Original scientific paper

Software Optimization for Fast Encoding and Decoding of Reed-Solomon Codes

Sergey Skorokhod and Andrey Barlit

***Abstract* — In this work, we propose a software library written in C for encoding and decoding Reed-Solomon codes. Library consists of one scalar codec and two vectorized codecs for x86 architecture. Vectorized codecs use the benefits of SSSE3 or AVX2 instruction sets. We compare the performance of our three codecs with the JPWL RS codec from the Open JPEG library. The performance comparison methodology is described, and it is based on the measuring of the encoding and decoding speed. The results demonstrate a 4.1x speed gain with the scalar codec and a 19.6x gain with the vectorized codec. Based on testing results and supported instruction sets, a dynamic selection of codec version is proposed.**

***Keywords* — FEC (Forward Error Correction), RS (Reed-Solomon) codes, SIMD (Single Instruction Multiple Data).**

I. Introduction

COMMUNICATION channels, especially wireless, are subject to channel noise and errors that may be introduced during signal transmission. To detect and correct such errors, FEC schemes are used, which all add some redundant data to a message. That data is used by the receiver so it can check the consistency of the delivered message and recover data if possible. In terms of required data overhead, maximum distance separable (MDS) codes are known to be optimal [1]. MDS codes include RS codes.

RS codes are a group of codes that enables error detection and correction in data blocks. All calculations are performed in Galois Fields (GF) using polynomial mathematics, which is computationally heavy. Reed-Solomon codes are usually denoted as RS(n, k), where k is the number of information symbols in the block, n is the total number of symbols, and $t = [(n - k) / 2]$ is the error correction capability. The amount of redundancy and error correction capabilities of RS codes are directly proportional. Therefore, the choice of RS code has a direct influence on the required computational resources.

In error-prone networks, an Automatic Repeat Request (ARQ) can also be used. It discards the damaged data packet and then requests the same packet again. This approach is very simple in implementation, but it increases both transmitting data volume and communication latency. Despite that RS codes use relatively complex mathematics they may improve both reliability and latency.

The ITU T.810 standard, targeting the JPEG 2000 coded image data protection for transmission over wireless channels and networks, specifies JPWL as a set of tools for error correction and signaling [2]. RS codes are the main FEC technique used in JPWL specification.

We have examined the implementation of the JPWL system as part of the open-source library Open JPEG [3], designed to encode images in the JPEG 2000 format. The performance requirements are especially high for streaming video since the JPWL system spends most of its time processing RS codes. This is generally not a problem because modern CPUs provide more than enough computational resources for such tasks. However, on laptops higher CPU usage draws a battery faster, which cannot be ignored.

Research has been done on the current usage of the FEC and RS codes. Different error correction schemes, including FEC, are used in wireless networks. Reference [4] shows the design of an optimal rate allocation for video streaming over device-to-device communications in 5G networks. Authors embed FEC before transmission of packets of the video segment. The concatenated codes in wireless environments are evaluated in [5] and it is shown that usage of RS codes leads to a better performance.

Another field where FEC is widely applied is distributed data storage. An overview of coding for distributed storage over the past decade has been done in [6]. The authors highlight that the repair process for erasure codes can be considered efficient if the amount of repair data transferred over the network is minimized. Encoding-aware data placement for erasure-coded storage has been proposed in [7] because the performance of degraded reads has been a critical issue. Reference [8] addresses the same issue with distributed storage. Authors present Clay codes that combine MDS codes with Minimum Storage Regenerating (MSR) codes. A single erasure repair method for RS codes that achieves the optimal repair bandwidth has been proposed in [9]. The main idea is to collect a sufficiently large number of traces of the erased symbol. This method has been extended to cope with two and three erasures in [10].

In [11], the authors propose a coded computation scheme based on MDS codes for distributed computing systems to make them robust to slower nodes. Such a

Paper received September 21, 2021; revised August 11, 2022; accepted August 15, 2022. Date of publication December 26, 2022. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Zorica Nikolić.

Sergey V. Skorokhod, Institute of Computer Technology and Information Security, Southern Federal University, 105/42 Bolshaya Sadovaya Str., Rostov-on-Don, 344006, Russia (phone: +7 863 218-40-00; e-mail: skorohodsv@sfedu.ru).

