accessing a request weight in CSSAC.

The process of accessing a request weight is executed as the sequence in Fig. 4 and the storage of CSSAC is organized as on-chip storage shown in Fig. 3. For instance, if $weight$ 3 is requested, the computation will be skipped. Because the synapse of $weight$ 3 is found as "0" in the adjacent vector. If $weight$ 8 is requested, its tag will not be found in weight memory. But in the adjacent vector, the synapse of $weight$ 8 will be found as "1". Thus, the first entry in the same set, $weight$ 0, is read out to replace it.

3.2 Metadata Overhead

Given all weights entries of in-degree synapses of a neuron is a if it supports fully connect with other neurons. These entries are divided equally into s sets. We reduce r entries per set to compress the storage for there are not that many synapses needed to be stored due to the static sparsity of LSM. Thus the total metadata overhead M is computed as follows, which consists of tag and synapse information stored in the adjacent vector.

$$M = [(\lceil log2(\frac{a}{s})\rceil) \times (a - s \times r)] + (1 \times a) \tag{1}$$

where $1 \leq s \leq a$ and $1 \leq r \leq \frac{a}{s}$. The compression ratio C is calculated as follows:

$$C = \frac{s \times r}{a} \times 100\% \tag{2}$$

The storage reduction rate R considered metadata overhead is calculated as follows:

$$R = [1 - \frac{M + (1 - C) \times a \times dw}{a \times dw}] \times 100\% \tag{3}$$

where dw is the quantization bit widths of the weights. The discussion about the sensitivity of storage reduction under different quantization bit widths is in Section 6.

Compared with the conventional compressed sparse storage method, CSSAC has a better effect on the medium sparse network such as LSM. Because it introduces less metadata overhead than conventional methods which are discussed in Section 6.

4 CSSAC-Improved Architecture

In this section, we will briefly introduce the working process of a typical LSM-oriented neuromorphic processor. Then we introduce the hardware neuron design in detail to demonstrate how the hardware neuron supports CSSAC storage organization to reduce weight storage and completes LSM computation.

Algorithm 1: Workflow of typical LSM-oriented neuromorphic processor

Input: external input $spike_{it}, i \leq I, t \leq T$ // I is the number of input neurons
Output: Classification result R
for $t = 1; t \leq T;$ *// T is the number of time steps in one sample picture* **do**
  for $i = 1; i \leq N;$ **do**
    for $j = 1; j \leq I + N;$ *// N is the number of hardware neurons* **do**
      if $spike_{jt} == 1 \&\& synapse_{ij} == 1$ **then**
        $voltage_i = voltage_i + weight_{ij};$
      end
    end
    if $voltage_i \geq threshold$ **then**
      $spike_{i+I,t+1} = 1;$ //Generate internal spikes as input for the next time step.
      $voltage_i = 0;$
      $state_i = state_i + 1;$
    end
  end
end
R = Readout(state); //Readout is the function of readout layer of LSM.

Typical LSM-oriented neuromorphic processor works by timestep as Algorithm 1. In each time step, the external input spike train and their indices are sent to the hardware neuron by shift registers cycle by cycle. Each neuron receives the spike and index in each clock cycle and the weight is accessed according to the received index to perform membrane voltage accumulation. When all the external input spikes in a time step are processed, each neuron compares its membrane voltage with the pre-stored threshold. And each neuron will generate an output spike if the voltage is larger than the threshold for the use in the next time step as an internal input spike. Then, the computation of the next time step can be started. After all the time steps of a sample picture are performed, the acquired liquid states are used to get the classification result.

