6-4-2019

A Flexible BCH decoder for Flash Memory Systems using Cascaded BCH codes

Arul K. Subbiah

Santa Clara University

Department of Electrical Engineering

Date: June 4, 2019

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY

Arul K. Subbiah

ENTITLED

A Flexible BCH decoder for Flash Memory Systems using Cascaded BCH codes

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY IN ELECTRICAL ENGINEERING

Dr. Tokunbo Ogunfunmi (Thesis Advisor)	**Dr. Nam Ling**
Dr. Shoba Krishnan	**Dr. Sim Narasimha**
(Department Chair)	**Dr. Yuling Yan**

A Flexible BCH decoder for Flash Memory Systems using Cascaded BCH codes

By

Arul K. Subbiah

Dissertation

Submitted in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy
in Electrical Engineering
in the School of Engineering at
Santa Clara University, 2019

Santa Clara, California

The night before he died Galois sensing his end was near wrote down many letters, both mathematical and political to his numerous friends and brother. These letters contained some very thought provoking mathematical ideas that has forever sealed Galois name in the annals of mathematical wizardry.

The famous mathematician Hermann Weyl while describing these letters said

"***This letter judged by the novelty and profundity of ideas it contain, is perhaps the most substantial piece of writing in the whole literature of mankind.***"

Acknowledgments

I would like to express my sincere gratitude to Professor *Tokunbo Ogunfunmi*, my advisor, who has provided invaluable support and guidance throughout the years I have been pursuing my Ph.D degree at Santa Clara University. I would also like to thank the members of my doctoral committee, Professors *Shoba Krishnan*, *Nam Ling*, *Madihally (Sim) Narasimha*, and *Yuling Yan* for their valuable comments and suggestions.

I have received assistance and ideas from many people at work over the years and would be impossible to enumerate. However, some of the major contributors are *Prakash Kamath* for introducing me to challenging areas in ECC, *Sudha Gopakumar* and *Joe Keirouz* for introducing me to mathematical aspects of abstract algebra and practical applications of it. I am also thankful to my fellow colleagues in the SPRL research group for their help and insightful discussions. My special thanks to my *San-Diego friends* and *Melvin Thomas* for their friendship and their invaluable moral support.

Finally, I would like to express my appreciation to my family for their continuous support and encouragement. Without their belief, support, endurance and love, this work could not have been even started, let along finished. Especially, I would like to thank my wife *Muhilmathi Shanmugaasokan*, and my lovable kids *Sarvesh & Sashanth*; I could not have come this far without their understanding and support.

A Flexible BCH decoder for Flash Memory Systems using Cascaded BCH codes

Arul K. Subbiah

Department of Electrical Engineering
Santa Clara University
Santa Clara, California
2019

ABSTRACT

NAND flash memories are widely used in consumer electronics, such as tablets, personal computers, smartphones, and gaming systems. However, unlike other standard storage devices, these flash memories suffer from various random errors. In order to address these reliability issues, various error correction codes (ECC) are employed. Bose-Chaudhuri Hocquenghem (BCH) code is the most common ECC used to address the errors in modern flash memories. Because of the limitation of the realization of the BCH codes for more extensive error correction, the modern flash memory devices use Low-density parity-check (LDPC) codes for error correction scheme. The realization of the LDPC decoders have greater complexity than BCH decoders, so these ECC decoders are implemented within the flash memory device. This thesis analyzes the limitation imposed by the state of the art implementation of BCH decoders and proposes a cascaded BCH code to address these limitations.

In order to support a variety of flash memory devices, there are three main challenges to be addressed for BCH decoders. First, the latency of the BCH decoders, in the case of no error scenario, should be less than 100us. Second, there should be flexibility in supporting different ECC block size; more precisely, the solution should be able to support 256, 512, 1024, and 2048 bytes of ECC block. Third, there should be flexibility

in supporting different bit errors.

A recent development with Graphical Processing Units (GPUs) has attracted many researchers to use GPUs for non-graphical implementation. These GPUs are used in many consumer electronics as part of the system on chip (SOC) configuration. In this thesis we studied the limitation imposed by different implementations (VLSI, GPU, and CPU) of BCH decoders, and we propose a cascaded BCH code implemented using a hybrid approach to overcome the limitations of the BCH codes. By splitting the implementation across VLSI and GPUs, we have shown in this thesis that this method can provide flexibility over the block size and the bit error to be corrected.

Table of Contents

1 Introduction 1

1.1 Flash Memory 1

1.1.1 Memory Cell 2

1.1.2 Memory organization 3

1.1.3 GPUs in SOCs 5

1.2 Motivation 5

1.2.1 Long BCH codes 5

1.2.2 Different sector size support 7

1.3 Contributions 7

1.3.1 Long BCH codes 8

1.3.2 Different sector size support 9

1.4 Outline of the thesis 10

2 Background 11

2.1 Preliminaries 11

2.1.1 Galois field 12

2.1.2 BCH code 13

2.1.3 BCH Encoder 14

2.1.4 BCH Decoder 15

2.1.4.1 Syndrome Generation 15

2.1.4.2 Key-equation solver 15

2.1.4.3 Error Locator 16

2.2 NAND Flash Error Analysis 16

2.3 Previous Work 17

2.3.1 CPU implementation 17

2.3.2 VLSI implementation . . . 19
2.3.2.1 Encoders . . . 19
2.3.2.2 Decoders . . . 21
2.3.3 GPU implementation . . . 22

3 VLSI Implementation . . . 24
3.1 Encoder . . . 25
3.2 Hardware Decoder . . . 26
3.2.1 Syndrome Generator . . . 28
3.2.2 Key-Equation Solver . . . 29
3.2.3 Error Corrector . . . 32
3.2.4 Timing Diagram . . . 33
3.2.5 Spare Area Management . . . 34
3.3 Multimode Encoder and Syndrome Generator . . . 35
3.3.1 Modified Encoder . . . 36
3.3.2 Residual Checker . . . 37
3.4 Programmable SRU . . . 39
3.4.1 Multimode Encoder and Syndrome Generator . . . 40
3.5 Experimental Results . . . 40
3.6 Summary . . . 43

4 GPU Implementation . . . 44
4.1 GPU system . . . 44
4.2 GPU Kernel routines . . . 46
4.2.1 Syndrome Generator . . . 48
4.2.2 Parallel Syndrome Generation . . . 49
4.2.2.1 Polynomial division . . . 51
4.2.2.2 Splitting received code . . . 52
4.2.2.3 Kernel implementation . . . 53
4.2.3 Key-equation solver . . . 54
4.2.4 Chien Search Kernel . . . 56

4.3 Memory enhancement . . . 57
4.3.1 Enhanced memory flow chart . . . 58
4.4 Experimental results . . . 60
4.4.1 GPU kernel performance . . . 62
4.4.1.1 GPU computation time . . . 62
4.4.1.2 Syndrome computation time . . . 63
4.4.1.3 Individual Kernel computation time . . . 64
4.4.2 Syndrome Generator with parallel division . . . 65
4.4.3 Memory Enhancement . . . 65
4.4.3.1 Setup1 . . . 66
4.4.3.2 Setup2 . . . 67
4.4.4 CUDA profile . . . 68
4.5 Summary . . . 68

5 Cascaded hybrid approach . . . 71
5.0.1 Main Idea . . . 72
5.0.2 Data Flow . . . 74
5.0.3 Error Analysis . . . 75
5.1 Hybrid Implementation . . . 77
5.1.1 Flash Data Layout . . . 78
5.1.2 Hybrid Flow Chart . . . 80
5.1.3 VLSI Implementation . . . 82
5.1.4 GPU Implementation . . . 84
5.1.4.1 Syndrome Generator . . . 84
5.1.4.2 Key-Eq Solver . . . 85
5.1.4.3 Chien Search . . . 85
5.2 Experimental Results . . . 86
5.2.1 GPU Analysis . . . 87
5.2.2 VLSI Analysis . . . 90
5.3 Conclusion . . . 90

6 Conclusion 94
6.1 Long BCH code 94
6.1.1 Breaking Syndrome computation 95
6.1.2 Hybrid approach 96
6.2 Future Work 97

Bibliography 98

List of Figures

1.1 NAND Flash cell 2
1.2 NAND Flash cell types 3
1.3 Flash Array organization 4
1.4 Generator Polynomial degree vs bit errors 6
1.5 Generator Polynomial degree vs bit errors (proposed) 8
1.6 Shared encoder and Syndrome generator 9

2.1 UBER for different sector sizes 18
2.2 Elastic ECC for NAND flash memory 19
2.3 Hardware implementation of long BCH encoder 20

3.1 Hardware Decoder 24
3.2 BCH encoder in serial fashion 25
3.3 Timing diagram for encoder 26
3.4 VLSI BCH Decoder 27
3.5 Syndrome Generator 28
3.6 Syndrome Generator systolic array 29
3.7 iBMA Key Equation Solver 30
3.8 Error corrector 32
3.9 Timing diagram for Error scenario 33
3.10 Timing diagram for non-error scenario 34
3.11 Spare memory area management 35
3.12 Multimode Encoder and Syndrome Decoder 36
3.13 Modified Parity generator 37
3.14 Modified Parity Checker 38
3.15 SRU systolic array 39

3.16 Syndrome generator and Encoder 40
3.17 MESG systolic array 41

4.1 GPU thread organization 45
4.2 Organization of memories within GPU 46
4.3 Flow chart for GPU implementation 47
4.4 Flow chart using parallel syndrome generation 50
4.5 Parallel polynomial division 53
4.6 Enhanced memory organization of Flash memory 58
4.7 Enhanced memory organization of Flash memory 59
4.8 GPU setup2 61
4.9 GPU comp. time 62
4.10 SK comp. time 63
4.11 Individual Kernel comp. time 64
4.12 Bit Error vs. GPU Syndrome Computation time 66
4.13 Memory enhancement for Setup1 67
4.14 Memory enhancement for Setup2 68
4.15 CUDA Profile 70

5.1 Cascaded BCH code 72
5.2 Data Flow for cascaded ECC method 74
5.3 Raw bit error vs. Sector error 76
5.4 Hybrid approach block diagram 77
5.5 Decoder Execution sequence 78
5.6 Data block layout for cascaded ECC method 79
5.7 Hybrid imp. flow chart 81
5.8 Hardware design of MRPU array 83
5.9 Generator polynomial degree vs bit error 88
5.10 Syndrome computation time for different arch. 89
5.11 Total computation time for different architecture for different sector size 92

6.1 Generator Polynomial degree vs bit errors (proposed) 95
6.2 Hardware design of MRPU array 96
6.3 Hybrid approach block diagram 97

List of Tables

2.1 *GF Addition* 13
2.2 *GF Multiplication* 13

3.1 BCH Encoder logic comparison 41
3.2 BCH Encoder area comparison 42

4.1 Experimental setup1 60
4.2 Experimental setup2 60

5.1 NAND Flash characteristic 87
5.2 Experimental setup 87
5.3 Area comparison for different syndrome generators 91

Chapter 1

Introduction

Flash memories are widely used in embedded systems because of low-power consumption, non-volatility, high random access performance, and a small physical footprint. More precisely, NAND flash memories are used as mass storage devices in consumer electronic devices such as digital cameras, smartphones, tablets, personal computers, and gaming systems [1, 2, 3, 4]. Although the cost of flash memory is higher than the hard drives, the fast access to the memory content makes it a more attractive solution. The technology scaling of VLSI has enabled flash memory vendors to integrate more bit cells thus having compact devices with more memories, which has led to the replacement of hard drives with Solid State Drives in more electronics devices[5]. Also, the need for more significant storage in consumer electronic devices has constrained systems to use flash memories as part of their ecosystem. There are two main challenges imposed by the flash memories which are reliability and block management[6]. We will improve the reliability of NAND flash by error correction code (ECC) techniques; more precisely, our focus will be on the Bose-Chaudhuri-Hocquenghem (BCH) code implementation.

1.1 Flash Memory

The technology scaling of VLSI has enabled flash memory vendors to integrate more bit cells thus having compact devices with more memories, which has led to the replacement of hard drives with Solid State Drives in more electronics devices[5].

1.1.1 Memory Cell

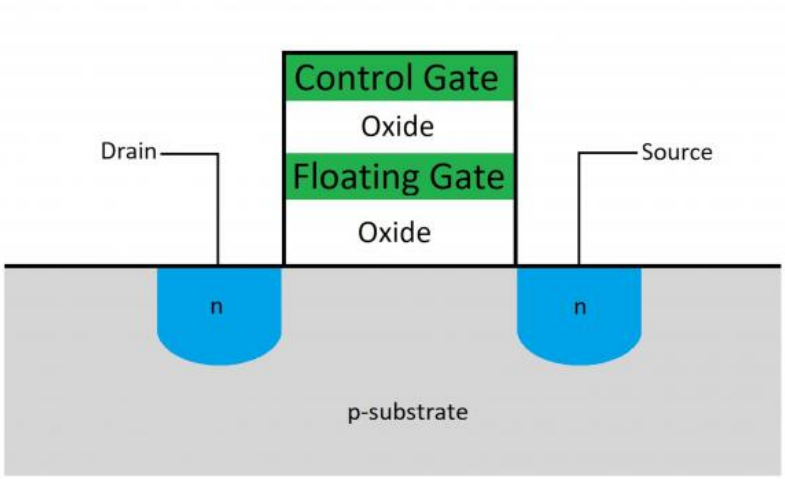


Fig. 1.1: NAND Flash cell

A flash memory cell is an n-channel metal-oxide-silicon (MOS) transistor with a floating gate (as shown in Fig.1.1), and the value of the data bit stored is represented by the ability to segregate the voltage level stored in this cell. The flash memory cells are classified into three different classes single-level cell (SLC), multi-level cell (MLC), and Tri-level cell (TLC). As the name implies, the SLC can store one-bit, the MLC can store 2-bit, and the TLC can store 3-bit information. Fig.1.2 depicts how the voltage level is used to differentiate for SLC,MLC and TLC memory bits. The reliability of the SLC devices is higher than the MLC Aand TLC devices, but the cost is high for such devices. On the contrary, the MLC and TLC devices are cheaper, but they are densely populated and have a short retention time. Because of the densely packed cells within a device, it is exposed to more random errors. The raw bit error rate (RBER) of the MLC and TLC NAND flash memory is about 10^{-6}, and at least two orders of magnitude worse than that of the SLC NAND flash memory[7]. Therefore, we employ different error correction codes (ECC) techniques like Hamming, Bose-Chaudhuri-Hocquenghem (BCH), Reed-Solomon (RS), and linear-density parity check (LDPC) codes[8, 9, 10, 11] to overcome

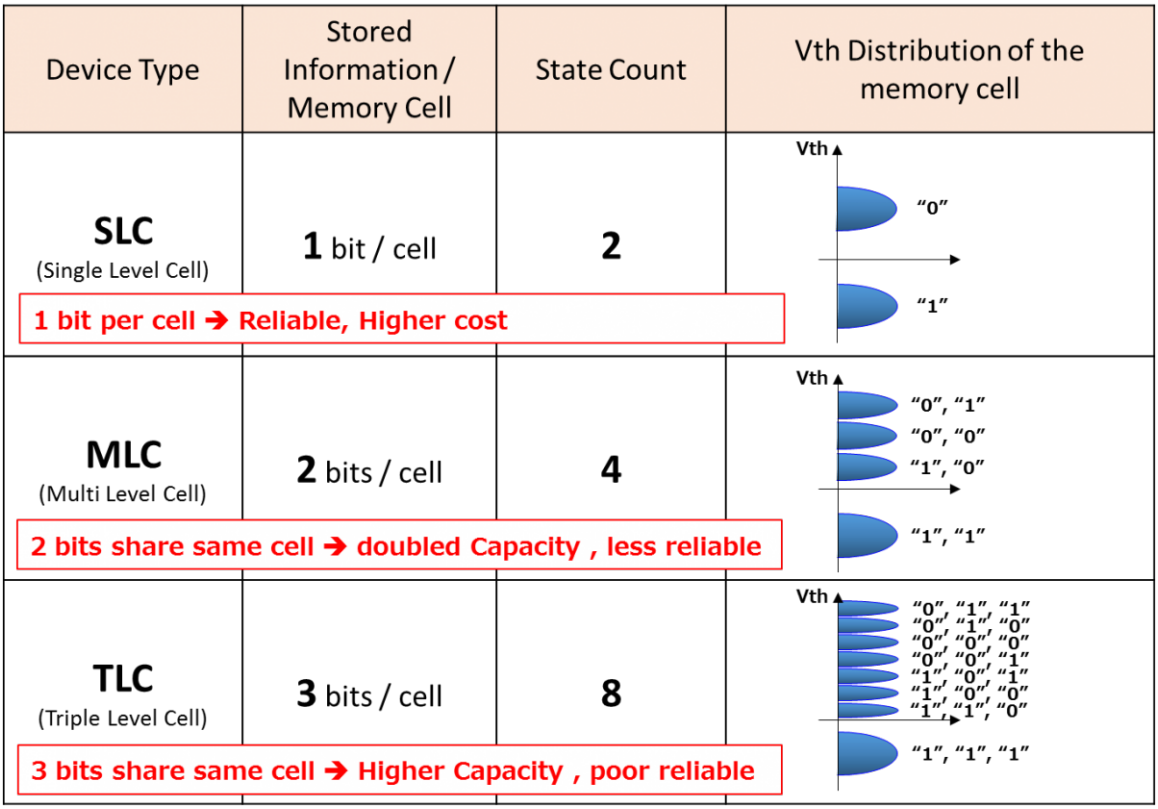


Fig. 1.2: NAND Flash cell types

these errors. For SLC devices, the most commonly used ECC is Hamming and BCH codes. For MLC and TLC devices, the BCH, RS, and LDPC codes are used. Since BCH code [12] is the most commonly used code across different flash memory types we focus our research on these types of code.

1.1.2 Memory organization

Fig.1.3 depicts the memory organization of a flash memory device. In this device, each page is 2112 bytes in length with 2048 bytes dedicated for data and 64 bytes for the spare area. A group of 64 pages makes a block, and 4096 blocks make a device plane. In some devices, there are more than one planes per device. The read, write and erase operations to these flash memories are asymmetric in latency and execution time. So,

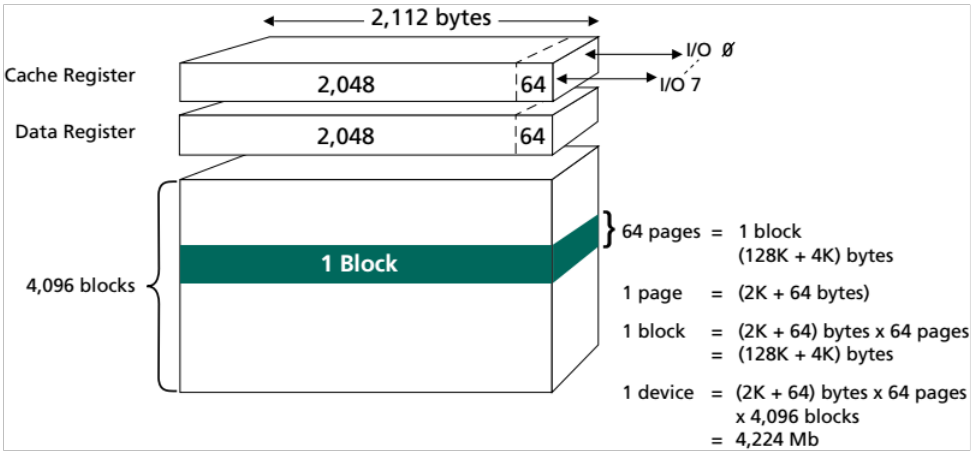


Fig. 1.3: Flash Array organization

managing these blocks memories in a flash has to be addressed with special software. Besides, NAND flash memories are not reliable because of their limited life cycle and interference which is addressed by ECC codes. Although the ECC can enhance the reliability of the page data, the system software often uses special software [13] called flash translation layer (FTL) to address block management. The FTL was introduced to provide a logical to physical address mapping mechanism, which provides the support for out of place update for flash memories. In addition to the logical mapping, the FTL captures the memory writes/reads and avoids any additional write/erase operation so that it could improve the lifetime of the flash memory systems. The mapping schemes offered by FTL are: block mapping, page mapping, and the hybrid mapping scheme. These mapping schemes enable garbage collection to perform efficiently [2, 4] which leads to better performance. However, these schemes demand faster Dynamic-Random Access Memory (DRAM) on the host systems for it to have a useful table.

1.1.3 GPUs in SOCs

Technology scaling has rendered the ability to integrate multiple GPUs within System On Chips (SOC), which has paved the way for many researchers to use the GPUs for many nongraphical applications. The term General Purpose Graphical Processing Unit (GPGPU) refers to the application of GPUs for nongraphical computation. Streaming Multiprocessors (SM) are the building blocks of the GPUs, which is similar to multiple CPUs. Each of the instantiated SM is capable of handling multiple threads which are scheduled by a warp scheduler. To program these Streaming Multiprocessors, we need an exclusive compiler like Computer Unified Device Architecture (CUDA) C[14] software. The CUDA software creates the necessary kernel routines which in turn creates the Single Instruction Multiple Data (SIMD) for the processors. The kernel subroutine executes across multiple cores and multiple threads. We use the term GPU throughout this paper since the GPU and GPGPU terms are interchangeable.

1.2 Motivation

There are many practical challenges to implement BCH codes. We found the following limitations intriguing and propose a solution in our thesis.

1.2.1 Long BCH codes

The number of parity bits ($plength$) to be generated is given as $plegnth = deg(g(x))$ where $g(x)$ is the generator polynomial. In most cases the degree of $g(x)$ is given as $deg(g(x) = m * t$, where m is the GF dimension and t is the number of bit errors to be corrected. Fig. 1.4 plots the number of parity bits to be generated versus bit error for different GF dimensions. For bit errors greater than eight, the length of the

parity bits to be generated is considerable, and there are two issues to be addressed. First, the feedback path of the LFSR style encoder is greater than two hundred for bit errors greater than fifteen; hence it impacts the frequency of the circuit for VLSI implementation. Second, most VLSI implementations require unfolded circuits of the LFSR for encoders, and they have a huge fan-out issue for the XOR gates. This is one of the reasons why LDPC codes are chosen for bit errors greater than twenty.

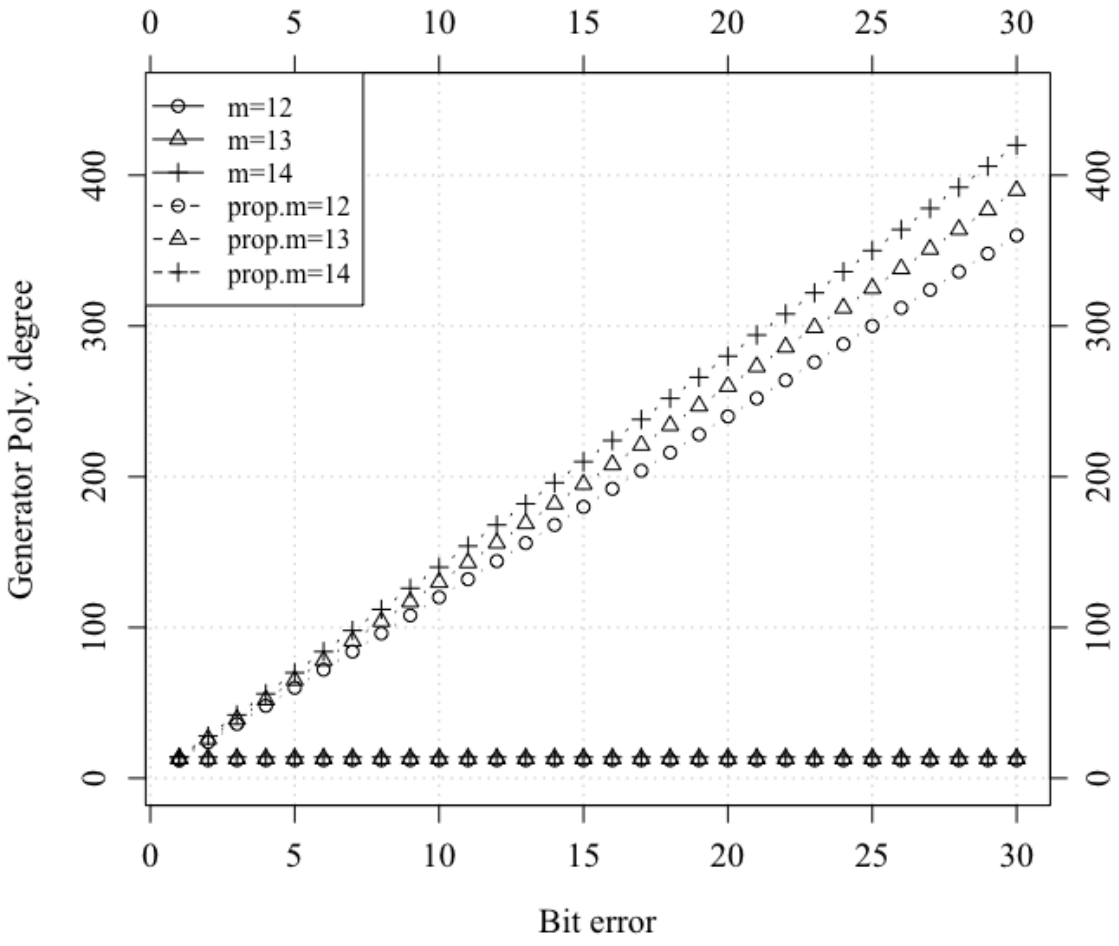


Fig. 1.4: Generator Polynomial degree vs bit errors

1.2.2 Different sector size support

There are many flash devices available with different bit error correction requirements. So, it is essential for host systems to be able to support many sector sizes for flash memory. The sector size dictates the GF dimension where the BCH code resides, and the most common GF dimensions used are $m = 11, .., 15$. There are many software solutions proposed [15, 16, 17] to support these multiple sector sizes for flash memories, but they lack good performance because of the CPU implementation. Besides, the proposed solution should have high throughput and high performance for read transactions from the flash memory. Chang et al. [18] proposed the parity bits for multiple pages to be accumulated and stored in the user data area. An attempt to support multiple $GF(2^m)$ in hardware was proposed by B. Park et al. [19], but the area consumed to support different dimensions are enormous. Another issue to support different sector size is the management of the spare area for the parity bits. Because of the spare area limitation within a page, some flash devices need dedicated pages to store the ECC parity bits [17]. This method degrades the performance for VLSI implementation of BCH decoders.

1.3 Contributions

In this thesis, we propose a novel method to generate the BCH encoded bits using minimal residual polynomial (MRP) architecture, and then protect the MRP bits using a cascaded BCH code(C-BCH).

1.3.1 Long BCH codes

We split the generator polynomial $g(x)$ into multiple minimal polynomials $\phi_i(x)$, where $i = 1..t$, to address the long feedback path in VLSI implementation [20]. This method also addresses the fan-out issue with large $g(x)$, and the encoder is only dependent on the GF dimension and not on the number of bit errors. Fig. 1.5 plots the bit errors vs generator polynomial degree (parity bits) for various GF dimensions. Since the $g(x)$ is split into multiple $\phi_i(x)$, the parity bits are exposed to multiple bit errors. This we address by using a C-BCH code for the concatenated parity bits[21].

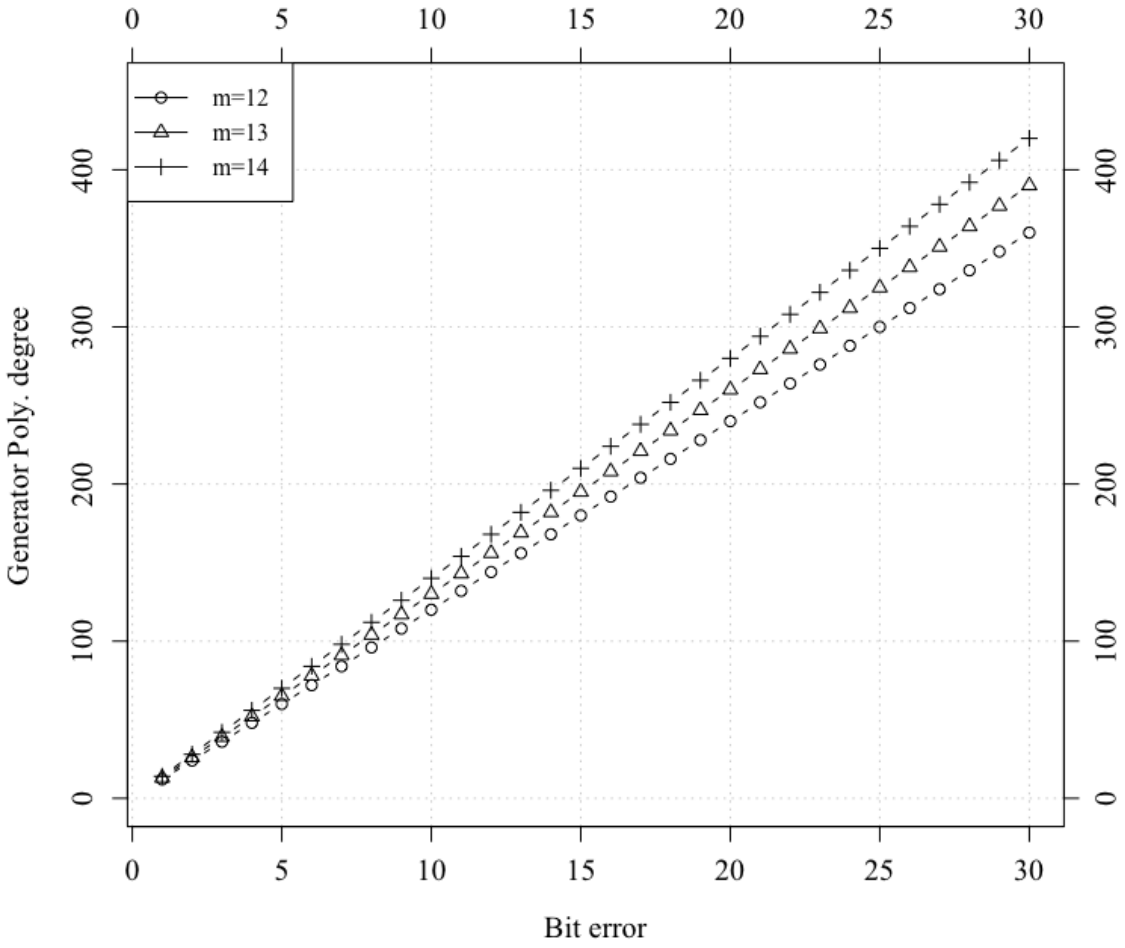


Fig. 1.5: Generator Polynomial degree vs bit errors (proposed)

1.3.2 Different sector size support

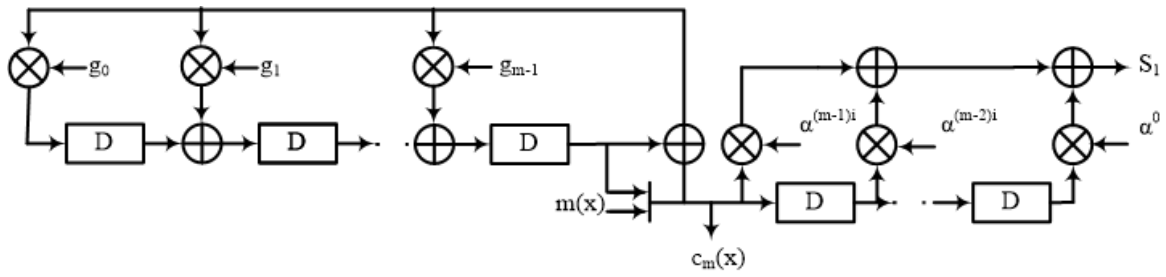


Fig. 1.6: Shared encoder and Syndrome generator

We have analyzed the different architectures like VLSI [20, 22] and GPUs for the BCH decoders [23, 24, 25, 26]. The hardware solution has the best performance and throughput, but it lacks support for multiple GF. In our thesis, we use the hybrid approach [27] where hardware is used for partial syndrome generation and the GPUs are used for syndrome generation, key-equation solver, and Chien search algorithm. By splitting the syndrome generation unit into residuals of minimal polynomials, we can support multiple GF dimensions in hardware thus harnessing the performance of hardware for pages without error. Also, we propose to use the minimal residual polynomial, for each minimal polynomial of the BCH code, to be concatenated to form the modified BCH parity bits [21]. This parity bit is then protected by cascading the parity bits using a conventional BCH code [21]. By using this method, the same hardware could be used for both encoding and syndrome residual generation thus saving VLSI area as shown in Fig.1.6. We finally propose to use the hardware to generate the MRP during encoding and to protect the same using C-BCH code. The same hardware could be used for partial syndrome generation, and in an error scenario, the GPUs are used for correcting the errors.

