A Coding Scheme for Reliable In-Memory Hamming Distance Computation

Zehui Chen*, Clayton Schoeny*, Yuval Cassuto⊤, and Lara Dolecek*

*Department Electrical Engineering, University of California, Los Angeles
⊤Department of Electrical Engineering, Technion - Israel Institute of Technology
chen1046@ucla.edu, cschoeny@ucla.edu, ycassuto@ee.technion.ac.il, dolecek@ee.ucla.edu

***Abstract*—Computation-in-memory is a technique that has shown great potential in reducing the burden of massive data processing. Allowing for ultra-fast Hamming distance computations to be performed in-memory will drastically speed up many modern machine-learning algorithms. However, these in-memory calculations have not been studied in the presence of process variabilities. In this paper, we develop coding schemes to reliably compute, in-memory, the Hamming distances of pairs of vectors in the presence of write-time errors. Using an inversion coding technique, we establish error-detection guarantees as a function of the number of errors and the non-ideality of the resistive array memory in which the data is stored. To correct errors in the vector similarity comparison, we propose codes that achieve error correction and useful techniques for bit level data access and error localization. We demonstrate the effectiveness of our coding scheme on a simple example using the k-nearest neighbors algorithm.**

I. Introduction

With the current data deluge, it is imperative to have the means to store and process vast amounts of high-dimensional data. In order to tackle this challenge, novel memory mediums with high density and low latency have been created, but the bottleneck between memory and processor stile largely persists [1]. In order to bypass this bottleneck, a revolutionary idea of *computation-in-memory* has been proposed, in which certain computations are performed in the physical memory itself [2]. Computing similarity metrics between vectors is a critical component in machine-learning algorithms used for applications such as image recognition and natural language processing [3], [4]. Recently, resistive memory has been shown to have natural properties conducive to efficiently performing Hamming distance calculations in-memory with low-level conductance measurements [5].

Existing techniques for the computation of in-memory Hamming distances have assumed that the data is stored error-free [5]. Furthermore, although in-memory Hamming distance computation offers promise of unprecedented latency performance, data can only be accessed at vector level, not the bit level. Thus, traditional coding theory based on bit level information does not apply. It is necessary to rethink code design and the role of coding when addressing the robustness issue of in-memory Hamming distance computation. We propose coding techniques that enable in-memory similarity computations in the presence of write-time errors; thus our coding framework provides a promising alternative to power hungry physical-level methods of protecting against write-time errors. Our work will pave the way for future practical implementation of in-memory Hamming distance computation.

The content of this paper is organized as follows. Our goal is first to provide a coding scheme to detect incorrect Hamming distance calculation while still using low-level measurements between vectors; then to provide coding scheme to correct those incorrect calculations. In Section II, we provide the necessary background about resistive memories, in-memory Hamming distance calculations, and the source of error. In Section III we establish the in-memory error-detection capabilities based on inversion coding. In Section IV, we construct a coding scheme for error correction along with techniques for bit level data access and error localization. We also demonstrate the effectiveness of our code on a simple machine-learning example.

II. Background

A. In-Memory Hamming Distance Computation

In resistive memory, memristors are placed at the intersections of a crossbar structure. The resistance of memristors, high or low, represent two logical states [6], where the Low/High conductance ratio is characterized by ϵ. In this work, we limit our operation with the resistive memory to simply measuring the conductance between pairs of rows (and pairs of sub-rows in Section IV).

We briefly review how to compute the Hamming distance in-memory between pairs of vectors using a technique presented in [5]. Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be binary vectors of length n. The normalized conductance measurement, $G(\boldsymbol{x}, \boldsymbol{y})$, can be viewed as the output of a deterministic multiple-access channel with binary vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ as inputs. $G(\boldsymbol{x}, \boldsymbol{y})$ is calculated by summing the outputs of the function f, detailed in Table I, over each bit of $\boldsymbol{x}$ and $\boldsymbol{y}$, i.e., $G(\boldsymbol{x}, \boldsymbol{y}) = \sum_{i=1}^{n} f(x_i, y_i)$.