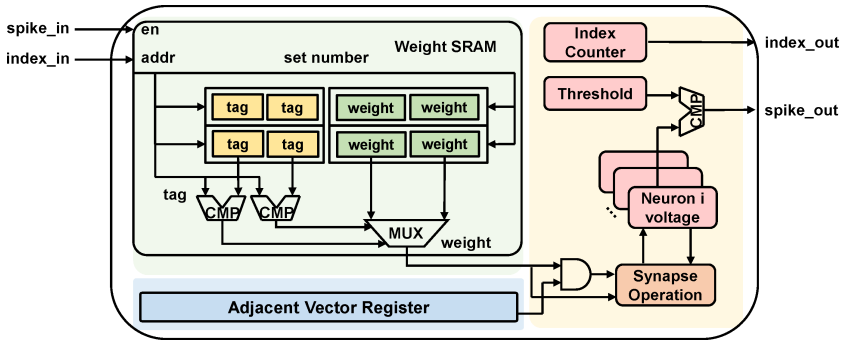
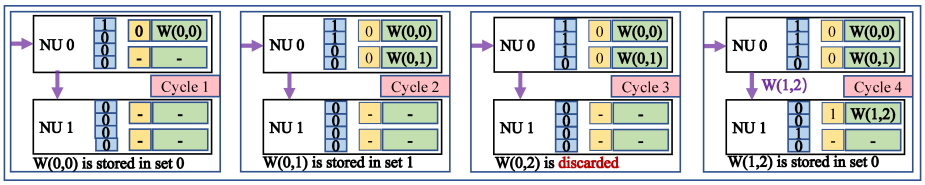


Fig. 5. Block diagram of the hardware neuron.

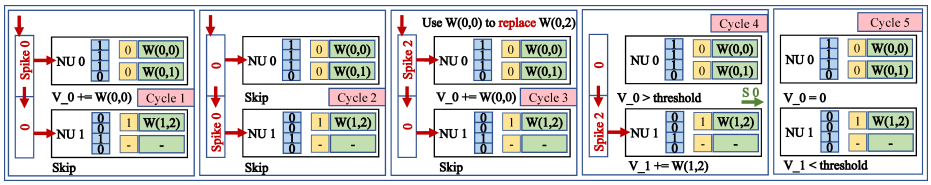
4.1 Hardware Neuron

The function of the hardware neuron is receiving the input spike, performing the accumulation of membrane voltage of neurons, and generating the output spike. As shown

in Fig. 5, the structure of a neuron consists of the following components. A weight S-RAM and a tag SRAM are used to store the weights of all logic neurons in a hardware neuron using CSSAC organization with parallel comparators and a multiplexer. An adjacent vector register that stores the uncompressed synapses information. Registers that stores the threshold and all membrane voltages of logic neurons. A synapse operation module is used to perform the accumulation of membrane voltage. A comparator used to compare the threshold with membrane voltage and to decide whether to generate an output spike or not. An index counter which indicates the index of the logic neuron being used.



(a) Initialization process of hardware neurons.



(b) Execution process of hardware neurons.

Fig. 6. Working process of hardware neurons organized using CSSAC.

The neuron works as follows, which is shown in Fig. 6. During weight initialization, the initial weights are input into the hardware neuron and stored in the weight SRAM using the CSSAC method. To simplify, assume that each neuron has 4 synapses and 2 sets. Each set has 1 entry. Weights are passed between neurons to be stored in neurons they belong to. Some weights will be discarded due to the storage limitation but their synapses will be stored in the adjacent vector such as the condition in $Cycle$ 3.

During the membrane voltage accumulation process, each neuron receives the input spike and index from the shift register. Using the index, each neuron checks the existence of the synapse in the adjacent vector. If the input spike is ”0” or the synapse is found not existent, the computation in this cycle is skipped as NU 1 dose in $Cycle$ 1, 2, and 3. If the input spike is ”1” and the synapse exists, set number and tag are generated according to the input index. Then all the tags of the target set are accessed to be compared with the generated tag. If the generated tag does not match any tags in the set, the weight in the first place of the set is directly fetched to perform the accumulation computation of membrane voltage as NU 0 does in $Cycle$ 3 of the execution process. The computational model of the neuron is performed as the leak-integration-firing (LIF) neuron model [7].

After processing all the input spikes, neurons will enter the phase of output spike generation. The calculated membrane voltage is compared with the pre-stored threshold. If the membrane voltage is greater than the threshold, an output spike is generated and the membrane voltage is reset as NU 0 does in $Cycle$ 4 and 5 of the execution pro-

cess. Otherwise, there is no output spike and the membrane voltage of the logic neuron is directly written back to the corresponding register. And the membrane voltage of the next logical neuron to be processed is taken out for computation during the same time step. Until voltage updating computations of all logic neurons are finished, computation of the next time step will be started then.