1.4 Outline of the thesis

Here is a brief outline of our thesis. Chapter. 2 explains the preliminaries, the theory behind the BCH codes, and it discusses the previous work related to our thesis. Chapter. 3 discusses the VLSI implementation of BHC decoders [22]. Here we discuss how area reduction could be achieved by joining the encoder and the syndrome generator [20]. Then, we analyze the GPU implementation of the BCH decoders in Chapter. 4. Here we study the GPU implementation using iBMA algorithm [23] and compare it against CPU implementation. Then we discuss how the memories for the spare area can be organized to support multiple sector sizes [24]. We also study the performance of GPUs for Reed Solomon codes [28]. Further, we study optimization techniques that could be used for GPU implementation [25, 26]. In Chapter. 5 we discuss our proposed C-BCH code [21] and hybrid implementation [27] of the same. Finally, we conclude our thesis in Chapter. 6 and propose future work.

Chapter 2

Background

There have been several methods proposed to enhance the reliability of the flash memories using write reduction, wear leveling and density reduction [1, 2, 3, 4]. Another level of enhancing reliability is by ECC methods [8, 9, 10, 11, 28]. The most common ECC methods used for NAND flash are BCH, RS, and LDPC. The BCH codes can be used to correct errors for SLC, MLC and TLC NAND devices. For TLC devices, the number of bit errors to be corrected per sector is higher than 12 bits, and this requires longer BCH codes. D. Kim et al. proposed a concatenated BCH codes to address long BCH codes in [29]. This method needs very long parity bits to be stored in the spare area. In our thesis, we propose an alternate solution to address these long BCH codes using C-BCH codes. Broadly, the enhancement using ECC can be classified according to their implementation: CPU implementation, VLSI implementation, and GPU implementation. First, we explain the preliminaries required for the block codes; then we explain the BCH codes and how they are constructed.

2.1 Preliminaries

In this section, we discuss the mathematical aspects required to understand the BCH codes. More details on the mathematical aspect to the BCH codes can be found in [30].

2.1.1 Galois field

A field $< \mathbb{F}, +, . >$ is a set of objects $\mathbb{F}$ on which the operation of addition and multiplication operates with the following properties

- $\forall a, b \in \mathbb{F}$, a+b is closed under addition
- $\forall a, b, c \in \mathbb{F}$, a+(b+c) = (a+b)+c (associative)
- $\forall a \in \mathbb{F}, \exists\, 0$ such that a+0=a (additive identity)
- $\forall a \in \mathbb{F}, \exists\, b \in \mathbb{F}$ such that $a + b = 0$ (additive inverse)
- $\forall a, b \in \mathbb{F}$, a+b = b+a (commutative)
- $\forall a.b \in \mathbb{F}$, a+b is closed under multiplication
- $\forall a, b, c \in \mathbb{F}$, a.(b.c) = (a.b).c (associative)
- $\forall a \in \mathbb{F}\, except\, 0, \exists\, 1 \in \mathbb{F}$ such that a.1=a (multiplicative identity)
- $\forall a \in \mathbb{F}\, except\, 0, , \exists\, b \in \mathbb{F}$ such that $a.b = 1$ (multiplicative inverse)
- $\forall a, b \in \mathbb{F}$, a.b = b.a (commutative)

when the number of elements is finite (q elements) then the field $\mathbb{F}_q$ is referred to as $GF(q)$ (Galois Field). The GF(2) is of more importance to our thesis since it is used to construct many block codes including the BCH codes. The tables for the binary operations over the $< \mathbb{F}_2, +, . >$ are given below

It's clear from the above table that the addition represents the truth table of a XOR gate, and the multiplication represents the truth table of an AND gate. All the polynomials of the block codes have coefficients either in $GF(2)$ or with the extended field $GF(2^m)$. In the case of BCH code, the polynomials of interest have the coefficients in

+	0	1
0	0	1
1	1	0

Table 2.1: *GF Addition*

.	0	1
0	0	0
1	0	1

Table 2.2: *GF Multiplication*

$GF(2)$, but the roots of this equation could be with the extended field $GF(2^m)$. For more details on the mathematical aspects to the error correction code refer to [12].

2.1.2 BCH code

BCH codes are cyclic block codes encoded by the generator polynomial $g(x)$ over the $GF(2)$. The roots of this polynomial equation reside in the extended field, also known as splitting field, $GF(2^m)$. Let $\phi_i(x)$ be the minimal polynomial of an arbitrary element β^i, then the generator polynomial for BCH code with t error correction capability is given by

$$g(x) = LCM(\phi_1(x), \phi_2(x), ..., \phi_{2t}(x)) \tag{2.1}$$

Narrow sense BCH codes use primitive element α^i for the minimal polynomial with i starting from 1. For simplicity, the narrow sense BCH code decoder is used in this thesis, which could be easily extended for other general BCH codes[12]. The parity bits is then generated using the equation

$$p(x) = m(x)\, mod\, g(x) \tag{2.2}$$

where $p(x)$ is the parity bits. The parity bits is then concatenated to the message polynomial $m(x)$ to form the fully encoded data bits which is expressed as

$$c(x) = m(x).x^{deg(g(x))} + m(x)\, mod\, g(x) \tag{2.3}$$

The Hamming distance of this BCH code is $2t+1$ [12], and the code has the capability to correct t bit errors. Once the codeword $c(x)$ is generated, they are stored in the flash memory. When the data is read from the flash memory, it is possible to have an error in the data stream which is expressed as

$$r(x) = c(x) + e(x) \tag{2.4}$$

The BCH decoder's responsibility is to identify the error vector $e(x)$ and retrieve the original message which is given as

$$r(x) + e(x) = c(x) + e(x) + e(x) = c(x) \tag{2.5}$$

From Eq.2.5, the $e(x)$ can be used to segregate if the error occurred in the $m(x)$ or $p(x)$.

2.1.3 BCH Encoder

BCH encoder creates the valid code word $c(x)$ from the message $m(x)$. The generator polynomial $g(x)$ given in Eq.2.1 can be rewritten as

$$g(x) = LCM(\phi_1(x), \phi_3(x), \phi_{2t-1}(x)) \tag{2.6}$$

This is because of the relationship between the minimal polynomial $\phi_i(x)$ and the conjugacy class of the element β^i. In most scenarios, the degree of the minimal polynomial is equal to the extension field m. Therefore, the length of the parity bits $\leq t * m$ where t is the number of bit errors to be corrected.

2.1.4 BCH Decoder

There are two primary methods of decoding the BCH codes: they are hard decision [31, 32] and soft decision algorithms [33, 34, 35]. The hard decision algorithm is the preferred algorithm in the VLSI implementation because of the open loop structure it offers. The soft decision algorithm has a feedback structure and has more coding gain than the hard decision algorithm. In our research, we use a hard decision algorithm since we propose to use a hybrid approach for the decoder, i.e., we use VLSI implementation and GPUs for the decoders. There are three modules within the BCH decoders: syndrome generation, key-equation solver, and error locator.

2.1.4.1 Syndrome Generation

Syndrome generator is the first step of the BCH decoding process. The syndromes S_i of the received vector $r(x)$ is given as $S_i = r(\alpha^i)$. In other words, the syndrome generator checks if the received code vector $r(x) = r_{n-1}x^{n-1} + ... + r_1 + r_0$ has the roots as α^1,α^2,..,α^{2t}. If so, then there are no errors in the received code vector. In the case of an error, the key-equation solver and the error locator steps are executed. For t bit error correction on a narrow sense BCH code, it is sufficient to find t syndromes, because the elements of a conjugacy class have the same minimal polynomial $\phi_i(x)$.

2.1.4.2 Key-equation solver

An error locator polynomial $\Lambda(x)$, which has dependency on the error location, gives the hint of the error location and it is given by the equation

$$\Lambda(x) = \sum_{i=0}^{t} \Lambda_i x^i = (1 - X_1 x)(1 - X_2 x)...(1 - X_t x) \tag{2.7}$$

where X_i represents the error location of the vector $r(x)$. The key equation

$$S(x).\Lambda(x) = \Omega(x)\, mod\, x^{2t} \tag{2.8}$$

shows the relationship between the error locator polynomial and the error evaluator polynomial; moreover, Newton identities [12] shows the relation between the error locator polynomial $\Lambda(x)$ and the syndromes S_i. There are many algorithms like Berlekamp-Massey(BM), Peterson, and others proposed to solve the key equation [12, 36], but inversion-less BM(iBM) algorithm is predominantly used in high throughput architectures [9, 31].

2.1.4.3 Error Locator

For the error locator, the equation $\Lambda(x)$ is evaluated for the element $(\alpha^{pos})^{-1}$ as a root. If $(\alpha^{pos})^{-1}$ is a root then the bit position *pos* has the error, and the error magnitude is "1" since the polynomial coefficients reside in $GF(2)$. The Chien search(CS) algorithm is used to locate the error position from the error locator polynomial equation[12].

2.2 NAND Flash Error Analysis

There are several mechanisms that can lead to bit errors in Flash memories, including program disturbance from tunneling and hot-electron injection, quantum- level noise effects, erratic tunneling, SLC- related data retention and read disturbance, and detrapping-induced retention. Also, shifts in memory-cell device characteristics such as the threshold voltage (VT) can also lead to different bit errors in NAND flash. The fraction of bits that contain incorrect data before applying ECC is called the raw bit error rate (RBER). The error rate after applying ECC is called the uncorrectable bit

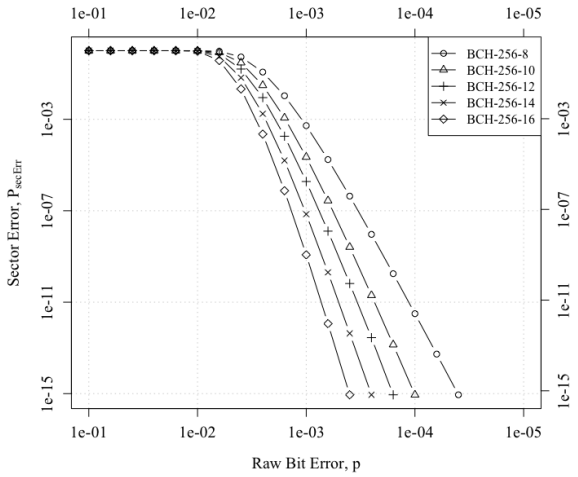
error rate (UBER). HDD manufacturers and NAND-based storage manufacturers often quote UBER values on their datasheets, typically 10^{-13} to 10^{-16}. UBER is a useful reliability metric for mass-storage devices such as HDDs because a bit error that damages one file out of many is not equivalent to a functional failure that destroys the drive. UBER is used to specify the data-corruption rate, whereas metrics such as mean time between failure (MTBF) specify the functional failure rate. Fig. 2.1 plots the UBER for different sector sizes, i.e. for 256-bytes, 512-bytes, and 1024-bytes sector sizes, against RBER.

2.3 Previous Work

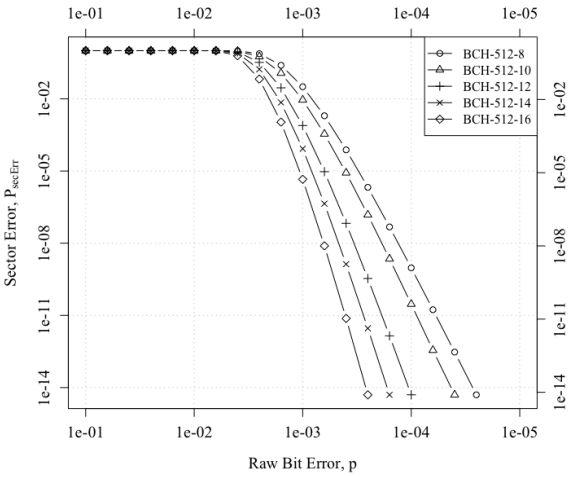
In this section, we have reviewed the previous work related to our thesis as CPU, VLSI and GPU implementation, and they are listed as below:

2.3.1 CPU implementation

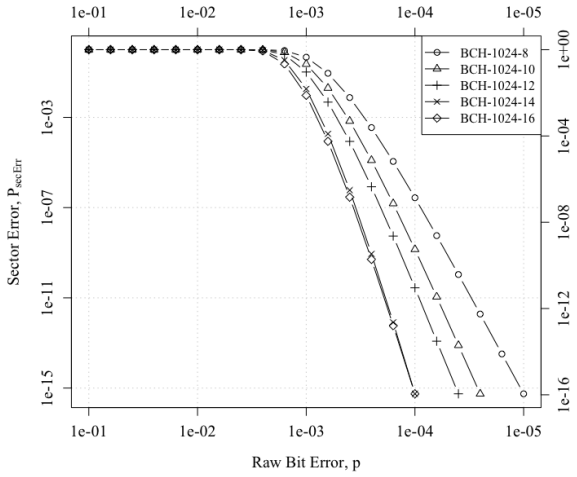
CPU implementation of ECC is the most effective solution used for embedded systems [15, 16, 17]. J.Cho et al. [37] proposed a software-based approach, and he proposed a minimized LUT approach to have high throughput, but the performance depends on the CPU used. This method has a dependency on the load of the CPU while the decoder is active, so this method is not viable to correct errors for all embedded systems. H. Lee et al. proposed a RAM assisted ECC scheme to improve the reliability of the flash memory [17]. Temporary storage was used to store log blocks for the ECC of the data blocks, which had proven to improve the performance by looking up in the temporary ECC memory. Yu-peng Hu et al. [16] proposed an elastic error correction technique, as shown in Fig.2.2, to modify the ECC parity bits while writing back the



(a) UBER for 256-byte sector



(b) UBER for 512-byte sector



(c) UBER for 1024-byte sector

Fig. 2.1: UBER for different sector sizes

data. In this, the number of errors of a page is tracked, and the parity bits are increased when the number of error detected is close to the limits of the code. The generator polynomial $g(x)$ of the block as shown in Fig.2.2 changes dynamically thus yielding the elasticity, and the finite field dimension m changes when the block sizes change. Hence, the elastic ECC improves the reliability of the hot data stored in a block exposed to variable random errors. Later, C. Kim et al. [15] proposed a robust error correction scheme to extend the lifetime of the flash memories. A gradual error correction code

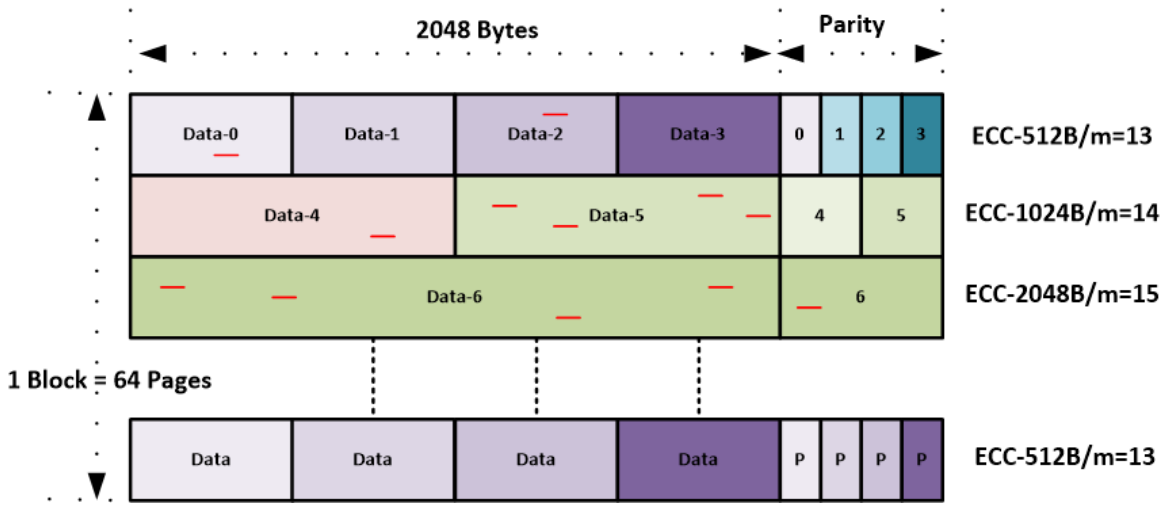


Fig. 2.2: Elastic ECC for NAND flash memory

(G-ECC) was proposed to dynamically change the ECC scheme to fit the need of the current data block errors. Instead of increasing the parity bits, with the same ECC scheme, as proposed in [16], changing the ECC scheme was proposed in [15]. This method, however, has the limitation of the ECC decoder techniques that are available. Also, there is a significant penalty on the performance of the decoders when an error is detected on a particular page; especially, when an LDPC decoder is employed for strong ECC. J. Cho et al. [37] proposed a highly efficient CPU based BCH decoders using the hard decision algorithm.

2.3.2 VLSI implementation

2.3.2.1 Encoders

BCH encoders have high fanout issues for long polynomials $g(x)$. This was addressed by X. Zhang et al. in [38] by breaking the $g(x)$ into smaller domains which is achieved by converting the $g(x)$ to $G(z)$, i.e., from x to z domain, and implementing the generator polynomial in the z domain. The relationship between the $G(z)$ and the

minimal polynomials $M_i(z)$ is given below

$$\frac{1}{G(z)} = \frac{1}{M_1(z)} * \frac{1}{M_3(z)} * .. * \frac{1}{M_{2t-1}(z)} \tag{2.9}$$

H. Chen [39] proposed the long BCH encoder using the Chinese Remainder Theorem (CRT) method. This method requires additional hardware because of the CRT method to convert $M_i(x)$ to $M_i''(x)$. Then, a multimode implementation of the BCH encoder to support multiple bit errors was proposed by H.Tang et al.[40] and H.Yoo et al. [41]. In Fig.2.3, each block computes the remainder divided by minimal polynomial, $\frac{1}{M_i(z)}$, and the outputs are cascaded to the adjacent blocks, and it also eliminates the combinational loop in the feedback path of $y(z)$. For a multimode BCH encoder with tsel error, the output could be tapped at the appropriate cascaded block. An enhancement to [41] was proposed in [40], which introduced a hybrid approach with a multiplexer. This method adds a re-encoder AND gate in the path of the cascade, which cuts the feedback path depending on the tsel, and the multiplexer allows the point at which the residual can be tapped.

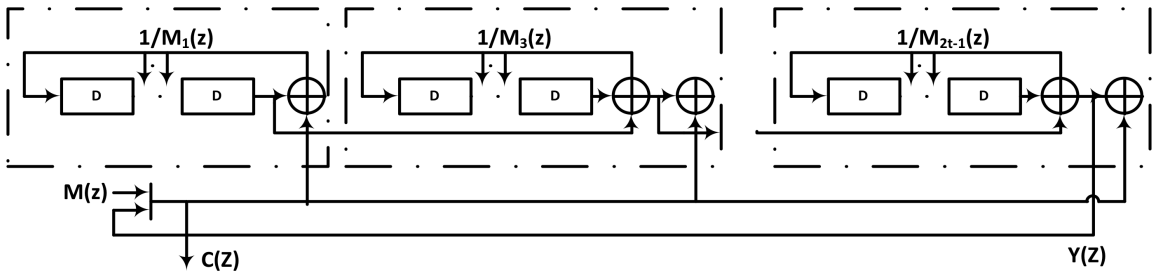


Fig. 2.3: Hardware implementation of long BCH encoder

2.3.2.2 Decoders

Various hardware implementations of the BCH, RS, and LDPC encoders/decoders are discussed in [36]. A high-throughput hardware implementation of BCH decoder was proposed by Lee et al.[31]. This paper proposed a hard decision BCH decoder with Berleykamp-Massey Algorithm (BMA) used for the key-equation solver and Chien search algorithm for the error correction stage. An alternative to hard decision algorithm there is soft decision-based BCH decoders[42, 43]. Zhang et al. proposed an interpolation based BCH decoders[34]. The soft-decision method provides a high SNR on the decoder, but the hardware complexity of such a system grows for high bit error rates. In this thesis, we focus on the hard decision algorithm since the area consumed by these methods are less than the soft decision algorithms. For BCH codes with 3-bit errors or less, X.Zhang proposed multiple optimization techniques for the Key-equation solver modules [44, 45]. The optimization is achieved by using the Newton identities that relate the syndrome S_i to the key-equation $\Lambda(x)$. For bit errors more than 3, we need an iterative approach to solve the key-equation, and the iBMA algorithm is used to solve these equations in hardware.

A full hardware implementation of the BCH decoder using iBMA was implemented and studied by J.Ernest [46]. Later, high-performance hardware implementations of BCH decoders were proposed in [9, 32, 44, 45]. Wei et al. [9] proposed the same BCH decoder with low power and high throughput architecture. Yoo et al. [47] proposed a low power architecture for Chien search. Park et al. provided a novel KES architecture for an area efficient decoders [32], but this method lacks high throughput because of the number of iterations required for decoding is higher than t clock cycles for an area-efficient architecture. A flexible bit error supporting BCH decoder was proposed by Chen et al.[48], but the arithmetic is fixed to specific block size ($GF(2^m)$). Later, Yoo et al. proposed a flexible BCH encoder and decoder for high-speed SSD systems [49].

The above-specified architectures are limited by the block size which is the finite field dimension $GF(2^m)$ where the arithmetic is performed. Later, Park et al. [19] proposed a method to support multiple $GF(2^m)$, but this method requires more area to support different $GF(2^m)$ because of the shared area between GF arithmetic. We intent to propose a pragmatic approach to the same problem using a hybrid approach in this thesis.

2.3.3 GPU implementation

Recent development with GPUs has attracted many researchers to use GPUs for ECC applications. To harness the parallelism offered by the GPUs, we need special software like CUDA [14] to program the kernel routines. There are many GPU solutions for BCH[50], Reed Solomon[23, 51], and LDPC codes[52]. The author et al. [28] have proposed a GPU based RS decoder using a frequency domain approach, and it proved to be more suitable for GPU implementation because of the parallel algorithm offered by the FFT method. The FFT method is not suitable for BCH codes because the error magnitude vector in the BCH code is '1'. X. Qi et al. [50] proposed a GPU implementation for BCH decoders, but there were some inefficiencies identified for this method. The data to be decoded is split into multiple bits and threads which had a significant impact on the memory consumed, and it had a significant impact on the warp scheduler for different threads. Also, the decoder used the Euclidean method which consumes more iteration cycles on the threads for decoding the error vector. This method has proven to have more than 125% increase in performance when compared with the CPU implementation of the BCH decoders. The author et al. [23] proposed an alternate method using inversion less Berleykamp-Massey algorithm (iBMA) and an efficient method for handling the number of threads. In addition to the GPU implementation, the organization of the parity bits within the NAND flash memory area provides high

throughput efficiency, and it was addressed by Lee et al. [17].

Chapter 3

VLSI Implementation

A general block diagram of the hardware BCH decoder is depicted in Fig. 3.1. The flash memory interface is a physical interface to the flash memory device, and the host interface is an interconnect to the SOC or a CPU system. The syndrome generator is responsible for generating the syndrome vector S_i. The iBMA key-equation solver forms the equation $\Lambda(x)$, and the Chien search module creates the error vector $e(x)$ which can be used to compensate and retrieve the message $m(x)$ from the received vector $r(x)$. The data read from the flash is stored in a temporary storage FIFO. Once the $\Lambda(x)$ is formed the data from the storage FIFO is read and the Chien search algorithm works in tandem to generate the $e(x)$.

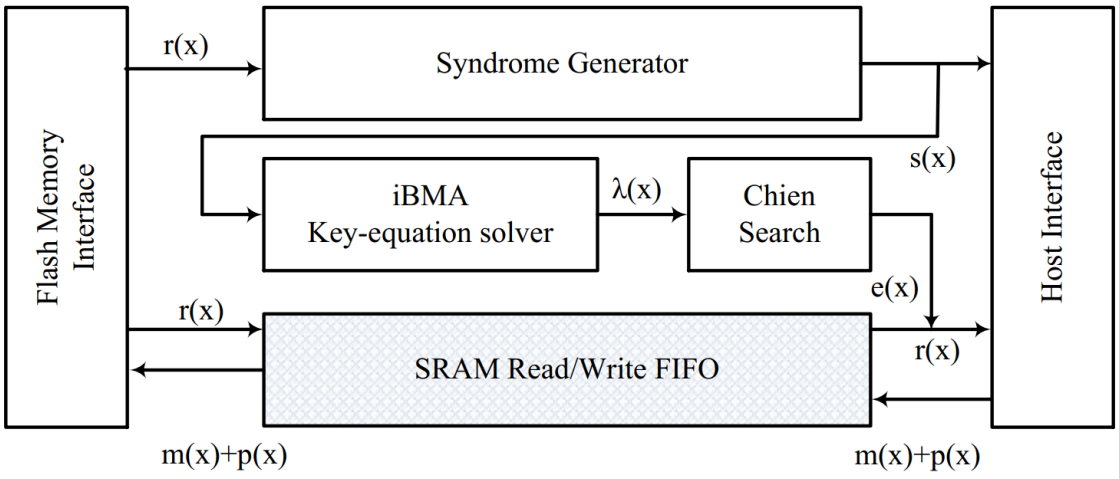


Fig. 3.1: Hardware Decoder

3.1 Encoder

Let us consider a BCH(n,k,t) code, where n, k, and t denote the code length, the message length, and the maximum number of correctable error bits, respectively. Generally, systematic encoding of the BCH is formulated in Eq.3.1.

$$c(x) = m(x).x^{deg(g(x))} + m(x)\, mod\, g(x) \tag{3.1}$$

where $m(x)\, mod\, g(x)$ denotes the remainder polynomial obtained by dividing $m(x)$ by $g(x)$, $m(x)$ is the message polynomial and $c(x)$ is the final code vector. The standard Linear Feedback Shift Register(LFSR) architecture computes the remainder polynomial by dividing the message polynomial by the polynomial $g(x)$ as shown in Fig. 3.2.

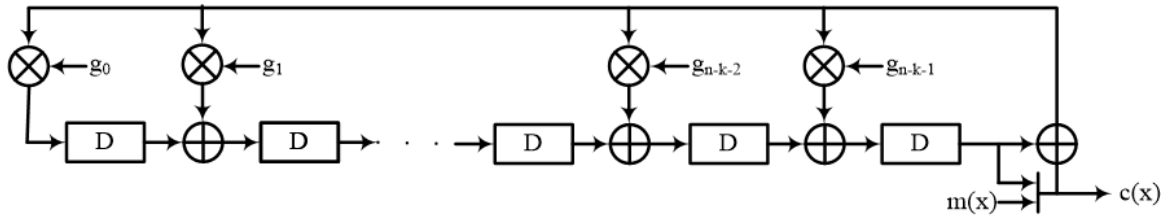


Fig. 3.2: BCH encoder in serial fashion

The polynomial $g(x)$ is the LCM of the minimal polynomial of the elements β^i and is given by the equation

$$g(x) = LCM(\phi_1(x), \phi_3(x), .., \phi_{2t-1}(x)) \tag{3.2}$$

The LFSR shown in Fig. 3.2 can be easily unfolded to the data width of the flash memory interface [36]. This allows more parallel processing of data stream instead of the serial data stream.

Fig. 3.3 depicts the timing diagram for the encoder with a data bus width of 32

bits. The calc_parity bit is asserted high throughout the encoding process. When this signal is de-asserted, the encoder resets itself. The signal data_valid qualifies the validity of the data on the data bus. The residue bits are generated for every clock cycle where the data bits are valid. When the last data is fed to the encoder, the final residue bits are presented in the next clock cycle which would be used to form the code vector $c(x)$.

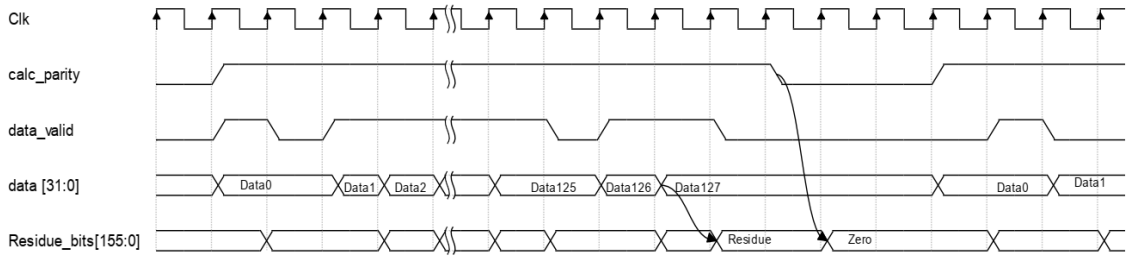


Fig. 3.3: Timing diagram for encoder

3.2 Hardware Decoder

The decoder block is responsible to decode the received vector and correct the errors if it exists. The decoder block is a complex module when compared with the encoder module. The decoder block consists of three significant submodules: the syndrome generator, the key equation solver (KES), and the error corrector block. The decoder module can be designed for different goals, like low gate count, low latency, or low area. In this section, the proposed design has low latency (high throughput) as its goal. That is the maximum clock period after which the errors can be corrected. Also, the input data stream and the data mask signal are assumed to be a parallel bus for efficiency. Fig. 3.4 depicts the block diagram for the decoder module

The syndrome generator module generates the syndrome on the received vector for each root, $\{\alpha, \alpha^2, .., \alpha^{2t}\}$, of the generator polynomial g(x). If the set, $\{\alpha, \alpha^2, .., \alpha^{2t}\}$, is

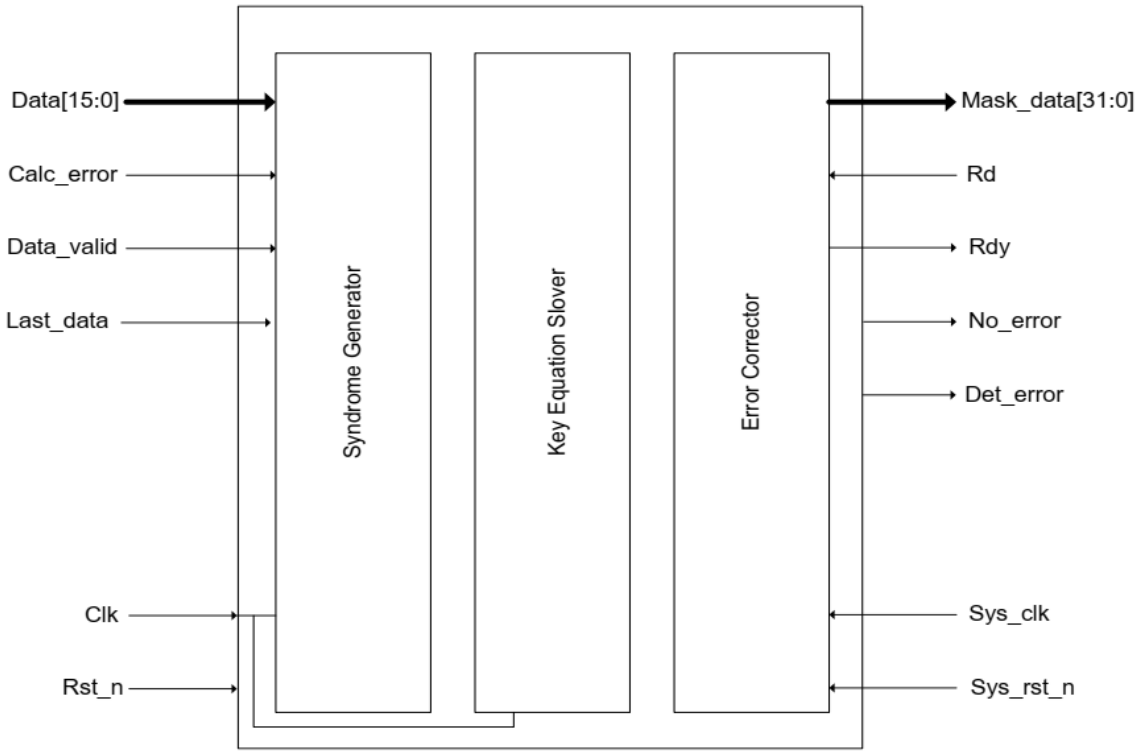


Fig. 3.4: VLSI BCH Decoder

a root of the received vector, then there exists no error. However, if there are errors induced in the received vector, then the syndromes are passed to the key equation solver module for further analysis. An optional pipeline stage can be implemented in order to achieve efficiency during back to back data transfers. However, if the system can tolerate a delay of t clock cycles, the pipeline stage can be excluded.

