# SpREM: Exploiting Hamming Sparsity for Fast Quantum Readout Error Mitigation

Hanyu Zhang[1], Liqiang Lu[1*], Siwei Tan[1], Size Zheng[2], Jia Yu[1], Jianwei Yin[1*]
[1]Zhejiang University [2]Peking University
Email: {hyzz, liqianglu, siweitan, yujia_cs, zjuyjw}@zju.edu.cn zhengsz@pku.edu.cn

## ABSTRACT

The current Noisy Intermediate-Scale Quantum (NISQ) era suffers from high quantum readout error that severely reduces the measurement fidelity. Matrix-based error mitigation has been demonstrated as a promising software-level technique, which performs matrix-vector multiplication to calibrate the probability distribution with noise. However, this approach shows poor scalability and limited fidelity improvement as the matrix size exponentially increases with the number of qubits. In this paper, we propose SpREM to exploit the inherent sparsity in the mitigation matrix. Inspired by the interaction mechanism between qubits, we identify structured sparsity patterns using Hamming distance. With this insight, we propose the Hamming-Distance Sparse Row (HDSR) compression method and its format, which can achieve higher sparsity than threshold-based pruning meanwhile exhibiting great fidelity improvement. Finally, we propose the computational dataflow of the HDSR format and implement it on hardware. Experiments demonstrate that SpREM achieves 98.9% sparsity and a 27.3× reduction in fidelity loss on the real-world quantum device, compared to threshold-based pruning. It achieves an average 11.2× ~ 36.4× speedup compared to Xilinx Vitis SPARSE library and NVIDIA A100 GPU implementations.

## KEYWORDS

quantum computing, accelerator, quantum error mitigation

**ACM Reference Format:**
Hanyu Zhang[1], Liqiang Lu[1*], Siwei Tan[1], Size Zheng[2], Jia Yu[1], Jianwei Yin[1*]
. 2024. SpREM: Exploiting Hamming Sparsity for Fast Quantum Readout Error Mitigation. In *61st ACM/IEEE Design Automation Conference (DAC '24), June 23–27, 2024, San Francisco, CA, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3649329.3655675

## 1 INTRODUCTION

Quantum computers hold the potential to significantly speed up the solving of certain problems when compared to classical computers, such as integer factorization and machine learning [20]. However, we are currently in the Noisy Intermediate Scale Quantum (NISQ) stage, suffering from gate errors and readout errors, e.g., qubit flip, where the error rate in quantum computers typically ranges from 0.1% to 10%. Remarkably, readout errors stand out as a primary contributor, exhibiting error rates in the range of 2% to 7% [19] on superconducting quantum computers.

To deal with readout errors, quantum error mitigation can be achieved through pre-processing methods, such as machine learning [9] and statistical analysis [15, 17] methods. The matrix-based mitigation method is a well-known and widely applied that utilizes matrix-vector multiplication to map a probability distribution with noise to a calibrated one. However, with the increasing number of quantum bits (qubits) in quantum computer, the size of the mitigation matrix grows exponentially. For example, a 20-qubit mitigation matrix requires 8 TB of memory, which far exceeds the typical computer's available memory capacity. Additionally, relying on machine learning method [9], the number of input and output nodes of the network grow exponentially with the number of qubits. Moreover, the statistical analysis method Q-BEEP [15] necessitates continuous iterations to enhance the effectiveness of mitigation, which demands a substantial amount of time. When mitigating a 16-qubit GHZ [7] algorithm, the mitigation process surpasses 1 hour. As a result, the application of these methods in large-scale quantum circuits is constrained by limitations in both time and space.

To accelerate the mitigation process, certain methods decompose the mitigation matrix into multiple sub-matrices and mitigate calculations through tensor products [3, 12]. However, these sub-matrices neglect the crosstalk between qubits, which severely reduces the measurement fidelity. Another possibility for memory reduction and performance acceleration is to exploit the sparsity in the mitigation matrix, as the mitigation matrix itself contains a considerable amount of near-zero-value quantum states. The challenge is that the sparsity follows a structured pattern determined by the quantum interaction mechanism, making standard threshold-based pruning ineffective. For example, a pruning threshold of $1 \times 10^{-6}$ leads to 98.6% sparsity in a 16-qubit mitigation matrix. Whereas, such pruning results in more than a 20× fidelity loss.

In this paper, we propose a hardware-software cooperative approach called SpREM to exploit the inherent sparsity in the mitigation matrix. The key insight of SpREM is that the values in the mitigation matrix decrease exponentially with Hamming distance (the number of qubit flips). Building on this observation, we introduce a pruning method with a structured pattern derived from Hamming distance. Then, leveraging the non-zero value distribution characteristics, we present a sparse format called Hamming-Distance Sparse Row (HDSR). This format exhibits less memory requirement as both row index and column index can be calculated according to Hamming distance, i.e., only need to store the non-zero values. Ultimately, based on HDSR sparse format, we design the sparse matrix-vector multiplication (SpMV) dataflow and its hardware architecture, which effectively eliminates the computation of zero output quantum state.

