Efficient Soft-Error Detection for Low-precision Deep Learning Recommendation Models

Sihuan Li*, Jianyu Huang*, Ping Tak Peter Tang*, Daya Khudia‡, Jongsoo Park*,
Harish Dattatraya Dixit*, and Zizhong Chen†
* Meta Platforms, Inc.
† University of California, Riverside, CA, USA
‡ MosaicML, Inc.
{sihuan, jianyuhuang, ptpt, jongsoo, hdd}@meta.com, daya@mosaicml.com, chen@cs.ucr.edu

***Abstract*—Soft error, namely silent corruption of signal or datum in a computer system, cannot be caverlierly ignored as compute and communication density grow exponentially. Soft error detection has been studied in the context of enterprise computing, high-performance computing and more recently in convolutional neural networks related to autonomous driving.**

Deep learning recommendation systems (DLRMs) have by now become ubiquitous and serve billions of users per day. Nevertheless, DLRM-specific soft error detection methods are hitherto missing. To fill the gap, this paper presents the first set of soft-error detection methods for low-precision quantized-arithmetic operators in DLRM including general matrix multiplication (GEMM) and EmbeddingBag. A practical method must detect error and do so with low overhead lest reduced inference speed degrades user experience. Exploiting the characteristics of both quantized arithmetic and the operators, we achieved more than 95% detection accuracy for GEMM with an overhead below 20%. For EmbeddingBag, we achieved 99% effectiveness in significant-bit-flips detection with less than 10% of false positives, while keeping overhead below 26%.

I. Introduction

Hardware faults can be separated into two categories: fail-stop and fail-continue. Fail-stop faults crash the executing process, thus detectable by the operating system, and can be handled by well-studied checkpoint-and-restart techniques [1], [2]. In contrast, fail-continue faults silently corrupt the results of an execution process without interrupting it. The induced errors are usually called soft errors [3] or silent data corruptions and are the focus of this paper.

Soft error is much more prevalent than one may realize: even experienced practitioners grossly underestimate their frequency of occurrences [4], [5]. The supercomputer Jaguar, for example, suffers a double-bit memory error once every 24 hours [6]. For another example, a recent research study taking more than 18 months [5] has confirmed the large-scale infrastructure at Facebook is experiencing silent data corruptions due to device characteristics inside hundreds of Central Processing Units (CPUs). The situation is only worsening: not only cosmic radiation triggers soft error but simple down-to-earth factors such as temperature and power consumption can also be the culprit [7]. Further exacerbating the situation is the rapid emergence of deep-learning application-specific integrated circuit (ASIC) accelerators which are prone to have higher error rates than general-purpose computing hardware do [8].

Work was done while Peter and Daya were at Meta.

978-1-6654-8045-1/22/$31.00

Enterprise computing is the first to employ soft error detection, followed by the High Performance Computing (HPC) community [9]–[13], and most recently, error detection methods are explored for convolutional neural networks (CNNs) deployed in autonomous driving [14]. While deep learning recommendation systems (DLRMs) may not be critical to personal safety, their computational integrity is crucial to maintain good experience of billions of users per day by ensuring content recommendation results are not corrupted. A deployable soft-error detection method for DLRMs must only incur low performance overhead lest the goal of maintaining user experience be self defeated, making algorithmic based fault tolerant method (ABFT) [15] the prime candidate. However, to the best of our knowledge, there is no previous ABFT work targeting DLRMs which typically compute in low-precision quantized arithmetic (details in III-A).

The two workhorse operators of DLRM are general matrix-matrix multiplication (GEMM) and EmbeddingBag (EB) which together account for over 70% of a DLRM's compute latency. Although ABFT for GEMM has been well studied in the literature, their straightforward adaption to DRLM results in high overhead due to DLRM's peculiar matrix sizes and shapes and its use of quantized arithmetic. In addition, EB is an operator not present in HPC or even in convolutional neural network. This paper considers error detection by ABFT for these two operators. We do not focus on error resilience as that is relatively simple for recommendation systems: once an error is detected a recommendation score can be recomputed easily assuming error striking twice is very rare.

The paper makes the following contributions on efficient soft-error detection for the key building blocks of DLRM in the quantized arithmetic domain:

- We propose the first ABFT implementation for quantized GEMM. By carefully customizing ABFT for GEMM, we optimize the performance and analyze its error detection ability;

- We propose the first ABFT implementation for EB, which is especially important for recommendation models.

In the following, we first briefly review related works in Section II, followed by Section III which explains the low-precision arithmetic used in many industrial machine learning models including DLRMs, and its two main operators that we focus on. Sections IV and V present our two ABFT algorithms and implementation considerations. Section VI presents our experiments to support our statements on the proposed algorithms' performance and efficacy. Section VII makes some concluding remarks.

II. Related Work

Soft error resiliency in deep learning models have been attracting more and more attention in recent years. Redundancy based protections are the most general and reliable solutions, where redundancy can be done at the hardware or software level. Hardware level redundancy is usually used in safety-critical task such as self-driving [16]. Software level redundancy can be done in the same hardware but with duplicated or tripled program or instruction executions [17]. Error detection by redundancy incurs at least 100% in overhead.

