Memory Controller Design with Bitmask Compression

Ege Ozgul
Electrical Engineering Department
Tufts University
Medford, United States
Ege.Ozgul@Tufts.edu

Marco Donato
Electrical Engineering Department
Tufts University
Medford, United States
Marco.Donato@Tufts.edu

***Abstract*—Inside most electronic devices, processors use various memory units to function. Before physically manufacturing these processors, a designer needs to test them while developing the hardware by using computer simulations. To simulate the designed processor's behaviour while interacting with a memory unit, a model of the memory should be accompanied into the simulation. There are lots of proprietary memory model designs, in fact there are not as many open-source memory models in the field. In this paper, an open source memory model is introduced that enables hardware designers to speed up their development process by allowing them to integrate this memory model into their hardware designs. This memory model is designed to be parametric, which allows designers to configure it based on their own project specifications. A compression hardware unit that is integrated into the memory model is introduced which uses bitmasking algorithms for compressing large sparse matrices used for storing the weights of neural networks in the memory for machine learning application [2], [3].**

***Index Terms*—Sparse matrices, bit-masking, compression, memory model, memory controller, efficiency, linear algebra, machine learning, memory model, hardware design, hardware simulation, processor**

I. Introduction

Almost all electronics devices have a processor that performs logical computations in order to provide sophisticated features to the user. System-On-Chip(SoC) processors are chips that integrate many components on the chip itself rather than requiring additional components to be mounted on the circuit board [2] [13], [12], [11]. Manufacturing a SoC processor though is a very expensive process, therefore a hardware design should be guaranteed to be flawless before producing the chip. In order to make sure that the processor design has no error, hardware designers use computer simulations. They test their digital design by simulating the entire processor design on a computer. This allows the designers to catch design bugs, validate their design, and it also helps them trace the source of the errors.

By integrating the existing hardware design units into their designs, hardware designers can improve the development speed and accuracy by eliminating the need for testing the designed unit. Open source hardware designs allows any hardware designer to integrate the design into their product.

In this paper, an open source memory model is developed which provides any hardware designer a memory model design that is proven to work. This memory model is fully parametric, which allows the hardware designer to configure the parameters of the memory model, so that the specification of the model matches with the requirements of their hardware design. This memory model has an optional compression feature that can be enabled or disabled by toggling a single bit. The compression feature uses bit-mask compression on large sparse matrices. Sparse matrices are commonly used in machine learning, scientific computing and numerical analysis applications [1]. Directly storing the sparse matrices on the memory leads to inefficient use of the memory. By applying compression, it is possible to only store the non zero data, which enables much more efficient usage of the physical memory. In this paper, bit-mask compression method is integrated into the memory model design in order to provide hardware developers the option to utilize the compression feature if their application involve using sparse matrices.

- **Parametric Sparse Matrix Generation**: A set of algorithms are introduced for generating parametric random sparse matrices that allows specifying the dimension and sparsity of the generated matrix. The generated matrices are used for testing the designed memory controller design, and it is also used for analyzing the performance of bit-mask error mitigation algorithm that is used in the design [8].
- **Sparse Matrix Compression With Bit-Masking**: Bit-mask compression and decompression, and error mitigation algorithms are explained. [8].
- **Probabilistic Analysis of Decompression with Error Mitigation**: The models used for analyzing the benefits of error mitigation for large sparse are presented [4], [5].
- **Memory Controller Model Design**: A memory controller design is introduced. The controller uses compression, decompression, and error mitigation algorithms for sparse matrices.

Thanks to the Tufts Emerging Circuits and Systems Laboratory (TECS).

II. Parametric Sparse Matrix Generation

The memory model that is designed in this paper needs to be initialized by providing a text file that contains matrix data. In order to generate this initialization file, a custom python script is written that uses the following algorithm to generate a parametric and random sparse matrix. The parameters of the algorithm are the size of the matrix, and the sparsity rate (a decimal value between 0 and 1).

