Hindawi
Applied Computational Intelligence and Soft Computing
Volume 2023, Article ID 5573438, 12 pages
https://doi.org/10.1155/2023/5573438

Research Article

Embedded Parallel Implementation of LDPC Decoder for Ultra-Reliable Low-Latency Communications

Mhammed Benhayoun,[1] **Mouhcine Razi**,[1] **Anas Mansouri**,[2] **and Ali Ahaitouf**[1]

[1]*University Sidi Mohammed Ben Abdellah, Faculty of Sciences and Technology, Laboratory of Intelligent Systems, Georesources and Renewable Energies, P.O. Box 2202, Fez, Morocco*
[2]*University Sidi Mohammed Ben Abdellah, National School of Applied Sciences, Laboratory of Intelligent Systems, Georesources and Renewable Energies, Fez, Morocco*

Correspondence should be addressed to Ali Ahaitouf; ali.ahaitouf@usmba.ac.ma

Received 26 May 2023; Revised 22 September 2023; Accepted 10 October 2023; Published 21 October 2023

Academic Editor: Kalapraveen Bagadi

Ultra-reliable low-latency communications, URLLC, are designed for applications such as self-driving cars and telesurgery requiring a response in milliseconds and are very sensitive to transmission errors. To match the computational complexity of LDPC decoding algorithms to URLLC applications on IoT devices having very limited computational resources, this paper presents a new parallel and low-latency software implementation of the LDPC decoder. First, a decoding algorithm optimization and a compact data structure are proposed. Next, a parallel software implementation is performed on ARM multicore platforms in order to evaluate the latency of the proposed optimization. The synthesis results highlight a reduction in the memory size requirement by 50% and a three-time speedup in terms of processing time when compared to previous software decoder implementations. The reached decoding latency on the parallel processing platform is 150 μs for 288 bits with a bit error ratio of 3.410^{-9}.

1. Introduction

Initially introduced by Gallager in 1962 [1] and then reworked by Mackay and Neal in 1996 [2], low density parity check (LDPC) codes have recently been used in several wireless standards such as WiMax (IEEE 802.16e), WiFi (IEEE 802.11n), 5G New Radio (NR), and DVB-S2.

LDPCs are linear block codes defined by a sparse binary parity check matrix H. They are typically represented by bipartite graphs formed by variable nodes and check nodes connected by bidirectional edges, also called Tanner graphs [3]. LDPC codes have attracted considerable attention because of their superior error correction capability based on the iterative log-likelihood-ratio belief propagation algorithm (LLR BP) [4].

However, LDPC codes also have the disadvantage of high decoding complexity, which makes it significantly challenging to meet the requirements for low latency in communication systems. For example, the latency of the ultra-reliable low-latency communications needs to achieve 32Octect in less than 1 ms with a bit error ratio at 10^{-5} [5].

There are two approaches to implement the LDPC decoder. The first one is hardware based on application-specific integrated circuits (ASICs) or field-programmable gate array (FPGA) circuits. This approach not only achieves low latency and high throughput [6–9] but also brings a high development cost. This is a limitation for applications requiring fast time-to-market or for technologies with multiple and fast-evolving standards.

The second approach is the software implementation based on coding the algorithm in a programming language and compiling it into a binary program that is loaded into the memory of the target architecture and then executed by its processor. The software solution considerably reduces the hardware resources and development time necessary for the deployment and allows modifying and updating functionalities by uploading and running a new software version without changing the hardware.

Several studies have recently focused on software implementations of LDPC decoders on multicore devices, using three soft implementation strategies for parallel processing: some works manipulate GPUs or x86 multicore to parallelize processing [10, 11], others use SIMD architecture [12] to accelerate computation, and the third exploits multicore system on chip (SoC) to take advantage of hard acceleration and soft flexibility [13, 14].

In [10], the comparison of parallelization strategies for the min-sum decoding algorithm of irregular LDPC codes has demonstrated that the GPU can achieve decoding with higher throughputs than a general processor. However, a GPU-based solution works well only for the large code length because of the time spent for the data transfer between the host and GPU. For medium or small code length, the data transfer times are revealed to be higher than the needed time for the decoding process.

In [12], we proposed a multi-Gbps LDPC decoder on a GPU, using single-instruction multiple data to parallelize the processing of many received packets. This approach helps to achieve higher throughputs on embedded mobile devices. However, the spent times for data transfer between the host and GPU device are still higher for low-latency applications.

In [13], Kharin et al. performed an implementation on 8 DSP- and 4 ARM-cores multicore system on chip (SoC). This solution takes advantage of the DSP flexibility and efficiency in signal processing tasks, but the interprocessor communication framework for ARM-DSP cooperative functionality takes about 2.75 ms for the DSP data loading, consequently increasing the latency decoding.

Therefore, our choice is oriented towards a software solution based on a new proposed algorithm optimization of the LDPC decoding process which reduces the computation complexity and, consequently, the decoding latency. This choice is also motivated by the emergence of general-purpose processors with high computing power and multicore embedded systems that allow very powerful parallel calculations, which allows for the near-performance of the hard solutions with a reduction in cost and an adaptation to the various application contexts.

The rest of the paper is organized as follows. Section 2 reviews the LDPC codes, the LLR BP, the min-sum (MSA), and the normalized min-sum (NMSA) algorithms. In Section 3, the proposed optimization algorithm and data structure are introduced, the complexity of the proposed algorithm in terms of the number of computational operation is discussed, and the parallel computational model is presented. Section 4 is dedicated to the simulation results in terms of CPU run time, the error-correcting performances, and the latency of the parallel software implementation on x86 and ARM multicore platforms. Section 5 concludes the paper.

2. Basic Decoding LDPC Algorithms

2.1. LDPC Codes. A binary LDPC channel code is a linear (N, K) block code used to correct transmission errors. From the transmitter, the coding generates an N-bit code word from a K-bit information message with the addition of $M = N - K$ parity bits. The decoding procession used a binary sparse parity-check matrix $\mathbf{H}$ of seize M x N; if $\mathbf{x}$ is a valid code-word vector, then $\mathbf{H} \cdot \mathbf{x}^T = 0$, where $\mathbf{x}^T$ is the transposed of the code-word vector and "$\cdot$" represents the matrix multiplication modulo 2.

If the parity check matrix contains the same number of ones per row (noted d_c), and the same number of ones per column (noted d_v), the code is called a regular LDPC code. Otherwise, the code is called irregular code. In this paper, we limited our focus to irregular codes for their good convergence [15] in terms of bit error ratio (BER) related to signal to noise ratio (SNR).

An example of the $\mathbf{H}$ matrix with $N = 12$ variable nodes v_j and $M = 9$ control nodes c_i is shown in Figure 1(b). This matrix can be represented by a Tanner graph (Figure 1(b)) containing M check nodes (c_i), N variable nodes (v_j), and E bidirectional edges connecting the check node c_i to the variable node v_j when the value the matrix element H_{ij} is 1.

The structured position of the nonzero elements in the parity check matrix $\mathbf{H}$ allows for a reduction in the LDPC encoding complexity. Quasi-cyclic LDPC codes (QC-LDPC) are a class of structured codes that have a good error correction performance [17, 18]. For these codes, the H matrix is composed by a set of $Z \times Z$ submatrices, where Z is called the expansion factor; each $Z \times Z$ submatrix is obtained by the right circular permutation of the $Z \times Z$ identity matrix. The permutation value defines the different submatrices. These values are called shift coefficients, and the set of shift coefficients is then collected in a matrix called the expansion matrix noted $\mathbf{H_{bm}}$.

