Improving the Error Resilience of Nanoscale Memories by Huffman Data Compression for Machine Learning

Jisheng Hu, Shanshan Liu, *Senior Member, IEEE*, Pedro Reviriego, *Senior Member, IEEE*, Shengxu Wang, Xiuyuan Qi, Zhen Gao, *Senior Member, IEEE*, Fabrizio Lombardi, *Life Fellow, IEEE*

***Abstract*—Data compression with Huffman codes has been commonly employed to reduce the memory size required for emerging applications with large storage needs, such as Machine Learning (ML). However, memories at the nanoscale can suffer from errors, causing data corruption. This issue is even more severe for memories storing compressed data, because the error can propagate after decompression. This paper proposes an efficient error-resilient data compression scheme applicable to nanoscale memories by employing interleaved data storage and Huffman coding. Based on the evaluation results on protecting two compressed ML models, this scheme reduces the memory overhead by over 99% compared to conventional error protection schemes (such as parity or Hamming codes), while limiting the accuracy loss in the range of 0.26% to 3.27%.**

***Index Terms*—Nanoscale memory, error resilience, Huffman coding, machine learning.**

I. Introduction

With the rapid advancement of Artificial Intelligence (AI), the parameter size of modern Machine Learning (ML) models has considerably increased, with GPT-3, for example, it exceeds 175 billion parameters [1]. These massive data volumes pose significant challenges to model storage, especially when the models are deployed on resource-constrained platforms. Additionally, the training and inference processes of modern ML models often face the "memory wall" problem [2], i.e. the growth rate of memory bandwidth lags the computational power, so resulting in time spent accessing memory to even exceed computation. To address these challenges, a common approach is to compress the model [3], [4]; this approach not only reduces storage burden and hardware overhead, but it also mitigates bandwidth limitations and improves computational unit utilization.

There is a number of lightweight techniques to perform model compression such as neuron pruning, data quantization, mixed-precision processing and parameter compression [4]; these techniques can be employed solely or some of them can be combined to reduce the memory requirements and computation complexity. For example, the model parameters can be quantized from a higher-precision data format (e.g., float-point) to a lower-format (e.g., integer), and then compressed by using compression techniques such as Huffman codes [4]. In this case, only the compressed parameters (i.e., symbols) need to be stored in the memory; once model inference must be performed, the symbols are decompressed to recover the parameters for model computation. Since Huffman coding can achieve a high compression rate relying on fixed-to-variable length coding, it is widely used for not only ML model compression [4]-[6] but also other data-intense applications, such as image/video procession [7], [8].

Although model compression effectively reduces memory requirements, it poses another challenge to model dependability. This is relevant because nanoscale memories are susceptible to errors that corrupt stored data [9], [10]. For instance, soft errors caused by radiation effects on memory cells can randomly flip the stored values [11]. Such errors may be inherently tolerated by the original ML models, but their impact can be devastating on compressed models because a single error tends to propagate after parameter decompression. Without any protection, the memory errors can degrade the performance of model inference [12]. Therefore, in addition to safety-critical applications (in which the original models must be protected), error protection techniques are also required for compressed models, to guarantee normal operation of the ML system.

A widely adopted method for memory protection against bit-flip errors is based on the utilization of Error Correction Codes (ECCs) [13]; through appending additional parity bits to each memory word, ECCs achieve error detection or correction. However, the protection capability of ECCs is dependent of the number of parity bits. For example, an error is only detectable by introducing a single parity bit (i.e., the single parity (SP) code); while to locate and correct it in a 32-bit (64-bit) memory word, six (seven) parity bits (i.e., the Hamming code) are needed [13]. This shows that traditional ECC schemes are likely

This work is supported by Natural Science Funds of China (62474030 and 62171313), by the FUN4DATE (PID2022-136684OB-C22) and SMARTY (PCI2024-153434) projects funded by the Spanish Agencia Estatal de Investigación (AEI) 10.13039/501100011033, TUCAN6-CM (TEC-2024/COM-460), funded by CM (ORDEN 5696/2024) and by the Chips Act Joint Undertaking project SMARTY (Grant no. 101140087). Corresponding author: S. Liu (email: ssliu@uestc.edu.cn); Z. Gao (email: zgao@tju.edu.cn).