TABLE I: MAC output function

x_i	y_i	f
0	0	ϵ
0	1	$\frac{2\epsilon}{1+\epsilon}$
1	0	
1	1	1

Note that our information about vectors $\boldsymbol{x}$ and $\boldsymbol{y}$ is limited solely to the normalized conductance measurement $G(\boldsymbol{x}, \boldsymbol{y})$. Cassuto and Crammer showed that—with knowledge of the Hamming weights—one can calculate the Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{y}$ as follows [5]:

$$D(\boldsymbol{x}, \boldsymbol{y}) = \frac{1+\epsilon}{(1-\epsilon)^2}[(1-\epsilon)(W(\boldsymbol{x})+W(\boldsymbol{y}))+2n\epsilon-2G(\boldsymbol{x}, \boldsymbol{y})], \tag{1}$$

where ϵ is a real-valued constant less than one greater than zero, $W(\boldsymbol{x})$, $W(\boldsymbol{y})$ are the Hamming weights of $\boldsymbol{x}, \boldsymbol{y}$, respectively, and $D(\boldsymbol{x}, \boldsymbol{y})$ is the Hamming distance between $\boldsymbol{x}$ and $\boldsymbol{y}$. The standard definitions of Hamming distance and Hamming weight are used. Additionally, by using a reference vector, the all-1s vector, it was shown in [5] that the Hamming weight of a vector is found by evaluating

$$W(\boldsymbol{x}) = \frac{(1+\epsilon)G(\boldsymbol{x}, \mathbf{1}) - 2n\epsilon}{1-\epsilon}.$$

Furthermore, coding strategies to calculate $D(\boldsymbol{x}, \boldsymbol{y})$ for a variety of ϵ ranges and known weight conditions are derived in [5].

B. Memory Error Source

In this paper, we consider errors due to process variability in the write operation. The switching time of a memristor follows a log-normal distribution with an exponential median switching time dependent on the external voltage [7]. Two common techniques used to mitigate write uncertainty are increasing the voltage of the write and using a feedback writing scheme [8]; however, these approaches are usually not energy efficient [9]. Unsuccessful write operations will lead to bit-errors when the former state of a memristor is different from the data to be written.

In the following sections that provide analysis for the case of t bit-errors, using the insights from [9], we assume the locations of the bits in error are independent and uniformly distributed. Specifically, in Section V, we use a binary symmetric channel (BSC) as our model for the bit-errors.

III. Coding for Error Detection

Let $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ be the (possibly noisy) vectors actually written to the device when we intend to store vectors $\boldsymbol{x}$ and $\boldsymbol{y}$, respectively. The discrepancy between $\hat{\boldsymbol{x}}$, $\hat{\boldsymbol{y}}$ and $\boldsymbol{x}$, $\boldsymbol{y}$ are related through the number of bit errors t.

In this section, our goal is to determine the conditions in which we can detect an erroneous $D(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ result based on $G(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$. The key to detecting the error is as follows. We restrict the information written to be vectors with known weight. That is, $W(\boldsymbol{x})$ and $W(\boldsymbol{y})$ are known and we further assume them to be constants, w_x and w_y. We define an augmented version of equation (1) in which we *a priori* know the Hamming weights of $\boldsymbol{x}$ and $\boldsymbol{y}$:

$$\tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = \frac{1+\epsilon}{(1-\epsilon)^2}[(w_x+w_y)(1-\epsilon)+2n\epsilon]-\frac{2(1+\epsilon)G(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})}{(1-\epsilon)^2}. \tag{2}$$

It is useful to denote the difference in conductance measurements between the noisy and the noise-free case as

$$\Delta G(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = G(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) - G(\boldsymbol{x}, \boldsymbol{y}).$$

Similarly, let us define $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$, which is later used to characterize bit errors, as follows:

$$\begin{aligned}\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) &= \tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) - D(\boldsymbol{x}, \boldsymbol{y}) \\ &= \tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) - \tilde{D}(\boldsymbol{x}, \boldsymbol{y}) \\ &= -\frac{2(1+\epsilon)\Delta G(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})}{(1-\epsilon)^2},\end{aligned} \tag{3}$$

noting that $\tilde{D}(\boldsymbol{x}, \boldsymbol{y}) = D(\boldsymbol{x}, \boldsymbol{y})$ by definition.

For two vectors $\boldsymbol{x}$, $\boldsymbol{y}$, we define N_{00} to be the number of element pairs that have $x_i = y_i = 0$ for $i \in \{1, ..., n\}$ and similarly for N_{01}, N_{10} and N_{11}. Then by the definition of $G(\boldsymbol{x}, \boldsymbol{y})$ we have

$$G(\boldsymbol{x}, \boldsymbol{y}) = N_{11} + (N_{10} + N_{01})\tfrac{2\epsilon}{1+\epsilon} + N_{00}\epsilon,$$

and similar expression can be derived for $G(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$.

