

LDPC Codes In Computer Memory

ELEC 433 Final Project

Tom Wang

Natalie Balashov

Abstract—TODO: add short abstract here.

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 memory sizes 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 has a column weight of j and a row weight of i [2]. For example, a parity-check matrix for a regular LDPC code where $j = 3$ and $i = 4$ may be 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 $R = \frac{k}{n} = 1 - \frac{i}{j} = \frac{1}{4}$.

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.

B. Tanner Graphs

A Tanner graph is a bipartite graph 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.

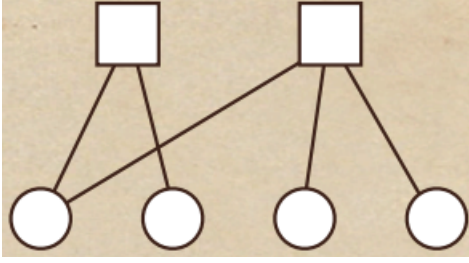


Fig. 1. Example of a figure caption.

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.

C. Encoding and Generator Matrices

Given a parity-check matrix H for an LDPC code, the generator matrix G can be constructed in the following way:

- 1) Assume that the encoded code c of length n is composed of the parity bits b and the message bits m in a form $c = [b, m]$.
- 2) Then if there are no errors in the code, $cH^T = [b, m]H^T = 0$.
- 3) Divide H into $H = [H_1, H_2]^T$ where H_1 is a square matrix of size $n - k$. Note that b is of length $n - k$ as well. So we can break the matrix multiplication into $[b, m][H_1, H_2]^T = bH_1^T + mH_2^T = 0$.
- 4) Since we are in a binary field, $bH_1^T = mH_2^T \implies b = mH_2^T(H_1^T)^{-1}$. Name the matrix $H_2^T(H_1^T)^{-1}$ as A .
- 5) So the generator matrix G is $[A, I_{n-k}]$.

D. Decoding and Parity Check Matrices

TODO

III. IMPLEMENTATION OF A REGULAR LDPC CODE IN HDL

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 for simplicity.

We implemented an LDPC code parity matrix that could achieve a rate similar to the hamming code we used for comparison. $R = \frac{k}{n} = \frac{64}{72} = 1 - \frac{8}{72} = \frac{j}{i}$.

To achieve this, we can attempt to implement encoding and decoding schemes for the LDPC code with parameters $n = 72, j = 2, i = 18$, or the LDPC code with parameters $n = 72, j = 4, i = 36$.

A. Parity Check Matrix

There are lots of possible parity check matrices, the one we will use would be linked in our GitHub repo since it is too large to be included in this report [1]. This is generated by a Python library called `pyldpc` [4].

B. Encoding

Encoding is quite simple for LDPC codes. The first 64 bits are 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 the previous section.

C. Decoding

We will use Bit-Flipping decoding algorithm for the LDPC code. The encoding is done by multiplying the message bits with the generator matrix [7]. The generator matrix is constructed from the parity check matrix as described in the previous section.

It uses Tanner graph to visualize the structure of LDPC code. Recursive majority vote and bit flipping are used to decode the LDPC code.

To summarize the algorithm:

Algorithm 1 Bit-Flipping Decoding Algorithm

```

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

```

RTL viewer can be found on GitHub for TODO!!!![1].

D. Performance

On DE1-SOC, Quartus analysis shows that the highest frequency it can run is **325.73 MHz** under *slow 1100 mV 85C mode*. (Remark: It is a common analysis criteria to analyze the complexity of combinational logic path. It limits the maximum frequency of the circuit.)

Here is a summary of resources usage and performance analysis by Quartus Prime:

- Combinational ALUT usage for logic: 114
 - 7 input functions: 0
 - 6 input functions: 4

- 5 input functions: 10
- 4 input functions: 28
- ≤ 3 input functions: 72
- Dedicated logic registers: 72

IV. COMPARISON WITH HAMMING CODE HDL IMPLEMENTATION

Typical Hamming code implementation uses 64-bit to 72-bit encoding. Input is 64 bits since it is a typical cache line size. Although it is not an optimal hamming code, it is a good visit pattern for computing systems.

This typical implementation is called **SECDED** (Single Error Correction, Double Error Detection) Hamming code. It can detect and correct single-bit errors and detect double-bit errors.

A. Implementation

The concept of Hamming code is detecting errors in half of the codewords. And then with different combinations of deleting patterns, we can locate a single bit.

	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. 2. Parity bits and data bits in a Hamming code.

Since the output is more than 64 bits, we need 7 parity bits as shown in 2. For an interactive example, check out the Excel sheet in our GitHub Repo[1].

The advantage of this kind of implementation would be that the difference of parity bits is the coordinate of the error bit which makes it easier to find the error bit.

1) *Example of correcting error:* We order the bits as shown below:

There are a total of 72 bits, which can be expressed in binary from $6'b000_0000$ to $6'b100_0111$. Note that there are exactly 7 bits which corresponds to 7 parity bits. Mark the unique position vector as $vec[6:0]$ and let it be bijective to parity bits $p[6:0]$.

Now if we let $p0$ monitor all of the bits where the $vec[0]$ is 1, $p1$ monitor all of the bits where the $vec[1]$ is 1, and so

	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. 3. Number parity bits and data bits in a Hamming code.

on, we can find the error bit by checking the parity bits. Note that the parity bit itself is included in monitoring as well.

This way, if we calculate the parity bits and compare them with the received parity bits, we can find the error bit. Since the parity bits are the coordinate of the error bit, we can find the error bit by XORing the parity bits.

B. Implementation on FPGA

Because of the nature of the Hamming code, it is easier to implement on FPGA. The error bit can be found by XORing the parity bits. The implementation can be found in our GitHub Repo[1].

RTL viewer can be found on GitHub for Hamming72out and Hamming64in.

Although there are nearly 71 xor on a single combinational logic path, the time delay is not significant. According to Quartus timing analysis, the highest frequency it can run is **387.3 MHz** under *slow 1100 mV 85C mode*.

Here is a summary of resources usage and performance analysis by Quartus Prime:

- 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

TODO: add conclusion

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>