Jisheng Hu, Shanshan Liu, Shengxu Wang, Xiuyuan Qi are with the School of Information and Communication Engineering, University of Electronic Science and Technology of China, Chengdu, Sichuan 611731, China.

Pedro Reviriego is with ETSI de Telecomunicación, Universidad Politécnica de Madrid, Madrid 28040, Spain.

Zhen Gao is with the School of Electrical and Information Engineering, Tianjin University, Tianjin, China.

Fabrizio Lombardi is with Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02215, USA.

to offset the advantages of model compression in terms of memory requirements when providing a high protection capability for the models. This is especially the case for advanced systems at the nanoscale ranges which often have strict requirements on multiple hardware metrics and performance.

Therefore, this paper investigates a simple yet effective solution to improve the error-resilience of nanoscale memories with Huffman compression for ML models. It aims at introducing simple ECCs (such as the SP codes) to reduce the protection overhead while achieving high protection performance, like the Hamming codes. The principle of this scheme is to mitigate the negative impact of a memory error instead of correcting the error itself. Specifically, the impact of errors is limited within a local range by leveraging the block implementation of large-scale memories. Evaluation results show that with the proposed scheme, the ML models with memory errors can achieve a very similar accuracy to the error-free case, while the memory redundancy required for protection is negligible.

The rest of the paper is organized as follows. Section II briefly introduces ML model compression techniques with parameter quantization and Huffman compression, as well as the impact of memory errors on such models. Then, section III presents the proposed block-based interleaving Huffman scheme for improving the error-resilience of compressed models. The effectiveness of the proposed scheme is evaluated in section IV by applying it to two typical neural networks. Finally, the paper ends with conclusion and some future work in section V.

II. Model Compression by Huffman Codes

Typically, ML models trained with single-precision floating-point (Float32) data can achieve very high classification accuracy. However, such data format poses a critical challenge to parameter storage; for example, the powerful AlexNet model approximately requires 240MB memory to store its Float32 parameters. As introduced previously, a common solution to reduce the storage requirement of ML models is to perform model compression, which typically includes two steps (Fig. 1): parameter quantization (e.g., quantizing data from Float32 format to 8-bit integers (INT8)) and then data compression (e.g., by using Huffman codes). Such model compression can significantly reduce volume of parameters that need to be stored; considering again the example of AlexNet, its memory requirement approximately reduces to 38MB with model compression [4]. To achieve a further reduction, techniques such as structured pruning or sparsification that “thin out” the model can also be applied.

A. *Parameter Quantization*

Quantization of ML models is a technique that converts the weights and activations from high-precision data formats (typically Float32) to low-precision formats (e.g., INT8) using specific algorithms. Therefore, this process not only significantly reduces the model's storage requirement, but it also transforms the computations during inference into low-complexity operations, thereby decreasing the computational load and accelerating model inference.

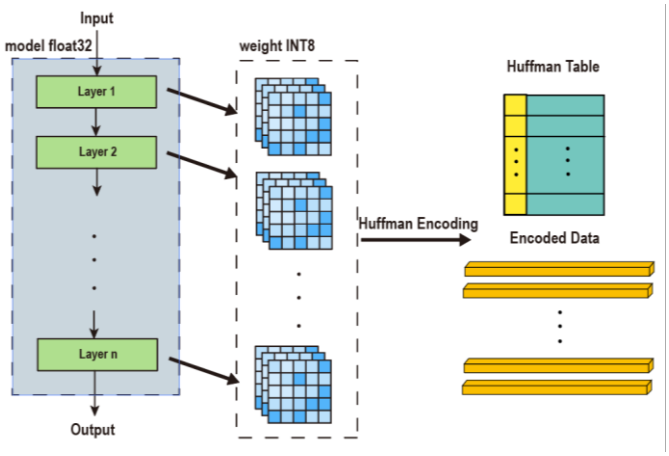

Fig. 1. The process of model compression (including parameter quantization and Huffman compression).