The main contributions of this paper are summarized as follows:

[*]Corresponding author

*DAC '24, June 23–27, 2024, San Francisco, CA, USA*

ACM ISBN 979-8-4007-0601-1/24/06…$15.00
https://doi.org/10.1145/3649329.3655675

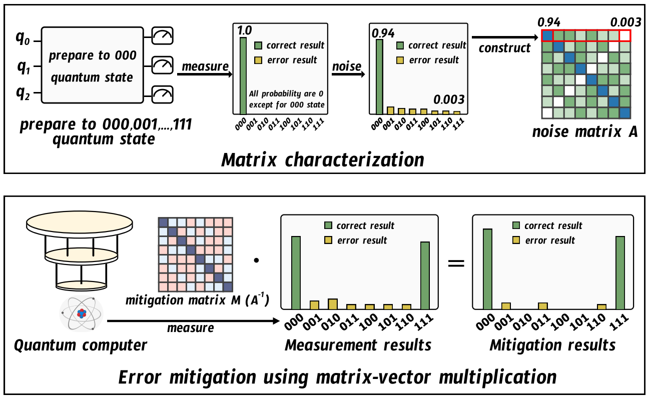


**Figure 1: Matrix-based readout error mitigation process**

- We identify the structured sparse pattern of the mitigation matrix using Hamming distance, which provides higher sparsity and greatly improves the fidelity.
- We propose a method of compressing the mitigation matrix based on the Hamming distance. We design the corresponding Hamming-Distance Sparse Row (HDSR) sparse format that achieves a high compression ratio.
- We propose a SpMV dataflow to accelerate readout error mitigation. The architecture utilizes an efficient XOR array and a reconfigurable adder tree to cooperate with HDSR features.

Compared to threshold-based pruning method, SpREM demonstrates 98.9% sparsity and achieves a 27.3× reduction in fidelity loss on real-world quantum device. SpMV dataflow is implemented on the Xilinx Alveo U50 platform, which shows 11.2× ∼ 36.4× speedup compared to Xilinx Vitis SPARSE library and NVIDIA A100 GPU implementations.

## 2 BACKGROUND

### 2.1 Quantum Readout Noise

In quantum computing, noise can be categorized based on the stage of occurrence: initialization noise, quantum gate noise and readout noise. Readout noise constitutes the primary source of interference in current superconducting quantum computers, with its generation outlined as follows:

(1) **Imperfect discriminator**: current discriminators are ineffective, with the cumulative accuracy of state-of-the-art discriminator only reaching 92.66% [11].
(2) **Readout latency**: qubit decoherence typically manifests as exponential decay over time. The readout operation features high latency, approximately taking 400 ns on Google Sycamore [1], thereby contributing to the decay towards the ground state.
(3) **Crosstalk**: quantum computers employ frequency division multiplexing technology to read resonant cavity frequency. However, this can lead to increased crosstalk between qubits.

### 2.2 Matrix-based Error Mitigation

Figure 1 illustrates the complete matrix-based mitigation process, delineated into two primary steps: matrix characterization and error mitigation using matrix-vector multiplication. From the purely classical noise models perspective, the overall effect of readout noise transfers the ideal possibility distribution $\vec{p}_{ideal}$ to the noisy possibility distribution $\vec{p}_{measure}$, formulated as:

$$\vec{p}_{measure} = A\vec{p}_{ideal} \tag{1}$$

where the noise matrix $A$ is a $2^n \times 2^n$ matrix obtained through $2^n$ readout-error measurements, referred to as matrix characterization. Figure 1 shows an example of constructing the first row of the noise matrix. This process involves preparing the '000' quantum state and measuring its probability. Ideally, all measurement outcomes should be '000' state. However, under the influence of readout noise, the probability of obtaining '000' is reduced to 0.94, and instances of errors, such as '111', occur with a probability of 0.003. This probability distribution with noise constructs the initial row of the noise matrix.

To mitigate readout noise, we can apply mitigation matrix $M$ (the inverse matrix of noise matrix $A$) through matrix-vector multiplication, which is the inverse process of Equation (1):

$$\vec{p}_{mitigate} = M\vec{p}_{measure} \tag{2}$$

For example, we diminish the probability of incorrect outcomes '001', ..., '101' and '110', while simultaneously enhancing the probability of the correct outcomes '000' and '111' in Figure 1. Typically, the mitigation matrix is a device-dependent property that characterizes the correlation with the environmental conditions and hardware defects. Therefore, the mitigation matrix $M$ can be applied in various quantum algorithms and serve as prior knowledge in Equation (2).