The KES module is triggered only when all the calculated syndromes are not equal to zero. The error locator polynomial equation is derived from the syndromes computed. This module uses the inversion-less Berlekamp algorithm to derive error locator polynomial equation. However, there are other algorithms available for this iterative process.

The error corrector module uses the parallel Chien search algorithm to find the error locations. The number of bits to be corrected in parallel depends on the system

bus width. Also, this module, error corrector, is clocked by the system clock in order to be efficient.

3.2.1 Syndrome Generator

The syndrome generator is the first step of the hardware decoder. The element α^i is evaluated on the received code vector for $2t$ positions. The S_i is expressed by the following equation

$$S_i = r(\alpha^i) = \sum_{j=0}^{n-1} r_j.\alpha^j \tag{3.3}$$

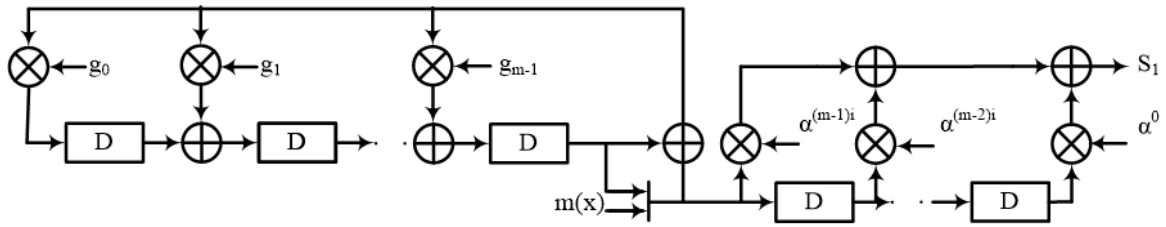


Fig. 3.5: Syndrome Generator

An alternate method to generate the syndrome was proposed in [36]. Here the $res_i(x)$ is introduced, which is given by

$$res_i(x) = r(x)\, mod\, \phi_i(x) \tag{3.4}$$

Now the final syndrome S_i can be computed using the formula

$$S_i = res_i(\alpha^i) = \sum_{j=0}^{m-1} res_j.\alpha^j \tag{3.5}$$

Fig. 3.5 represents the structure to generate the syndrome S_i using Eq.3.5. Each syndrome S_i is independent of each other, so each of the syndromes can be calculated in tandem. Also, it is sufficient to find t syndromes. Fig. 3.6 depicts a systolic array

of syndrome generator unit for t syndromes. The syndromes calculated by this unit is then passed to the KES module for further progress in the error correction.

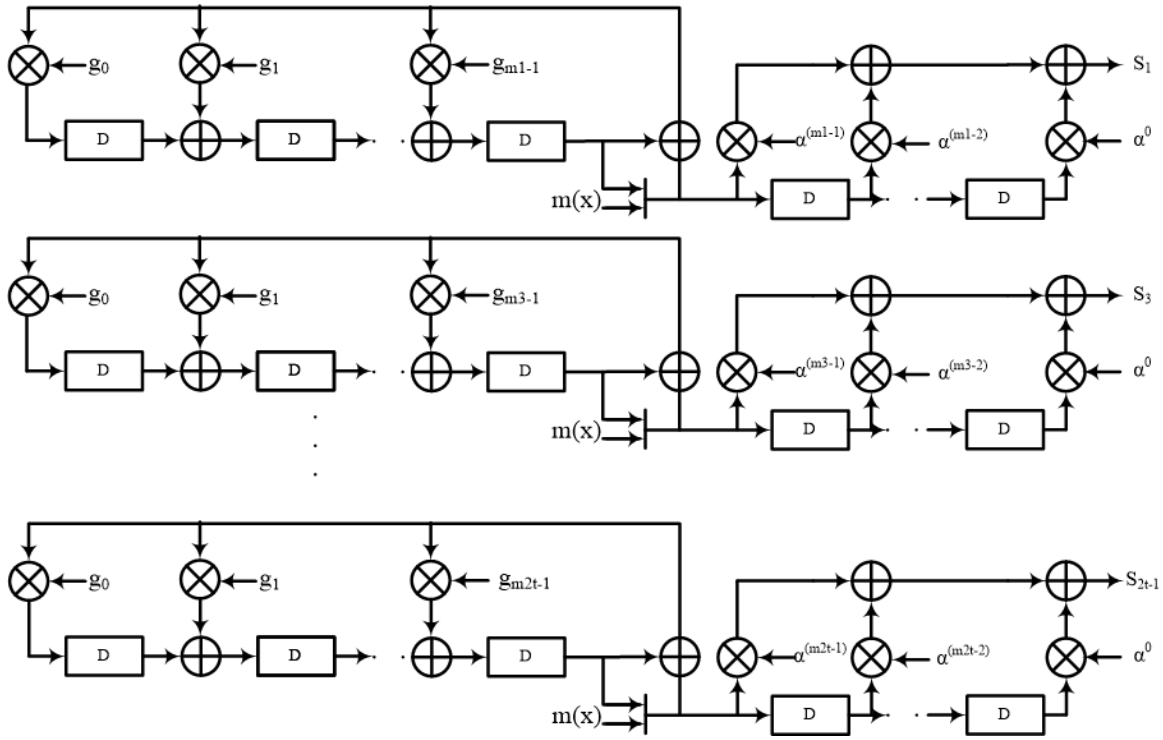


Fig. 3.6: Syndrome Generator systolic array

3.2.2 Key-Equation Solver

The key equation solver generates the error locator polynomial $\sigma(x)$, which has the inverse of the error positions as their roots. This equation is derived from the syndrome that is generated from the received vector. The KES is the arduous design in the decoder module. The error locator polynomial is given by the equation

$$\sigma(x) = \sigma_t.x^r + \sigma_{t-1}.x^{t-1} + .. + \sigma_0 \tag{3.6}$$

The relationship between the error locator polynomial and the syndromes is given by the following equation

$$\sum_{j=0}^{t} S_{t+i-j}\sigma_j = 0 \tag{3.7}$$

where $i = 1, .., t$.

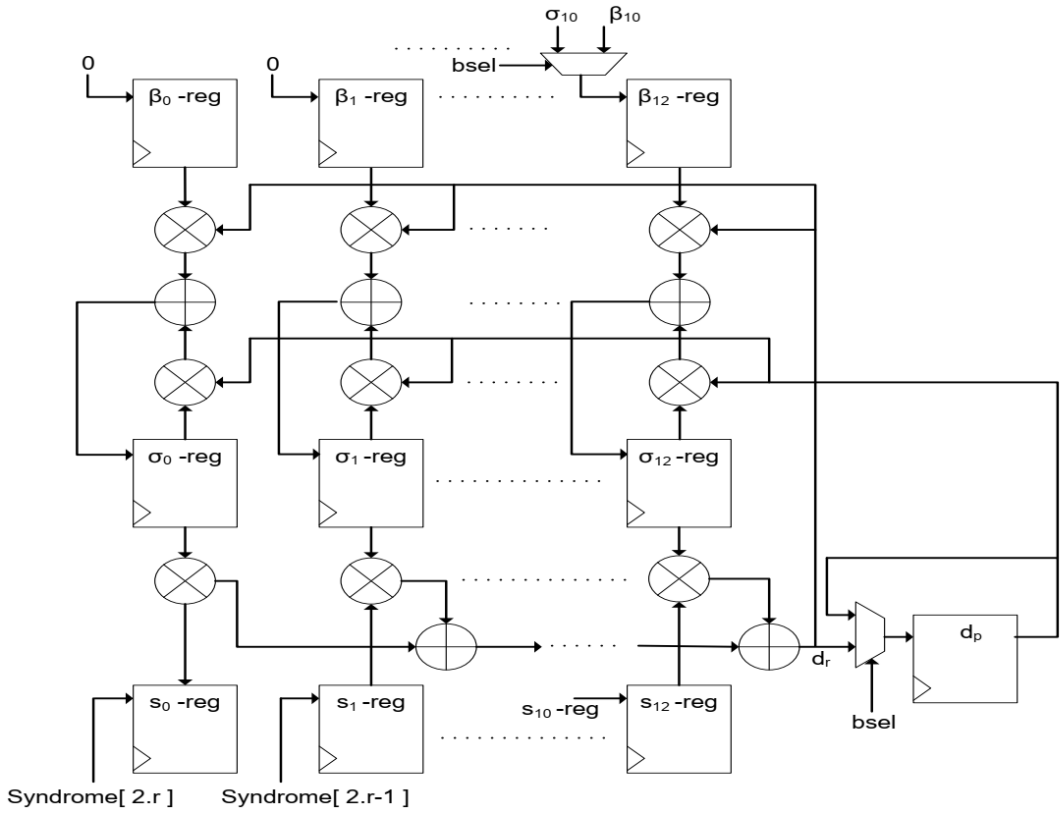


Fig. 3.7: iBMA Key Equation Solver

This relationship between the syndrome and the error locator polynomial is known as Newton identity. The error locator polynomial has the information about the error positions. In order to find the error location, the coefficients of the error locator polynomial should be derived. There are different algorithms to find the coefficients of $\sigma(x)$, some of them are Peterson-Gorenstein-Zieler algorithm, Euclid's algorithm, Berlekamp-Massey Algorithm, and inversion-less Berlekamp-Massey algorithm (iBMA).

In this design, the iBMA algorithm is implemented in order to have low latency during this iterative method. It is assumed that the reader is familiar with the Berlekamp-Massey algorithm. For more information, please refer to [12, 36].

The iBMA provides an iterative algorithm to find the error locator polynomials coefficients. Fig. 3.7 shows the block diagram for the key equation solver module for a 512-byte and 12-bit error-correction. Alg. 1 represents the iterative iBMA algorithm for hardware implementation.

Algorithm 1 Key-equation Solver

procedure KES(σ, S)
  $\sigma^{(0)} \leftarrow 1 + S_1 x$
  if $S_1 = 0$ **then**
    $d_p \leftarrow 1; \beta^{(1)} \leftarrow x^3; l_1 \leftarrow 0$
  else
    $d_p \leftarrow S_1; \beta^{(1)} \leftarrow x^2; l_1 \leftarrow 1$
  end if
  for $r \leftarrow 1, t-1$ **do**
    $d_r \leftarrow \sum_{i=1}^{t} \Lambda_i^{(r)} S_{2r-i+1}$
    $\sigma^{(r)} \leftarrow d_p \sigma^{(r-1)} + d_r \beta^{(r)}$
    if $d_r = 0 \, or \, r < l_r$ **then**
      $\beta^{(r+1)} \leftarrow x^2 \beta^{(r)}; l_{r+1} \leftarrow l_r; d_p \leftarrow d_p$
    else
      $\beta^{(r+1)} \leftarrow x^2 \sigma^{(r)}; l_{r+1} \leftarrow l_r + 1; d_p \leftarrow d_r$
    end if
  end for
end procedure

At the end of t-1 iterations, the register $\sigma(x)$ will have the coefficients for the error polynomial equation. These values are then passed to the error corrector module to correct errors. Since the magnitude of the error vector is 1 (binary), there is no need to calculate the error magnitude.

3.2.3 Error Corrector

The error corrector module is responsible to correct the errors in parallel (up to the bus width). This module uses the Chien search algorithm to correct the errors. Since the errors have to be evaluated for parallel bit stream, the coefficients are multiplied with the appropriate field element and summed to yield the error value. If the resultant of the sum is zero, the bit position is reported for error. The multiplied values for the higher order bit position is stored in the register for the next iteration. Hence, this module is iterated until the last parallel data is read from the FIFO. For more information on p-parallel Chien search algorithm, refer to [12, 36]. The block diagram below (Fig. 3.8) shows the implementation for parallel Chien search algorithm for 512 bytes and 12-bit error-correction.

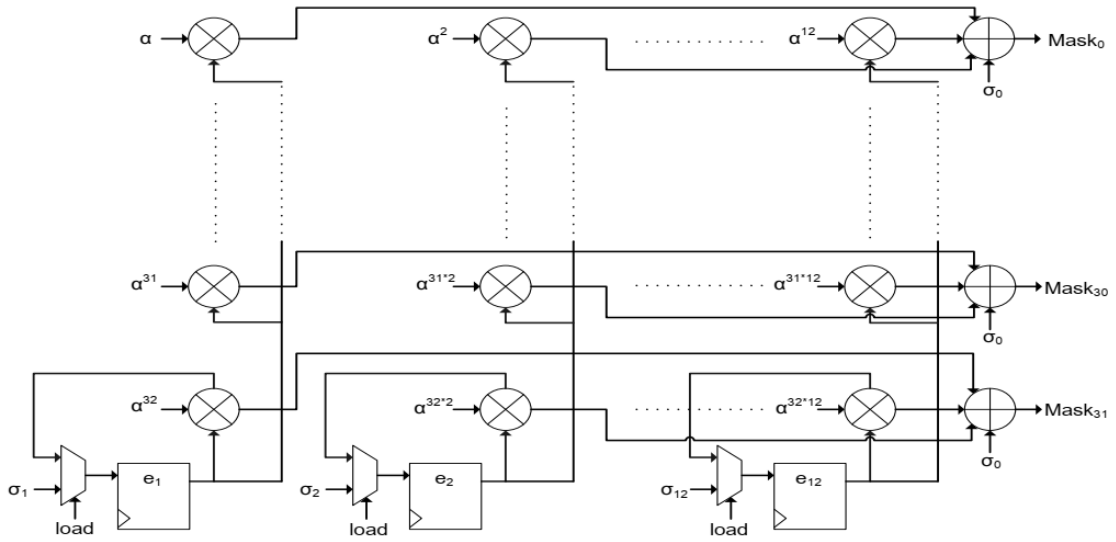


Fig. 3.8: Error corrector

Since the error magnitude is only 1, the output of this block is treated as a mask bit to the data read from the FIFO. Also, the register content is updated for every read strobe; this makes it synchronous to the system bus.

3.2.4 Timing Diagram

The diagram below (Fig. 3.9) represents the timing for an error scenario, where there is no pipe line stage added for the syndrome generator. The signal, calc_error, is asserted until the signal rdy is asserted. The signals synchronous to the sys_clk domain are rdy, rd, and Mask_data. Initially, the signal, det_error, is asserted to indicate that an error is detected; the signal, rdy, is then asserted to indicate the validity of the signal, mask_data. If an additional pipeline stage is added to the syndrome generator, the signal, calc_error, can be de-asserted immediately after the de-assertion of the signal, det_error.

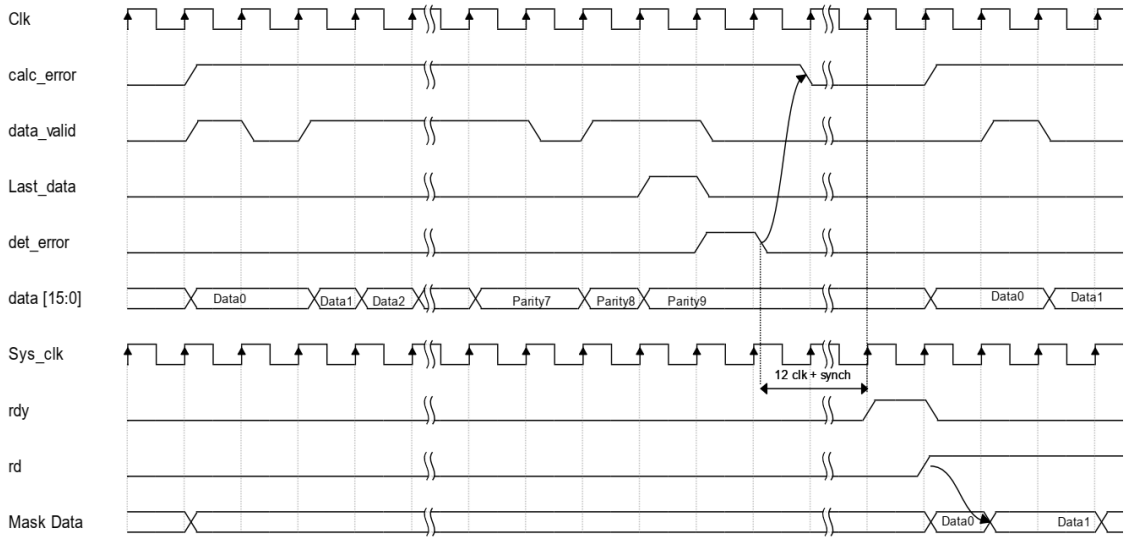


Fig. 3.9: Timing diagram for Error scenario

However, it is the responsibility of the system to empty the erroneous data in the buffer before the last data is provided to the decoder block. Fig. 3.10 shows the timing diagram for a no error scenario. In the case of no error, there is no latency introduced by the system, since there is no need to solve for the key-equation. The KES and the Error corrector are not employed during a non-error scenario, and this is the reason for quicker response time.

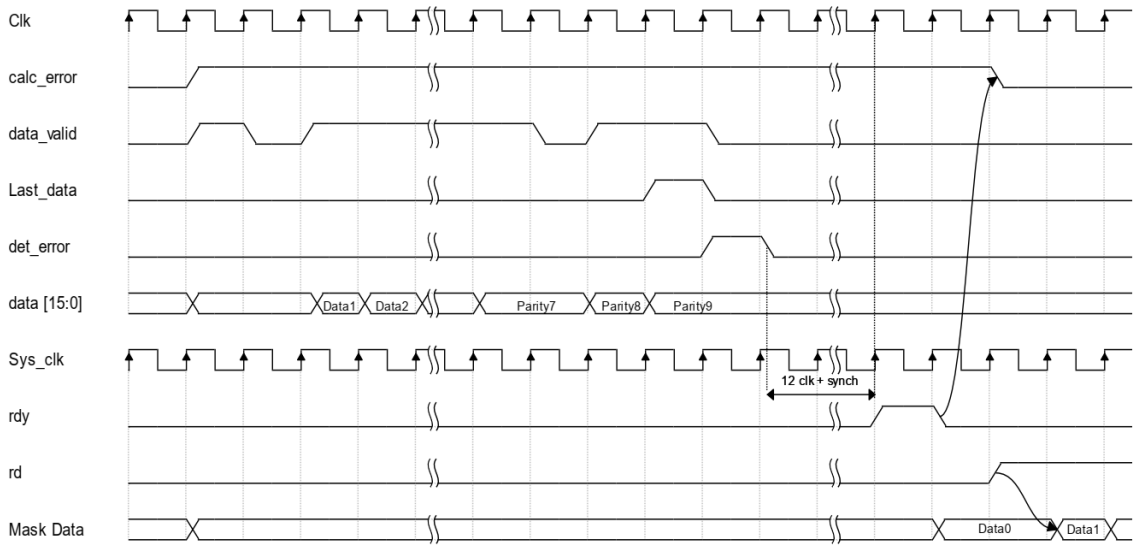


Fig. 3.10: Timing diagram for non-error scenario

3.2.5 Spare Area Management

The Flash memory is organized as set of block memories; a block consists of 64 pages, and each page consists of 2048 bytes and a spare area of 64 bytes per page (Fig. 3.11). The spare area is dedicated to store the parity bits of an ECC code, and it can be used for bad block management. Fig. 3.11 represents the spare area utilization for a 512-byte sector with 8 bit error correction. The parity bits are stored in the same page so that the flash memory page read could fetch both the data and the parity within a single read command. Even though the ECC provides an improvement of reliability for the data stored, the reliability of the same will degrade with time; the program and erase operation are destructive for the memory. Once a block is identified to have more errors or to approach the limits of the ECC, they can be marked as bad blocks, and the contents of this block are moved to another block location.

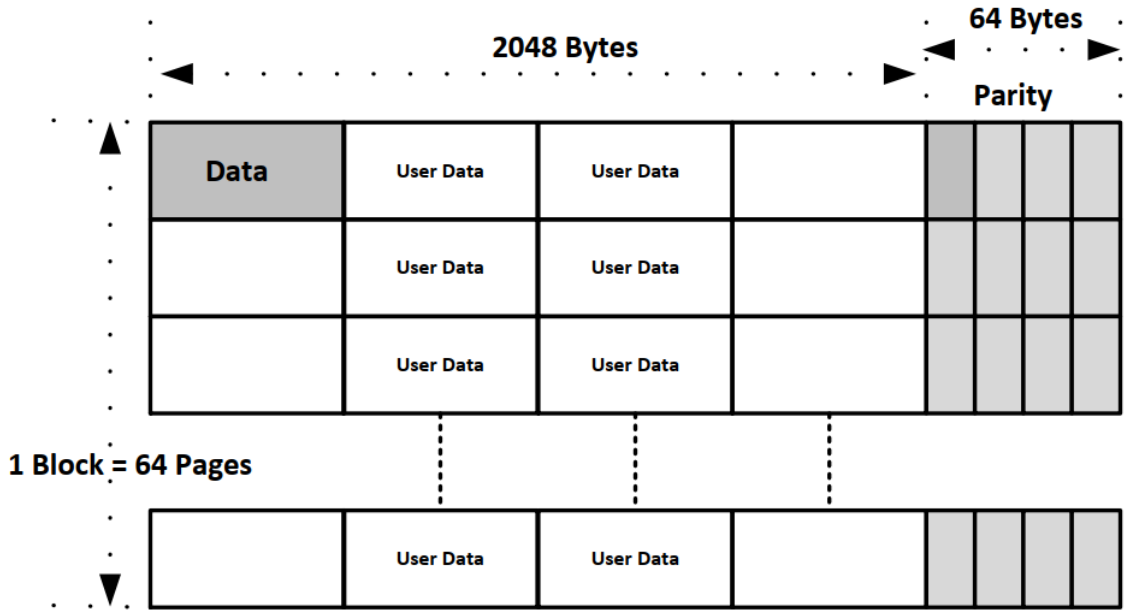


Fig. 3.11: Spare memory area management

3.3 Multimode Encoder and Syndrome Generator

In this thesis, we propose a hardware optimization by combining the syndrome generator and the encoder together. For this, we first introduce the concept of minimal residual polynomial which is given as

$$res_i(x) = m(x) \, mod \, \phi_i(x) \tag{3.8}$$

We propose to use residual of the minimal polynomial $\phi_i(x)$ instead of the residual of $g(x)$. A new BCH encoding scheme for the multimode encoder is shown in Fig. 3.13 which is area efficient with low latency. The conventional syndrome generator [33] has high complexity when it has to be implemented in parallel. An alternate method to generate the syndrome is by using the residue of the received message divided by the minimal polynomial instead of accumulating the multiplier values. Fig. 3.12 shows the

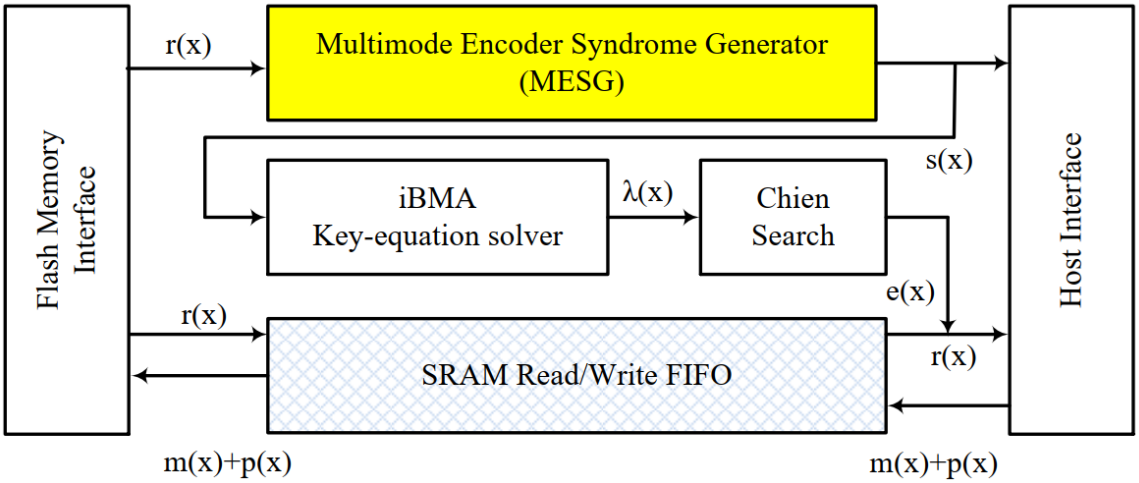


Fig. 3.12: Multimode Encoder and Syndrome Decoder

block diagram of the flash memory interface with the encoder and decoder combined.

3.3.1 Modified Encoder

The proposed encoded code word $c(x)$ is an accumulation of different residual polynomial generated for each minimal polynomial $res_i(x)$. The final code vector is given as

$$c(x) = m(x).x^{deg(g(x))} + \sum_{i=1}^{t} res_{2.i-1}(x).x^{i-1} \tag{3.9}$$

Fig. 3.13 depicts the hardware implementation of the modified encoder. The message $m(x)$ is fed to each independent LFSR implementation of the minimal polynomial. The constants g_{m1-1} represent the coefficients of the $\phi_1(x)$. Similarly, all the other coefficient constants related to $g(x)$ are enumerated. The parity bits of this encoding scheme can be represented as

$$p(x) = \sum_{i=1}^{t} res_{2.i-1}(x).x^{i-1} \tag{3.10}$$

This method reduces the high fanout issues encountered for long BCH codes. However, the parity bits $p(x)$ is not protected for t bit error. So, to overcome this limitation, we

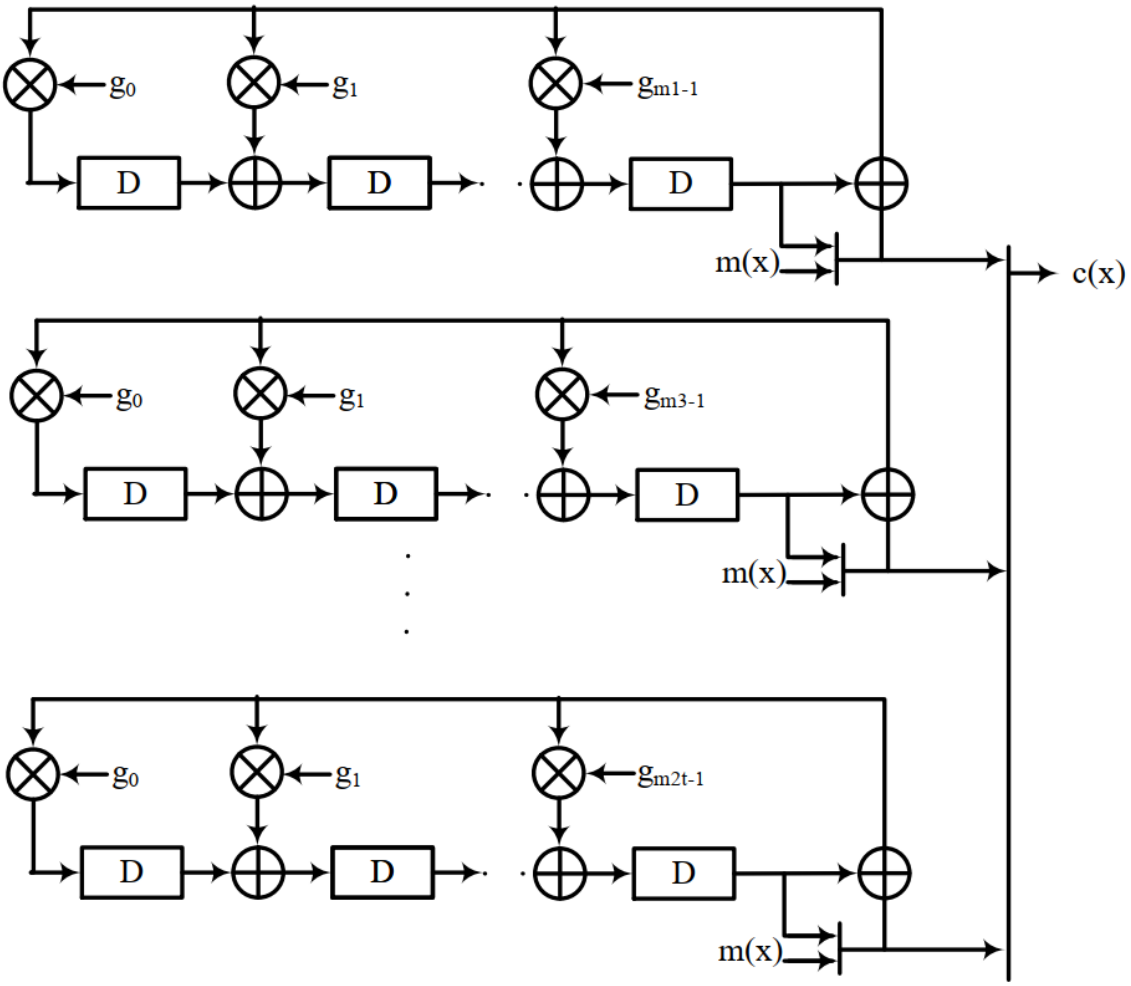


Fig. 3.13: Modified Parity generator

propose to store the parity bits in an SLC memory area.

3.3.2 Residual Checker

The conventional received vector r(x) is split into multiple received vector $r_j(x)$ which is given as

$$r_j(x) = (m(x) + e(x)).x^{deg(g(x))} + res_j(x) \tag{3.11}$$

Note that the message polynomial $m(x)$ and the error vector $e(x)$ are passed to the syndrome checker. Fig. 3.14 represents the checker for the proposed encoding scheme.

If there are no errors then each residual checker $S_{res,j}$ would zero

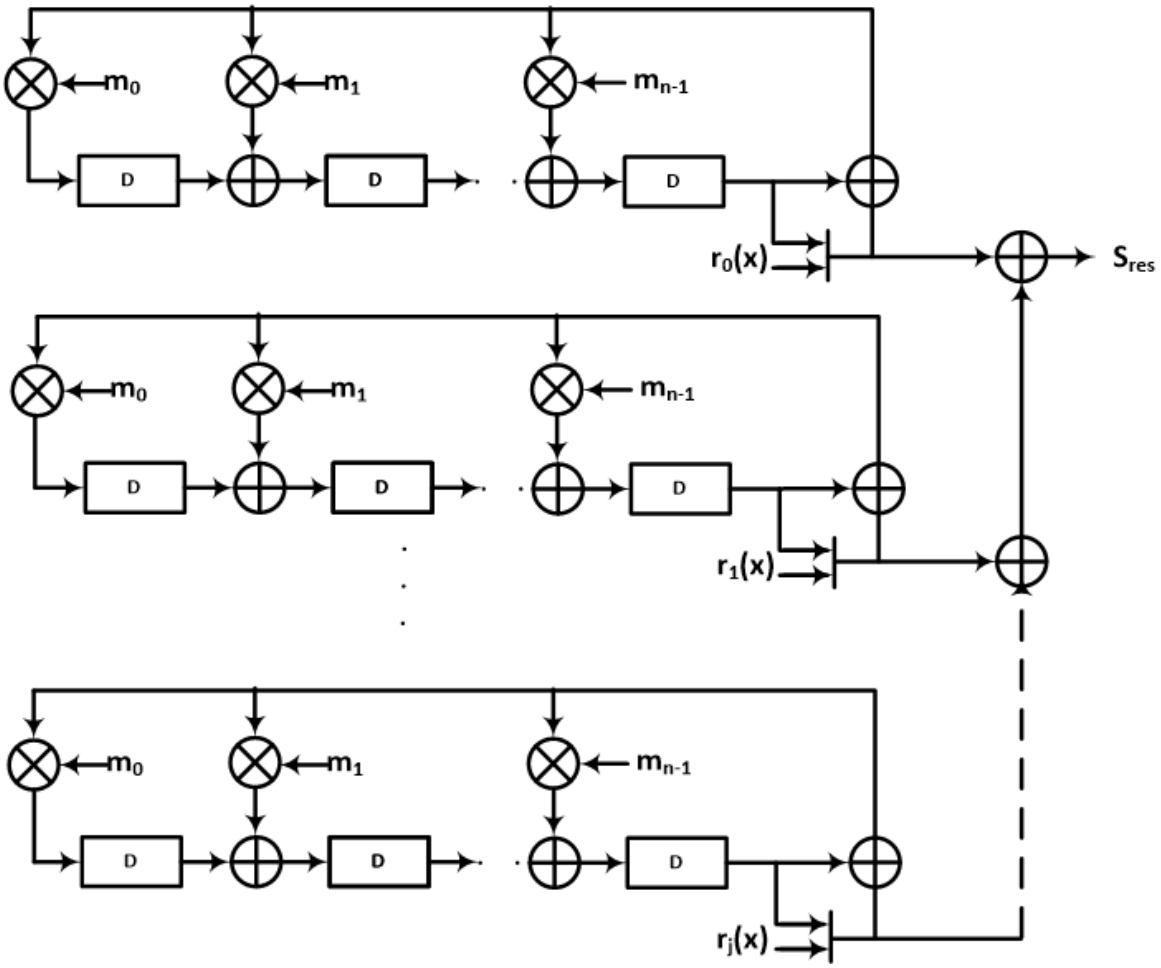


Fig. 3.14: Modified Parity Checker

$$S_{res,j} = r_j(x) \, mod \, \phi_j(x) = 0 \tag{3.12}$$

The final residual check S_{res} is the sum of all the $S_{res,j}$ and would be zero if there are no errors. This method is proven to be very efficient in [20].