A. Single Error Analysis and Detection with Fixed Weight Vectors

We now calculate how $\Delta G(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ and $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ are affected by a single bit error, i.e. $t = 1$. Due to symmetry in the function f, there are only four different fundamental types of errors for a pair of elements x_i and y_i, $(0,0) \to (0,1)$, $(0,1) \to (0,0)$, $(0,1) \to (1,1)$ and $(1,1) \to (0,1)$ which we call them error types A, B, C, and D, respectively. All other error types are expressed in terms of these four error types. Each of these error types affects the value of $\{N_{00}, N_{01}, N_{10}, N_{11}\}$. For example, an error that changes $(0,0)$ into $(0,1)$ will increase N_{01} by 1, decrease N_{00} by 1 (and leave N_{10}, N_{11} intact). Table II lists the resulting values of $\Delta G(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ and $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ for each error type.

TABLE II: The 4 Types of Errors

Error Type	$(x_i, y_i) \to (\hat{x}_i, \hat{y}_i)$	$\Delta G(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$	$\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$
A	(0,0) → (0,1)	$\frac{2\epsilon}{1+\epsilon} - \epsilon$	$\frac{-2\epsilon}{1-\epsilon}$
B	(0,1) → (0,0)	$-\left(\frac{2\epsilon}{1+\epsilon} - \epsilon\right)$	$\frac{2\epsilon}{1-\epsilon}$
C	(0,1) → (1,1)	$-\left(\frac{2\epsilon}{1+\epsilon} - 1\right)$	$\frac{-2\epsilon}{1-\epsilon} - 2$
D	(1,1) → (0,1)	$\frac{2\epsilon}{1+\epsilon} - 1$	$\frac{2\epsilon}{1-\epsilon} + 2$

Lemma 1. *If one of $\hat{\boldsymbol{x}}$ or $\hat{\boldsymbol{y}}$ contains a single bit-error, i.e., $D(\boldsymbol{x}, \hat{\boldsymbol{x}}) + D(\boldsymbol{y}, \hat{\boldsymbol{y}}) = 1$, then we determine that $D(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) \neq D(\boldsymbol{x}, \boldsymbol{y})$ for known $W(\boldsymbol{x})$ and $W(\boldsymbol{y})$ if $0 < \epsilon < \frac{1}{3}$.*

Proof. If no error is present, we have $D(\boldsymbol{x}, \boldsymbol{y}) = D(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = \tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$, i.e $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = 0$. As a result, $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) \neq 0$ implies $D(\boldsymbol{x}, \boldsymbol{y}) \neq D(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$. For $0 < \epsilon < \frac{1}{3}$, each error type will have $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ assuming a non-integer value and in turns lead to non-integer $\tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ because $D(\boldsymbol{x}, \boldsymbol{y})$ is always an integer and

$\tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = D(\boldsymbol{x}, \boldsymbol{y}) + \Delta D(\boldsymbol{x}, \boldsymbol{y}, \hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$. A single bit error can thus be detected by first computing $\tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}})$ and checking whether it is an integer or not. ■

This process of error detection is later referred to as an *integer check*. When error-free, $\tilde{D}(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) = D(\boldsymbol{x}, \boldsymbol{y})$ provides the desired output.

B. Inversion Coding

The single error detection capability is an inherent property of the in-memory Hamming distance computation as described in the previous section and requires no coding if the Hamming weights of vectors are known. However, in most applications, knowledge about the Hamming weights are not *a priori* known. We use the following coding technique to force every vector to have the same Hamming weight, thus enabling single error detection for most applications [5].

Auxiliary Code 1. *We define an inversion encoding of the vector $\boldsymbol{x}$ to be $\boldsymbol{x}_c = [\boldsymbol{x}|\neg\boldsymbol{x}]$, where $\neg\boldsymbol{x}$ is the bitwise complement of $\boldsymbol{x}$ and $|$ denotes concatenation.*

With Auxiliary Code 1, we have $W_H(\boldsymbol{x}_c) = w_h = n$ for $\boldsymbol{x} \in \{0,1\}^n$. The notation $\boldsymbol{x}$ and $\boldsymbol{y}$ from this point throughout the paper is saved for general vectors with unknown weights and the corresponding $\boldsymbol{x}_c$ and $\boldsymbol{y}_c$ are used to denote the coded message with known weights $w_h = n$ constructed from Auxiliary Code 1. Based on Lemma 1, single error detection is achieved for the computation of $D(\hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$. After error correction, which is discussed in Section IV, the resulting $D(\boldsymbol{x}_c, \boldsymbol{y}_c)$ can be used to recover the desired output $D(\boldsymbol{x}, \boldsymbol{y})$ by noting the relation $D(\boldsymbol{x}_c, \boldsymbol{y}_c) = 2D(\boldsymbol{x}, \boldsymbol{y})$.

C. Multiple-Error Detection Capability

So far we have shown that single bit error can be detected with the help of Auxiliary Code 1 by checking whether $\tilde{D}(\hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$ is an integer or not. Next, we generalize the idea of an *integer check* to multiple errors.

