micromachines

*Article*

# A Flexible Hybrid BCH Decoder for Modern NAND Flash Memories Using General Purpose Graphical Processing Units (GPGPUs)

**Arul Subbiah * and Tokunbo Ogunfunmi**

Department of Electrical Engineering, Santa Clara University, 500 El Camino Real, Santa Clara, CA 95053, USA; togunfunmi@scu.edu

* Correspondence: asubbiah@scu.edu; Tel.: +1-408-368-5099

Received: 20 April 2019; Accepted: 23 May 2019; Published: 31 May 2019

**Abstract:** Bose–Chaudhuri–Hocquenghem (BCH) codes are broadly used to correct errors in flash memory systems and digital communications. These codes are cyclic block codes and have their arithmetic fixed over the splitting field of their generator polynomial. There are many solutions proposed using CPUs, hardware, and Graphical Processing Units (GPUs) for the BCH decoders. The performance of these BCH decoders is of ultimate importance for systems involving flash memory. However, it is essential to have a flexible solution to correct multiple bit errors over the different finite fields (GF($2^m$)). In this paper, we propose a pragmatic approach to decode BCH codes over the different finite fields using hardware circuits and GPUs in tandem. We propose to employ hardware design for a modified syndrome generator and GPUs for a key-equation solver and an error corrector. Using the above partition, we have shown the ability to support multiple bit errors across different BCH block codes without compromising on the performance. Furthermore, the proposed method to generate modified syndrome has zero latency for scenarios where there are no errors. When there is an error detected, the GPUs are deployed to correct the errors using the iBM and Chien search algorithm. The results have shown that using the modified syndrome approach, we can support different multiple finite fields with high throughput.

**Keywords:** BCH; decoder; iBM; GPU; hybrid; flash memory; Galois field; CUDA

## 1. Introduction

NAND flash memories are widely used in many electronic devices. These devices face reliability issues because of the densely-populated memory cells [1]. In fact, the 3D method used to manufacture flash memories, discussed in detail by Spinelli et al. [2], enforces the necessity to have high throughput error correction techniques. Bose–Chaudhuri–Hocquenghem (BCH) codes [3] are the most common error correction mechanisms for flash memory devices and other digital communications like optical networks. The increasing efficiency and throughput of the flash memory systems have drawn researchers to provide highly-efficient BCH decoders. The three major categories of the BCH decoders proposed are Central Processing Units (CPUs), hardware circuits, and Graphical Processing Units (GPUs). Cho proposed an efficient CPU-based implementation in [4], and Poolakkaprambil discussed multi-bit error using Hamming, BCH, and Low-Density Parity Check (LDPC) codes in [5]. Later, Lee et al. proposed a high throughput hardware architecture in [6]. Moreover, Zhang discussed different hardware implementation techniques in [7]. Qi et al. [8] proposed a GPU-based BCH decoder; later, we proposed an efficient algorithm for BCH decoders using GPUs in [9]. In addition to the requirement of high throughput, modern BCH decoders are required to support multiple bit error correction across various block sizes, which is the focus of this paper. Technology scaling has rendered the ability to integrate multiple GPUs within a System On Chip (SOC), which has enabled researchers to use GPU for non-graphical applications. In fact, the term General Purpose Graphical Processing Unit (GPGPU) refers to the application of GPU for nongraphical applications. We use the term GPU instead of GPGPU since these terms are interchangeable in practice. Streaming Multiprocessors (SMs) are the building blocks of these GPUs, which has multiple CPUs within them. Each of the instantiated SMs is capable of handling multiple threads, which are scheduled by a warp scheduler. Therefore, we need an exclusive compiler like the Computer Unified Device Architecture (CUDA) C [10] software to program these SMs. The CUDA software creates the necessary grid of kernel routines, which in turn create the same instruction that operates on a different data path; this technique is referred to as the single instruction multiple data (SIMD) stream. The kernel subroutines are executed across multiple cores and in a multiple thread fashion. The GPU-based BCH decoders [9] are flexible, and they can support multiple BCH block sizes.

We have organized this paper as follows: Section 2 discusses the background and previous works. Section 3 describes our proposed hybrid method using GPUs and hardware design. Section 4 presents the results observed, and we conclude in Section 5.

## 2. Background

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 the splitting field, $GF(2^m)$. Let $\phi_i(x)$ be the minimal polynomial of an arbitrary element $\beta^i$, then the generator polynomial for BCH code with $t$ error correction capability is given by the following equation:

$$g(x) = LCM(\phi_1(x), \phi_2(x), ..., \phi_{2t}(x)) \tag{1}$$

Narrow sense BCH codes use primitive element $\alpha^i$ for the minimal polynomial with $i$ starting from one. For simplicity, the narrow sense BCH code decoder is discussed and implemented in this paper, and it could be easily extended for other general BCH codes [3]. The parity bits are then generated using the equation $p(x) = m(x)\, mod\, g(x)$, and these parity bits are concatenated to form the message polynomial $m(x)$. This concatenation is given as:

$$c(x) = m(x) \cdot x^{deg(g(x))} + m(x)\, mod\, g(x) \tag{2}$$

The generated parity bits are then stored in the spare area allocated in the page within the flash memory device. In general, the hard decision BCH decoder has three steps in the decoding process: syndrome generation, key-equation solver, and an error locator.

### 2.1. Encoder

The main issue when using large BCH codes, i.e., $t$ greater than 30, is the fan-out issue created by implementing the Linear Feedback Shift Register (LFSR) method of the generator polynomial. Parhi has addressed this fan-out issue by breaking down the LFSR register into multiple cascaded LFSRs by realizing the circuit in the Z-domain [11]. Hao addressed the same issue by using the Chinese Remainder Theorem (CRT) method [12], but this method requires more computation on the encoder and is applicable for encoders that have $t$ higher than 100. Later, Tang et al. proposed a hybrid approach for long BCH encoders that is area efficient [13]. The authors of this paper had proposed an area efficient method by sharing the hardware between the encoder and syndrome generator [14].

### 2.2. Decoder

The BCH decoders can be categorized as hard decision and soft decision decoders [15,16]. These decoders'realization can be broadly classified into three categories: Central Processing Unit (CPU) [4], Very Large Scale implementation (VLSI) [17], and GPU implementation [8]. Various hardware implementations of BCH decoders were discussed by Zhang [7]. BCH decoders can be categorized by the place of the decoders. The decoders can be either located on-chip within memory device [18] or outside the memory device [19]. The focus of this paper is on the decoder being outside the memory device.

The syndrome generator is the first step of the BCH decoding process [6,20]. The syndromes $S_i$ of the received vector $r(x)$ are given as:

$$S_i = r(\alpha^i) \tag{3}$$

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 $\alpha^1, \alpha^2, ..., \alpha^{2t}$. 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 $\phi_i(x)$. We have discussed an alternate approach to share the syndrome generator and encoders in [14]; Figure 1 depicts the area sharing between the encoder and the syndrome generator presented in [14]. This method requires separate error protection to the parity bits, and one proposal is to use a Single-Level Cell (SLC) for the parity bits to reduce error probability.

