HAL open science

High-performance hard-input LDPC decoding on multi-core devices for optical space links

Bertrand Le Gal, Christophe Jego, Vincent Pignoly

▶ **To cite this version:**

Bertrand Le Gal, Christophe Jego, Vincent Pignoly. High-performance hard-input LDPC decoding on multi-core devices for optical space links. Journal of Systems Architecture, 2023, 137, pp.102832. 10.1016/j.sysarc.2023.102832 . hal-04406492

HAL Id: hal-04406492

https://hal.science/hal-04406492v1

Submitted on 5 Apr 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Version of Record: https://www.sciencedirect.com/science/article/pii/S1383762123000115
Manuscript_9dc479a89a4acc3a123eec42f25a99c3

Noname manuscript No.
(will be inserted by the editor)

High-Performance Hard-Input LDPC Decoding on Multi-core Devices for Optical Space Links

Bertrand LE GAL · Christophe JEGO · Vincent PIGNOLY

Received: July 2022 / Accepted: xxxx 2022

Abstract LDPC codes are a family of error-correcting codes that are present in most space communication standards. Thanks to their large processing power and their parallelization capabilities, prevailing multicore devices facilitate real-time implementations of digital communication systems, which were previously implemented into dedicated hardware devices. Previous works were done over the last decade on the implementation of Gbps decoders on programmable devices. However, these works focus on the soft input LDPC decoding algorithms. But, hard-input LDPC decoders are also required to design and prototype for instance next optical-based satellite communication systems. These systems should provide high throughput ($\geq$ 10 Gbps) and low latency ($\leq$ 1 ms) internet links. In this article, the first software-based implementation of a hard-input multi-Gbps LDPC decoder is detailed. Thanks to different parallelization strategies and deeply optimized SIMD codes, throughput up to 7.5 Gbps is achieved when 10 Gallager-E iterations are executed onto an INTEL Xeon device making possible the design of software base station systems providing throughputs of tens of Gbps for optical system evaluation or base station design.

Keywords LDPC · Gallager-E · multicore · SIMD · optical space links

Bertrand LE GAL · Christophe JEGO and Vincent PIGNOLY
IMS laboratory, Bordeaux-INP
CNRS UMR 5218, Univ. of Bordeaux
351 cours de la libération, 33405 Talence, France
E-mail: bertrand.legal@ims-bordeaux.fr@example.com

1 Introduction

Low-Density Parity-Check (LDPC) codes are a popular class of Error Correction Codes (ECC) used in digital communication systems to make reliable transmissions. Due to their excellent error correction performance, LDPC codes were selected for terrestrial wireless standards (e.g., Wi-Fi and 5G) but also for spatial ones (CCSDS, DVB-S2, and DVB-S2x). FPGA or ASIC technologies were for a long time the single way to provide real-time LDPC decoding when hundreds of Mbps or Gbps are necessary [1,2]. These dedicated hardware implementations provide high-throughput and low-energy features at the costs of low flexibility and low reusability. On the opposite, flexible CPU or GPU-based implementations were previously discarded due to the high computational complexity of the LDPC decoding algorithms.

Since a decade, the computational power offered by multicore or many-core programmable devices associated with easy-to-use programming models opened new opportunities. Indeed, coping with low flexibility and long development cycles, researchers and industrials tried to use these programmable devices to implement ECC decoders which are the receiver design bottlenecks [3–7]. Programmable architectures associated with optimized software descriptions made possible the implementation of high-throughput receiver systems. They can be helpful as real-life wireless communication systems and/or prototype for next-generation ones.

Software Defined Radio (SDR) [8] or cloud-RAN [9,10] systems were targeted by previous works. In these works, the RF front-end and demodulation stage provide soft input to the LDPC decoding process. Consequently, like in the field of ASIC/FPGA LDPC decoders, previous works focused on the soft-input Min-

© 2023 published by Elsevier. This manuscript is made available under the Elsevier user license
https://www.elsevier.com/open-access/userlicense/1.0/

Sum (MS) algorithm. Unfortunately, to reach high throughput of several Gbps in optical space communications, ECC decoders are limited to processing hard input values due to current optical technology limitations.

Hard input decoding presents lower error correction performance than soft input ones. Moreover, it involves the implementation of other LDPC decoding algorithms such as Gallager-B, Gallager-E or their variations (e.g. bit-flipping algorithms [11], Probabilistic Gallager-B [12]). Efficient FPGA implementations of Probabilistic Gallager-B (PGaB) and Gallager-E are detailed in [12] and [13], respectively. From a software point of view, the PGaB decoding algorithm needs random number generation and involves computation hazards at runtime making it clearly incompatible with processor features and thus inappropriate for Gbps performance. On the opposite, the Gallager-E decoding algorithm has a formulation close to the MS one used in hardware and software-related works [1,4,5,14–24]. Its computation parallelism is almost regular but its high memory footprint and the logical bit-level computations are not adapted to software processor targets. However, we focused on it and propose efficient SIMD (Single Instruction Multiple Data) and SPMD (Single Program Multiple Data) implementations of the Gallager-E algorithm. Indeed, as demonstrated in [17,19], multicore devices are more efficient than GPU devices to execute horizontal layered formulation of LDPC decoding algorithms in terms of throughput and latency. Moreover, a comparison with an optimized FPGA-based decoder illustrates the weakness and strengths of each kind of implementation.

The remainder of the paper is organized as follows. Section 2 introduces LDPC codes and the horizontal-layered Gallager-E decoding algorithm. Then, the parallelization strategy and algorithm optimizations are provided in Section 3. Section 4 summarizes the experimental results obtained for the proposed decoder implementations. Finally, the conclusion and future works are reported.

2 LDPC decoding algorithm

2.1 LDPC codes

An LDPC code is a linear block code defined by a binary sparse $N \times M$ parity-check matrix called $\mathbf{H}$. A $\mathbf{H}$ matrix, is composed of N columns representing the received information from the system named variable nodes (VN) whereas the $M = N - K$ rows are associated to parity-check information also called check nodes (CN) with K the number of information bits in the received frame. An example of an $\mathbf{H}$ matrix is presented in Equation 1. The code rate of an LDPC code is defined by $R = \frac{K}{N}$.

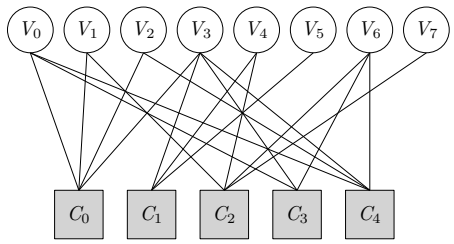

Fig. 1 Tanner graph representation of an LDPC code

$$\mathbf{H} = \begin{array}{c} \\ C_0 \\ C_1 \\ C_2 \\ C_3 \\ C_4 \end{array} \begin{array}{c} \begin{array}{cccccccc} V_0 & V_1 & V_2 & V_3 & V_4 & V_5 & V_6 & V_7 \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \end{pmatrix} \end{array} \quad (1)$$

A Tanner graph is a bipartite graph useful to illustrate an LDPC code. The Tanner graph of the $\mathbf{H}$ matrix provided in Equation 1 is given in Figure 1. The N columns and M rows of $\mathbf{H}$ represent VN and CN, respectively. For each non-null element in $\mathbf{H}$, an edge exists between VN and CN elements. The number of nodes in the jth row and the ith column of $\mathbf{H}$ are VN and CN degrees and are denoted $d_c(j)$ and $d_v(i)$, respectively.

In their general form, $\mathbf{H}$ matrices are unstructured and irregular. It means that non-zero values are randomly distributed and thus d_c and d_v are not constant. Designing an efficient LDPC decoder for unstructured LDPC codes is a challenging task [18,25–27]. To facilitate hardware and software decoder implementations, Quasi-Cyclic LDPC codes (QC-LDPC) were proposed [28,29]. Their regular structures ensure parallel calculations and avoid memory access conflicts. QC-LDPC codes were adopted in spacial communication standards such as CCSDS [30] and DVB-S2x [31].

QC-LDPC codes are composed of $(N/Z) \times (M/Z)$ sub-matrices whose sizes are $Z \times Z$ with Z as the expansion factor as shown in Figure 2. These $Z \times Z$ sub-matrices are shifted identity matrices or null matrices. This structure defines sets of independent data (CNs and VNs) elements. So for QC-LDPC codes, the Z expansion factor represents the minimum achievable computation parallelism level during the decoding process. It ensures that at least Z CNs can be parallelly computed data dependency [1] independently of the computation scheduling [33].