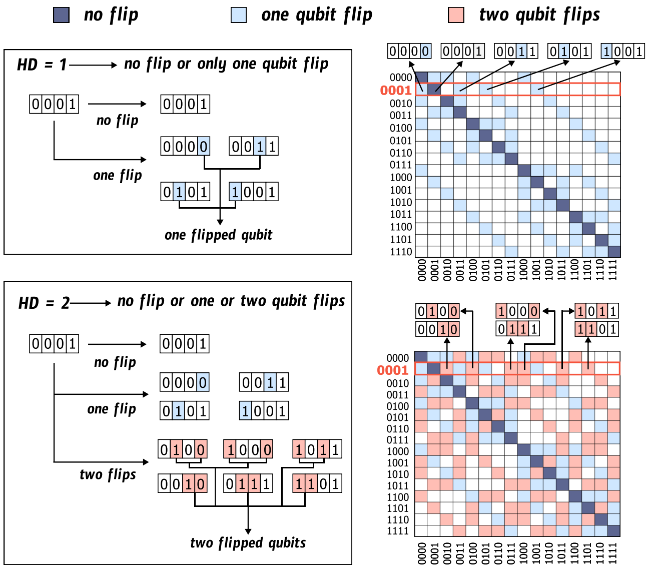


**Figure 2: Compressing mitigation matrix**

## 3 COMPRESSION USING HAMMING DISTANCE

### 3.1 Hamming Sparsity

In quantum computing, Hamming distance serves as a metric to quantify the disparity between quantum states, reflecting the number of qubits undergoing flips. A larger Hamming distance implies a greater disparity between two quantum states, indicating a higher frequency of flips. In this paper, we consider the infidelity of the output quantum state as flip errors. Therefore, Hamming distance is employed to quantify the probability of flip errors occurring.

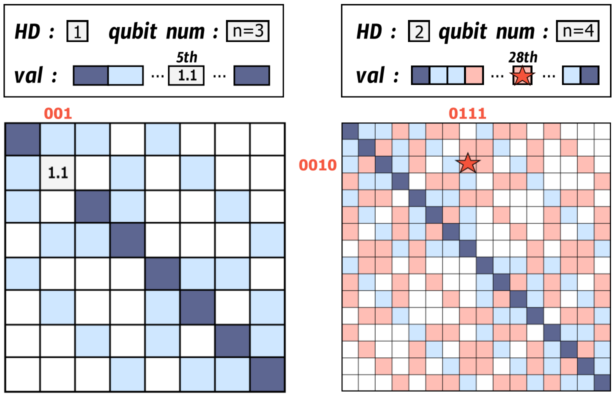


**Figure 3: HDSR sparse format**

Figure 2 shows two examples in a 4-qubit system, where $HD = 1$ means there is no flip or only one qubit flip, and $HD = 2$ means no flip or one or two flips. Clearly, not flipping or flipping one qubit in '0001' quantum state results in five states '0000', '0001', '0011', '0101' and '1001'. When $HD = 2$, we need to flip zero, one or two qubits in '0001' to yield '0000', '0001', '0010', '0011', ... , '1011' and '1101'. Given an ideal quantum state, the Hamming distance determines the number of flipped states $N_{nz}$ by the following formula:

$$N_{nz} = \sum_{i=0}^{HD} C_n^i = \sum_{i=0}^{HD} \frac{n!}{i!\,(n-i)!} \tag{3}$$

where $i = 0$ means no qubit flip.

Given a hamming distance, our compression method is to preserve the value of all flipped states and prune the rest. In other words, each row in the mitigation matrix only has $N_{nz}$ non-zero values. The column index of each non-zero value is calculated from the binary of the flipped states. As shown in Figure 2, the ideal state '0001' has five flipped states '0000', '0001', '0011', '0101' and '1001', retaining the values corresponding to column indices 0, 1, 3, 5, 9. Similarly, $HD = 2$ will preserve more values, which means a larger Hamming distance will lead to lower sparsity. Generally, the probability of a specific readout error exponentially reduces with the number of qubit flips, signifying a very low probability of encountering many flips. For example, the probability of four qubit flips is less than $10^{-5}$. In summary, we can easily prune redundant values by setting the Hamming distance threshold $HD$, balancing the trade-off between mitigation latency and mitigation accuracy.

## 3.2 HDSR Sparse Format

In order to reduce index decoding overheads, we introduce a sparse matrix format called Hamming-Distance Sparse Row (HDSR) that fully utilizes the characteristics of Hamming sparsity. HDSR format consists of one one-dimensional array $val$ and two scalars $n$ and $HD$. The $val$ vector stores all non-zero values. Two scalars $n$ and $HD$ store the number of qubits and the Hamming distance threshold, respectively. Compared to conventional COO or CSR sparse format, there's no need to store the column index or row index for every non-zero value. Instead, we can leverage the Hamming distance threshold to compute these indices, resulting in significant memory conservation. Concretely, for the $k^{th}$ value, its row index and column index can be calculated as follows,

$$\begin{aligned} row\_idx &= \lfloor \frac{k}{N_{nz}} \rfloor \\ re &= k \mod N_{nz} \\ col\_idx &= flipfunc(HD, n, row\_idx, re) \end{aligned} \tag{4}$$

where $N_{nz}$ can be obtained from Equation (3). $flipfunc$ is the flip function that gives the column index for the flipped state. Clearly, as illustrated in Figure 2, given an ideal quantum state and Hamming distance threshold, we can enumerate all possible flipped states and sort them according to the binaries. Thus, in Equation 4, the row index represents the ideal state, and $flipfunc$ output the $re^{th}$ flipped state, i.e., the column index.