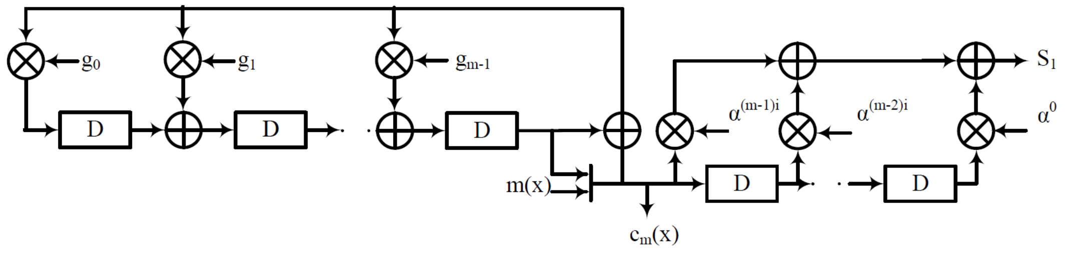


**Figure 1.** Area-efficient syndrome generator.

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

$$\Lambda(x) = \sum_{i=0}^{t} \Lambda_i x^i = (1 - X_1 x)(1 - X_2 x)...(1 - X_t x) \tag{4}$$

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

$$S(x).\Lambda(x) = \Omega(x)\, mod\, 2t \tag{5}$$

shows the relationship between the error locator polynomial and the error evaluator polynomial; moreover, Newton identities [3] show the relation between the error locator polynomial $\Lambda(x)$ and the syndromes $S_i$. There have been many algorithms like Berlekamp–Massey (BM), Peterson, and others proposed to solve the key equation [3,7], but the inversion-less BM (iBM) algorithm is predominantly used in high throughput architectures [6,21]. Park et al. [22] proposed a novel folded method to reduce the area in the hardware architecture, but the proposed method takes more clock cycles and is proportional to the folding factor. For the final step, the Chien Search (CS) algorithm is used to locate the error position from the error locator polynomial equation. Yoo et al. proposed a low power and high throughput parallel CS algorithm in [23].

## 2.3. Motivation

This paper intends to propose a solution that can address two configurable parameters of a BCH decoder. First, the solution should be scalable across different $GF$ fields, i.e., it should be able to support different $GF$ field extensions ($GF(2^m)$). Second, the solution should be able to scale across different bit errors $t$. Different configurable BCH decoder solutions have been proposed [20,24], but they lack support for both configurable parameters of the BCH decoders. Inspired by the attempt to solve BCH decoders for multiple $GF$ dimensions in [20], we propose an alternate hybrid approach to have a flexible solution. In [20], a hardware solution was proposed to support multiple BCH codes; however, the circuit area increases in order to support multiple $GF$ dimensions because of the dual-mBCH decoders' requirement. We have proposed a method to share the hardware logic between the BCH encoder and BCH syndrome generator by modifying the encoding method in [14]. In this paper, we extend our previous work by using a programmable modified syndrome generator algorithm and GPU to have a decoder that works with multiple $GF$ dimensions.

## 3. Hybrid Method

We propose a high throughput system that can correct $t$ bit errors over different BCH codes $(n, k, t)$, i.e., error correction over different finite field dimensions, using hardware design and GPU kernel routines. Figure 2 depicts the architectural block diagram for our proposed hybrid method. The flash memory interface is a physical interface to a flash memory device, and the host interface is a standard bus interface, which communicates with the GPU. The GPU could either reside inside the host interface (system on chip) or external to the host system. It is important to note that the GPU system in the system is used for dual purposes, i.e., for the graphical display and error correction. Furthermore, in our proposed method, the GPUs are only deployed when there is an error detected in the page, and the GPUs are not used for pages without error. It is assumed that the host system exercises a memory copy routine to transfer data whenever there is an interaction between the host system and the GPU system. The syndrome generation, proposed in [14], is split into two modules: the Syndrome Residual Unit (SRU) and the syndrome kernel. Then, we propose to implement the modules SRU and the FIFO (shaded area) in hardware and to use GPU kernel routines for the modules' syndrome calculator, key-equation solver, and an error corrector.

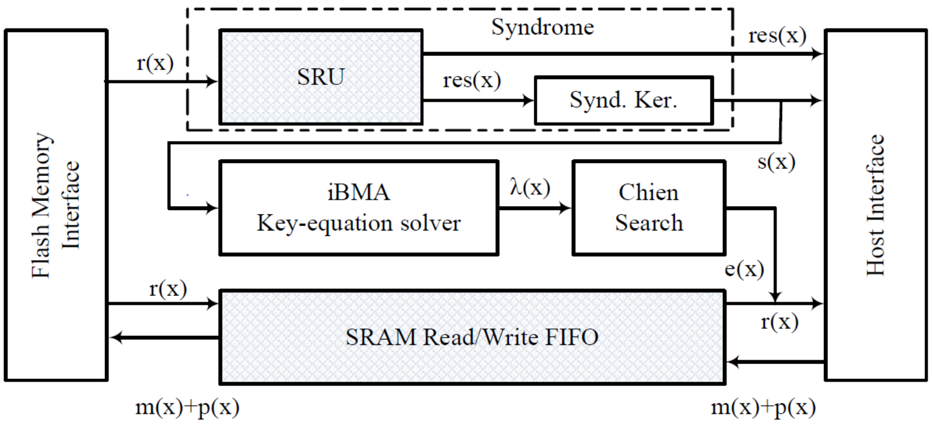


**Figure 2.** Hybrid BCH decoder block diagram.

### 3.1. Flowchart

Figure 3 depicts the flow 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 [9]. Next, a page read command is initiated to the flash memory interface. The SRU calculates the $t$ residuals of the minimal polynomial, while the data are 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 error locator polynomial $\Lambda(x)$. Once the $\Lambda(x)$ is formed, the Chien

Search Kernel (CSK) is executed for each bit location. 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 intended data from the flash memory are read, we repeat the previously mentioned steps (Node 1 in Figure 3).