Among different quantization methods for ML, static quantization is the most widely used due to its benefits such as high inference throughput, low power dissipation and high flexibility [3]. Therefore, it is very attractive for implementing ML models in mobile/edge devices and embedded systems that usually require high performance and energy efficiency [14]. This paper also considers static quantization in the implementation and evaluation parts, and its implementation is briefly reviewed as follows:

- Static quantization is a post-training quantization approach. Prior to inference, the original Float32 model processes a set of calibration data - typically a small batch of representative samples - to statistically analyze the value distribution and range of each layer's inputs and outputs. This analysis determines the pre-calibrated quantization parameters, mainly including scaling factors and zero points.
- During inference, the pre-calibrated quantization parameters are applied to quantize layer inputs for model computation (Fig. 2). This eliminates the need for dynamic range calculation during runtime (as required in dynamic quantization). This pre-calibration characteristic makes static quantization more computationally efficient, though it may sacrifice some adaptability to input variations. More details for model quantization can be found in [15], due to lack of space they are not further introduced in this paper.

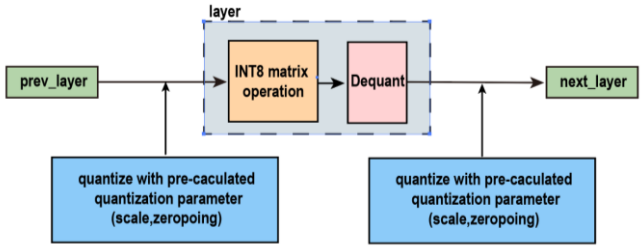

Fig. 2. Static quantization data flow.

B. *Huffman Compression*

Since quantized weights exhibit specific characteristics such as sparsity and non-uniform symbol frequency distributions, they are particularly suitable for additional compression to further reduce storage. Huffman coding is a widely used entropy encoding algorithm in data compression that enables lossless data compression, it is also often employed for ML systems [4], [5].

The basic principle of Huffman coding involves constructing a Huffman tree to assign variable-length symbols to parameters based on their frequency. This is illustrated by taking the examples as shown in Fig. 3; specifically, more frequent parameters (such as w_1 with the highest occurrence probability of 0.25) receive shorter symbols (00) while less frequent ones (w_4 or w_5) receive longer symbols (0111 or 0110). This strategy achieves a higher compression rate compared to other fixed-length compression techniques such as Tunstall codes [16].

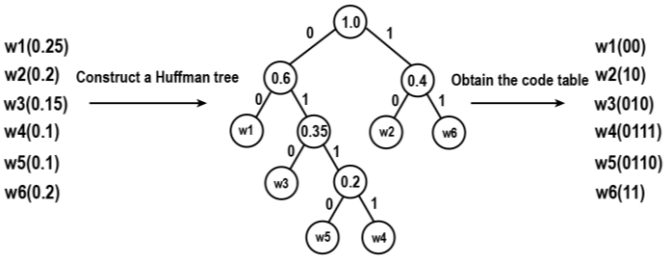

Fig. 3. Huffman Encoding Process.

For Huffman decoding, a commonly used method is to traverse the Huffman tree. Starting from the root node, read one bit from the encoded stream. If it is 0, move to the left child node of the current node; if it is 1, move to the right child node of the current node. This process continues until reaching a leaf node to obtain the decoded symbol, and then return to the root node to start decoding the next symbol. For an hardware implementation of Huffman decoding, the traditional method is a state-machine-driven serial decoding approach; the basic scheme is to parse the encoded stream bit by bit and traverse the Huffman tree [17]. Alternatively, relevant work has also proposed parallel schemes to accelerate the Huffman decoding process [18].

C. *Impact of Errors on Compressed Parameters*

When the memory stores compressed data, an error tends to have a larger impact [16]; this is especially the case for data compressed with Huffman codes. As shown in Fig. 4, if a single memory error occurs in the compressed data, it may propagate after decoding; consequently, starting from the symbol corresponding to the erroneous bit, all subsequently decoded symbols could become incorrect. Moreover, due to the construction principle of the Huffman tree and the variable-length nature of symbols, the corrupted symbol can easily become another symbol and shift the decoding process forward or backward.

