LDPC Codes In Computer Memory

ELEC 433 Final Project

Tom Wang Natalie Balashov

***Abstract*—After discussing the importance of error-correcting codes in modern memory systems, we include a short overview of low-density parity check (LDPC) codes. Motivated by the literature which suggests that LDPC codes manifest superior performance to alternative coding schemes, we describe and assess a simple hardware implementation that aimed to compare an LDPC and Hamming code.**

I. Introduction

Error-correcting codes (ECC) have become an integral part of computer memory. With the advent of new memory technologies and an increase in memory density, bit errors occur more frequently, leading to data corruption. In 2007, the CERN computer centre measured and analyzed the quantity and types of errors that incurred data corruption [5]. It was noted that the high volume of memory accesses executed by the computers was a strong factor in the high number of errors, and that in such complex systems, simple redundancies or single-bit error checks are not always sufficient. As memory systems gain in complexity, the need for data reliability and integrity remains crucial. Moreover, for highly specialized applications such as data storage and transportation in space missions [6], more robust error correction schemes are required due to overly noisy channels created by radiation.

This problem has been studied, with various coding schemes and techniques used to detect and rectify the errors. For older systems, simple parity checks or single-bit correction, executed upon the retrieval of data from storage, used to be sufficient for safe-guarding memory integrity. However, with larger memories and a high volume of data throughput, additional probing and correction schemes are necessary [5]. For example, scrubbing is the periodic scanning of memory regions, in order to detect bit flips [6]. While such error-preventive measures increase data integrity, they also incur significant performance and hardware overhead, since more resources must be allocated for error-checking. Due to this trade-off, effective and low-complexity codes are considered for such applications in maintaining data reliability in memory systems.

II. LDPC Codes

Low-density parity check codes, also known as LDPC codes, are a class of linear block codes that are characterized by sparse parity check matrices. The sparsity of the parity check matrix allows for efficient encoding and decoding algorithms. LDPC codes are also known to achieve near Shannon capacity performance when decoded using iterative message-passing algorithms. LDPC codes were first proposed by Gallager, as part of his Ph.D. thesis in 1960 [3].

An LDPC code with parameters (n, j, i) has a block length of n, where each column of the parity check matrix contains at most j ones, and where each row of the parity matrix contains at most i ones.

A. Types of LDPC Codes

There are two main types of LDPC codes.

A regular (n, j, i) LDPC code [2] has a column weight of j and a row weight of i. For example, a parity check matrix for a regular LDPC code where $j = 3$ and $i = 4$ may resemble the following:

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

The dimension of the code for the above parity matrix is $n = 12$. The rate is calculated as $R = \frac{k}{n} = 1 - \frac{j}{i} = \frac{1}{4}$.

With irregular LDPC codes, each column may have a weight that is at most j and each row may have a weight that is at most i. In fact, all of the column and row weights are not necessarily equal. Due to the distribution of parity checks throughout the parity check matrix, irregular LDPC codes can achieve better performance than regular LDPC codes. However, irregular LDPC codes are more difficult to analyze and implement. An example of an application where irregular LDPC codes offer efficiency is the correction of asymmetric bit errors in spin-transfer torque random access memory (STT RAM) [8], which is an error type where traditional codes like BCH and Reed-Solomon fall short in terms of effectiveness. Irregular LDPC codes are able to combat this issue in STT RAM by providing more flexibility with regards to the location and number of parity check bits, thus enabling superior performance for bit errors with skewed probabilities.

B. Tanner Graphs

A Tanner graph is a bipartite graph [2] that represents the parity check matrix of an LDPC code. The Tanner graph

consists of two sets of nodes: variable nodes and check nodes. The edges of the graph connect variable nodes to check nodes and vice versa. Moreover, since the graph is bipartite, no variable nodes can be interconnected, nor can check nodes share an edge.

The Tanner graph is used to visualize the structure of the LDPC code and to develop efficient decoding algorithms.

For example, the matrix

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

may be represented as the following Tanner graph.

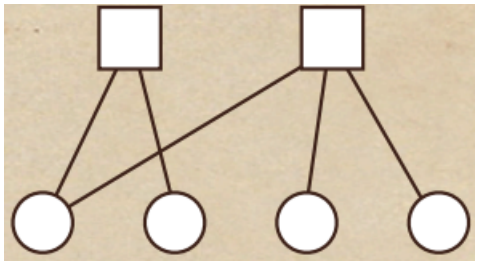

Fig. 1. Tanner graph representation of a parity check matrix [2].