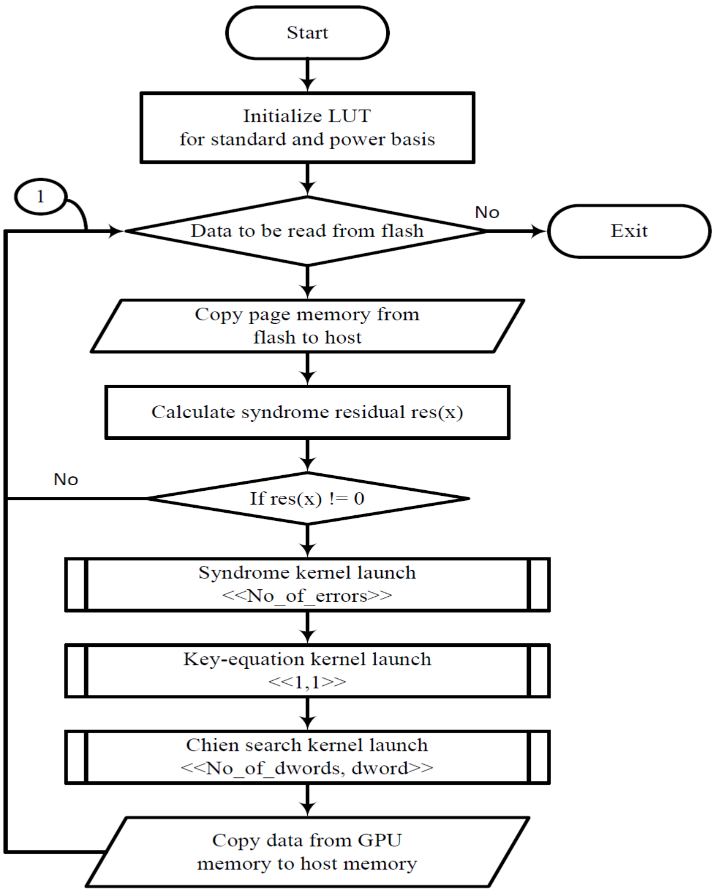

Start
Initialize LUT for standard and power basis
1
Data to be read from flash
No
Exit
Copy page memory from flash to host
Calculate syndrome residual res(x)
No
If res(x) != 0
Syndrome kernel launch <<No_of_errors>>
Key-equation kernel launch <<1,1>>
Chien search kernel launch <<No_of_dwords, dword>>
Copy data from GPU memory to host memory

**Figure 3.** Flow chart for the hybrid system.

### *3.2. Modified Syndrome Generator*

The conventional syndrome generator as discussed in Section 2.2 can be split into two steps: First, the residual polynomial $res_i(x)$, of the received code word $r(x)$, is generated by the equation $res_i(x) = r(x)\, mod\, \phi_i(x)$, where the minimal polynomial $\phi_i(x) = g_{i,m}x^m + ... + g_{i,0}$ and residual polynomial $res_i(x) = res_{i,m-1}x^{m-1} + ... + res_{i,0}$. Second, the syndrome can be calculated by substituting the primitive element $\alpha^i$ in the residual polynomial and is expressed as:

$$S_i = \sum_{k=0}^{deg(\phi_i(x))-1} res_{i,k}.(\alpha^i)^k \tag{6}$$

It is clear that by splitting the syndrome generation, $res_i(x)$ does not have any dependency on the field extensions. In fact, the polynomial division used in $res_i(x)$ is identical to a Linear Feedback Shift Register (LFSR) with its coefficient from $\phi_i(x)$. We introduce the idea to have the coefficients of the LFSR as programmable. Figure 4 represents a hardware realization of the SRU array in a serial fashion with programmable feedback coefficients. In most cases, depending on the data interface width of the flash memory interface, we can unfold the serial interpretation of the SRU to process more bits in parallel. Because of the relationship between the conjugacy class and $\phi_i(x)$ [7], it is sufficient to generate $t$ SRU units. These SRUs can compute $res_i(x)$ of different minimal polynomials in tandem.

Once the residuals are computed, the values of the $res_i(x)$ are compared for non-zero values. An error is triggered if any of the $res_i(x)$ has non-zero coefficients in them, and the GPU kernel routines for the other stages of the BCH decoder are executed.

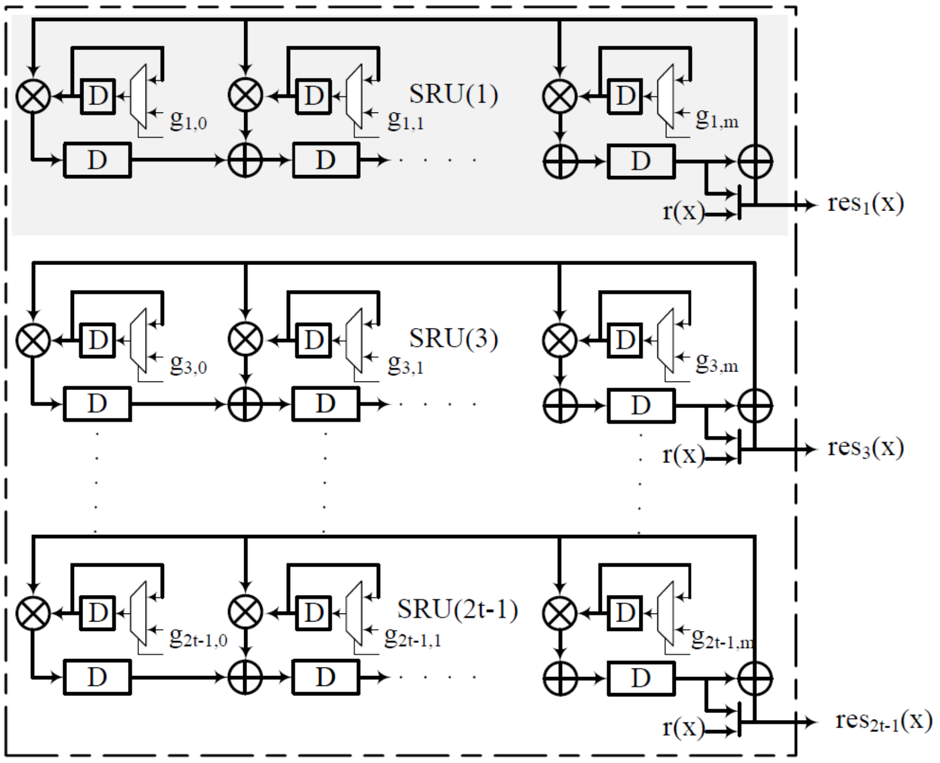


**Figure 4.** Array of the syndrome residual unit.

### 3.3. GPU Kernel Routines

Kernel routines are the fundamental sub routines, representing the SIMDtype of parallelism, executed by the GPU for our proposed decoder. Figure 5 illustrates a systematic execution of the kernel routines where PG2 and PG4 represent pages with errors. PG0, PG1, and PG3 represent pages without error. When there are no errors, the latency incurred is the computation time consumed by the SRU systolic array, as shown in Figure 5. Furthermore, to achieve better throughput, the SRU units can compute $res_i(x)$ for the next page, while the GPU kernel routines are triggered during an error scenario. For *GF* multiplication in the algorithm, the multiplicand and multiplier are converted to the power basis by referring to the LUT in the global memory. Thus, the multiplication is transformed into a simple XOR operation in the power basis domain. After the multiplication is computed, a reverse transformation is performed by referring to another basis converter LUT in the global memory. The three basic GPU kernel routines used in our approach are explained in detail below.

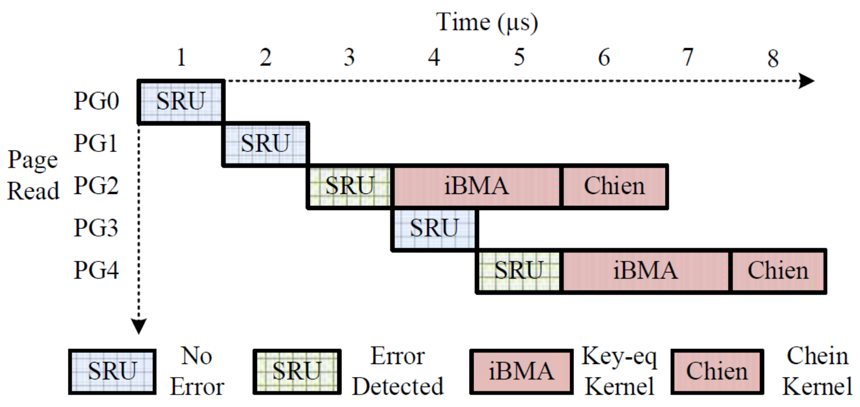


**Figure 5.** Decoder execution sequence. PG, Page.

### 3.3.1. Syndrome Kernel

In this routine, the $S_i$ is calculated by substituting the $\alpha_i$ in the equation $res_i(x)$. Since there are no dependencies on the syndromes, $t$ parallel SK routines are launched within the GPU. Algorithm 1 represents the pseudocode for the syndrome routine. The *atomicXor* operation is required to synchronize the value updated by the SK routines across multiple threads.

