2023 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 979-8-3503-1199-0/23/$31.00 ©2023 IEEE | DOI: 10.1109/IPDPSW59300.2023.00061

A High-dimensional Algorithm-Based Fault Tolerance Scheme

Xiang Fu, Hao Tang, Huimin Liao, Xin Huang, Wubiao Xu, Shiman Meng, Weiping Zhang
Nanchang Hangkong University
{*fuxiang, htang, hliao, xhuang, wxu, smeng, wzhang*}*@nchu.edu.cn*

Luanzheng Guo
Pacific Northwest National Laboratory
lenny.guo@pnnl.gov

Kento Sato
R-CCS, RIKEN
kento.sato@riken.jp

***Abstract*—Tensor Algebra is a powerful tool for carrying out high-order data analytics in scientific applications, such as finite element analysis, N-body simulation, and quantum chemistry. Many of these applications are critical in terms of correctness and safety. Since these applications often run on High Performance Computing (HPC) systems, which are susceptible to soft errors caused by cosmic rays, unstable voltage, etc., we must ensure that the execution of these applications is reliable and resilient, and the execution outcome is highly trustworthy. However, traditional fault tolerance methods like error-correcting codes cannot protect computations. Checkpointing and redundancy techniques like triple modular redundancy (TMR) suffer from high-performance overhead, while existing algorithm-based fault tolerance (ABFT) approaches focus only on 2D linear algebra computations that are inefficient for tensor algebra computations. We understand that high-level tensor algebra computations can be decomposed into 2D linear algebra computations to be protected by existing ABFT methods, but this often introduces unacceptable performance overhead. Hence, for the first time, we propose a collection of different ABFT algorithms for addressing three fundamental tensor algebra operations. We make the best use of the algorithmic semantics of these tensor algebra computations to achieve better performance.**

I. Introduction

Tensor algebra has been extensively used in various science and engineering applications, such as quantum chemistry, physics simulationmachine learning,and high-dimensional data analytics. The resilience of these applications becomes increasingly critical as they often run on cyber infrastructures, such as HPC, and safety-critical scenarios, such as aerospace and autonomous vehicles. Unfortunately, an increasing number of soft errors are observed in those systems, as they grow in computational power while calling for lower energy consumption. Soft errors, also known as bit-flip errors, often caused by high-energy particle strikes, unstable voltage, and aging devices [1], can lead to Silent Data Corruption (SDC). We learn from recent studies that soft errors are unavoidable in today's large-scale HPC systems [2], [3], [4], [5]. However, even a single bitflip error can affect the correctness of the application's execution outcome.

There have been various fault tolerance solutions aiming to protect tensor algebra computations from soft errors. Traditional hardware ECC approaches protect memory architectures from single-element and multi-element soft errors effectively. However, ECC approaches are very expensive [6]. Instruction duplication performs the same operation multiple times to detect and correct errors. However, instruction duplication introduces high performance overhead that can exceed 100%, which is not acceptable. Algorithm-Based Fault Tolerance (ABFT) [7], also known as software ECC, is famous for its flexibility and effectiveness in protecting matrix multiplication computations. A recent research work FT-CNN [8] extends ABFT to convolution computations by flattening them into 2D matrix multiplication operations. However, to the best of our knowledge, there isn't a high-dimensional algorithm-based fault tolerance scheme, especially for high-dimensional tensor algebra computations. To fill the gap, we propose a foundation high-dimensional ABFT algorithm for high-dimensional tensor algebra and extend the algorithm to three high-dimensional tensor algebra kernels, including Tensor Element-Wise Operation(TEW), Tensor Scalar Operation (TS), and Matricized Tensor Times Khatri-Rao Product (MTTKRP) [9]. The foundation ABFT algorithm includes two parts. First, adding checksums to each dimension of the two input high-dimensional tensors that participate in the kernel computation. Second, verify the result correctness on the output tensor by calculating the new checksums and comparing the new checksums with the counterpart by kernel computation after the kernel computation.

For TEW, we apply the foundation ABFT directly. To extend the foundation ABFT to the other two tensor algebra kernels, for TS, we skip checksum on the input scalar; for MTTKRP, we enforce checksums on the Khatri-Rao product with two matrices and the multiplication between the matricied tensor and the output of the Khatri-Rao product. The proposed tensor algebra-specific high-dimensional ABFT algorithm is able to detect and localize all single-element errors and certain types of multi-element errors, depending on the distribution and number of the corrupted elements.