ABFT is a low-overhead error detection method. It was first proposed for the purpose of verifying matrix-matrix multiplication computation results [15], [18]. ABFT is less general but it has very low performance overhead. Thus, it has been applied to other matrix operations like matrix decomposition [19]–[22] and also iterative methods [23]. Recently, it has been applied and optimized for convolutional neural networks (CNN) [14], [24], [25]. These ABFT works either target convolution specifically or rely on extra-precision intermediate computation. While one can adapt to some extent these work to GEMM, we aim to eliminate the use of extra precision to further reduce overhead and devise ABFT for GEMM that is DLRM-specific. Furthermore, the EB operator is peculiar to DLRMs and hitherto unexplored.

III. Arithmetic and Operators

A. Quantized Arithmetic

Deep learning intrinsically relies on computing with real numbers. If the representation and computation of these real numbers can use, say, 8-bit integers instead of 32-bit floating-point numbers, significant memory saving and performance boost can be obtained. Arithmetic in integer to approximate floating-point computation is commonly called quantized arithmetic [26]. One first transforms linearly an interval $[x_{\min}, x_{\max}]$ of interest to the domain of the integer arithmetic in question. For example, $[0, 255]$ for 8-bit unsigned integer: Determine floating-point numbers α, β so that $(x-\beta)/\alpha \in [0, 255]$ for all $x \in [x_{\min}, x_{\max}]$. The resulting value is then rounded to an integer, yielding x_I, hence $x \approx \alpha x_I + \beta$. In quantized arithmetic, instead of multiplying two floating-point matrices $A \times B$ of dimension $m \times k$ and $k \times n$, the matrices are represented by (A_I, α_A, β_A) and (B_I, α_B, β_B) and the corresponding matrix product is realized as integer matrix product:

$$\begin{aligned} AB &\approx (\alpha_A A_I + \beta_A \vec{e}_m \vec{e}_k^T)(\alpha_B B_I + \beta_B \vec{e}_k \vec{e}_n^T) \\ &= \alpha_A \alpha_B A_I B_I + \\ &\quad \alpha_A \beta_B (A_I \vec{e}_k) \vec{e}_n^T + \alpha_B \beta_A \vec{e}_m (\vec{e}_k^T B_I) + k \beta_A \beta_B \vec{e}_m \vec{e}_n^T \end{aligned} \tag{1}$$

where $\vec{e}_\ell$ is the dimension-ℓ vector of all ones. Note all the terms following $A_I B_I$ are rank-1 matrices.

B. Low-precision GEMM in DLRM

Industrial implementations of DLRMs exploiting quantized arithmetic typically use specialized high-performance libraries such as FBGEMM [27]. As shown in Equation 1, the dominant operation is the integer matrix product $C_{\text{temp}} = A_I B_I$, consisting of $2mnk$ operations. This C_{temp}, a 32-bit integer matrix, together with the other rank-1 matrices and miscellaneous scale factors are than combined in a requantization process producing the $C \approx AB$ where C is represented by the tuple (C_I, α_C, β_C). We show the workflow in Figure 1. In the rest of the paper, when we refer to the matrices using A, B, C, they are corresponded to integer matrices A_I, B_I,

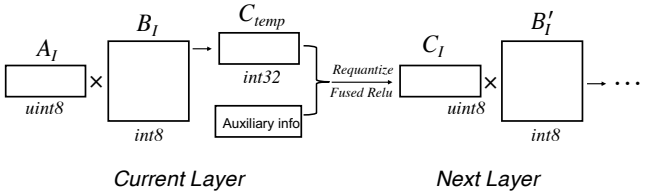


Fig. 1. Low-precision GEMM in deep learning context

C. EmbeddingBag and its low-precision variant

Embedding is a technique that maps discrete categorical data into a d-dimensional Euclidean spaces of real numbers. It is widely used in many recommendation systems [28]–[30]. An embedding table contains a number of d-length row vectors each corresponding to a categorical data and algebraic operations corresponds to combination of these categories. EmbeddingBag (EB)[1] is one of the most frequently called operators in these embedding based recommendation systems. An EB with batch size of one simply picks out the set of rows given by an index set $\mathcal{I}$ from an embedding table and sum them up, illustrated in Figure 2. It is also called one embedding lookup. Mathematically, given $\mathcal{I}$ and an embedding table T, EB returns $\vec{R} = \sum_{i \in \mathcal{I}} \vec{eb}_i$, where $\vec{eb}_i$ is the i-th row of the embedding table T. Note that for notational convenience we use $\vec{r}$ here to denote a *row* vector instead of the usual convention of a column vector.

Industrial scale DLRMs often have many embedding tables totaling hundred billions of parameters. Hence instead of using floating-point to represent these real numbers, quantized arithmetic is often used to reduce the DLRMs' memory footprints [31]. Specifically, each d-length embedding row at index i is represented by d-length vector of short (8

[1]https://pytorch.org/docs/stable/generated/torch.nn.EmbeddingBag.html

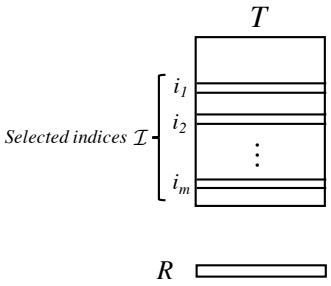


Fig. 2. Illustration of one embedding lookup in the EmbeddingBag operator with batch size of one

bits for example) integers $\vec{eb}_i$ and one pair of floating-point quantization parameter α_i, β_i. The corresponding EB operator must then compute $\vec{R} = \sum_{i \in \mathcal{I}} \alpha_i \vec{eb}_i + \beta_i \vec{e}_d$ where $\vec{e}_d$ is a d-length row vector of all ones.