Figure 3 illustrates two examples of HDSR sparse format: one for a 3-qubit mitigation matrix with $HD = 1$ and another for a 4-qubit mitigation matrix with $HD = 2$. For the 3-qubit mitigation matrix, we can calculate that the number of non-zero values in each row is 4 based on Equation (3). Dividing the array indices of the 5th element (1.1) in the $val$ array by 4 provides its row index 1 (001) in the mitigation matrix, with a remainder of 1, indicating that it is the 1st non-zero element in this row. Then, according to Equation (4), the column index is 1 (001). For the 4-qubit mitigation matrix, we mark the 28th element in the $val$ array with a red star. Similarly, dividing by $N_{nz}$ (11) yields a row index of 2 (0010), with a remainder of 6, indicating that it is the 6th non-zero element in this row. Not flipping, flipping one or two bits on the binary representation '0010' and sorting results yield '0000', '0001', '0010', ..., '1110', we can identify that the column index is 7 (0111).

Instead of directly pruning the matrix using a threshold, SpREM considers the mechanism of qubit flip and leverages the characteristics of Hamming distance. Specifically, our method shows two advantages:

- **Higher sparsity and higher accuracy.** Fundamentally, the real-world mitigation matrix is derived from a large amount of measurement to profile each value. To go further, the mitigation matrix may involve some values that cannot perfectly model the transfer of quantum states due to limited profiling. On the other hand, our method chooses to theoretically locate the non-zeros and quantify the qubit interactions with different Hamming distances, therefore, achieving higher sparsity and higher accuracy.
- **Structured sparsity and higher compression ratio.** According to the definition of Hamming distance, each row in the pruned matrix has the same number of non-zeros, which makes it easy to conduct parallel computing with a balanced workload. Moreover, HDSR format only needs to store two scalars and one vector of values, leading to an exponential compression ratio (e.g., from $O(2^{2n})$ to $O(n2^n)$ when $HD = 1$).

# 4 ARCHITECTURE DESIGN

## 4.1 HDSR Dataflow

Algorithm 1 presents the pseudocode for the HDSR-based sparse matrix-vector multiplication (SpMV) algorithm. The algorithm has a total of four inputs and one output. In addition to the one array $val$ and two scalars $HD$ and $n$ in HDSR sparse format, the array $meas$ signifies the raw data of measured probability. The output array $miti$ represents the probability after readout error mitigation.

The entire computational process consists of four steps. Firstly, we need to calculate the number of non-zero elements in each row using $HD$ and $n$ based on Equation (3) (line 2). And then we initialize

**Algorithm 1:** HDSR Dafaflow

```
   Data: $HD$, $n$, $val[]$, $meas[2^n]$
   Result: $miti[2^n]$
1  //Calculate the number of non-zero elements in each row
2  int $N_{nz} = \sum_{i=0}^{HD} \frac{n!}{i!(n-i)!}$;
3  //Implement flip function: get flip counts
4  int flip_array [$N_{nz}$], t=0;
5  for ($int\ i = 0; i < 2^n; i++$){
6      if (# of '1' in binary(i) <= HD) {
7          $flip\_array[t] = i$;
8          t = t+1;
9  }} //Implement flip function: get column index
10 int col_idx [$N_{nz}$];
11 for ($int\ i = 0; i<2^n; i++$){
12     for ($int\ j = 0; j < N_{nz}; j++$){
13         $col\_idx[j] = flip\_array[j] \oplus i$;
14     }
15     col_idx.sort(); //in ascending order
16 //Perform SpMV
17     for ($int\ k = 0; k < N_{nz}; k++$){
18         $miti[i] += val[i * N_{nz} + k] * meas[col\_idx[k]]$;
19 }}
```

a flip array by selecting the binary strings, of which the number of '1' is less than or equal to the Hamming distance threshold (lines 4-9). For example, the 4-qubit flip array with $HD = 1$ is:

$$flip\_array = [0000, 0001, 0010, 0100, 1000]$$

Next, we need to find the column indices of non-zero values in each row by performing XOR calculations between the row index $i$ and $flip\_array$ (lines 12-14). For example, considering an ideal state '0001', the valid flips with $HD = 1$ and their column indices can be obtained as follows.

$$\begin{aligned} valid\_flip &= [0000, 0001, 0010, 0100, 1000] \oplus 0001 \\ &= [0001, 0000, 0011, 0101, 1001] \\ col\_idx &= [1, 0, 3, 5, 9] \end{aligned}$$

After sorting the column indices in ascending order, we carry out element-wise multiplication between the corresponding values in $val$ and $meas$, followed by the accumulation of results into $miti$ (lines 17-19).