Lemma 2. *If together $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ contain an odd number of bit-errors t, i.e., $D(\boldsymbol{x}, \hat{\boldsymbol{x}}) + D(\boldsymbol{y}, \hat{\boldsymbol{y}}) = t, t$ odd, then we determine that $D(\hat{\boldsymbol{x}}, \hat{\boldsymbol{y}}) \neq D(\boldsymbol{x}, \boldsymbol{y})$ for the following constraint on ϵ:*

$$0 < \epsilon < \frac{1}{2t+1}.$$

Proof. Define $\boldsymbol{e}_{i,p}, i \in \{1, ..., t\}, p \in \{\boldsymbol{x}_c, \boldsymbol{y}_c\}$ to be the error vectors corresponding to the i-th error in the vector indexed by p. For example, if the i-th bit flip is at the j-th position of $\boldsymbol{x}_c$, $\boldsymbol{e}_{i,\boldsymbol{x}_c}$ is the all zero vector with 1 at the j-th position and $\boldsymbol{e}_{i,\boldsymbol{y}_c}$ is an all zero vector. (We permit error at the j-th position in $\boldsymbol{y}_c$ but there will be another pair of error vectors corresponding to it.) We further define $\Delta D(\boldsymbol{e}_i)$ to be the change in output induced by $\boldsymbol{e}_{i,\boldsymbol{x}_c}$ and $\boldsymbol{e}_{i,\boldsymbol{y}_c}$, i.e., $\Delta D(\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{x}+\boldsymbol{e}_{i,\boldsymbol{x}_c}, \boldsymbol{y}+\boldsymbol{e}_{i,\boldsymbol{y}_c})$. The following relation is observed:

$$\begin{aligned}
&\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c) \\
&= \sum_{i=1}^{t} [\tilde{D}(\boldsymbol{x}_c + \boldsymbol{e}_{i,\boldsymbol{x}_c}, \boldsymbol{y}_c + \boldsymbol{e}_{i,\boldsymbol{y}_c}) - D(\boldsymbol{x}_c, \boldsymbol{y}_c)] \\
&= \sum_{i=1}^{t} \Delta D(\boldsymbol{e}_i).
\end{aligned}$$

Each $\Delta D(\boldsymbol{e}_i)$ can be viewed as the change of the output for a single bit error, thus assuming the same value as the last column in Table II. For $0 < \epsilon < \frac{1}{2t+1}$, $\tilde{D}(\hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$ is non-integer because any odd t choices from those values of $D(\boldsymbol{e}_i)$ will be summed to $\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$ where $0 < |\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)| \pmod 1 < 1$. Here and elsewhere, the operation $x \pmod 1$ is defined as the fraction part of x. As a result, any odd number of bit-errors can be detected by *integer check*. ■

We now present the result for an even t number of errors.

Lemma 3. *We denote the fraction of errors that are detectable as $r_d(t)$. If together $\hat{\boldsymbol{x}}$ and $\hat{\boldsymbol{y}}$ contain an even number of bit-errors t, i.e., $D(\boldsymbol{x}, \hat{\boldsymbol{x}}) + D(\boldsymbol{y}, \hat{\boldsymbol{y}}) = t, t$ even, and the channel parameter satisfies $0 < \epsilon < \frac{1}{2t+1}$, then*

$$r_d(t) = 1 - \frac{\binom{t}{t/2}}{2^t}, t \text{ even}.$$

Proof. For an even number of errors, some error patterns are undetectable. We define $t_{A,C}$ to be the number of errors with either type A or C and similarly for $t_{B,D}$. Notice that error type A and C have same non-integer parts in $\Delta D(\boldsymbol{e}_i)$ thus contribute equally to $\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$, so do error type B and D. The analysis on $t_{A,C}$ and $t_{B,D}$ thus covers all the possible error combinations. The error pattern is undetectable when $t_{A,C} = t_{B,D} = t/2$. By viewing this error detection problem as a fair-coin flipping problem in which $t/2$ heads and $t/2$ tails occur in t trials, we calculate the probability of undetectable pattern occurring to be $\binom{t}{t/2}/2^t$. When $t_{A,C} \neq t_{B,D}$, let $t' = |t_{A,C} - t_{B,D}|$. We have $|\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)| \pmod 1 = \frac{2\epsilon t'}{1-\epsilon}, 0 < t' \leq t$. With $0 < \epsilon < \frac{1}{2t+1}$, $0 < |\Delta D(\boldsymbol{x}_c, \boldsymbol{y}_c, \hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)| \pmod 1 < 1$ which leads to error detection. ■

IV. Coding for Error Correction

Now that we have a coding scheme for detecting errors in the calculation of $D(\hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$. Our final objective is to recover $D(\boldsymbol{x}_c, \boldsymbol{y}_c)$ correctly and then to recover $D(\boldsymbol{x}, \boldsymbol{y})$ (error correction). We first provide a technique that can narrow each errors to two positions and then propose code schemes that have the capability to further determine the exact location. Based on another technique of bit value reading, error correction can be achieved by knowing the location and value for each error.