2.2 Gallager-E decoding algorithm

The LDPC decoding process is usually performed thanks to a message passing (MP) approach where VNs and CNs exchange m messages in an iterative way with m the number of one values in the $\mathbf{H}$ matrix. When the decoding process benefits from soft inputs, the sum-product algorithm (SPA) [34–36] or its Min-Sum (MS) based approximations [37–39] are applied to compute the exchanged message values.

Currently, the optical technology used for prototyping digital communication systems is not capable of supplying soft information (received bits and their associated probabilities) [39] to the decoding process under a 10 Gbps throughput constraint. Consequently, the receiver frontend provides hard information (information bits only). The hard input constraint due to optical technology, greatly reducing the information diversity in the decoding process, involves another algorithmic choice. Gallager-B algorithm was initially proposed in [40] to process hard input values. However, its decoding performance is relatively poor due to binary values used to represent exchanged messages. This algorithm was recently improved in terms of error correction performance by inserting decoding noise (e.g. Probabilistic Gallager-B [12]) or gradient descent-based algorithms [41, 42]. These algorithms (PGaB, GDBF, and PGDBF) were introduced for hardware decoders manipulating short frames. However, in the optical communication context, they are useless due to spatial-related constraints: (a) the codewords should be long ($>$ 16k bits), and (b) to reach high-throughput performances randomness and computation irregularity should be avoided. Consequently, they were discarded. The original Gallager-B decoding algorithm can be extended by considering erasures in message passing. This extended algorithm, called Gallager-E algorithm in [43], manages

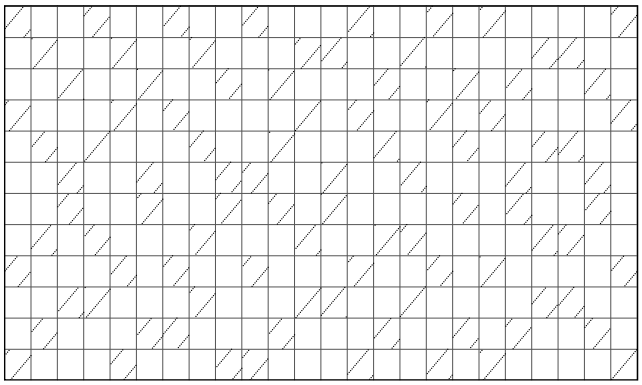

Fig. 2 Example of Quasi-Cyclic LDPC code from [32] standard with $N = 1296$, $K = 648$, and $Z = 54$ meaning that $N/K = 24$ and $M/K = 12$.

Algorithm 1 Horizontal-layered Gallager-E

▷ Input (received word) :
$\boldsymbol{y} = (y_1, y_1, ..., y_N) \in \{0,1\}^N$
init
  $t = 0$,
  $Y_i = 2y_i - 1, \ \forall i \in [1,..,N]$
  $V_i = 0, \ \forall i \in [1,..,N]$
repeat
  ▷ **L1** - loop 1: Horizontal layered scheduling
  for all j $\in [1,..,M]$ **do**
    $C_j^{(t)} = 1$, $Sum_0 = 0$
    ▷ **L2** - Loop 2
    for all i $\in$ N(j) **do**
      ▷ **L2S1** - VN→CN message $\left(v2c_{ij}^{(t)}\right)$ processing
      $$M_{ij} = \begin{cases} V_i, & t = 0 \\ V_i - c2v_{ji}^{(t-1)}, & \text{otherwise} \end{cases}$$
      $$v2c_{ij}^{(t)} = sign(M_{ij} + \omega^{(t)} * Y_i)$$
      ▷ **L2S2** - Check node $C_j^{(t)}$ processing
      $$C_j^{(t)} = \begin{cases} C_j^{(t)}, & M_{ij} \geqslant 0 \\ -C_j^{(t)}, & \text{otherwise} \end{cases}$$
      if $M_{ij} = 0$ **then** $Sum_0 = Sum_0 + 1$ **end if**
    end for
    ▷ **L3** - Loop 3

    for all i $\in$ N(j) **do**
      ▷ **L3S1** - CN→VN message $c2v_{ji}^{(t+1)}$ processing
      $$c2v_{ji}^{(t)} = \begin{cases} 0, & Sum_0 \geqslant 2 \\ 0, & Sum_0 = 1 \text{ and } M_{ij}^{(t)} \neq 0 \\ C_j^{(t)}, & Sum_0 = 1 \text{ and } M_{ij}^{(t)} = 0 \\ C_j^{(t)} \times M_{ij}^{(t)}, & \text{otherwise} \end{cases}$$
      ▷ **L3S2** - VN accumulator V_i updating
      $$V_i = M_{ij} + c2v_{ji}^{(t+1)}$$
    end for
  end for
  $t = t + 1$
until $t \leq t_{max}$
$$\forall i \in [1,..,N], \ x_i = \begin{cases} y_i, & Y_i + V_i = 0 \\ 0, & Y_i + V_i > 0 \\ 1, & Y_i + V_i < 0 \end{cases}$$
▷ Output (decoded word)
$\boldsymbol{x} = (x_1, x_1, ..., x_N) \in \{0,1\}^N$

ternary values $\{-1, 0, +1\}$ for exchanged messages instead of binary $\{-1, +1\}$ ones in the Gallager-B algorithm. This third value which represents doubt during the decoding process drastically improves error correction performance as shown in Figure 3. The bit error rate (BER) performances for the three LDPC codes (C_1, C_2, and C_3) detailed in the Experimental section (Table 1) were obtained when a Monte-Carlo simulation is applied on a BSC channel and with an OOK modulation. As expected, the curves show that the Gallager-E algorithm outperforms the Gallager-B one in the overall use cases when 10 decoding iterations are executed such as in most related works on LDPC decoder implementation [6].

An evaluation of the impact of this low amount of decoding iterations required to obtain high-throughput implementation on error correction performance was also done. Figure 3 also provides the performance level reached by the Gallager-E decoding algorithm when 300 flooding-based decoding iterations are executed. It can

be pointed out that this variation in the number of iterations has a slight impact on the decoding of the C_1 and C_2 LDPC codes but has a higher impact on the code C_3 for which convergence is slowed down. These performance losses are however essential to have a computation complexity reduction of 30× and without which no implementation at several Gbps is possible as reported for instance for Min-Sum decoder implementation [3, 6, 19, 26, 44–47].

Gallager-E algorithm is summarized in Algorithm 1. Contrary to its presentation in [43] which is formulated using flooding scheduling [33], its recent horizontal layered-based scheduling formulation [13] is preferred here. Indeed, this scheduling improves the error correction performance of the Gallager-E decoding process even if only half of the decoding iterations are executed. Halving the number of decoding iterations heavily reduces the global computation complexity of the decoding process.

The algorithmic skeleton provided in Algorithm 1 is close to the ones used for Min-Sum decoding works [1, 4, 39]. However, some differences exist in terms of computational and memory complexities. For instance, contrary to the Min-Sum algorithm that executes 8-bit arithmetic operations (addition, subtraction, minimum, and comparison) commonly supported by programmable processor cores, the Gallager-E algorithm mainly uses on its side 1-bit or 2-bit logic operations. Moreover, in the Gallager-E algorithm, channel memory is expressed with only 1-bit that cannot be devoted to this task. Consequently, a new memory of depth N with $\log_2(max(d_c))$ is added. These differences involve dedicated optimizations to achieve high-efficiency implementation on multicore devices as explained in the following sections.

The decoding algorithm is divided into three main stages:

1. The first stage (Lines 4 to 6) consists of the initialization of the internal values. Received Y_i bit values come from the channel. To ease algorithm understanding the $\{0, 1\}$ are projected on $\{-1, +1\}$ range whereas for implementation purposes they are coded with $\{0, 1\}$ values. Initial values in accumulator nodes V should be null as explained in [21]. The variable named t represents the current decoding iteration. t_{max} value is the maximum number of decoding iterations specified for the decoding process.
2. Then during the second step (Lines 7 to 30), the decoding process executes t_{max} times the same layered decoding process (Line 7). Each CN of the **H** matrix (loop **L2**) is then checked according to the messages received from its VN neighbors. More precisely, the $M_{ij}^{(t)}$ values used to produce $v2c$ messages that come into the check node are evaluated on the fly using channel values (Y), vote accumulators (V) and previously generated messages ($c2v$). The $v2c$ message values are in the range $\{-1, 0, +1\}$. These values are then reused (loop **L2S1**) to compute the parity of the CN (C_j) and to count the number of incoming zero values (Sum_0). These values are then involved a second time (loop **L3**) to generate the $c2v_{ji}^{(t+1)}$ message. The message from CN_j to VN_i is equal to the product of incoming messages except for $v2c_{ij}^{(t)}$. If one or more $c2v$ input messages are equal to zero then the output message is equal to the null value too. In this step, contrary to the MS decoding algorithm, the $c2v_{ji}^{(t+1)}$ value cannot be deduced from $v2c_{ij}^{(t)}$. Indeed, the vote operation cannot be inverted. For this reason, the Y_i values should be kept in memory during the overall CN computations. Finally, the accumulators associated with VN elements are directly updated (line 26).
3. The final step (Lines 31 to 33) decides the value of the decoded bits according to the accumulator values and the Y input ones. If the decoding algorithm has not reliably decided on a bit ($Y_i + V_i = 0$), then the received value from the channel (Y_i) is selected.