- First the number of non zero values N is calculated using the following formula:

$$N = (1 - sparsity) * Width * Height$$

- Let K be a list of size N, and each or its element is set to a random number.

$$K = \begin{bmatrix} 254 & 227 & 33 & 120 & 242 & 188 \end{bmatrix}$$

- Let M be a list of size N, and each of its element is set to a unique random value that is between 0 and (width*height). In the lists K and M, the M_i indicates the position of the data K_i, where i is the index from 0 to (Width*Height-1).

$$M = \begin{bmatrix} 46 & 13 & 8 & 42 & 54 & 63 \end{bmatrix}$$

- A matrix with the specified size is created. All values in the matrix are initialized to zero.
  Zero Matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- Finally, the list K is used to compute the position of each data in list M to form the sparse matrix. To do so, for all values of i in list K and M, data K_i is inserted into the index M_i on the zero matrix. Since the matrices are defined as a single dimensional lists, it is possible to use single index values to indicate the position of each element. In the case of using a two dimensional matrix, the x and y indices could be computed using the following formulas:

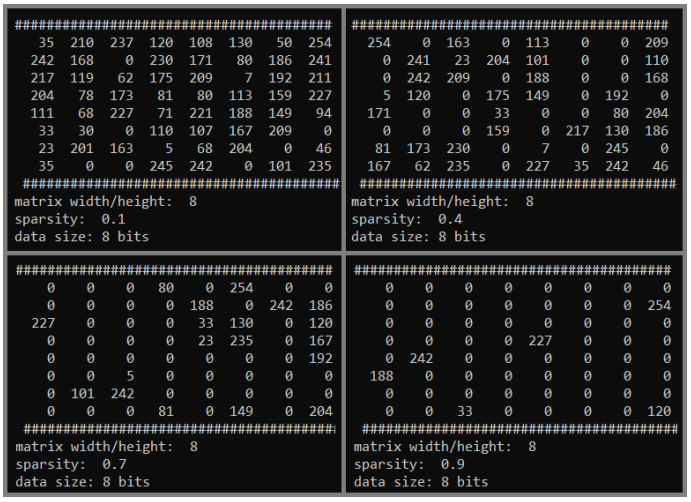
$$x_i = M_i \% width$$

$$y_i = M_i / width$$

Sparse Matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 33 & 0 & 0 & 0 & 0 & 227 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 120 & 0 & 0 & 0 & 254 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 242 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 188 \end{bmatrix}$$

The following figure is a four sample terminal output of the python sparse matrix generation script. Each of the four sparse matrix is randomly generated with different sparsity rates. It is possible to generate infinitely large sparse matrices with arbitrary sparsity rates using this program.



III. Sparse Matrix Compression With Bit-masking

A. Bit-masking compression overview

Using bit-masking algorithm, highly sparse matrices can be significantly compressed which enables using the memory space to only store the non zero data which helps using the memory more efficiently [8]. The following part explains the compression of the sparse matrix below.

Sparse Matrix:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 254 & 0 \\ 0 & 192 & 0 & 0 & 0 & 0 & 242 & 227 \\ 0 & 0 & 0 & 0 & 188 & 0 & 0 & 0 \\ 0 & 0 & 0 & 130 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 33 & 0 \\ 0 & 0 & 0 & 120 & 167 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- First, non zero elements are extracted from the sparse matrix, and a new non-zero-list that only contains the non zero values is created.
  nonZerolist:

$$\begin{bmatrix} 254 & 192 & 242 & 227 & 188 & 130 & 33 & 120 & 167 \end{bmatrix}$$

- Then a bit-mask matrix that has the same dimension as the original sparse matrix is created. Each element of this

bit-mask matrix is a 1 if that element is a non zero value at the same position on the sparse matrix, otherwise the element holds 0.
Bit-mask:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

B. Bit-mask Error Injection overview

In physical memory devices, some error occur and the data might get corrupted [10]. There are some algorithms that help to correct the error, or mitigate it by preventing an error in one region of the memory from affecting the bits in other regions. In order to mathematically model the benefits of error correction and mitigation techniques used in this paper, an error injection script is developed. The error injection algorithm is parametric, which means that is can be used for any size of matrix, and the error rate to be injected can also be specified by setting the errorRate parameter between 0 and 1.

