A Memory-Efficient Edge Inference Accelerator with XOR-based Model Compression

Hyunseung Lee, Jihoon Hong, Soosung Kim, Seung Yul Lee, Jae W. Lee
Seoul National University, Seoul 08826, Republic of Korea
{hs_lee, athlon77, soosungkim, triomphant1, jaewlee}@snu.ac.kr

2023 60th ACM/IEEE Design Automation Conference (DAC) | 979-8-3503-2348-1/23/$31.00 ©2023 IEEE | DOI: 10.1109/DAC56929.2023.10248005

***Abstract*—Model compression is widely adopted for edge inference of neural networks (NNs) to minimize both costly DRAM accesses and memory footprints. Recently, XOR-based model compression has demonstrated promising results to maximize compression ratio and minimize accuracy drop. However, XOR-based decompression alone produces bit errors and requires auxiliary data for error correction. To minimize model size and hence DRAM traffic, we propose an enhanced decompression algorithm and a low-cost hardware accelerator for it. Since not all errors are equal, our algorithm selects only important errors to correct with no accuracy drop. Compared with the baseline XOR compression scheme correcting all errors, the compressed model size of ResNet-18 and VGG-16 is reduced by 23% and 27% respectively. We also present a low-cost hardware implementation of on-line XOR decompression and error-correction logic built on Gemmini, an open-source systolic array accelerator, at the cost of only a 0.39% and 0.46% increase in area and power.**

I. Introduction

The model size of deep neural networks (DNNs) continues to scale for applications to more complex tasks. Unfortunately, the amount of DRAM accesses, which consumes three orders of magnitude more energy than a simple arithmetic of the same bitwidth [1], also increases proportionally to the model size. One way to address this challenge is *weight pruning* [2]–[4], which can greatly reduce the model size with negligible end-to-end accuracy drop by removing parameters of low importance (e.g. with small absolute values).

Once pruned, the model parameters must be stored in a compressed format to reduce the memory footprint and DRAM traffic. An ideal compression format would feature small DRAM traffic, lossless encoding, small hardware cost, and high decompression throughput. The last requirement is also important because compressed parameters should be decompressed first to a dense format before computation as direct computation on the compressed parameters is not practical for the range of sparsity used in DNN pruning [5]. This places decompression on the critical path of inference. Thus, the decompression frontend must provide a throughput high enough to match that of the downstream stages to not stall the inference pipeline.

Compressed Sparse Row (CSR) is a popular compression format to encode pruned weights. However, for decoding, it uses multiple levels of indirection requiring complex hardware [6]. Also, the number of non-zero elements can vary widely, which leads to over-provisioning of hardware resources (e.g., on-chip buffers, decompression units) to handle the worst-case scenario. Double Viterbi [7] addresses this issue by encoding pruned parameters with a fixed number of bits. However, double Viterbi has a low decompression throughput as it needs to process the encoded bitstream in a serial manner.

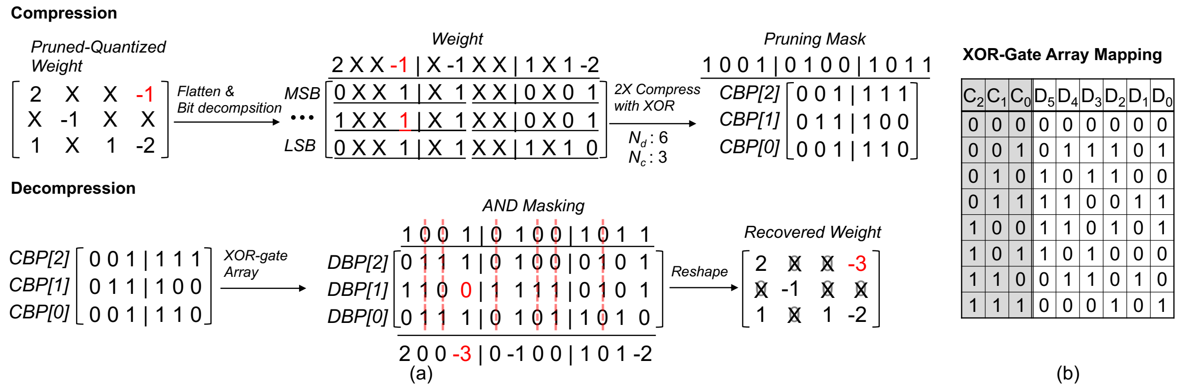
Recently, a new compression format using an XOR-gate network has been introduced [8], [9]. Like double Viterbi, this format uses fixed-size encoding, while achieving higher decompression throughput as it can take multiple bits for an input. However, this fixed-size encoding scheme also comes with its own drawbacks. The fixed encoding format lacks representability to leave some errors in the decoded bitstream. To mitigate the accuracy loss induced by errors, the double Viterbi uses iterative compression and retraining procedures. However, retraining after compression is not feasible for XOR-based compression since it cannot be represented with arithmetic operations. Thus, it uses auxiliary information that records the position of each bit error for error correction, hence recovering model accuracy.