This decoding algorithm suitable for hard inputs gives an interesting correction power when the $\omega^{(t)}$ penalty factor, whose value depends on the decoding iteration, is correctly set. Traditionally, ω value is fixed as follows: $\omega = 2$ when $t \in [0, tmax/2[$ whereas to $\omega = 1$ otherwise [43]. The undeniable advantage of this algorithm is its reduced computational and memory complexities.

3 Parallelization strategies

Over the last decade, with the advent of software-defined radio and the growing computational capacity of multicore and manycore platforms, researchers and engineers have been trying to implement the basic building blocks of these communication systems on novel platforms to provide flexibility and efficiency.

However, this is still challenging work because many architectural constraints limit the efficient parallelization of computations or memory accesses. These constraints are not in accordance with the application requirements. In this section, we detail the parallelization strategies and the optimizations performed to improve data access efficiency. This section details multicore implementations of the Gallager-E decoding algorithm because it seems to be the most relevant architecture to execute (1) horizontal layered-based formulation of the decoding algorithm [4] and (2) execute in parallel 8-bit

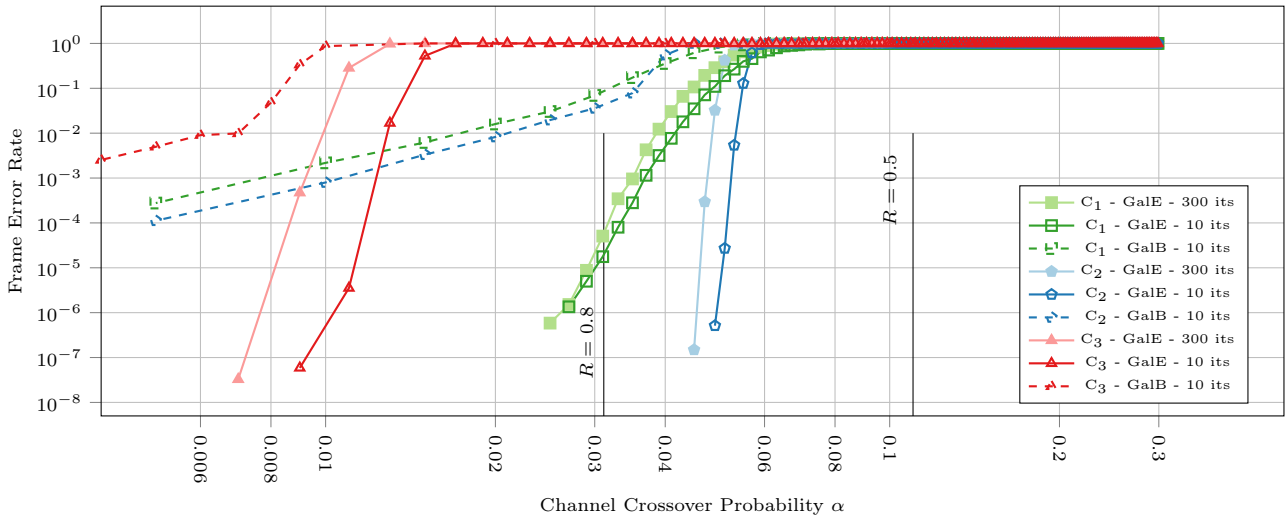

Fig. 3 Comparison of error correction performance of Gallager-B and Gallager-E algorithms.

logical and arithmetic computations. Many works on Min-Sum algorithms such as [6,23,24,44] show the viability and the efficiency of GPU-based implementation. Our previous experiences combined with the intrinsic properties of the decoding algorithm and the need to control the data processing latency discouraged us from undertaking a GPU implementation.

3.1 Multicore systems

INTEL x86 and ARM processors currently provide three distinct main parallelization features to speed up program executions [48]:

1. At the higher level, multicore devices include many physical processor cores that can process tasks in parallel. Currently, the number of cores can reach up to dozens for server-grade circuits. Programming paradigms such as MIMD (multiple instructions, multiple data) or SPMD (single program, multiple data) can be applied to benefit from this paradigm [49–51].
2. At the same time, each physical processor core includes SIMD (Single Instruction Multiple Data) units [52–54] that execute parallel computations. SIMD units are 128b up to 512b wide, authorizing from 16×8b up to 64×8b computations per instruction. Branch divergence between computation flows is also possible thanks to bitwise masking operations making single instruction, multiple threads (SIMT) programming model achievable. However, branch divergence is not a good practice to reach high-performance levels.
3. Finally, at the lower level, x86 and ARM processor architectures are superscalar [55] and thus implement Instruction-Level Parallelism (ILP). Superscalar instruction execution that is handled transparently by the CPU itself authorizes multiple instructions to be executed within a single clock cycle according to the pipelined resource availabilities and the data dependencies (for instance, up to 6 $\sim$ 8 μops per clock cycle for INTEL processors).

To reach high implementation efficiency, all these features should be addressed together. The different parallelization strategies and the optimizations applied at these three levels for Gallager-E LDPC decoding are reported in the following subsections.

3.2 SIMD parallelization

Parallelization of the message-passing algorithm was widely studied in the context of traditional Min-Sum decoder implementations [6]. For multicore implementations, two efficient coarse-grain SIMD parallelization strategies were proposed [4,5]. These generic approaches proposed for Min-Sum decoding can also be applied to the Gallager-E algorithm. They provide different advantages and drawbacks.

Inter-frame parallelization strategy [4] takes advantage of SIMD units to decode multiple frames in parallel. The decoding process executes the same computations on data from different frames as shown in Figure 4. Inter-frame strategy eases the software description and provides regular computation parallelism at runtime. Indeed, when the number of frames processed in parallel ($F \times 8$b) equals the SIMD width, the SIMD efficiency is constant at 100%. Moreover, the frame values are interleaved before and after the decoding [20].

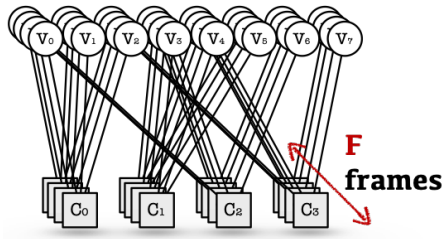

Fig. 4 Inter-frame parallelization behavior

The memory accesses become regular and straightforward making decoder implementation efficient. However, the inter-frame strategy drawback effect comes from its memory footprint (Δ_{inter} in Equation 2). Indeed, this footprint becomes quickly larger than L1, L2 and L3 memory caches as F is in $\{16, 32, 64\}$ range. Its high memory bandwidth requirement limits the performances and the scalability of the decoder implementations when $N \geq 2^{10}$. It also produces high processing latency.

$$\Delta_{\text{inter}} = \mathbf{F} \times (2 \times N + m) + m \tag{2}$$

On the opposite, the intra-frame strategy proposed in [5] takes advantage of SIMD units to parallelize internal computations from a single frame. It takes advantage of the QC-LDPC code structure to compute Z independent CNs in parallel as illustrated in Figure 5. The main advantage of this strategy is the limitation of the memory footprint (Δ_{intra} in Equation 3) at runtime as a single frame is handled. The drawback effect comes from the sophistication of the software description of the decoder and the memory access slow down. Indeed, **H** structure management is done at runtime, generating noncontiguous memory accesses to load and store SIMD registers. Moreover, depending on the LDPC code, a SIMD usage rate of 100% is rarely obtained. Indeed, it depends on the Z expansion factor that should be a multiple of the SIMD width. Nevertheless, from a system point of view, the intra-frame implementation delivers low-latency feature and scale quite linearly in an SPMD context.

$$\Delta_{\text{intra}} = 2 \times N + m + \frac{m}{Z} \tag{3}$$

Both SIMD parallelization strategies were used successfully for Min-Sum decoder implementations [4–6] and can be adapted and applied to speed up the execution of loop **L1** defined in Algorithm 1. Both approaches are evaluated because they provide different features, different optimization opportunities and thus different trade-off solutions as highlighted in section 4.