C. Injecting Error Into Bit-mask

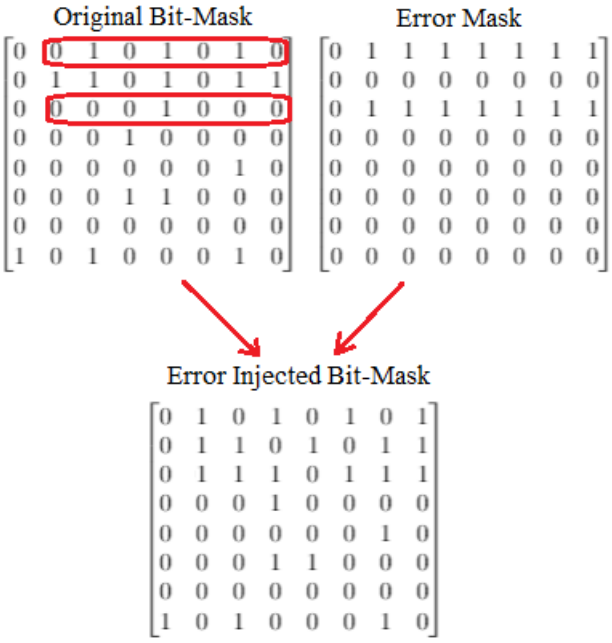
- To inject error into the boolean Bit-Mask matrix, an additional error matrix is generated. To generate an Error Mask, an index List is generated. This size of this index list is determined using the following formula, where width and height correspond to the size of the original matrix. The error rate is between 0 and 1.

$$N = ErrorRate * width * Height$$

- A list of size N is generated, and the elements are set to a unique random integer ranging form 0 to $(width \times height - 1)$. A boolean matrix is then created where only the elements at positions indicated by the numbers on the IndexList are set to 1. Other values are set to zero.
Index List:

$$\begin{bmatrix} 2 & 4 & 6 & 9 & 10 & 12 & 14 & 15 & 20 & 27 \\ 38 & 43 & 44 & 56 & 58 & 62 & & & & \end{bmatrix}$$

- The values of ones represent the positions of the error to be injected in the original Bit-Mask. So the Error-Mask is used for flipping the corresponding bits on the original Bit-Mask in order to produce the Error Injected Bit-Mask.



D. Injecting Error Into Non-zero List

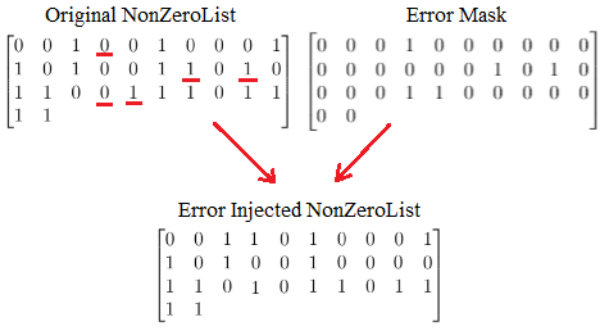
- The non-zero list contains integer values, so in order to inject error, the bits of the integer values are utilized: Let the following NonZeroList be an example list of integers:

$$NonZeroList = \begin{bmatrix} 2 & 4 & 6 & 9 & 10 & 12 & 14 & 15 \end{bmatrix}$$

The same technique could be applied to any other data type such as floating point numbers by using their binary encoding to
- NonZeroList is converted into binary, and the following binary list is generated.
BinaryNonZeroList:

$$\begin{bmatrix} 0010 & 0100 & 0110 & 1001 & 1010 & 1100 & 1110 & 1111 \end{bmatrix}$$

- Then, an Error-Mask (see the part C-1) is generated in order to inject error by flipping the bits on the list. By setting the errorRate parameter of the Error-Mask, the number of bits to be flipped is determined which effects the error density on the matrix.



- Finally, the following list is produced after injecting error into NonZeroList:

$$\begin{bmatrix} 3 & 4 & 6 & 9 & 0 & 13 & 6 & 15 \end{bmatrix}$$

E. Bit-Mask Decompression

Decompression algorithm follows the following procedure:

- Let $Matrix\_D$ be the decompressed matrix, and all of its elements are initialized to all zeroes. This matrix $Matrix\_D$ has the same dimension as the Bit-mask matrix.
- For all element of the matrix, $Matrix\_D_i$ is kept 0 if $Bitmask_i$ equals to 0, otherwise the first element of the nonZeroList is removed and overwritten at $Matrix\_D_i$.