In Fig. 1, we see that there are two check nodes, and four variable nodes.

A **cycle** in a Tanner graph is a closed path that starts and ends at the same node.

The **girth** of a Tanner graph is the length of the shortest cycle in the graph. A larger girth is desirable because it leads to better error-correction performance.

A Tanner graph is useful for visualizing the relationship between parity check equations, in addition to acting as a visual mnemonic for certain LDPC decoding schemes.

C. LDPC Parity Check and Generator Matrices

Given the parameters (n, j, i), a valid LPDC parity check matrix may be constructed using the following procedure [2]:

1) Construct a sub-matrix H_0 of dimensions $\frac{n-i}{j}$ by n, where a "diagonal" of i ones is constructed. If we denote $1_1, 1_2, \ldots, 1_i$ as the "indexed" ones in each row, then the sub-matrix H_0 will look something like

$$\begin{bmatrix} 1_1 & \ldots & 1_i & 0 & 0 & 0 & 0 & 0 & \ldots & 0 \\ 0 & \ldots & 0 & 1_1 & \ldots & 1_i & 0 & 0 & \ldots & 0 \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & 0 & \ldots & 0 & 0 & 1_1 & \ldots & 1_i \end{bmatrix}$$

2) Find $j-1$ different matrices that have pseudo-random column permutations of H_0. Denote these new sub-matrices as $H_1, H_2, \ldots, H_{j-1}$.
3) Construct the final parity check matrix H by combining sub-matrices $H_0, H_1, \ldots, H_{j-1}$ side-by-side into a single $n-k$ by n matrix.

Given a parity check matrix H for an LDPC code, the generator matrix G can be constructed. If necessary, re-organize the codeword $\mathbf{c}$ of length n so that it is of the form $\mathbf{c} = [\mathbf{b}, \mathbf{m}]$, where $\mathbf{b}$ denotes the parity bits vector and $\mathbf{m}$ denotes the message bits vector. We note that, if there are no errors in the codeword, then it must be that $\mathbf{c}H^T = [\mathbf{b}, \mathbf{m}]H^T = 0$.

Sub-divide the parity matrix H into two sub-matrices such that $H = [H_1, H_2]^T$ where H_1 is a square matrix of size $n-k$. Note that $\mathbf{b}$ is of length $n-k$ as well. Therefore, we can break up the matrix multiplication into $[\mathbf{b}, \mathbf{m}][H_1, H_2]^T = \mathbf{b}H_1^T + \mathbf{m}H_2^T = 0$. Since we are dealing with a code defined on a binary field, $\mathbf{b}H_1^T = \mathbf{m}H_2^T$ implies that $\mathbf{b} = \mathbf{m}H_2^T(H_1^T)^{-1}$.

With $A = H_2^T(H_1^T)^{-1}$, the generator matrix G is $[A, I_{n-k}]$.

III. HDL Implementation of a Regular LDPC Code

In order to assess the effectiveness of LDPC codes in memory, we implemented a sample LDPC code in System Verilog, a Hardware Description Language (HDL), which can be synthesized into logic gates and simulated on a Field-Programmable Gate Array (FPGA) board.

An interesting point that Gallager makes in Chapter III of his thesis [3] is the fact that the notion of an "optimal" (j, i) is not well defined. Gallager states that a regular LDPC code's limitation of capacity implies that the "best" (j, i) code would perform essentially on the same level as a "typical", i.e. "less optimal" LDPC code with parameters (j, i). Consequently, when implementing an LDPC code in HDL, a convenient parity check matrix was selected. In terms of hardware implementation, a sparse parity check matrix is more advantageous, since fewer gates could be used to implement the encoding and decoding logic.

We implemented an LDPC code parity matrix that had the same rate as the Hamming code we used for comparison. Thus, we calculated that $R = \frac{k}{n} = \frac{64}{72} = 1 - \frac{8}{72} = \frac{j}{i}$. To achieve this rate with the most sparse matrix, we worked with the LDPC code with parameters $n = 72, j = 2$ and $i = 18$.

A. Parity Check Matrix

There are many possible parity check matrices for the above-mentioned parameters. The matrix we used for our implementation is included in our GitHub repository [1] in the file `LDPC_Decoding/H.txt`, since the matrix is too large to be included in this report. The matrix was generated by a Python library called `pyldpc` [4], using the procedure described in section II-C of this report.