Furthermore, the new high-dimensional ABFT algorithm can correct all soft errors localized in specific locations. Finally, we manage to parallelize the proposed high-dimensional ABFT algorithm with OpenMP, with which the performance of the

979-8-3503-1199-0/23/$31.00 ©2023 IEEE
DOI 10.1109/IPDPSW59300.2023.00061

high-dimensional ABFT algorithm is improved by up to 16.32 times with 20 threads.

In summary, the contributions of this paper are

1) The paper proposes a specialized high-dimensional ABFT algorithm for tensor algebra that can handle high-dimensional data and is customized for three popular tensor algebraic kernels. The algorithm can detect, localize, and correct both single and multiple element errors;
2) We systematically investigate the distribution of localized multi-element errors, and identify and differentiate error patterns of localized errors that can or cannot be correctable within our high-dimensional ABFT framework;
3) The proposed high-dimensional ABFT algorithm underwent evaluation using high-dimensional tensors from the FROSTT database [10]. The evaluation results show its effectiveness and efficiency, with an overhead average of 6.27% for TEW and TS, and an average of 72.95% for MTTKRP.

II. Background

A. Fault Model

The work focuses on soft errors [11], [12], [13], also known as transient faults, that impact the application-visible state, such as machine registers and memory. Soft errors can be caused by various factors, such as alpha particle decay, cosmic rays, thermal neutrons, and random noise. Like similar previous studies [14], we consider soft errors in logic circuits and bit triggers in memory. The paper does not consider errors that are detected and potentially corrected in hardware through techniques such as memory scrubbing, ECC, or other methods. Specifically, the paper examines single-element and multi-element soft errors in high-dimensional tensor algebra operations.

Single-element error. One element of the output tensor is corrupted by single-bit or multi-bit errors. The corruption can be caused by either single-bit or multi-bit errors.

Multi-element error. More than one element of the output tensor is corrupted by errors, which can be single-bit or multi-bit errors. For discussions on more complicated cases, see the discussion on 'localization' in Section III-B.

ABFT Model. Algorithm-Based Fault Tolerance detects, localizes and corrects bitflip errors in algorithms at the application level. In common practice of ABFT, all bitflip errors are detectable, except for very rare cases, where two or multiple corruptions result in a zero sum, and the checksum is regarded as correct. However, even when corruptions are detected, not all bitflip errors are localizable. **Detection** means to check if there is any data corruption while **localization** means to find out exactly where the corruption is. If the error is deterministically localized, then the localized error is likely to be corrected. **Correction** means to correct the localized error.

B. High-Dimensional Tensor Algebra Kernels

We consider three commonly used high-dimensional tensor algebra kernels. They are binary operations [9]. We describe them as follows.

- Tensor Element-Wise (TEW): Two tensors with the same shape perform algebraic operations between elements of the same coordinate.
- Tensor Scalar (TS): A tensor and a scalar do algebraic operations.
- Matricized Tensor Times Khatri-Rao Product (MTTKRP): A matricized tensor times the Khatri-Rao product of two matrices. The Khatri-Rao product performs a column-wise Kronecker product between two matrices with the same number of columns. In contrast, the Kronecker product multiplies each element of the left matrix with the right matrix to produce a larger matrix.

III. High-Dimensional ABFT Algorithms for Tensor Computations

There are two options for our high-dimensional ABFT algorithm design. First, like the high-dimensional ABFT algorithm proposed in FT-CNN, transform the high-dimensional tensor into a matrix, and then perform traditional 2D ABFT on matrix, and finally transform it back to a high-dimensional tensor once the computation is finished. Second, add checksums directly to the two input high-dimensional tensors to each of their dimensions. For the first option, we must understand the performance overhead of *matricization*, i.e., transformation between high-dimensional tensors and 2D matrices.

A. Performance Characterization of Matricization

We aim to understand the cost of adding matricization to high-dimensional tensor algebra kernels. Matricization means that a tensor is reshaped into a matrix. For tensor $A \in \mathbb{R}^{I_1 \times \cdots \times I_n \times \cdots \times I_N}$, its matricized tensor along with mode-n is $A(n) \in \mathbb{R}^{I_1 \cdots I_{n-1} I_{n+1} \cdots I_N \times I_n}$. Because MTTKRP has internal processes for matricization, therefore, we only compare the performance of TEW and TS tensor algebra kernels on representative high-dimensional tensors with and without matricization. For TEW and TS tensor algebra kernel, we calculate normalized total flop count of the original high-dimensional tensor algebra kernel by its counterpart with matricization. We use three representative real-world tensors, Nell-2, Chicago_crime, and Patents, from the FROSTT Tensor Collection [10]. We describe the three tensors in Table I. The result is shown in Figure 1, where the normalized FLOP count=$\frac{matricization \quad tensor \quad operation}{original \quad tensor \quad operation}$. As shown, the throughput without matricization is on average 2.39 times higher than the throughput with matricization. In consequence, we take the second option that directly adds checksums to high-dimensional tensors.