As EB with batch size of one returns one row vector, EB with batch size n returns n vectors, each corresponding to sums of relevantly selected embedding rows from a particular embedding table:

$$R = \begin{bmatrix} \vec{R}_1 \\ \vec{R}_2 \\ \vdots \\ \vec{R}_n \end{bmatrix} \tag{2}$$

IV. Optimized ABFT for GEMM in DLRMs

The bulk of the computation in a quantized matrix product (Equation 1) is the usual matrix product of two integer matrices of the form $C = AB$ (dropping the subscripts of I) where A and B are matrices of dimensions m-by-k and k-by-n, respectively. Our aim is to detect soft error that happen during this computation after both A and B have been loaded into memory.

We start with this common ABFT method: One encodes A into an augmented matrix A' by appending a row vector S_A where $S_A[j] = \sum_{i=0}^{m-1} A[i][j]$. Similarly, B is encoded into an augmented matrix B' with an extra column vector S_B where $S_B[i] = \sum_{j=0}^{n-1} B[i][j]$. Figure 3 illustrates the augmented matrices A' and B' and their product C'. Mathematically, the

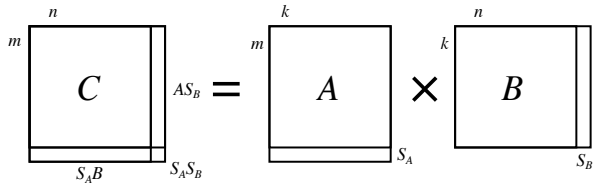


Fig. 3. Illustration of ABFT for GEMM

upper-left m-by-n block of C' is $C = AB$, first n columns of $C'[m,:]$ is $S_A\,B$, first m rows of $C'[:,n]$ is S_AB, and $C'[m,n] = S_AS_B$. Simple algebraic derivations show that a correctly computed C' satisfies the relationships

$$\forall \text{column } j \in [0, n-1], C'[m][j] = \sum_{i=0}^{m-1} C'[i][j] \tag{3a}$$

$$\forall \text{row } i \in [0, m-1], C'[i][n] = \sum_{j=0}^{n-1} C'[i][j] \tag{3b}$$

Equality checks of these equations on the computed C' form the basis of ABFT: If equality fails to hold at exactly one row i for Equation 3b together with exactly one column j for Equation 3a, then the value at the computed $C'[i][j]$ is faulty. Furthermore, a corrupted $C'[i][j]$ – revealed as a single violation at row i and at column j – can be corrected using the equation

$$\text{correct } C'[i][j] = C'[m][j] - \sum_{p \neq i} C'[p][j],$$

or

$$\text{correct } C'[i][j] = C'[i][n] - \sum_{p \neq j} C'[i][p].$$

Straightforward as this common ABFT method for GEMM is, adopting them with low enough overhead that does not impede DLRM user experience requires a number of techniques that we now discuss.

A. Performance optimizations

1) Encoding only matrix B: Existing work of ABFT for GEMM considers soft error detection and single error correction. We stated previously (Section I) that we aim solely at error detection. Thus, we just need to encode A or B, but not both. The question is which matrix to encode. To better understand this question, we first derive the theoretical error detection overheads with encoding A or encoding B. Remember that the error detection includes basically 3 stages: encode matrix A (or B); do GEMM with encoded A (or B); verify result matrix by checking Equation 3. The overheads are:

$$\text{overhead if encoding } A\text{: } \frac{mk + 2nk + mn}{2mnk} = \frac{1}{2n} + \frac{1}{m} + \frac{1}{2k}$$
$$\text{overhead if encoding } B\text{: } \frac{kn + 2mk + mn}{2mnk} = \frac{1}{2m} + \frac{1}{n} + \frac{1}{2k}$$

We follow the convention in PyTorch where A corresponds to activations and B the weight parameters of the neural network. Common in DLRMs, m is relatively much smaller than n or k. According to the theoretical overhead equation, encoding matrix B will have smaller overheads than encoding A.

Another fact also makes encoding B preferable in the aspects of both performance and memory error detection ability: B, being the trained weight matrix, stays still in the memory for a much longer time. From the perspective of performance overheads, the fact implies we can encode matrix B once for multiple GEMM operations thus amortizes the encoding overheads. From the perspective of memory error detection ability, the fact implies matrix B have much higher chances to experience memory errors than matrix A. (Recall that encoding matrix A will not detect memory errors in B

while encoding matrix B will do. In order to cover the errors in B, we choose to encode B.) In conclusion, we encode B instead of A so as to minimize ABFT performance overheads while maximizing detection ability.

2) Keeping encoded column in low precision: The encoded row sum vector for matrix B seems to require 32-bit integer as value container to ensure correctness. This implies a high overhead because ABFT work has to be done in 32-bit integers while the original GEMM work is done in 8-bit integers. Computation with 32-bit integers can be 2 to 4 times slower than that with 8-bit. To reduce the overheads, we use modulo operations to map the 32-bit row sums into 8-bit. The Equations 3 are proved to still hold under the same modulus [15]. Using modulo operations in the ABFT context is not novel. But we exploit them for better performance rather than to bypass limitation of computer word length [15].