**Algorithm 1** Syndrome kernel.

```
1: procedure SYND KERNEL($res_i, S_i$)
2:     $sum \leftarrow 0$
3:     for $j \leftarrow 0, deg(\phi_i(x)) - 1$ do
4:         $sum \leftarrow sum + res_{i,j}.\alpha^{i.j}$
5:     end for
6:     $atomicXor(S_i, sum)$                    ▷ synchronize between threads
7: end procedure
```

### 3.3.2. Key-Equation Kernel

The KEK is the only single thread routine, in our proposal, because of the iterative nature of the iBM algorithm. Algorithm 2 represents the pseudocode for iBM implementation in the GPU routine. There are other methods like the simplified iBM (siBM) algorithm [7] proposed for the key-equation solver module, but experimental results have proven that siBM does not have significant improvement on the performance of the GPU kernel routines.

**Algorithm 2** Key-equation kernel.

```
1: procedure KEQ EQ KERNEL($\Lambda, S$)
2:     $\Lambda^{(0)} \leftarrow 1 + S_1 x$
3:     if $S_1 = 0$ then
4:         $d_p \leftarrow 1; \beta^{(1)} \leftarrow x^3; l_1 \leftarrow 0$
5:     else
6:         $d_p \leftarrow S_1; \beta^{(1)} \leftarrow x^2; l_1 \leftarrow 1$
7:     end if
8:     for $r \leftarrow 1, t-1$ do
9:         $d_r \leftarrow \sum_{i=1}^{t} \Lambda_i^{(r)} S_{2r-i+1}$
10:        $\Lambda^{(r)} \leftarrow d_p \Lambda^{(r-1)} + d_r \beta^{(r)}$
11:        if $d_r = 0$ or $r < l_r$ then
12:            $\beta^{(r+1)} \leftarrow x^2 \beta^{(r)}; l_{r+1} \leftarrow l_r; d_p \leftarrow d_p$
13:        else
14:            $\beta^{(r+1)} \leftarrow x^2 \Lambda^{(r)}; l_{r+1} \leftarrow l_r + 1; d_p \leftarrow d_r$
15:        end if
16:    end for
17: end procedure
```

### 3.3.3. Chien Search Kernel

The CS algorithm is the final step within the decoder. The primitive element $\alpha^{pos^{-1}}$ is checked if it is a root for the error locator polynomial $\Lambda(x)$ as specified in [9]. This kernel routine is an ideal candidate for the GPU because of the parallelism it offers. Each element of the finite field is evaluated in the equation $\Lambda(x)$ as shown in Algorithm 3. This evaluation kernel routine is independent for each GF element; hence these routines can be launched in parallel threads. 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 by different CSK routines. Once the error vector is formed, the error is masked with the data in the memory to yield corrected data.

**Algorithm 3** Chien search. Kernel

```
1: procedure CHIEN KERNEL($\Lambda, pos, err$)
2:     $sum \leftarrow 1$                                    ▷ Always KEQ is minimal
3:     for $j \leftarrow 0, deg(\phi_i(x)) - 1$ do
4:         $sum \leftarrow sum + \Lambda_j (\alpha^{pos^{-1}})^j$
5:     end for
6:     if $sum = 0$ then                                  ▷ $\alpha^{pos^{-1}}$ is a root
7:         atomicXor($err[pos], sum$)                      ▷ Prevent overlap write
8:     end if
9: end procedure
```

## 4. Experimental Results and Analysis

The proposed hybrid approach was compared against conventional GPU [8,9] and hardware [20] architectures. The hardware implementation of the syndrome generator was synthesized for 28-nm technology, and it achieved an operational frequency of 1 GHz. The setup used for the GPU implementation is given in Table 1. In our experiments, we analyzed the performance and the area consumed for different BCH code sizes. We have used the finite field dimension of $m = 12, 13, 14, 15$ in our comparison, which corresponds to block sizes of 256, 512, 1024, 2048 bytes. Furthermore, we have analyzed the results for different bit errors $(t = 2, ..., 40)$ in our experiments.

**Table 1.** Experimental setup.

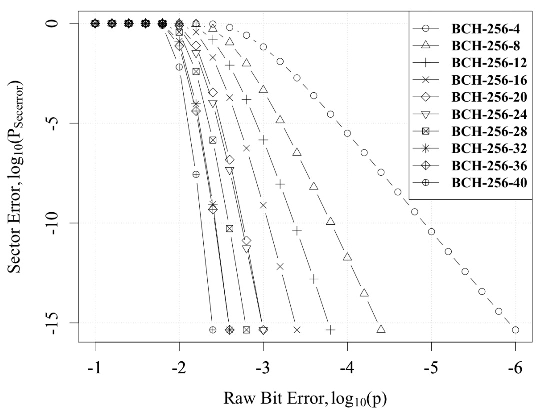
| | **GPGPU** | **CPU** |
|---|---|---|
| Platform | Geforce GTX 760. 1152 cores | Intel Xeon i7 |
| Clock Freq. | 1.033 GHz | 3.7 GHz |
| Memory | GDDR5(2 GB), 6 Gbps | DDR2 (32 GB), 102.4 Gbps |

## 4.1. Error Analysis

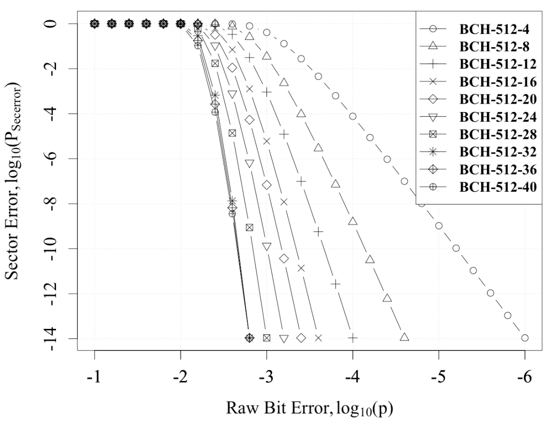
The error correction capability increased with $n$, but the larger the $n$, the higher the probability of random bit error. Based on the raw bit error probability $p$, parity bits, and message code $cpar = 2 \cdot t \cdot m + |m(x)|$, the sector with correctable error $(P_{secErr})$ might increase and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

$$P_{secErr} = 1 - \sum_{i=t+1}^{cpar} \binom{cpar}{i} \cdot p^i \cdot (1-p)^{(cpar)-i} \tag{7}$$

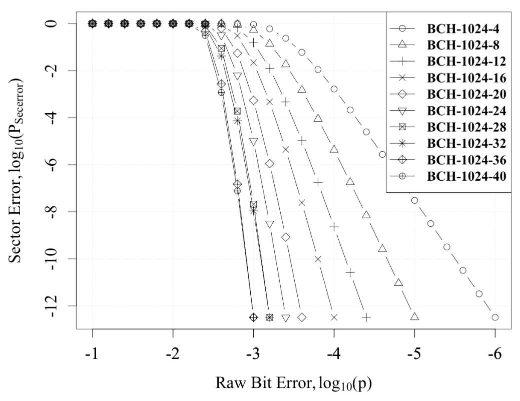
Figure 6 plots the bit error vs. sector error for different BCH codes. We can also see that the $P_{secErr}$ decreased as $p$ decreased. Furthermore, there was a slight increase in $P_{secErr}$ when compared against different $m$. This was due to the increase in the probability of error within bigger sector sizes.