For a $\mathbf{H}$ matrix of size $M \times N$, the related expansion matrix $\mathbf{H_{bm}}$ size is $m \times n$, where $m = M/Z$ and $n = N/Z$. The expansion matrix $\mathbf{H_{bm}}$ is expanded to the $\mathbf{H}$ matrix by replacing each negative shift coefficient with a $Z \times Z$ all zeros matrix, each zero shift coefficient with a $Z \times Z$ identity matrix, and each positive shift coefficient with a right circular permutation $Z \times Z$ identity matrix.

Figure 2 shows an example of the $\mathbf{H_{bm}}$ expansion matrix of size (24, 12) and expansion factor $Z = 96$, expanded to the WiMAX $\mathbf{H}$ matrix of size (2304, 1152). Each shift coefficient is replaced by the right circular permutation of 96×96 identity matrices; the "0" elements are replaced by a 96×96 identity matrix, and the "-1" elements are replaced by 96×96 all-zero matrices.

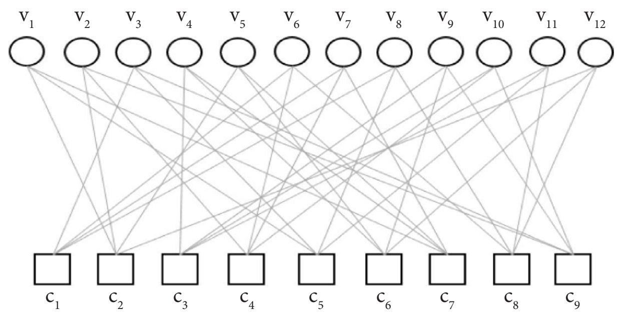
2.2. LLR BP Decoding for LDPC Codes. The LLR BP decoding is based on the belief propagation of the log-likelihood ratio (LLR) messages between connected nodes. The LLR value is used to evaluate the ratio between the probabilities of a binary random variable to be 0 or 1.

The C2V messages $m_{c_i \longrightarrow v_j}$ propagated from c_i to v_j are initialized at zero and the V2C messages $m_{v_j \longrightarrow c_i}$ propagated from v_j to c_i are initialized as follows:

$$m(0)_{v_j \longrightarrow c_i} = L(0)_{v_j} = \log\left(\frac{p(y_j/v_j = 0)}{p(y_j/v_j = 1)}\right), \quad (1)$$

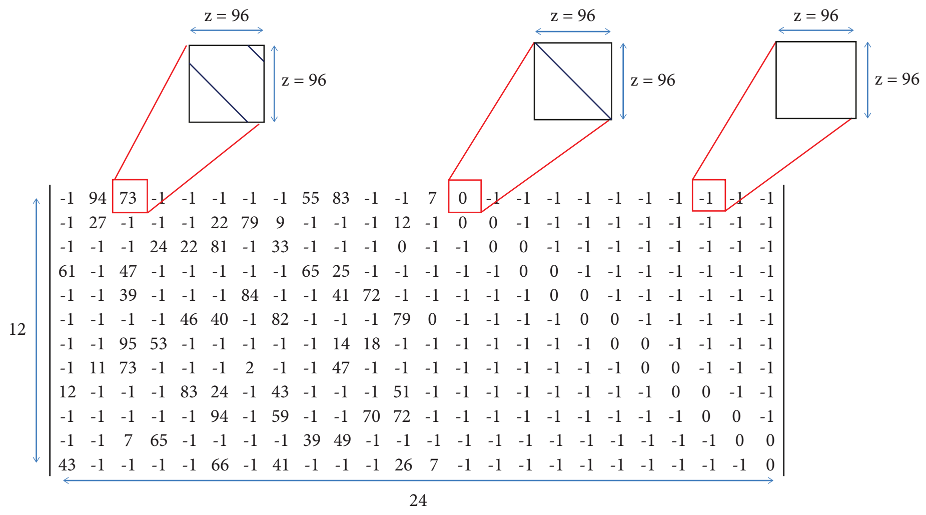
$$\begin{array}{c|cccccccccccc|}
 & v_1 & v_2 & v_3 & v_4 & v_5 & v_6 & v_7 & v_8 & v_9 & v_{10} & v_{11} & v_{12} \\
c_1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
c_2 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
c_3 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 \\
c_4 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \\
c_5 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\
c_6 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
c_7 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
c_8 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 \\
c_9 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0
\end{array}$$

(a)



(b)

FIGURE 1: (a) **H** matrix with $N = 12$ variable nodes v_j and $M = 9$ control nodes c_i. (b) Corresponding Tanner graph. Figure 1 is reproduced from Benhayoun et al. [16] (under the creative commons attribution license/public domain).



$$\begin{array}{|cccccccccccccccccccccccc|}
-1 & 94 & \boxed{73} & -1 & -1 & -1 & -1 & -1 & 55 & 83 & -1 & -1 & 7 & \boxed{0} & -1 & -1 & -1 & -1 & -1 & -1 & -1 & \boxed{-1} & -1 & -1 \\
-1 & 27 & -1 & -1 & -1 & 22 & 79 & 9 & -1 & -1 & -1 & 12 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
-1 & -1 & -1 & 24 & 22 & 81 & -1 & 33 & -1 & -1 & -1 & 0 & -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
61 & -1 & 47 & -1 & -1 & -1 & -1 & -1 & 65 & 25 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 & -1 \\
-1 & -1 & 39 & -1 & -1 & -1 & 84 & -1 & -1 & 41 & 72 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 & -1 \\
-1 & -1 & -1 & -1 & 46 & 40 & -1 & 82 & -1 & -1 & -1 & 79 & 0 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 & -1 \\
-1 & -1 & 95 & 53 & -1 & -1 & -1 & -1 & -1 & 14 & 18 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 & -1 \\
-1 & 11 & 73 & -1 & -1 & -1 & 2 & -1 & -1 & 47 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 & -1 \\
12 & -1 & -1 & -1 & 83 & 24 & -1 & 43 & -1 & -1 & -1 & 51 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 & -1 \\
-1 & -1 & -1 & -1 & -1 & 94 & -1 & 59 & -1 & -1 & 70 & 72 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 & -1 \\
-1 & -1 & 7 & 65 & -1 & -1 & -1 & -1 & 39 & 49 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0 & 0 \\
43 & -1 & -1 & -1 & -1 & 66 & -1 & 41 & -1 & -1 & -1 & 26 & 7 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & -1 & 0
\end{array}$$

FIGURE 2: $\mathbf{H_{bm}}$ Wimax expansion matrix, with $n = 24$, $m = 12$, and $Z = 96$.

where y_j denotes the channel-information of the j-th variable node, v_j denotes the j-th code word bit, and L and p denote, respectively, the LLR value and the conditional probability.

After initialization, each iteration is mainly described by horizontal and vertical intensive processing blocks. In the horizontal processing, the algorithm updates the messages propagated from each check node to each variable node. The updated C2V messages $m_{c_i \longrightarrow v_j}$ are generated according to

$$m_{c_i \longrightarrow v_j} = 2\tanh^{-1}\left(\prod_{vb \in N(c_i) \backslash v_j} \tanh\left(\frac{m_{v_b \longrightarrow c_i}}{2}\right)\right), \quad (2)$$

where $N(c_i) \backslash v_j$ denotes the neighboring variable nodes that are connected to check node c_i, excluding variable node v_j.