3) Keeping BLAS level-3 updating: A straightforward implementation of ABFT for GEMM (encoding only B) will be: ① calculate row sums of B and store the result (S_B) in a separate vector; ② compute $C = A * B$; ③ compute $A * S_B$; ④ check if row sums of C equal $A * S_B$. This implementation does not need to modify the normal data structure of B to accommodate an extra column, but results in high performance overhead. This is because Step ③ is a matrix-vector product, a BLAS (Basic Linear Algebra Subprograms) level 2 operation. An alternative implementation that relies BLAS level 3 operations can be: ① allocate new memory for encoded matrix B' and new memory for C'; ② do GEMM between A and B' and store result in C'; ③ check Equation 3; ④ copy former m rows and n columns of C' back into C. The drawback of this implementation is its high memory overhead.

We found a way to implement ABFT for low-precision GEMMs in BLAS level 3 operations and with small memory overhead. This is possible because of two facts: 1. matrix B is packed into blocks before being sent to the efficient GEMM kernel; 2. the C matrix in 32-bit integers are intermediate result (as shown in Figure 1 by C_{temp}). The first fact means that we can pack the original B and the separate vector storing row sums together into blocks so that the blocks look like they are from encoded B' in contiguous memory space. The second fact means that we can directly allocate one more column for the intermediate result matrix than before. Notice we are not increasing the number of columns of 8-bit result matrix. We just need to modify the requantization procedure to let it exclude the last column of the intermediate 32-bit matrix.

B. ABFT detection before requantization

One may ask if we can delay the checksum equality check from examining C_{temp} (Equation 3) to examining C_I so as to detect silent errors in requantization process. Unfortunately, the answer is no. The main reason is that requantization is not a linear operation, i.e., $Q(a) + Q(b) \neq Q(a + b)$ generally where Q is the requantization operator. Thus, our linear encoding scheme cannot make equality hold in C_I. Lack of error detection for requantization process is not serious considering that this process is less error prone as it only takes around 2% of execution time for larger matrices and around 5% for smaller matrices.

C. Modulus selection and detection ability analysis

As we use modulo to keep the encoded column in low precision to reduce ABFT overhead, the downside is the weakened error detection ability. In this section, we want to discuss how to choose the modulus wisely so that the detection ability degradation is minimized. We assume elements in 8-bit unsigned integer matrix A and 8-bit signed integer matrix B are both random numbers in the uniform distribution independently. We also assume the there are no errors for the encoded column considering its much smaller memory usage and operations numbers compared to the original computation. First, let us look at the situations when the modulus, mod, will fail to detect errors. For some rows in the result matrix C, we denote its row sum (excluding the encoded checksum column) by $rsum$ without any soft error. If soft errors happen and corrupt that row, we denote its row sum by $rsum'$. Then there is a fact that when the absolute value of difference between $rsum'$ and $rsum$ is divisible by mod, the soft error will not be detected. Also, that is only the case when the errors are not detected. More formally, soft errors corrupting that row are not detected if and only if $|rsum' - rsum| \% mod = 0$.

We consider two fault models. The first commonly used fault model is the random single-bit flip model which means a random bit of the data in the memory or register flips from 0 to 1 or 1 to 0. The intuition to this model is that $|rsum' - rsum|$ will be powers of 2. That is, any odd modulus can detect all errors in this model. The second model is random data fluctuation which means the correct value of the data is changed to some arbitrary value representable in its data type. For example, a 32-bit signed integer data can be changed to any value in the range of $[-2^{31}, 2^{31} - 1]$ with equal likelihood. The intuition to this model is that the larger the modulus is, the smaller number of its multiples is (i. e., the better detection ability is). Those two intuitions give us a good modulus which is 127 for matrix B as it is the biggest odd number in the range of B. In the rest of this section, we then use 127 as the modulus to simplify the calculation of detection ability.

We quantify the detection ability in terms of probability. Specifically, the detection ability is measured by the probability our modulus based error detection method can detect error(s) when the result matrix C is indeed corrupted. This metric is also known as the *true positive rate*.

1) Memory error in 8-bit matrix B: An error in matrix B can propagate to corrupt a whole column of matrix C. Specifically, suppose the corruption happens at $B[i][j]$ and result in a difference of d. The j-th column of result matrix will be corrupted. Since $B[i][j]$ will be multiplied by $A[p][i]$ for all p in $[0, m - 1]$, the difference of the corresponding row sums in the result matrix will be $d * A[p][i]$. Recall that ABFT cannot detect soft errors if $|d * A[p][i]|$ is a multiple of 127. Notice that 127 is a prime number. By Euclid's lemma[1],

[1]If a prime number a divides the product, $b * c$, a divides b or c.

$|d * A[p][i]|$ is multiple of 127 if and only if $|d|$ or $|A[p][i]|$ is a multiple of 127. In the first fault model (random bit-flip at $B[i][j]$), $|d|$ could be 2^l where $l \in [0, 7]$. $|d * A[p][i]|$ is a multiple of 127 if and only if $|A[p][i]|$ equals 127, 254, or 0 since matrix A is in 8-bit unsigned integers. i. e., the p-th row will not detect the soft error in probability of $\frac{3}{256}$ assuming $A[p][i]$ randomly ranges in $[0, 255]$. Since all rows will be checked by ABFT, the probability that the error is not detected by all rows will be $(\frac{3}{256})^m$. Thus, the error is detected in the probability of $1 - (\frac{3}{256})^m \geq 98.83\%$.