Given the potential occurrence of error scenarios, it is of significant importance to improve the error-resilience of Huffman-coded model weights to ensure correct inference in the presence of memory errors. One typical solution is to employ ECCs (such as SP or Hamming codes) to detect or correct the error as introduced in section I. However, this may either have limited protection performance (with parity codes) or require a large memory redundancy (with Hamming codes). Therefore, this paper pursues a more efficient solution that performs a trade-off between the protection capability and hardware overhead, as presented and evaluated in the following sections.



Fig. 4. The impact of errors on Huffman decoding (weights corrupted by the error are marked in red).

III. Proposed Error-Resilient Huffman Scheme

To ensure that the ML model has a certain level of error resilience while maintaining the efficiency of model compression, this paper proposes a block-based interleaving Huffman scheme for compressing the model parameters. Next, the opportunities for establishing this scheme are pointed out; its detailed implementation and benefits are then presented and discussed.

Opportunities: ML models tend to have inherent tolerance to small numerical deviations. This feature is usually exploited to improve the error-resilience of ML models in both the original and compressed versions. For example, the scheme proposed in [16] focuses on protecting ML models compressed with Tunstall codes against memory errors; it allows errors with a local impact after data decompression and it only handles global errors by converting them to local errors as per a specific symbol mapping rule. In this case, although these errors are not corrected, they only corrupt a limited number of model parameters and cause a small accuracy loss; meanwhile, the required protection overhead is also very low. Although this strategy is also applicable to the Huffman case, it faces a major challenge: the identification of different symbols in the compressed stream (required for error conversion) is difficult due to their variable lengths. Therefore, if the location of different Huffman symbols can be determined, exploiting the inherent error-resilience of ML models by using strategies similar to [16], is an opportunity to efficiently protect the Huffman compressed parameters against memory errors.

However, the implementation features of memory provides an opportunity to address the difficulty of identifying Huffman

symbols. Nanoscale memories with large capacity usually consist of a number of banks or blocks; splitting the large memory array into several small regular regions can facilitate its addressing, parallel operation, and testing/yield management [19], [20]. Such regions can actually provide a scale for the entire memory data because they have the same size. This can be potentially exploited to split the Huffman symbols.

Implementation: The aforementioned opportunities in terms of both ML models and memory implementation allow the development of the block-based interleaving Huffman scheme proposed in this paper. In this scheme (Fig 5), the Huffman symbols are interleaved across different memory blocks by storing the first symbol w_1 in block1, the second symbol w_2 in block2, … the 64th symbol w_{64} in block64, and the 65th symbol w_{65} back to block1, and so on. Then, a single parity bit is embedded in each block to detect errors. Once an error occurs, the parity bit helps to identify the erroneous block, and all weights stored in that block are replaced by zeros. This avoids error propagation because the corrupted symbol is replaced by a zero instead of incorrectly decoding it as a shorter or longer symbol. Hence, a global error is converted to several discrete local errors/zeros in all parameters. This is equivalent to neuron pruning, so tending to have limited impact on the model performance.

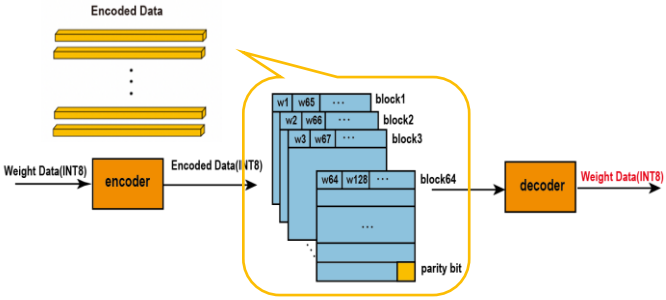

Fig. 5. The proposed block-based interleaving scheme for storing Huffman symbols (the obtained weights marked in red refers to those that can be slightly corrupted by replacing some with zeros).