B. High-dimensional ABFT algorithm

We design a foundation ABFT scheme for high-dimensional tensor algebra which includes four parts: building checksum, detection, localization, and correction. We then tailor the

TABLE I: The tensors used in our evaluation.

Benchmarks	Description	Size
Chicago-crime (CC)	Crime reports in the city of Chicago, ranging from January 1st, 2001 to December 11th, 2017. Non-zeros are counts and modes provide information such as time, location, and type of crime	6,186 x 24 x 77 x 32
VAST Mini-Challenge (VAST)	This tensor represents events ('check-in', 'movement') of attendees at a theme park over one weekend. Hidden within the data are various malicious events.	165,427 x 11,374 x 2 x 100 x 89
Uber Pickups (Uber)	Six months of Uber pickup data in New York City, provided by fivethirtyeight after a Freedom of Information request	183 x 24 x 1,140 x 1,717
NELL-2	NELL-1 is a snapshot of the Never Ending Language Learner knowledge base, part of the Read the Web project at Carnegie Mellon University	12,092 x 9,184 x 28,818
NIPS Publications (NIPS)	Papers published in NIPS from 1987 to 2003, collected by Globerson et al. The modes represent paper-author-word-year, and the values are counts of words.	2,482 x 2,862 x 14,036 x 17
Patents	Pairwise co-occurence of terms within a 7 word window in the US utility patents on a per-year basis. The date associated with each patent is the year of the priority date of the patent.	46 x 239,172 x 239,172

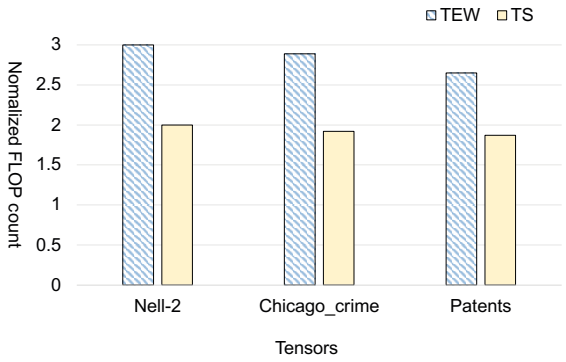


Fig. 1: Performance without matricization as normalized to that with matricization.

foundation ABFT scheme to three high-dimensional tensor algebra kernels with distinct modifications.

Building Checksum. We first build checksums to the input tensor. Figure 2 shows how we apply checksums to a 3D tensor of $2 \times 2 \times 2$, in which the tensor is in light blue and the checksums are in red. As it shows, we calculate the sum of each row, column, and tube of the tensor as the checksums that will be used for data integrity check.

Detection. Once the tensor operation (e.g., TS) is completed, the detection opts in to detect data corruptions at the output tensor. We describe the detection algorithm in lines 1 to 5 of Algorithm 1. The algorithm takes a tensor as input and outputs the corrupted rows. It performs a summation at each row in parallel within the tensor and check if the summation is equal to the checksum at the row or not; if yes, there is no error; otherwise, the corrupted row is detected and recorded. Note that in very rare cases the detection can be erroneous, where two or multiple corruptions result in a zero sum but the checksum is matched and regarded as correct, which leads to a false negative.

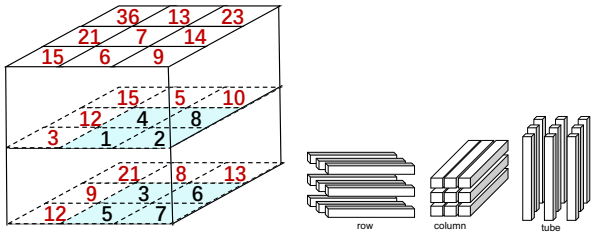
Localization. To localize the corrupted element, given a