Andrew V. Barlit, Institute of Computer Technology and Information Security, Southern Federal University, 105/42 Bolshaya Sadovaya Str., Rostov-on-Don, 344006, Russia (phone: +7 863 218-40-00; e-mail: abarlit@sfedu.ru).

scheme can be used in large matrix multiplication where one of the matrices is small enough to fit into a single compute node. A general framework for private information retrieval from arbitrary coded databases if the storage code is a generalized RS code is presented in [12].

New algorithms for specific cases are also being developed. An efficient partial decoding algorithm for an m-interleaved RS is proposed in [13]. Reference [14] describes a fast RS encoding algorithm with four to seven parity symbols.

Research has been done on software implementation of GF. There are not many authors who use SIMD instructions in their software. For example, [15] only outlines some basic principles of using SIMD. Most useful is [16], which demonstrates how to leverage SIMD commands to perform multiplication. This technique was used in our vectorized codecs. Reference [17] explains the functionality of new instructions recently announced by Intel [18] for its future architecture (codename 'Ice Lake'). Authors demonstrate their usage for some fast computations in GF.

Error correction capability of RS codes was also analyzed. The decoding algorithm that can go beyond error-correction bound is proposed in [19]. However, this algorithm is much more computationally heavy than the standard decoding algorithm for RS codes, so it should not be used without an absolute necessity.

We propose three optimized codecs for both encoding and decoding of RS codes for general applications. A codec spends most of the time in loops and reducing the number of operations inside them leads to a substantial increase in computational speed. Optimization of the non-vectorized codec (scalar version) relies on an extended look-up table (LUT) for multiplication in GF. Optimization of vectorized codecs (SSSE3 and AVX2 versions) uses a more complex approach, but an increase in performance is considerably higher. A detailed description of all of the vector functions that can be used in the C programming language can be found in [20].

We have evaluated the performance of our three codecs against JPWL implementation from the Open JPEG library. Four RS codes with different error correction capabilities have been used for testing. Results show that our scalar decoder achieves a speedup of 4.1x, and the SSSE3 decoder achieves a speedup of 19.6x. The proposed codecs can be used for any application where FEC is required.

In Section II the optimization strategies are discussed. Section III describes the testing setup and Section IV presents the results. Conclusions are summarized in Section V.

II. Proposed Optimizations

In this section, we describe optimization strategies for our scalar and vectorized codecs. The scalar codec is compared against the JPWL RS codec from Open JPEG.

The general strategy for optimization of multiplication in GF involves LUT [21]. This LUT contains logarithms and exponents of all possible values of the selected finite set of numbers. Most software applications use a finite set consisting of 2^8-1 numbers, because in this case values are represented by bytes. LUT for this set is presented in Table 1.

Table 1: Multiplication LUT for GF

LUT index	Values				
0…255	255	log 1	…	log 254	log 255
256…511	exp 0	exp 1	…	exp 254	exp 0

Multiplication of two 8-bit values a and b with LUT is performed as follows:

$$a * b = \exp((\log a + \log b) \bmod 255). \quad (1)$$

It should be noted that log 0 value is undefined and usually represented by a “forbidden” value, that is 255, because exp 255 = exp 0. Therefore, before multiplication both operands must be checked for zero value.

A. Proposed LUT optimization

Checking every operand for zero value when multiplying is costly in terms of the required CPU time. The complexity of most calculations for RS codes is $O(n^2)$ because of loops. It is obvious that each additional line of code inside loops, especially if it is a branch, has a significant impact on the performance.

Modulo operation can be skipped if we will use integer 511 as a “forbidden” value for log 0 and extend the LUT for exponentials up to 511 values. Then, it is necessary to remove the multiplication check. To multiply by zero without checking both operands, 1025 zeros must be added to the LUT after 511 exponents.

With these changes, the sum of logarithms of any 2 numbers will point to the correct exponent value in the LUT. The overall LUT size is 2KB (512 + 1536 bytes), as we have to use 16-bit variables for log values. The resulting LUT is represented in Table 2.

Table 2: Extended LUT for scalar version

LUT index	Values				
0…255	511	log 1	…	log 254	log 255
256…511	exp 0	exp 1	…	exp 254	exp 0
512…767	exp 1	exp 2	…	exp 0	0
768…1535	0	0	…	0	0

Without modulo operation and zero checking, multiplication of any 2 values is reduced to determining the logarithms from the LUT, summing them, and finding the result by determining the exponent from the LUT.

B. Integer division optimization

Mathematics for RS codes also involves raising the given value a to a certain power b. In a logarithmic form it is performed as follows:

$$a^b = \exp((b * \log a) \bmod 255). \quad (2)$$

In this case, modulo operation cannot be avoided. In x86 processors, there is a *div* instruction (used by the modulo operator) that performs integer division while at the same time calculating the remainder. However, this instruction takes up to 46 CPU cycles even on modern CPUs [22], so a faster way is required.