In addition to the interleaving storage, the implementation of the scheme also contains parity encoding and decoding prior to data storage and data decompression, respectively. The encoding overhead can be ignored because the parameters are always written once while they are frequently read for model inference. For decoding, although it incurs in an additional latency for error detection in each inference, this overhead is smaller than existing ECC schemes such as conventional parity code or Hamming code. Note that storing o, Huffman symbols in different memory blocks has no impact on data extraction because read operations are just performed by accessing different addresses.

IV. Evaluation

This section evaluates the effectiveness of the proposed scheme. Two typical ML models with four popular datasets are considered in the evaluation; for comparison purpose, the existing parity and Hamming schemes are also evaluated. Finally, an improvement of the proposed scheme with deeper interleaving is also studied.

A. Simulation Setup

ML models and datasets: The ResNet50 (consisting of 49 convolutional layers and one fully connected layer) and LeNet (consisting of two convolutional layers and three fully connected layers) models are trained in this paper, for the CIFAR-100/CIFAR-10 [21] and MNIST [22]/Fashion-MNIST [23] datasets respectively. The models are trained using Float32 data, and then both the original model and the compressed version with static quantization of INT8 data and Huffman compression, are evaluated for inference. The parameter size and classification accuracy for each model are given in Table I; model compression offers a singinicant reduction (over 78.64%) in memory requirement, while the accuracy loss is within a limited range of 0.04% to 3.46%.

Error injection: The injection of errors are performed by simulations. Although multiple bit errors can occur in advanced memories at the nanoscale, random single-bit errors are still the most frequent type [24]. Therefore, this paper considers single-bit errors as the error model in an initial investigation and the study on multiple-bit errors is left for future work. The memory word length considered in this work is 64 bits; therefore, in the simulations, an error is injected by randomly selecting a bit of a 64-bit memory word (i.e., a random weight bit in its Huffman symbol representation as stored in the memory) and flipping its value from 0 to 1 or the other way around. Then, the accuracy of the model with the corrupted parameters is checked. Note that the number of decompressed weights may be mismatched with the number of neurons when the error causes a global impact; in this case, the weights are padded (truncated) by zeros if the number is smaller (or larger) than that in the error-free case. Finally, the error-injection process is repeated 1,000 times to obtain the average impact of the error.

B. Error-resilience and Memory Redundancy

Prior to evaluating the error-resilience of models with different protection schemes, the impact of errors on the unprotected models is evaluated. The distribution of the model accuracy in the presence of errors across 1,000 simulations is plotted in Fig. 6 and the average impact is given in Table II; a single error can either slightly degrade the model performance (i.e., having a local impact) or even make the model fail to perform classification (i.e., having a global impact), resulting in an accuracy loss of 31.36% on average.

Next, the error-resilience of models protected by existing SP and Hamming schemes as well as the proposed scheme is evaluated. The results are also shown in Table II; the memory redundancy required in each scheme is also compared in this table.

As per Table II, the model accuracy in the SP scheme is the same as the unprotected case, because it only detects the error without any manipulation on the error (intuitively the weights/symbols stored in the memory word with the detectable errors can be replaced by zeros, but it is not doable in the plain SP scheme because the number of replaced weights is not

TABLE I PERFORMANCE OF ML MODELS IN BOTH ORIGINAL AND COMPRESSED VERSIONS

Model	Dataset	Original model (Float32)		Compressed model (INT8 + Huffman)	
		Accuracy	Parameter Size	Accuracy	Parameter Size
Resnet50	CIFAR100	75.73%	79.66MB	72.27%	17.02MB
	CIFAR10	94.49%	78.96MB	93.63%	14.34MB
LeNet	MNIST	98.40%	0.235MB	98.36%	0.04MB
	Fashion-MNIST	85.40%	0.235MB	85.27%	0.04MB