Algorithm 1: detection, localization, and correction algorithm for ABFT

Input: $C\_checksum$, a checksum-enabled 3D tensor, which can be either the input or output tensor.
Output: err_rows, the row detected from $C\_checksum$.
// Detecting corruption at every row of $C\_checksum$.
1 **Function** detect($C\_checksum$):
2 **for** *each row of* $C\_checksum$ **do in parallel**
  // if the sum of row is not equal to its checksum.
3 **if** $SUM(row) \neq row.checksum$ **then**
4 $err\_rows.append(row)$;
5 return err_rows;
Input: $C\_checksum$ and err_rows
Output: err_pos, the position of corrupted elements in err_rows
// Localizing corrupted elements from corrupted rows.
6 **Function** localize($C\_checksum$, err_rows):
7 **for** *each row of* err_rows **do in parallel**
8 **for** *each* $elem \in row$ **do**
  // Checking the col where the element is
9 **if** $SUM(elem.col) \neq col.checksum$ **then**
  // Appending position of corrupted elements to the output.
10 $err\_pos.append(row, pos)$;
11 return err_pos;
Input: $C\_checksum$, and err_pos
Output: Corrected $C\_checksum$
// Correcting detected errors from $C\_checksum$
12 **Function** correct($C\_checksum$, err_pos):
13 **for** *each* $pos \in err\_pos$ **do in parallel**
14 flag=0;
15 **if** *pos.row or pos.col or pos.tube don't exist other error* **then**
  // Correcting the corrupted element
  // vector is row or column or tube
16 $C\_checksum(pos) = vector.checksum -$ the other elements;
17 flag=1;
18 $err\_pos.remove(pos)$;
19 **if** *flag==0* **then**
  // Errors located in the pos position cannot be corrected
20 redo the computation of tensor;
21 break;
22 **if** $err\_pos.empty$ **then**
  // A empty err_pos means that all errors were successfully corrected
23 return $C\_checksum$;

corrupted row, we play detection again to each element within the row on one of the other directions (e.g., column). The goal is to check if a corruption is detected on the other direction, such as 'column', given an element of the row. If yes, the element in question is deemed corrupted and recorded; otherwise, the element is uncorrupted. Note that we don't consider very rare cases when the two corruptions in the column generates a zero sum, and the checksum is matched. We describe the localization algorithm in lines 6 to 11 of Algorithm 1. The input to this algorithm are the corrupted rows provided by the detection algorithm and the corrupted tensor under consideration, and the output is the set of positions of corrupted elements in the row.

Note that when the summation at the other direction is not equal to its checksum, there are two possible cases: 1) the element in question is corrupted, meaning that we localized the corruption correctly; 2) the element in question

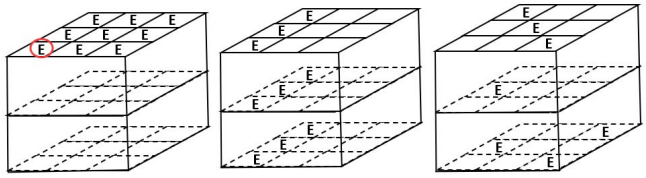


(a) (b)

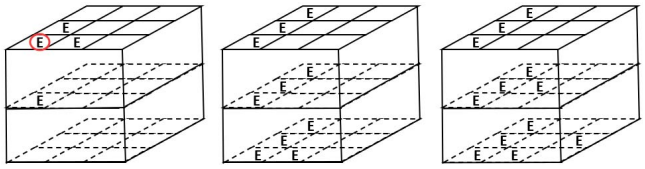
Fig. 2: The checksum tensor.

is correct while the neighbor element at the other direction is corrupted, resulting in a false positive localization. However, for both cases, we count the element in question as corrupted elements and record their positions, from which we generate a distribution of localized errors. We show examples of such distribution of localized errors in Figure 3, which uses a 3D tensor to showcase different data corruption patterns. The 3D tensor is illustrated by three layers each with nine elements, and the letter 'E' indicates that an error is present on the particular element. The distribution of localized errors will help determine whether the localized corrupted elements can be corrected or not. For cases that contain false positive localization, it is often that localized errors cannot be corrected in such cases. For example, the error circled in red in Figure 3(b) has neighbor errors in all its three directions (row, column and tube), for which we cannot determine whether the error is true or a false positive. Suppose that the error circled in red is actually uncorrupted, the localization algorithm still recognizes it as corrupted, which is incorrect. Figure 3(b) describes the error patterns of localized errors that are **non-deterministic**, meaning that the localized errors could be false positives. Figure 3(a) describes the cases when the error patterns of localized errors that are **deterministic**, meaning all localized errors are true errors. If the distribution of localized errors falls in the deterministic error patterns, the localized errors can be corrected successfully in this case. Otherwise, if it's non-deterministic, the localized errors are unlikely to be corrected.