In vertical processing, the updated V2C messages $m_{v_j \longrightarrow c_i}$ are calculated as follows:

$$m_{v_j \longrightarrow c_i} = L(0)_{v_j} + \sum_{ca \in N(v_j) \backslash c_i} m_{c_a \longrightarrow v_j}, \quad (3)$$

where $N(v_j) \backslash c_i$ denotes the neighbors check nodes that are connected to variable node v_j, except for the check node c_i.

After the generation and propagation of all the updated V2C and C2V messages, a hard decision on the variable node v_j is made, based on the new LLR update $L(v_j)$ calculated as follows:

$$L(v_j) = L(0)_{v_j} + \sum_{ca \in N(v_j)} m_{c_a \longrightarrow v_j}, \quad (4)$$

where $N(v_j)$ denotes all the neighbors check nodes that are connected to the variable node v_j.

The iterative decoding process will not stop until all the check equations are satisfied, i.e., $\mathbf{H} \cdot \mathbf{v}^T = 0$, or the predefined maximum number of iterations is reached.

The decoding complexity can be significantly reduced thanks to various algorithms available for C2V messages $\mathrm{m}_{c_i \longrightarrow v_j}$ updates simplification. The widely used ones, in the recent works, are min-sum (MSA) and normalized min-sum (NMSA) algorithms [19–21]. For the MSA algorithm, the update equation became

$$m_{c_i \longrightarrow v_j} = \prod_{n' \in N(m)\backslash n} \operatorname{sign}\left(m_{v_b \longrightarrow c_i}\right) \min_{n' \in N(m)\backslash n} \left(\left|m_{v_b \longrightarrow c_i}\right|\right). \tag{5}$$

For the NMSA algorithm, this value is normalized by a factor α, where $\alpha < 1$:

$$m_{c_i \longrightarrow v_j} = \alpha \prod_{n' \in N(m)\backslash n} \operatorname{sign}\left(m_{v_b \longrightarrow c_i}\right) \min_{n' \in N(m)\backslash n} \left(\left|m_{v_b \longrightarrow c_i}\right|\right). \tag{6}$$

These two algorithms are mainly based on the determination of the first and second minimum of the modules of the V2C messages noted as min1 and min2. The normalized min-sum algorithm (NMSA) improves the correction performance compared to the MSA approximation.

LLR-BP decoding is typically performed by repeating the flooding schedule, where all check-to-variable messages (C2V) are updated in the horizontal step, and subsequently all variable-to-check messages (V2C) are updated in the vertical step.

However, the convergence process is slowed down as the latest updated information, in the current iteration, must be used in the next iteration. To speed up convergence and increase error correction performance, sequential scheduling methods have been proposed, with both a predetermined and fixed sequence of updates. This sequential scheduling strategy differs from flooding in that the last updated information is used in the current iteration. Shuffled [22, 23] and dynamic [16, 24] scheduling are two variants of this strategy and allow decoding convergence to accelerate twice as fast as flooding scheduling.

In order to have a better BER convergence performance, the NMSA algorithm is associated with shuffled scheduling that allows accelerating decoding convergence. Algorithm 1 depicts the pseudocode of this association. Initialization of C2V and V2C is carried out (line 1) according to equation (1), and then the maximum number of iterations is set (line 2). The C2V computation is performed in the vertical processing according to equation (6), followed by the V2C and the new LLR calculation in the vertical processing according to equations (3) and (4). The hard decision is made based on the new LLR update (line 22). If the sign of the LLR value is positive, then the code-word bit is set to 1, else it is set to 0. Once the estimated code word is obtained, the syndrome is executed to evaluate if a valid code-word is found (line 24); otherwise, a new decoding iteration is started (line 2), and the iterative decoding process will not stop until the valid code-word is found or the predefined maximum number of iterations is reached.

3. Proposed Parallel Software Implementation

Before going to throw, a profiling procedure using the Valgrind [25] profiler is first executed to determine the total execution CPU time for each block of the LDPC decoder. Blocks that require more computing time are then identified and considered as more suitable candidates for eventual optimizations.

3.1. Results of the Profiling Analysis.

3.1. Results of the Profiling Analysis. In our case, the NMSA HS algorithm is implemented in the C language, profiled with Vilgrand, and tested on the quasi-cyclic irregular LDPC codes (576, 288), (1152, 576), (2304, 1152), (4608, 2304), and (9216, 4608) constructed based on the IEEE 802.16e standard under the white Gaussian noise channel (AWGN) and the BPSK modulation.

The memory size requirement and the run time spent in the write/read memory accesses depend on the adopted data structure for both the **H** matrix and message storage. The data structure used in previous works [10–14] is used in this run-time analysis. In this representation, the **H** matrix is represented as two separate two-dimensional arrays: the first contains the column indexes of each matrix row, which is used for the horizontal processing, and the second contains the row indexes of each matrix column, which is used for the vertical processing. Some values in the two-dimensional arrays are set at −1 because these arrays represent an irregular LDPC code, which has different column weights and row weights. Therefore, the size of the first two-dimensional array is $M \times d_{\text{cmax}}$ and the size of the second one is $N \times d_{\text{vmax}}$, where d_{cmax} and d_{vmax} are respectively the maximum value of the number of ones per row d_c and the maximum value of the number of ones per column d_v.

For the message storage, two arrays of size E are used for the C2V and V2C message updates, and two other arrays of size N are used for initial and updated LLR values.

Therefore, the memory size required for this data structure is calculated as blow (equation (7)). The first part of the equation concerns the **H** matrix storage, and the second part concerns the message storage:

$$\text{memory size} = \left(M \times d_{\text{cmax}} + N \times d_{\text{vmax}}\right) \times \text{size of (interger)} + 2 \times (E + N) \times \text{size of (float)}, \tag{7}$$

where E is the total number of edges in the entire Tanner graph, N is the number of the variable node, M is the number of the check node, and sizeof() refers to the operator which gives the amount of storage, in bytes, required to store an object of the type of the operand.

Table 1 reports the overall CPU run-time in cycles spent in different algorithm blocks depicted in Algorithm 1. Table 2 reports the same time spent by arithmetic operations, data memory accesses in read/write mode, and the cycles spent in searching data outcomes at level

```
(1) Initialize all $m_{c_i \longrightarrow v_j} = 0$, $m_{v_j \longrightarrow c_i} = L(0)_{v_j}$ and Itermax
(2) for $i = 1$ to Itermax do
    Horizontal processing (C2V computation):
(3)   for each check node $c_i$
(4)     for each variable node $v_j$ connected to $c_i$
(5)         Calculate $\min_1$ & $\min_2$
(6)     end for line 4
(7)     for each variable node $v_j$ connected to $c_i$
(8)   $m_{c_i \longrightarrow v_j} = \alpha \prod_{n' \in N(m)\backslash n} \text{sign}(m_{v_b \longrightarrow c_i}) \min 1 \text{ or } \min 2$
(9)     end for line 7
    Vertical processing (V2C computation) and LLR update:
(11)    for each variable node $v_j$ connected to $c_i$
(12)      for every check node $c_a$ connected to $v_j$
(13)        tmp = tmp + $m_{c_a \longrightarrow v_j}$
(14)      end for line 12
(15)      $L(v_j) = L(0)_{v_j}$ + tmp
(16)      for every check node $c_a$ connected to $v_j$
(17)        $m_{v_j \longrightarrow c_i} = L(v_j) - m_{c_i \longrightarrow v_j}$
(18)      end for line 16
(19)    end for line 11
(20)  end for line 3
    Hard decision:
(21)  for each variable node
(22)  Make hard decision if $\text{sign}(L(v_j)) > 0$ then $v_j = 1$; else $v_j = 0$
(23)  end for line 19
    Parity check equations (Syndrome) and stopping criteria:
(24)  if $\mathbf{H} \cdot \mathbf{v}^T = 0$ then break else $i = i + 1$
(25) end for line 2
```

Algorithm 1: Horizontal shuffle NMSA.