TABLE II ERROR-RESILIENCE AND MEMORY REQUIREMENT IN DIFFERENT PROTECTION SCHEMES

Model	Dataset	Unprotected scheme		SP scheme		Hamming scheme		Proposed scheme	
		Model accuracy	Memory requirement	Model accuracy	Memory requirment	Model accuracy	Memory requirment	Model accuracy	Memory requirment
Resnet50	CIFAR100	46.54%	100%	46.54%	101.56%	72.27%	110.94%	69.00%	100.000045%
	CIFAR10	37.63%	100%	37.63%	101.56%	93.63%	110.94%	92.90%	100.000053%
LeNet	MNIST	77.87%	100%	77.87%	101.56%	98.36%	110.94%	98.16%	100.000190%
	Fashion-MNIST	62.42%	100%	62.42%	101.56%	85.27%	110.94%	82.78%	100.000190%

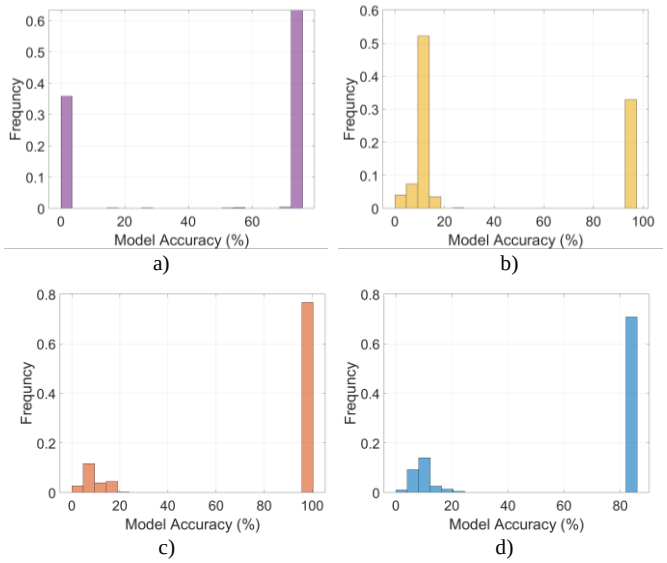

Fig. 6. The distribution of model accuracy under single memory errors across 1,000 simulations: a) ResNet50 with CIFAR-100 dataset (with error-free accuracy of 72.27%); b) ResNet50 with CIFAR-10 dataset (with error-free accuracy of 93.63%); c) LeNet with MNIST dataset (with error-free accuracy of 98.36%); d) LeNet with Fashion-MNIST dataset (with error-free accuracy of 85.27%).

known in advance). The difference from the unprotected scheme is that an error detection signal is available to the system or user; this however is at the cost of an additional parity bit stored in each memory word, i.e., 1.56% memory redundancy in total.

In the Hamming scheme, the (71, 64) code is employed for each memory word to handle single-bit errors. Since the errors can be fully corrected, the model accuracy is the same as in the error-free case (given in Table I). However, the memory requirements are increased by 10.94%; such a considerable overhead offsets the saving offered by the model compressing to some extent and thus, it is not attractive or even not applicable to ML models deployed in resource-limited platforms.

Compared to these existing methods, the proposed scheme achieves a superior tradeoff between error-resilience and memory overhead. This can also be visually observed from Fig. 7 in addition to the results given in Table II. The accuracy degradation is limited in the range of 0.26% to 0.73% for CIFAR-10 and MNIST, datasets. A slightly higher degradation

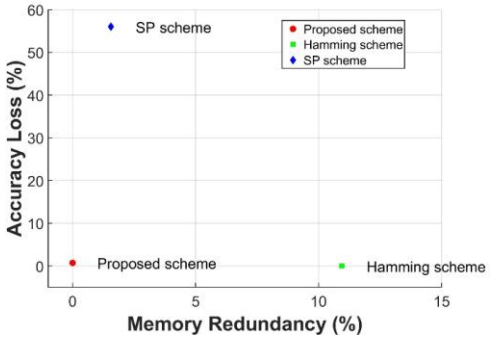

Fig. 7. An overview of different schemes when protecting the Resnet50 model with CIFAR10 (as approaching the origin the proposed scheme is the best option considering the overall performance).