Original scientific paper

In the Open JPEG library, there is the "*modnn*" function (see Table 3). While it is still faster than the modulo operator, it is not optimal, since it contains a loop. Optimization of this function is based on two considerations. Firstly, the maximum number from which the remainder will be calculated is 255 * 254 < 2^{16}. Therefore, the loop will be iterated no more than 2 times. Secondly, the preliminary subtraction of 255 can be omitted. This may in some cases result in a remainder of 256, but for an extended LUT this is a valid index. Both *modnn* and our *mod255* functions are shown in Table 3.

TABLE 3: REMAINDER CALCULATION OPTIMIZATION

Open JPEG	Optimized

int modnn(int x) {
while (x >= 255) {
x = x – 255;
x = (x >> 8) + (x & 255); }
return x; }
	void inline mod255(int* x) {
	*x = (*x >> 8) + (*x & 255);
	*x = (*x >> 8) + (*x & 255);
	}

As a result, there is no loop, no subtraction of 255, and the inline directive allows the compiler to efficiently replace a function call with several commands.

C. Vector-by-scalar multiplication

In [16], a very efficient vector-by-scalar multiplication method is presented. This method is based on the vector instruction "*pshufb xmm, xmm*" from the SSSE3 set. This instruction considers the first register *xmm* as an array of 16 indices, and the second as a table of 16 bytes from which the selection is made. Thus, it acts as the LUT, but with a maximum of 16 elements.

The multiplication of a vector $\bar{a}$ by a scalar s can be represented as follows (& – *bitwise and* operator):

$$\bar{a} * s = ((\bar{a} \,\&\, 0xF0) * s) + ((\bar{a} \,\&\, 0x0F) * s). \quad (3)$$

In the left bracket, the most significant 4 bits of the vector $\bar{a}$ are multiplied by the scalar s, and in the right bracket, the lower 4 bits are multiplied by the same scalar. Thus, for the left and right brackets separately, there are 2^4 variants of the multiplication result.

To perform multiplication in this form, an array of 256 elements must be calculated (for each value of the s). One element is comprised of two 16-bytes vectors. These vectors hold the multiplication result of the scalar s by all 16 combinations of the higher and the lower 4 bits. A resulting LUT will require an additional 8KB memory and can be represented in Table 4 (2 elements are shown).

TABLE 4: LUT FOR VECTOR-TO-SCALAR MULTIPLICATION

Scalar	Two 16-byte vectors			
1 = 1 + (0 * 2^4)	0	1*1	…	15*1
	0	0	…	0
92 = 12 + (5 * 2^4)	0	1*12	…	15*12
	0	1*5	…	15*5

The multiplication itself is performed as follows. Firstly, the lower 4 bits are multiplied by a scalar. Secondly, the higher 4 bits are shifted to the place of the lower ones and also multiplied. A shift is required because the *pshufb* instruction needs indices in the lower 4 bits of each byte of the vector. The bitmask is used to clear the most significant 4 bits in both cases. Finally, the results of these two multiplications are summed (in GF a sum is performed by the bitwise XOR operation).

D. Preliminary calculated vectors for exponentiation

Calculating syndromes S and error positions during decoding a message x requires an exponentiation operation because we need to substitute roots in the message polynomial [21]. Substitution of roots means that all bytes of the message are multiplied by the same root s, but raised to the sequential powers, starting with 0. An example for a message length of 4 bytes can be written as

$$S(x) = x_0 * s^3 + x_1 * s^2 + x_2 * s^1 + x_3 * s^0. \quad (4)$$

At first glance, the vector-to-scalar multiplication method is not suitable here, because it is a vector-to-vector operation. But specific exponentiation LUT solves this problem because when calculating syndromes, we always take known roots starting at 1, and they are always raised to sequential powers. An example of the root substitution for a message length of 32 bytes is shown in Fig. 1.

$$\begin{aligned}
&b_0 \cdot 1^{31} + b_1 \cdot 1^{30} + \ldots + b_{31} \cdot 1^0 = S_0\\
&b_0 \cdot 2^{31} + b_1 \cdot 2^{30} + \ldots + b_{31} \cdot 2^0 = S_1\\
&\vdots\\
&b_0 \cdot 15^{31} + b_1 \cdot 15^{30} + \ldots + b_{31} \cdot 15^0 = S_{14}\\
&b_0 \cdot 16^{31} + b_1 \cdot 16^{30} + \ldots + b_{31} \cdot 16^0 = S_{15}\\
&\vdots\\
&b_0 \cdot 32^{31} + b_1 \cdot 32^{30} + \ldots + b_{31} \cdot 32^0 = S_{31}
\end{aligned}$$

Fig. 1. Root substitution in a vector form.