Correction. To correct the corrupted elements, we check the row, column, and tube where the corrupted element is located, and use the one where the corrupted element is the only corruption for correction. To correct the corruption, we simply subtract the other elements except the corrupted element from the checksum. For instance, suppose that elements '1', '4', and '8' in Figure 2(a) are corrupted. In this case, we can utilize the checksum '7' of the tube '3-4-7' and subtract '3' from it to obtain the accurate value for the position of element '4'. Note that if we cannot find any of the row, column, and tube where the corrupted element is the only corruption, it suggests that the error pattern of the localized errors is non-deterministic, which are not correctable. For non-deterministic cases, we recommend to rerun the tensor operation for error



(a) The localized error can be corrected in these cases.



(b) The localized error cannot be corrected in these cases.

Fig. 3: Example patterns of localized multi-element errors. Note that there is no checksum included.

correction. The correction algorithm is shown in lines 12 to 23 in Algorithm 1. The input to this 'correct' function are the tensor in question and the set of corrupted positions. The output is the corrected tensor.

Note that we use OpenMP parallelism to detect and process corrupted rows and correct corrupted positions concurrently.

Adaptation of the ABFT Scheme to the Three Kernels. All algorithms operate on input dense tensors. For TEW, it is necessary to perform a checksum operation on both input tensors. For TS, only the input tensor requires a checksum operation. For MTTKRP, checksums must first be built for the two matrices involved in the Khatri-Rao product, followed by building a checksum for the matricized tensor and the Khatri-Rao product of matrices.

IV. Evaluation

Platforms: All experiments are run on a Linux server consisting of 32 GB DDR4 memory, two Intel Xeon E5-2678 V3 CPUs. The program are compiled with gcc-8.3 and OpenMP-4.5 with 20 OpenMP threads.

We use the assignment statement to inject a single error by changing the value of the elements of result_checksum_tensor after the result_checksum_tensor is generated. When error injection is absent, the highest runtime overhead for TEW and TS is 6.63%, while the lowest is 5.15%. For MTTKRP, the highest runtime overhead is 73.86% and the lowest is 69.17%. With error injection, the highest runtime overhead for TEW and TS is 7.04%, and the lowest is 5.28%. MTTKRP has a highest runtime overhead of 75.11% and the lowest of 69.24%.

V. Related Work

Algorithm-Based Fault Tolerance. There has been a variety of ABFT schemes developed with different focuses. For matrix operations, Chen et al. [14] developed an ABFT technique to detect (and sometimes locate and correct) soft errors. Wu et al. [15] implemented an ABFT scheme for high-performance

TABLE II: Efficiency study on 6 benchmarks with TEW/TS Operation.

Benchmarks	Execution time(No ABFT)(s)	Execution time(with ABFT)(s)	Execution time(with ABFT with fault)(s)	Overhead (with ABFT)(%)	Overhead(with ABFT with fault)(%)	Overhead (matricization) (%)
CC	4.66/3.75	4.96/3.99	4.98/4.01	6.29/5.25	6.71/5.41	7.18/5.76
VAST	47.98/38.57	51.06/40.99	51.22/41.14	6.42/5.15	6.76/5.40	7.26/5.75
Uber	109.61/88.12	116.23/93.65	116.68/94.22	6.04/5.17	6.45/5.28	7.01/5.71
NELL-2	40804.12/ 32803.31/	43432.72/ 34916.83/	43585.74/ 35070.68/	6.44/6.04	6.82/6.26	7.38/6.86
NIPS	21610.95/ 17373.51/	22908.26/ 18514.95/	23132.36/ 18538.23/	6.01/5.59	7.04/5.78	7.72/6.23
Patents	33549.70/ 26971.33/	35773.04/ 28665.94/	35896.17/ 28841.79/	6.63/5.65	6.99/5.82	7.68/6.30

TABLE III: Efficiency study on 6 benchmarks with MTTKRP operation.