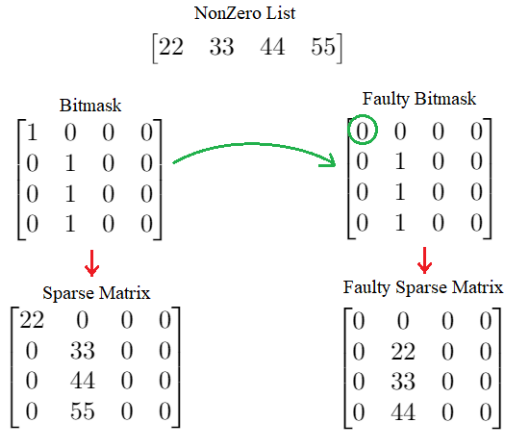
Below is a sample BitMask decompression implementation using Python language:

```
def decompress(nonzeroList, bitmask):
    decompressedList = zeros(len(bitmask));

    nonZeroList_i = 0;
    for i in range(len(bitmask)):
        if(bitmask[i] == 1):
            decompressedList[i] = nonzeroList[nonZeroList_i];
            nonZeroList_i += 1;

    return(decompressedList);
```

F. Error Mitigation Overview for Bitmask Compression

When sparse memory compression is being used, if an error occurs on the memory, then some portion of the non-zero and the bit-mask bits can get corrupted. If the bit-mask bits change due to memory error, there is a high chance that the non-zero data list might no longer match with the ones on the bit-mask.

In the sample case below, the left hand side of the figure below displays the decompression output when there is no error. The right hand side displays the output of the decompression when the first bit of the Bit-mask is flipped. When the results are compared, the positions of the non-zero values on the faulty sparse matrix does not match with positions of those in the original sparse matrix. Therefore, a single bit corruption on the first bit of the Bit-mask have a potential to ruin the entire decompression output.



G. Error Mitigation Algorithm for Bit-mask Compression

To solve the problem defined in the previous section, an error mitigation technique is used [8]. The goal of the error mitigation is to prevent an error in any region of the memory from effecting the output of the other regions when the region is decompressed.

1) Generating non-zero data counter list: The error mitigation algorithm uses an additional list of integers [8]. Each element of this list stores the number of ones on the bit-mask at each n bit group where n is 128.

Let $n = 4$ for visualization purposes:

Sample bit-mask:

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Each green circled group represents a chunk of 4 bits on the sample bit-mask. Each element of the nonZero counter list below counts the number of ones on each chunk.

$NonZeroCounterList$:

$$\begin{bmatrix} 0 & 1 & 3 & 2 & 0 & 2 & 2 & 0 & 2 & 1 \end{bmatrix}$$

Using the nonZero counter list allows the decompression algorithm to keep track of the number of non-zero values in each chunk which makes it possible to isolate the chunks and prevent any error in any chunk from effecting the other chunks.

2) Decompression using non-zero data counter list: The algorithm decompresses each chunk of the sprase matrix separately [8]. Since the number of non-zero values on each chunk is stored in the nonZero counter list, the index range of the non-zero values that belong to each chunk could be calculated using the following formula:

To calculate the index of the non-zero values for each chunk, the cumulative sum of the nonZero list is calculated, and each element is decremented by 1 which yields the following cumulative nonZero counter list.

$CumulativeNonZeroCounterList$:

$$C = \begin{bmatrix} -1 & 0 & 3 & 5 & 5 & 7 & 9 & 9 & 11 & 12 \end{bmatrix}$$

The index range of the nonZero values on nonZero List for chunk i is then calculated as follows, where $index_0$ refers to the starting index and $index_1$ refers to the last index on the nonZero list.

$$index_0 = C_i$$

$$index_1 = C_i - 1$$

To decompress each chunk, the non-zero values in the range $index_0$ and $index_1$ are used. If the number of ones on the