The volume of this auxiliary information is significant, especially at a high compression ratio that exceeds the pruning sparsity. Our evaluation shows that for convolution layers of 80% sparse ResNet-18, the correction overhead is 28.4% at a $6\times$ compression ratio and 48.3% at an $8\times$ compression ratio. Reducing this overhead of error correction is critical to further increase the effective compression ratio of a pruned model.

To address this, we propose a *partial* error correction scheme that can push the limit of compression beyond the conventional full error correction. Our premise is that errors are not equally important for final accuracy. Thus, we reduce the size of the auxiliary data by only correcting a few most significant bit (MSB) errors while ignoring errors in lower-order bits. We propose an effective heuristic to determine how many bits should be rectified for each layer as noise resilience varies greatly across layers. Our partial correction scheme reduces DRAM memory traffic by 23-27% compared to full correction, which is significant on resource-constrained edge devices.

In summary, this paper makes the following contributions:

- We identify opportunities for skipping error correction for low-order bits in XOR-based model compression with no end-to-end accuracy loss.
- We introduce a model accuracy recovery algorithm, which determines the minimum number of bits to be corrected for each layer.
- We propose an area- and power-efficient hardware accelerator for this enhanced XOR-based compression algorithm and evaluate it on Gemmini, an open-source systolic array accelerator.



C_2	C_1	C_0	D_5	D_4	D_3	D_2	D_1	D_0
0	0	0	0	0	0	0	0	0
0	0	1	0	1	1	1	0	1
0	1	0	1	0	1	1	0	0
0	1	1	1	1	0	0	1	1
1	0	0	1	1	0	1	0	1
1	0	1	1	0	1	0	0	0
1	1	0	0	1	1	0	1	0
1	1	1	0	0	0	1	0	1

Fig. 1. XOR-based Compression Scheme. (a) Simple Illustration of Compression and Decompression. (b) Truth Table of Bit Matrix.

II. Background

A. Compression Format

CSR is a widely used compression format to represent sparse matrices. It stores three information: value, column index of the non-zero elements, and the number of non-zero elements in each row. This data structure yields compact storage but requires multiple steps of pointer-chasing to recover the position of each non-zero element. Also, CSR has inherent irregularity in its structure as the number of non-zero elements may vary widely across rows, hence requiring more hardware resources than those for the average case of non-zeros.

Double Viterbi encoding scheme [7] is a compression format that encodes the non-zero elements and bitmask into a bitstream of a constant size. Decompression reconstructs a fully dense array, greatly simplifying the indexing procedure. Also, since there is no irregularity in its representation, no over-provisioning of hardware resources is necessary. The downside of the Viterbi format is that a decompression unit can only take one bit as an input. This limits the decompression throughput, which is problematic when used for on-line decompression.

B. XOR-based Model Compression

XOR-gate network based on fixed-to-fixed compression scheme [8], [9] is developed to utilize constant encoding size with scalable decompression input. The operations of this scheme are illustrated in Fig. 1. It encodes a fixed-length (N_d) bit vector into a shorter fixed-length (N_c) bit vector. The regularity of the format allows efficient DRAM access. Furthermore, decompression can be implemented with a simple bit matrix multiplication through an XOR gate array. This operation can typically be conducted within one or a few cycles. Therefore, the decompression can be performed on-the-fly in a fast and efficient manner.