3.4 Programmable SRU

Another important advantage of the residual equation (Eq.3.12) is the ability to make it programmable. Fig. 3.15 represents a systolic array of the syndrome residual unit (SRU) which gives the flexibility to program to any desired $\phi_i(x)$. There are additional flops required when compared with Fig. 3.14 in the SRU systolic array. But, the flexibility provided by this approach is of interest for many application targeting different NAND flash devices. The important feature in this implementation is the

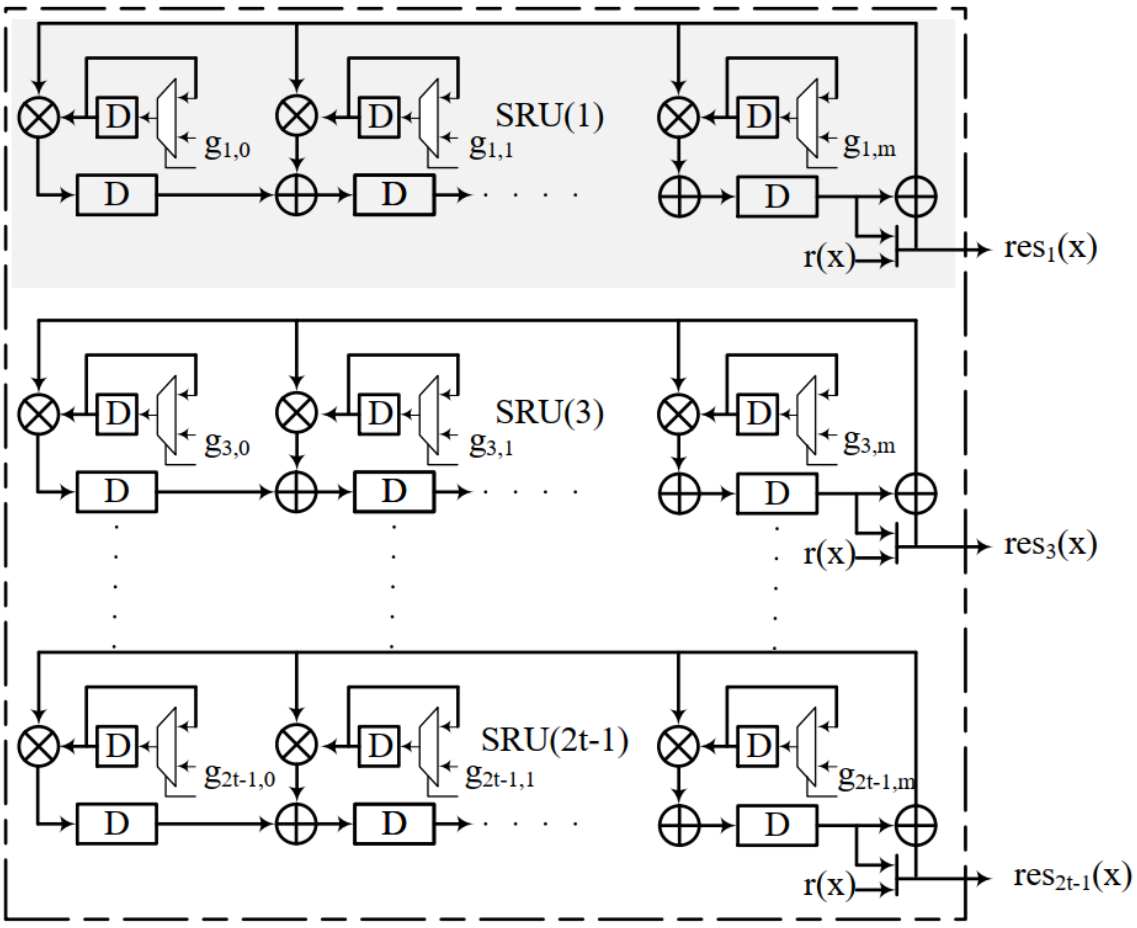


Fig. 3.15: SRU systolic array

configurable number of $res_i(x)$ and the length of the generator polynomial $g_i(x)$. For

example, the length of the $g_i(x)$ can be set to 16, and this is sufficient to support any $\phi_i(x)$ within the GF dimension of m. Similar to the modified parity checker, it is sufficient to generate t sets of $res_i(x)$. The programmable feature will be harnessed in the hybrid approach discussed in Chapter.5.

3.4.1 Multimode Encoder and Syndrome Generator

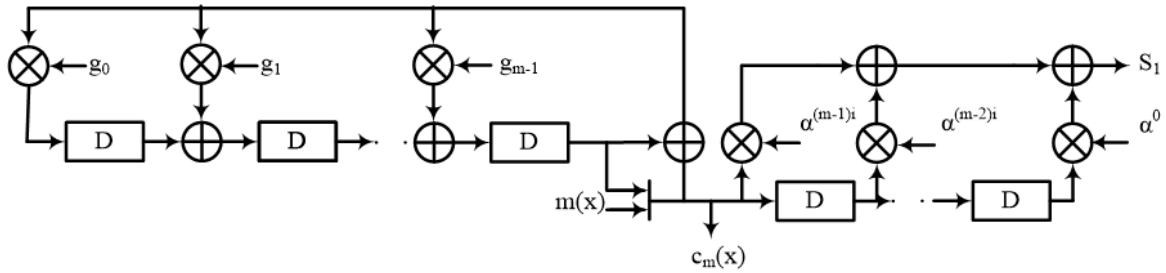


Fig. 3.16: Syndrome generator and Encoder

From Eq. 3.9 and Eq. 3.12, we can see the similar LFSR structure for the encoder and the decoder. Fig. 3.16 represents the serial fashion implementation of the syndrome generator and modified encoder parity bits. The code vector $c_m(x)$ represents the tap point for the encoder, and S_i is the syndrome generated for the received data stream. The same structure can be used in systolic array fashion to have the complete encoder and syndrome generator together. Fig. 3.17 represents the complete syndrome generator and encoder which we refer to as the Multimode Encoder and Syndrome Generator (MESG).

3.5 Experimental Results

In this section, we present the implementation results of the conventional BCH encoder, multimode encoder [41], hybrid multimode encoder [40], and our proposed

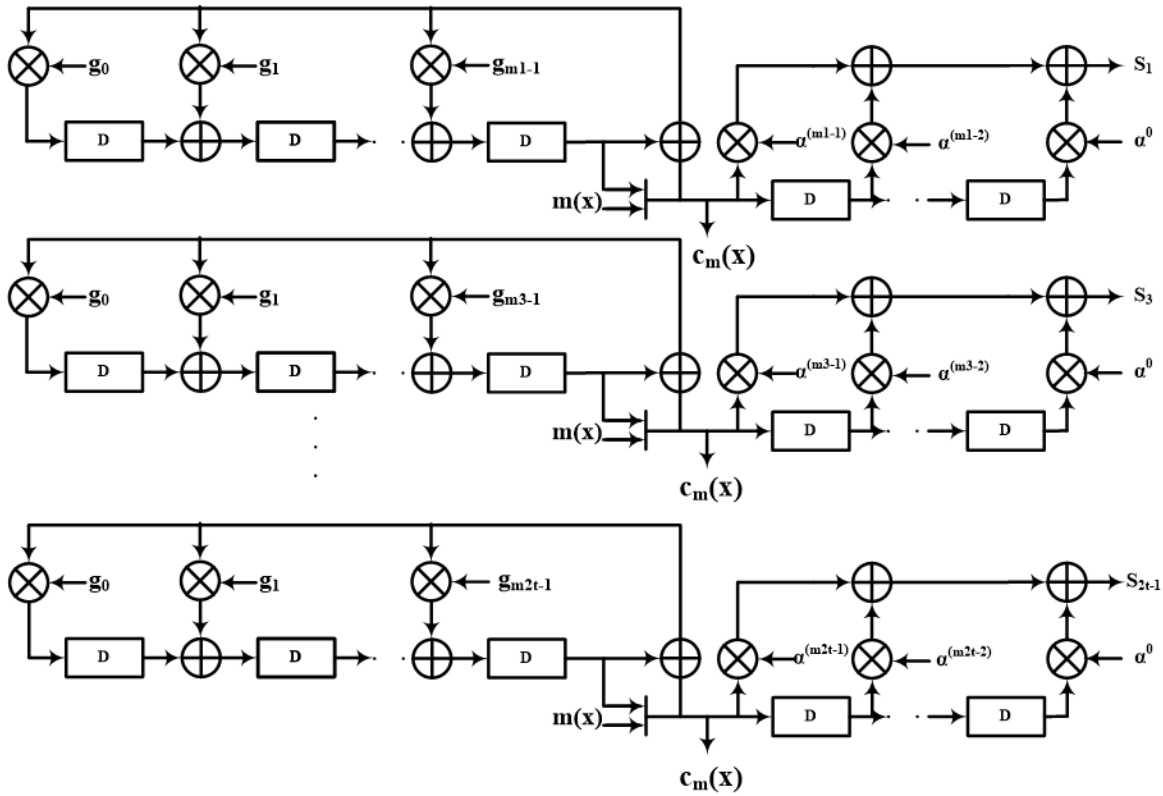


Fig. 3.17: MESG systolic array

encoder scheme [20]. The comparison is based on the number of logical elements used by each scheme. For a generic BCH (n, t, k) different methods are compared in Table 3.1. m in the table represents the $GF(2^m)$ dimension of a 512-byte sector size, i.e. $m = 13$.

Table 3.1: BCH Encoder logic comparison

Architecture	**t**	**XOR**	**D/FF**	**AND**	**Latency**
BCH Encoder (conv)	32	$\frac{1}{2}mt$	mt	0	n
Mulitmode Encoder [41]	32	$\frac{1}{2}mt + (t-1)$	mt	0	n
Hybrid Mulitmode Encoder [40]	32	$\frac{1}{2}mt + (t-1)$	mt	t-1	n
MESG [20]	32	$\frac{1}{2}mt + (t-1)$	mt	0	n

The number of XOR gates shown in the Table 3.1 are obtained considering average addition numbers of GF constant multiplication without common sub-expression

elimination (CSE) to prevent increased critical path and large fan-out issues. From the results, the proposed encoder results are comparable to [41] and [40], and it occupies the same area as [41].

For the encoding scheme proposed in [41] and [40], the cascaded module can be unfolded to improve timing. The case analysis for BCH code (m=13,t=16) for general BCH encoder and the syndrome generator with the method proposed is shown in Table 3.2. The different error correcting capabilities used for this analysis are 4, 8, and 16; the VLSI implementation, with an unfolding factor of 16, is performed with 18nm technology. The final gate count of the implementation is shown in Table 3.2. For 16-bit error correction, and with multimode error capability, there is an area saving of 25%. It is interesting to note the increase in MUXes for higher bit error correction because the number of configurable errors to be corrected increases as t increases. For smaller error correction, the area saving is more because the number of errors to be corrected in multimode is less than higher order bit rates. Also, the area consumed for 16-bit error correction is more because of the buffers added to meet fan-out in 18nm technology.

Table 3.2: BCH Encoder area comparison

Architecture	**t**	**Gates**	**D/FF**	**Mux**	**Area** (μm^2)	**Saving**
BCH Encoder [41]	4	3120	104	0	94272	NA
MESG [20]	4	1610	52	916	60586	35.7%
BCH Encoder [41]	8	6694	208	0	130128	NA
MESG [20]	8	3514	104	1462	90586	30.3%
BCH Encoder [41]	16	12714	416	0	378192	NA
MESG [20]	16	6013	208	3671	282344	25.3%

The area consumed by the KES and the Error corrector module is not discussed in our table since the area consumed is the same as in [31]. There are some improvements to the KES algorithm proposed in [36]. The focus of this research is on the hybrid

approach, so we limited our investigation on the programmable feature of the syndrome generator and area optimization of the same.

3.6 Summary

In this chapter, we presented a novel method to encode the BCH code for NAND Memory systems. The encoding scheme splits the remainder of each minimal polynomial and stores them in a separate error-free spare area. Later this stored remainder is used to initialize the syndrome generator at the receiver end, assuming this residue is free of errors. The proposed encoding scheme provides a mechanism to share the same block for the encoder and the syndrome generator, thus saving 25% of the combined area of the syndrome generator and the encoder. This makes the proposed method more suitable for current NAND Flash memory system with error correcting capability. For more reliability on the remainders in the spare area, a sub-field error correction on the spare area is proposed in Chapter.5.

The limitations of this VLSI implementation is that it is dependent on a fixed $GF(2^m)$, and the VLSI area consumed by the KES and CS grows exponentially as the number of bit errors increases. This led us to investigate a flexible solution using GPGPUs in Chapter 4

Chapter 4

GPU Implementation

4.1 GPU system

The GPU utilizes the Single Instruction Multiple Thread (SIMT) architecture which is similar to the Single Instruction Multiple Data (SIMD) architectures. A key difference is that SIMD vector organizations expose the SIMD width to the software, whereas SIMT instructions specify the execution and branching behavior of a single thread. In contrast with SIMD vector machines, SIMT enables programmers to write thread-level parallel code for independent, scalar threads, as well as data-parallel code for coordinated threads. When a multiprocessor is given one or more thread blocks to execute, it partitions them into warps, and each warp gets scheduled by a warp scheduler for execution. The way a block is partitioned into warps is always the same; each warp contains threads of consecutive, increasing thread IDs with the first warp containing thread. Fig.4.1 represents the thread organization of the GPU platform. The number of blocks and warps that can reside and be processed together on the multiprocessor for a given kernel depends on the number of registers and shared memory used by the kernel and the number of registers and shared memory available on the multiprocessor. There are also a maximum number of resident blocks and a maximum number of resident warps per multiprocessor. It is also possible to share the GPU memory across multiple hosts as shown in [53].

There are two broad classifications of memories: host memory and device memory.

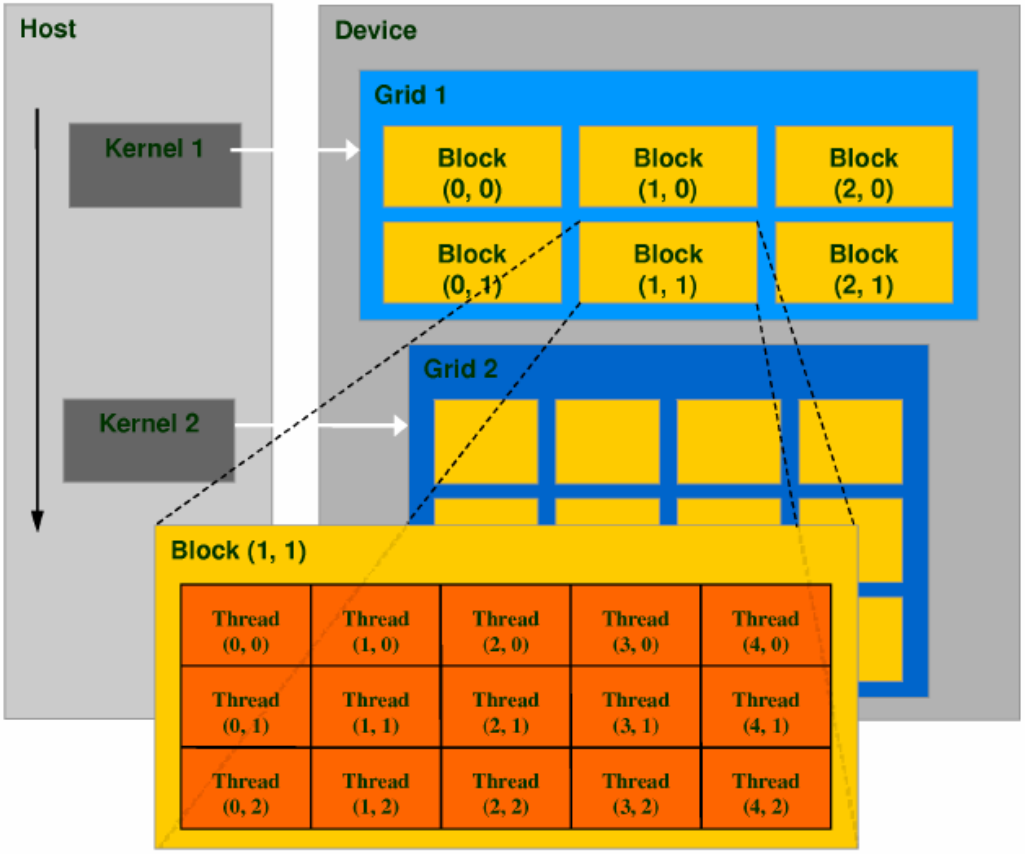


Fig. 4.1: GPU thread organization

The host memory is the memory that resides on the system, and the device memory is the memory that resides on the GPU. The device memory can further be classified into three categories: global memory, shared memory, and local memory, as shown in Fig. 4.2. The global memory is accessible across all the grids, the shared memory is accessible within the blocks, and the local memory can be accessible across threads. The data is copied from the host memory to the GPU device memory, and the computations are done at the GPU end. For high throughput efficiency, the data read from the device are page aligned (2Kbytes), and the typical latency is between 80 and 100 microseconds (us). More details on GPU architectures and CUDA programming can be found in [14].

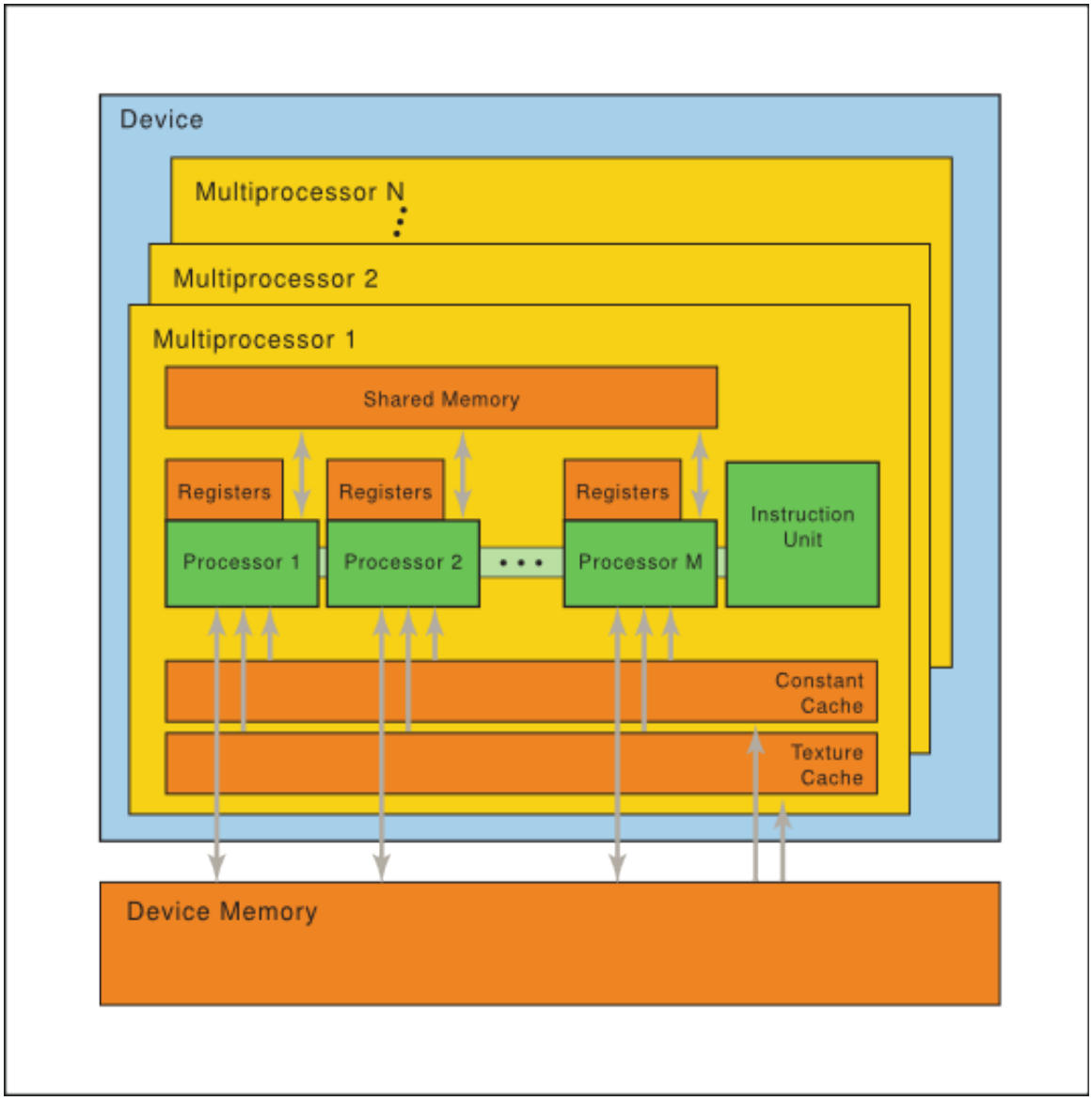


Fig. 4.2: Organization of memories within GPU

4.2 GPU Kernel routines

Kernel routines are an integral part of the GPU implementation. Similar to the VLSI implementation, there are three kernel routines used for GPU implementation: Syndrome Kernel (SK), Key Equation Kernel (KEK), and Chien search kernel (CSK). Fig. 4.3 represents the flow chart for the GPU implementation. For consistency, we explain the flow for a sector size of 512 bytes (i.e., GF dimension of 13) below. Initially,

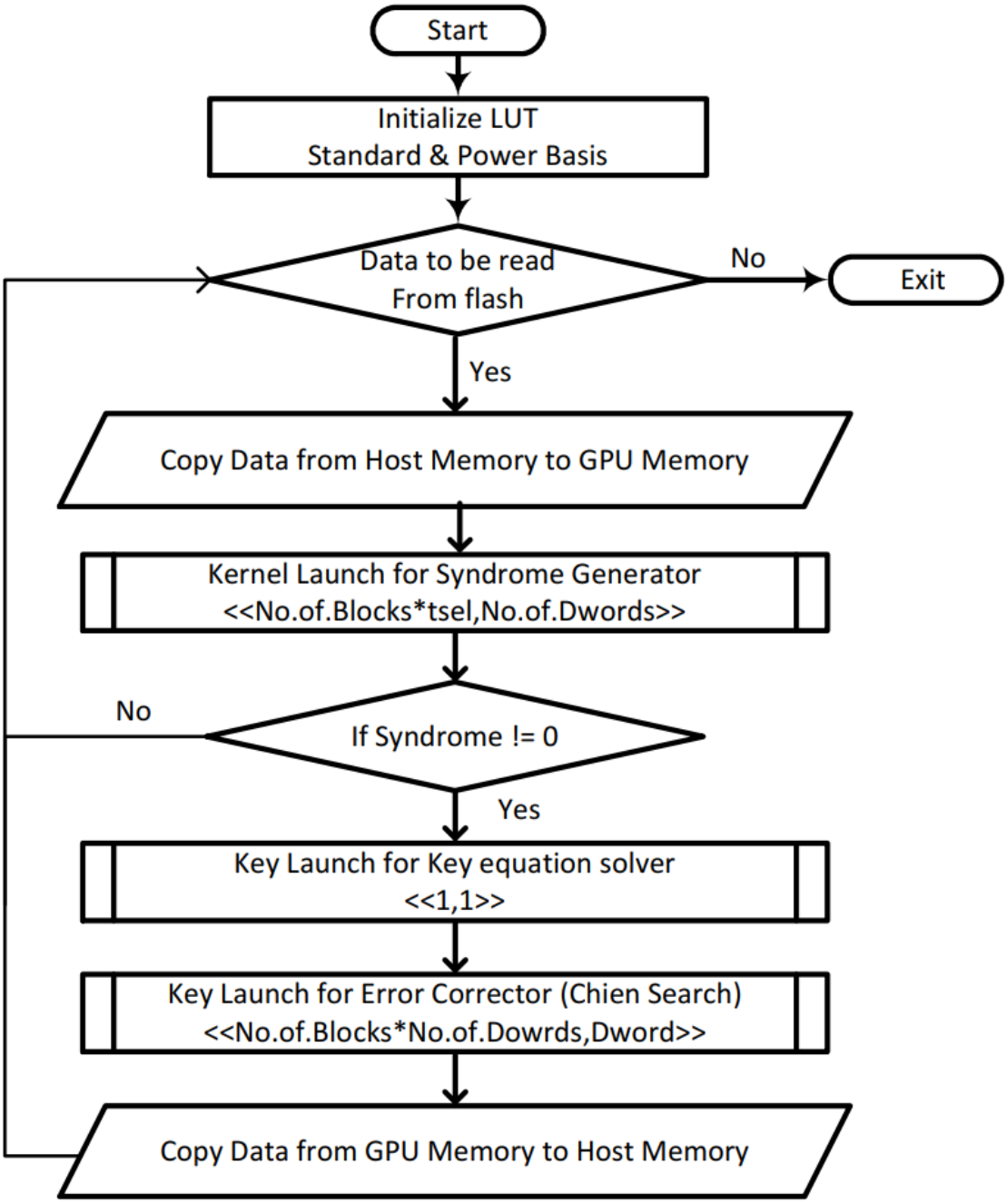


Fig. 4.3: Flow chart for GPU implementation

an internal LUT for the finite field is initialized. The LUT occupies 8 Kilobytes of global memory space in the GPU for the standard basis to the power basis converter. Also, another 8 Kilobytes is used for the inverse function, i.e., power basis to standard basis converter. Once the GPU memory is initialized a single page of 2K is read from the

flash memory and stored to the GPU local memory. Once the data is moved from host memory to the GPU device memory, each data block is moved to the kernel routines for easier access. The kernel routines can work on this shared memory of data moved from the device memory. The advantage of the GPU implementation is the parallel threads offered by the warp schedulers. Another advantage of the GPUs is the fast memory access between the threads. The memory shared across the threads have the fastest memory access followed by the memory in the global device memory. The syndrome generator and the error locator are the obvious choices for the kernel routines. First, the SK routine is launched with threads spawning for each DWord and the number of blocks. Next, the syndromes are checked for non zero values. If so, the KEK and the CSK routines are launched with the appropriate thread dimensions. Finally, the corrected data is then moved from the GPU memory to the host memory.

4.2.1 Syndrome Generator

The first step in the error correction is to find the syndrome from the received data pattern. The syndromes are generated for each of the minimal polynomial used to formulate the BCH generator polynomial $g(x)$. Each of the syndromes are represented by S_i, which is computed as given in Eq.4.1

$$S_i = \sum_{j=0}^{n-1} r_j \cdot \alpha^{ij} \tag{4.1}$$

Algorithm 2 represents the pseudo code for the SK routine.

The thread launch for the syndrome generator is viewed as a two-dimensional vector. The first variable (dimension) provides the dword position of a given BCH block, and the second variable (dimension) provides the position of the block within a given page. The

Algorithm 2 Syndrome Kernel

1: **procedure** SYND KERNEL($Data, DWPos, S_{k,i}$)
2: $synd \leftarrow 0$
3: **for** $j \leftarrow 0, 31$ **do**
4: $pos \leftarrow DWPos * 32 + j$
5: $synd \leftarrow synd + Data[DWPos * 32][j] \cdot (\beta^i)^{pos}$
6: **end for**
7: $atomicXor(S_{k,i}, synd)$ ▷ synchronize writes
8: **end procedure**

different syndromes to be generated are launched as GPU blocks instead of threads. The syndromes do not have dependencies with each other, so they are launched as different blocks in the GPU kernel instead of different threads. The GPU blocks do not share local memory, so it is preferred to use GPU blocks for different syndromes. If the syndromes calculated are zero, then there is no error detected. In the case of an error, the syndromes are not zero, and the key-equation is formed to find the error location.

4.2.2 Parallel Syndrome Generation

Hard-decision BCH decoders have three submodules: syndrome generator, key-equation solver, and an error corrector. Figure 4.4 represents the flowchart for a GPU realization of a BCH decoder for a Flash memory system. The key two basic operations performed on the GF are addition, and multiplication. The GF addition is a simple XOR operation, and the multiplication could be performed on a different basis. [36] discusses these methods in detail. Results from [23] has proven that LUT based multiplication is a suitable solution for GPU implementation. The focus of this section would be the optimization of the syndrome generation routine for GPU systems. The first step is to initialize the GPU global memory with the LUT for power basis, and standard basis representation. The data is then read from the flash memory and copied to the GPU device memory. The kernel routine for the syndrome generator is launched in parallel, where the parallel thread depends on the number of errors to be corrected (t),

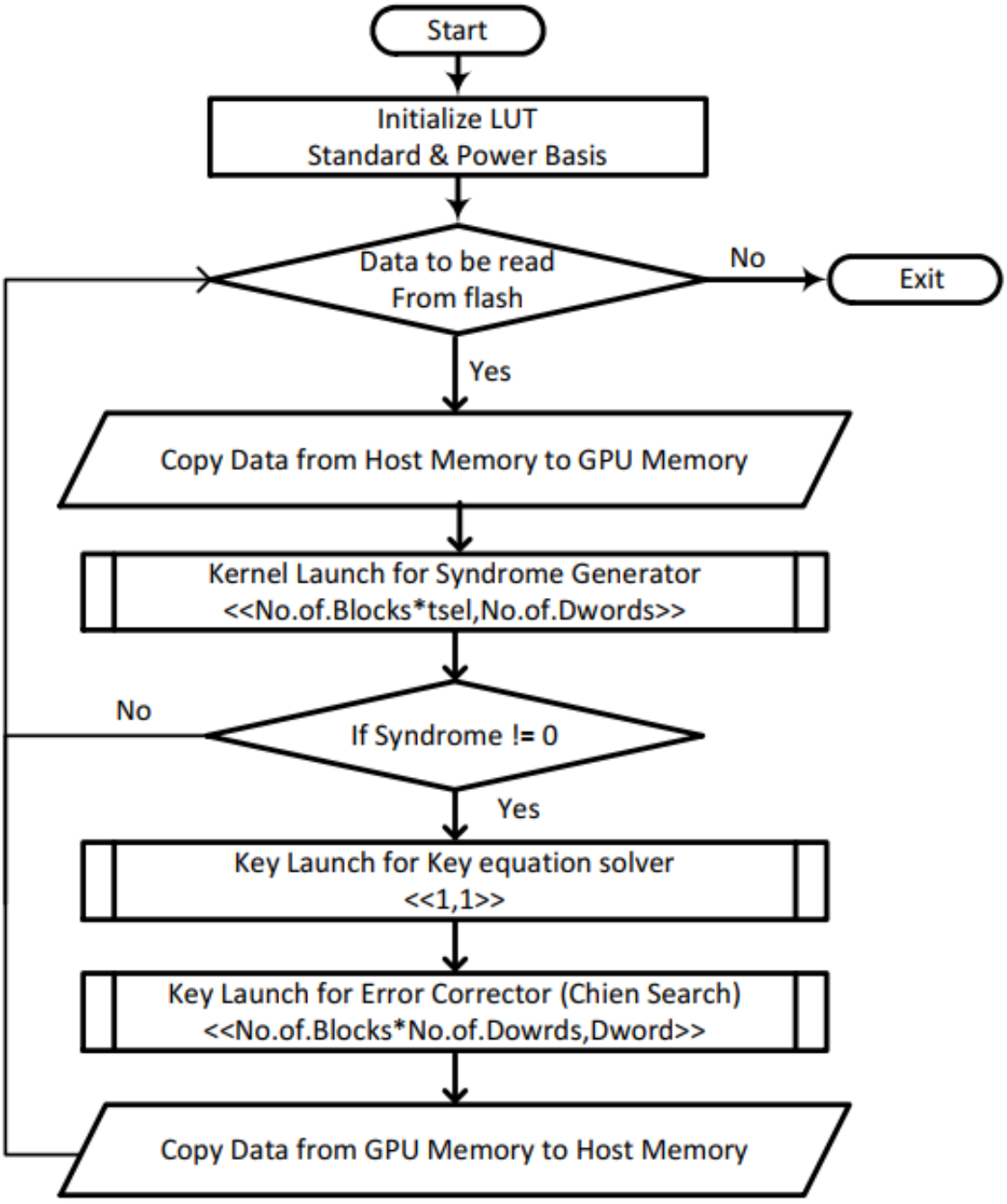


Fig. 4.4: Flow chart using parallel syndrome generation

and the number of blocks read from the flash memory. The syndrome kernel launches a sub kernel routine for each 32-bit. If the syndromes collected are non zero values, the

key-equation solver is launched. Finally, the roots of the key-equation are found by the Chein search kernel routine.