3.3 ILP improvement

An efficient implementation of loops **L2** and **L3** in Algorithm 1 is crucial. Indeed, they are executed M times per decoding iteration. Consequently, a specific mapping of the algorithmic operations on available SIMD instructions is required. Gallager-E decoding algorithm mainly manipulates bit or ternary values and has many conditional statements. So Gallager-E decoding is more challenging to be efficiently implemented on processor cores by comparison to the Min-Sum algorithm [4]. At the same time, the algorithmic description should provide higher instruction level parallelism (ILP) while reducing as possible the length of the instruction critical path. To achieve these objectives many algorithmic transformations were applied to the original formulation. They are successively described below.

First, to reduce the complexity of the M_{ij} computation (loop **L2S1**, line 14) that depends on the iteration counter, initialization of the $c2v$ messages to zero was done. It discards the $t = 0$ comparison and thus the conditional moves or jumps at runtime. Consequently, Algorithm 1, line 14 becomes:

$$M_{ij} = V_i - c2v_{ji}^{(t-1)} \tag{4}$$

A second optimization was done for the $v2c$ message computation (Line 15) whereas ω is in the set $\{1, 2\}$. Indeed, it is necessary because no 8-bit multiplication instruction nor 8-bit shifting instruction is available in INTEL AVX/SSE SIMD ISA. A solution is to perform conversions to/from 16-bit data format to execute a 16-bit multiplication operation. But, it produces impacting time penalties. Consequently, the behavioral multiplication operation was implemented thanks to an 8-bit addition of the Y_i value with the result of a logical *and* instruction between the Y_i value and a binary mask value $\{0x00, 0xFF\}$. The binary mask value depends on the current decoding iteration. The transformation results for line 15 from Algorithm 1 are provided in Equation 5. It is the currently most efficient way to im-

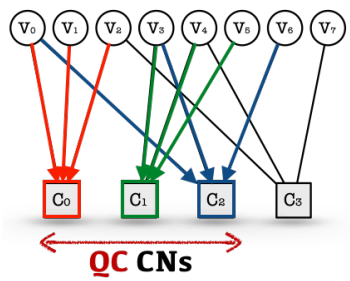

Fig. 5 Intra-frame parallelization behavior

plement $\omega \in \{1, 2\}$ multiplication on INTEL or ARM cores.

$$v2c = sign(M + Y + (Y \& \ bitmask_{\omega})) \quad (5)$$

Finally, the zero counting operation was modified such as the conditional part using this value (Lines 18 and 24). A normal way to manage this behavior is to compare values with zero and then increment by one the counter value. However, SIMD comparison instruction produces mask values {0x00, 0xFF} with a 0x01 masking instruction before the addition one. Consequently, instead of adding one values, -1 (0xFF) values are added. The counter values become 0xFF, 0xFE, etc. depending on the number of zero elements. This transformation is possible because in the **L3S1** loop, the first part of the conditional structure can be reformulated as $(Sum_0 - (M_{ij} = 0)) \neq 0$. This tricky optimization removes logical instructions and comparisons from the execution critical path. Moreover, it facilitates the implementation of the $c2v$ conditional computation making it possible to describe it as a value selection in the range {0, 1}, and then a conditional sign inversion to regenerate the outgoing message in the range { -1, 0, 1 }.

After these transformations, the number of instructions in the processing kernels (loops **L2** and **L3**) is quite small whereas the number of **L2**/**L3** loop iterations is limited to range $[7, 20]$ (due to benchmarked LDPC codes). Specialized kernel codes are generated at compile time to remove useless control instructions and improve ILP at runtime. To this end, the features of the C++11 language (i.e., template specialization) are applied as in [56]. For each CN degree value, a dedicated and optimized kernel is generated. Consequently, the number of instructions is then minimized for each **L2** loop execution. At runtime, an array of function pointers is preferred to select the right binary code to execute without miss prediction penalties.

3.4 Memory compression on the fly

The memory bandwidth can become the performance bottleneck of inter-frame decoder implementations [4,5] on multicore and many-core devices when LDPC codes longer than $N \geq 2^{10}$ are processed [4, 5]. Indeed, the decoder memory footprint becomes quickly higher than L1/L2 memory cache capacities. In the inter-frame configuration, the number of processed frames depends on the SIMD width. To benefit optimally from SIMD unit capabilities, F should be fixed to {16, 32, 64} for {128 256 512} bit-wide SIMD units, respectively. These numbers of frames processed in parallel involve high memory footprints (Equation 2).

However, contrary to Min-Sum-based decoder implementations [4–6] that need 8-bit values internally for all datasets (Y_i, V_i and $c2v$), the Gallager-E decoding algorithm manipulates low word-length information. Indeed, the Gallager-E decoding algorithm consumes and produces a large set of binary or ternary values. Typically, all these values are stored on bytes because they are involved later in 8-bit arithmetic operations. However, as the memory bandwidth is a bottleneck for real use cases, a memory compression technique can be relevant.

Compression divides the memory footprint by 4 for exchanged messages ($c2v$) that contain ternary values (i.e. ternary values are stored on 2 bits). It also enables dividing by 8 the footprint for channel values (Y_i) that are binary ones. Thus, the memory footprint of inter-frame decoders becomes:

$$\Psi_{\text{inter-compressed}} = \mathbf{F} \times \left(\frac{9N}{8} + \frac{m}{4} \right) + m \quad (6)$$

whereas for intra-frame decoders the memory footprint can be formulated as follows:

$$\Psi_{\text{intra-compressed}} = \frac{9N}{8} + \frac{Z \times m + 4 \times m}{4 \times Z} \quad (7)$$

Memory footprint reduction involves the execution of additional SIMD instructions at runtime to compress and decompress data during load and store operations. Note that data compression is lossless and thus does not impact error correction performance.

AVX-512 instruction set[1] with masked operation makes easy function implementations as demonstrated in Listing 1. The Y_i compression function extracts sign bits of the 64×8-bit values and stores them on 64 bits in memory (8 bytes). The Y_i decompress function performs the opposite operation to recreate the 64×8 bit content. It loads the 64 bits from memory, which masks and selects -1 or $+1$ values contained in neg_one or pos_one registers. The message compression function is a bit more tricky. The 64×8 bit values are compared with $+1$ and -1. Both comparison results are stored on 64 bits in memory. The opposite function selects $+1$ or -1 value according to the comparison results stored in memory. If both results are false then the 0 value is selected.

[1] Note that SSE4 and AVX2 function descriptions have approximately the same computational complexity even if they use other SIMD instructions.

```
const __m512i zero    = _mm512_setzero_si512();
const __m512i pos_one = _mm512_set1_epi8(0x01);
const __m512i neg_one = _mm512_set1_epi8(0xFF);

void compress_and_store_yi(__mmask64* ptr, const __m512i x) {
    ptr[0] = _mm512_movepi8_mask(x);
}

__m512i load_and_uncompress_yi(const __mmask64 x) {
    return _mm512_mask_blend_epi8( x, pos_one, neg_one );
}

void compress_and_store_msg(__mmask64* ptr, const __m512i x) {
    ptr[0] = _mm512_cmpeq_epi8_mask( x, pos_one );
    ptr[1] = _mm512_cmpeq_epi8_mask( x, neg_one );
}

__m512i load_and_uncompress_msg(const __mmask64* ptr) {
    __m512i w1  = _mm512_mask_blend_epi8( ptr[0], zero, pos_one );
    __m512i w2  = _mm512_mask_blend_epi8( ptr[1], zero, neg_one );
    return _mm512_or_si512(w1, w2);
}
```

Listing 1 SIMD functions for (de)compression of binary and ternary values

As shown in Listing 1, compression and decompression functions could be executed on x86 architecture with a low latency penalty whereas a single memory cache miss produces a latency penalty of at least hundreds of clock cycles. Benefits achieved using lossless memory compression at runtime are estimated in terms of memory footprint and decoding throughputs on real LDPC codes in the Experimental section.

3.5 SPMD parallelization

Different parallelization techniques could be applied to take advantage of the **P** cores for message-passing algorithm implementation. It is possible to use them to speed up the executions of loop 1 (**L1**) defined in Algorithm 1. However, this approach is currently inefficient as demonstrated in [4] because:

- Loop elements are not independent when a horizontal layered-based decoding algorithm is applied. It can involve memory access conflicts at runtime, requiring precise synchronization barriers between the different threads.
- The time spends on forking and joining tasks is not negligible compared to **L1** kernel execution time.