## 4.2 Accelerator Architecture Design

We design a hardware accelerator to implement the SpMV in Algorithm 1, mainly consisting of two parts: transformation unit and reconfigurable adder tree. Though the mitigation matrix is device-dependent, it varies at different time points (e.g., environmental noise) and with different qubits usage (e.g., not all qubits are utilized in a chip). Thus, we regard the mitigation matrix, compressed in HDSR format, as initially stored in off-chip memory. In a word, all input data in Algorithm 1 are stored in the external memory at the beginning. Since SpMV is a communication-intensive operator, the double buffer is applied to overlap the communication with computation.

Figure 4 depicts the key component of our architecture that generates column index and performs SpMV. To acquire the column index, we introduce an XOR array to parallel the loop in line 12 of Algorithm 1, which conducts XOR operation between every row index and the static flip array in ROM. The sort function is implemented via a comparator. As the measurement often contains many zero-probability quantum states, the comparator not only sorts the column indices but also detects zeros in the measurement vector. For example, in Figure 4 we perform XOR operations on row index 5 with the flip array, acquiring column index 5, 4, ... , 15 and 9, respectively. We sort them in ascending order, then fetch the corresponding readout possibility (0.02, 0, 0.02, ... , 0.01 and 0.01) and enter the compute unit.

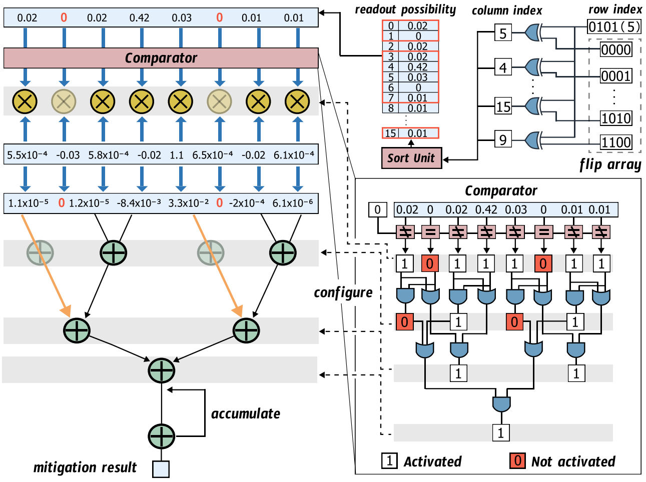


**Figure 4: SpREM architectural details**

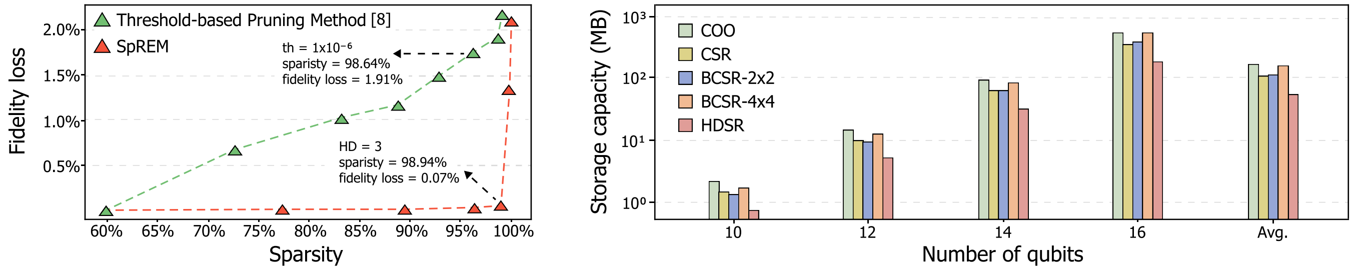
In the compute unit, we design a reconfigurable adder tree leveraging the sparsity of the measurement vector. Specifically, the comparator gets the readout probability and detects the zeros to configure adder tree and multiplier array through a series of AND gates and OR gates. Figure 4 also shows an example of the entire configuration process. The values of readout possibility are compared with 0, resulting in 1, 0, 1, 1, 1, 0, 1, 1. Consequently, we activate the multipliers except for the first and fifth ones. Subsequently, these results are pairwise to perform bitwise AND operations, yielding 0, 1, 0, 1. Therefore, we disable $0^{th}$ and $2^{nd}$ adders in the first layer. The rest layers of adders also employ the same method. Modules in our architecture design are pipelined to reduce latency.