In this case, the byte from the message is a scalar (yellow rectangle) and known roots (magenta rectangle) are stored as vectors in exponentiation LUT. That pre-calculated data allows multiplication of the vector-to-scalar type (cyan rectangle) using one additional load from RAM for each multiplication of one vector. Although this LUT turns out to be quite large (64KB), it gives a significant performance gain.

Exponentiation LUT contains 255 elements because the maximum power of the message polynomial is 255. Each element is an array of 256 values with all possible roots, raised to powers from 0 to 255. Exponentiation LUT can be represented in the form of Table 5 (2 elements are shown):

TABLE 5: EXPONENTIATION LUT

LUT index	256-bytes element				
0…256	1^0	1^1	1^2	…	1^{255}
23552…23807	92^0	92^1	92^2	…	92^{255}

It should be noted that for the scalar version such a table would be useless. On the one hand, it would be possible to save one lookup on the multiplication table. On the other hand, due to the relatively large size of exponentiation LUT, data requests are much less likely to end up in a fast CPU cache. As a result, for a scalar version performance with exponentiation LUT would be lower.

E. Proposed dynamic selection scheme of subroutine

We have developed two versions of the vectorized algorithm, using SSSE3 and AVX2 instruction sets respectively. The AVX2 version is similar to the SSSE3 version, but it is not supported by older CPUs and does not bring a performance increase for RS codes with a low error correction capability.

The algorithm of subroutine version selection for a given code RS(n, k) is described below.

if CPU supports AVX2 **and** $n - k > 16$ **then**
  select AVX2 subroutine
else if $n - k \leq 16$ **or** CPU supports SSSE3 **then**
  select SSSE3 subroutine
else
  select scalar subroutine

A specific limitation for AVX2 version $n - k > 16$ is based on the fact that this instruction set uses 32-byte vectors. Therefore, there is no point in using it for RS codes that cannot fill more than half of the register size. As will be shown in testing results, using the AVX2 subroutine to decode such RS codes drops performance in comparison with the SSSE3 subroutine.

III. Experimental Setup

A. Test Environment

All tests were performed on a computer with the following characteristics:

CPU: Intel® Xeon® 1660v4 @ 4GHz
RAM: 32Gb DDR4-2666 QC
Compiler: MSVC v142
OS: Windows 10 Pro 1909

B. Test suite

The following RS codes were tested: RS (48, 32), RS (64,32), RS (96, 32), RS (128, 32).

The performance will be compared between the following subroutines: JPWL – implementation from Open JPEG library; scalar – version of our library that uses only scalar instructions; SSSE3 and AVX2 – versions of our library that use instructions from the respective sets.

C. Test application

For testing purposes, a C++ console application was developed in MS VS 2019 IDE. The testing algorithm is described below.

for each subroutine **and** RS code **do**
  encode data
  measure encode time
  insert errors into data
  decode corrupted data
  measure decode time
end for

IV. Test Results

The test file and the RS code are used as input data for the operation of the test application. The test file is loaded into a RAM buffer, and two additional buffers are created: to store the encoded data and the result of subsequent decoding.

Let the processing speed of the JPWL codec be v_j and the processing speed of the codec under test be v^*. The absolute value of the coding rate in MB/s is not very informative, since it is several times higher for RS (48, 32) than for RS (128, 32). It also significantly depends on the CPU clock speed. Based on these considerations, the test results were normalized. The normalized result of the codec under test is defined as $v_{norm} = v^* / v_j$.

During the testing phase, the same set of tests is run sequentially for all codecs. The execution time of each test operation is measured using system function calls. A 9.9MB image was used as a test file for each experiment.

A. Encoding of data

The first test is the encoding of the source file. The input buffer is represented as sequentially written data blocks, which are encoded and sequentially written to the buffer for the encoded data. Results for this test are represented in Fig. 2. The vertical axis displays a normalized speed (higher is better).