The best way to increase the performance level is to allocate **P** independent LDPC decoders to execute **P** distinct frames. It avoids L1/L2 memory sharing at runtime between the cores. Moreover, it increases L3 cache usage linearly with the **P** factor. It also increases the pressure on the memory bandwidth when the overall dataset does not fill in the L3 cache. This acknowledgment mainly concerns the inter-frame-based implementations where $\mathbf{P} \times \mathbf{F}$ frames are processed in parallel with $\mathbf{F} \in \{16, 32, 64\}$ involving large memory footprints.

To enable an asynchronous behavior of the decoders, the LDPC decoders are encapsulated in C++11 threads. Threads are executed asynchronously according to input data availability.

4 Experimentation results

4.1 Experimentation setup

The software-based LDPC decoder implementations were described in C++ 11 language. The targeted device was an INTEL Xeon Gold processor. Consequently, the AVX512 instruction subset was selected. INTEL intrinsics which are C-style functions were applied to benefit from INTEL SIMD features. To optimize the instruction scheduling and generate the executable file, the software decoder descriptions were compiled with the CLANG++/LLVM 10.0 toolchain. The compilation flags provided to the toolchain are: *-march=native -mtune=native -Ofast -funroll-loops*.

The host platform was a multicore system composed of a dual-socket INTEL Xeon Gold 6148 CPU. Each Xeon processor contains 20 physical processor cores. The overall processor cores share a 28160K L3 memory cache and 256 GB of RAM. A working frequency

Table 1 Properties of the selected QC-LDPC codes

Code	C_1	C_2	C_3
Source	[32]	custom	custom
(N, K)	$(1296, 648)$	$(32768, 16384)$	$(20480, 16384)$
Rate	1/2	1/2	4/5
Z	54	256	256
d_c	$\{8\}$	$\{7, 8, 9\}$	$\{19, 20\}$
m	5184	131072	81664

up of to 3, 70 GHz is achievable on this platform thanks to the turboboost feature when a single processor core is activated. The average working frequency is 2,40 GHz and 2,20 GHz when 50% and 100% of the cores are activated, respectively. This frequency reduction is due to power dissipation constraints. Note that these values were confirmed using the *i7z* tool during experiments.

The behavior and the decoding performance of the proposed optimized implementation were evaluated using three different QC-LDPC codes. The main characteristics of the selected codes (N, K, m) are summarized in Table 1. The first code comes from related works on hard input LDPC decoding [32]. The two others are custom LDPC codes described specifically for spatial optical communications [13] with our Airbus S&D partner. The BER performance of these codes is provided in Figure 3 (section 2.A). These results are consistent with the published literature and the simulations performed with the AFF3CT toolbox [57]. Moreover, the achieved BER performance levels are compliant with the optical space links.

4.2 Absolute performances

For benchmarking purposes, a complete digital communication system simulation is executed during a period of at least 120 seconds per experiment to avoid for instance working frequency scaling impact on averaged throughput or latency values. The throughput (Γ) and the latency (Δ) measures are reported in Table 2. Performances are provided for the following decoder implementations:

- **Inter-frame setup** (d_1) - Each physical processor core decodes $\mathbf{F} = 64$ frames in parallel to fully utilize the 512b SIMD units. All the values (channel, accumulators and messages) are stored on 8-bits.
- **Inter-frame with memory compression setup** - (d_2). Each physical processor core processes $\mathbf{F} = 64$ frames in parallel. The channel values are compressed and stored using 1 bit whereas exchanged message values are compressed on 2 bits. The accumulator values are stored using 8-bits.
- **Intra-frame setup** (d_3) - A single LDPC frame is processed ($\mathbf{F} = 1$). All the values (channel, accumulators and messages) are stored on 8-bits.

Experimental results provided in Table 2 show that for a single-core configuration, throughputs from 77 Mbps up to 294 Mbps were achieved for the d_1 implementation. The highest throughput was obtained when the C_1 code is decoded. Indeed, in this case, the inter-frame decoder has a small memory footprint (491 KB) that fills in L2/L3 caches. However, for long codes (C_2 and C_3), the throughputs are reduced up to 4× due to the memory footprint that grows up to 12416 KB as reported in Table 3. The acknowledgment concerning the memory bandwidth limitation even in single-core configuration is confirmed by d_2 implementation results. Indeed, the d_2 implementation gives higher throughputs for C_2 and C_3 LDPC codes due to memory compression that reduces memory bandwidth requirement and thus cache misses. In these cases, the memory footprint of the decoder decreases to 6272 KB. However, the additional arithmetic and logical computations used to compress the information at runtime in d_2 make it less efficient for C_1 code where cache miss penalty reduction does not compensate for the compression penalty. Finally, the d_3 decoder implementation provides the highest decoding throughputs for C_2 and C_3 LDPC codes (from $\approx 1.2\times$ up to $\approx 2.4\times$ higher than d_2). This high-performance level is due to its reduced memory footprint whose maximum value is 194 KB (Table 3). In parallel, the processing latencies of the d_3 implementation are about 99% shorter than the d_1 and d_2 ones. However, d_3 implementation is not the best solution for the C_1 LDPC code in terms of throughput. It comes from two points: first, the usage rate of the SIMD units for d_3 implementation is lower than 100% at runtime for the C_1 LDPC code. Indeed, the Z factor coming from the QC-LDPC $\mathbf{H}$ matrix equals 54 whereas $Q = 64$ (SIMD efficiency $\approx 84\%$). Secondly, d_3 implementation involves more complex memory accesses whose cost is not compensated by a better L1/L2 cache efficiency. However, from a latency point of view, it remains the best solution.

Experiments were also conducted with multiple processor cores activated to check the scalability of the decoder implementations. Numerical values when 20 and 40 cores are activated are provided in Table 2. Figure 6 provides the complete benchmarking results ($1 \leq \mathbf{P} \leq 40$). One can note that the d_1 implementation is better in terms of throughput for C_1 LDPC code in all $\mathbf{P}$ setups with speedup factors of 15× and 25× for 20 and 40 cores, respectively. However, d_1 decoder performances fall for C_2 and C_3 LDPC codes where the speedup factors are limited to 7× to 10×.

Table 2 Performances of Gallager-E LDPC decoders on INTEL Xeon Gold 6148 CPU

		Γ in Mbps			Δ in μs			$\mathbf{P}$ in Watts			$\mathbf{e}$ in nJ/bit		
Code	#cores	Γ_{d_1}	Γ_{d_2}	Γ_{d_3}	Δ_{d_1}	Δ_{d_2}	Δ_{d_3}	$\mathbf{P}_{d_1}$	$\mathbf{P}_{d_2}$	$\mathbf{P}_{d_3}$	$\mathbf{e}_{d_1}$	$\mathbf{e}_{d_2}$	$\mathbf{e}_{d_3}$
C_1	1	294	272	220	281	304	6	180	180	180	613	662	819
C_2	1	108	136	180	12034	9523	113	167	167	172	1547	1228	956
C_3	1	77	106	247	27010	19671	132	171	170	169	2221	1228	685
C_1	20	4487	4041	3207	369	410	8	300	299	290	67	74	91
C_2	20	813	2412	2546	53106	10865	162	419	411	297	516	171	117
C_3	20	876	2030	3177	48785	21192	218	412	416	298	471	205	94
C_1	40	7532	6833	5460	440	485	10	351	342	331	47	51	61
C_2	40	784	3447	4298	66739	15213	191	437	472	340	558	137	80
C_3	40	712	2286	5446	118795	37893	248	434	473	341	610	207	63

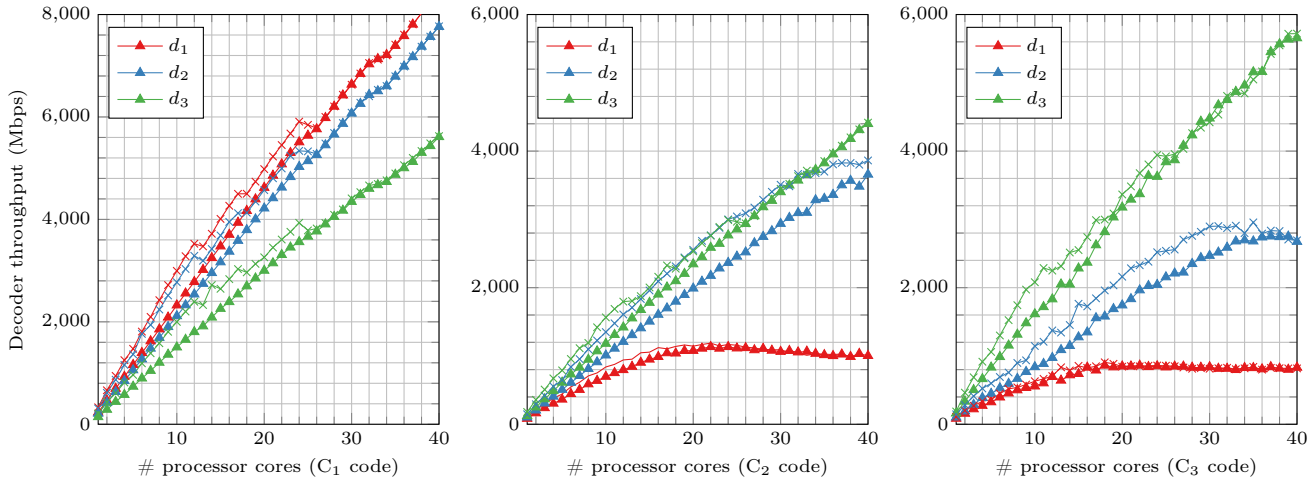

Fig. 6 Throughput performances of AVX512 decoder implementations depending on the number of activated processor cores and the processed LDPC codes. Colored lines with crosses represent values obtained when the turboboost feature is activated.