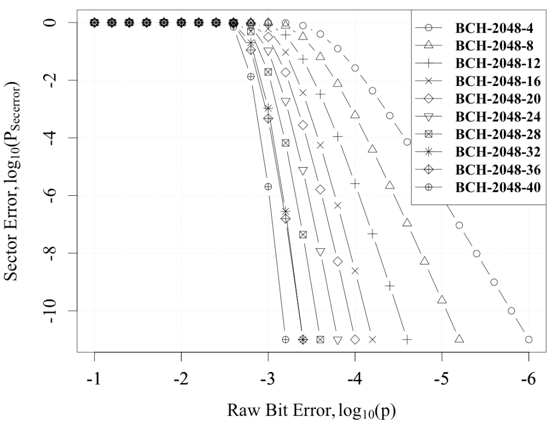
(**a**) Block size, 256 bytes ($m = 12$)



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



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



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

**Figure 6.** Raw bit error vs. sector error.

## *4.2. Syndrome Generation Analysis*

Syndrome generation is the critical area where the proposed hybrid method provides an advantage over the GPU methods [8,9]. Figure 7 shows the plot of the syndrome computation time vs. different bit errors $(t = 2, ..., 40)$ across different finite fields $(m = 12, 13, 14, 15)$ for different architectures: GPU [9], hardware [21], and hybrid (proposed). The computation of the SRU engine depended on the number of clock cycles required for a page read. Since all the $res_i(x)$ that were necessary for the key-equation solver were calculated in tandem, the latency only depended on the read cycles for flash memory. The hardware architecture for syndrome generation consumed the same clock cycle as the hybrid approach since the approach to syndrome generation was similar. It should be noted that the GPU unit was used as a display unit, so the results of kernel profiling depended on the load of the GPU during the execution of the kernel routine. The execution time for the syndrome on the GPU architecture depended on the number of threads getting executed, and typically, it was from 30–100 µs.

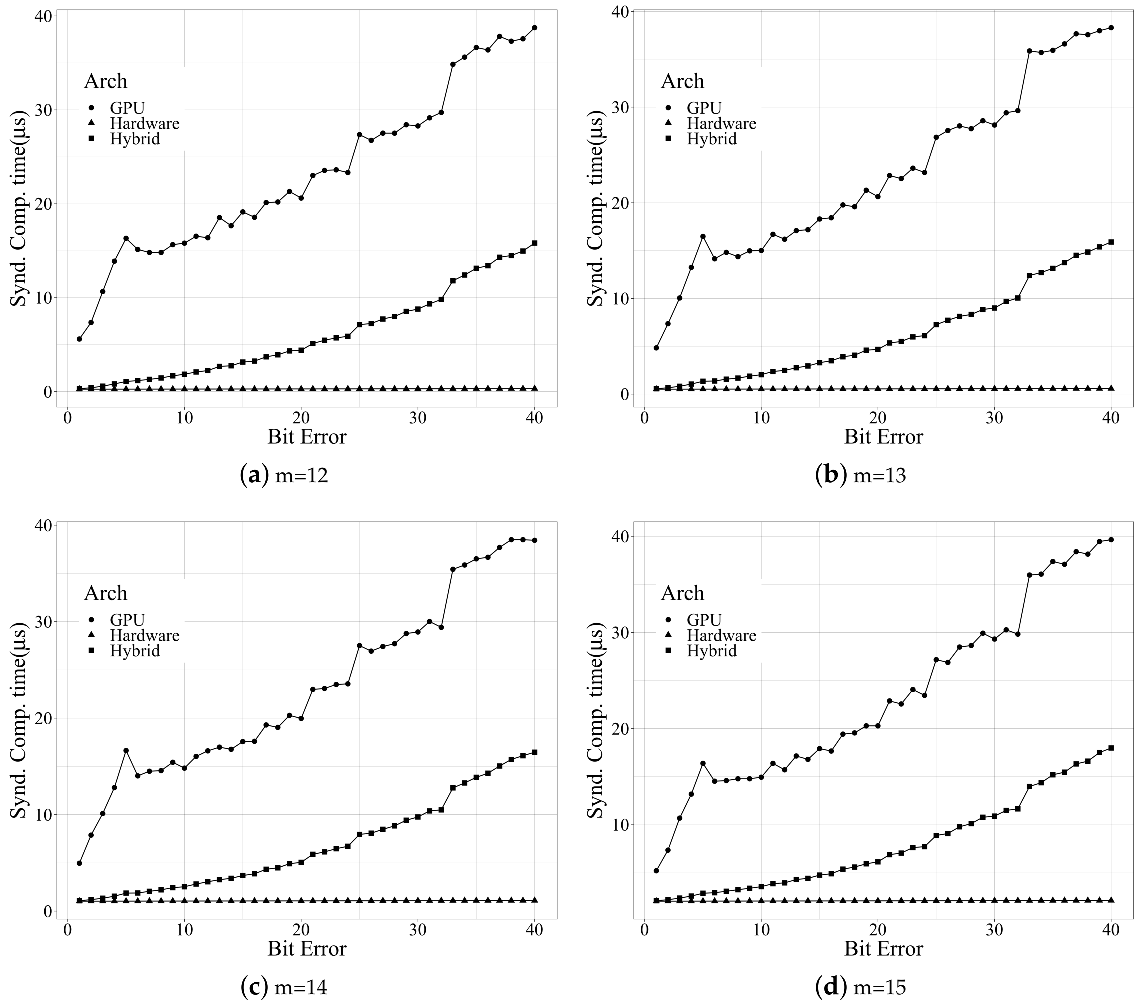


**Figure 7.** Syndrome computation time for different arches.

## *4.3. VLSI Analysis*

Table 2 compares the hardware area required for different methods of the syndrome generator ([20] and the proposed). Since the GPUs were employed for key-equation and Chien search, the hardware was only compared for the syndrome generator unit for multi-bit error correction for different *GF* dimensions. This is a fair comparison since the area of the GPUs was already accounted for systems with graphical display. The hardware implementations were targeted for 28-nm and met a frequency of 1 GHz. In order to support different finite field ($m = 12, 13, 14, 15$) and 40-bit error correction, the hardware method [20] consumed 30,247 µm$^2$, whereas our proposed hybrid architecture consumed 10,633 µm$^2$, thus saving two-fold of the area. This area savings was due to the splitting of the syndrome generation into two units (SRU and syndrome kernel). When there were no errors in the page, the total time taken by the proposed decoder was less than 5 µs (Figure 7), which was less than the average read latency of 100 µs. Table 3 compares the power consumed by the conventional method [20] and our proposed method. The last entry in the table provides the power consumption required to support error correction over different fields ($m = 12, 13, 14, 15$) and until 40-bit error correction. There was a savings of 4 mW in our proposed method.