A. Error Localization and Single Bit Value Reading

Our error detection is based on the recognition of non-integer $\tilde{D}(\hat{\boldsymbol{x}}_c, \hat{\boldsymbol{y}}_c)$. However, this doesn't uniquely determine the error type. For example, error type B and C both have non-integer part $\frac{2\epsilon}{1-\epsilon}$ but error type B increases $D(\boldsymbol{x}, \boldsymbol{y})$ by one where error type C decreases $D(\boldsymbol{x}, \boldsymbol{y})$ by one. In principle, we need bit-level information in order to recover $D(\boldsymbol{x}, \boldsymbol{y})$. This section provides methods to access bit-level information while still using the conductance measurement between two vectors.

1) Error Localization: To recover $D(\boldsymbol{x}_c, \boldsymbol{y}_c)$, we first need to know the location of each bit flip. This can be achieved by comparing the corrupted vector with other preset vectors and infer information from the results.

Claim 1. *Define L to be the set of vectors $\boldsymbol{l}_i \in \{0,1\}^{2n}, 1 \leq i \leq n$ whose all bits are one except the the i-th and the $(i+n)$-th bits. Also define $\mathbf{1}$ to be the all-1s vector with length $2n$. With $n+1$ pairwise measurements between $\hat{\boldsymbol{x}}_c$ and vectors in $L \cup \mathbf{1}$, we can narrow the location of each error to two positions, i and $i+n$.*

With the measurement of $G(\hat{\boldsymbol{x}}_c, \mathbf{1})$, we compute $\tilde{D}(\hat{\boldsymbol{x}}_c, \mathbf{1})$. We perform n measurements between $\hat{\boldsymbol{x}}_c$ and each of the vectors in L to get $G(\hat{\boldsymbol{x}}_c, \boldsymbol{l}_i)$. Then $\tilde{D}(\hat{\boldsymbol{x}}_c, \boldsymbol{l}_i)$ is computed for all $1 \leq i \leq n$. Error localization is achieved by computing $\Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{l}_i) = \tilde{D}(\hat{\boldsymbol{x}}_c, \boldsymbol{l}_i) - \tilde{D}(\hat{\boldsymbol{x}}_c, \mathbf{1})$ for all $1 \leq i \leq n$. Let $I_{error} = \{i \in I | \Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{l}_i) = \frac{4\epsilon}{1-\epsilon} + 4\} \cup \{i \in I | \Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{l}_i) = \frac{4\epsilon}{1-\epsilon}\}$. There is an error either at position i or at position $i+n$ for each $i \in I_{error}$.

By Auxiliary Code 1, for a given $\boldsymbol{l}_i$, if both the i-th and the $(i+n)$-th positions of $\hat{\boldsymbol{x}}_c$ are error-free, the error patterns are (1,1) $\rightarrow$ (0,1) and (0,1) $\rightarrow$ (0,0) which result in $\Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{d}_i) = \frac{2\epsilon}{1-\epsilon} + 2$. If a bit error occurs at either the i-th or the $(i+n)$-th position of $\hat{\boldsymbol{x}}_c$, then the corresponding error patterns are two (1,1) $\rightarrow$ (0,0) which lead to $\Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{d}_i) = \frac{4\epsilon}{1-\epsilon} + 4$ for bit flip from 1 to 0 or two (0,1) $\rightarrow$ (0,0) with $\Delta D_i(\hat{\boldsymbol{x}}_c, \mathbf{1}, \hat{\boldsymbol{x}}_c, \boldsymbol{d}_i) = \frac{4\epsilon}{1-\epsilon}$ for bit flip from 0 to 1. If errors occur at both the i-th or the $(i+n)$-th positions of $\hat{\boldsymbol{x}}_c$, we are unable to localize those two errors. These error patterns are analyzed in later section.

2) Bit Value Reading: Since $D(\boldsymbol{x}_c, \boldsymbol{y}_c)$ depends on both vectors, $\boldsymbol{x}_c$ and $\boldsymbol{y}_c$, we also need to have knowledge of the corresponding bit values in $\hat{\boldsymbol{y}}_c$. Next we propose a method to infer bit value in vector, i.e., $\hat{\boldsymbol{y}}_c$, using measurements between vectors.