Table 1: CPU run-time in cycles spent by the algorithm block.

Code size	V2C computation	C2V computation	Decision	Initialization	Syndrome
(9216, 4608)	7 001 278	2 959 459	173 541	125 181	253 439
(4608, 2304)	3 491 957	1 472 692	120 639	51 508	115 542
(2304, 1152)	1 725 562	739 557	43 557	32 895	72 600
(1152, 576)	862 872	368 852	23 929	15 534	35 676
(576, 288)	430 294	185 021	9 671	9 534	16 382

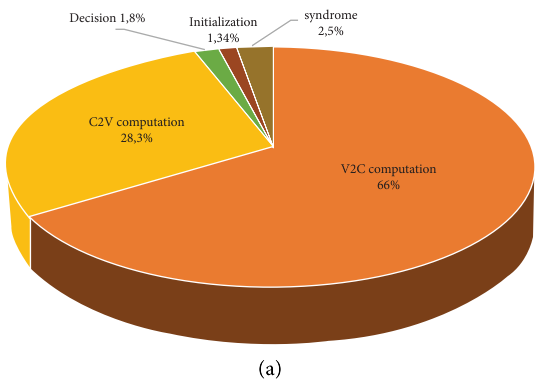
Table 2: CPU run-time in cycles spent by instruction and memory accesses.

Code sizes	Total cycles	Instruction cycles	Data read memory accesses	Data write memory accesses	L1 memory cache miss	Last level memory cache miss
(9216, 4608)	105 128 98	3 727 900	5 137 660	1 394 103	24 275	105
(4608, 2304)	5 252 338	1 866 554	2 563 633	695 571	12 136	52
(2304, 1152)	2 614 171	928 109	1 276 110	346 712	6 064	26
(1152, 576)	1 306 863	465 771	635 694	173 865	3 023	13
(576, 288)	650 901	231 316	317 418	86 468	1 505	6

one (L1 miss) and the last level (LL miss) of the memory cache.

The profiling results of Tables 1 and 2 are reported in Figure 3, in terms of the CPU percentage cycles. Figure 3(a) illustrates that the V2C message updates block is the most time-consuming module (66%) followed by the C2V message update block which takes 28.3% of the execution time. The higher percentage of run-time taken by the V2C block is justified by the fact that the number of V2C updating ($M \times d_c \times d_v = d_v$. E update) higher than that of the C2V updates ($M \times d_c = E$ update).

Figure 3(b) shows that 62% of the CPU time is spent on memory access (49% for the data read and 13% for the data write), 36% for instructions, and only 2% for memory access out of the cache memory. The higher percentage of access memory can be justified by the separated processing of



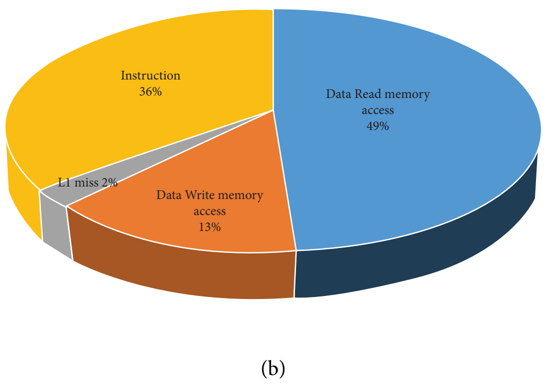


FIGURE 3: (a) CPU percentage cycles spent by code block. (b) CPU percentage cycles spent by memory access and instructions.

horizontal and vertical stages; in fact, for the same data, the memory access is done in read mode in the horizontal stage using the column-mapped matrix table and in write mode in the vertical step using the row-mapped matrix table, and vice-versa.

According to these analysis results, the V2C computation bloc is chosen to be optimized in order to decrease the access memory and instruction required in the decoding process.

Unlike the previous works [10–14], in this paper we propose an optimized version of the NMSA horizontal shuffled scheduling algorithm by merging the horizontal and vertical steps into one step, thereby allowing to decrease the memory accesses and minimize the number of arithmetic operations.

We also propose a compacted data structure to represent the **H** matrix that is organized by the processing order and that is suitable for parallel implementation in order to take advantage of the multicore platforms for lower decoding latency.

3.2. Proposed Optimization.

3.2. Proposed Optimization. Separate data processing increases the memory access for the same data. In order to solve this problem, an optimized algorithm is proposed to compute all the message updates (C2V, V2C, and $L(v_j)$) corresponding to the data overloaded in the CPU processor in the same step before loading the next data. This allows taking advantage of the temporal locality of the cache memory and makes all computations on the current CPU-fetched data.

From equations (3) and (4), we can observe that the initial LLR value of the variable node and the current iteration's V2C message value are both stored in the current iteration's LLR value, so we can calculate the V2C message value from the difference between the variable node's LLR value and the connected C2V message according to the equation as follows:

$$m_{v_j \longrightarrow c_i} = L(v_j) - m_{c_i \longrightarrow v_j}. \tag{8}$$

Also, as presented in Figure 4, once one C2V message is updated, the calculation of the $L(v_j)$ can be directly done by replacing the old C2V value with the new one. The difference between the old LLR value and the old C2V is equal to the old V2C value (equation (7)), so the updated $L(v_j)$ is calculated according to the following equation:

$$L(v_j)^{\text{new}} = m_{v_j \longrightarrow c_i}{}^{\text{old}} + m_{c_a \longrightarrow v_j}{}^{\text{new}}. \tag{9}$$

Algorithm 2 depicts the pseudocode of the proposed optimization. Initialization of C2V and V2C is carried out (line 1) according to equation (1), and then for each check node, the processor uploads the old C2V and LLR values of each variable node connected to the check node (lines 3 to 4). The min1 and min2 values are calculated from the difference between $L(v_j)$ and C2V uploaded (lines 5 and 6), then the C2V and $L(v_j)$ messages update are calculated in the same loop (lines 9, 10, 11) before passing to the next check node. The hard decision and syndrome blocks are the same as shown in Algorithm 1.

3.3. Proposed Data Structure. The proposed data structure is generated by scanning the **H** matrix in a row-major order and by sequentially mapping the column index associated with nonzero elements in the **H** matrix. These column indexes are collected and stored in consecutive memory positions inside the table of size E, noted **Col**.

In order to take advantage of the spatial locality of the memory cache, the C2V messages are also mapped in memory in a row-major order in consecutive memory positions. In this way, each element of the **Col** table records the address of the corresponding C2V value. Using the proposed algorithm, the V2C messages are not memorized because they will be calculated in the decoding process using equation (8).