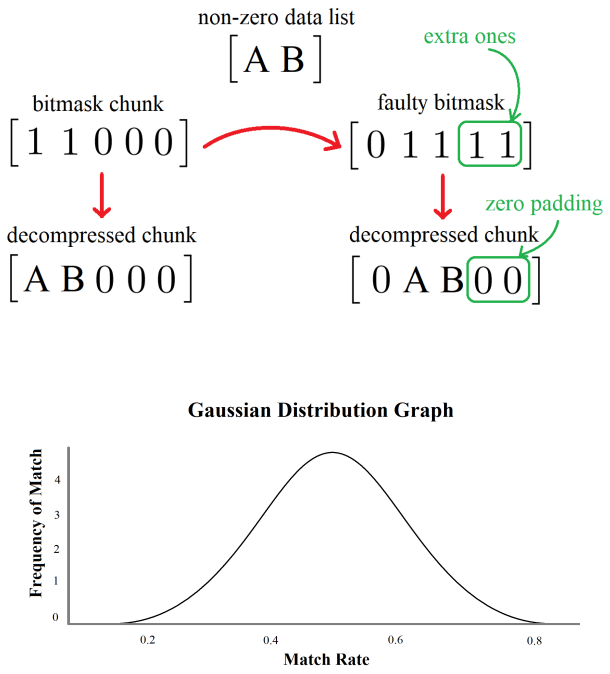


Fig. 1. A sample Gaussian distribution plot

faulty bit-mask chunk is greater than the number of non-zero values on the original chunk, then the rest of the chunk is padded with zero.

On the sample diagram below, each list represents a chunk of a bigger list or matrix. The original bit-mask below is injected error by flipping the third bit(circled in green). Since there is not a third non-zero data on the non-zero data list, rest of the values are set to 0.

IV. Probabilistic Analysis of Decompression with Error Mitigation

A. Match Rate

To compare two matrices that are very similar, only the non zero data and their positions are compared. Let A be the number of same non zero values at the same positions in each matrix. Let B be the total number of non zero values in the original matrix. Then the term match rate is used to indicate the similarity between these matrices, where match rate = A/B

B. Statistical Modeling

Gaussian distribution model is used for modeling the match rates of random matrices with identical error rates. In the sample Gaussian distribution graph (see Figure 1), the horizontal axis represents the match rate of the sparse matrices. The vertical axis represents the frequency of matches.

By changing the error rate, multiple Gaussian distribution plots are produced to generate a 3D surface where the additional third axis represents the varying error rate. On figures 2 and 3, the **Output Match Rate** and **Frequency of Accuracy** axes are used for producing 2 dimensional Gaussian distribution. For each value of the **Input Error Rate** axis, a slice of 2D Gaussian distribution plots are produced to make a 3D surface. The red curves on figures 2 and 3 indicate the mean points of the Gaussian plots, in other words the read curves show the spine of the 3D shape.

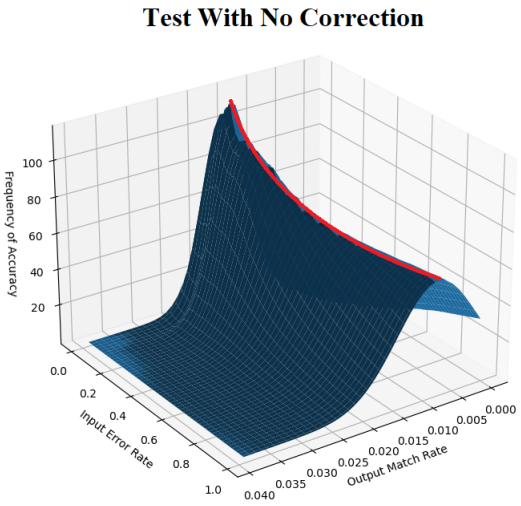


Fig. 2. Gaussian surface With no correction

Regarding the figure 3, where correction is applied on sparse matrices with some error injected, the red line approaches to 1 on **Output Match Rate** axis as the **Input Error Rate** decreases to 0, which indicates that sparse matrices match more as the error rates is reduces. On the other hand, on the figure 2 the red line stays almost constant on **Output Match Rate** axis as the input error rate is changed. This red line which indicates the mean of the Gaussian plots stays at 0.012 which shows that even when the error rate is reduced, the matrices only match by 1.2%. Therefore, applying correction on sparse matrices with error helps to reduce the error rate. Note that on these both of the figures 2 and 3, the surface does not cover the case where input error rates are 0 since it is not possible to make a Gaussian distribution plot when all random sparse matrix samples match exactly.