Indeed, even if the number of cores is increased from 20 to 40, a performance floor due to the memory bandwidth appears. This performance floor is visible in Figure 6. Memory bottleneck assertion is validated by the results obtained for the d_2 decoder implementation. Indeed, the d_2 decoder implementation executes memory compression over-classed d_1 solution for the long codes (C_2 and C_3) codes. For d_2 decoder implementation, the measured speedups are 17× and 24× when 20 cores and 40 cores are activated, respectively. The throughput grows with the number of activated cores. However, the performance improvement is not linear due to working frequency scaling. In 20 and 40 core setups, the d_3 decoder implementation offers the best performances and achieved up to 5446 Mbps for C_3 LDPC code. The speedup factors obtained in comparison with single-core experiments are $22 \approx 24\times$. At the same time, the working frequency reduction from 3.7 GHz down to 2.4 GHz increases the processing latency by $[1.5\times \approx 2\times]$.

Table 3 Memory footprint of the decoding process depending on the implementation type

	AVX-512 (1 core)			AVX-512 (40 cores)		
	Ψ_{d_1}	Ψ_{d_2}	Ψ_{d_3}	Ψ_{d_1}	Ψ_{d_2}	Ψ_{d_3}
C_1	491 kB	248 kB	8 kB	19 MB	10 MB	320 kB
C_2	12416 kB	6272 kB	192 kB	485 MB	245 MB	8 MB
C_3	7744 kB	3916 kB	120 kB	302 MB	153 MB	5 MB

In parallel to the throughput and latency evaluations, power and energy per bit comparisons were also done. Measurements are provided in Table 2. The reported power consumption values include the consumption of the CPU and RAM packages. These values were obtained at runtime with the *turbostat* tool. This tool captures the power consumption from sensors. The measured power consumption depends on the number of activated cores and the memory bandwidth. The power consumption results are equivalent in the single-core configuration because the RAM bandwidths are equivalent. The most efficient energy-efficient solution for C_2 and C_3 codes is the d_3 implementation due to its higher throughput performance. Multiple cores activation increases the energy efficiency gap. Indeed, for C_2 and C_3 codes, the d_1 and d_2 implementations have a higher power consumption ($\approx 40\%$) than the d_3 implementation due to their high usage rates of the RAM. The RAM consumes up to 180 W. The energy per decoded

bit metric reinforces this performance gap about the inefficiency of the inter-frame parallelization scheme.

4.3 Performance evaluation on other devices

In the previous section, we mainly focused on performance in terms of throughput in accordance with our application context i.e. optical space links. This is the reason why we selected an INTEL Xeon Gold multicore device. In this section we discuss in detail the investigation carried out on the energy performance of multicore based implementations.

Several heterogeneous multicore architectures were evaluated: two ARM targets and another INTEL multicore target. The ARM cores evaluated came from an Apple computer (MacBook Pro 2021) and an NVIDIA Orin platform (2022). A MacBook Pro (2019) computer with an INTEL Core-i9 processor was also benchmarked. The software descriptions of the Gallager-E decoders were optimized for these different architectures. Software layers were updated to efficiently support NEON SIMD ISA for ARM processors. The same efforts were performed to dedicate the AVX512 codes to the AVX2 and SSE4 processors for INTEL processors.

The experimental results as well as the specifications of the different platforms are summarized in Table 4. Similar experimental settings and measurement methods were employed for all platforms. To facilitate analysis of the results for all architectures, the measured throughputs have been normalized to the INTEL Core-9 architecture running the SIMD unit in 128-bit mode.

Throughput evolution in single-core configuration shows that switching from 128-bit to 256-bit mode on the INTEL Core-i9 architecture provides an average speedup of 1.55× while 2× the amount of data is processed in parallel. Comparing this performance in terms of throughput with the INTEL Xeon values show that the INTEL Core-i9 processor provides better results in SSE4 and AVX2 mode due to its higher working frequency. However, the INTEL Xeon processor recovers the advantage in AVX512 mode even if the theoretical gains between the parallelization modes are not reached. The NVIDIA Orin platform composed of ARM A78 processor cores enables it to achieve lower performance (approximately 2 times less) than INTEL Core-i9. It is mainly due to the difference in working frequency. The ARM M1 architecture achieves high-performance levels despite its SIMD 128-bit units. Indeed, the performance in terms of throughput reaches those of INTEL Core-i9 or INTEL Xeon processors in AVX2 mode (256 bits) thanks to an equivalent operating frequency and 2× larger L1 cache size. From an energy point of view, the ARM M1 core delivers gains of 2× up to 32× compared to other INTEL devices and a 2× saving compared to the NVIDIA Orin platform.

When the platforms are in multicore mode, the performance in terms of throughput is clearly in favor of the INTEL Xeon architecture (AVX512) which has 40 physical cores. The latter has a level of performance about 8 times higher than the Core-i9 architecture with its 6 cores. These two platforms have an equivalent energy consumption per decoded bit. The platform composed of ARM A78 is less efficient in terms of throughput despite its 12 physical cores. However, from an energy point of view, it outperforms the INTEL multicores by a factor of 2. This observation is consistent because it has 10 times fewer physical cores than the INTEL Xeon, but the size of the SIMD is 4 times smaller. From an energy-per-decoded-bit perspective, the ARM M1 core is also at least 2 times more efficient than INTEL solutions.

For our research, the main constraints were throughput and latency performances. Under these conditions, INTEL multicore systems currently offer the best performance levels. However, for the design of energy-constrained digital communication systems with lower throughput requirements, the current generations of ARM processors may be a relevant solution.

4.4 Comparison with FPGA implementations

Finally, the proposed Gallager-E LDPC decoder implementations on multicore devices were compared with FPGA-based ones [13] to estimate the feature differences. Indeed, works presented in [13] details hardware-optimized Gallager-E architectures for a Zynq Ultrascale+ FPGA (xczu9eg-3ffvb1156e). This semi-parallel architecture is based on the work described in [18]. An overview of the hardware Gallager-E decoder architecture is given in Figure 7.

Multi-core execution modifies the previous acknowledgment. The throughput difference is in this setup in the range of 5× to 11×. Indeed, the number of decoding cores that can be instantiated in the FPGA device is lower than the number of cores in the Xeon processor. But at the same time, the Xeon working frequency is approximately halved. Consequently, the power consumption per decoded bit drops sharply for the Xeon solution. Indeed, the power consumption is only double when 40 cores are activated compared to 1 core configuration and the throughput gain is thus over 20×. As a consequence, the differences in terms of power consumption between the FPGA implementation and the multicore implementation vary finally from 52× to 82×.