4.2.2.1 Polynomial division

One of the performance degradation for the syndrome generator is the frequent memory access across different threads. It is, therefore, causing memory congestion on the device memory of the GPGPUs. The congestion could be avoided by dividing the incoming stream with the minimal polynomial for the syndrome instead of computing the sum as given in eq.4.1. An alternate way to generate the syndrome shall be given by eq.4.2

$$\begin{aligned} res_i(x) &= r(x)\, mod\, \phi_i(x) \\ S_i &= \sum_{k=0}^{m-1} res_i(\alpha^i)^k \end{aligned} \tag{4.2}$$

where the $res_i(x)$ is the residual polynomial, $r(x)$ is the received code word, and $\phi_i(x)$ is the minimal polynomial of the BCH code with degree m. This method removes access to the LUT for the GF arithmetic and the atomic operation to accumulate the sum across multiple threads. Also, it is sufficient to generate t residual syndromes $res_i(x)$ because of the redundancy across $2t$ syndromes. Hence, it will reduce the number of syndromes to be calculated by a factor of two.

4.2.2.2 Splitting received code

The received code vector $r(x)$ can be represented as sum of orthogonal vectors which is represented in eq.4.3,

$$r(x) = \sum_{j=0}^{bsize-1} r_j(x) \tag{4.3}$$

where $bsize$ is the size of the code block in Dwords, and $r_j(x)$ is the vector $r(x)$ split into orthogonal vectors. The vector $r_j(x)$ is given by eq.4.4, which is true only for $GF(2)$.

$$r_j(x) = r(x).(x^{32.j} \sum_{k=0}^{31} x^k) \tag{4.4}$$

The bisected received vector $r_j(x)$ is then divided by the minimal polynomial as given in eq.4.2. The splitting of $r(x)$ and the polynomial division by $\phi_i(x)$ to compute the syndrome S_i is depicted in Figure 4.5. Since there is no dependency on each of the $r_j(x)$, this proposed method provides the ability to split the code $r(x)$ into multiple independent threads. Now the syndrome can be calculated across the bisected code vector $r(x)$, and it is given by eq.4.5.

$$\begin{aligned} res_{i,j}(x) &= r_j(x) \, mod \, \phi_i(x) \\ S_i &= \sum_{j=0}^{bsize-1} \sum_{k=0}^{m-1} res_{i,j}(\alpha^i)^k \end{aligned} \tag{4.5}$$

The vector $res_{i,j}$ represents the residual polynomial for each bisected received code vector $r(x)$, and $\phi_i(x)$ is the minimal polynomial for the GF element α^i. Since S_i is composed into two variable sums, this is easily realizable as multiple threads in GPGPUs routine. It is possible to sum the $res_{i,j}$ first and then calculate the syndrome by substituting the appropriate field element α^i in the equation.

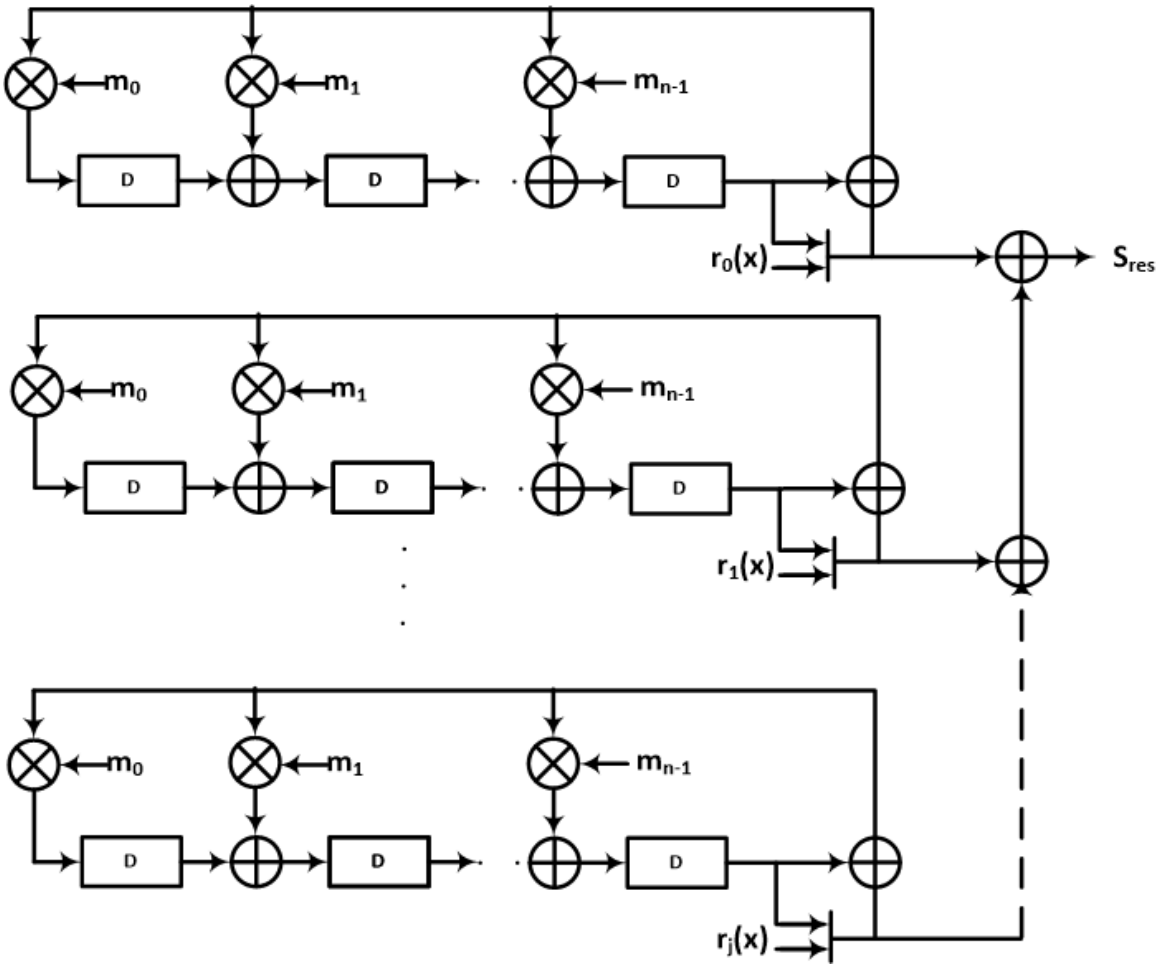


Fig. 4.5: Parallel polynomial division

4.2.2.3 Kernel implementation

The syndrome generator and the error corrector are the perfect candidates for the GPGPUs kernel routine. The kernel for the syndrome generator are split into two folds. First, the number of syndromes to be calculated depends on the t bit error, and the number of code blocks within a BCH code block. Second, each syndrome can spawn a separate thread for each of the Dword with it's position. The first order kernel routine is launched with a two dimensional vector, where the first vector represents the code block and the second vector represents the syndrome identification. Although the key-equation solver $\lambda(x)$ requires $2t$ syndromes, it is sufficient to calculate only t syndromes

Algorithm 3 Syndrome generator Polydiv Kernel

```
1: procedure SYND POLYDIV KERNEL(data, pos, S_i)
2:     l ← data[pos]
3:     p ← pos
4:     φ_i(x) ← minimal polynomial
5:     while p ≠ 0 do
6:         for i ← 0, 31 do
7:             if l[i] then
8:                 l ← l mod φ_i(x)
9:             end if
10:        end for
11:        p ← p − 1
12:    end while
13:    atomicXor(S_i, l)                    ▷ to synchronize between threads
14: end procedure
```

which are because of the redundancy of minimal polynomials used to generate $g(x)$.

The steps to create the syndrome from a given data vector is given in Alg.3. It is important to note that these are kernel routines and each of the procedure (SYND KERNEL) calls are launched in parallel. The *atomicXor* operation is used as a synchronization mechanism between different threads. This number of threads launched for this routine is given by the number of Dwords($bsize$) for the received code vector $r(x)$.

4.2.3 Key-equation solver

The KEK routine is responsible for forming the key-equation $\sigma(x)$ which will be used for the error corrector. Since this KEK routine uses the iterative algorithm (iBMA), there is no room for parallel threads here. The alternate algorithm of choice is Peterson's algorithm, but this algorithm requires solving a matrix which consumes more time because of the requirement of inversion. It is observed that there is no dependency on the GF dimension for the BCH code. However, as the number of bit errors to be corrected increases, the time consumed by this thread also increases. The details of the

KEK routine using iBMA is explained below. First, the variables used in the iBMA are initialized as shown in the following sequence.

(i) $d_p \leftarrow \begin{cases} 1;\ S_1 = 0 \\ S_1;\ S_1 \neq 0 \end{cases}$

(ii) $\beta^{(1)} \leftarrow \begin{cases} x^3;\ S_1 = 0 \\ x^2;\ S_1 \neq 0 \end{cases}$

(iii) $l_1 \leftarrow \begin{cases} 0;\ S_1 = 0 \\ 1;\ S_1 \neq 0 \end{cases}$

(iv) $\sigma = 1 + S_1 x$

(v) $r \leftarrow 1$

Second, the iterative portion of the iBMA algorithm is given as

Step 1: $d_r \leftarrow \sum_{i=1}^{t} \sigma_i^{(r)} S_{2r-i+1}$

Step 2: $\sigma^{(r)} \leftarrow d_p \Lambda^{(r-1)} + d_r \beta^{(r)}$

Step 3: $bsel \leftarrow \begin{cases} 0;\ d_r = 0\, or\, r < l_r \\ 1;\ d_r \neq 0\, and\, r \geq l_r \end{cases}$

Step 4: $\beta^{(r+1)} \leftarrow \begin{cases} 0; x^2\beta^{(r)};\ bsel = 0 \\ 1; x^2\sigma^{(r)};\ bsel \neq 0 \end{cases}$

Step 5: $l_{r+1} \leftarrow \begin{cases} 0; l_r;\ bsel = 0 \\ 1; l_r + 1;\ bsel \neq 0 \end{cases}$

Step 6: $d_p \leftarrow \begin{cases} 0; d_p;\ bsel = 0 \\ 1; d_r;\ bsel \neq 0 \end{cases}$

Step 7: $r \leftarrow r + 1$

The steps 1 through 7 are repeated for $t - 1$ iterations. At the end of $t - 1$ iteration, the variable $\sigma(x)$ will be the equation whose roots point to the position of the error. If the syndromes S_i are no zeros, and the $\sigma(x)$ does not have the root in the splitting field dimension of $GF(2^m)$, then the routine has encountered an uncorrectable error. In the case of uncorrectable error, the bad block management software should mark this as a bad block and remove this block from further writing. Also, the number of errors detected in the block is registered. When the correctable errors are closer to the maximum number of error within a sector, the software can mark it as a bad block and move the data to a different block. This method will provide proper wear level handling for the NAND flash.

4.2.4 Chien Search Kernel

The CSK is the final routine to be executed in the decoder. Alg. 4 represents the pseudo code for the CSK routine. The primitive element $\alpha^{pos^{-1}}$ is evaluated as a

Algorithm 4 Chien search Kernel

1: **procedure** CHIEN KERNEL(Λ, pos, err)
2: $sum \leftarrow \sum_{j=1}^{deg(\Lambda)} \Lambda_j(\alpha^{pos^{-1}})^j$
3: *atomicXor(err[pos], sum)* ▷ Prevent overlap write
4: **end procedure**

root for $\sigma(x)$. This CK routine is an ideal candidate for GPU because the evaluation of each $\alpha^{pos^{-1}}$ in the equation $\sigma(x)$ is independent. Similar to the SK routine, the memory

within the GPU device is shared between threads, so the *atomicXOR* operation is used to avoid writing overlap between different CSK routines. Once the error vector is formed, the error is masked with the data in memory to yield corrected data.

4.3 Memory enhancement

In this section, we propose an alternate scheme for storing the parity bits of the BCH encoder, which provides a scalable architecture for GPU implementation. The user data area is used to store the parity bits instead of the spare area. Typically, the user data is divided into a BCH block size of 512 bytes, and the associated BCH parity bits are stored as part of the spare area. Therefore, there is a limitation on the number of bit-error that can be corrected using the spare area. For instance, a spare area of 64 bytes on a 2 Kilobyte block would have a limitation of 9-bit error correction. The field of operation in this scenario would be $GF(2^{13})$, and for 9-bit error correction, the parity bits needed would be 117 bits. Fig. 4.6 depicts the proposed organization of parity bits as part of the flash memory page. For consistency, all the finite field arithmetic in our discussion within this section will be in $GF(2^{13})$ unless stated.

The parity bits for the bit block (64 pages) are stored in the first two pages of the block. Thus, the parity bits are accumulated in a single area instead of spreading across pages. The user data is then stored from the subsequent pages. The spare area of the user data blocks is then proposed to be used for bad block management or other software management parameters. The blocks containing the parity bits are read to the GPU memory and stored in GPU′s device global memory for reference. Since the hardware implementation of this scheme requires more memory to store the parity bits or many page reads, this method is suitable for software-based solutions. The unused spare areas are then proposed to be used for bad block and boot sector management.

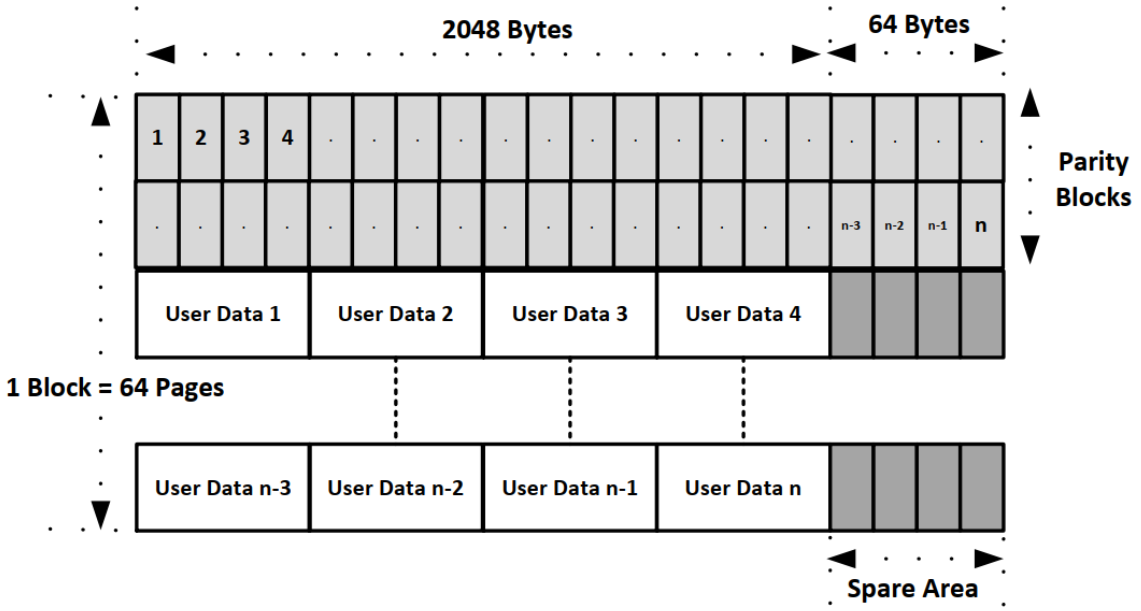


Fig. 4.6: Enhanced memory organization of Flash memory

4.3.1 Enhanced memory flow chart

Fig. 4.7 represents the flow chart for the proposed algorithm on a GPU platform. Initially, an internal LUT for the finite field is initialized. Experiments have shown that having a LUT based approach for GF multiplication yield faster results than conventional standard basis multiplier in software. The LUT occupies 8 Kilobytes of global memory space in the GPU for the standard basis to the power basis converter. Also, another 8 Kilobytes is used for the inverse function, i.e., power basis to standard basis converter. After the GPU global memory is initialized, a 4 Kilobyte memory space is allocated as a cache area for the parity. When a new page is read from the flash, the first two blocks are stored in this cache area for parity reference. The subsequent blocks are then read and then follows the standard BCH decoding procedure. The cache management for the parity bits is not limited to 4 Kilobytes, and it is possible to extend the cache to more significant memory and store parity bits that can span multiple pages.

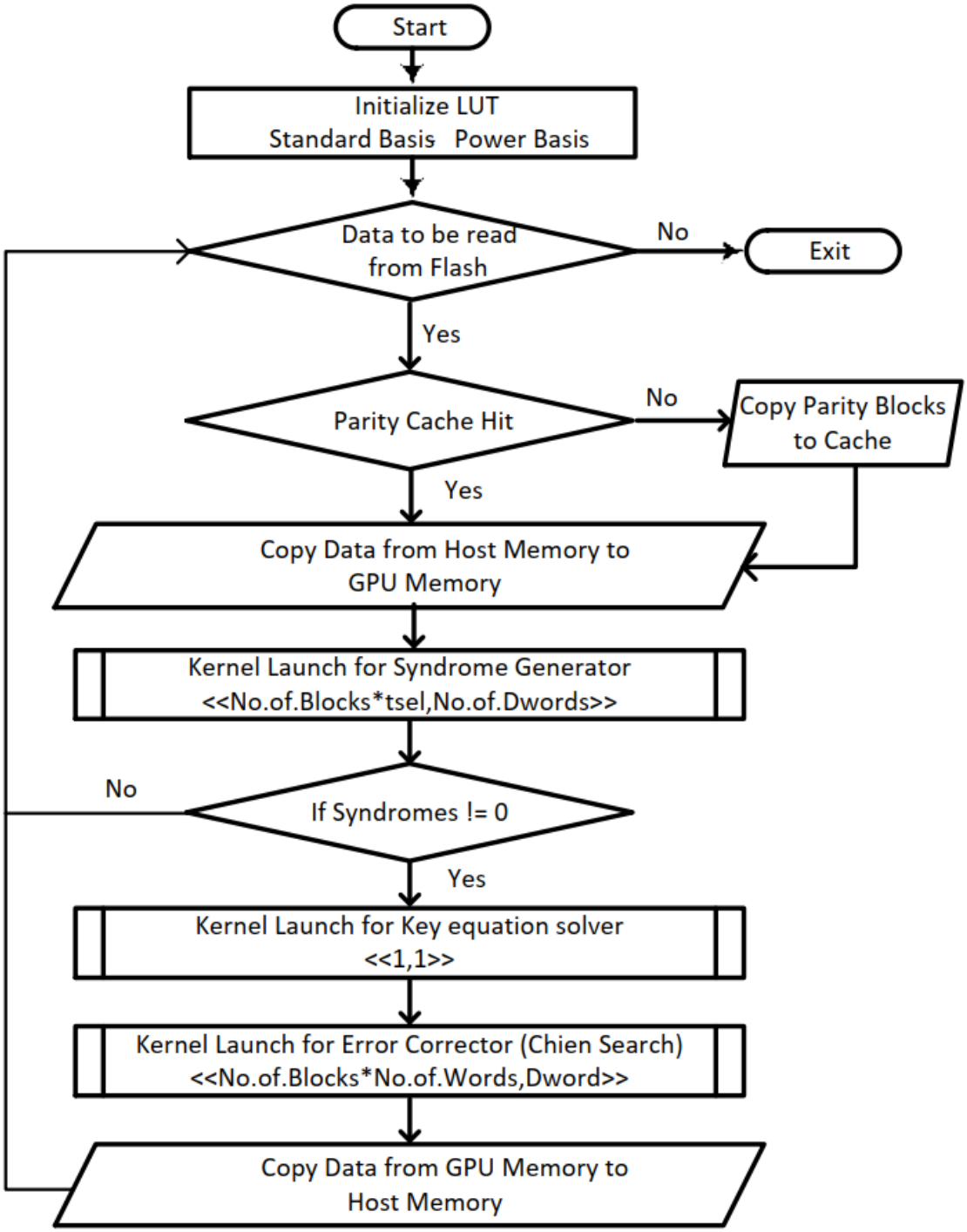


Fig. 4.7: Enhanced memory organization of Flash memory

Once a parity cache hit is resolved, the data from the flash memory is moved to the GPU memory for error correction.

4.4 Experimental results

We have studied the performance of the GPU implementation in two setups which are listed in Tab. 4.1 and Tab. 4.2. Setup1 is a GPU card used for GPGPU application, and it is important to note that the GPU was used for both graphical display and ECC computation. Tab. 4.1 describes the configuration used for setup1.

Table 4.1: Experimental setup1

	GPGPU	**CPU**
Platform	Geforce GTX 760. 1152 cores	Intel Xeon i7
Clock Freq.	1.033 GHz	3.7 GHz
Memory Type	GDDR5(2GB)	DDR2(32GB)
Memory Speed	6 $Gbps$	102.4 $Gbps$
Architecture	Kepler	X85
OS	RedHat Linux	RedHat Linux

Table 4.2: Experimental setup2

	GPGPU	**CPU**
Platform	Jetson TX1. 256 cores	Quad core ARM
Clock Freq.	1.024 GHz	1.024 GHz
Memory Type	LPDDR4, 4 GB	LPDDR4, 4 GB
Memory Speed	4 $Gbps$	4 $Gbps$
Architecture	Maxwell	ARM
OS	Ubuntu Linux	Ubuntu Linux

Setup2 is an embedded GPU card used for GPGPU application, and it is important to note that the embedded GPU was used for the display and for ECC computation time. Also, this setup ran the complete OS with the inbuilt ARM processor. Tab. 4.1 describes the configuration used for setup2. Fig. 4.8 shows the embedded kit (setup2) for our experiment.

Fig. 4.8: GPU setup2

4.4.1 GPU kernel performance

For this experiment, we used setup1, Alg. 2 for SK, iBMA for KEK, and Alg. 4 for CSK.

4.4.1.1 GPU computation time

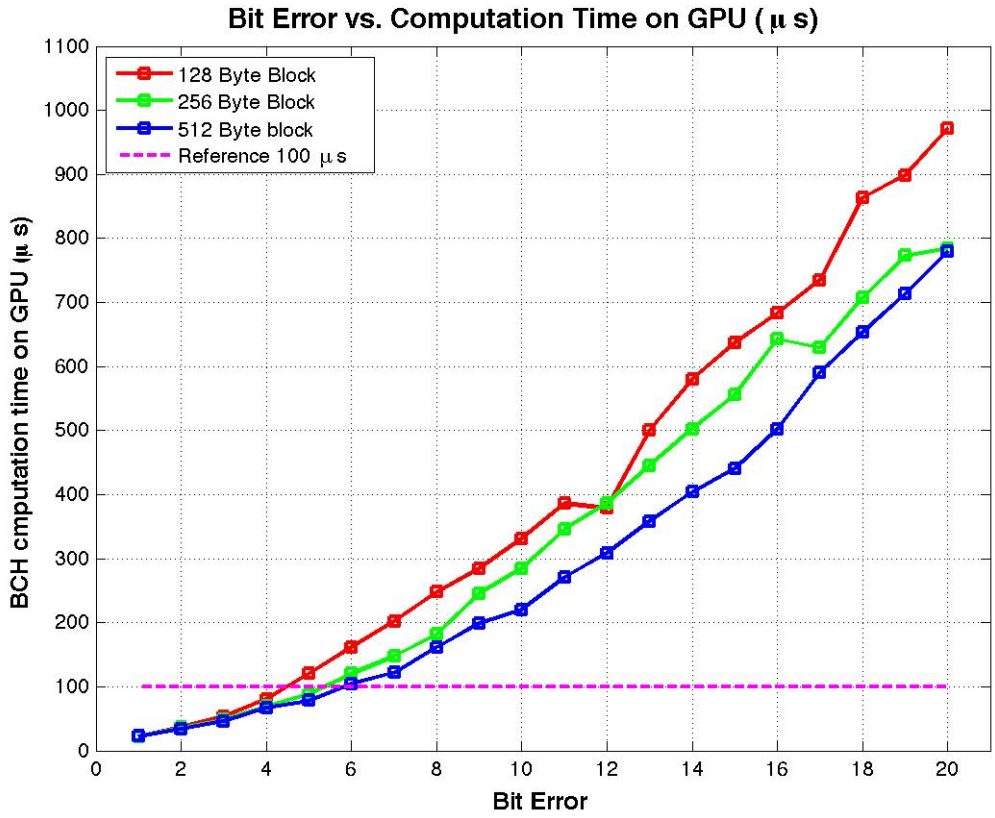


Fig. 4.9: GPU comp. time

Fig. 4.9 shows the Bit Error vs. GPU computation time for various BCH block sizes on a 2Kbyte page. The GPU computation time comprises of the following routines: memory from/to host/GPU memories, syndrome generator, iBMA solver, and Chein search module. The computation time is less than the page memory read latency for bit- error rate less than six on a 512-byte BCH block, and it allows the system to have

the ability to correct 24 bits on a 2Kbyte page without any performance hit. There are many threads required to cover a 2K Byte for different BCH block size, so a 20-bit error correction on a 128-byte BCH block has more computation time than a 512 Byte BCH block which is due to the execution time for the threads.

4.4.1.2 Syndrome computation time

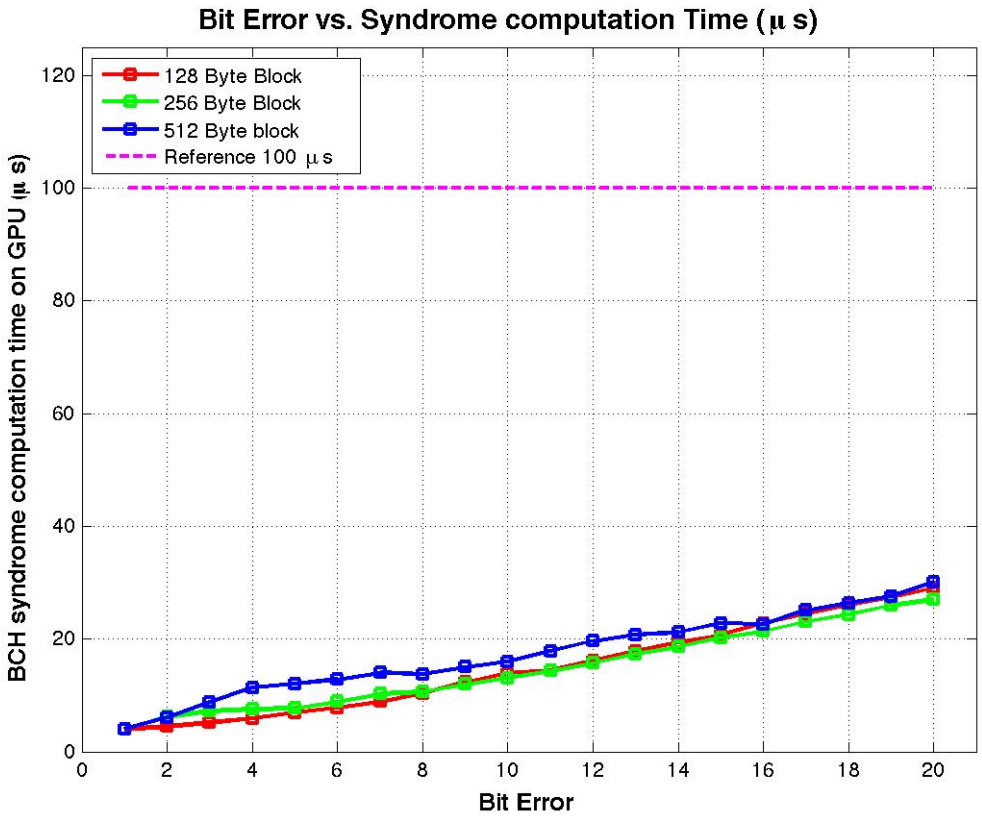


Fig. 4.10: SK comp. time

Fig. 4.10 depicts the Bit Error vs. Syndrome generator CUDA kernel routine for various BCH block sizes over a 2Kbyte page. It is evident that the syndrome generator CUDA routine computation time is less than 25% of the page read latency of the flash memory device. The iBMA and the Chien search algorithm are executed only in the case

of an error which makes GPU implementation feasible for systems with flash memory. For non-error scenarios, without any performance degradation which is confirmed with [50]

4.4.1.3 Individual Kernel computation time

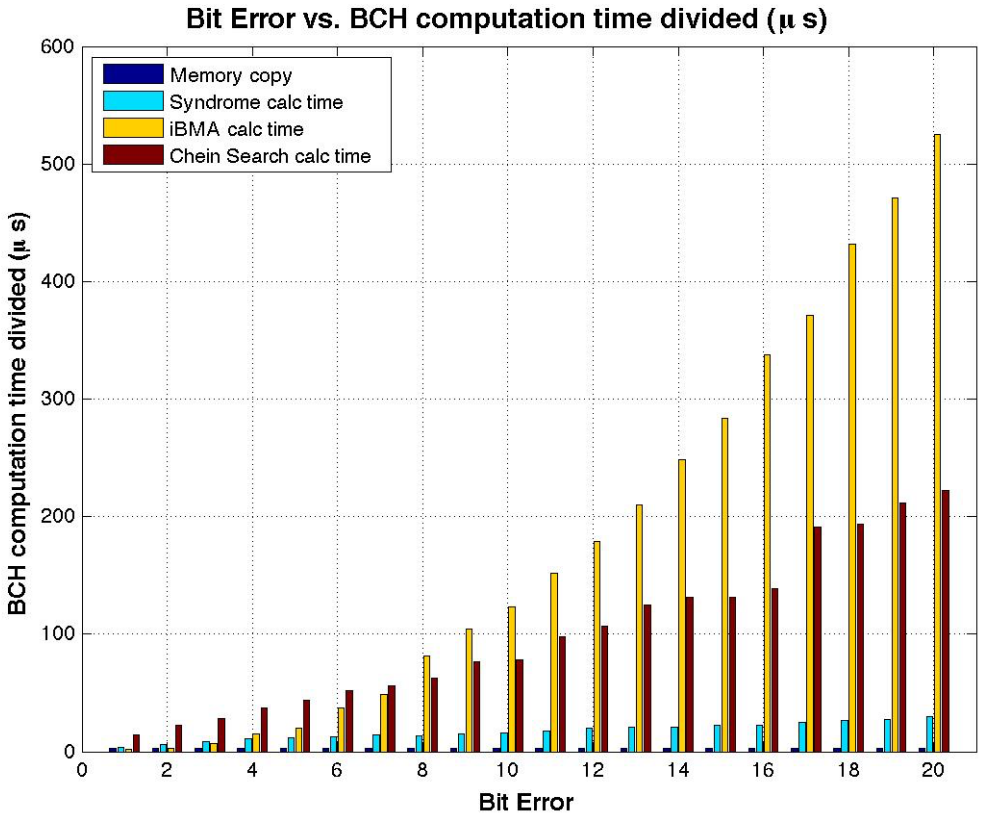


Fig. 4.11: Individual Kernel comp. time

Fig. 4.11 shows the GPU time spent on each individual sub module system for a 512 Byte BCH block size with different bit errors. The maximum time consumed by the BCH decoder is the key-equation solver, which uses the iBMA algorithm. However, compared to the Euclidean method of decoding [50], the computation time is less than 10x fold. Also, the computation time grows exponentially with the bit error rate because

the key-equation has a linear growth in the number of iterations for the iBMA algorithm, and the number of polynomials in the key-equation solver. The computation time by the syndrome generator and memory copy are very minimal when compared to the page read latency. The Chein search has a linear curve with the bit error rate, and an interesting observation is that the performance is the same as non-byte aligned when the bit errors are byte aligned. For example, the Chein search on the GPU implementation has a latency of 120 us for 14,15 and 16-bit error correction.

4.4.2 Syndrome Generator with parallel division

Figure 4.12 plots the bit error vs the syndrome computation time, where method1 represents the Alg. 2 and method2 represents the Alg. 3. We could observe a minimum computation time improvement of 12% until 15-bit error correction. The gain in computation time is due to the following reasons: first, the residual calculation across multiple threads uses independent threads because of the independent summation; second, the memory access to convert standard basis to power basis is removed; finally, the thread execution duration depends on the position of the data for which the thread is executed. Another important factor is the computation time for the syndrome kernel is less than 25µs, which is less than the typical read latency of 100µs for flash memory systems.