Figure 5 shows the different arrays required for the proposed algorithm using, as an example, the matrix shown in Figure 1. The access memory to the C2V and **Col** tables is made directly because the C2V messages and column indexes are stored in the memory following the execution order (row-major order); the access memory to the $\mathbf{L}(\mathbf{v_j})$ message is done towards the **Col** table. For example, the check node c1 contains 4 nonzero elements: the first one is connected to the third variable node (**Col[1] = 3**), the second one is connected to the sixth variable node (**Col[2] = 6**), the third one is connected to the seventh variable node (**Col[3] =**

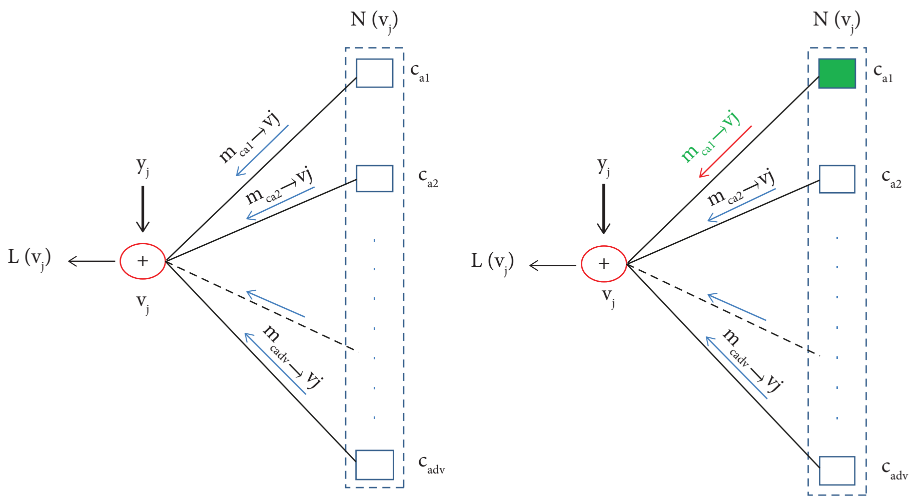


Figure 4: LLR value update.

```
(1)  Initialize all $m_{c_i \longrightarrow v_j} = 0$, and $L(v_j) = y_j$ Itermax
(2)  for $i = 1$ to Itermax do
     Horizontal processing:
(3)    for each check node $c_i$
(4)      for each variable node $vj$ connected to ci
(5)          Calculate $v2c = L(v_j) - m_{c_i \longrightarrow v_j}$
(6)          Calculate $\min_1$ & $\min_2$
(7)      end for line 4
(8)      for each variable node $v_j$ connected to $c_i$
(9)    Calculate $c2v = \alpha \prod_{n' \in N(m)\setminus n} \operatorname{sign}(m_{v_b \longrightarrow c_i})$ min 1 or min 2
(10)       Calculate $L(v_j) = v2c + c2v$
(11)           $m_{c_i \longrightarrow v_j} = c2v$
(12)     end for line 8
(13)   end for line 3
     Hard decision:
(14)   for each variable node
(15)   Make hard decision if $L(v_j) > 0$ then $v_j = 1$; else $v_j = 0$
(16)   end for line 19
     Parity check equations (Syndrome) and stopping criteria:
(17)   if $\mathbf{H}\cdot\mathbf{v}^T = 0$ then break else $i = i + 1$
(18) end for line 2
```

Algorithm 2: Proposed optimization NMSA.

7), and the fourth one is connected to the eighth variable node (**Col[4] = 8**).

The memory size required for this data structure is calculated by the following equation:

$$\text{memory size} = E \times \text{size of (interger)} + (E + N) \times \text{size of (float)}. \quad (10)$$

The first part of the equation concerns the H matrix storage (**Col** table), and the second part concerns the messages storage (**C2V**[] and $\mathbf{L}(\mathbf{v_j})$). It is clear from equations (7) and (10) that the memory size required for the proposed data structure is lower by 50% than the data structure proposed in [10–14], and consequently, the run-time required for the memory access is highly reduced.

3.4. Parallel Computational Models. The complexity can also be highly reduced depending on the multicore platform and the algorithm parallelism level, which is correlated to the

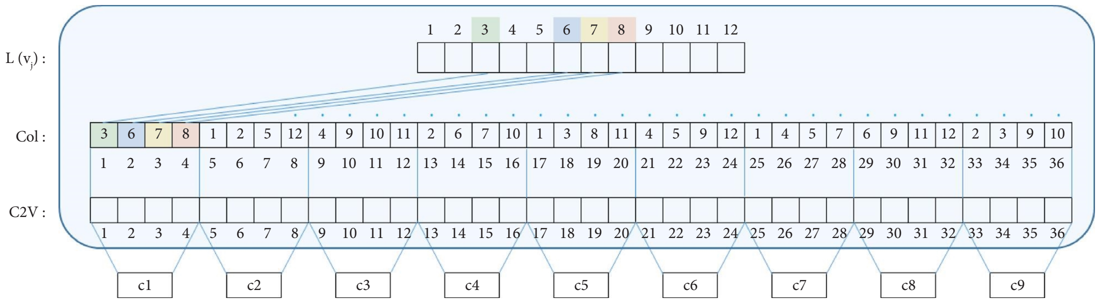


Figure 5: Proposed data structure.

data dependencies during the decoding process, allowing the parallel memory access.

The proposed data structure allows parallel execution because the related data is grouped into consecutive memory locations; each check node has its independent part of the Col and C2V array in the consecutive order. However, the $\mathbf{L}(\mathbf{v_j})$ array is shared between all check nodes and several kernels can read or write the same $\mathbf{L}(\mathbf{v_j})$ in the same times, which generates expensive synchronization run-time for memory access.

Therefore, the parallel processing is allowed between k check nodes if their neighboring variable nodes that are connected to these check nodes do not share any variable nodes; otherwise,

$$\bigcap_{\text{kchecknodes}} \mathbf{N}(c_k) = \varnothing. \tag{11}$$

The value k represents the algorithm parallelism level. For the quasi-cyclic LDPC codes, each row of the $\mathbf{H}_{\text{bm}}$ expansion matrix is expended to Z rows by replacing each shift coefficient by a Z x Z identity matrix or circularly shifted $Z \times Z$ identity matrix, so the Z rows generated from one raw of $\mathbf{H_{bm}}$ satisfy the equation (11), because there is no data dependency among different rows of the identity matrix. Therefore, as depicted in Figure 6, Z threads are lunched in a parallel way; each thread processes one $\mathbf{H}$ matrix row, the transition to the next Z rows is performed sequentially, and so on, until all M rows are processed. Since the entire Z rows generated from one row of the $\mathbf{H_{bm}}$ expansion matrix have the same d_c value, Z threads processing the same $\mathbf{H_{bm}}$ row have almost exactly the same run-time when calculating the updates messages. Consequently, no synchronization is required between the Z threads.

3.5. Complexity Analysis. The complexity is evaluated for each iteration of the LDPC decoding, where one iteration means the process of selecting and updating all edges in the Tanner graph. The total number of edges in the entire Tanner graph is $E = d_c.\ M = d_v.\ N$, where d_c and d_v denote the average degree of check nodes and variable nodes, respectively.