V. Memory Controller Model Design

A. Design Overview

In this part, the design of the memory model is explained in detail. The memory model is composed of smaller units. The main sub-units are the Controller Unit, Compression Unit, and the Memory Unit which are all synchronized by and external clock. The Bus Select units allow toggling the compression feature. The Test-Bench acts as a host that can communicate with the entire unit through the controller unit. The controller unit allows the host to read from and write to the memory module.

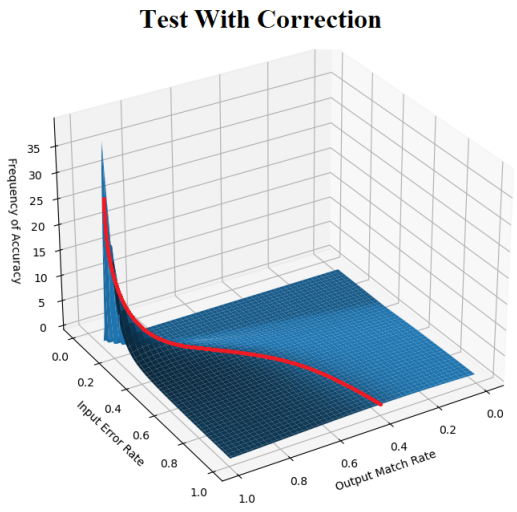


Fig. 3. Gaussian surface with correction applied

The memory module models the physical memory. When the simulation is executed, memory Module is initialized by automatically loading a initialization file that is generated by a Python script that runs parametric randomized sparse matrix generation algorithms.

B. Parent and Child

In figure 4, a sub-unit on the left is the parent of a sub-unit that is on the right. The parent controls the behaviour of the child by using the bus lines defined below.

C. Communication Bus

All units in the hardware design are interconnected by the same type of bus. This bus has different groups of wires, and each wire group is explained below:

- **address**: This address bus is used for specifying the address of the data that is being requested by the host from the controller unit. It is also used between other sub-units that request data from each other.
- **data_tx**: This is the data transmission line that allows sending data by a left sub-unit to the right sub-unit on the figure 4.
- **data_rx**: This is the data transmission line that allows sending data by a right sub-unit to the left sub-unit on the figure 4.
- **flag_tx**: When the parent sends a data to a child, it pulls this flag_tx line high for one clock cycle to produce a rising edge.
- **flag_rx**: When the parent asks for data to be received from a child, it pulls this flag_tx line high for one clock cycle to produce a rising edge.
- **ready**: A child sub-unit pulls the ready wire high when it is ready to listen to any command from a parent. At reset, this wire is initially set to low until the sub-unit is initialized.
- **mem_compress**: This is a single bit wire that enables compression.

D. Memory Module

The memory model contains an array that represents the physical memory on a chip.

E. Controller Unit

The controller Unit allows the host to communicate with the entire system and it controls the other sub-units depending on the input received from the host.

F. Bus Select

The bus select unit connects the input bus to one of the output busses that can be selected using a control line that connects to the controller unit. The bus select unit uses multiplexers to connect bidirectional busses.

G. Compression Unit

The compression unit has two functions: Compression for writing, and decompression for reading from the memory. When the controller unit needs to write data on the memory module, it first sends the data to the compression, then the data is compressed by the compression unit, and finally the compressed data is sent to the memory unit from the compression unit. To read data from the memory, compression unit receives the data, decompresses it and sends it to the controller unit.

H. Transmitting a Data Package between two units

Transmitting a data package from parent unit A to the child unit B over bus C involve several Steps(see Figure 4).First, unit A sets the flag_tx on bus C high. Unit B detects state of this flag and responds by sending K bits of data for N clock cycles by using the data_rx line on the bus. The bandwidth of the bus is K bits, therefore sending K bits for N clock cycles allows the unit B to send a total of $K * N$ bits of data. Once the transmission is done, the unit B sets the line **ready** high, to indicate that it finished the transmission and it is ready to listen to a new request. Transmitting data from unit B to unit A involves repeating the same steps above but with the opposite data and flag lines: Flag_rx instead of flag_tx ,data_tx instead of data_rx

I. Read Operation

The **address** line is set to the target address value by the host(test-bench) (see Figure 4). The flag_rx line is set high to start the transmission. Finally the host waits until the memory modules completes sending bits for multiple cycles and sets the **ready** bit high to indicate that the transmission is over.