4.4.3 Memory Enhancement

In this section, we discuss the results obtained for BCH decoder (n=8191, k=4096, t=2..20) over setup1 (Tab. 4.1) and setup2 (Tab. 4.2).

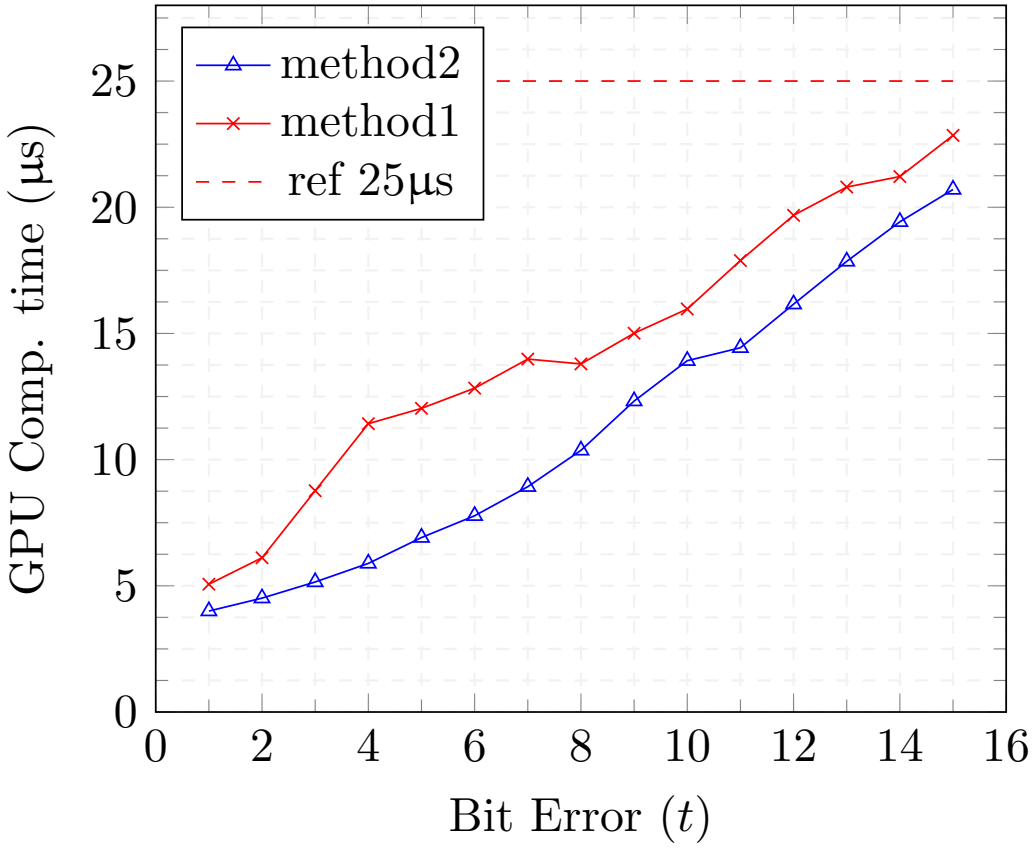


Fig. 4.12: Bit Error vs. GPU Syndrome Computation time

4.4.3.1 Setup1

Fig. 4.13 depicts the bar chart representation of kernel computation time for a 512 bytes BCH block in Setup1. The computation time for the syndrome is the least among all the kernel routines. The memory copy routines (including caching for parity bits) and the initialization routines are negligible since these routines are only used at the beginning of memory data transfer. Because of the lack of parallelization within the key-equation solver routine, for bit error rates higher than 10, the key-equation solver consumes more computation time relative to the syndrome and the Chien search routines. The Chien search routine in comparison to syndrome generator routine consumes more simulation time because the key-equation has to be evaluated for the entire GF elements.

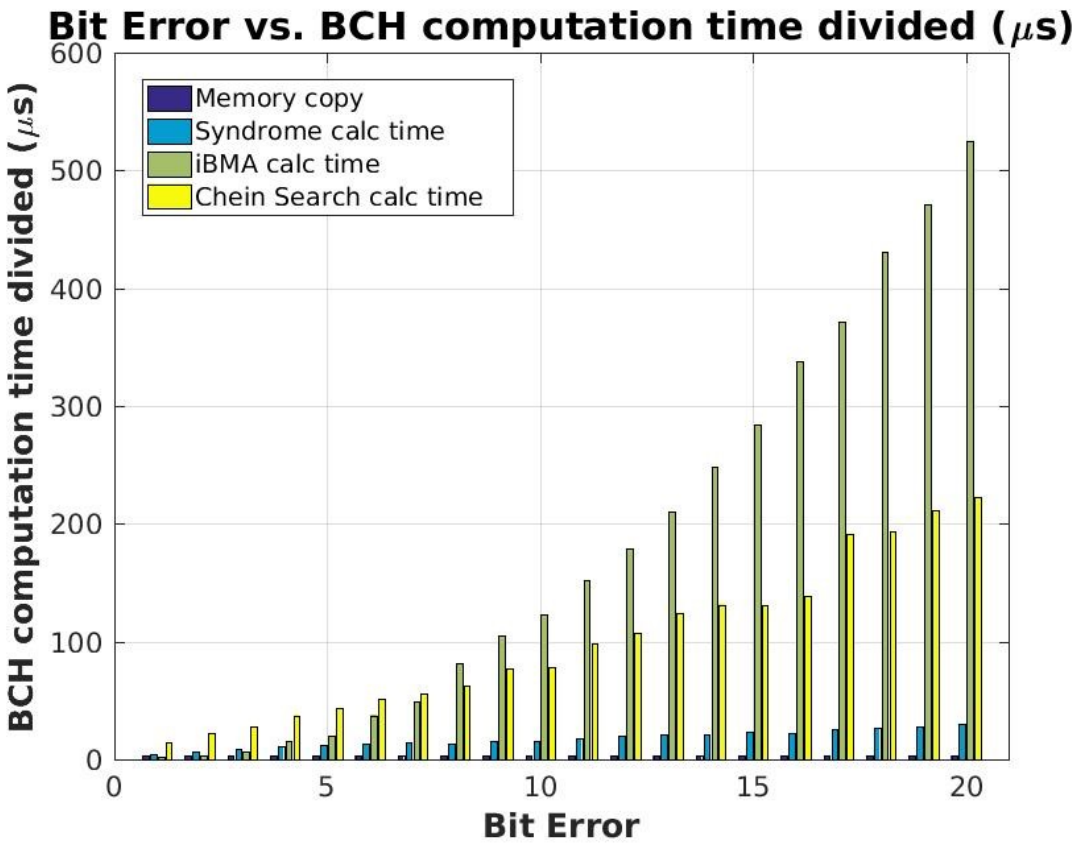


Fig. 4.13: Memory enhancement for Setup1

4.4.3.2 Setup2

Fig. 4.14 shows the bar chart representation of computation time for 512 bytes block on Setup2. On the contrary to setup1, all the subroutines consume the same amount of computation time when the bit error is greater than 15. Another factor is that the number of cores available for setup2 is less than setup1. The amount of time consumed by the syndrome generator is greater than setup1 platform because of the memory congestion.

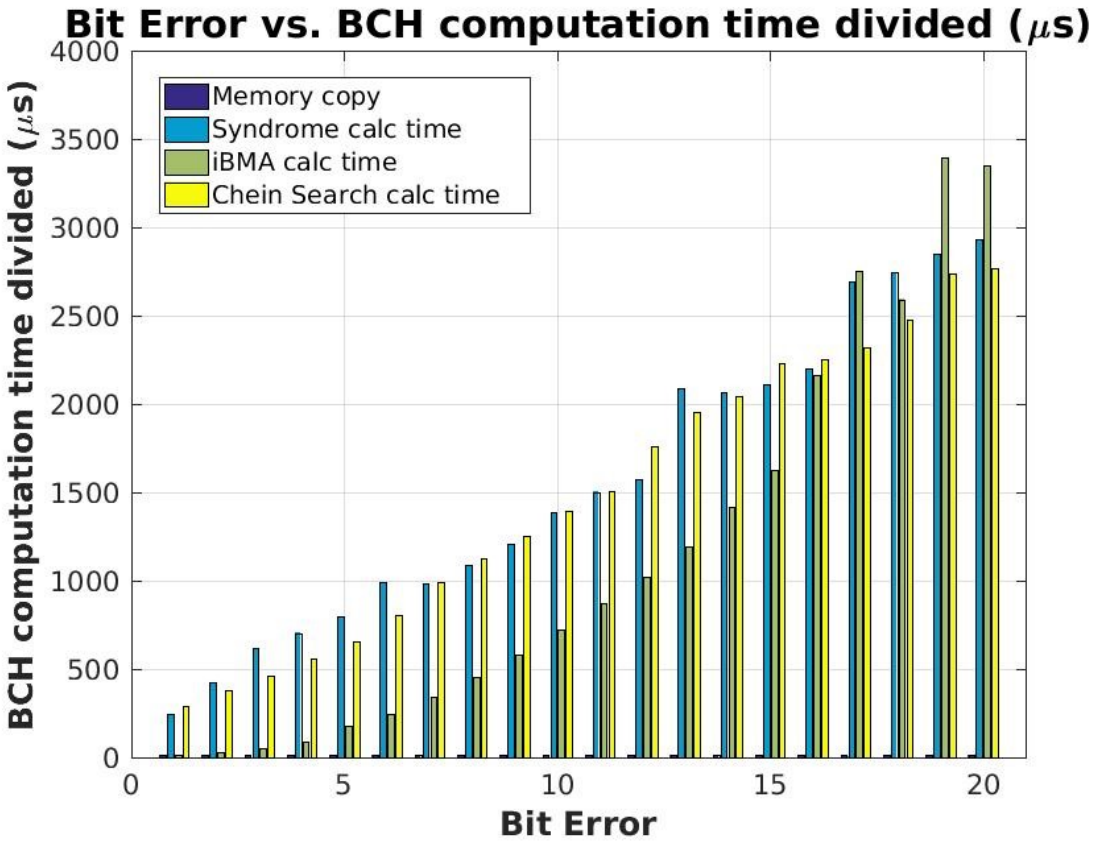


Fig. 4.14: Memory enhancement for Setup2

4.4.4 CUDA profile

Fig. 4.15 represents the CUDA profiling for different kernel routines in setup2 (Tab. 4.2. It could be seen that most of the parallel threads for the syndrome generator takes about 68% of the total computation time.

4.5 Summary

In this chapter, we studied the GPU implementation of the BCH decoders. The KEK routine consumes the most time in our experiments. The CSK and SK routine has more parallel threads and the execution time is faster when compared with the KEK.

We have presented two methods for syndrome generator Alg. 2 and Alg. 3. The results have shown an improvement of 12% until t=16 for Alg. 3 when compared against Alg. 2. We also presented an alternate method to store the parity bits of the BCH codes in the user data area. By adopting this method, we can scale the BCH code for bit errors higher than nine for MLC Flash memories which is a limitation in VLSI implementation. Since GPUs do not have strict buffer limitation, it is tolerable to have dedicated memory to store the parity bits for each Flash memory block.

Our results have shown that there is no degradation in performance for bit errors higher than nine using the proposed storage of parity bits. For bit errors greater than five, the computation time taken by the embedded setup (Table 4.2) consumes more than 100µs and this approach makes it not feasible for low-end GPUs. So, there is a necessity to combine the features from VLSI implementation and GPU implementation which is discussed in detail in Chapter 5.

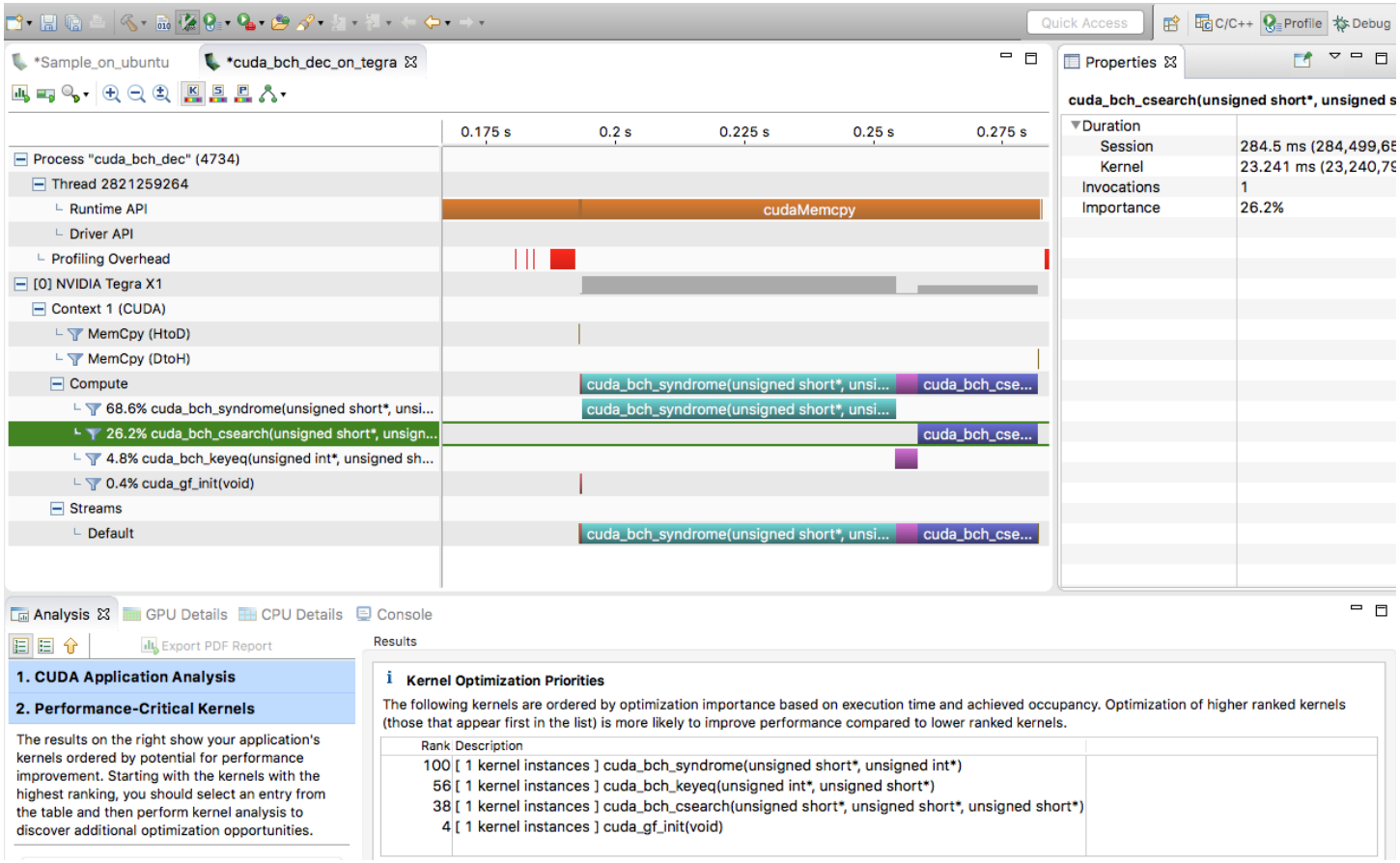


Fig. 4.15: CUDA Profile

Chapter 5

Cascaded hybrid approach

In this section, we will first introduce the proposed C-BCH codes, and we describe the mathematical idea of the cascaded structure and the data flow of our proposed method. For the readers convenience, we have listed the frequently used variables as follows:

k, n : message and code length of BCH code,

t : error correcting capability of BCH code,

$\phi_i(x)$: minimal polynomial of element β^i,

$m_k(x)$: message polynomial of k^{th} block,

$c_k(x)$: code polynomial of k^{th} block code,

$p_k(x)$: parity polynomial of k^{th} block ,

$res_{k,i}(x)$: residual for $\phi_i(x)$ of k^{th} block ,

$eres_{k,i}(x)$: error residual for $\phi_i(x)$ of k^{th} block ,

k_c, n_c : message and code length of C-BCH code,

t_c : error correcting capability of C-BCH code,

N_b : No of blocks used for C-BCH code,

$cm(x), cc(x)$: cascaded message polynomial,

$cc(x)$: cascaded code polynomial,

5.0.1 Main Idea

Let $g(x)$ be the generator polynomial used to encode the cyclic BCH codes over $GF(2)$. The roots of this polynomial equation reside in the extended field, also known as splitting field, $GF(2^m)$. Let $\phi_i(x)$ be the minimal polynomial of an arbitrary element β^i, then the generator polynomial for BCH code(n, k, t) is given by $g(x) = LCM(\phi_1(x), \phi_2(x), ..., \phi_{2t}(x))$. Narrow sense BCH codes use primitive element α^i for

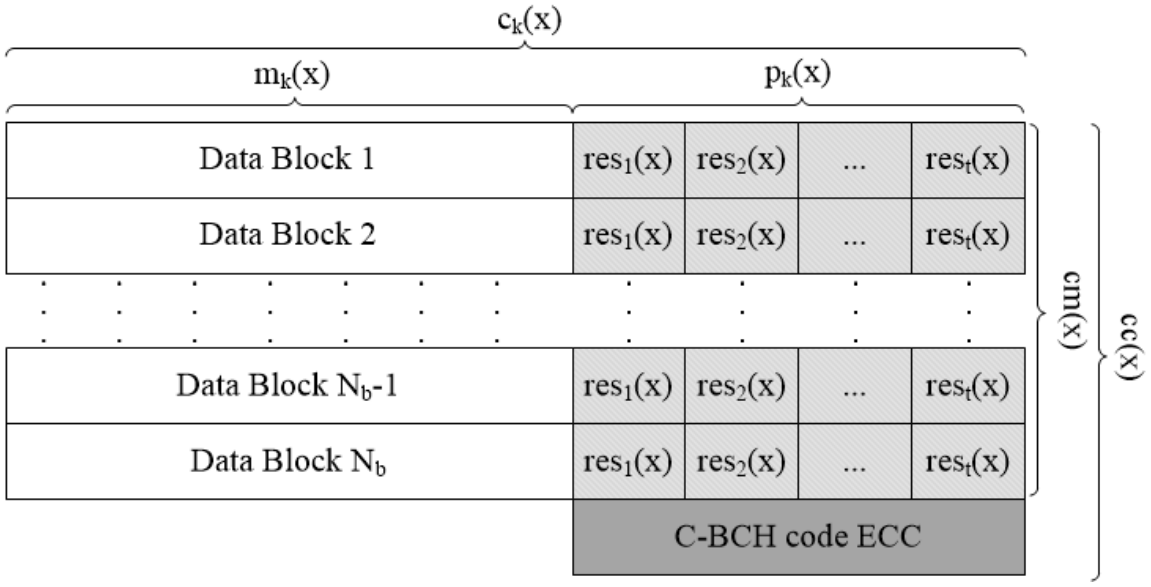


Fig. 5.1: Cascaded BCH code

the minimal polynomial with $i = 1, .., 2t$ instead of an arbitrary starting point for i. For this paper, the narrow sense BCH code decoder is used, which could be easily extended to general BCH codes[12]. Because of the relationship between the conjugacy elements and minimal polynomials[12], the generator polynomial can be optimized as $g(x) = LCM(\phi_1(x), \phi_3(x), ..., \phi_{2t-1}(x))$. The conventional encoded code vector $c(x)$ is given as $c(x) = m(x) \cdot x^{deg(g(x))} + m(x) \, mod \, g(x)$. We propose to use the minimal polynomial $\phi_i(x)$ for modulo arithmetic, instead of the conventional $g(x)$, as in [24], so we can encode the proposed code vector as $c_k(x) = m_k(x) + p_k(x)$. The parity polynomial

$p_k(x)$ is the concatenated residual polynomial for the k^{th} block code and is expressed as

$$p_k(x) = \sum_{i=1}^{t} res_{k,2i-1}(x) \cdot \prod_{j=1}^{i-1} x^{deg(\phi_{2j-1}(x))} \tag{5.1}$$

where $res_i(x) = m(x) \, mod \, \phi_i(x)$. For narrow sense BCH codes, the $deg(\phi_i(x)) \leq m$, where m is the finite field extension ($GF(2^m)$). We propose to simplify the $x^{deg(\phi_i(x))}$ as x^m and add padded zeros where $deg(\phi_i(x)) < m$. Now, Eq.5.1 can be simplified as

$$p_k(x) = \sum_{i=1}^{t} x^{m.(i-1)} res_{k,2i-1}(x) \tag{5.2}$$

Let N_b be the number of blocks that is used for the C-BCH code. The parity polynomial $p_k(x)$ for all the blocks are concatenated to form the cascaded message polynomial $cm(x)$ which is given as

$$cm(x) = \sum_{k=1}^{N_b} p_k(x) \cdot x^{(k-1).t.m} \tag{5.3}$$

Let $g_c(x) = LCM(\phi_1(x), \phi_2(x), .., \phi_{2t_c-1})$ be the generator polynomial to correct errors over the C-BCH code (n_c, k_c, t_c), then the cascaded code vector polynomial $cc(x)$ is expressed as

$$cc(x) = cm(x).x^{deg(g_c(x))} + cm(x) \, mod \, g_c(x) \tag{5.4}$$

Fig. 5.1 represents the proposed layout of the BCH code and the C-BCH code. Let $cr(x) = cc(x) + ce(x)$ be the received code vector with the error vector $ce(x)$. The BCH hard decision decoding algorithm is performed to correct the $cr(x)$ and retrieve the $cm(x)$. The $res_{k,i}(x)$ is then retrieved from the $cm(x)$ which is used to correct the main data block code vector $c_k(x)$. Let $r_k(x) = m_k(x) + e_k(x)$ be the received code vector for the data block with error vector $e_k(x)$, then the $eres_{k,i}(x)$ is generated similar

to the $res_{k,i}(x)$ which is given as

$$\begin{aligned} eres_{k,i}(x) &= (m_k(x) + e_k(x))\, mod\, \phi_i(x) + res_{k,i}(x) \\ &= e_k(x)\, mod\, \phi_i(x) \end{aligned} \tag{5.5}$$

Now the syndrome can be expressed as $S_{k,i} = eres_{k,i}(\beta^i)$. Once the syndrome is generated the iBMA and Chien search algorithms are employed to retrieve the message polynomial $m_k(x)$.

5.0.2 Data Flow

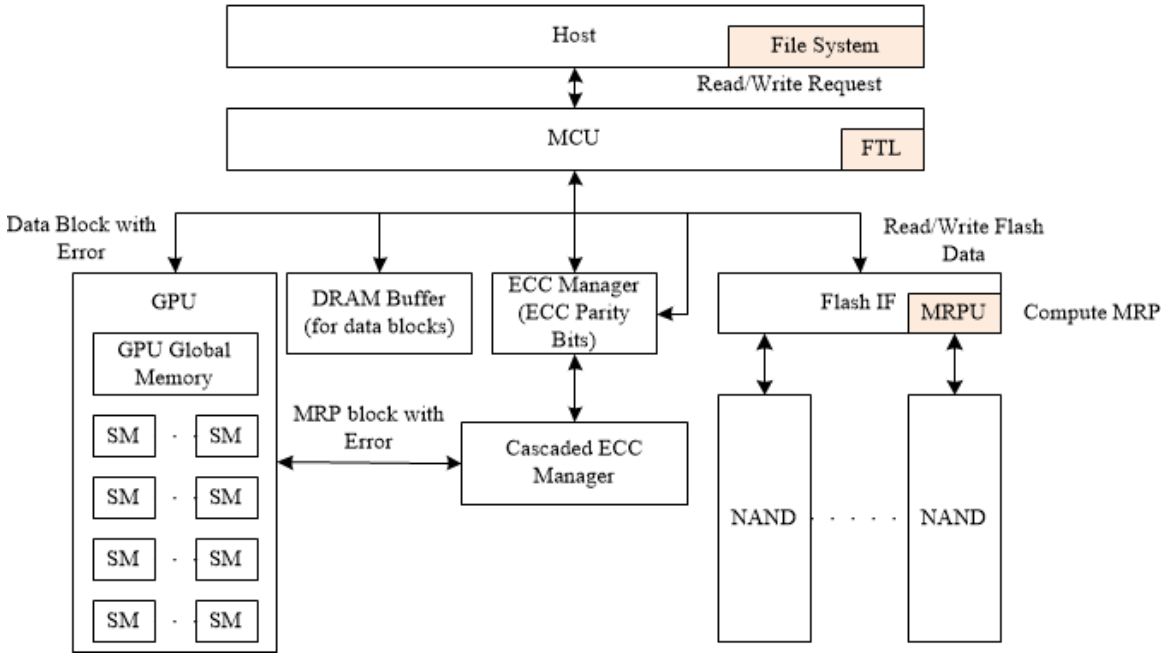


Fig. 5.2: Data Flow for cascaded ECC method

Figure5.2 represents the data flow of our proposed hybrid method using GPUs and hardware circuits. The host system is responsible for the file system, and it initiates a read or a write cycle to the Memory control Unit which has the FTL transaction layer. The Flash interface is a hardware module that establishes the physical interconnect

with the NAND flash memory device. Here we propose to use an array of minimal residual processing units (MRPU) which is used for generating polynomials $res_i(x)$ and $eres_i(x)$. Further, we propose to implement this module in hardware for faster throughput during encoding and syndrome generation step of the decoder. The cache to store the ECC parity bits for the current page within the data block resides within the DRAM memory. The GPU module, as shown in Fig. 5.2 is a subsystem used within the proposed ecosystem. This GPU is used to correct errors within the C-BCH code [23], and they are used for the iBMA and Chien search algorithm for the BCH code.

5.0.3 Error Analysis

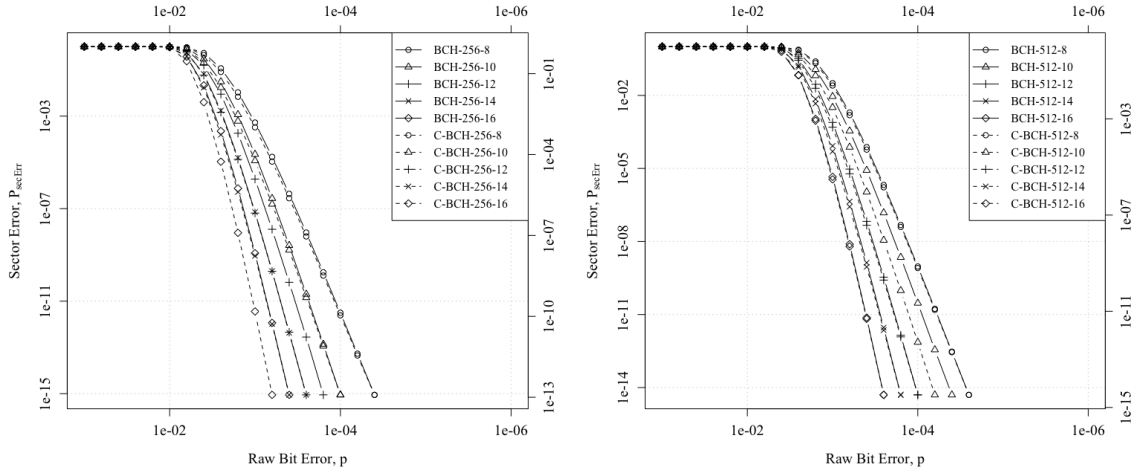
The error correction capability increases with n, but larger the n the higher the probability of random bit error. Based on the raw bit error probability p, parity bits and message code $cpar = 2.t.m + |m(x)|$, the sector with correctable error (P_{secErr}) might increase and is given as

$$P_{secErr} = 1 - \sum_{i=t+1}^{cpar} \binom{cpar}{i} \cdot p^i \cdot (1-p)^{(cpar)-i} \tag{5.6}$$

Whereas, the P_{secErr} for C-BCH code, having the parity bits as $ccpar = 2.t_c.m_c + |cm(x)|$, is revised and given as

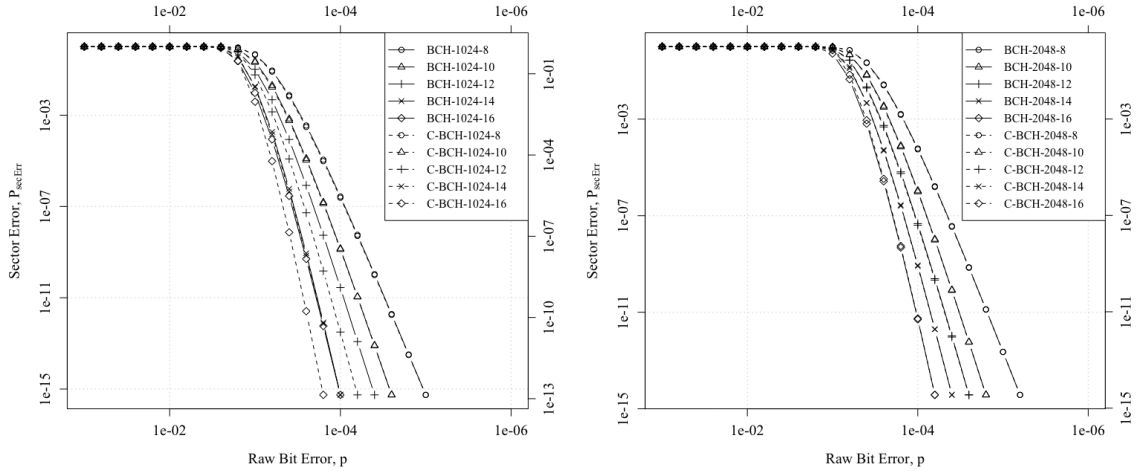
$$\begin{aligned} P_{secErr} = 1 - \sum_{i=t+1}^{cpar} \binom{cpar}{i} \cdot p^i \cdot (1-p)^{cpar-i} \\ \cdot \sum_{j=t_c+1}^{ccpar} \binom{ccpar}{j} \cdot p^j \cdot (1-p)^{ccpar-j} \end{aligned} \tag{5.7}$$

From Eq.5.7, we can see a slight gain in P_{secErr} because of the extra protection on the $res_{k,i}(x)$ by the C-BCH code. Fig. 5.3 illustrates the gain in P_{secErr} with the C-BCH code over BCH code for sector size 256 bytes with error correction capability as



(a) Block size, 256 bytes($m = 12$)

(b) Block size, 512 bytes($m = 13$)



(c) Block size, 1024 bytes($m = 14$)

(d) Block size, 2048 bytes($m = 15$)

Fig. 5.3: Raw bit error vs. Sector error

$t = 8, 10, 12, 14, 16$. We could also see that the P_{secErr} decreases as p decreases. Fig. 5.3 plots the bit error vs. sector error for both the BCH code and the C-BCH code. We could see that the proposed C-BCH code provides more gain than the BCH code because of the additional error protection over the parity bits.