# 5 EVALUATION

## 5.1 Experiments Setup

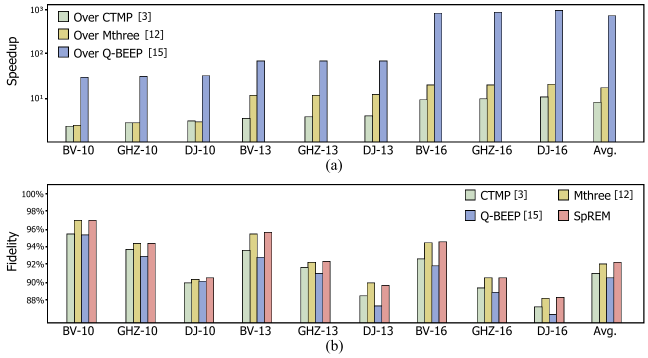
We evaluate SpREM on the Rigetti [14] quantum platform with a readout error rate ranging from 0.6% to 10.6%. We compare SpREM against state-of-the-art error mitigation techniques, including CTMP [3], Mthree [12] and Q-BEEP [15] using three well-known quantum algorithms: Bernstein-Vazirani (BV) [2], GHZ [7] and Deutsch-Jozsa (DJ) [6] algorithms. SpREM sets the $HD$ to 3 and Mthree uses a Hamming distance of 3. Meanwhile, Q-BEEP sets the update frequency to 20. These comparisons are performed on a server equipped with two AMD EPYC 2.25GHz 64-core CPUs and 1.6TB of memory, using a single thread for execution.

The SpREM architecture is implemented on the Xilinx Alveo U50 platform operating at 262 MHz. The baseline consists of GPU and FPGA accelerators. For GPU, we employ cuSPARSE [13] library and execute it on the NVIDIA A100 GPU. For FPGA, we use Vitis



(a) (b)

**Figure 5: (a): Comparison of fidelity loss between threshold-based pruning method [8] and SpREM (b): Comparison of storage capacity among COO, CSR, Block CSR and HDSR sparse formats**



(a)

(b)

**Figure 6: (a): Speedup of SpREM over CTMP [3], Mthree [12] and Q-BEEP [15] (b): Comparison of fidelity among CTMP [3], Mthree [12], Q-BEEP [15] and SpREM**

SPARSE library [21], running it on the Alveo U50 platform under 309MHz. The mitigation matrix is stored in off-chip memory in the corresponding sparse format.

## 5.2 Compression Quality

In Figure 5 (a), we compare SpREM against the threshold-based pruning method [8] on the 16-qubit BV circuit. We find that under the same sparsity level, SpREM has a lower fidelity loss than the threshold-based pruning method. With HD of 3 and a pruning threshold of $1 \times 10^{-6}$, their sparsity is 98.94% and 98.64%, respectively. However, the fidelity loss of the threshold-based pruning method is 27.3× greater than that of SpREM. This is because pruning based on Hamming distance, exclusively linked to the physical principles of qubit flips, results in more effective outcomes. Moreover, this point is also the optimal pruning point for SpREM. The fidelity loss changes gradually when the sparsity is below 98.94%; however, beyond this point, the fidelity loss increases sharply, exceeding 1%.

Figure 5 (b) shows the storage capacity required for the mitigation matrix in COO, CSR, Block CSR [4] and HDSR sparse formats under varying qubit numbers. Due to the streamlined storage requirements of HSDR, involving only one array and two scalars, it achieves a significantly higher compression ratio, averaging 3.0×, 2.0×, 2.1× and 2.8× compared to COO, CSR, Block CSR-2x2 and Block CSR-4x4 sparse format, respectively. Due to the irregular distribution of zero elements, a 4x4 Block CSR consumes more storage than a 2x2 Block CSR sparse format.

## 5.3 Comparison with Prior Methods

In Figure 6 (a), we compare SpREM against state-of-the-art readout error mitigation. On average, SpREM is 8.6×, 17.7× and 742.8× faster than CTMP [3], Mthree [12], and Q-BEEP [15], respectively. As the number of qubits increases, the sparsity of the pruned matrix also expands, allowing for the avoidance of more calculations involving zero values. Consequently, the speedup ratio continues to rise. CTMP accelerates computation speed by decomposing the mitigation matrix into several sub-matrices, presenting a speed advantage over Mthree. Due to the need for multiple iterations to improve fidelity, Q-BEEP consumes a significant amount of time.

Although we demonstrate a higher speedup compared to previous methods, SpREM's fidelity still maintains a high level. Compared to CTMP, Mthree and Q-BEEP, the fidelity is improved by 1.2%, 0.05% and 1.7% on average, respectively in Figure 6 (b). Due to the reduction in the size of the mitigation matrix, Mthree introduces system error, and CTMP only considers local crosstalk, resulting in a decrease in fidelity. Q-BEEP is a statistically-based mitigation method based on the parameters of quantum circuits, and it cannot effectively mitigate the readout noise.