In the second fault model, $|d|$ can be random in the range of $[1, 255]$. $|d * A[p][i]|$ is multiple of 127 if and only if $|d|$ equals 127 or $A[p][i]$ equals 127, 254, or 0. That is, the p-th row will not detect the soft error in probability of

$$\frac{1 * 256 + 255 * 3 - 3}{255 * 128} = \frac{1018}{32640}$$

assuming $A[p][i]$ is uniformly distributed in $[0, 255]$. Similar to the above analysis, the probability that the error is not detected by all rows will be $(\frac{1018}{32640})^m$. Thus, the error is detected with probability $1 - (\frac{1018}{32640})^m \geq 96.89\%$.

2) Memory error in 32-bit intermediate result matrix C (C_{temp}): In the first fault model, a random bit-flip in C implies the absolute value of difference of its corrupted row sum from its expected value to be 2^i for $i \in [0, 31]$. Thus, the error will be detected with probability 100% since 127 cannot divide any 2^i for $i \in [0, 31]$.

In the second fault model, suppose a random element c in C is changed to another arbitrary value c'. Then the difference in absolute value of its corrupted row sum and expected one is also $|c' - c|$. Think about c is located somewhere in an interval of $[-2^{31}, 2^{31} - 1]$. We can conclude the range of $|c' - c|$ is $(0, 2^{31} - 1 - c]$ or $(0, c + 2^{31}]$ where $2^{31} - 1 - c$ is taken when $c' = 2^{31} - 1$ and $c + 2^{31}$ is taken when $c' = -2^{31}$. Denote the number of multiples of mod in the range of $(0, a]$ by $f(a)$. We can prove the following property, $f(a) + f(b) \leq f(a + b)$. The key is that if mod divides a and b, $f(a) + f(b) = f(a + b)$.

$$\begin{aligned} f(a) + f(b) &= f(a - a\%mod) + f(b - b\%mod) \\ &= f(a - a\%mod + b - b\%mod) \\ &= f(a + b - (a\%mod + b\%mod)) \\ &\leq f(a + b) \end{aligned}$$

Thus, number of multiples of mod in the range of $(0, 2^{31} - 1 - c]$ and $(0, c + 2^{31}]$ is less than $f(2^{31} - 1 - c + c + 2^{31}) = f(2^{31} - 1) = \frac{2^{31}-1}{mod}$. Thus the detection probability of an error in this model will be at least $1 - \frac{1}{mod} = 99.21\%$.

3) Memory error in matrix A and computational error: As we mentioned, matrix B takes much larger memory space and resides in the memory much longer than matrix A. To keep ABFT overhead low, we only encode matrix B and this means we do not provide memory error detection for matrix A. A computational soft error will corrupt the intermediate result of $A[i][k] * B[k][j]$. Thus it has the same behaviour as memory errors do in the 32-bit result matrix, C where we discussed in Section IV-C2.

Algorithm 1 ABFT for low-precision GEMM

Input: 8-bit integer matrix A, B; dimension sizes m, n, k
Output: 32-bit integer matrix C_{temp}; number of corrupted rows

```
mod ← 127
for i from 0 to k − 1 do
    rowSum[i] ← Σ_{j=0}^{n−1} B[i][j]
    rowSum[i] %= mod
end for
packedEncodedB ← pack(B, rowSum[])
allocate C_temp[m][n + 1]
C_temp[][] ← A * packedEncodedB
errCount ← 0
for i from 0 to m − 1 do
    tSum ← Σ_{j=0}^{n−1} C_temp[i][j]
    if tSum%mod ≠ C_temp[i][n]%mod then
        errCount++
    end if
end for
return C_temp; errCount
```

The complete look of our customized ABFT for low-precision GEMM is presented in Algorithm 1.

V. ABFT for low-precision EmbeddingBag

A. ABFT for EB

Based on the EB operator we introduced in Section III-C, we propose the ABFT technique for EB. To our best knowledge, this is the first ABFT technique for EB operator. Recall that we use d to denote the embedding row dimension. The method is illustrated in Figure 4. $\vec{C_T}$ is a column vector

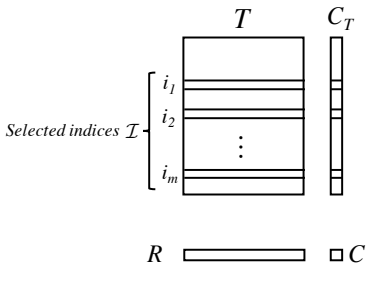


Fig. 4. Illustration of ABFT for EB with batch size one

storing all the row sums of the embedding table. If we sum the elements of $\vec{C_T}$ at the indices of $\mathcal{I}$, it is easy to find the result, C, will be equal to the sum of all elements in $\vec{R}$. Specifically, the following equality holds. ABFT will check if this equality holds to detect soft errors.

$$\sum_{j=0}^{d-1} \vec{R}[j] = C = \sum_{i \in \mathcal{I}} \vec{C_T}[i] \tag{4}$$

If the batch size is more than one, we just apply the equality check for all EBs in the batch.