As depicted in Algorithm 1, the NMSA horizontal shuffle algorithm, the horizontal processing involves the calculation of the min1 and min2 values (line 5) which require $2.d_c.M = 2.E$ comparison and $2.d_c.M = 2.E$ read access memory. The calculation of the C2V messages update (line 8) requires $d_c.M$ multiplication and $dc.M$ write access memory. The vertical processing involves the calculation of the summation of the C2V (line 13) that require $d_v.d_c.M = d_v.E$ addition, and $d_v.d_c.M = d_v.E$ read access memory operation, a LLR update (line 15) require $d_c.M = E$ addition operation, $d_c.M = E$ reads access memory operation, and $d_c.M = E$ writes access memory operation, and finally, the V2C messages update (line 17) which requires $d_v.dc.M = d_v.E$ subtraction, $2.d_v.d_c.M = 2.d_v.E$ reads access memory and $d_v.d_c.M = d_v.E$ writes access memory operation.

As depicted in Algorithm 2, for each check node, the processor uploads the i-1 iteration's C2V messages and $L(v_j)$ message, and all update calculations corresponding to the data uploaded are done before passing to the next check node. The min1 and min2 values are calculated from the differences between $L(v_j)$ and C2V uploaded (lines 5 and 6) that require $d_c.M = E$ subtraction, $2.d_c.M = 2.E$ read access memory, and $2.d_c.M = 2.E$ comparison. Then, the C2V and $L(v_j)$message update are calculated in the same loop (lines 9, 10, and 11) which require $d_c.M = E$ multiplication operation, by the normalized factor α (line 9), $d_c.M = E$ addition operation for $L(v_j)$ update using the equation (line 10), $d_c.M = E$ writes access memory operation required for writing the $L(v_j)$ value (line 10), and $d_c.M = E$ written access memory operation (line 11).

Table 3 reports the overall computation operation and memory access needed for both algorithms. Because of the spearing horizontal and vertical processing proposed in previous works, the complexity of the algorithms can be expressed as $O(9.E + 6.d_v.E)$ with a significant number of arithmetic operations and memory access for the vertical steps, which contributes to increasing the decoding complexity, while the complexity of the optimized algorithm is about $O(9.E)$. The d_v value is always strictly superior to two because a variable node is connected at least to two control nodes. For the WiMAX $\mathbf{H}$ matrix shown in Figure 2, the value d_v is 4. Therefore, the proposed optimization clearly reduces complexity compared to the previous works.

4. Experimental Results

4.1. CPU Run-Time Cycles Evaluation. Figure 7 shows a comparison of the proposed LDPC decoder in terms of CPU cycles with the no-optimized HS and NMSA algorithms, for three codes. IR, DR, and DW corresponds, respectively, to the CPU cycle spent in the decoding instruction, the CPU cycle number spent in the read

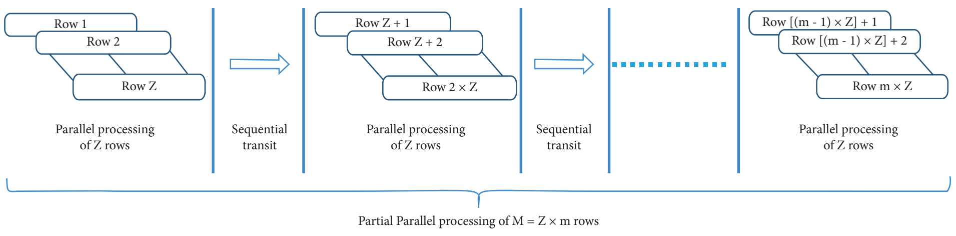

Row 1
Row 2
Row Z
Parallel processing of Z rows
Sequential transit
Row Z + 1
Row Z + 2
Row 2 × Z
Parallel processing of Z rows
Sequential transit
Row [(m - 1) × Z] + 1
Row [(m - 1) × Z] + 2
Row m × Z
Parallel processing of Z rows
Partial Parallel processing of M = Z × m rows

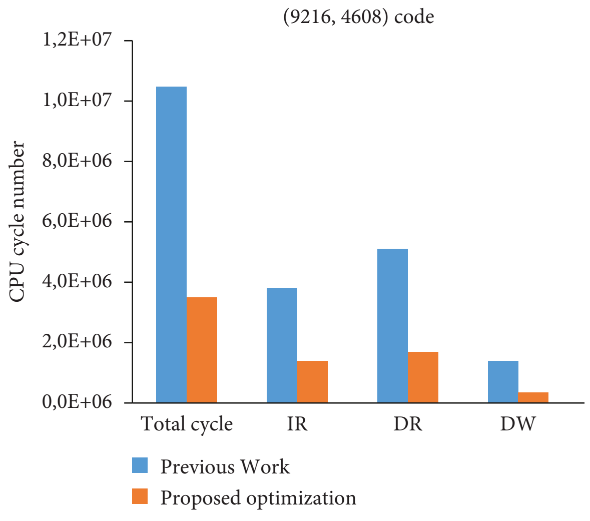
Figure 6: Parallel computational models for LDPC quasi-cyclic.

Table 3: Computation operations and memory by iteration.

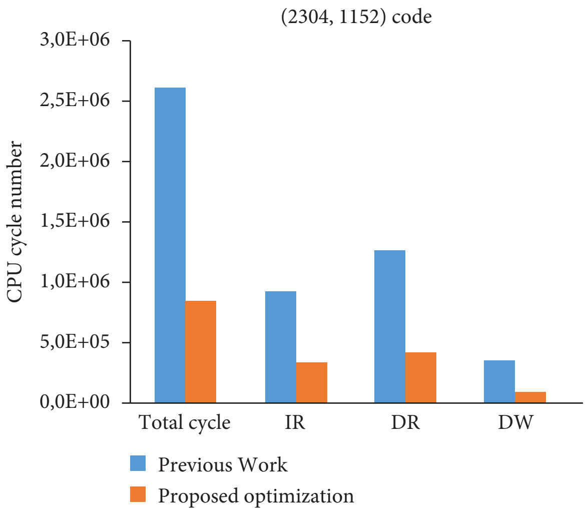
		+	−	×	Compare	Memory access
Data and algorithm structures proposed in [17–19]	Horizontal processing			E	$2.E$	$3.E$
	Vertical processing	$(1+dv).E$	$dv.E$			$4.dv.E+2.E$
	Total computation	$(1+dv).E$	$dv.E$	E	$2.E$	$4.dv.E+5.E$
Proposed optimization		E	E	E	$2.E$	$4.E$

Table 4: CPU run-time reduction compared to previous works.

Code size	CPU cycles previous work	CPU cycles proposed work	Reduction ratio
(9216, 4608)	1.05×10^7	3.45×10^6	66.67%
(2304, 1152)	2.6×10^6	8.5×10^5	66.00%
(576, 288)	6.5×10^5	2.1×10^5	67.7%

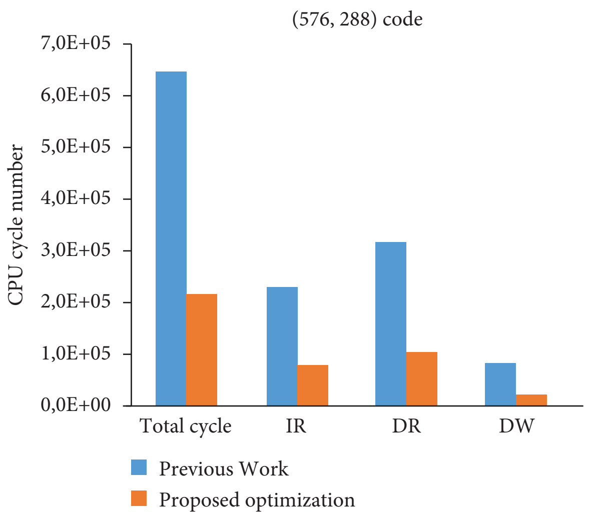


(a)



(b)

Figure 7: Continued.