5 Experiment Setup

5.1 Simulation Infrastructure

We use Brain 2 [15], a spiking neural network simulator, to generate the topology of an LSM network containing 256 input neurons, 1024 reservoir neurons, and a 1024*10 fully-connected readout layer. Then we initialize the weights in LSM, train the network to convergence, test the LSM accuracy, and conduct the random disturbance experiment of weights in the reservoir layer. We use a python-based simulator to simulate the stored procedure of CSSAC and acquire the discard ratios under different compression ratios. Note that the effect of the discard ratio and the random disturbance ratio on the LSM accuracy is nearly equal.

5.2 Hardware Implementation

We implement the hardware architecture using RTL-level code and evaluate in 32nm ASIC technology to get experiment data of hardware. We first implement an uncompressed version of the neuromorphic processor using 1024 uncompressed hardware liquid neurons. It uses shift registers for spike transmission and contains the necessary buffers for data exchange. We then use CSSAC-improved hardware neuron implementation to replace the neurons in the uncompressed version of the neuromorphic processor. Then we get a compressed version of the neuromorphic processor implementation.

5.3 Datasets

For image recognition, we use a subset of the MNIST and N-MNIST, each including 10k images for training and 10k samples for testing. For the DVS dataset, we use the DVS128 Gesture dataset [10] which contains 11 hand gestures from 29 subjects under 3 illumination conditions.The highest accuracies the LSM we use can achieve in MNIST, N-MNIST, DVS128 Gesture datasets are 87.1%,93.1%, and 85.6%, respectively while they are 99%, 99%, and 94% [10] in state of the art work [16] using DNN.

5.4 Measurements

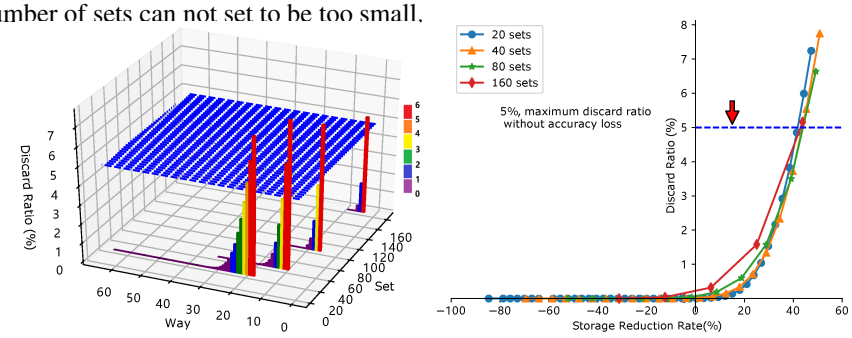
First, we measure the sensitivity of the discard ratio under the different number of sets to find the optimal parameter configuration of CSSAC. Second, we measure the compression effect under different weight data widths compared with CSR compressed sparse format. Then we measure the power consumption reduction of the processor brought by CSSAC. The baseline is the implementation without storage compression.

6 Evaluation

6.1 Discard Ratio Sensitivity Analysis on Number of Sets

To study the impact of the number of sets on the discard ratio in CSSAC stored procedure, we examine the storage compression under different numbers of set. In infrastructure, each reservoir neuron has 1280 synapses (256 from input neurons and 1024

from internal reservoir neurons). Fig. 7 (a) shows different configurations of CSSAC. First, we determine the number of sets. Then we compress the storage by reducing the number of entries in each set, i.e. ways. The configuration points under the blue dotted line plane will not bring accuracy loss. Each CSSAC configuration point corresponds to a storage reduction rate. As shown in Fig. 7 (b), the fewer sets there are, the more tolerant the discard ratio is to the storage reduction rate. Because the memory is becoming more similar to the fully associative cache when the number of sets gets smaller. But the hardware overhead of the memory increases when the number of sets gets smaller for that it needs more parallel comparators to compare the tags in the same set. So the number of sets can not set to be too small.



(a) CSSAC Configuration Space (b) Storage Reduction Rate

Fig. 7. The relationship between the storage compression and the weight discard ratio under different sets and ways numbers configuration when the data width of weight is 8-bit.