**Table 2.** Area comparison for different syndrome generators vs. the proposed SRU.

| | **Setup** | **Area for *t* (µm$^2$)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ***4*** | ***8*** | ***12*** | ***16*** | ***20*** | ***24*** | ***28*** | ***32*** | ***36*** | ***40*** |
| **m = *12*** | [20] | 606 | 1287 | 2079 | 2871 | 3256 | 3661 | 3959 | 4252 | 4611 | 4917 |
| | Prop. | 853 | 1704 | 2550 | 3399 | 4209 | 5061 | 5903 | 6747 | 7592 | 8436 |
| **m = *13*** | [20] | 858 | 1863 | 2962 | 4043 | 4648 | 5246 | 5814 | 6387 | 6988 | 7573 |
| | Prop. | 924 | 1846 | 2767 | 3682 | 4607 | 5532 | 6390 | 7305 | 8308 | 9134 |
| **m = *14*** | [20] | 916 | 2002 | 3167 | 4330 | 5016 | 5723 | 6375 | 7048 | 7714 | 8390 |
| | Prop. | 994 | 1985 | 2976 | 3966 | 4966 | 5959 | 6948 | 7942 | 8935 | 9927 |
| **m = *15*** | [20] | 988 | 2172 | 3435 | 4707 | 5485 | 6292 | 7043 | 7818 | 8587 | 9367 |
| | Prop. | 1064 | 2125 | 3186 | 4249 | 5318 | 6383 | 7448 | 8504 | 9570 | 10,633 |
| **m = *12, ..., 15*** | [20] | 3368 | 7324 | 116,43 | 15,951 | 18,405 | 20,922 | 23,191 | 25,505 | 27,900 | ***30,247*** |
| | Prop. | 1064 | 2125 | 3186 | 4249 | 5318 | 6383 | 7448 | 8504 | 9570 | ***10,633*** |

**Table 3.** Power comparison for different syndrome generators vs. the proposed SRU.

| | **Setup** | **Power for *t* (mW)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | ***4*** | ***8*** | ***12*** | ***16*** | ***20*** | ***24*** | ***28*** | ***32*** | ***36*** | ***40*** |
| **m = *12*** | [20] | 0.179 | 0.374 | 0.599 | 0.819 | 0.897 | 0.978 | 1.037 | 1.097 | 1.170 | 1.232 |
| | Proposed | 0.167 | 0.336 | 0.499 | 0.673 | 0.841 | 1.014 | 1.185 | 1.354 | 1.524 | 1.695 |
| **m = *13*** | [20] | 0.226 | 0.491 | 0.773 | 1.054 | 1.178 | 1.304 | 1.426 | 1.546 | 1.674 | 1.797 |
| | Proposed | 0.178 | 0.355 | 0.534 | 0.714 | 0.889 | 1.061 | 1.248 | 1.424 | 1.615 | 1.778 |
| **m = *14*** | [20] | 0.231 | 0.503 | 0.784 | 1.068 | 1.213 | 1.363 | 1.499 | 1.640 | 1.780 | 1.923 |
| | Proposed | 0.187 | 0.373 | 0.561 | 0.747 | 0.938 | 1.122 | 1.309 | 1.493 | 1.678 | 1.863 |
| **m = *15*** | [20] | 0.235 | 0.512 | 0.812 | 1.110 | 1.270 | 1.438 | 1.593 | 1.755 | 1.915 | 2.078 |
| | Proposed | 0.194 | 0.388 | 0.588 | 0.776 | 0.975 | 1.168 | 1.370 | 1.564 | 1.755 | 1.953 |
| **m = *12, ..., 15*** | [20] | 0.692 | 1.506 | 2.369 | 3.232 | 3.661 | 4.105 | 4.518 | 4.941 | 5.369 | ***5.798*** |
| | Proposed | 0.194 | 0.388 | 0.588 | 0.776 | 0.975 | 1.168 | 1.370 | 1.564 | 1.755 | ***1.953*** |

## *4.4. Performance Analysis*

The comparison of the total time taken, in case of an error, is compared for the hardware [20], GPU [8,9], and hybrid architecture (proposed) in Figure 8 (the variable *be* represents bit error). We can see that the hybrid approach was better than the GPU method because of the SRU implementation in hardware. However, the hardware implementation took less than 1 µs because of the high performance (which is indistinguishable in Figure 8). We can observe a gain of more than 25% when the system has errors. We can find the probability of sector error from a given bit error rate using Equation (7). For a given $P_{secError}$, the throughput was calculated for a second's worth of data transfer.