Off-line Model Compression. Given a DNN model, its layers first have to be pruned into a sparse matrix. Since XOR-based compression is tolerant to an irregular distribution of non-zero elements across layers, unstructured pruning is used to minimize model accuracy drop. The pruned weights are then split into bit arrays, namely bit planes, by grouping the bits of multiple data in the same bit position. Each bit plane is flattened and divided into a sequence of vectors of length N_d, which is then compressed into a vector of length N_c. The upper half of Fig. 1 illustrates an example of this process. First, the pruned, quantized weight matrix stored with a bit-width of 3 is flattened and decomposed to bit planes. Then, each bit plane is split into vectors of size $N_d = 6$, each of which is compressed to a vector of size $N_c = 3$. A bit matrix is randomly generated and used in this step of compression. The compressed vectors corresponding to the same bit positions together form $CBP[0]$ (LSB) to $CBP[2]$ (MSB).

Compression heavily depends on the existence of pruned weights. The flattened bits of pruned weights are represented as don't-care bits during compression, as they will be suppressed with the prune mask after decompression. Although there are many don't-care bits, the reduction of vector size makes the compression lossy. Therefore, not all original bits may be correctly recovered through decompression. Compression is hence a process of identifying the sequence of N_d sized vectors that exhibits a minimum number of errors after decompression. All possible sequences are examined through dynamic programming, and the one with the smallest number of errors is chosen to be the compressed result [8].

On-line Model Decompression. Decompression is a simple reversal of the compression process, as shown in Fig. 1. The same bit matrix generated and used during XOR compression is used again. Each N_c sized bit vector is converted to those of length N_d through multiplication with the matrix. Then, the recovered bit planes are combined to recover the original weight.

Instead of a naive multiplication between $N_d \times N_c$ bit matrix and a single N_c sized vector, the concatenation of N_s preceding vectors with the current vector can be used to reduce errors [9]. Then, a $N_d \times N_c \cdot (N_s + 1)$ bit matrix is multiplied to the concatenated vector. This bit-matrix-vector multiplication can be efficiently implemented with an XOR-gate array in hardware, as multiple stages of XOR gates can be processed within a single cycle. Therefore, decompression latency can be perfectly hidden by computation latency in the matrix multiply hardware in a pipelined manner.

III. Model Accuracy Recovery Algorithm

A. Bit-wise Effect of Weight Parameters on Accuracy

The decompression algorithm of the XOR-based compression scheme cannot reproduce the original data by itself. A

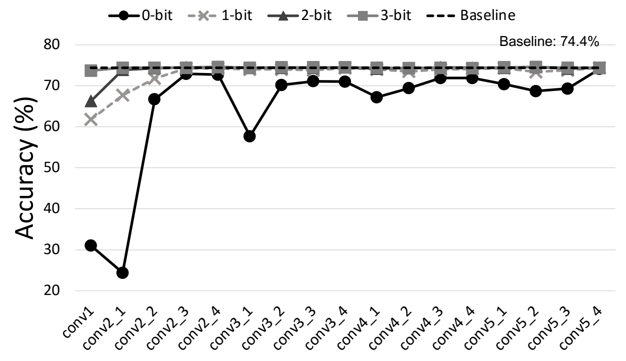


Fig. 2. Layerwise Accuracy Sensitivity of Convolution Layers (ResNet-18).

naive way to mitigate the compression loss and the consequent drop in model accuracy is to fix every single incorrect bit. However, the number of bits that need to be corrected rapidly increases with the compression ratio. Also, the cost of the auxiliary information storing the positions of erroneous bits can easily outweigh the benefits of compact representation.

Fortunately, it is not necessary to correct every single bit to recover the accuracy of the original model. A neural network is resilient to small perturbations in weight values, and the errors in different bit positions do not contribute equally to the accuracy drop. This is well illustrated in Fig. 2, which shows the accuracy of ResNet-18 when each layer's uncompressed weights are replaced with the decompressed weight values. In the graph, n-bit refers to correcting the first n bits starting from the MSB. Error corrections in a few sequential bit positions from the MSB result in sufficient restoration of the model accuracy. For example, the model accuracy is 31% when no error in the first convolution layer is corrected, but it increases to 61.8% and 66.3% respectively as we correct the errors up to the 1st and 2nd MSBs. Also, it's worthwhile to note that errors in different layers have varying effects on accuracy; errors in the 1st, 2nd, and 6th convolution layers resulted in greater accuracy drop.