6.2 Compression Effect Sensitivity Analysis on Weight Data Width

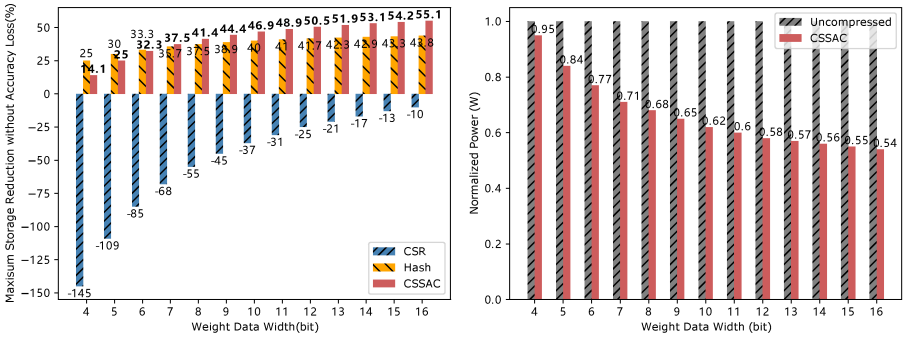
To compare the compression effect of the CSSAC, hash addressing, and CSR [11], we conduct storage compression using these three methods under different weight data widths. Fig. 8(a) shows the maximum storage reduction that can be achieved by the three methods under different weight data widths. Although the CSR method only stores non-zero weights and the storage utilization is 100%, its compression effect is worse than CSSAC due to its high metadata overhead. Another reason for the poor performance of CSR applied to LSM is that the sparsity of LSM is not particularly high, thus limiting the effect of CSR. So stored in CSR, the total storage with metadata overhead is even larger than the storage before compression.

As for hash addressing, we use a simple hash function to recode addresses for non-zero weights. The non-zero weights whose addresses are collided will be discarded. We use this method as a lossy sparse compression method to compare with CSSAC. Because this approach has less metadata overhead, it works better when the weight data width is lower. However, due to the high collision rate (more than 6%) brought by this method, the accuracy of LSM will be reduced, so we do not adopt this method.

6.3 Power Consumption Evaluation

Compressing storage with CSSAC brings power consumption reduction of the processor. We synthesize the uncompressed implementation and the compressed implementation with CSSAC of the neuromorphic processor to show the relationship between the power consumption reduction and the compression ratio. As shown in Fig. 8(b),

CSSAC brings 5%-46% reduction in power consumption under different weight data widths.



(a) Storage Reduction (b) Power Consumption Comparison

Fig. 8. The compressing effect and power consumption sensitivity under different weight quantization.

6.4 Performance Evaluation

Before storage compression using LSM sparsity, memory access in the neuromorphic processor is accomplished by direct addressing. Therefore, only one clock cycle is required when accessing the weight. In the case of using CSSAC, access to the synaptic information in the adjacent vector is also done by direct addressing and thus requires one clock cycle. But access a weight requires two clock cycles for this process contains two steps, tag comparing and weight selecting out which need two clock cycles in total. However, only if the input spike is ”1” and the synapse is existing will the two-cycle weight access process be performed in full. Due to the sparsity of LSM in time and space, the actual probability fully-executed weight access is about 0.1% averagely in the LSM we use under the mentioned three data sets. Thus, using the CSSAC results in less than 0.2% reduction in the performance of the neuromorphic processor approximately.

7 Related Work

Wang et al. [7] presented a general-purpose LSM-oriented neuromorphic learning processor with integrated training and recognition for real-world pattern recognition problems. They did not take advantage of the sparsity in LSM for storage compression. Jin et al. [17] proposed a novel sparse and self-organizing LSM architecture with a spike-timing-dependent plasticity mechanism for efficient on-chip training. In their work, they exploited the sparsity of the readout layer and realized up to 29.2% synapse reduction. But they didn’t pay attention to the sparsity of unchanged weights and synapses in the reservoir layer of LSM, which took up more storage than the readout layer. Because in their work, the number of reservoir layer neurons was only 135, the storage of weights in the reservoir layer might not be a big deal. TrueNorth [4] was a neuromorphic processor which was composed of 4096 neurosynaptic cores tiled in a 2D array. It implemented sparse memory access patterns to exploit the sparsity of SNNs but it did not adopt storage compression. Loihi [5] was a neuromorphic processor which advanced the state-of-the-art modeling of spiking neural networks in silicon. It supported three sparse matrix compression models in which fan-out neuron indices were computed based on index state stored with each synapses state variables.