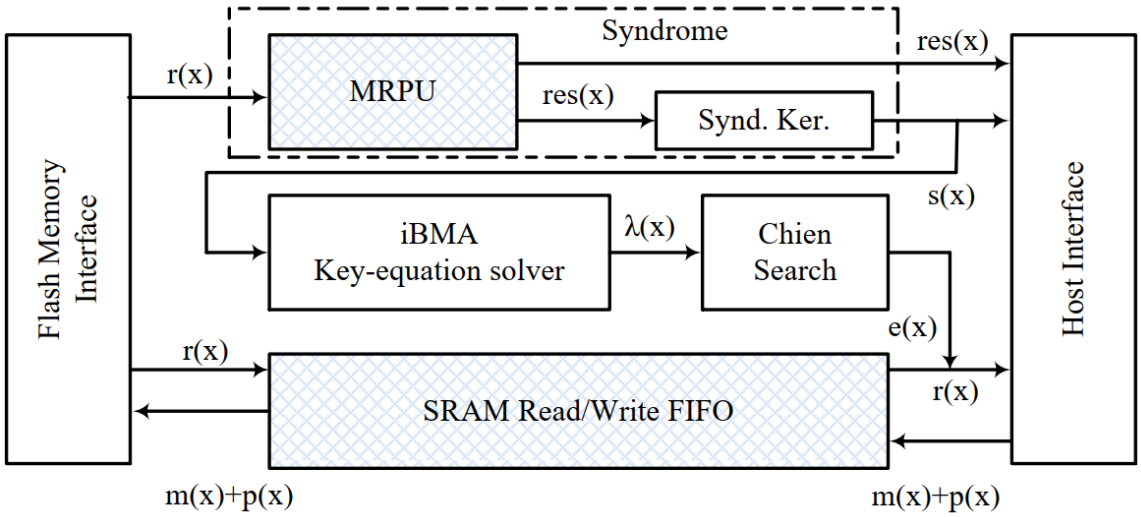


Fig. 5.4: Hybrid approach block diagram

5.1 Hybrid Implementation

In this section, we will discuss the hybrid approach to the C-BCH code and BCH code. Also, we discuss the practical issues encountered during the implementation of the same. Since the GPUs are ubiquitously used in modern SOC designs, we propose to use the GPUs in such a system for our decoders. Fig. 5.4 represents the block diagram for our proposed hybrid approach. However, we propose to use hardware design for timing critical paths in our method to provide high throughput when there are no errors encountered in the flash memory. This method seems to be a pragmatic approach to support multiple NAND flash devices with different ECC block sizes. Often, the length of the $m_k(x)$ is multiple of 4 bytes, and for typical NAND flash devices, these are 256, 512, 1024, and 2048 bytes. As a first step, we need to find the splitting field for the given code, and for a given $m_k(x)$ it could be expressed as $m = \lceil log_2(|m_k(x)| + 1) \rceil$. Now the n has the upper bound as $n = 2^m - 1$, and the code length $k = 2^m - 1 - 2 \cdot t \cdot m$. The $|m_k(x)|$ might not be equal to the k of the BCH code, so to compensate for the difference in the length we propose to add padded zeros, $p_{zero} = k - |m_k(x)|$, at the

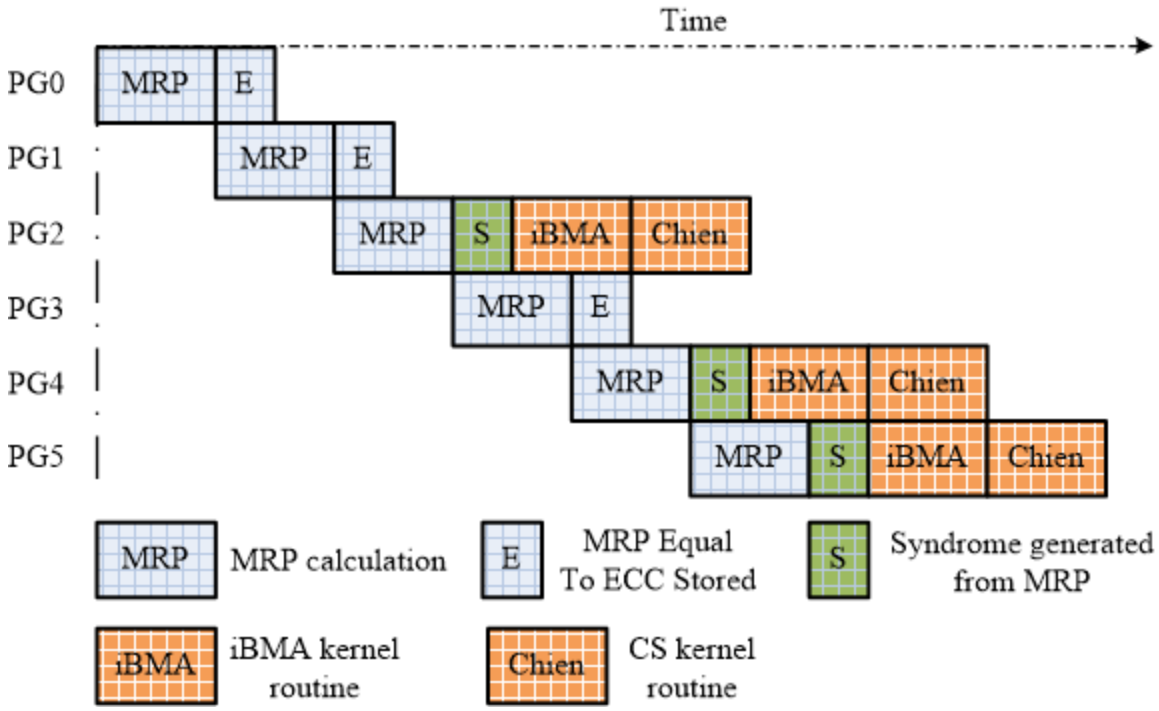


Fig. 5.5: Decoder Execution sequence

encoder and the decoder. Similar to the BCH code, the C-BCH code dimension m_c is given as $m_c = \lceil log_2(|cm(x)| + 1) \rceil$, where n_c has the upper bound $n_c = 2^{m_c} - 1$. Also, we use the same padded zero technique to make $cm(x)$ equal to k_c.

Fig. 5.5 depicts the systematic execution of the kernel routines. PG2, PG4, and PG5 represent pages with errors whereas PG0, PG1, and PG3 represent pages without error. When there are no errors, the latency incurred is the computation time consumed by MRPU systolic array as shown in Fig. 5.5. Also, we propose to compute the $eres_i(x)$, using MRPU arrays, for the next page in tandem with the execution of the GPU kernel routines to achieve better throughput.

5.1.1 Flash Data Layout

Fig. 5.6 represents the layout of the data and the ECC within a NAND flash memory device. There are two PLANEs of NAND flash memory array for this device.

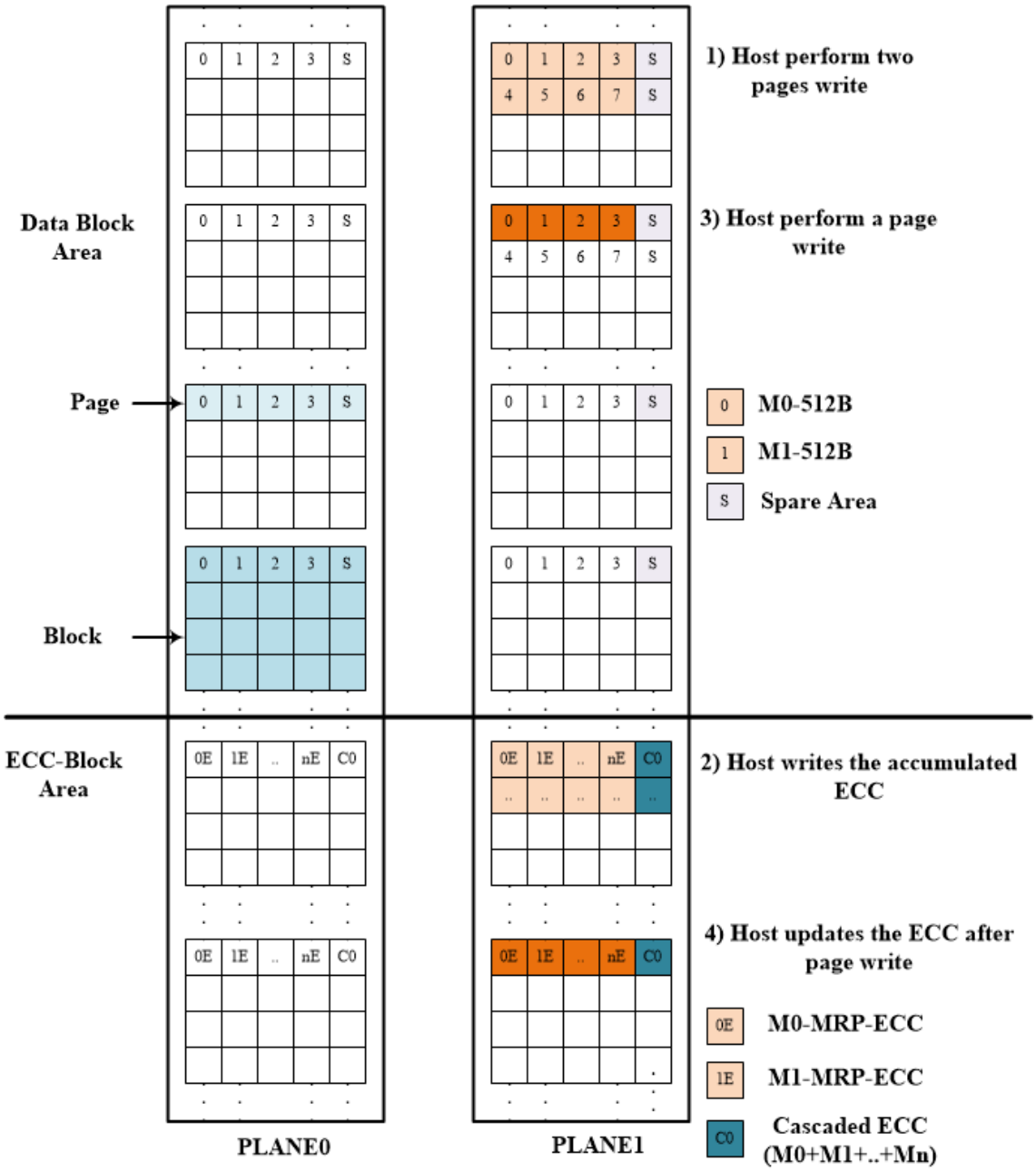


Fig. 5.6: Data block layout for cascaded ECC method

The area within the NAND flash is segregated into two major areas which are data block area (DBA) and ECC block area (EBA). In Fig. 5.6 each smallest square represent a message sector($m_k(x)$), and they are labelled as M0,M1,..,etc. A group of sectors forms a page, and a group of pages makes a block. In our implementation, we have used a NAND flash memory device with a page size of 2 KBytes (additional 64 bytes of the spare area), with a block size of 64 pages, and with a sector size of 512 bytes. The spare area in the DBA is assigned for bad block management and garbage collection software. The C-BCH code $cc(x)$ is stored in the ECC block area, and is used to protect the $res_{k,i}(x)$ within DBA. The additional area within the EBA is used to store the parity bits of the $cc(x)$. When the host performs a single page write, the EBA sector where the $res_{k,i}(x)$ is to be updated is read first and then written with the updated value. The ECC manager is responsible for managing the cache the $res_{k,i}$ during continuous writes and continuous reads. If the host performs multiple pages write, the $res_{k,i}(x)$ is accumulated in the ECC area with the RAM, and then a final write to the EBA is performed with the updated C-BCH code.

5.1.2 Hybrid Flow Chart

Fig.5.7 depicts the flow chart of our proposed hybrid method. Initially, the GPUs create a LUT memory for faster GF multiplication; this method has been proven to be faster on GPUs than threads spawning sub-kernel routines[23]. Next, a page read command is initiated to the flash memory interface. The syndrome residual unit (SRU) calculates the t residuals of the minimal polynomial while the data is written into the FIFO. If all the residuals are zero, then we conclude that there are no errors detected in the received vector, and the host shall transfer the data to the application layer. If there are non-zero residuals, then the host calls the Syndrome Calculation Kernel(SK) routine to calculate the syndrome and then calls the Key Equation Kernel(KEK) to form the

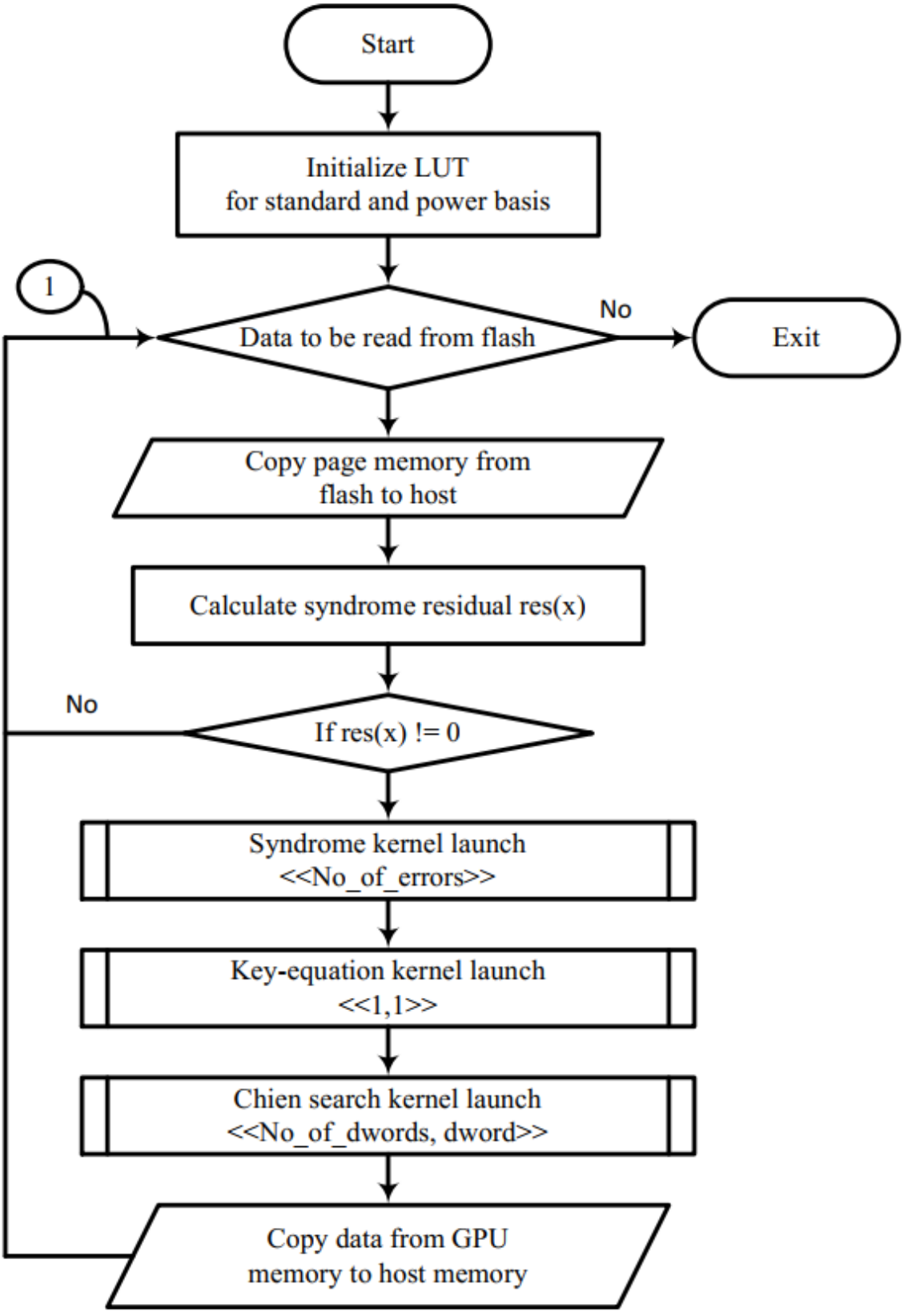


Fig. 5.7: Hybrid imp. flow chart

error locator polynomial $\Lambda(x)$. Once the $\Lambda(x)$ is formed, the Chien search Kernel(CSK) is executed for each bit locations. The final error vector is then added to the data in the FIFO to correct the bit errors and then copied to the host memory. Until all the desired data from the flash memory is read, we repeat the before mentioned steps (node 1 in Fig. 5.7).

5.1.3 VLSI Implementation

Fig. 5.8 depicts the hardware architecture for the MRPU array in a serial fashion. This hardware can be unfolded to process multiple bit streams in parallel [36]. This unit is used during the encoding process, and it is used during the syndrome generation process of the decoder. Besides, the MRPU is a programmable polynomial divisor for the received code vector $r(x)$. The programmable polynomial $g_i(x)$ in the Fig. 5.8 is programmed with the appropriate coefficient of $\phi_i(x)$. The remainder of the division is then compared with the previously stored content $res_i(k)$ to yield $eres_i(k)$. This unit is active during a page write, and a page read from the flash memory. During the write operation, the $res_{k,i}(x)$ is generated by programming the $res_i(x)$ as zeros which is given as $eres_i(x) = m_k(x)\, mod\, \phi_i(x) + 0 = res_{k,i}(x)$. While reading, the $eres_{k,i}(x)$ is generated by the same structure which is given as $eres_i(x) = r(x)\, mod\, \phi_i(x) + res_{k,i}(x)$. Since the MRPU array covers only a single sector within a page, a FIFO storage is recommended to store the entire $eres_i(x)$ contents for a complete page. The stored $eres_i(x)$ is then moved to the ECC area within the RAM. The ECC manager then performs a write or appropriate calls other GPU kernel routines depending on the write/read sequence and error/without error scenario.

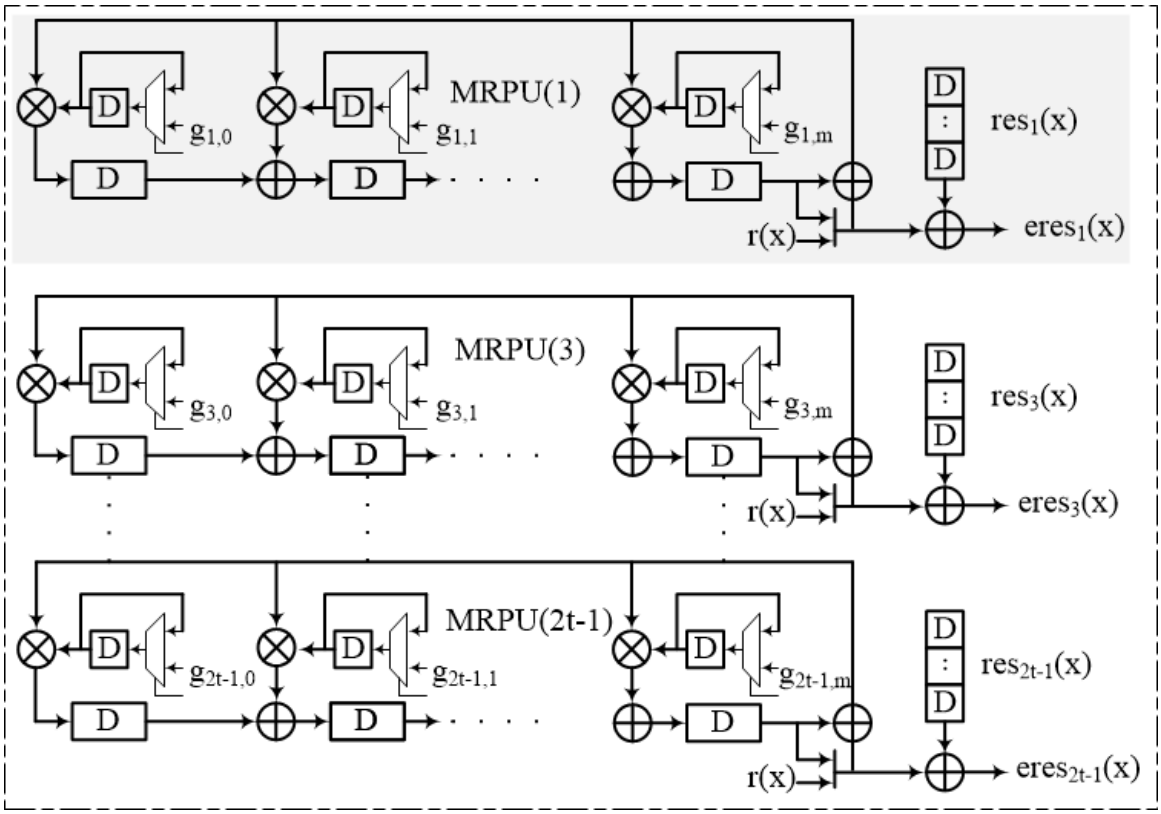


Fig. 5.8: Hardware design of MRPU array

5.1.4 GPU Implementation

Kernel routines are the fundamental sub routines, representing SIMD type of parallelism, executed by the GPU for our proposed decoder. For GF multiplication in the algorithm, the multiplicand and multiplier are converted to power basis by referring to a pre-initialized LUT in the global memory, so the multiplication is transformed into a simple XOR operation in power basis representation. After the multiplication is computed, a reverse transformation is performed by referring to another basis converter LUT in the global memory. The three key GPU kernel routines, in addition to the memory copy routines, used in our implementation are described in detail below.

5.1.4.1 Syndrome Generator

There are two types of syndrome kernel(SK) routine defined in our implementation. In the first method, the S_i is calculated from $eres_i(x)$ generated from the MRPU unit(Algorithm 5). This routine is used when an error is detected in the DBA. In the

Algorithm 5 Syndrome Kernel

1: **procedure** SYND KERNEL($eres_{k,i}, S_{k,i}$)
2: $synd \leftarrow 0$
3: **for** $j \leftarrow 0, deg(\phi_i(x)) - 1$ **do**
4: $synd \leftarrow synd + eres_{k,i,j} \cdot (\beta^i)^j$
5: **end for**
6: $atomicXor(S_{k,i}, synd)$ ▷ synchronize writes
7: **end procedure**

second method, the syndrome is generated in the conventional method(Algorithm 6). Since there are no dependencies on the syndromes, t parallel SK routines are launched within the GPU for this algorithm. The *atomicXor* operation is required to synchronize the value updated by the SK routines across multiple threads hence they are used in both algorithms. The second method is necessary to find syndromes for the blocks that

Algorithm 6 Syndrome Kernel Full

1: **procedure** SYND KERNEL FULL($Data, Dwpos, S_{k,i}$)
2: $synd \leftarrow 0$
3: **for** $j \leftarrow 0, 31$ **do**
4: $pos \leftarrow Dwpos * 32 + j$
5: $synd \leftarrow synd + Data[Dwpos * 32][j] \cdot (\beta^i)^{pos}$
6: **end for**
7: $atomicXor(S_{k,i}, synd)$ ▷ synchronize writes
8: **end procedure**

hold the ECC for the main data blocks.

5.1.4.2 Key-Eq Solver

Due to the iterative nature of the algorithm [12], the Key-Eq Solver kernel (KEK) is the only single thread routine in our implementation Algorithm7 represents the pseudo code for iBM implementation within the GPU kernel. There are other methods like simplified iBM (siBM) algorithm [36] proposed for the key-equation solver module, but experimental results have proven that siBM does not have significant improvement on performance of the GPU kernel routines.

5.1.4.3 Chien Search

The Chien search kernel (CSK) is the final routine to be executed in the decoder (Algorithm8). The primitive element $\alpha^{pos^{-1}}$ is evaluated as a root for $\Lambda(x)$. This CSK routine is an ideal candidate for GPU because the evaluation of each $\alpha^{pos^{-1}}$ in the equation $\Lambda(x)$ is independent.

Similar to the SK routine, the memory within the GPU device is shared between threads, so the *atomicXOR* operation is used to avoid writing overlap between different CSK routines. Once the error vector is formed, the error is masked with the data in memory to yield corrected data.

Algorithm 7 Key-equation Kernel

procedure KEQ EQ KERNEL(Λ, S)
  $\Lambda^{(0)} \leftarrow 1 + S_1 x$
  if $S_1 = 0$ **then**
    $d_p \leftarrow 1; \beta^{(1)} \leftarrow x^3; l_1 \leftarrow 0$
  else
    $d_p \leftarrow S_1; \beta^{(1)} \leftarrow x^2; l_1 \leftarrow 1$
  end if
  for $r \leftarrow 1, t-1$ **do**
    $d_r \leftarrow \sum_{i=1}^{t} \Lambda_i^{(r)} S_{2r-i+1}$
    $\Lambda^{(r)} \leftarrow d_p \Lambda^{(r-1)} + d_r \beta^{(r)}$
    if $d_r = 0 \, or \, r < l_r$ **then**
      $\beta^{(r+1)} \leftarrow x^2 \beta^{(r)}; l_{r+1} \leftarrow l_r; d_p \leftarrow d_p$
    else
      $\beta^{(r+1)} \leftarrow x^2 \Lambda^{(r)}; l_{r+1} \leftarrow l_r + 1; d_p \leftarrow d_r$
    end if
  end for
end procedure

Algorithm 8 Chien search Kernel

procedure CHIEN KERNEL(Λ, pos, err)
  $sum \leftarrow \sum_{j=1}^{deg(\Lambda)} \Lambda_j (\alpha^{pos^{-1}})^j$
  $atomicXor(err[pos], sum)$ ▷ Prevent overlap write
end procedure

5.2 Experimental Results

The proposed method is compared against conventional GPU[23] and hardware [19] architectures in this section. For a fair comparison, we compare against the VLSI area required for syndrome generation alone and not the complete decoder area. Table.5.1 lists the different parameters of the NAND flash memory used. The hardware implementation of the MRPU is synthesized for 28-nm technology, and it achieved an operational frequency of 1 GHz. In our experiments, we analyze the performance and the area consumed for different BCH code sizes. The setup used for GPU is shown in Table.5.2. We have used different sector sizes of 128, 256, 512 bytes which corresponds

Table 5.1: NAND Flash characteristic

Basic Operation	**Latency**	**Parameter**	**Size**
Page Read	50 µs	*Sector size*	128,256,512 Bytes
Page Write	200 µs	*Page size*	2048+64 Bytes
Erase cycle	2000 µs	*Block size*	128 K.Bytes

to the finite field dimension of $m = 11, 12, 13$ for our comparison. Also, we analyzed the results for different bit errors ($t = 2, .., 20$) for the before specified configurations.

Table 5.2: Experimental setup

	GPGPU	**CPU**
Platform	1152 GPU cores	8 Processor cores
Clock Freq.	1.033 GHz	3.7 GHz
Memory	GDDR5(2GB),6$Gbps$	DDR2(32GB),102.4$Gbps$

The critical path of the encoder depends on the degree of the polynomial of the generator. Our proposed method has the critical path dependent on the $GF(2^m)$ and not the number of bit errors as shown in Fig. 5.9. This is the reason why the MRPU design is able to meet the frequency constraints for bit error equal to 20.

5.2.1 GPU Analysis

Syndrome generation is the critical area where the proposed hybrid method provides the advantage over the GPU methods [23, 50]. Fig. 5.10 plots the syndrome computation time vs. different bit errors($t = 2, .., 20$) for different finite fields($m = 11, 12, 13$) and different architectures: GPU[23], Hardware[19], and Hybrid (proposed). The computation of the MRPU engine depends on the number of clock cycle required for a page read. Since all the $eres_i(x)$ that are necessary for key-equation solver are calculated in tandem, the latency only depends on the read cycles for flash memory. In the hybrid approach, the time consumed by the MRPU unit and the hardware architecture is the

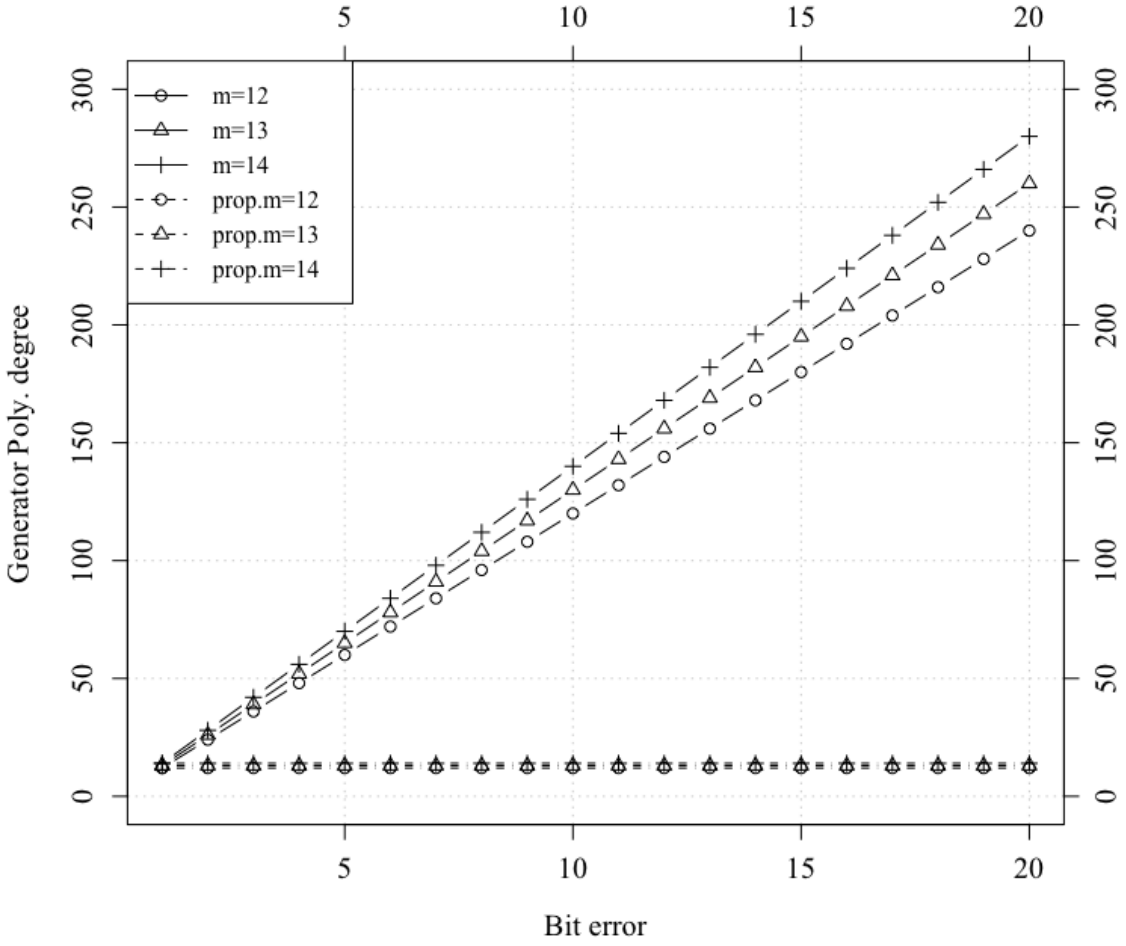


Fig. 5.9: Generator polynomial degree vs bit error

same, but there is a new kernel routine executed in the hybrid approach(Algorithm 5). The execution time for SK on GPUs depends on the load of the GPUs at the launch time. In our setup, the GPUs are used as display adapter and computational source, so the execution time recorded depends on the load on the GPUs at the specified sampling time which is an additional variable that should be noted in our results. When there are no errors in a page, the total time taken by our proposed decoder is less than 2µs which is less than the read latency of the flash.

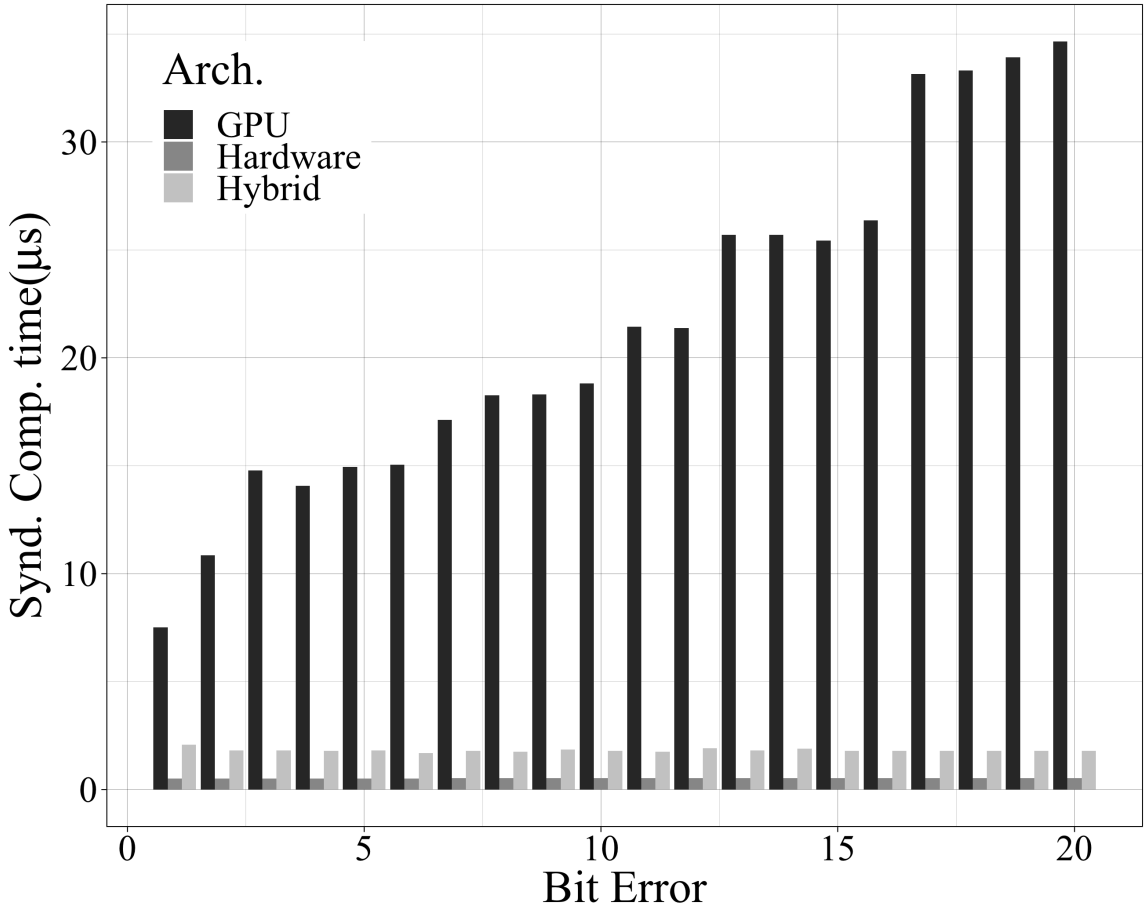


Fig. 5.10: Syndrome computation time for different arch.