B. Accuracy Recovery Algorithm

To exploit the varying impact of errors in different bit positions and layers, we partially correct the errors of each layer. The errors in only the most significant k_i bits are corrected for layer i, while the remaining less significant bits are left unfixed. k_i ranges from 0, not correcting any error in layer i, to $bitwidth$, correcting every error in all bit positions of layer i. Therefore, the number of all possible error correction options is $(bitwidth + 1)^n$ which grows exponentially with respect to n.

We introduce a heuristic algorithm searching for an n-tuple s, whose ith element is k_i, which represents a correction option that can fully recover the model accuracy while keeping the number of corrections as small as possible. It proceeds as follows: ① $s = (1, 1, ..., 1)$ is chosen and the accuracy of the model under partial correction is evaluated ② n candidates for the next s are constructed by incrementing each element in s by 1, and model accuracies are evaluated

Algorithm 1 Accuracy Recovery Algorithm

```
1:  function SEARCH($model, acc_{target}$)
2:      $bitwidth \leftarrow 8$
3:      $n \leftarrow$ Number of layers compressed
4:      for $i \leftarrow 1$ to $n$ do
5:          $s[i] \leftarrow 1$                ▷ correcting all $1^{st}$ MSB errors
6:      $acc_{curr} \leftarrow EVAL(model, v)$
7:      while $acc_{curr} < acc_{target}$ do       ▷ Loop updating v
8:          for $i \leftarrow 1$ to $n$ do
9:              $cand \leftarrow s$
10:             if $cand[i] \neq bitwidth$ then
11:                 $cand[i] \leftarrow cand[i] + 1$
12:                 $accs[i] \leftarrow EVAL(model, cand)$
13:                 $cand[i] \leftarrow cand[i] - 1$
14:         $idx \leftarrow argmax(accs)$
15:         if $acc_{curr} \leq accs[idx]$ then       ▷ Improvement
16:             $s[idx] \leftarrow s[idx] + 1$
17:             $acc_{curr} \leftarrow accs[idx]$
18:         else                                ▷ No improvement
19:             for $i \leftarrow 1$ to $n$ do
20:                 $s[i] = min(s[i] + 1, bitwidth)$
21:     return $s$
```

under each candidate ③ The candidate exhibiting the highest improvement in accuracy is chosen as the next v, if there is any non-negative improvement ④ If all candidates exhibit a deterioration of accuracy, all elements of s are incremented by 1 ⑤ Steps 2-4 of updating s are repeated until original model accuracy is fully recovered. Our heuristic algorithm has $O(n^2 \cdot (bitwidth + 1))$ complexity, which is much smaller compared to the number of all possible options.

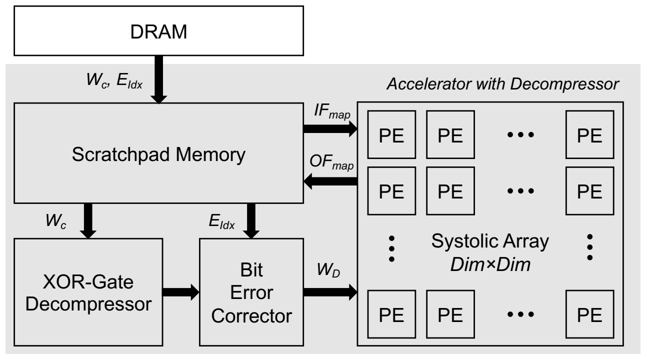


Fig. 3. Memory Access and Data-flow within the Accelerator.

IV. Decompress and Error-correct Hardware Implementation

A. Overview

We have implemented the scheme elaborated above on top of Gemmini [10], an open-source systolic array accelerator. The overall compression and error correction procedures are illustrated in Fig. 3. ① The compressed model weights are transferred from DRAM to the scratchpad memory. The error location in each layer and bit position that we have chosen to correct with the aforementioned algorithm are also loaded. ② The XOR-decompressor in the accelerator inflates the weights to the original bit-width. ③ Then, the bits that require corrections are flipped by the Bit Error Corrector. ④ Finally, the corrected weights are loaded into the systolic array. The systolic array can execute MAC operations without additional latency from correction due to the double-buffering registers in processing elements (PEs). Details of the hardware are described in the following sections.