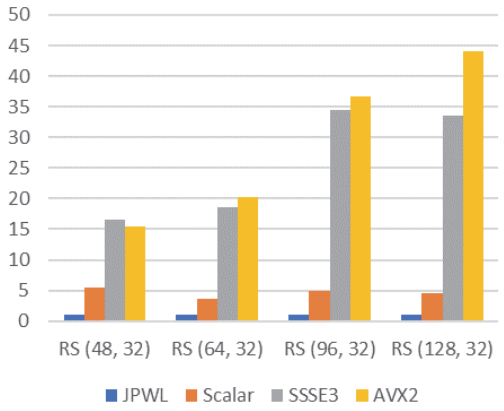

Fig. 2. Comparison of the data encoding speed.

In absolute values, JPWL codec spends 0.87s to encode a buffer with RS code (48, 32) and 2.86s to encode a buffer with RS code (128, 32).

The scalar subroutine shows a nearly constant gain in performance against JPWL for all codes, 4.6x on average. With such an improvement, encoding with RS code (128, 32) takes only 0.64s.

Vectorized subroutines show a higher gain for codes (96, 32) and (128, 32), and 25.7x on average for the SSSE3 version. Codec spends about 0.1s for any RS code from the test suite. We can see that there is almost no difference between AVX2 and SSSE3 versions. These versions are using identical computational algorithm, but instruction latencies in AVX2 are larger.

B. Decoding of data

The second test is to decode the buffer containing errors. To insert errors into the buffer, a pseudo-random number generator (PRNG) is used. The buffer is represented as a set of blocks. For each block, the number of errors is generated in the range [0; t], where t is the error correction bound of the RS code under test. Therefore, the average number of errors will be equal to $t/2$. Then, random positions of errors in the block and non-zero error values are obtained from PRNG. Error-values are superimposed on the original data by the XOR

Original scientific paper

operation. Since randomly generated error positions within a block are not necessarily distinct, the number of errors generated may be less than the specified one. However, for a large number of blocks, the average number of errors in a block is still very close to $t/2$. The buffer with errors is fed to the decoder, which writes the data to the buffer for the decoding result. Results for this test are represented in Fig. 3 (higher is better).

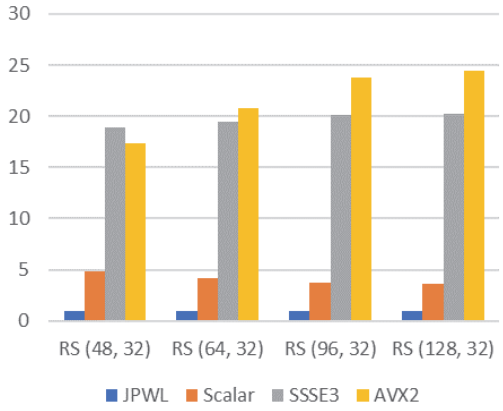

Fig. 3. Comparison of the corrupted data decoding speed.

In absolute values, JPWL codec spends 1.82s to decode the buffer with RS code (48, 32) and 21.1s to encode the buffer with RS code (128, 32).

The scalar subroutine shows a slowly decreasing gain for stronger codes, 4.1x on average. To decode RS (128, 32), codec spends 5.91s, which is still large.

Vectorized versions show a nearly constant gain for all codes, 19.7x on average for the SSSE3 version. Even for the strongest RS code, measured decoding time was about 1.04s. AVX2 version is slightly faster in this case with 0.86s. Decoding RS (48, 32) takes only about 0.1s.

V. Conclusions

In this work, the performance of our three codecs for RS codes were compared to the codec from the Open JPEG JPWL library.

Experimental results indicate that our scalar codec is advantageous for both encoding and decoding RS codes. An obtained gain in performance can significantly reduce CPU load and, respectively, can lower power consumption. Our SSSE3 subroutine is even faster, and, if it is supported on a target system, will reduce CPU load by almost 95%. As for the AVX2 version, the performance comparison against the SSSE3 shows that we have gained only a 12% advantage for data encoding and 9% for decoding. This is so because latencies for the AVX2 instructions are higher [22]. In addition, RS (48, 32) code shows a decrease in performance when switched from SSSE3 to AVX2, since calculations for it can fill only a half of 32-byte registers.

This work is the first step towards a video streaming system for lossy wireless networks. The next stage should include benchmarking of the complete system to assess the influence of RS codes on performance. Finally, the proposed library should be evaluated on actual field measurements.

References

[1] T.K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms,* 1edn, Wiley-Interscience, 2005.

[2] *Information technology – JPEG 2000 image coding system: Wireless*, ITU T.810, 2006.