## 5.4 Hardware Performance

TABLE 1 compares the execution time for SpREM, Xilinx Vitis SPARSE library [21], cuSPARSE library [13]. On average, SpREM is 11.2×, 29.5× and 36.4× faster than Vitis SPARSE of CSC format and cuSPARSE of COO and CSR format. While the COO incurs extra storage overhead, it enables direct access to the corresponding values, resulting in 1.2× faster computation compared to the CSR format. When mitigating a 10-qubit quantum algorithm, the overhead of data loading prevents the full utilization of parallel computing advantages. As a result, SpREM achieves a remarkable speedup of 1,003 × compared to the NVIDIA A100 GPU with cuSPARSE-COO library when *HD* is 3. As the number of qubits increases, the gap between data loading and parallel computation diminishes. When the number of qubits is 16, SpREM demonstrates a 21.2× speedup. The Xilinx SpMV accelerator is tailored for sparse matrices with varying structures (differing numbers of non-zero values in each row), necessitating the use of split and merge logic to distribute multiplication results to corresponding rows. However, this introduces additional time cycles, and the supplementary split and merge overhead contributes to an overall increase in computation time, rendering it slower compared to SpREM.

TABLE 2 shows a breakdown of end-to-end execution time for SpREM and Xilinx Vitis SPARSE library [21] when $HD = 4$. SpREM

**Table 1: Execution time for SpREM, Xilinx Vitis SPARSE library [21], cuSPARSE library [13] under varying qubit numbers**

| | Hamming distance threshold (HD) = 3 | | | | Hamming distance threshold (HD) = 4 | | | |
|---|---|---|---|---|---|---|---|---|
| | 10 qubits | 12 qubits | 14 qubits | 16 qubits | 10 qubits | 12 qubits | 14 qubits | 16 qubits |
| Nonzeros | 180.22K | 1.22M | 7.70M | 45.68M | 395.26K | 3.25M | 24.10M | 164.95M |
| SpREM | $\mathbf{6.00 \times 10^{-3}}$**ms** | **0.034ms** | **0.21ms** | **1.39ms** | **0.011ms** | **0.088ms** | **0.60ms** | **3.95ms** |
| Vitis SPARSE [21] | 0.058ms (9.7 ×) | 0.34ms (10.0×) | 2.15ms (10.2 ×) | 12.23ms (8.8 ×) | 0.10ms (9.1×) | 0.77ms (8.8×) | 7.51ms (12.5×) | 47.21ms (12.0×) |
| cuSPARSE-COO [13] | 6.02ms (1,003×) | 6.29ms (185×) | 10.17ms (48.4×) | 29.46ms (21.2×) | 6.19ms (563×) | 7.63ms (86.7×) | 18.41ms (30.7×) | 89.99ms (22.8×) |
| cuSPARSE-CSR [13] | 6.87ms (1,145×) | 7.51ms (221×) | 11.55ms (55.0×) | 36.35ms (26.2×) | 7.05ms (641×) | 8.89ms (101×) | 22.53ms (37.6×) | 115.14ms (29.2×) |

**Table 2: Breakdown of execution time for SpREM and Xilinx Vitis SPARSE library [21]**

| | | Data Loading | Computation | Data Storage | Overall |
|---|---|---|---|---|---|
| 10 qubits | SpREM | 0.011ms | $9.02 \times 10^{-3}$ms | $1.13 \times 10^{-3}$ms | 0.011ms |
| | Vitis SPARSE [21] | 0.075ms | 0.078 ms | $5.96 \times 10^{-3}$ms | 0.10ms |
| 12 qubits | SpREM | 0.084ms | 0.072 ms | $8.80 \times 10^{-3}$ms | 0.088ms |
| | Vitis SPARSE [21] | 0.59ms | 0.61ms | 0.045ms | 0.77ms |
| 14 qubits | SpREM | 0.59ms | 0.49 ms | 0.054ms | 0.60ms |
| | Vitis SPARSE [21] | 5.56ms | 5.93ms | 0.45ms | 7.51ms |
| 16 qubits | SpREM | 3.78ms | 3.20ms | 0.38ms | 3.95ms |
| | Vitis SPARSE [21] | 35.41ms | 37.28ms | 2.50ms | 47.21ms |

experiences a performance bottleneck related to data loading, accounting for 96.5% of the total time. The non-zero values in each row of the channel partition are padded to address accumulation latency in the Xilinx SpMV accelerator. Furthermore, compared to SpREM, additional arrays—column pointer array and row index array need to be transmitted. All these factors collectively contribute to an elevated overhead in data loading.

## 6 RELATED WORK

**Matrix-based error mitigation.** Some work [16] involves constructing 2 × 2 noise matrices for each qubit and mitigating readout error through tensor product. CTMP [3] mitigates crosstalk between two qubits based on continuous-time Markov processes. Mthree [12] leverages the zero-probability in the measurement distribution to reduce the size of the mitigation matrix.

**State-based error mitigation.** In light of the observation that errors are more prone to occur in state "1" as opposed to state "0", some works [10, 18] minimize measurements of state "1" through the inversion of measurement outcomes to mitigate readout noise.