(c)

Figure 7: Profiling results for (9216, 4608), (2304, 1152), and (576,288) codes.

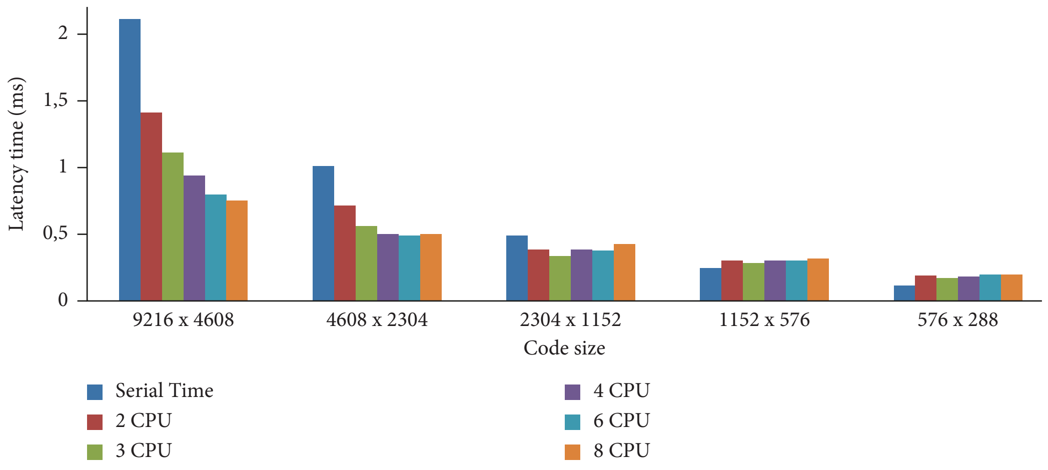


Figure 8: Latency time comparison between serial and parallel processing on the 1st platform for different code sizes.

memory accesses and the CPU cycle number spent in the write memory accesses. The proposed optimization allowed a reduction of the total CPU cycle from 1.05×10^7 to 3.45×10^6 for (9216, 4608) code, from 2.6×10^6 to 8.5×10^5 for (2304, 1152) code, and from 6.5×10^5 to 2.1×10^5 for (576, 288) code. The CPU run-time reduction percentage of the proposed optimization is presented in Table 4. The reduction is about 66% which corresponds to a speedup of 3 times in terms of processing time compared to previous software LDPC decoder using separate horizontal and vertical steps.

The lowest data read (DR) and data write (DW) memory accesses compared to previous work, confirms that the size of memory is compacted and the proposed optimization allows minimizing the memory accesses to the same data.

4.2. Latency Results on Multicore Platforms. The proposed parallel implementation is lunched on two different parallel processing platforms. The first platform is a Marwan HPC platform with an Intel Xeon gold 6130 with 08 CPU at 2.10 Ghz [26]. The second platform is a quad-core Cortex-A72 (ARM v8) @ 1.5 GHz. This platform is running with the Linux distribution and is used in IoT solutions.

Figures 8 and 9 report the latency time in milliseconds between the serial and parallel processing for different code sizes. In the first platform, the speedup between the serial and parallel processing shows significant results for the bigger code sizes; it was approximately 2.8 for (9216×4608) and 2 for (4608×2304).

However, the parallel decoding results of matrix size below (1152×576) demonstrate worst performances than the serial results. An important reason is that this matrix does not have so many edges, so the computation of each thread needs less time than the run time required for OpenMP's thread creation and synchronization.

On the quad-core Cortex-A72 (ARM v8) platform, the parallel processing showed a significant speedup between the serial and parallel processing for matrix sizes higher than 576×288. In contrast to the first platform, even for the 1152 matrix, we can notice a speedup of 2. This is mainly due to

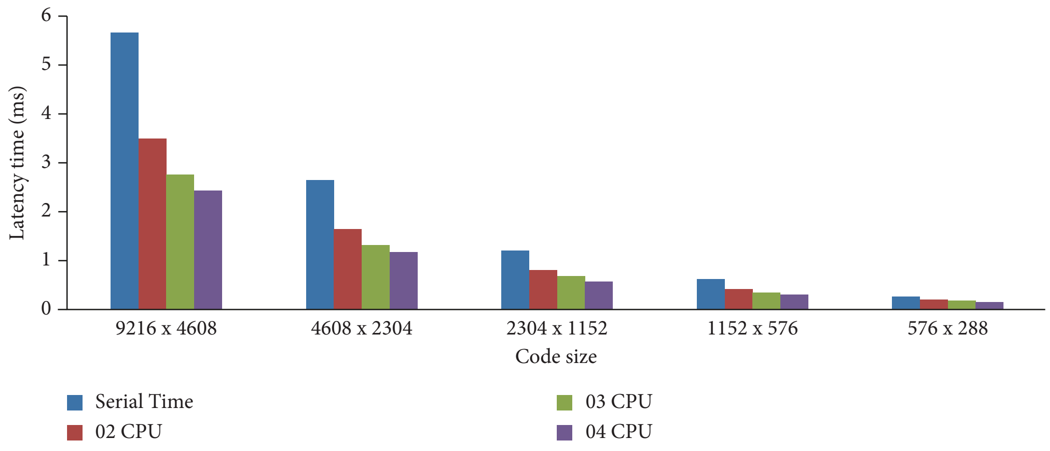


Figure 9: Latency time in (ms) comparison between serial and parallel processing on the 2nd platform for different code sizes.

Table 5: Decoding performance on Intel Xeon gold 6130 multicore platforms using OpenMP.

Matrix sizes	Edge	Z	SNR (dB)	BER	Intel Xeon gold 6130		Quad-core Cortex-A72	
					Latency (in ms)	CPU number	Latency (in ms)	CPU number
9216×4608	29 184	384	4.5	7×10^{-10}	0.74	8	2.4	4
4608×2304	14 592	192	4.5	8.7×10^{-10}	0.48	6	1.16	4
2304×1152	7 296	96	4.5	2.2×10^{-9}	0.33	3	0.56	4
1152×576	3 648	48	4.5	5.2×10^{-9}	0.24	1	0.29	4
576×288	1 824	24	5	3.4×10^{-9}	0.11	1	0.15	4

the clock speed of 1.5 GHz less than the first platform one (2.1 GHz). The calculation time on the quad-core Cortex-A72 platform is always higher than the time needed for the creation and synchronization threads.

According to the results reported in Figures 8 and 9, we define thresholds for enabling parallelism, and the number of CPUs used is chosen dynamically depending on the code size.

Table 5 reports the latency result for several code sizes obtained by a dynamic choice between serial or parallel processing using the clause "if" in OpenMP. The latency is 0.11 ms for the (576, 144) code and 0.74 ms for the big code with a bit error ratio balanced between 3.4×10^{-9} and 7×10^{-10}. In the quad-core Cortex-A72 platform, the latency is 0.15 ms for the (576×288) code (equivalent to 32 Byte) with a bit error ratio of 3.4×10^{-9}, so it has clearly shown that the latency result obtained responds widely to the latency and reliability required for the ultra-reliable low-latency communications.