B. Weight Decompression Logic

As the bits in the same position are grouped and compressed together, each bit of a single weight is acquired from separate decompressions. We have used an INT8 quantized model, so eight N_c sized vectors are decompressed to eight N_d sized vectors corresponding to each bit position. The relevant bits in the decompressed bit planes are combined to restore the original weight.

The first step of decompression is to load the compressed weight vectors. Fig. 4 shows compression with $N_s = 1$, where one previous vector is also used to decompress the current compressed vector. Therefore, compressed vectors of length N_c are loaded into two 8-bits shift registers (W_{c1} and W_{c2} in Fig. 4(b)). Then, they are passed through the XOR-gate array to acquire eight N_d sized vectors. Since these vectors contain bit errors, some of the errors are flipped through a compact and efficient error correction logic explained in Section IV-C.

Afterward, the corrected vectors ($W_{i1}[7:0]$) are masked with the mask vector to suppress the pruned weights. Only $\frac{1}{8}$ of the size needed to hold the decompressed weights is required to hold the mask as the mask values of a given weight are either 0 or 1 in every bit position. Finally, the eight bits corresponding to the same weight are packed to construct an INT8 value, and loaded into the weight registers in PEs. Since a single decompression generates N_d INT8 elements that can fill N_d PEs respectively, multiple rounds of decompression are needed to fill the entire systolic array. Thus, it is crucial for the weights in one phase to be reused over a sufficient number of inputs so that the weights for the next phase can be prepared and loaded in advance.

C. Error Bit Correction Logic

The decompressed weight vectors have to be corrected in order to restore the accuracy of the quantized INT8 model. After compression, each vector is decompressed and compared with the original data. The indices of the error bits in each decompressed vector identified by the algorithm are then recorded along with the compressed result. Each index is saved as $\lceil log_2(N_d) + 1 \rceil$ bits, representing its position in the N_d sized vector. This information is streamed into the correction logic to implement efficient hardware and save the area. Therefore, if there are a total of N_e errors to correct in a single decompression, N_e additional clock cycles are consumed to correct them.

Error corrections in a single decompression are conducted by the logic illustrated in Fig. 4 (b) in 3 steps. It begins with the N_d sized vectors inflated by the XOR-gate arrays loaded in the flip-flops. ① The *init* signal sets all DEMUX signals to 2. ② One $\lceil log_2(N_d) + 1 \rceil$-bit index information is assigned to E_{id} which selects and flips the error bit in that particular index. Meanwhile, DEMUX selects the flip-flop that needs to be updated, which enables the bit flipped to the same flip-flop. ③ Repeat steps 1-2 until all recorded indices have been processed. Since it takes only a single clock cycle to correct one bit, and a new index is loaded to E_{id} every cycle, a total of N_e cycles are required to correct all the selected errors.

As illustrated, the number of cycles consumed in this logic is determined by the total number of errors to fix in a particular vector. This number may vary according to not only the parameters of compression, but also the pruning rate. Under compression with $4\times$ ($N_d = 32$), and $10\times$ ($N_d = 80$) compression ratio, the average number of bits to be corrected in each decompressed vector in ResNet-18 pruned with a sparsity of 0.8 is 0.03 and 1.99.

D. Systolic Array

A systolic array is a two-dimensional square array of pipelined PEs with Dim rows and columns. In our work, we have used a systolic array of $Dim = 16$. It takes a minimum of Dim clock cycles to fill the systolic array, and a few additional cycles are required to decompress and correct the errors before loading if the values are weights. Thus, it is best to reuse a set of loaded weights over as much input data as possible, so that there is enough time for the weights to be used in the next phase to be prepared and preloaded onto the double-buffered PEs. In a convolution layer, which is of our main interest, a single kernel weight is used to conduct MAC operation with roughly every single pixel across the input image. Therefore, weight-stationary dataflow is chosen for our systolic array.