**Other error mitigation.** Jigsaw [5] separately measures a portion of the quantum circuit and reconstructs the complete results using Bayesian reconstruction, effectively eliminating readout crosstalk. Hammer [17] uses Hamming distance for weighted computation to eliminate noise. Q-BEEP [15] constructs an error model based on circuit parameters and corrects errors through iterative Bayesian network update.

## 7 CONCLUSION

In this paper, we propose SpREM, an effective Hamming-Distance Sparse Row (HDSR) compression method that leverages the structured sparse pattern of the mitigation matrix. This demonstrates 98.9% sparsity and a 27.3× reduction in fidelity loss on the real-world quantum computer, compared to threshold-based pruning method. In addition, our hardware accelerator leverages an XOR array and a reconfigurable adder tree, achieving 11.2× ∼ 36.4× speedup compared to Xilinx Vitis SPARSE library and NVIDIA A100 GPU implementations.

## ACKNOWLEDGMENTS

This work was supported by the National Key Research and Development Program of China (No. 2023YFF0905200). This work was also funded Zhejiang Pioneer (Jianbing) Project (No. 2023C01036) and the National Natural Science Foundation of China under Grant (No. 61825205).

## REFERENCES

[1] Google Quantum AI. 2021. Exponential suppression of bit or phase errors with cyclic error correction. *Nature* 595, 7867 (2021), 383.

[2] Charles H Bennett, Ethan Bernstein, et al. 1997. Strengths and weaknesses of quantum computing. *SICOMP* 26, 5 (1997), 1510–1523.

[3] Sergey Bravyi, Sarah Sheldon, et al. 2021. Mitigating measurement errors in multiqubit experiments. *Phys. Rev. A* 103, 4 (2021), 042605.

[4] Luc Buatois, Guillaume Caumon, et al. 2007. Concurrent number cruncher: An efficient sparse linear solver on the GPU. In *HPCC*. Springer, 358–371.

[5] Poulami Das, Swamit Tannu, et al. 2021. Jigsaw: Boosting fidelity of nisq programs via measurement subsetting. In *MICRO*. 937–949.

[6] David Deutsch and Richard Jozsa. 1992. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A* 439, 1907 (1992), 553–558.

[7] Daniel M Greenberger, Michael A Horne, et al. 1989. Going beyond Bell's theorem. In *Bell's theorem, quantum theory and conceptions of the universe*. Springer, 69–72.

[8] Song Han, Huizi Mao, et al. 2015. Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv:1510.00149* (2015).

[9] Jihye Kim, Byungdu Oh, et al. 2022. Quantum readout error mitigation via deep learning. *NJP* 24, 7 (2022), 073009.

[10] Hyeokjea Kwon and Joonwoo Bae. 2020. A hybrid quantum-classical approach to mitigating measurement errors in quantum algorithms. *IEEE Trans. Comput.* 70, 9 (2020), 1401–1411.

[11] Satvik Maurya, Chaithanya Naik Mude, et al. 2023. Scaling Qubit Readout with Hardware Efficient Machine Learning Architectures. In *ISCA*. 1–13.

[12] Paul D Nation, Hwajung Kang, et al. 2021. Scalable mitigation of measurement errors on quantum computers. *PRX Quantum* 2, 4 (2021), 040326.

[13] Maxim Naumov, L Chien, et al. 2010. Cusparse library. In *GTC*.

[14] Rigetti. 2022. *Rigetti Computing: Quantum Computing*. https://www.rigetti.com/about-rigetti-computing

[15] Samuel Stein, Nathan Wiebe, et al. 2023. Q-BEEP: Quantum Bayesian Error Mitigation Employing Poisson Modeling over the Hamming Spectrum. In *ISCA*. 1–13.

[16] Mingyu Sun and Michael R Geller. 2018. Efficient characterization of correlated SPAM errors. *arXiv:1810.10523* (2018).

[17] Swamit Tannu, Poulami Das, et al. 2022. Hammer: boosting fidelity of noisy quantum circuits by exploiting hamming behavior of erroneous outcomes. In *ASPLOS*. 529–540.

[18] Swamit S Tannu and Moinuddin K Qureshi. 2019. Mitigating measurement errors in quantum computers by exploiting state-dependent bias. In *MICRO*. 279–290.

[19] Benjamin Villalonga, Dmitry Lyakh, et al. 2020. Establishing the quantum supremacy frontier with a 281 pflop/s simulation. *Quantum Sci. Technol.* 5, 3 (2020), 034003.

[20] Hanrui Wang, Jiaqi Gu, et al. 2022. QuantumNAT: quantum noise-aware training with noise injection, quantization and normalization. In *DAC*. 1–6.

[21] Xilinx. 2023. *Vitis Sparse Library*. https://github.com/Xilinx/Vitis_Libraries/tree/main/sparse/