B. Adaption to low-precision EB

Recall the low-precision EB variant we introduced in Section III-C. Each embedding row vector in low-precision integers will be multiplied by a scale factor α_i and added by a bias value β_i. Then equation 4 should also be updated to accommodate the scale factor and bias as shown in Equation 5.

$$\sum_{j=0}^{d-1} \vec{R}[j] = \sum_{i \in \mathcal{I}} \alpha_i * \vec{C_T}[i] + d * \beta_i \tag{5}$$

The correctness of the above equation is shown as following. Recall that $\vec{e_d}$ is a d-length vector of all ones.

$$\begin{aligned}
\sum_{j=0}^{d-1} \vec{R}[j] &= \sum_{j=0}^{d-1} (\sum_{i \in \mathcal{I}} (\alpha_i * \vec{eb_i}[j] + \beta_i * \vec{e_d}[j])) \\
&= \sum_{i \in \mathcal{I}} (\sum_{j=0}^{d-1} (\alpha_i * \vec{eb_i}[j] + \beta_i * \vec{e_d}[j])) \\
&= \sum_{i \in \mathcal{I}} (\alpha_i * \sum_{j=0}^{d-1} \vec{eb_i}[j] + \sum_{j=0}^{d-1} \beta_i) \\
&= \sum_{i \in \mathcal{I}} \alpha_i * \vec{C_T}[i] + d * \beta_i
\end{aligned}$$

Notice that instead of storing the scaled and bias row sums in 32-bit float type, we still store the row sums in 32-bit integers without being scaled or biased in $\vec{C_T}$. This way we can minimize the accumulation of round off errors when we sum up the elements in $\vec{C_T}$. The details of ABFT for low-precision EB is presented in Algorithm 2.

Algorithm 2 ABFT for low-precision EB

Input: embedding table T; length of embedding vector, d; scale factor array, $\alpha[]$; bias array, $\beta[]$; set of selected indices, $\mathcal{I}$; pre-computed row sums of T, $C_T[]$
Output: EB result, $R[]$; *err*

1: $R[] \leftarrow$ EmbeddingBag(T, d, $\alpha[]$, $\beta[]$, $\mathcal{I}$)
2: $RSum \leftarrow \sum_{j=0}^{d-1} R[j]$
3: $CSum \leftarrow \sum_{i \in \mathcal{I}} (\alpha[i] * C_T[i] + d * \beta[i])$
4: $err \leftarrow$ "False"
5: **if** $|RSum - CSum| >$ roundOffErrorBound **then**
6: $\quad err \leftarrow$ "True"
7: **end if**
8: **return** $R[]$; err

C. Overhead analysis

Denote the number of selected indices by m and the length of the embedding vector by d. Notice that in Algorithm 2, the row sums of embedding table is pre-computed. This can be done because once the embedding table is trained, it will stay unchanged like the weight matrix (matrix B) in FC layers. Thus, we do not include the operations to calculate row sums as the ABFT overhead. The number of operations in the original EB without ABFT is $3md$ and extra operations for ABFT is $3m + d$. So the overhead in fraction is $\frac{1}{d} + \frac{1}{3m}$. In terms of memory overhead, the 32-bit row sums will take $\frac{32}{pd}$ more memory space where p is the number of bits (4 or 8) of the low-precision integer in the embedding table.

D. Round off error bound

Unlike low precision GEMM where all calculations involve only integer, EmbeddingBag operators have floating point numbers where round off error can accumulate. We set a bound to differentiate soft error from round off error in $RSum$ and $CSum$ (as shown in line 5 of Algorithm 2). Setting an appropriate bound is nontrivial [11] because too large a bound will let lots of soft error escape from the detection and too small means very high false positive rate. We choose a relative bound 1E-5 for our EmbeddingBag operators. This is a loose bound but its detection accuracy is good enough as we will show later. Why we choose a loose bound is because soft errors leading to small fluctuation of floating point results usually does not have big impact to the final machine learning inference [32].

VI. Evaluation

In this section, we evaluate our proposed ABFT soft error detection for low-precision GEMM and EmbeddingBag. The solutions are evaluated in both error free case and erroneous case. A good soft error detector should have two properties: low performance overhead and low (or no) false positives in error free case; great detection ability (or high true positives) in erroneous case.

A. Performance overhead

1) ABFT for low-precision GEMM: Without any soft errors, Figure 5 shows the the performance overhead of our ABFT for low-precision GEMM with different input matrix shapes. Notice that those shapes are frequently used in DLRM and they are not square. We can see from the figure that the ABFT overheads are under 20% for all the 28 shapes. Actually, ABFT overheads are under 10% for many of the shapes (17 out of 28 shapes); under 5% for 7 of the shapes. Notice that for the shape (m, n, k) of 1, 800, 3200, GEMM runs faster than its unprotected version. We think the reason is for that specific setting, adding one more column to matrix B makes the cache performance better.

2) ABFT for low-precision EmbeddingBag: We test the performance overheads of our proposed error detection method (Algorithm 2) using quantized 8-bit integer embedding table. We flush the cache since the embedding table is too large to be hold in the cache in real world scenario. We tested both regular sum and weighted sum with prefetching optimization turned on and off. The specific parameters we use is listed in Table I. The table columns are also known as embedding dimensions. The average pooling size is the average number of pooled embedding table rows by all EBs in a batch. For example, suppose a batch of two EBs. The first one takes 3 rows from the table and the second takes 5. The average pooling size will be 4.