of 3.27% and 2.49% is observed for CIFAR-100 and Fashion-MNIST, respecitvely; this is likely due to their lower classification performance in the error-free models, so more sensitive to the replacement of neurons by zeros. Overall, when significantly improving the error-resilience of ML models, the proposed scheme only incurs in a negligible memory redundancy (i.e., up to 0.000053%, which in practical terms is almost free). Note that when considering the memory implementation for the entire model, in addition to the Huffman compressed weights, the protection for pre-calibrated quantization parameters (i.e., scaling, zeropointing) discussed in section II.A also need to be stored. Their protection should employ Hamming codes in all schemes to guarantee data integrity; this memory overhead is not included in the results given in this section because they are the same in the different considered schemes and negligible compared to the overhead for protecting weights. Also, the encoder/decoder circuit overhead is not further evaluated (although the proposed scheme has the smallest circuits) because they are very low compared to the memory array.

C. Impact of the Depth of Interleaving

Since the number of blocks contained in large scale memories usually varies, we further evaluate the protection performance of the proposed scheme with different number of interleaved blocks from 64 to 256. The same error injection process is conducted for the case of 128 and 256 blocks; the simulation results are given in Table III.

As per the results, the average model accuracy achieved in

the proposed scheme tends to increase with a larger number of interleaved memory blocks. This is expected because when storing the same number of compressed parameters, the use of more blocks implies that fewer parameters are replaced by zeros once an error is detected, i.e., the error causes a local impact of a smaller range. Moreover, although this is at the cost of a larger number of parity bits for protecting the blocks, the increase in memory redundancy is negligible (from 0.000053% to 0.000212%). Therefore, when more blocks are available in the memory, a deeper interleaving scheme is suggested for further improving the protection performance of the proposed scheme.

TABLE III MODEL ACCURACY ACHIEVED IN THE PROPOSED SCHEME WITH DIFFERENT DEPTH OF INTERLEAVING

Model	Dataset	Model accuracy with different depth of interleaving		
		64 blocks	128 blocks	256 blocks
Resnet50	CIFAR100	69.00%	71.10%	71.11%
	CIFAR10	92.90%	93.04%	93.39%
LeNet	MNIST	98.16%	98.27%	98.34%
	Fashion-MNIST	82.78%	83.39%	83.71%

V. CONCLUSION AND FUTURE WORK

This paper has proposed the so-called block-based interleaving Huffman scheme to improve error-resilience of compressed ML models at negligible memory redundancy. In the presence of single-bit memory errors (the most frequent pattern in nanoscale memories), this scheme reduces error propagation across model parameters after Huffman decompression by interleaving the Huffman symbols in different memory blocks. With a parity bit, the block encountering an error is detected and its stored interleaved Huffman symbols are replaced by zeros. In this case, the impact of errors is limited within a local range, so the model can still achieve an accuracy similar to the error-free case; meanwhile, the required memory redundancy is negligible. These advantages make the proposed scheme superior to existing ECC schemes when comprehensively considering the protection performance and hardware overhead.

Future work will investigate efficient protection schemes against multiple-bit memory errors (such as double/triple-adjacent bit errors) which also frequently occur in advanced memories. In addition, the design of protection mechanisms can alternatively be pursued by employing a deeper integration of Huffman compressing and error protection, such as embedding error detection features into symbol mapping when constructing the Huffman tree, to further improve the overall performance.

REFERENCES

[1] T. Brown, B. Mann, N. Ryder, et al., "Language models are few-shot learners", in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, pp. 1877–1901, 2020.

[2] A. Gholami, Z. Yao, S. Kim, et al, "AI and memory wall," *IEEE Micro*, vol. 44, no. 3, pp. 33–39, May 2024.

[3] Y. Gong, L. Liu, M. Yang, et al, "Compressing deep convolutional networks using vector quantization", *arXiv preprint*, arXiv:1412.6115, 2014.

[4] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural networks with pruning, trained quantization and Huffman coding", *arXiv preprint*, arXiv:1510.00149, 2015.