Claim 2. *Define B to be the set of vectors $\boldsymbol{b}_i \in \{0,1\}^{2n}, 1 \leq i \leq 2n$ whose bits are all one except the i-th bit. The i-th bit value of a given vector $\hat{\boldsymbol{y}}_c$ can be inferred from two pairwise measurements between, $\hat{\boldsymbol{y}}_c$ and $\boldsymbol{b}_i$; $\hat{\boldsymbol{y}}_c$ and $\mathbf{1}$.*

Two measurements, $G(\hat{\boldsymbol{y}}_c, \mathbf{1})$ and $G(\hat{\boldsymbol{y}}_c, \boldsymbol{b}_i)$, are taken and the corresponding $\tilde{D}(\hat{\boldsymbol{y}}_c, \mathbf{1})$ and $\tilde{D}(\hat{\boldsymbol{y}}_c, \boldsymbol{b}_i)$ are computed. The inference of the i-th bit value in $\hat{\boldsymbol{y}}_c$ is based on $\Delta D_i(\hat{\boldsymbol{y}}_c, \mathbf{1}, \hat{\boldsymbol{y}}_c, \boldsymbol{b}_i) = \tilde{D}(\hat{\boldsymbol{y}}_c, \boldsymbol{b}_i) - \tilde{D}(\hat{\boldsymbol{y}}_c, \mathbf{1})$. For $y_{c,i} = 1$, the error pattern is (1,1) $\rightarrow$ (1,0) with $\Delta D_i(\hat{\boldsymbol{y}}_c, \mathbf{1}, \hat{\boldsymbol{y}}_c, \boldsymbol{b}_i) = \frac{2\epsilon}{1-\epsilon} + 2$ and for $y_{c,i} = 0$, the error pattern is (0,1) $\rightarrow$ (0,0) with $\Delta D_i(\hat{\boldsymbol{y}}_c, \mathbf{1}, \hat{\boldsymbol{y}}_c, \boldsymbol{b}_i) = \frac{2\epsilon}{1-\epsilon}$.

B. Multiple Parity Check Codes for Error Correction

Using error localization and bit value reading, the two possible error locations can be computed for a vector (say $\boldsymbol{x}_c$), as well as the value at the corresponding positions of the other vector (say $\boldsymbol{y}_c$). We then propose a code to reconcile the ambiguity between the two positions. The goal of this code is to correct a single bit error but we also leave rooms for multiple error correction by design. We note that the exact location for each error can be determined by an error detection code that contains one of the locations.

Auxiliary Code 2. *For vector $\boldsymbol{x} \in \{0,1\}^n$, the Multiple Parity Check Coding with parameter n_r (n_r divides n) is constructed as $[\boldsymbol{x}|r(\boldsymbol{x})]$ where $r(\boldsymbol{x}) \in \{0,1\}^{n_r}$ and the i-th element of $r(\boldsymbol{x})$ is $r(\boldsymbol{x})_i = \sum_{k=n(i-1)/n_r+1}^{ni/n_r} x_k$ for all $1 \leq i \leq n_r$ with modulo 2 summation.*

This code can be interpreted as a single parity check code for each block $x_{n(i-1)/n_r+1}, ..., x_{ni/n_r}$ with parity bit $r(\boldsymbol{x})_i$, thus single bit error is detectable per block. If this code is applied on $\boldsymbol{x}$ (the first half of $\boldsymbol{x}_c$), the exact location of the error is determined and the error is corrected. In order to read bit values from $r(\boldsymbol{x})$, we also append the inverse of $r(\boldsymbol{x})$.

Code 1. *To achieve error correction, we encode vector $\boldsymbol{x} \in \{0,1\}^n$ to be $[\boldsymbol{x}|\neg\boldsymbol{x}|r(\boldsymbol{x})|\neg r(\boldsymbol{x})]$ where $r(\boldsymbol{x})$ is defined in Auxiliary Code 2. This encoding of $\boldsymbol{x}$ is denoted as $c(\boldsymbol{x})$.*

This encoding is equivalent to $[\boldsymbol{x}_c|r(\boldsymbol{x}_c)]$ where $\boldsymbol{x}_c$ is defined in Auxiliary Code 1. The caveat here is that although $c(\boldsymbol{x}) = [\boldsymbol{x}_c|r(\boldsymbol{x})|\neg r(\boldsymbol{x})]$ is the codeword stored in the resistive memory, only $\boldsymbol{x}_c$ is used in nominal operation of in-memory Hamming distance computation (the parity parts $r(\boldsymbol{x})$ are only read when an error is already detected). As a result, error correction capability requires the ability to measure the conductance between two sub-vectors in the resistive memory (assumed feasible for this subsection). The necessity of this sub-vector measurement is shown in Lemma 4.