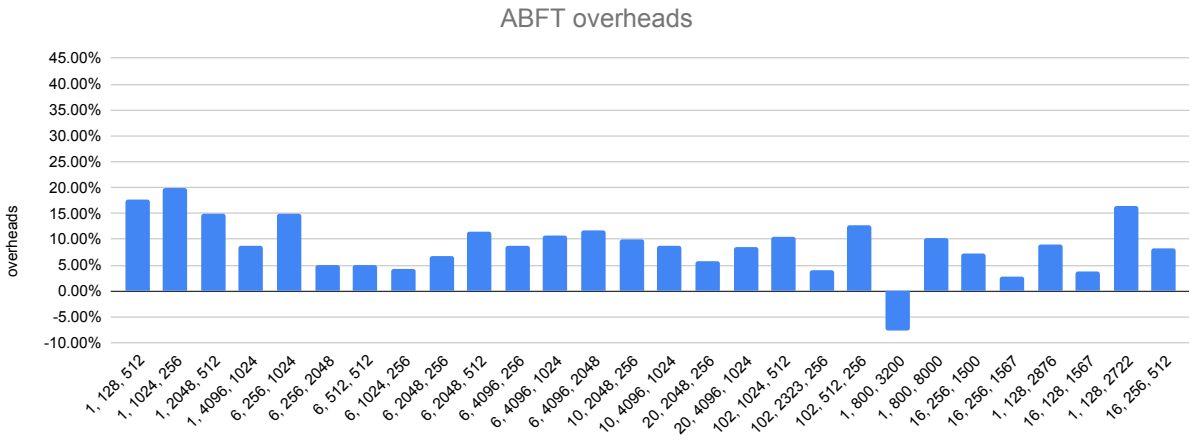


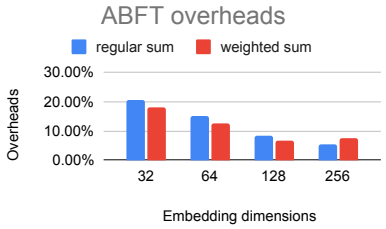
Fig. 5. Performance overhead of ABFT for low-precision GEMMs with different shapes (m, n, k)

TABLE I
EMBEDDING TABLE SIZE AND EXPERIMENTAL PARAMETERS FOR ABFT EMBEDDINGBAG

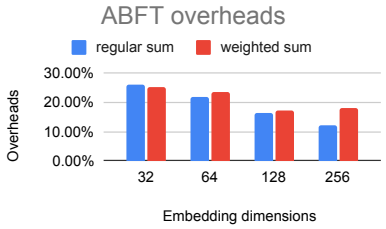
table rows	table columns	average pooling size	batch size
4,000,000	32	100	10
4,000,000	64	100	10
4,000,000	128	100	10
4,000,000	256	100	10

TABLE II
NUMBER OF DETECTED RUNS AND NOT DETECTED RUNS WITH SIMULATED ERROR IN GEMM

	error in B	error in C	no error
detected runs	2663	2800	0
not detected runs	137	0	2800
total	2800	2800	2800



(a) without prefetching



(b) with prefetching

Fig. 6. Performance overheads of ABFT for low-precision EmbeddingBag with different settings

B. Experiments with simulated error

We evaluate the detection accuracy of our proposed detection with simulated errors at source code level. The simulated errors are done by randomly selecting an element in the input or output and flipping a random bit in that element.

1) ABFT for low-precision GEMM: We first inject a random bit flip in the input matrix B after the checksum of B has been calculated and repeat the experiments for each shape 100 times totalling 2800 samples. Then we do the random bit flip injection to the 32-bit intermediate result matrix and conduct another 2800 samples. The results are shown in table II. We can see that the detection accuracy when matrix B is injected with error is $\frac{2663}{2800} = 95.11\%$. This is 3.72% less than the theoretical estimation in Section IV-C1 but still very high. We achieve 100% detection accuracy when the random bit flip happens in matrix C and it is consistent with our analysis in Section IV-C2. It is worth noting that we also conducted 2800 runs of error free experiments to validate our false positive rate is zero since there is no round off error in integer operations.

2) ABFT for low-precision EmbeddingBag: We tested the detection accuracy of our proposed solution with 8-bit integer embedding table. For each run, we randomly choose an element and flip a random bit in it. We repeated 400 runs with injected errors and 400 runs without injected errors. Among those 400 runs with errors, 200 of them are injected with bit flips in the upper 4 significant bits and the other 200 are injected with bit flips in the lower 4 insignificant bits. The results are shown in Table III. We can see the detection rate for significant 4 bits are pretty high at 99.5%. The detection rate for insignificant 4 bits are dropped to 47%. The false positive rate is 9.5%. As we can see from the results, our bound is chosen to be loose so that we can have lower false positive rates and the bad thing is that for an insignificant bit flip, detection rate is not high.

TABLE III
NUMBER OF DETECTED RUNS AND NOT DETECTED RUNS WITH SIMULATED ERROR IN EMBEDDINGBAG

	high bits	low bits	no error
detected runs	199	94	38
not detected runs	1	106	362
total	200	200	400

VII. CONCLUSION AND FUTURE WORK

In this paper, we propose efficient algorithm-based soft error detections for two important low-precision operators, GEMM and EmbeddingBag, in DLRM. This is the first work to benefit those deep learning operators unlike others focusing on convolutional workloads. By careful design and optimization, our soft-error detection can achieve greater than 95% in error detection ability and introduces small overheads less than 26%.

ACKNOWLEDGMENTS

This work was supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, Scientific Discovery through the Advanced Computing (SciDAC) program under Award Number DESC0022209.

REFERENCES

[1] S. Di, M. S. Bouguerra, L. Bautista-Gomez, and F. Cappello, "Optimization of multi-level checkpoint model for large scale hpc applications," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 1181–1190.

[2] D. Hakkarinen and Z. Chen, "Multilevel diskless checkpointing," *IEEE Transactions on Computers*, vol. 62, no. 4, pp. 772–783, 2012.