5.2.2 VLSI Analysis

Table.5.3 compares the VLSI area required for different methods of syndrome generator ([19] and proposed). To support different finite fields and 20-bit error correction, hardware method [19] consumes 9103μm^2 whereas our proposed hybrid architecture consumes 4558 μm^2 thus saving 50% of the area. The GF multiplication is not required in our proposed method hence the savings in area. The final step is moved to the SK routine, so modulo arithmetic is alone required in our proposed hardware circuits. The comparison of total computation time, in case of an error, is compared for hardware, GPU and hybrid architecture in Fig. 5.11, and we can see that there is a minimal 25% performance improvement in the hybrid approach. The hybrid approach has performance improvement, when compared to GPU, because of the gain in cycles by MRPU implementation. The hardware implementation is the best performance, but the area required to cover a flexible code is significant, and that makes our hybrid approach a pragmatic solution.

5.3 Conclusion

In this chapter, we have proposed a hybrid approach to improve page read performances in both error and no-error scenarios. We have extended the idea proposed in [20, 23] by protecting the $res_{k,i}(x)$ using C-BCH code. The GPU implementation [23] is used to correct any errors within the C-BCH code and stored in the local cache for faster reference. When there are no errors in a page, our proposed method is comparable to hardware implementation because of the MRPU arrays. Also, the MRPU array resides on the *Euclidean* domain of $GF(2)$ polynomials thus making it very flexible and able support different finite field. When there are errors, the GPU kernels are employed to solve the errors, and our results have shown that there is a minimum 25% improvement

Table 5.3: Area comparison for different syndrome generators

	Setup	**Area for t (μm^2)**									
		2	***4***	***6***	***8***	***10***	***12***	***14***	***16***	***18***	***20***
m = *11*	[19]	241	479	727	1014	1189	1486	1801	2090	2218	2399
	Proposed	*365*	*733*	*1145*	*1512*	*1901*	*2285*	*2782*	*3178*	*3575*	*3971*
m = *12*	[19]	242	515	774	1098	1306	1639	1947	2266	2402	2610
	Proposed	*414*	*806*	*1214*	*1630*	*2020*	*2580*	*3010*	*3438*	*3868*	*4297*
m = *13*	[19]	333	774	1203	1678	2138	2594	3061	3514	3804	4095
	Proposed	*436*	*869*	*1310*	*1741*	*2195*	*2739*	*3193*	*3647*	*4103*	*4558*
m = *11,12,13*	[19]	816	1768	2704	3791	4633	5718	6809	7870	8424	***9103***
	Proposed	*436*	*869*	*1310*	*1741*	*2195*	*2739*	*3193*	*3647*	*4103*	***4558***

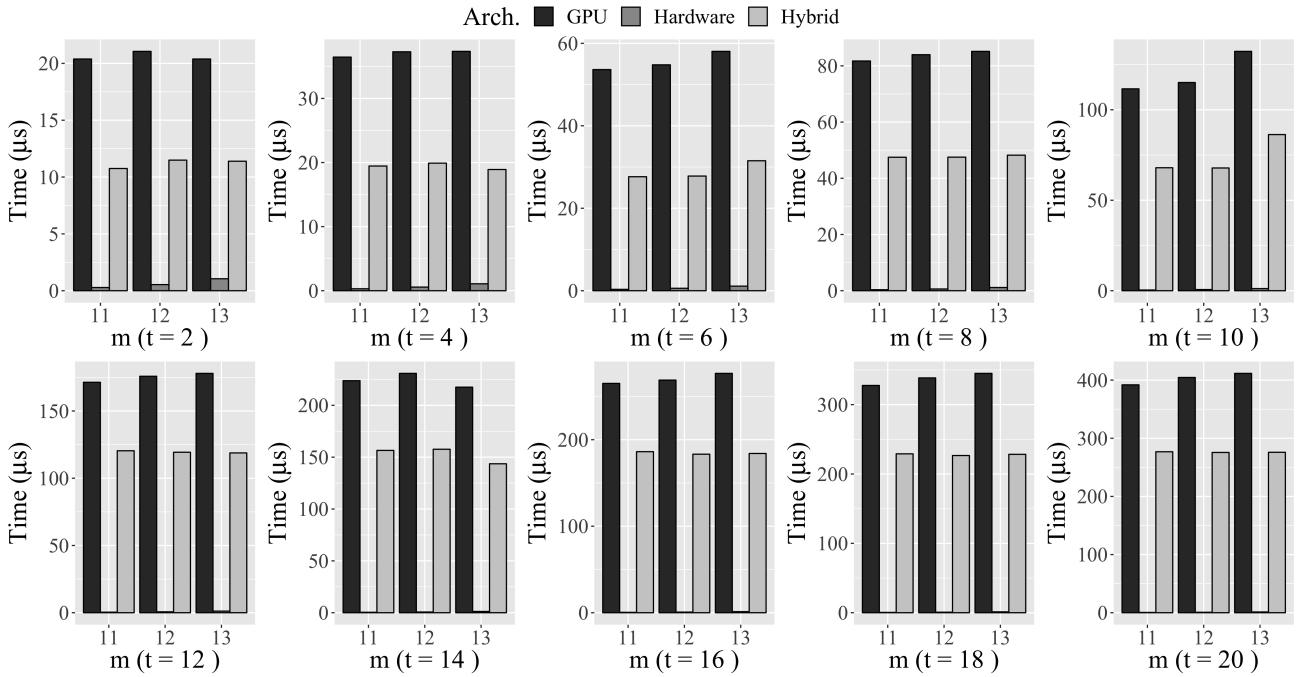


Fig. 5.11: Total computation time for different architecture for different sector size

in performance when compared with [23]. For future work, we propose extending the same algorithm to the Reed Solomon codes. Since the residual for each polynomial is received without error at the receiver, one can investigate the usage of Chinese Reminder Theorem for the decoder to make a soft decision instead of a hard decision.

Chapter 6

Conclusion

The intent of this thesis is to address the problems stated in Section. 1.2

- Long BCH codes have a long critical path for encoders
- Supporting different $GF(2^m)$ requires more VLSI area
- To have a high throughput, the solution should consume less than 100µs for a page read

We have addressed these problems using C-BCH code and hybrid implementation.

6.1 Long BCH code

The long BCH code is broken down using the minimal polynomial $\phi_i(x)$ and it's residuals,which is given by the following three equations

$$p_k(x) = \sum_{i=1}^{t} res_{k,2i-1}(x) \cdot \prod_{j=1}^{i-1} x^{deg(\phi_{2j-1}(x))} \tag{6.1}$$

$$cm(x) = \sum_{k=1}^{N_b} p_k(x) \cdot x^{(k-1).t.m} \tag{6.2}$$

$$cc(x) = cm(x).x^{deg(g_c(x))} + cm(x)\, mod\, g_c(x) \tag{6.3}$$

As shown in Figure 6.1, the proposed method has broken the critical path into sub blocks. This method makes it easy to scale to higher bit error rates.

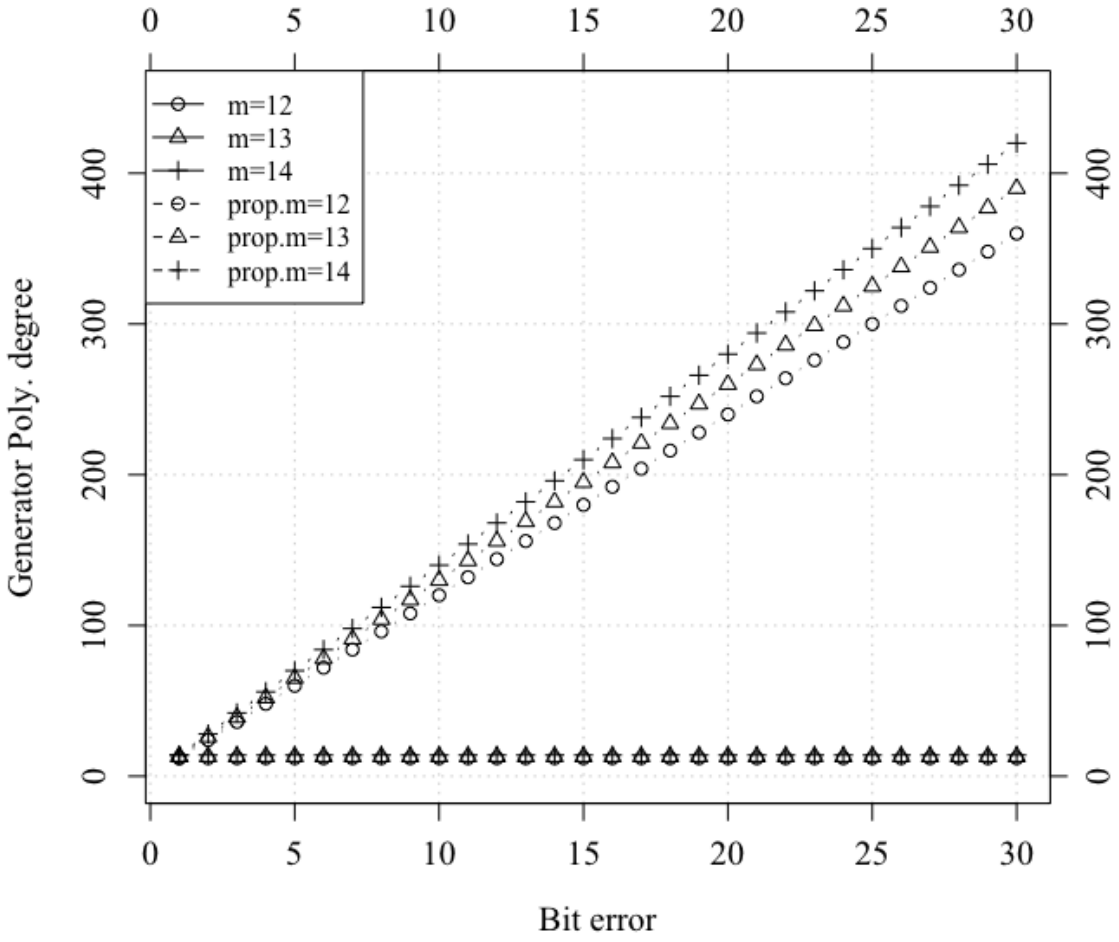


Fig. 6.1: Generator Polynomial degree vs bit errors (proposed)

6.1.1 Breaking Syndrome computation

Breaking the encoder helps in creating the parity bits in an easy fashion, and the same circuit could be used for syndrome generator. Fig. 6.2 shows the circuit that could be used for the encoder and partial syndrome generator. Since this circuit is programmable, the same circuit could be used for multiple sector sizes and multiple bit

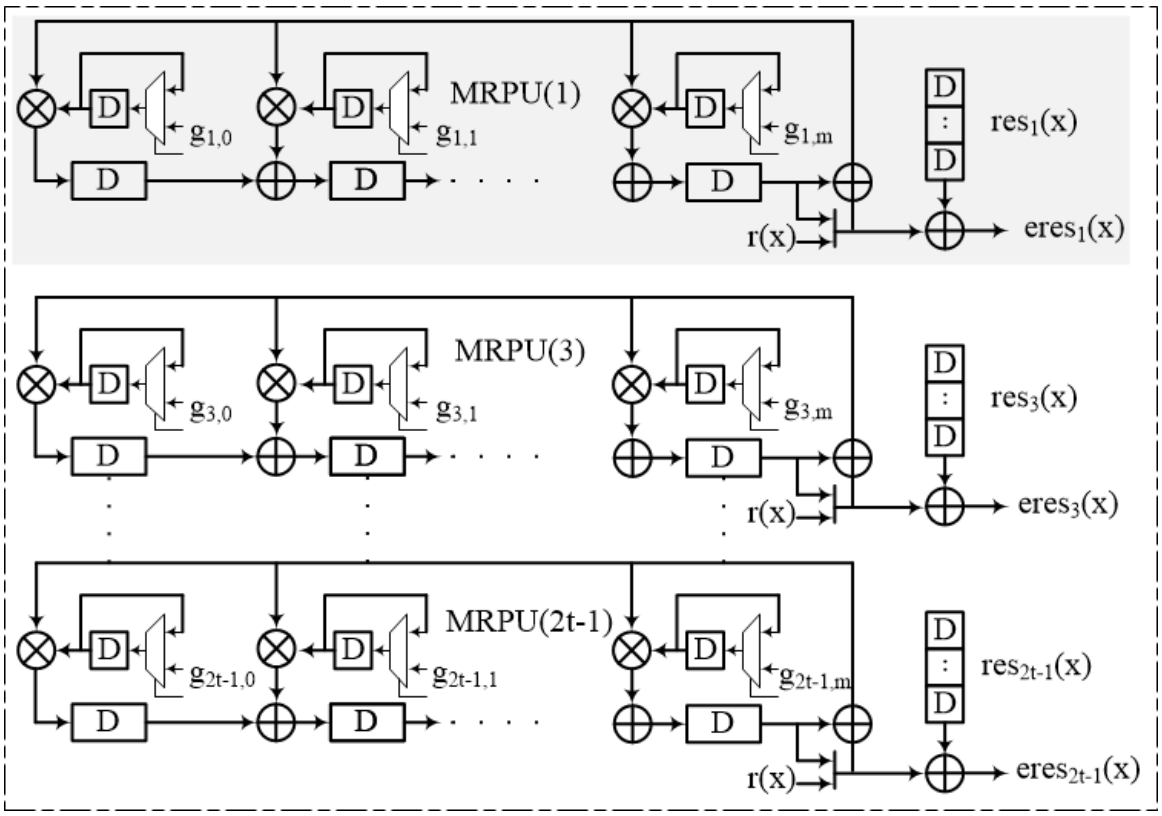


Fig. 6.2: Hardware design of MRPU array

errors. We recommend using a GF dimension of $m = 16$ for it could support SLC, MLC, and TLC NAND flash devices.

6.1.2 Hybrid approach

Using the GPUs as part of the hybrid solution provides high throughput for page read with no error. The syndrome generation is done in the hardware, and the GPUs are used for KES and CS algorithms. Most of the page read from the flash memory are without error, and the hardware solution for syndrome generator is the suitable solution. The time taken by the hardware to compute is less than two clock cycles, and it does not add any latency to the page read. For pages with error, the performance has 25% improvement over the conventional GPU implementation. Hence, this approach is

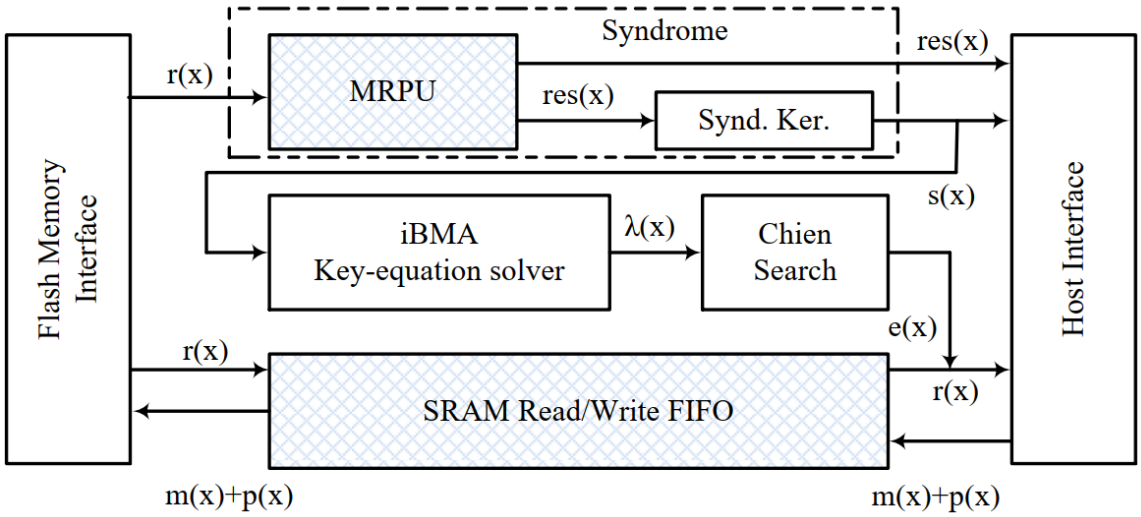


Fig. 6.3: Hybrid approach block diagram

a very pragmatic solution to address the long BCH code problem, and it could support multiple GF dimensions without compromising on the performance. Fig. 6.3 represents the block diagram for our proposed hybrid approach.

6.2 Future Work

We propose the following for future work: Since the residues are protected with a cascaded BCH code, we can investigate the CRT method to be used for the decode. Also, the same technique can be used for soft decision using Lagrange interpolation technique. Since our thesis has proven to be pragmatic for BCH code, we can port the same cascaded BCH technique to RS decoders and be able to support multiple burst errors, and this is more attractive for the wireless type of applications. The CRT method for C-BCH code can also be studied for RS codes. Another exciting area is to study the feasibility of soft decision algorithm, using the interpolation technique, proposed by [34] for GPU systems.

Bibliography

[1] Seunghwan Hyun, Hyokyung Bahn, and Kern Koh. LeCramFS: An efficient compressed file system for flash-based portable consumer devices. *IEEE Transactions on Consumer Electronics*, 53(2):481–488, 2007. ISSN 00983063. doi: 10.1109/TCE.2007.381719. 1, 11

[2] Guangxia Xu, Yanbing Liu, Xiaoqin Zhang, and Mingwei Lin. Garbage collection policy to improve durability for flash memory. *IEEE Transactions on Consumer Electronics*, 58(4):1232–1236, 2012. ISSN 00983063. doi: 10.1109/TCE.2012.6414990. 1, 4, 11

[3] Riqing Chen and Mingwei Lin. Energy-aware Buffer Management Scheme for NAND Flash-based Consumer Electronics. *IEEE Transactions on Consumer Electronics*, 61(4):484–490, 2015. doi: 10.1109/TCE.2015.7389803. 1, 11

[4] Hua Yan, Yong Huang, Xinzhi Zhou, and Yinjie Lei. An Efficient and Non-time-sensitive File-aware Garbage Collection Algorithm for NAND Flash-based Consumer Electronics. *IEEE Transactions on Consumer Electronics*, 65(1):73–79, 2018. ISSN 0098-3063. doi: 10.1109/TCE.2018.2885102. 1, 4, 11

[5] Lingyan Fan, Jian-jun Luo, Yuehui Mei, Troy Rutt, and Zuliang Wang. Independent Module Based Solid State Drives. *IEEE Transactions on Consumer Electronics*, 64(3):328–333, 2018. doi: 10.1109/TCE.2018.2859624. 1

[6] Rino Micheloni, Alessia Marelli, and Luca Crippa. *Inside NAND flash memories*. Springer, New York, 2010. ISBN 9789048194308. doi: 10.1007/978-90-481-9431-5. 1

[7] Neal Mielke, Todd Marquart, Ning Wu, Jeff Kessenich, Hanmant Belgal, Eric Schares, Falgun Trivedi, Evan Goodness, and Leland R. Nevill. Bit error rate in NAND flash memories. In *IEEE International Reliability Physics Symposium Proceedings*, pages 9–19, 2008. ISBN 9781424420506. doi: 10.1109/RELPHY.2008.4558857. 2

[8] Juhee Kim and Yong Jee. Hamming product code with iterative process for NAND flash memory controller. In *ICCTD 2010 - 2010 2nd International Conference on Computer Technology and Development, Proceedings*, pages 611–615, 2010. ISBN 9781424488438. doi: 10.1109/ICCTD.2010.5645968. 2, 11

[9] Liu Wei, Rho Junrye, and Sung Wonyong. Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories. In *2006 IEEE*

Workshop on Signal Processing Systems Design and Implementation, SIPS, pages 303–308, 2006. ISBN 1424403820. doi: 10.1109/SIPS.2006.352599. 2, 11, 16, 21

[10] Bainan Chen, Xinmiao Zhang, and Zhongfeng Wang. Error correction for multi-level NAND flash memory using reed-solomon codes. In *IEEE Workshop on Signal Processing Systems, SiPS: Design and Implementation*, pages 94–99, 2008. ISBN 9781424429240. doi: 10.1109/SIPS.2008.4671744. 2, 11

[11] Kai Zhao, Wenzhe Zhao, Hongbin Sun, Tong Zhang, Xiaodong Zhang, and Nanning Zheng. LDPC-in-SSD: Making Advanced Error Correction Codes Work Effectively in Solid State Drives. In *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, 2013. ISBN 978-1-931971-99-7. doi: 10.1007/s00104-014-2893-9. 2, 11

[12] S.S. Lin Costell and D.J. Costello. *Error Control Coding - Fundamentals and Applications*. Prentice-Hall, Englewood Cliffs, NJ, USA, 2nd edition, 2004. ISBN 0163-6804. 3, 13, 14, 16, 31, 32, 72, 85

[13] Yi Wang, Zhiwei Qin, Renhai Chen, Zili Shao, Qixin Wang, Shuai Li, and Laurence T. Yang. A Real-Time Flash Translation Layer for NAND Flash Memory Storage Systems. *IEEE Transactions on Multi-Scale Computing Systems*, 2(1): 17–29, 2016. ISSN 23327766. doi: 10.1109/TMSCS.2016.2516015. 4

[14] NVIDIA. Cuda C Programming Guide, 2015. ISSN 15284972. URL https://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf. 5, 22, 45

[15] Chanha Kim, Chanik Park, Sungjoo Yoo, and Sunggu Lee. Extending Lifetime of Flash Memory Using Strong Error Correction Coding. *IEEE Transactions on Consumer Electronics*, 61(2):206–214, 2015. doi: 10.1109/TCE.2015.7150595. 7, 17, 18, 19

[16] Yu-peng Hu, Nong Xiao, and Xiao-fan Liu. An Elastic Error Correction Code Technique for NAND Flash-based Consumer Electronic Devices. *IEEE Transactions on Consumer Electronics*, 59:1–8, 2013. doi: 10.1109/TCE.2013.6490234. 7, 17, 19

[17] Hakyong Lee, Sanghyuk Jung, Student Member, and Yong Ho Song. PCRAM-assisted ECC Management for Enhanced Data Reliability in Flash Storage Systems. *IEEE Transactions on Consumer Electronics*, 58:849–856, 2012. doi: 10.1109/TCE.2012.6311327. 7, 17, 23

[18] Yuan-Hao Chang and Tei-Wei Kuo. A reliable MTD design for MLC flash-memory storage systems. In *Proceedings of the tenth ACM international conference on Embedded software - EMSOFT '10*, page 179, 2010. ISBN 9781605589046. doi: 10.1145/1879021.1879045. 7

[19] Byeonggil Park, Jongsun Park, and Youngjoo Lee. Area-Optimized Fully-Flexible BCH Decoder for Multiple GF Dimensions. *IEEE Access*, 6:14498–14509, 2018. ISSN 21693536. doi: 10.1109/ACCESS.2018.2815640. 7, 22, 86, 87, 90, 91

[20] Arul K. Subbiah and Tokunbo Ogunfunmi. Area-effcient re-encoding scheme for NAND Flash Memory with multimode BCH Error correction. In *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1–5, 2018. ISBN 9781538648810. doi: 10.1109/ISCAS.2018.8351503. 8, 9, 10, 38, 41, 42, 90

[21] Arul K Subbiah and Tokunbo Ogunfunmi. A Flexible Hybrid BCH Decoder for modern NAND Flash Memories using GPGPUs. *Micromachines*, 10(2):1–15, 2019. doi: 10.3390/mi10060365. 8, 9, 10

[22] Arul.K. Subbiah and S. Viswanath. Evolving throughput driven architecture for error correction in NAND memory. In *DesignCon 2010*, volume 1, 2010. ISBN 9781617385469. 9, 10

[23] Arul K. Subbiah and Tokunbo Ogunfunmi. Efficient implementation of BCH decoders on GPU for flash memory devices using iBMA. *2016 IEEE International Conference on Consumer Electronics (ICCE)*, pages 275–278, 2016. ISSN 2158-4001. doi: 10.1109/ICCE.2016.7430612. 9, 10, 22, 49, 75, 80, 86, 87, 90, 93

[24] Arul K. Subbiah and Tokunbo Ogunfunmi. Memory-efficient Error Correction Scheme for Flash Memories using GPU. In *2018 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 118–122. IEEE, 2018. ISBN 9781538663189. 9, 10, 72

[25] Arul K Subbiah and Tokunbo Ogunfunmi. Three-bit fast error corrector for BCH codes on GPUs. In *2019 IEEE International Conference on Consumer Electronics (ICCE)*, pages 1–4. IEEE, 2019. doi: 10.1109/ICCE.2019.8661993. 9, 10

[26] Arul K Subbiah. Fast BCH syndrome generator using parallel polynomial division algorithm for GPGPUs. *IEEE industial electronics and applications*, (2):2–5, 2019. 9, 10

[27] Arul K Subbiah and Tokunbo Ogunfunmi. Cascaded ECC for NAND Flash-based electronic devices. *IEEE Transactions on Circuits and Systems I: Regular Papers (under review)*, pages 1–8, 2019. 9, 10

[28] Arul K. Subbiah and Tokunbo Ogunfunmi. Frequency-domain based Reed Solomon decoders for GPU. In *2017 IEEE International Conference on Consumer Electronics (ICCE)*, pages 352–354. IEEE, 2017. ISBN 9781509055449. doi: 10.1109/ICCE.2017.7889352. 10, 11, 22

[29] Daesung Kim, Krishna R. Narayanan, and Jeongseok Ha. Symmetric block-wise concatenated BCH codes for NAND flash memories. *IEEE Transactions on Communications*, 66(10):4365–4380, 2018. ISSN 15580857. doi: 10.1109/TCOMM.2018.2839606. 11

[30] Todd K. Moon. *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley-Interscience, Hoboken, New Jersey, 2005. ISBN 0471648000. doi: 10.1002/0471739219. 11

[31] Youngjoo Lee, Hoyoung Yoo, Injae Yoo, and In Cheol Park. High-throughput and low-complexity BCH decoding architecture for solid-state drives. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(5):1183–1187, 2014. ISSN 10638210. doi: 10.1109/TVLSI.2013.2264687. 15, 16, 21, 42

[32] Byeonggil Park, Seungyong An, Jongsun Park, and Youngjoo Lee. Novel folded-KES architecture for high-speed and area-efficient BCH decoders. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 64(5):535–539, 2017. ISSN 15497747. doi: 10.1109/TCSII.2016.2596777. 15, 21

[33] Yi Min Lin, Hsie Chia Chang, and Chen Yi Lee. Improved high code-rate soft BCH decoder architectures with one extra error compensation. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(11):2160–2164, 2013. ISSN 10638210. doi: 10.1109/TVLSI.2012.2227847. 15, 35

[34] Xinmiao Zhang. An efficient interpolation-based chase BCH decoder. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(4):212–216, 2013. ISSN 15497747. doi: 10.1109/TCSII.2013.2251941. 15, 21, 97

[35] Chia Hsiang Yang, Ting Ying Huang, Mao Ruei Li, and Yeong Luh Ueng. A 5.4 uw soft-decision bch decoder for wireless body area networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(9):2721–2729, 2014. ISSN 15498328. doi: 10.1109/TCSI.2014.2312478. 15

[36] Xinmiao Zhang. *VLSI Architectures for Modern Error-Correcting codes*. CRC Press, Boca Raton, FL, USA, 2nd edition, 2016. 16, 21, 25, 28, 31, 32, 42, 49, 82, 85

[37] Junho Cho and Wonyong Sung. Efficient software-based encoding and decoding of BCH codes. *IEEE Transactions on Computers*, 58(7):878–889, 2009. ISSN 00189340. doi: 10.1109/TC.2009.27. 17, 19

[38] Xinmiao Zhang and Keshab K Parhi. High-Speed Architectures for Parallel Long BCH Encoders. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 13(7):872–877, 2005. 19

[39] Hao Chen. CRT-based high-speed parallel architecture for long BCH encoding. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 56(8):684–686, 2009. ISSN 15497747. doi: 10.1109/TCSII.2009.2024247. 20

[40] Hoyoung Tang, Gihoon Jung, and Jongsun Park. A hybrid multimode BCH encoder architecture for area efficient re-encoding approach. In *Proceedings - IEEE*

International Symposium on Circuits and Systems, volume 2015-July, pages 1997–2000, 2015. ISBN 9781479983919. doi: 10.1109/ISCAS.2015.7169067. 20, 40, 41, 42

[41] Hoyoung Yoo, Jaehwan Jung, Jihyuck Jo, and In Cheol Park. Area-efficient multimode encoding architecture for long BCH codes. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 60(12):872–876, 2013. ISSN 15497747. doi: 10.1109/TCSII.2013.2281941. 20, 40, 41, 42

[42] Guiqiang Dong, Ningde Xie, and Tong Zhang. On the use of soft-decision error-correction codes in nand flash memory. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 58(2):429–439, 2011. ISSN 15498328. doi: 10.1109/TCSI.2010.2071990. 21

[43] Chia Hsiang Yang, Ting Ying Huang, Mao Ruei Li, and Yeong Luh Ueng. A 5.4 uw soft-decision bch decoder for wireless body area networks. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 61(9):2721–2729, 2014. ISSN 15498328. doi: 10.1109/TCSI.2014.2312478. 21

[44] Xinmiao Zhang and Zhongfeng Wang. A low-complexity three-error-correcting BCH decoder for optical transport network. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 59(10):663–667, 2012. ISSN 15497747. doi: 10.1109/TCSII.2012.2208678. 21

[45] Xinmiao Zhang and Micheal O'Sullivan. Ultra-Compressed Three-Error-Correcting BCH Decoder. *2018 IEEE International Symposium on Circuits and Systems (ISCAS)*, (x):1–5, 2018. doi: 10.1109/ISCAS.2018.8350969. URL https://ieeexplore.ieee.org/document/8350969/. 21

[46] Jamro Ernest. *THE DESIGN OF A VHDL BASED SYNTHESIS TOOL FOR BCH CODECS*. PhD thesis, University of Huddersfield, 1997. 21

[47] Hoyoung Yoo, Youngjoo Lee, and In Cheol Park. Low-Power Parallel Chien Search Architecture Using a Two-Step Approach. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 63(3):269–273, 2016. ISSN 15497747. doi: 10.1109/TCSII.2015.2482958. 21

[48] Te Hsuan Chen, Yu Ying Hsiao, Yu Tsao Hsing, and Cheng Wen Wu. An adaptive-rate error correction scheme for NAND flash memory. In *Proceedings of the IEEE VLSI Test Symposium*, pages 53–58, 2009. ISBN 9780769535982. doi: 10.1109/VTS.2009.24. 21

[49] Hoyoung Yoo, Youngjoo Lee, and In Cheol Park. 7.3 Gb/s universal BCH encoder and decoder for SSD controllers. In *Proceedings of the Asia and South Pacific Design Automation Conference, ASP-DAC*, pages 37–38, 2014. ISBN 9781479928163. doi: 10.1109/ASPDAC.2014.6742862. 21

[50] Xiaoxia Qi, Xiao Ma, Dou Li, and Yuping Zhao. Implementation of accelerated BCH decoders on GPU. In *2013 International Conference on Wireless Communications and Signal Processing (WCSP)*, pages 1–6, 10 2013. ISBN VO -. doi: 10.1109/WCSP.2013.6677084. 22, 64, 87

[51] Matthew L. Curry, Anthony Skjellum, H. Lee Ward, and Ron Brightwell. Arbitrary dimension reed-solomon coding and decoding for extended RAID on GPUs. In *Proceedings of the 2008 3rd Petascale Data Storage Workshop, PDSW 2008*, pages 1–3, 2008. ISBN 9781424442089. doi: 10.1109/PDSW.2008.4811887. 22

[52] Yue Zhao and Francis C.M. Lau. Implementation of decoders for LDPC block codes and LDPC convolutional codes based on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):663–672, 2014. ISSN 10459219. doi: 10.1109/TPDS.2013.52. 22

[53] Huahai Liu, Pan Wang, Kewen Wang, Xun Cai, Liang Zeng, and Sikun Li. Scalable multi-GPU decoupled parallel rendering approach in shared memory architecture. In *Proceedings - 2011 International Conference on Virtual Reality and Visualization, ICVRV 2011*, 2011. ISBN 9780769546025. doi: 10.1109/ICVRV.2011.46. 44