[5] R. R. Gajjala, S. Banchhor, A. M. Abdelmoniem, et al, "Huffman coding based encoding techniques for fast distributed deep learning", in *Proceddings of 1st Workshop on Distributed Machine Learning*, pp. 21–27, 2020.

[6] C. Pal, S. Pankaj, W. Akram, et al, "Modified Huffman based compression methodology for deep neural network implementation on resource constrained mobile platforms", in *IEEE International Symposium on Circuits and Systems (ISCAS)*, pp. 1-5, 2018.

[7] C. Karthikeyan and C. Palanisamy, "An efficient image compression method by using optimized discrete wavelet transform and Huffman encoder", *Journal of Computational and Theoretical Nanoscience*, vol. 15, no. 1, pp. 289–298, 2018.

[8] T. Bernatin and G. Sundari, "Video compression based on hybrid transform and quantization with Huffman coding for video codec", in *Proc. 2014 Int. Conf. on Control, Instrumentation, Communication and Computational Technologies (ICCICCT)*, pp. 452–456, 2014.

[9] C. M. Jeffery, A. Basagalar, and R. J. O. Figueiredo, "Dynamic sparing and error correction techniques for fault tolerance in nanoscale memory structures", in *IEEE Conference on Nanotechnology*, pp. 1-3, 2004.

[10] A. K. Shukla, H. Nehete, P. Kaushik, et al, "Machine learning assisted soft error variation analysis of radiation hardened MRAM circuit", in *IEEE Conference on Nanotechnology*, pp. 1-6, 2023.

[11] M. Juliato and C. Gebotys, "A quantitative analysis of a novel SEU-resistant SHA-2 and HMAC architecture for space missions security", *IEEE Trans. on Aerospace and Electronic Systems*, vol. 49, no. 3, pp. 1536–1554, 2013.

[12] S. Liu, Z. Wang, Z. Gao, et al, "Are emereging machine learning models dependable at the nanoscales?", *IEEE Nanotechnology Magazine*, vol. 18, no 6, pp. 33-42, 2024.

[13] S. Lin, D. J. Costello, Error control coding, Scarborough: Prentice hall, 2001.

[14] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for AI-enabled IoT devices: A review", *Sensors*, vol. 20, no. 9, Art. no. 2533, 2020.

[15] A. Gholami, S. Kim, Z. Dong, et al, "A survey of quantization methods for efficient neural network inference", in *Low-Power Computer Vision*, pp. 291–326, Chapman and Hall/CRC, 2022.

[16] S. Liu, P. Reviriego, A. Ullah, et al, "Error-resilient data compression with Tunstall codes", *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 70, no. 5, pp. 1963–1975, 2023.

[17] P. N. A. A. M. Sabri, A. Abas, et al, "Analysis of lossless compression in Huffman coding and Lempel-Ziv-Welch (LZW)", *Journal of Advanced Research in Computing and Applications*, vol. 39, no. 1, pp. 1–10, 2025.

[18] Z. Li, Z. Wang, L. Xu, et al., "RRAM-DNN: An RRAM and model-compression empowered all-weights-on-chip DNN accelerator", *IEEE Journal of Solid-State Circuits*, vol. 56, no. 4, pp. 1105–1115, 2021

[19] Y. H. Son, O. Seongil, Y. Ro, et al, "Reducing memory access latency with asymmetric DRAM bank organizations", in *Proceedings of the 40th annual international symposium on computer architecture*, pp. 380-391, 2013.

[20] Y. Zhou, K. M. Al-Hawaj, Z. Zhang, "A new approach to automatic memory banking using trace-based address mining", in *Proceedings of the 2017 ACM/SIGDA International Symposium on Fiedl-Programmable Gate Arrays*, pp. 179-188, 2017.

[21] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images", Citeseer, 2009.

[22] Y. LeCun, L. Bottou, Y. Bengio et al., "Gradient-based learning applied to document recognition", *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278-2324, 1998.

[23] H. Xiao, K. Rasul and R. Vollgraf, "Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms", *arXiv preprint*, arXiv:1708.07747, 2017.

[24] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies", *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305-316, 2005.