The following Python script was used to generate the parity check matrix:

```
from pyldpc import make_ldpc
# parameters (n,j,i) = (72, 2, 18)
n = 72
d_v = 2
d_c = 18
H, G = make_ldpc(n, d_v, d_c, systematic=True)
```

B. Encoding

The encoding was kept quite simple for this LDPC code. The first 64 bits are interpreted as the message bits, and the last 8 bits are the parity bits. The encoding is done by multiplying the message bits with the generator matrix, which

can be constructed from the parity check matrix as described in section II-C. The encoding is done by multiplying the message bits with the generator matrix [7].

C. Decoding

For our HDL implementation, we used the bit-flipping decoding algorithm for LDPC codes. To examine the resulting hardware, the logic gate diagram can be found on the GitHub repository [1], in the file called `LDPC_Decoding/RTL.pdf`. The decoding method uses a Tanner graph to visualize the structure of the LDPC code. Furthermore, a recursive majority vote and bit flipping are used.

To summarize the algorithm in pseudo-code:

Algorithm 1 Bit-Flipping Decoding Algorithm

```
while parity check not satisfied and maximum iteration not reached do
    for each check node do
        if check node is not satisfied then
            vote for flipping the bit
        else
            vote for not flipping the bit
        end if
    end for
    flip the bit corresponding to the majority vote
end while
if parity check is satisfied then
    return the decoded message
else
    return failure
end if
```

D. Performance

For a DE1-SOC FPGA board, the analysis done in the software tool Quartus Prime[1] shows that the highest frequency the hardware can be clocked is bounded by **325.73 MHz**, as estimated by the *slow 1100 mV 85C model*, which is a specification of typical hardware operating conditions (i.e. operating at 1100 millivolts and 85 degrees Celsius). It is a common criterion to analyze the complexity of the combinational logic, which is often quantified by the maximum clock frequency at which the hardware can operate without race conditions.

We include a concise summary of hardware resource usage and performance analysis:

- Combinational ALUT[2] usage for logic: 114
  - 7 input functions[3]: 0
  - 6 input functions: 4
  - 5 input functions: 10
  - 4 input functions: 28
  - $\leq$ 3 input functions: 72
- Dedicated logic registers[4]: 72

IV. Comparison with Hamming Code HDL Implementation

Typical Hamming code implementation uses 64-bit to 72-bit encoding. An input is 64 bits since it corresponds to the size of a typical cache line in memory. Although these parameters do not correspond to an optimal Hamming code, it is a desirable and convenient size for 64-bit computing systems.

The Single Error Correction Double Error Detection (SECDED) Hamming code can detect and correct single-bit errors, as well as detect double-bit errors.

The main concept behind the practical hardware implementation of the Hamming code is to detecting errors in such a way that the ensemble of parity bits, if non-zero, corresponds exactly to the index of the corrupted bit.

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	17	18	19	20	21	22	23
3	24	25	26	27	28	29	30	31
4	32	33	34	35	36	37	38	39
5	40	41	42	43	44	45	46	47
6	48	49	50	51	52	53	54	55
7	56	57	58	59	60	61	62	63
8	64	65	66	67	68	69	70	71

Fig. 2. Indices of parity bits and data bits in a Hamming code.

To achieve this error indexing feature, we assign each codeword bit an index, and ensure that the parity bits are placed at indices 2^m, for increasing natural numbers m until the end of the codeword is reached. For example, Fig. 2 shows a codeword that is organized in tabular form. The parity bits are highlighted in yellow, and the remaining bits are the data bits of the original message, except for the zeroth bit, which acts as an additional parity check bit for double-error detection. Furthermore, as described above, we notice that the highlighted bits correspond to indices $2^0 = 1$, $2^1 = 2, 2^2 = 4, \ldots, 2^6 = 64$, since there is a total number of 72 bits shown. For an interactive example, please see the Excel spreadsheet `Hamming_code_demo.xlsx` in our GitHub repository [1].

[1] https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html

[2] An FPGA board uses configurable logic blocks instead of hard-wired combinations of logic gates. These configurable logic units are are called Adaptive Look-up Tables, or ALUTs.

[3] An n-input function is an array of logic blocks that take an n-bit input. It is desirable to have few blocks where n is large, i.e. 2-input blocks are preferred.

[4] Logic registers are used to store temporary data during hardware computation. A register can be thought of as a "memory cell".

	0	1	2	3	4	5	6	7
0		1	1	1	0	0	0	0
1	0	0	1	0	1	0	0	0
2	0	0	1	1	1	1	0	1
3	0	0	1	1	0	0	1	0
4	0	0	0	1	0	1	1	0
5	1	0	1	0	1	0	1	1
6	1	1	0	0	1	0	1	0
7	0	1	0	1	0	1	1	0
8	0	0	1	0	1	1	0	0

Fig. 3. Parity bits and data bits in a Hamming code.