Lemma 4. *Let $c(\boldsymbol{x}) = [\boldsymbol{x}_c|r(\boldsymbol{x}_c)]$ and $c(\boldsymbol{y}) = [\boldsymbol{y}_c|r(\boldsymbol{y}_c)]$ where r is arbitrary an encoding that map $\boldsymbol{x}_c$, $\boldsymbol{y}_c$ to $r(\boldsymbol{x}_c)$, $r(\boldsymbol{y}_c)$. For $dim(r(\boldsymbol{x}_c)) < dim(\boldsymbol{x}_c)$, $D(c(\boldsymbol{x}), c(\boldsymbol{y})) \neq f(D(\boldsymbol{x}, \boldsymbol{y}))$ for any bijective function f.*

Proof. This lemma is proved by contradiction. Assume there exist r and f such that $D(c(\boldsymbol{x}), c(\boldsymbol{y})) = f(D(\boldsymbol{x}, \boldsymbol{y}))$. By $D(\boldsymbol{x}_c, \boldsymbol{y}_c) = 2D(\boldsymbol{x}, \boldsymbol{y})$, we have

$$D(r(\boldsymbol{x}_c), r(\boldsymbol{y}_c)) = f(D(\boldsymbol{x}_c, \boldsymbol{y}_c))$$

Now consider the case where $\boldsymbol{x} = \boldsymbol{y}$, we have $0 = f(0)$ because $\boldsymbol{x}_c = \boldsymbol{y}_c$ and $r(\boldsymbol{x}_c) = r(\boldsymbol{y}_c)$. Then since $dim(r(\boldsymbol{x}_c)) < dim(\boldsymbol{x}_c)$, there exists $r(\boldsymbol{x}_c) = r(\boldsymbol{y}_c)$, with $\boldsymbol{x}_c \neq \boldsymbol{y}_c$. This pair of $\boldsymbol{x}$ and $\boldsymbol{y}$ lead to $0 = f(D(\boldsymbol{x}_c, \boldsymbol{y}_c))$ with $D(\boldsymbol{x}_c, \boldsymbol{y}_c) \neq 0$, which contradicts the notion that f is a bijective function. ■

Because of Lemma 4, in the nominal error-free case, the desired output $D(\boldsymbol{x}, \boldsymbol{y})$ can not be computed from $D(c(\boldsymbol{x}), c(\boldsymbol{y}))$. If only $\boldsymbol{x}_c$ and $\boldsymbol{y}_c$ are used for nominal operation, by noting $D(\boldsymbol{x}_c, \boldsymbol{y}_c) = 2D(\boldsymbol{x}, \boldsymbol{y})$, the desired output can be computed.

C. Multiple Error Correction Capability

We have shown Code 1 can correct a single error, now we present the multiple error correct capability of Code 1. We only consider the case that all t errors occur in $\boldsymbol{x}_c$. At the error localization step, given t errors in $\boldsymbol{x}_c$, t pairs of possible error

locations can be correctly located except for the specific case in which two errors are exactly n apart. Denote the fraction of error patterns that we can correctly localize by $r_l(t)$,

$$r_l(t) = \frac{\prod_{k=0}^{t-1} 2n-2k}{\prod_{k=0}^{t-1} 2n-k}, \quad t < n_r. \tag{4}$$

After the location of each error is narrowed down to two locations, the exact location can be determined using the parity bits in $r(\boldsymbol{x})$. However, Auxiliary Code 2 only provides single error detection in each block, i.e., if two or more errors occurs in the same block, we are unable to resolve the ambiguity of the two locations and error correction can not be done. In $\boldsymbol{x}_c \in \{0,1\}^{2n}$, the subvectors $x_{c,n(i-1)/n_r+1}, ..., x_{c,ni/n_r}$ and $x_{c,n(i-1)/n_r+1+n}, ..., x_{c,ni/n_r+n}$ are considered to be in the same block for $1 \leq i \leq n_r$ since they both rely on the parity bit $r(\boldsymbol{x})_i$. Define $r_c(t)$ to be the fraction of t error patterns that are correctable given that the errors can be detected and localized. We have

$$r_c(t) = \frac{\prod_{k=0}^{t-1}(2n - \frac{2nk}{n_r})}{\binom{2n}{t} t!}, t < n_r \tag{5}$$

Combining the discussions above with the $r_d(t)$ in section III.C and define $r_d(t) = 1$ for odd t, we have the following claim.