5. Conclusion

In this paper, we present a new software algorithm optimization, in order to decrease the decoding latency with the same performance obtained by the horizontal shuffle NMSA algorithm. In the optimization algorithm, the separate horizontal and vertical steps are replaced by one step without storage of the V2C message update and with a compact data structure matrix representation, allowing a net reduction of the memory size requirement by about 50% and implementation on a multicore platform. The proposed algorithm achieves much better BER; the bit error ratio is reaching 3.4×10^{-9} and the complexity is reduced by 66%. In addition, the latency is reaching 150 μs for 288 on the ARM quad-core Cortex-A72 system on chip (SoC), used for IoT projects and applications, showing that the proposed optimization behaves much better when considering the BER, decoding complexity, and latency for the ultra-reliable low-latency communication even on IoT devices with very limited computing resources.

Based on the work performed in this thesis and the obtained results, we can identify the following future perspectives:

(i) Increase the expansion coefficient per parity-check matrix in order to make the parallel section wider and reduce the number of crossings between parallel sections, which occurs by creating threads at each entry of the parallel section, thus increasing the time required for thread creation.

(ii) Use codesign methodology to implement the optimized NMSA algorithm on hardware (an FPGA circuit) and the shuffled scheduling and variable memory organization on software.

Data Availability

Data and material are available from the corresponding author upon request.

Conflicts of Interest

The authors declare that there are no conflicts of interest.

Acknowledgments

The authors would like to thank the Sidi Mohammed ben Abdellah University and the SIGER laboratory (SIGER-USMBA laboratory) for financially supporting this work.

References

[1] R. G. Gallager, “Low density parity check codes,” *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[2] D. J. C. MacKay and R. M. Neal, “Near Shannon limit performance of low density parity check codes,” *Electronics Letters*, vol. 32, no. 18, p. 1645, 1996.

[3] R. Tanner, “A recursive approach to low complexity codes,” *IEEE Transactions on Information Theory*, vol. 27, no. 5, pp. 533–547, Sept.1981.

[4] F. R. Kschischang, B. J. Frey, and H. Loeliger, “Factor Graphs and the sum-product algorithm,” *IEEE Transactions on Information Theory*, vol. 47, no. 2, pp. 498–519, 2001.

[5] GPP, “3GPP release 38.913-h00 of URLLC requirement,” 2022, https://www.3gpp.org/ftp/Specs/archive/38_series/38.913/.

[6] C. Kun, S. Qi, L. Shengkai, and P. Chengzhi, “Implementation of encoder and decoder for LDPC codes based on FPGA,” *Journal of Systems Engineering and Electronics*, vol. 30, no. 4, pp. 642–650, 2019.

[7] W. Chen, W. Zhao, H. Li, S. Dai, C. Han, and J. Yang, “Iterative decoding of LDPC-based product codes and FPGA-based performance evaluation,” *Electronics*, vol. 9, no. 1, p. 122, 2020.

[8] S. A. Tamkeen and A. A. Hamad, “FPGA implementation of scaled “quasi-cyclic LDPC” decoder using high-level synthesis,” in *Proceedings of 1st International Conference on Mathematical Modeling and Computational Science*, S. L. Peng, R. X. Hao, and S. Pal, Eds., vol. 1292, Springer, Singapore, 2021.

[9] A. Verma and R. Shrestha, “Low computational-complexity SOMS-algorithm and high-throughput decoder architecture for QC-LDPC codes,” *IEEE Transactions on Vehicular Technology*, vol. 72, no. 1, pp. 66–80, 2023.

[10] J. Ling and P. Cautereels, “Fast LDPC GPU decoder for cloud RAN,” *IEEE Embedded Systems Letters*, vol. 13, no. 4, pp. 170–173, 2021.

[11] A. Li, Q. Jiang, K. Xie, M. Wang, L. Li, and W. Luo, “Low latency LDPC hard-decision algorithm for 5G NR,” *IET Circuits, Devices and Systems*, vol. 14, no. 2, pp. 229–234, 2020.

[12] J. Dai, H. Yin, Y. Lv, W. Xu, and Z. Yang, “Multi-gbps LDPC decoder on GPU devices,” *Electronics*, vol. 11, no. 21, p. 3447, 2022.

[13] A. Kharin, S. Vityazev, E. Likhobabin, and V. Vityazev, “LDPC decoder implementation on DSP+ARM platform with OpenCL,” in *Proceedings of the 2018 7th Mediterranean Conference on Embedded Computing (MECO)*, pp. 1–4, Montenegro, Balkans, August 2018.

[14] M. Razi, M. Benhayoun, A. Mansouri, and A. Ahaitouf, “An improvement and a fast DSP implementation of the bit flipping algorithms for low density parity check decoder,” *International Journal of Electrical and Computer Engineering*, vol. 11, no. 6, p. 4774, 2021.

[15] L. Mostari, R. Meliani, and A. Bounoua, “Iterative effect on LDPC code performance,” *Revue Méditerranéenne des Télécommunications*, vol. 1, no. 2, 2012.

[16] M. Benhayoun, M. Razi, A. Mansouri, and A. Ahaitouf, “Low-complexity LDPC decoding algorithm based on layered vicinal variable node scheduling,” *Modelling and Simulation in Engineering*, vol. 2022, Article ID 1407788, 12 pages, 2022.

[17] J. Li, K. Liu, S. Lin, and K. Abdel-Ghaffar, “Decoding of quasi-cyclic LDPC codes with section-wise cyclic structure,” in *Proceedings of the IEEE Information Theory and Applications Workshop (ITA’14)*, pp. 1–10, San Diego, CA, USA, February 2014.

[18] H. Zhu, L. Pu, H. Xu, and B. Zhang, “Construction of quasi-cyclic LDPC codes based on fundamental theorem of arithmetic,” *Wireless Communications and Mobile Computing*, vol. 2018, Article ID 5264724, 9 pages, 2018.

[19] A. Abotabl, J. H. Bae, and K.-B. Song, “Offset min-sum optimization for general decoding scheduling: a deep learning approach,” in *Proceedings of the 2019 IEEE 90th Vehicular Technology Conference*, pp. 1–5, Honolulu, HI, USA, September 2019.

[20] L. Wang, S. Chen, J. Nguyen, D. Dariush, and R. Wesel, “Neural-network-optimized degree-specific weights for LDPC minsum decoding,” 2021, https://arxiv.org/abs/2107.04221.

[21] Q. Wang, Q. Liu, S. Wang et al., “Normalized min-sum neural network for LDPC decoding,” *IEEE Transactions on Cognitive Communications and Networking*, vol. 9, no. 1, pp. 70–81, 2023.

[22] Q. Chen, Y. Ren, L. Zhou, C. Chen, and S. Liu, “Design and analysis of joint group shuffled scheduling decoding algorithm for double LDPC codes system,” *Entropy*, vol. 25, no. 2, p. 357, 2023.

[23] Z. Xu, L. Wang, S. Hong, and G. Chen, “Generalized joint shuffled scheduling decoding algorithm for the JSCC system based on protograph-LDPC codes,” *IEEE Access*, vol. 9, pp. 128372–128380, 2021.

[24] S. Liang, S. Xie, X. Liu, and Z. Wang, “LDPC decoding with locally informed dynamic scheduling based on the law of large numbers,” *IET Communications*, vol. 16, no. 6, pp. 634–648, 2022.

[25] Valgrind, “Valgrind documentation,” 2023, https://valgrind.org/docs/manual/valgrind_manual.pdf.