V. Evaluation

A. Compression Environment

Evaluation of effective compression ratio has been conducted on two models, ResNet-18 and VGG-16, on CIFAR100 dataset. Only convolution layers (11.2M, 14M parameters respectively) are compressed and evaluated since our error correction hardware requires a high arithmetic intensity to hide the correction latency. Model preparation and accuracy evaluation were conducted using the PyTorch framework. First, floating-point (FP) models are trained on CIFAR100. Then, the models were pruned with sparsities ranging from 70-90% and retrained to recover the consequent accuracy drop. Finally, FP weights are quantized to INT8 through post-training quantization.

The models acquired through the above steps are used as baselines to assess compression. They were compressed under several parameter configurations, with $N_c = 8$ and N_d ranging from 32 to 128 in steps of 16. $N_s = 1$ is used due to diminishing returns in the trade-off of compression error rate and time complexity. For efficient compression, each convolutions layer is divided into multiple strips, which then are decomposed to bit planes. Each bit plane is flattened and cut into a sequence of N_d-sized vectors. All sequences are then compressed in parallel using Python through multi-processing.

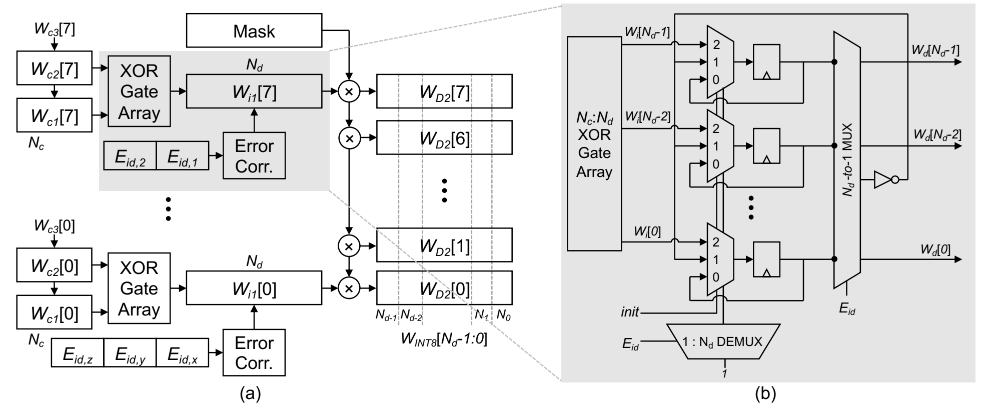


Fig. 4. (a) Parallel Decompression of INT8 Bit Planes. (b) Decompression and Error Correction Logic.

B. Accuracy Recovery Algorithm

The process of selecting errors to be corrected according to Algorithm 1 is demonstrated in Fig. 5. The search begins from correcting only the errors in MSB, then selects and adds errors to correct in an iterative manner. Thus, the accuracy of the compressed model increases with iterations, which continues until the baseline quantized model accuracy is achieved. Compressing at a higher ratio results in more errors and consequently a lower model accuracy when only the MSB errors are corrected, so more iterations of the search are required.

C. Analysis of Compression Ratio vs. Effective Model Size

Given N_d and N_c, the parameters are compressed with the ratio of N_d/N_c by the XOR-based scheme. In addition to the compressed parameters, correction data also has to be saved, increasing the effective data size to be transferred during inference. We have studied the relationship between this effective data size and the compression ratio of the scheme.

Previous literature [9] has argued that all mismatching bits can be corrected to recover the accuracy of the baseline model. They analyzed that maximal memory reduction is achieved when a compression ratio of 1/(pruning rate) is used, where the trade-off between the reduced size of compressed parameters and the increasing overhead of correction data is at best. In fact, it can be found in Fig. 6 that effective model size is the smallest under full error-correction (FEC) [9] when a compression ratio of 6 is used for both ResNet-18