There is a total of 72 bits, whose range can be expressed in binary form as $0000000 - 1000111$. Note that any number within the range can be written down using exactly 7 bits, which correlates with the fact that the code has 7 parity bits. We let `vec[6:0]` be the unique position vector as and let there be a bijective mapping from it to the parity bits `p[6:0]`.

If we let `p[0]` monitor all of the data bits where the `vec[0]` is 1, `p[1]` monitors all of the data bits where the `vec[1]` is 1, and so on, then the erroneous bit will be indicated by the whole group of parity bits, arranged in order by index. Note that the parity bit itself is included in the XORing calculating as well. Fig. 3 illustrates an example scenario where the values of the parity bits `p[0]...p[7]` are set.

A. FPGA Implementation

Due to the nature of the Hamming code, it is easier to implement in HDL than an LDPC code; the error bit can be found by merely XORing the parity bits. Our implementation of Hamming code encoding and decoding can be found in our GitHub repository [1].

The synthesized hardware can be examined in the files `Hamming72out/Hamming72out_RTL.pdf` (the decoding logic) and `Hamming64in/Hamming64inRTL.pdf` (the encoding logic).

Although there are nearly 71 XOR gates on a single combinational logic path, the time delay is not significant. According to Quartus timing analysis, the clock frequency permitted by the design is **387.3 MHz**, as estimated with the *slow 1100 mV 85C model*.

As with the LDPC implementation, we include a short summary of resource usage and performance analysis:

- Combinational ALUT usage for logic: 154
  - 7 input functions: 0
  - 6 input functions: 121
  - 5 input functions: 10
  - 4 input functions: 13
  - ≤ 3 input functions: 10
- Dedicated logic registers: 69

V. Conclusion

The HDL implementation of both LDPC and Hamming codes revealed the practical challenges behind error correction in memory. In addition to speed and error-correcting capacity, it is desirable for hardware to use a minimal number of logic gates and registers for encoding and decoding.

After comparing the two implementations, we note that the two codes performed fairly similarly, in terms of required logic gates and the maximum clock frequency (and thus speed) of operation. The most significant attribute that was noticeable in the comparison was the fact that the regular LDPC code required fewer 6 input functions than the Hamming code, which would be more desirable since large input functions occupy more area on chips. We therefore conclude that a simple regular LDPC code does not offer a significant advantage when compared with a Hamming code with the same rate. However, we concede that existing literature [6][7][8] points out the potential of LDPC codes for special applications that require more flexible codes. In particular, irregular LDPC codes, despite their higher complexity, attain better performance in these fields.

References

[1] T. Wang and N. Balashov, ELEC433-Projects, 2024, GitHub repository, https://github.com/luckunately/ELEC433-Projects.

[2] B. Kurkoski, Introduction to Low-Density Parity Check Codes, https://www.jaist.ac.jp/~kurkoski/teaching/portfolio/uec_s05/S05-LDPC%20Lecture%201.pdf.

[3] R.G. Gallager, Low-Density Parity-Check Codes. Cambridge, MA: MIT Press, 1963 (Sc.D. MIT, 1960).

[4] H. Janati, pyldpc, 2020, GitHub repository, https://github.com/hichamjanati/pyldpc.

[5] B. Panzer-Steindel, Data integrity. CERN/IT, 2007, retrieved from https://indico.cern.ch/event/13797/contributions/1362288/attachments/115080/163419/Data_integrity_v3.pdf.

[6] S. Jeon, E. Hwang, B. V. K. V. Kumar and M. K. Cheng, "LDPC Codes for Memory Systems with Scrubbing", 2010 IEEE Global Telecommunications Conference GLOBECOM 2010, Miami, FL, USA, 2010, pp. 1-6, doi: 10.1109/GLOCOM.2010.5683367.

[7] S. Verma, S. Sharma, (2016) 'FPGA implementation of low complexity LDPC iterative decoder', International Journal of Electronics, 103(7), pp. 1112-1126. doi:https://doi.org/10.1080/00207217.2015.1087052.

[8] Bohua Li, Yukui Pei, and Wujie Wen. 2018. Efficient LDPC Code Design for Combating Asymmetric Errors in STT-RAM. J. Emerg. Technol. Comput. Syst. 14, 1, Article 10 (January 2018), 20 pages. doi: https://doi.org/10.1145/3154836.