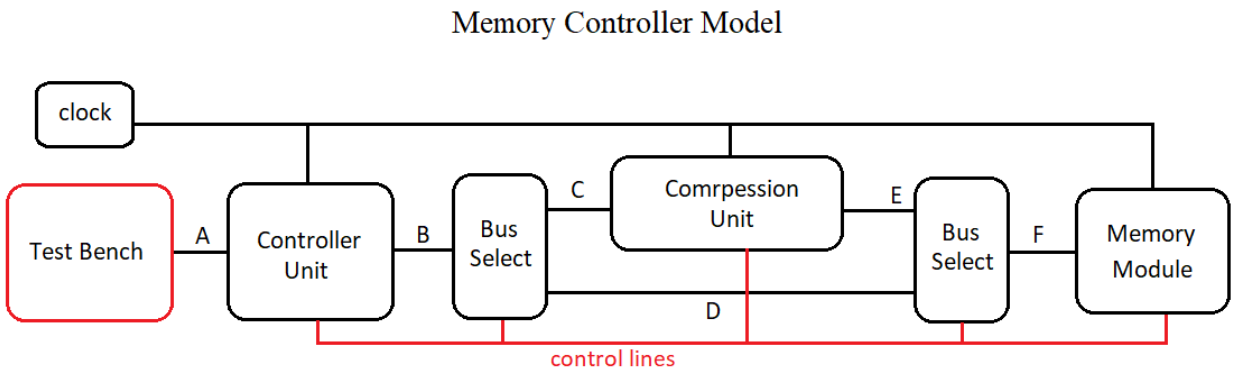


Fig. 4. The block diagram of the memory controller model

J. Write Operation

- To write data on the memory, the host sets the target address on lines **address** and the data bits on **data_tx** lines. Then the host sets the flag_tx line high to start the transmission.
- The controller unit reads the address line on bus A, then it receives the corresponding data block from the memory block by using the bus B.
- After the controller unit receives the data block from the memory module, it modifies this data block by overwriting the data sent from the host.
- Finally it send the modified data block back to the memory module by using the bus B, and sets the **ready** flag on bus A high once the transmission is finished.

K. Enabling and Disabling Compression

All data busses have the **Mem_compress** wire to toggle compression feature. This line is set by the host device only. When the **Mem_compress** line is set high, bus B connects to bus C, and the bus E connects to bus F allowing to use the compression unit. Wen **Mem_compress** is set low, the compression unit is bypassed by connecting the bus B to D and D to F.

VI. Conclusion

In this paper, an open source memory controller model design is introduced which is fully parametric and has an optional bit-masking compression feature. This feature makes this memory controller ideal for applications that involve sparse matrices such as machine leaning and numerical analysis. For highly sparse large matrices, bit-mask compression can significantly reduce the memory usage. The introduced memory controller can also perform some error mitigation which prevents an error in the compressed format from ruining the entire sparse matrix by isolating the faulty region from other regions. The benefits of error mitigation algorithm used in this paper is analyzed by performing some probabilistic tests. These tests involved generation of random sparse matrices and utilization of Gaussian distribution plots to generate two surfaces that reveals the significant benefits of using error mitigation.

As a future research direction, the data transmission between the units of the memory controller can be optimized by using banking which allows simultaneously reading and writing multiple words on the memory. Besides bit-masking compression feature, other compression units such as compressed sparse column(CSR) could be integrated into the design [10].

VII. Related Work

Sparse matrix compression methods have been developed and utilized since 1960s when compressed sparse row format was developed to store sparse matrices by using less memory space [1]. The SMASH hardware-software was designed in the past which was intended for recognizing and exploiting the sparsity of the stored data to achieve indexing acceleration for matrix operations. MaxNVM was another project in the same field which used Sparse Encoding and Error Mitigation Techniques to maximize DNN storage density and increase the performance of inference.

References

[1] W. Tinney and J. Walker, “Direct Solutions of Sparse Network Equations by Optimally Ordered Triangular Factorization,” Proc. IEEE, vol. 55, no. 11, pp. 1801-1809, Nov. 1967.