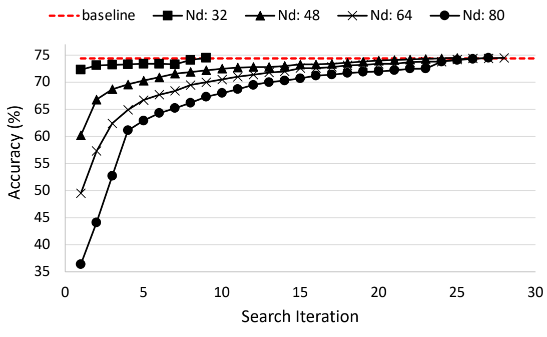


Fig. 5. Accuracy Recovering Algorithm Run on ResNet-18 ($N_c = 8$).

and VGG-16 pruned with a pruning rate of 0.8. In contrast, under partial error-correction (PEC) that we have proposed, only the selected *important* error data needs to be retained. Due to the smaller size of correction data, the effective model size is reduced significantly compared to that of FEC. This is also illustrated in the same figure, where maximal memory reduction is enjoyed at a larger compression ratio under PEC. As a result, the size of the compressed model and the correction overhead is reduced by 23.3% and 27.9% under maximum compression compared to FEC in ResNet-18 and VGG-16 respectively.

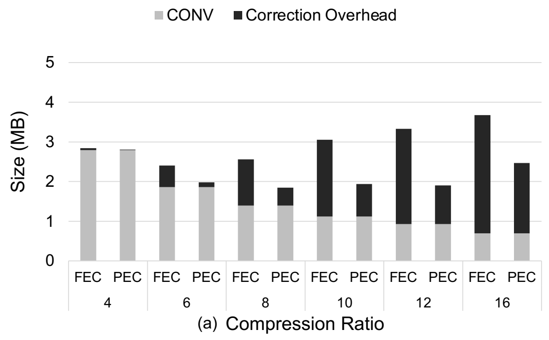
TABLE I
AREA AND POWER BREAKDOWN

Unit	Area (μm^2)	Power (mW)
Counter Controller	19,816	4.6
Load Controller	19,093	5.4
Store Controller	123,910	2.1
Systolic Array	1,561,425	164.7
SRAM	787,629	315.9
Decompression Logic	**9,824**	**2.3**

D. Hardware Performance and Overhead

In order to evaluate the area/power efficiency of our hardware implementation, DNN accelerator elements designed in Gemmini [10] are synthesized with 40-nm logic process. The main components are three controllers, SRAM, and a systolic array containing 16×16 PEs. The controllers (counter, load, and store) manage data movement between the CPU and the accelerator, while SRAM is used as scratchpad memory and the accumulator. As shown in Table I, the decompression and error correction logic that we have designed occupies only 0.39%, 0.46% of the total area and power respectively.

In terms of energy consumption, data movement through DRAM access is of prime importance as it requires several orders higher energy compared to arithmetic operations [1]. Thus, we have estimated the energy expenditure of fetching ResNet-18 and VGG-16 models from DRAM to SRAM. They are calculated using the operation-energy relationship analyzed in EIE [1], 640pJ per DRAM access to 4B data. The total amount of energy required to transfer FEC and PEC models



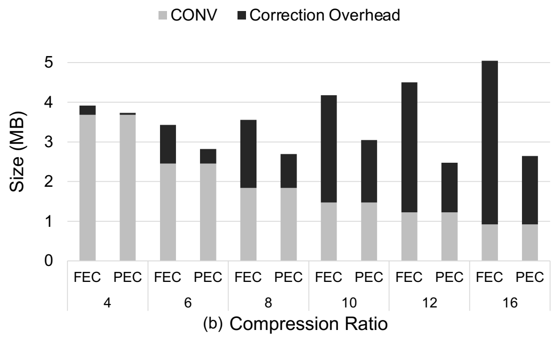


Fig. 6. Size of Convolution Layers and Correction Overhead in (a) ResNet-18 and (b) VGG-16. Both Pruned with Sparsity 0.8, Compressed with $N_c = 8$.

TABLE II
ENERGY CONSUMPTION FOR CONV TRANSFER

Energy	FEC	PEC	Saved Energy
ResNet-18	0.38 mJ	0.30 mJ	23.3 %
VGG-16	0.55 mJ	0.40 mJ	27.9 %

is described in Table II. Compared with FEC, the energy consumption of the PEC models are only 76.7% and 72.1%.