Benchmarks	Execution time(No ABFT)(s)	Execution time(with ABFT)((s)	Execution time(with ABFT with fault) (s)	Overhead (with ABFT) (%)	Overhead (with ABFT with fault)(%)
CC	91.73	156.93	158.79	71.08	73.11
VAST	256.32	433.62	433.79	69.17	69.24
Uber	609.36	1036.52	1053.58	70.10	72.90
NELL-2	58447.85	101617.43	102342.18	73.86	75.10
NIPS	60890.54	105242.44	106625.42	73.63	75.11
Patents	54016.14	914385.21	930265.96	69.28	72.22

LINPACK (HPL) benchmarks to demonstrate the feasibility in large scale high-performance benchmark. Li et al. [16] proposed a soft error detection method based on the Generalized Matrix Multiplication Method (GEMM) for the low precision quantization operator in DLRM. Zhao et al. [8] further extended ABFT to convolutional computations. However, all these ABFT approaches are designed to detect and correct soft errors in 2D linear algebra, which are limited and cannot be applied to tensor algebra computations. To address this issue, we propose a collection of specific, low-cost ABFT algorithms for tensor algebra computations.

VI. Conclusion

In response to the low efficiency, limited scope of application, and poor error correction ability of traditional fault-tolerant mechanisms, we have developed an algorithm-based fault tolerance scheme for high-dimensional tensor algebraic operations. This scheme is able to detect, localize, and correct soft errors effectively and efficiently, without the need to transform high-dimensional algebraic computations into 2D linear algebra. Additionally, we have systematically studied the error distribution of localized errors, identifying and differentiating error patterns that can or cannot be corrected theoretically. Moving forward, we plan to investigate fault tolerance techniques for high-dimensional sparse tensors using ABFT, and explore the possibility of integrating ABFT algorithms into the compilation process.

References

[1] P. Du, A. Bouteiller, G. Bosilca, T. Herault, and J. Dongarra, "Algorithm-based fault tolerance for dense matrix factorizations," *Acm sigplan notices*, vol. 47, no. 8, pp. 225–234, 2012.

[2] L. Guo, D. Li, I. Laguna, and M. Schulz, "Fliptracker: Understanding natural error resilience in hpc applications," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 94–107.

[3] L. Guo, G. Georgakoudis, K. Parasyris, I. Laguna, and D. Li, "Match: An mpi fault tolerance benchmark suite," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2020, pp. 60–71.

[4] L. Guo, D. Li, and I. Laguna, "Paris: Predicting application resilience using machine learning," *Journal of Parallel and Distributed Computing*, vol. 152, pp. 111–124, 2021.

[5] L. Guo and D. Li, "Moard: Modeling application resilience to transient faults on data objects," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 878–889.

[6] D. Li, Z. Chen, P. Wu, and J. S. Vetter, "Rethinking algorithm-based fault tolerance with a cooperative software-hardware approach," in *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2013, pp. 1–12.

[7] Z. Chen, "Online-ABFT: An Online ABFT Scheme for Soft Error Detection in Iterative Methods," *PPoPP*, 2013.

[8] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Ft-cnn: Algorithm-based fault tolerance for convolutional neural networks," *IEEE Transactions on Parallel and Distributed Systems*, 2020.

[9] J. Li, Y. Ma, X. Wu, A. Li, and K. Barker, "Pasta: a parallel sparse tensor algorithm benchmark suite," *CCF Transactions on High Performance Computing*, vol. 1, no. 2, pp. 111–130, 2019.

[10] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. (2017) FROSTT: The formidable repository of open sparse tensors and tools. [Online]. Available: http://frostt.io/

[11] S. S. Mukherjee, J. Emer, and S. K. Reinhardt, "The soft error problem: An architectural perspective," in *11th International Symposium on High-Performance Computer Architecture*. IEEE, 2005, pp. 243–247.

[12] Z. Li, H. Menon, K. Mohror, S. Liu, L. Guo, P.-T. Bremer, and V. Pascucci, "A visual comparison of silent error propagation," *IEEE Transactions on Visualization and Computer Graphics*, 2022.

[13] L. Guo, J. Liang, and D. Li, "Understanding ineffectiveness of the applicationlevel fault injection," in *Poster in ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis. SC*, 2016.

[14] J. Chen, X. Liang, and Z. Chen, "Online algorithm-based fault tolerance for cholesky decomposition on heterogeneous systems with gpus," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2016, pp. 993–1002.

[15] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, J. Chen, and Z. Chen, "Towards practical algorithm based fault tolerance in dense linear algebra," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 31–42.

[16] S. Li, J. Huang, P. T. P. Tang, D. Khudia, J. Park, H. D. Dixit, and Z. Chen, "Efficient soft-error detection for low-precision deep learning recommendation models," *arXiv preprint arXiv:2103.00130*, 2021.