To sum up, the above works do not pay attention to the storage compression of the reservoir layer in LSM. And we do not know about the details of how Loihi support the sparse compression because the authors only introduced the methods briefly in their papers.

8 Conclusion

In neuromorphic processors, the storage of weights and synapses is redundant if the deployed network is sparse, such as LSM. In this work, we design a lossy storage compression method, CSSAC which makes use of the sparsity and the robustness of LSM. CSSAC does not introduce much metadata overhead nor does it decrease the accuracy of LSM or the performance of the processor. We apply CSSAC on an LSM-oriented neuromorphic processor. Experimental results show that in our implementation, CSSAC can bring much reduction in storage and power consumption of the neuromorphic processor, which is meaningful to enabling a single chip to handle more complex tasks and the construction of multi-core neuromorphic processors to simulate larger-scale biological neural networks in the next generation.

Acknowledgement

This work is founded by National Key R&D Program of China [grant numbers 2018YFB2202603] , HGJ of China (under Grant 2017ZX01028-103-002)and in part by the National Natural Science Foundation of China [grant numbers 61802427] and [grant numbers 61832018]. And thanks to the reviewers for their efforts.

References

1. Maass, Wolfgang, Thomas Natschlager, and Henry Markram. "Real-time computing without stable states: a new framework for neural computation based on perturbations." Neural Computation 14.11 (2002): 2531-2560.
2. Diehl, Peter U., et al. Conversion of Artificial Recurrent Neural Networks to Spiking Neural Networks for Low-Power Neuromorphic Hardware. 2016 IEEE International Conference on Rebooting Computing (2016): 1-8.
3. Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks." Communications of The ACM 60.6 (2017): 84-90.
4. Akopyan, Filipp, et al. "Truenorth: Design and tool flow of a 65 mW 1 million neuron programmable neurosynaptic chip." IEEE transactions on computer-aided design of integrated circuits and systems 34.10 (2015): 1537-1557.
5. Davies, Mike, et al. "Loihi: A neuromorphic manycore processor with on-chip learning." IEEE Micro 38.1 (2018): 82-99.
6. Benjamin, Ben Varkey, et al. "Neurogrid: A mixed-analog-digital multichip system for large-scale neural simulations." Proceedings of the IEEE 102.5 (2014): 699-716.
7. Wang, Qian, Yingyezhe Jin, and Peng Li. "General-purpose LSM learning processor architecture and theoretically guided design space exploration." biomedical circuits and systems conference (2015): 1-4.
8. Berner, R., et al. "Dynamic vision sensor for low power applications." international symposium on consumer electronics (2014): 1-2.
9. Orchard, Garrick, et al. "Converting Static Image Datasets to Spiking Neuromorphic Datasets Using Saccades." Frontiers in Neuroscience (2015): 437-437.
10. Amir, Arnon, et al. "A Low Power, Fully Event-Based Gesture Recognition System." computer vision and pattern recognition (2017): 7388-7397.
11. Buluc, Aydin, et al. "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks." ACM symposium on parallel algorithms and architectures (2009): 233-244.
12. Smith, Shaden, et al. "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication." international parallel and distributed processing symposium (2015): 61-70.
13. Zhang, Shijin, et al. "Cambricon-x: an accelerator for sparse neural networks." international symposium on microarchitecture (2016): 1-12.
14. Pal, Subhankar, et al. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator." high-performance computer architecture (2018): 724-736.
15. Stimberg, Marcel, Romain Brette, and Dan F M Goodman. "Brian 2, an intuitive and efficient neural simulator." eLife (2019).
16. He, Weihua , et al. "Comparing SNNs and RNNs on Neuromorphic Vision Datasets: Similarities and Differences." arXiv (2020).
17. Jin, Yingyezhe, Yu Liu, and Peng Li. "SSO-LSM: A Sparse and Self-Organizing architecture for Liquid State Machine based neural processors." international symposium on nanoscale architectures (2016): 55-60.