Claim 3. *Suppose t bit errors occurs in $\boldsymbol{x}_c$ and $r(\boldsymbol{x}_c)$ is error-free, define $\bar{R}(t)$ to be the fraction of all error patterns that we can correctly recover the original $D(\boldsymbol{x}, \boldsymbol{y})$. We have:*

$$\bar{R}(t) = r_d(t) \times r_l(t) \times r_c(t).$$

$\bar{R}(t)$ can be viewed as the probability of recovery given t errors in $\boldsymbol{x}_c$ and no error in $r(\boldsymbol{x})$. Our goal is to find the probability of recovery given t errors in the whole sequence $c(\boldsymbol{x})$, denoted as $R(t)$. We have

$$\begin{aligned} R(t) = &\sum_{k=0}^{t} \bar{R}(t-k) \times \\ &P\{t-k \text{ errors in } \boldsymbol{x}_c \cap k \text{ errors in } r(\boldsymbol{x}_c)\} \times \\ &P\{k \text{ errors in } r(\boldsymbol{x}_c) \text{ not used for correction}\} \end{aligned} \tag{6}$$

where we define $\bar{R}(0) = 1$. The latter two probabilities, denoted as $P_1(n, n_r, t, k)$ and $P_2(n_r, t, k)$, can be computed as

$$P_1(n, n_r, t, k) = \frac{\binom{2n}{t-k}\binom{2n_r}{k}}{\binom{2n+2n_r}{t}},$$

and

$$P_2(n_r, t, k) = \sum_{j=0}^{k} \frac{\binom{n_r}{j}\binom{n_r-t+k}{k-j}}{\binom{2n_r}{k}}.$$

For example, when $n = 64, n_r = 8$, we can calculate $R(2) = 0.5435$ and $R(3) = 0.5932$ which means our code can correct more than half of double and triple errors. The multiple error correction capability provided by our proposed code will make in-memory Hamming distance computation more robust.

D. Benefits on a kNN Classifier

We test our coding scheme on the simple application of digit recognition using k-nearest neighbor classifier. The testing and training data sets are 64-bit bitmap images processed from computer generated digits. We corrupt both the testing and training sets by implementing a BSC channel with various crossover probability p. The kNN classifier performs classification using Hamming as the distance metric with and without the protection of our code. Experimental results have shown the same performance of recognition under 3 times higher crossover probability with the protection of our code. This preliminary result shows that our code is promising for error-tolerant in-memory Hamming distance computation application.

ACKNOWLEDGMENT

Research supported in part by a grant from UC MEXUS and an NSF-BSF grant no.1718389.

V. CONCLUSION

This paper provides coding schemes for error detection and error correction in order to facilitate reliable in-memory Hamming distance computation under write noise due to power constrains. Future efforts includes the analysis when the channel parameter ϵ is not a fixed constant and extended analysis on other machine learning algorithms.

REFERENCES

[1] C. P. Chen and C.-Y. Zhang, "Data-intensive applications, challenges, techniques and technologies: A survey on big data," *Inf. Sciences*, vol. 275, pp. 314–347, Aug. 2014.

[2] S. Hamdioui *et al.*, "Memristor based computation-in-memory architecture for data-intensive applications," in *Proc. DATE*, Grenoble, France, Mar. 2015, pp. 1718–1725.

[3] B. Kulis and T. Darrell, "Learning to hash with binary reconstructive embeddings," in *Proc. NIPS*, Vancouver, Canada, Dec. 2009, pp. 1042–1050.

[4] M. Norouzi *et al.*, "Fast search in hamming space with multi-index hashing," in *Proc. IEEE CVPR*, Providence, RI, July 2012, pp. 3108–3115.

[5] Y. Cassuto and K. Crammer, "In-memory hamming similarity computation in resistive arrays," in *Proc. IEEE ISIT*, Hong Kong, China, June 2015, pp. 819–823.

[6] P. O. Vontobel *et al.*, "Writing to and reading from a nano-scale crossbar memory based on memristors," *Nanotechnology*, vol. 20, no. 42, p. 425204, Sep. 2009.

[7] A. A. Adeyemo *et al.*, "Exploring error-tolerant low-power multiple-output read scheme for memristor-based memory arrays," in *Proc. IEEE DFT*, Amherst, MA, Nov. 2015, pp. 17–20.

[8] W. Yi *et al.*, "Feedback write scheme for memristive switching devices," *Appl. Phys. A: Materials Science & Processing*, vol. 102, no. 4, pp. 973–982, Jan. 2011.

[9] D. Niu *et al.*, "Low power memristor-based reram design with error correcting code," in *Proc. IEEE ASP-DAC*, Sydney, Australia, Jan./Feb. 2012, pp. 79–84.

[10] G. Medeiros-Ribeiro *et al.*, "Lognormal switching times for titanium dioxide bipolar memristors: origin and resolution," *Nanotechnology*, vol. 22, no. 9, p. 095702, Jan. 2011.