[3] *Official repository of the Open JPEG project*. Available: https://github.com/uclouvain/openjpeg.

[4] N.-S. Vo, T.Q. Duong, H.D. Tuan, and A. Kortun, "Optimal Video Streaming in Dense 5G Networks with D2D Communications," *IEEE Access*, vol. 6, pp. 209–223, 2018.

[5] S.B. Sadkhan, "Performance Evaluation of Concatenated Codes applied in Wireless Channels," *NICST 2019 – 1st Al-Noor International Conf. for Science and Technology*, pp. 89–93, Oct. 2019.

[6] S.B. Balaji, M.N. Krishnan, M. Vajha, V. Ramkumar, B. Sasidharan, and P.V. Kumar, "Erasure coding for distributed storage: an overview," *Science China Information Sciences*, vol. 61, Issue 101, Oct. 2018.

[7] Z. Shen, P.P.C. Lee, J. Shu, and W. Guo, "Encoding-aware data placement for efficient degraded reads in XOR-coded storage systems: Algorithms and evaluation," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, Issue 12, pp. 2757–2770, Dec. 2018.

[8] M. Vajha, V. Ramkumar and B. Puranik, "Clay codes: Molding MDS codes to yield an MSR code," *Proc. of the 16th USENIX Conference on File and Storage Technologies*, pp. 139–153, Feb. 2018.

[9] V. Guruswami and M. Wootters, "Repairing Reed-Solomon Codes," *IEEE Transactions on Information Theory*, vol. 63, Issue 9, pp. 5684–5698, Sep. 2017.

[10] H. Dau, I.M. Duursma, H.M. Kiah, and O. Milenkovic, "Repairing Reed-Solomon Codes with Multiple Erasures," *IEEE Transactions on Information Theory*, vol. 64, Issue 10, pp. 6567–6582, Oct. 2018.

[11] K. Lee, C. Suh, and K. Ramchandran, "High-dimensional coded matrix multiplication," *IEEE International Symposium on Information Theory – Proc.*, pp. 2418–2422, Aug. 2017.

[12] R. Freij-Hollanti, O.W. Gnilke, C. Hollanti, and D.A. Karpuk, "Private information retrieval from coded databases with colluding servers," *SIAM Journal on Applied Algebra and Geometry*, vol. 1, Issue 1, pp. 647–664, 2017.

[13] S. Puchinger and J. Rosenkilde Ne Nielsen, "Decoding of interleaved Reed-Solomon codes using improved power decoding," *IEEE International Symposium on Information Theory – Proc.*, pp. 356–360, Aug. 2017.

[14] L. Yu, Z. Lin, S.-J. Lin, Y.S. Han, and N. Yu, "Fast Encoding Algorithms for Reed-Solomon Codes with between Four and Seven Parity Symbols," *IEEE Transactions on Computers*, vol. 69, Issue 5, pp. 699–705, May 2020.

[15] R. Bhaskar, P.K. Dubey, V. Kumar, A. Rudra, and A. Sharma, "Efficient Galois field arithmetic on SIMD architectures," *Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 256–257, 2003.

[16] J.S. Plank, K.M. Greenan, and E.L. Miller, "Screaming Fast Galois Field Arithmetic Using Intel SIMD Instructions," *11th USENIX Conference on File and Storage Technologies (FAST '13)*, pp. 299–306, 2013.

[17] N. Drucker, S. Gueron and V. Krasnov, "The Comeback of Reed Solomon Codes," *2018 IEEE 25th Symposium on Computer Arithmetic (ARITH)*, 2018, pp. 125-129, doi: 10.1109/ARITH.2018.8464690.

[18] *Intel Architecture Instruction Set Extensions Programming Reference*, Intel, October 2017. Available: https://softwareintel.com/sites/default/files/managed/c5/15/architectureinstruction-set-extensions-programming-reference.pdf.

[19] M. Sudan, "Decoding of Reed Solomon codes beyond the error-correction bound", *Journal of Complexity*, vol. 13, Issue 1, pp. 180–193, Mar. 1997.

[20] *Intel® Intrinsics guide*, Intel. Available: https://software.intel.com/sites/landingpage/IntrinsicsGuide/.

[21] C.K.P. Clarke, "Reed-Solomon Error Correction," *BBC Research & Development White Paper* WHP 031, 2002.

[22] F. Agner, "List of instruction latencies," The Technical University of Denmark, Mar. 2021. Available: https://www.agner.org/optimize/instruction_tables.pdf.