Table 4 Throughput and energy comparisons for various INTEL and ARM platforms

	ARM M1	ARM-A78	INTEL Core-i9		INTEL Xeon Gold		
SIMD	128b	128b	128b	256b	128b	256b	512b
Frequency (GHz)	3,20	2,20	4,60	4,60	3,50	3,50	3,50
Power (W)	5,5	4	27	30	180	180	180
L1 cache	128 KB	64 KB	32 KB	32 KB	32 KB	32 KB	32 KB
L2 cache	128 KB	3 MB	256 KB	256 KB	1 MB	1 MB	1 MB
L3 cache	12 MB	6 MB	12 MB	12 MB	55 MB	55 MB	55 MB
# cores	1	1	1	1	1	1	1
Power (W)	5,5	4	27	30	180	180	180
Γ_{d_3}/core (C_1)	$1,28\times$	$0,47\times$	110 Mbps	$1,62\times$	$0,88\times$	$1,47\times$	$1,90\times$
Γ_{d_3}/core (C_2)	$1,49\times$	$0,54\times$	100 Mbps	$1,40\times$	$0,86\times$	$1,20\times$	$1,80\times$
Γ_{d_3}/core (C_3)	$1,31\times$	$0,43\times$	127 Mbps	$1,57\times$	$0,84\times$	$1,47\times$	$1,94\times$
$\mathbf{e}_{d_3}$ in nJ/bit (C_1)	39	77	245	169	1856	1111	861
$\mathbf{e}_{d_3}$ in nJ/bit (C_2)	37	74	270	214	2093	1500	1000
$\mathbf{e}_{d_3}$ in nJ/bit (C_3)	33	73	213	150	1682	963	729
# cores	4	12	6	6	40	40	40
Power (W)	18	20	50	50	340	340	340
Γ_{d_3}/platform (C_1)	$1,67\times$	$2,13\times$	293 Mbps	$2,56\times$	$7,49\times$	$13,47\times$	$18,63\times$
Γ_{d_3}/platform (C_2)	$1,20\times$	$1,40\times$	460 Mbps	$1,30\times$	$4,37\times$	$6,24\times$	$9,34\times$
Γ_{d_3}/platform (C_3)	$1,26\times$	$1,42\times$	462 Mbps	$1,60\times$	$4,93\times$	$8,67\times$	$11,79\times$
$\mathbf{e}_{d_3}$ in nJ/bit (C_1)	37	32	171	67	155	86	62
$\mathbf{e}_{d_3}$ in nJ/bit (C_2)	32	31	109	83	169	118	79
$\mathbf{e}_{d_3}$ in nJ/bit (C_3)	31	30	108	68	149	85	62

Table 5 Comparisons of software Gallager-E decoders with FPGA hardware implementations [13].

LDPC	Xeon implementation (d_3 with AVX512 ISA)				FPGA implementation [13]			
code	# cores	Γ (Mbps)	Δ (μs)	E (nJ/bit)	#cores	Γ (Mbps)	Δ (μs)	E (nJ/bit)
C_1	1	220	6	819	1	620	2.1	1.94
C_2	1	180	113	956	1	3060	10.7	1.15
C_3	1	247	132	685	1	3020	6.8	1.13
C_1	40	5460	10	61	65	40300	2.1	0.9
C_2	40	4298	191	80	15	45900	10.7	0.97
C_3	40	5446	248	63	15	45300	6.8	0.89

The architecture based on the Application-Specific Instruction set Processor (ASIP) paradigm is built around a microprogrammable controller for flexibility reasons. The architecture structure is optimized for the implementation of QC-LDPC matrices. The number of processing units (Figure 7) scales adequately with the Z expansion factor of the parity check matrices to optimize the hardware resource usage and maximize the throughput. In the same way, the memories are split into Z memory banks to provide the required bandwidth. Each processing unit can process **L2** and **L3** loops in parallel on an independent dataset contrary to multicore architecture that processes **L2** and **L3** sequentially. More information concerning the LDPC decoder implementation could be found in [13].

From a hardware point of view, the number of hardware decoders allocated in the FPGA device varies according to the LDPC code as reported in Table 5. This value was fixed to occupy the FPGA at 75% of its capacity to avoid place & route issues. The working frequencies post-PaR of the overall hardware experiments reach 500 MHz. Results in terms of throughput, latency and energy are given in Table 5.

First, the throughput and latency measurements demonstrate that the FPGA solutions provide $2\times$ to $16\times$ higher decoding throughputs when a single processing core is considered on both platforms. This performance gap despite the favorable working frequency of the Xeon processor is due to the inefficiency of the x86 ISA. Indeed, a large set of basic computations requires several clock cycles on the Xeon processor whereas in a hardware implementation they can be trivially executed in one clock cycle. The observations related to the decoding latencies between the two approaches are similar to the observations made for the throughput. The difference in terms of energy consumption per bit is more

important. Factors are in the range of 315× up to 830×. This is due to the high-power consumption of the Xeon processor when one core is active.

This work highlights the differences in terms of performance between dedicated architectures on FPGA devices and more flexible software solutions. As expected, dedicated hardware solutions are more energy-efficient. However, the gap with optimized software-based implementations that are faster to develop constantly decreases.

5 Conclusion

In this paper, three different parallelized software LDPC decoder implementations are first detailed. These software implementations process the Gallager-E decoding algorithm which is efficient for hard input decoding of LDPC codes contrary to related works that manage soft input values. The parallelization scheme applied, and arithmetic optimizations studied to implement this algorithm on an INTEL Xeon multicore target are detailed. Throughput up to 7,5 Gbps is reported when 10 decoding iterations are executed. Moreover, a comparison in terms of throughput, latency and power measurement is done with other multicore devices and also with FPGA-based implementations. Software versus hardware implementation highlights the efficiency of the proposed software decoder implementations that offer high flexibility and runtime adaptability features. The results obtained show that it is possible to build a real-time prototype based on multi-cores for the development of these future optical communication systems. In addition, the performance levels achieved demonstrate the feasibility of a 10 Gbps software-only receiver for base stations by using multiple Xeon cores and/or reducing the number of decoding iterations. However, the measured power consumption shows that this software-based approach is not viable for satellite use where energy and thermal constraints are high.

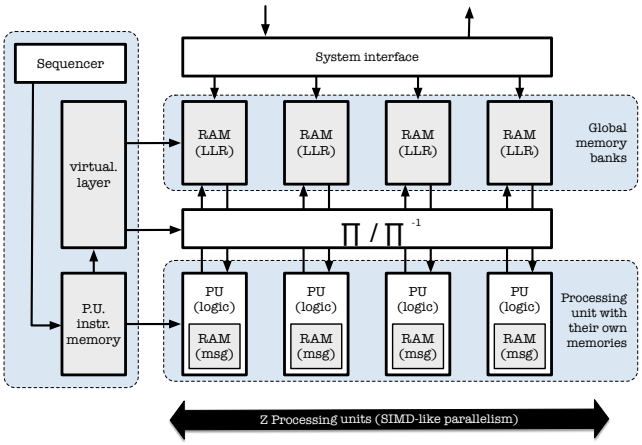

Fig. 7 The Gallager-E hardware decoder architecture detailed in [13]

References

1. C. Marchand and E. Boutillon, "Ldpc decoder architecture for dvb-s2 and dvb-s2x standards," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*, Hangzhou, China, October 2015.
2. V. Pignoly and al., "High data rate and flexible hardware QC-LDPC decoder for satellite optical communications," in *Proceedings of ISTC*, Dec 2018, pp. 1–5.
3. G. Falcao, J. Andrade, V. Silva, and L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection," *Electronics Letters*, vol. 47, no. 9, pp. 542–543, 2011.
4. B. Le Gal and C. Jego, "High-throughput multi-core LDPC decoders based on x86 processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 5, pp. 1373–1386, May 2016.
5. B. Le Gal and C. Jego, "Low-latency software LDPC decoders for x86 multi-core devices," in *Proceedings of SiPS*, 2017.
6. J. Andrade, G. Falcao, V. Silva, and L. Sousa, "A survey on programmable LDPC decoders," *IEEE Access*, vol. 4, pp. 6704–6718, 2016.
7. M. K. Roberts and P. Anguraj, "A comparative review of recent advances in decoding algorithms for low-density parity-check (LDPC) codes and their applications," *Archives of Comput. Methods in Engineering*, 2020.
8. E. Grayver, *Implementing Software Defined Radio*. Springer, 2013.
9. D. Wubben and al., "Benefits and impact of cloud computing on 5G signal processing: Flexible centralization through cloud-RAN," *IEEE Signal Processing Magazine*, vol. 31, no. 6, pp. 35–44, Nov 2014.
10. A. Checko and al., "Cloud RAN for mobile networks - a technology overview," *IEEE Communications Surveys Tutorials*, vol. 17, no. 1, pp. 405–426, Firstquarter 2015.
11. K. Le, F. Ghaffari, L. Kessal, D. Declercq, E. Boutillon, and C. Winstead, "A probabilistic parallel bit-flipping decoder for low-density parity-check codes," *IEEE Trans. on Circuits and Systems I: Regular Papers*, vol. 66, no. 1, pp. 403–416, July 2018.
12. B. Unal, F. Ghaffari, A. Akoglu, D. Declercq, and B. Vasić, "Analysis and implementation of resource efficient probabilistic gallager B LDPC decoder," in *Proceedings of NEWCAS*, June 2017, pp. 333–336.
13. V. Pignoly, B. Le Gal, C. Jégo, and B. Gadat, "Horizontal layered gallager decoding of low-density parity-check codes for wireless up-link optical space communication," in *Proceedings of the ICECS*, Glasgow, Scotland, November 23-25 2020.
14. G. Falcao et al., "Massively LDPC decoding on multicore architectures," *IEEE Trans. on Parallel and Distributed Systems*, vol. 22(2), 2011.
15. P. Murugappa, R. Al-Khayat, A. Baghdadi, and M. Jezequel, "A flexible high throughput multi-ASIP architecture for LDPC and turbo decoding," in *Proceedings of the Design, Automation Test in Europe Conference Exhibition*, ser. (DATE), march 2011, pp. 1 –6.