[2] K. Basu and P. Mishra, ”Test Data Compression Using Efficient Bitmask and Dictionary Selection Methods,” in IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 18, no. 9, pp. 1277-1286, Sept. 2010, doi: 10.1109/TVLSI.2009.2024116.

[3] S. Seong and P. Mishra, ”Bitmask-Based Code Compression for Embedded Systems,” in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, no. 4, pp. 673-685, April 2008, doi: 10.1109/TCAD.2008.917563.

[4] C. Murthy and P. Mishra, ”Lossless Compression Using Efficient Encoding of Bitmasks,” 2009 IEEE Computer Society Annual Symposium on VLSI, 2009, pp. 163-168, doi: 10.1109/ISVLSI.2009.18.

[5] Seok-Won Seong and Prabhat Mishra. 2006. A bitmask-based code compression technique for embedded systems. In ¡i¿Proceedings of the 2006 IEEE/ACM international conference on Computer-aided design¡/i¿ (¡i¿ICCAD ’06¡/i¿). Association for Computing Machinery, New York, NY, USA, 251–254. DOI:https://doi.org/10.1145/1233501.1233551

[6] Jeremiah Willcock and Andrew Lumsdaine. 2006. Accelerating sparse matrix computations via data compression. In ¡i¿Proceedings of the 20th annual international conference on Supercomputing¡/i¿ (¡i¿ICS '06¡/i¿). Association for Computing Machinery, New York, NY, USA, 307–316. DOI:https://doi.org/10.1145/1183401.1183444

[7] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In ¡i¿Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures¡/i¿ (¡i¿SPAA '09¡/i¿). Association for Computing Machinery, New York, NY, USA, 233–244. DOI:https://doi.org/10.1145/1583991.1584053

[8] Lillian Pentecost, Marco Donato, Brandon Reagen, Udit Gupta, Siming Ma, Gu-Yeon Wei, and David Brooks. 2019. MaxNVM: Maximizing DNN Storage Density and Inference Efficiency with Sparse Encoding and Error Mitigation. In ¡i¿Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture¡/i¿ (¡i¿MICRO '52¡/i¿). Association for Computing Machinery, New York, NY, USA, 769–781. DOI:https://doi.org/10.1145/3352460.3358258

[9] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. 2019. SMASH: Co-designing Software Compression and Hardware-Accelerated Indexing for Efficient Sparse Matrix Operations. In ¡i¿Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture¡/i¿ (¡i¿MICRO '52¡/i¿). Association for Computing Machinery, New York, NY, USA, 600–614. DOI:https://doi.org/10.1145/3352460.3358286

[10] Guangyu Sun, Jishen Zhao, Matt Poremba, Cong Xu, Yuan Xie, Memory that never forgets: emerging nonvolatile memory and the implication for architecture design, National Science Review, Volume 5, Issue 4, July 2018, Pages 577–592, https://doi.org/10.1093/nsr/nwx082

[11] M. Poremba, T. Zhang and Y. Xie, "NVMain 2.0: A User-Friendly Memory Simulator to Model (Non-)Volatile Memory Systems," in IEEE Computer Architecture Letters, vol. 14, no. 2, pp. 140-143, 1 July-Dec. 2015, doi: 10.1109/LCA.2015.2402435.

[12] M. Donato, L. Pentecost, D. Brooks and G. Wei, "MEMTI: Optimizing On-Chip Nonvolatile Storage for Visual Multitask Inference at the Edge," in IEEE Micro, vol. 39, no. 6, pp. 73-81, 1 Nov.-Dec. 2019, doi: 10.1109/MM.2019.2944782.

[13] A. Keshavarzi, K. Ni, W. Van Den Hoek, S. Datta and A. Raychowdhury, "FerroElectronics for Edge Intelligence," in IEEE Micro, vol. 40, no. 6, pp. 33-48, 1 Nov.-Dec. 2020, doi: 10.1109/MM.2020.3026667.

[14] (Naik), Yogesh Awdhut Gadade. "Sparse Matrix in Machine Learning." Limitless Data Science, 26 Nov. 2020, limitlessdatascience.wordpress.com/2020/11/26/sparse-matrix-in-machine-learning/.