[3] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design & Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[4] A. Geist, "How to kill a supercomputer: Dirty power, cosmic rays, and bad solder," *IEEE Spectrum*, vol. 10, pp. 2–3, 2016.

[5] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent data corruptions at scale," 2021.

[6] A. Geist, "Supercomputing's monster in the closet," *IEEE Spectrum*, vol. 53, no. 3, pp. 30–35, 2016.

[7] B. Nie, J. Xue, S. Gupta, C. Engelmann, E. Smirni, and D. Tiwari, "Characterizing temperature, power, and soft-error behaviors in data center systems: Insights, challenges, and opportunities," in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE, 2017, pp. 22–31.

[8] J. Zhang, K. Rangineni, Z. Ghodsi, and S. Garg, "Thundervolt: enabling aggressive voltage underscaling and timing error resilience for energy efficient deep learning accelerators," in *Proceedings of the 55th Annual Design Automation Conference*, 2018, pp. 1–6.

[9] P. Wu, N. DeBardeleben, Q. Guan, S. Blanchard, J. Chen, D. Tao, X. Liang, K. Ouyang, and Z. Chen, "Silent data corruption resilient two-sided matrix factorizations," in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2017, pp. 415–427.

[10] J. Chen, H. Li, S. Li, X. Liang, P. Wu, D. Tao, K. Ouyang, Y. Liu, K. Zhao, Q. Guan *et al.*, "Fault tolerant one-sided matrix decompositions on heterogeneous systems with gpus," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 854–865.

[11] X. Liang, J. Chen, D. Tao, S. Li, P. Wu, H. Li, K. Ouyang, Y. Liu, F. Song, and Z. Chen, "Correcting soft errors online in fast fourier transform," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.

[12] Z. Chen, "Online-abft: An online algorithm based fault tolerance scheme for soft error detection in iterative methods," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 167–176, 2013.

[13] D. Tao, S. L. Song, S. Krishnamoorthy, P. Wu, X. Liang, E. Z. Zhang, D. Kerbyson, and Z. Chen, "New-sum: A novel online abft scheme for general iterative methods," in *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, 2016, pp. 43–55.

[14] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," *arXiv preprint arXiv:2006.04984*, 2020.

[15] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[16] E. Talpes, D. D. Sarma, G. Venkataramanan, P. Bannon, B. McGee, B. Floering, A. Jalote, C. Hsiong, S. Arora, A. Gorti *et al.*, "Compute solution for tesla's full self-driving computer," *IEEE Micro*, vol. 40, no. 2, pp. 25–35, 2020.

[17] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August, "Swift: Software implemented fault tolerance," in *International Symposium on Code Generation and Optimization*. IEEE, 2005, pp. 243–254.

[18] Z. Chen, "Optimal real number codes for fault tolerant matrix operations," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, 2009, pp. 1–10.

[19] 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.

[20] Z. Chen, "Extending algorithm-based fault tolerance to tolerate fail-stop failures in high performance distributed environments," in *2008 IEEE International Symposium on Parallel and Distributed Processing*. IEEE, 2008, pp. 1–8.

[21] D. Hakkarinen, P. Wu, and Z. Chen, "Fail-stop failure algorithm-based fault tolerance for cholesky decomposition," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 5, pp. 1323–1335, 2014.

[22] P. Wu, Q. Guan, N. DeBardeleben, S. Blanchard, D. Tao, X. Liang, Jieyang, and Z. Chen, "Towards practical algorithm based fault tolerance in dense linear algebra," in *Proceedings of HPDC'16*. ACM, 2016, pp. 31–42.

[23] D. Tao, S. Di, X. Liang, Z. Chen, and F. Cappello, "Improving performance of iterative methods by lossy checkponting," in *Proceedings of the 27th international symposium on high-performance parallel and distributed computing*, 2018, pp. 52–65.

[24] F. F. dos Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2018.

[25] K. Zhao, S. Di, S. Li, X. Liang, Y. Zhai, J. Chen, K. Ouyang, F. Cappello, and Z. Chen, "Algorithm-based fault tolerance for convolutional neural networks," *arXiv preprint arXiv:2003.12203*, 2020.

[26] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 2704–2713.

[27] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy, "Fbgemm: Enabling high-performance low-precision deep learning inference," *arXiv preprint arXiv:2101.05615*, 2021.

[28] W. Zhao, J. Zhang, D. Xie, Y. Qian, R. Jia, and P. Li, "Aibox: Ctr prediction model training on a single node," in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 319–328.

[29] W. Zhao, D. Xie, R. Jia, Y. Qian, R. Ding, M. Sun, and P. Li, "Distributed hierarchical gpu parameter server for massive scale deep learning ads systems," *arXiv preprint arXiv:2003.05622*, 2020.

[30] M. Xie, K. Ren, Y. Lu, G. Yang, Q. Xu, B. Wu, J. Lin, H. g. Ao, W. Xu, and J. Shu, "Kraken: memory-efficient continual learning for large-scale real-time recommendations," in *2020 SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE Computer Society, 2020, pp. 278–294.

[31] H. Guan, A. Malevich, J. Yang, J. Park, and H. Yuen, "Post-training 4-bit quantization on embedding tables," *arXiv:1911.02079*, 2019.

[32] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding error propagation in deep learning neural network (dnn) accelerators and applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017, pp. 1–12.