E. Coverage of Compression

Currently, the proposed partial error correction scheme and decompression hardware target convolution layers. Although convolution layers often account for the dominating portion of compute FLOPS and hence execution time in executing convolutional neural networks (CNNs), other layers and parameters can also take a substantial portion of total FLOPS and DRAM traffic. Examples include fully connected (FC) layers. Also, pruning masks can generate a significant amount of DRAM traffic, whose size is one-eighth of the original INT8 model size as it costs one mask bit per each distinct weight (1 bit/8 bits). Due to Amdahl's Law, if we include the cost of transferring pruning masks, the overall DRAM energy savings decrease from 23.3% and 27.9% to 14.7% and 12.4% for ResNet-18 and VGG-16, respectively. However, especially at a high sparsity, the pruning mask contains a lot of zeros to make it amenable to compression such as binary matrix compression method [11]. We leave the expansion of coverage in the compression target for future work.

VI. CONCLUSION

The XOR-based compression scheme is revisited and we have thoroughly examined the issue of error correction in its applications. First, we analyze the end-to-end accuracy drop of ResNet-18 and VGG-16 models due to the compression error. The results clearly show that an accuracy recovery algorithm is indeed critical. Then, we take full advantage of the different impacts on model accuracy that each bit position in each layer has, and effectively reduce the effective compressed model size while fully recovering baseline accuracy. The final model size of ResNet-18 and VGG-16 are 76.7% and 72.1% of their respective baseline under a compression ratio of 0.8. Lastly, since conventional processing units such as CPU and GPU are not optimized to handle the decompression procedure with an XOR-gate array, a compact and power/area-efficient hardware is implemented. The power consumption and area of this hardware are 0.39% and 0.46% of the vanilla Gemmini accelerator, with no additional latency for inference.

ACKNOWLEDGMENTS

This work was supported by research grants from Samsung Advanced Institute of Technology (SAIT) and Institute of Information & Communications Technology Planning & Evaluation (IITP) funded by the Korea government (MSIT) (2021-0-00105, Development of Model Compression Framework for Scalable On-device AI Computing on Edge Applications). The EDA tool was supported by IC Design Education Center (IDEC), Korea. Jae W. Lee is the corresponding author.

REFERENCES

[1] 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," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 243–254, 2016.
[2] S. Han, J. Pool, J. Tran, and W. Dally, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.
[3] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," *CoRR*, vol. abs/1902.09574, 2019.
[4] M. Zhu and S. Gupta, "To prune, or not to prune: exploring the efficacy of pruning for model compression," *arXiv preprint arXiv:1710.01878*, 2017.
[5] T. Gale, M. Zaharia, C. Young, and E. Elsen, "Sparse gpu kernels for deep learning," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–14.
[6] Y. Yang, J. S. Emer, and D. Sanchez, "Spzip: architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2021, pp. 1069–1082.
[7] D. Ahn, D. Lee, T. Kim, and J.-J. Kim, "Double viterbi: Weight encoding for high compression ratio and fast on-chip reconstruction for deep neural network," in *International Conference on Learning Representations*, 2018.
[8] S. J. Kwon, D. Lee, B. Kim, P. Kapoor, B. Park, and G.-Y. Wei, "Structured compression by weight encryption for unstructured pruning and quantization," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020, pp. 1906–1915.
[9] B. S. Park, S. J. Kwon, D. Oh, B. Kim, and D. Lee, "Encoding weights of irregular sparsity for fixed-to-fixed model compression," in *International Conference on Learning Representations*, 2022.
[10] H. Genc, S. Kim, A. Amid, A. Haj-Ali, V. Iyer, P. Prakash, J. Zhao, D. Grubb, H. Liew, H. Mao, A. Ou, C. Schmidt, S. Steffl, J. Wright, I. Stoica, J. Ragan-Kelley, K. Asanovic, B. Nikolic, and Y. S. Shao, "Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration," in *Proceedings of the 58th Annual Design Automation Conference (DAC)*, 2021.
[11] D. Lee, S. J. Kwon, B. Kim, P. Kapoor, and G. Wei, "Network pruning for low-rank binary indexing," *CoRR*, vol. abs/1905.05686, 2019.