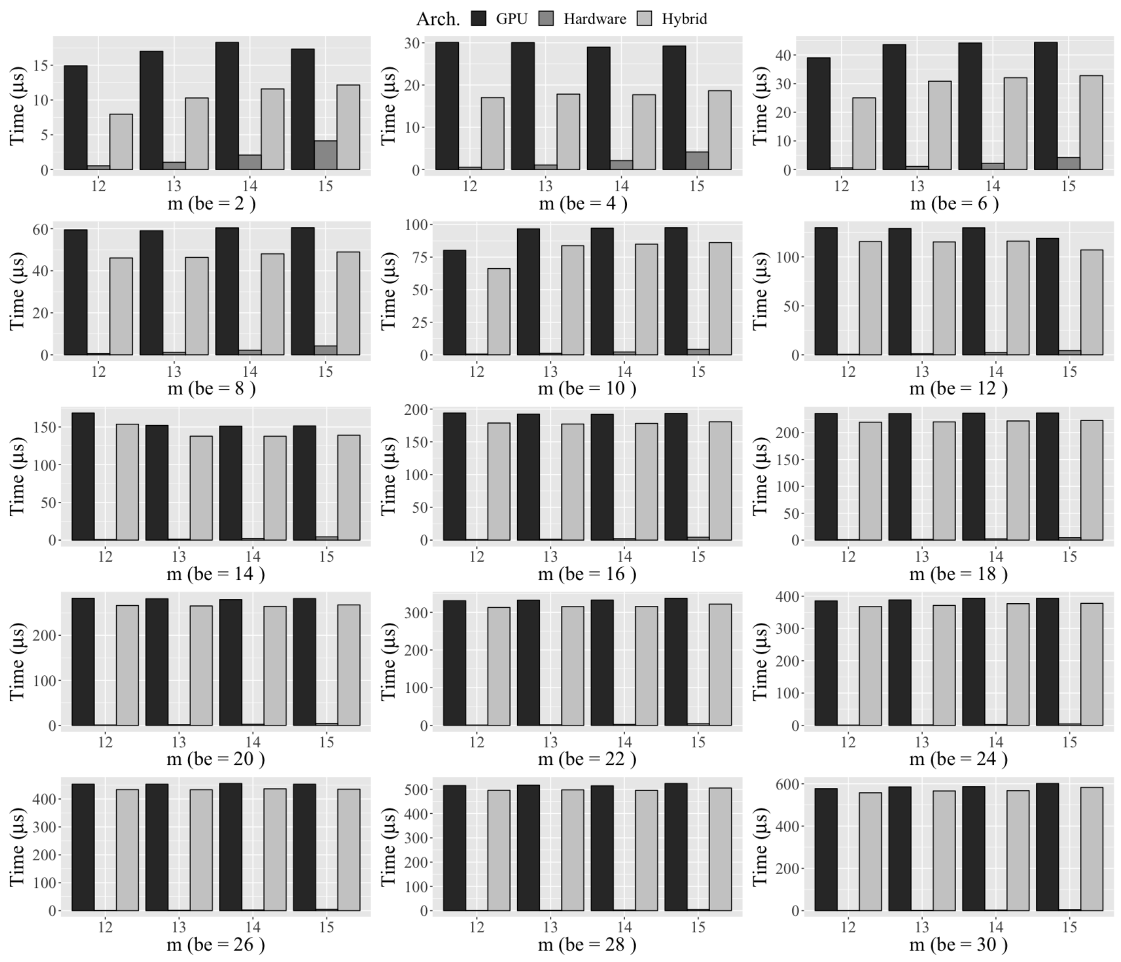
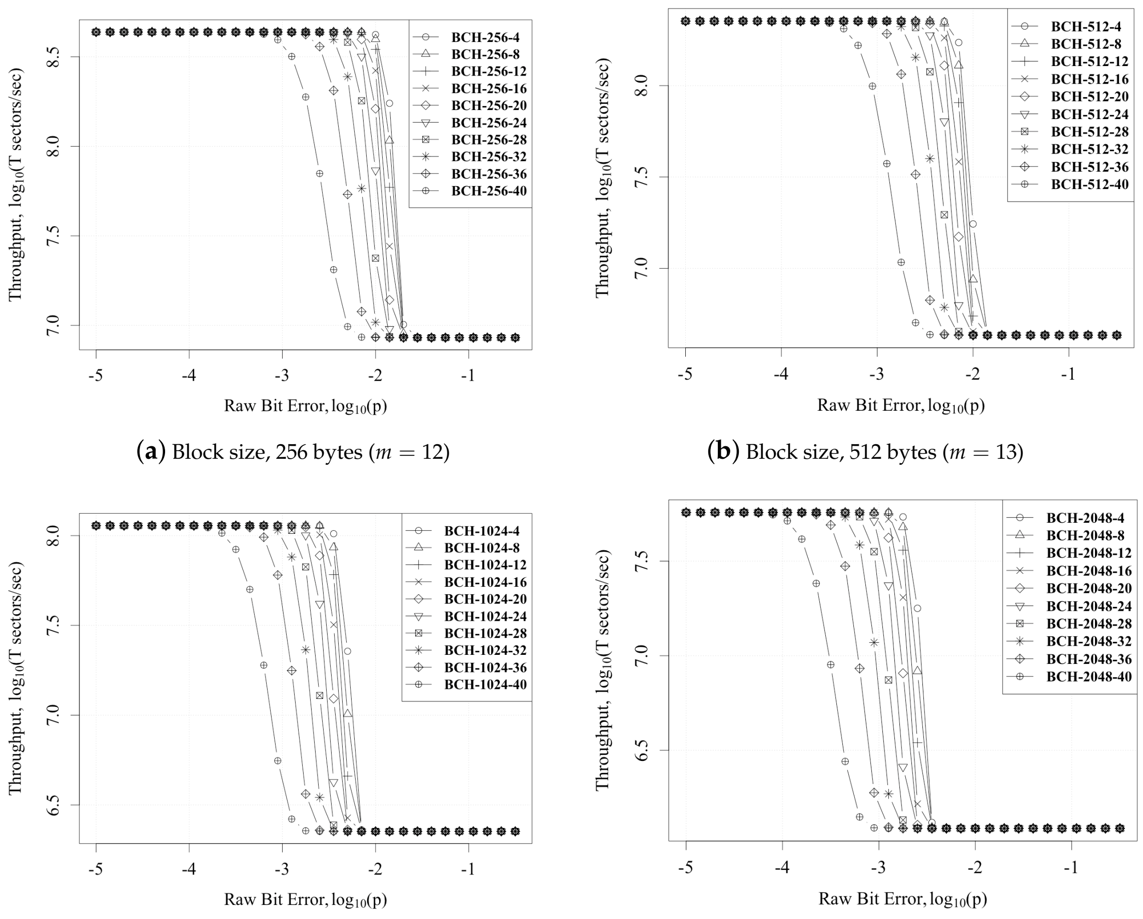


**Figure 8.** Total computation time for different architectures and different finite fields.

Figure 9 represents the plot for throughput vs. bit error rate for different finite fields ($m = 12, 13, 14, 15$) and different $t = 4, ..., 40$. We can see that for $m = 12, 13$, the throughput was sustained till $10^{-3}$, and for $m = 14, 15$, the throughput was sustained till $10^{-3.5}$. This throughput is sustainable for flash memories that have an Uncorrectable Bit Error Rate (UBER) of $10^{-15}$, an it is also sustainable for the end of life for flash memories which is greater than $10^{-5}$.



(**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 ($m = 15$)

**Figure 9.** Raw bit error vs. throughput (sectors/s).

## 5. Conclusions

In this paper, we have proposed a novel hybrid method to implement an efficient BCH decoder for different finite field extensions by having the SRU module in hardware and the rest implemented in GPU kernel routines. By using this method, we have given flexibility on two parameters: first, the flexibility over different finite fields $GF(2^m)$; second, the flexibility over different bit error support. The flexibility over $GF(2^m)$ was achieved by splitting the syndrome into the SRU unit and the syndrome kernel. The SRU module resided on the Euclidean domain of $GF(2)$ polynomials, thus making it programmable across multiple finite fields. The syndrome kernel was executed only when an error was encountered. The latency taken by our method, without error, was superior to the CPU and GPU implementations and was equal to the performance observed in [20]. Besides, we had two-fold area savings in the SRU unit to achieve flexibility over $GF(2^m)$ and bit errors. Therefore, this hybrid approach is a pragmatic solution to have a flexible error correction for modern NAND flash devices.

**Author Contributions:** Conceptualization, A.S. and T.O.; methodology, A.S.; software, A.S.; validation, A.S.; formal analysis, A.S. and T.O.; investigation, A.S. and T.O.; resources, T.O.; data curation, T.O.; writing, original draft preparation, A.S.; writing, review and editing, A.S. and T.O.; visualization, A.S.; supervision, T.O.; project administration, T.O.