16. G. Falcao, V. Silva, J. Marinho, and L. Sousa, "LDPC decoders for the wimax (ieee 802.16e) based on multicore architectures," *in WIMAX New Developments. Upena D Dalal and Y P Kosta (Ed.)*, 2009.
17. B. Le Gal, C. Jego, and J. Crenne, "A high throughput efficient approach for decoding LDPC codes onto GPU devices," *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 29–32, 2014.
18. B. Le Gal, C. Jego, and C. Leroux, "A flexible NISC-based LDPC decoder," *IEEE Transactions on Signal Processing*, vol. 62, no. 10, pp. 2469–2479, May 2014.
19. B. Le Gal and C. Jego, "GPU-like on-chip system for decoding LDPC codes," *ACM Transactions on Embedded Computing Systems*, vol. 13, no. 4, pp. 1–19, 2014.
20. B. Le Gal and C. Jego, "High-throughput LDPC decoder on low-power embedded processors," *IEEE Communications Letters*, vol. 19, no. 11, pp. 1861–1864, November 2015.
21. V. Pignoly, B. Le Gal, C. Jégo, and B. Gadat, "Horizontal layered gallager decoding of low-density parity-check codes for wireless up-link optical space communication," in *Proceedings of the IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2020.
22. B. Gokalgandhi and I. Seskar, "Distributed processing for encoding and decoding of binary LDPC codes using MPI," in *Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM)*, Paris, France, May 2019, pp. 596–601.
23. J. Ling and P. Cautereels, "Fast LDPC GPU decoder for Cloud RAN," *IEEE Embedded Systems Letters*, vol. 13, no. 4, pp. 170–173, January 2021.
24. C. Tarver, M. Tonnemacher, H. Chen, J. Zhang, and J. R. Cavallaro, "GPU-based, LDPC decoding for 5G and beyond," *IEEE Open Journal of Circuits and Systems*, vol. 2, pp. 278–290, January 2021.
25. G. Masera, F. Quaglio, and F. Vacca, "Implementation of a flexible LDPC decoder," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 54, no. 6, pp. 542–546, June 2007.
26. S. M. E. Hosseini, K. S. Chan, and W. L. Goh, "A reconfigurable fpga implementation of an ldpc decoder for unstructured codes," in *International Conference on Signals, Circuits and Systems*, 2008.
27. C. Beuschel and H.-J. Pfleiderer, "Fully programmable decoder architecture for structured and unstructured LDPC codes," in *1st International Conference on Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology*, Aalborg, 17-20 May 2009, pp. 747–751.
28. M. Fossorier, "Quasicyclic low-density parity-check codes from circulant permutation matrices," *IEEE Transactions on Information Theory*, vol. 50, no. 8, pp. 1788–1793, July 2004.
29. R. Tanner, D. Sridhara, A. Sridharan, T. Fuja, and D. Costello, "Ldpc block and convolutional codes based on circulant matrices," *IEEE Transactions on Information Theory*, vol. 50, no. 12, pp. 2966–2984, December 2004.
30. Consultative Committee for Space Data Systems (CCSDS), *CCSDS 131.0-B-3 - TM Synchronization and Channel Coding (Blue Book)*, September 2017.
31. *Digital Video Broadcasting (DVB) - Part II: S2-Extensions (DVB-S2X)*, DVB Document A83-2, March 2014.
32. F. Ghaffari and al., "Efficient FPGA implementation of probabilistic gallager B LDPC decoder," in *Proceedings of ICECS*, Dec 2017, pp. 178–181.
33. D. E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS)*, October 2004, pp. 107–112.
34. M. P. C. Fossorier, M. Mihaljevic, and H. Imai., "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on communications*, vol. 47, no. 5, pp. 673–680, 1999.
35. F. Guilloud, E. Boutillon, and J.-L. Danger, "λ-min decoding algorithm of regular and irregular LDPC codes," in *Proceedings of the 3rd International Symposium on Turbo Codes and Related Topics*, Brest, France, September 2003.
36. C. Jones, E. Valles, M. Smith, and J. Villasenor, "Approximate-min* constraint node updating for LDPC code decoding," in *Proceedings of the IEEE Military Communication Conference (MILCOM)*, October 2003, pp. 157–162.
37. J. Chen and M. Fossorier, "Density evolution of two improved BP-based algorithms for LDPC decoding," *IEEE Communication Letters*, vol. 6, no. 5, pp. 208–210, May 2002.
38. J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288–1299, August 2005.
39. C. Marchand, L. Conde-Canencia, and E. Boutillon, "Architecture and finite precision optimization for layered LDPC decoders," *Journal of Signal Processing Systems*, vol. 65, 2011.
40. R. Gallager, *Low density parity-check codes.* IRE Trans. Inform. Theory, 1962.
41. T. Wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding LDPC codes," *IEEE Trans. on Communications*, vol. 58, no. 6, pp. 1610–1614, June 2010.
42. F. Ghaffari and B. Vasic, "Probabilistic gradient descent bit-flipping decoders for flash memory channels," in *Proceedings of ISCAS*, May 2018, pp. 1–5.
43. T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Information Theory*, vol. 47, 2001.
44. G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *Proceedings of the IEEE GlobalSIP Conference*, 2013, pp. 1258–1261.
45. C. Beuschel and H.-J. Pfleiderer, "FPGA implementation of a flexible decoder for long LPDC codes," in *International Conference on Field Programmable Logic and Applications*, ser. (FPL'08), 8-10 September 2008, pp. 185–190.
46. J. Cardoso, S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "Fpga-based channel coding architectures for 5g wireless using high-level synthesis," *International Journal of Reconfigurable Computing, Hindawi*, April 2017.
47. A. Katyushnyj, A. Krylov, A. Rashich, C. Zhang, and K. Peng, "Fpga implementation of ldpc decoder for 5g nr with parallel layered architecture and adaptive normalization," in *Proceedings of the IEEE International Conference on Electrical Engineering and Photonics (EExPolytech)*, St. Petersburg, Russia, October 2020, pp. 34–37.
48. D. Padua, Ed., *Encyclopedia of Parallel Computing.* Springer, 2011.

49. H. Hum and G. Gao, "Supporting a dynamic SPMD in a multi-threaded architecture," in *Proceedings of the Digest of Papers, Compcon Spring*, San Francisco, CA, USA, February 1993.
50. R. M. D. R. E. Luque, "How spmd applications could be efficiently executed on multicore environments?" in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*, New Orleans, LA, USA, August 2009.
51. T.-H. Weng, S.-W. Huang, W. W. Ro, and K.-C. Li, "Implementing FFT using SPMD style of OpenMP," in *Proceedings of the 6th International Conference on Networked Computing and Advanced Information Management*, Seoul, Korea, August 2010.
52. H. Tanaka, Y. Ota, N. Matsumoto, T. Hieda, Y. Takeuchi, and M. Imai, "A new compilation technique for simd code generation across basic block boundaries," in *Proceedings of 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*, Taipei, Taiwan, January 2010.
53. P. Estérie, M. Gaunard, J. Falcou, J.-T. Lapresté, and B. Rozoy, "Boost.SIMD: Generic programming for portable SIMDization," in *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Minneapolis, MN, USA, September 2012.
54. A. Barredo, J. M. Cebrian, M. Moretó, M. Casas, and M. Valero, "Improving predication efficiency through compaction/restoration of SIMD instructions," in *Proceedings of IEEE International Symposium on High Performance Computer Architecture (HPCA)*, San Diego, CA, USA, February 2020.
55. W. m. Hwu, *Superscalar Processors, Encyclopedia of Parallel Computing*. Boston, MA: Springer US, 2011, pp. 1962–1966.
56. P. Giard, G. Sarkis, C. Leroux, C. Thibeault, and W. J. Gross, "Low-latency software polar decoders," *Journal of Signal Processing Systems, Springer*, July 2016.
57. A. Cassagne, O. Hartmann, M. Léonardon, K. He, C. Leroux, R. Tajan, O. Aumage, D. Barthou, T. Tonnellier, V. Pignoly, B. Le Gal, and C. Jégo, "Aff3ct: A fast forward error correction toolbox!" *Elsevier SoftwareX*, vol. 10, 2019.