**Funding:** This research received no external funding.

**Conflicts of Interest:** The authors declare no conflict of interest.

## Abbreviations

The following abbreviations are used in this manuscript:

| | |
|---|---|
| BCH | Bose–Chaudhuri–Hocquenghem |
| iBMA | inversion-less Berlekamp-Massey algorithm |
| CPU | Central Processing Unit |
| CS | Chien Search |
| CSK | Chien Search Kernel |
| KEK | Key Equation Kernel |
| GPU | Graphical Processing Unit |
| GPGPU | General Purpose Graphical Processing Unit |
| LDPC | Low Density Parity Check |
| LFSR | Linear Feedback Shift Register |
| MLC | Multi-Level Cell |
| RS | Reed–Solomon |
| SK | Syndrome calculation Kernel |
| SLC | Single-Level Cell |
| SRU | Syndrome Residual Unit |
| UBER | Uncorrectable Bit Error Rate |

## References

1. Micheloni, R.; Marelli, A.; Crippa, L. *Inside NAND Flash Memories*; Springer: New York, NY, USA, 2010; doi:10.1007/978-90-481-9431-5.
2. Spinelli, A.; Compagnoni, C.; Lacaita, A. Reliability of NAND Flash Memories: Planar Cells and Emerging Issues in 3D Devices. *Computers* **2017**, *6*, 16, doi:10.3390/computers6020016. [CrossRef]
3. Costell, S.L.; Costello, D. *Error Control Coding—Fundamentals and Applications*, 2nd ed.; Prentice-Hall: Englewood Cliffs, NJ, USA, 2004; pp. 192–233.
4. Cho, J.; Sung, W. Efficient software-based encoding and decoding of BCH codes. *IEEE Trans. Comput.* **2009**, *58*, 878–889, doi:10.1109/TC.2009.27. [CrossRef]
5. Poolakkaparambil, M.; Mathew, J.; Jabir, A. Multiple Bit Error Tolerant Galois Field Architectures over GF (2m). *Electronics* **2012**, *1*, 3–22, doi:10.3390/electronics1010003. [CrossRef]
6. Lee, Y.; Yoo, H.; Yoo, I.; Park, I.C. High-throughput and low-complexity BCH decoding architecture for solid-state drives. *IEEE Trans. Very Large Scale Integr. Syst.* **2014**, *22*, 1183–1187, doi:10.1109/TVLSI.2013.2264687. [CrossRef]
7. Zhang, X. *VLSI Architectures for Modern Error-Correcting Codes*, 2nd ed.; CRC Press: Boca Raton, FL, USA, 2016; pp. 189–225.
8. Qi, X.; Ma, X.; Li, D.; Zhao, Y. Implementation of accelerated BCH decoders on GPU. In Proceedings of the 2013 International Conference on Wireless Communications and Signal Processing (WCSP), Hangzhou, China, 24–26 October 2013; pp. 1–6, doi:10.1109/WCSP.2013.6677084. [CrossRef]
9. Subbiah, A.K.; Ogunfunmi, T. Efficient implementation of BCH decoders on GPU for flash memory devices using iBMA. In Proceedings of the 2016 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, USA, 7–11 January 2016; pp. 275–278, doi:10.1109/ICCE.2016.7430612. [CrossRef]
10. NVIDIA. *Cuda C Programming Guide*; NVIDIA: Santa Clara, CA, USA, 2015; PMCID:PMC3074485, NIHMSID:Nihms253063, doi:10.1016/j.pedhc.2005.10.011.
11. Parhi, K.K. Eliminating the fan-out bottleneck in parallel long BCH encoders. *IEEE Trans. Circuits Syst. I Regul. Pap.* **2004**, *51*, 512–516, doi:10.1109/TCSI.2004.823655. [CrossRef]
12. Chen, H. CRT-based high-speed parallel architecture for long BCH encoding. *IEEE Trans. Circuits Syst. II Express Briefs* **2009**, *56*, 684–686, doi:10.1109/TCSII.2009.2024247. [CrossRef]
13. Tang, H.; Jung, G.; Park, J. A hybrid multimode BCH encoder architecture for area efficient re-encoding approach. In Proceedings of the IEEE International Symposium on Circuits and Systems, Lisbon, Portugal, 24–27 May 2015; Volume 2015, pp. 1997–2000, doi:10.1109/ISCAS.2015.7169067. [CrossRef]

14. Subbiah, A.K.; Ogunfunmi, T. Area-effcient re-encoding scheme for NAND Flash Memory with multimode BCH Error correction. In Proceedings of the 2018 IEEE International Symposium on Circuits and Systems (ISCAS), Florence, Italy, 27–30 May 2018; pp. 1–5, doi:10.1109/ISCAS.2018.8351503. [CrossRef]
15. Zhang, X. An efficient interpolation-based chase BCH decoder. *IEEE Trans. Circuits Syst. II: Express Briefs* **2013**, *60*, 212–216, doi:10.1109/TCSII.2013.2251941. [CrossRef]
16. Yang, C.H.; Huang, T.Y.; Li, M.R.; Ueng, Y.L. A 5.4 uw soft-decision bch decoder for wireless body area networks. *IEEE Trans. Circuits Syst. I: Regul. Pap.* **2014**, *61*, 2721–2729, doi:10.1109/TCSI.2014.2312478. [CrossRef]
17. Jamro, E. The Design of a Vhdl Based Synthesis Tool for Bch Codecs. Ph.D. Thesis, University of Huddersfield, Huddersfield, UK, 1997.
18. Sun, F.; Devarajan, S.; Rose, K.; Zhang, T. Design of on-chip error correction systems for multilevel NOR and NAND flash memories. *IET Circuits Devices Syst.* **2007**, *1*, 241–249, doi:10.1049/iet-cds. [CrossRef]
19. Sun, F.; Rose, K.; Zhang, T. On the Use of Strong BCH Codes for Improving Multilevel NAND Flash Memory Storage Capacity. In Proceedings of the IEEE Workshop on Signal Processing, Banff, AB, Canada, 2–4 October 2006; pp. 1–5.
20. Park, B.; Park, J.; Lee, Y. Area-Optimized Fully-Flexible BCH Decoder for Multiple GF Dimensions. *IEEE Access* **2018**, *6*, 14498–14509, doi:10.1109/ACCESS.2018.2815640. [CrossRef]
21. Wei, L.; Junrye, R.; Wonyong, S. Low-power high-throughput BCH error correction VLSI design for multi-level cell NAND flash memories. In Proceedings of the 2006 IEEE Workshop on Signal Processing Systems Design and Implementation (SIPS), Banff, AB, Canada, 2–4 October 2006; pp. 303–308, doi:10.1109/SIPS.2006.352599. [CrossRef]
22. Park, B.; An, S.; Park, J.; Lee, Y. Novel folded-KES architecture for high-speed and area-efficient BCH decoders. *IEEE Trans. Circuits Syst. II Express Briefs* **2017**, *64*, 535–539, doi:10.1109/TCSII.2016.2596777. [CrossRef]
23. Yoo, H.; Lee, Y.; Park, I.C. Low-Power Parallel Chien Search Architecture Using a Two-Step Approach. *IEEE Trans. Circuits Syst. II: Express Briefs* **2016**, *63*, 269–273, doi:10.1109/TCSII.2015.2482958. [CrossRef]
24. Freudenberger, J.; Spinner, J. A Configurable Bose–Chaudhuri–Hocquenghem Codec Architecture for Flash Controller Applications. *J. Circuits Syst. Comput.* **2013**, *23*, 1450019, doi:10.1142/s0218126614500194. [CrossRef]