High-Performance Gallager-E Decoders for Hard Input LDPC Decoding on Multi-core Devices

Bertrand Le Gal(✉), Vincent Pignoly, and Christophe Jego

IMS laboratory (UMR 5218), Bordeaux INP, Univ. Bordeaux, Talence, France
{bertrand.legal,vincent.pignoly,christophe.jego}@ims-bordeaux.fr

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 multi-core devices facilitate real-time implementations of digital communication systems, which were previously implemented thanks to dedicated hardware circuits. A lot of works were done over the last decade on the implementation of Gbps decoders on programmable devices. However, these works focus on soft-input LDPC decoding algorithms. But, hard-input LDPC decoders are also required to design and prototype optical-based satellite communication systems. 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, throughputs up to 7.5 Gbps are achieved when 10 Gallager-E iterations are executed onto an INTEL Xeon device.

Keywords: LDPC · Gallager E · multi-core · SIMD · High-throughput

1 Introduction

Low-Density Parity-Check (LDPC) codes are a popular class of Error Correction Codes (ECC) used in digital communication systems to provide reliability. Due to their excellent error correction performance, LDPC codes were selected for terrestrial wireless standards (e.g., WiFi and 5G) but also in RF space ones (CCSDS, DVB-S2 and DVB-S2x). FPGA or ASIC technologies were during a long time the single way to provide real-time LDPC decoding when hundreds of Mbps or Gpbs are targeted. Indeed, LDPC decoding algorithms are characterized by a high computational complexity discarding other kinds of implementations. These dedicated hardware implementations provide high-throughput and low-energy features at the costs of low-flexibility and low-reusability.

For ten years, the computational power offered by multi-core or many-core devices associated with easy-to-use programming models opened new horizons. Indeed, coping with low-flexibility and long development cycles, researchers and industrials tried to use these programmable devices to implement ECC decoders

K. Desnos and S. Pertuz (Eds.): DASIP 2022, LNCS 13425, pp. 3–15, 2022.
https://doi.org/10.1007/978-3-031-12748-9_1

that are the receiver design bottlenecks [1,2,11,18]. Programmable architectures associated with optimized software descriptions made possible the implementation of high-throughput receiver systems. They can be used as real life wireless communication systems and/or prototype for next generation ones.

Software Defined Radio (SDR) [9] or cloud-RAN [3,21] 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 soft-input Min-Sum (MS) algorithm. Unfortunately, to reach a high throughput of several Gbps in optical space communications, ECC decoders can be limited to process hard input values due to current optical technology limitations.

Hard-input decoding presents lower error correction performance than soft-input ones. Moreover, it needs the implementation of other LDPC decoding algorithms such as Gallager-B, Gallager-E or their variations (e.g., Bit flipping algorithms [10], Probabilistic Gallager-B [19]). Even if efficient FPGA implementations of Probabilistic Gallager B (PGaB) and Gallager-E are detailed in [16,19]. Implementing them efficiently in software is not an easy task. 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. At the opposite, the Gallager-E decoding algorithm has a formulation closed to the MS one used in related works. Its computation parallelism is almost regular but its high memory footprint and the logical bit-level computations are not clearly adapted to software processor targets. However, in this work we focused on it and propose its efficient SIMD (Single Instruction Multiple Data) and SIMT (Single Instruction Multiple Threads) 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 the applied optimizations are provided in Sect. 3. Section 4 summarizes the experimental results obtained with the proposed decoder implementations. Finally, conclusion and future works are reported.

2 LDPC Decoding Algorithm

An LDPC code is a linear block code defined by a binary sparse $M \times N$ parity-check matrix called $\mathbf{H}$. This $\mathbf{H}$ matrix is composed of N columns representing the received bit information (VN) from the channel whereas the $M = N - K$ rows are associated to parity-check information (CN) with K the number of information bits in the received frame. To ease the implementation of LDPC decoders, a special class of LDPC codes is used in standards. Quasi-Cyclic (QC) LDPC codes are codes that are composed of an array of $Z \times Z$ shifted identity sub-matrices. Z is the order that defines the computation parallelism level. It eases $\mathbf{H}$ matrix storage and ensures that Z CNs can be processed in parallel without conflicts independently of the computation scheduling.

Algorithm 1. Horizontal-layered Gallager E

▷ Input (received word) : $\boldsymbol{y} = (y_1, y_1, ..., y_N) \in \{0,1\}^N$
init
$t = 0$, $Y_i = 2y_i - 1$ and $V_i = 0$ for $i \in [1, .., N]$
repeat
▷ **L1** - loop 1: Horizontal layered scheduling
for all j ∈ [1, .., M] **do**
$C_j^{(t)} = 1$, $Sum0 = 0$
▷ **L2** - Loop 2
for all i ∈ N(j) **do**
▷ **L1S1** - Stage S1: Variable to parity check message $v2c_{ij}^{(t)}$ processing

$$M_{ij} = \begin{cases} V_i, & t = 0 \\ V_i - c2v_{ji}^{(t-1)}, & \text{otherwise} \end{cases}$$

$$v2c_{ij}^{(t)} = sign(M_{ij} + \omega^{(t)} * Y_i)$$

▷ **L1S2** - Stage S2: 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** $Sum0 = Sum0 + 1$ **end if**
end for
▷ **L3** - Loop 3
for all i ∈ N(j) **do**
▷ **L3S1** - Stage S1: Check to variable message $c2v_{ji}^{(t+1)}$ processing

$$c2v_{ji}^{(t)} = \begin{cases} 0, & Sum0 \geqslant 2 \\ 0, & Sum0 = 1 \text{ and } M_{ij}^{(t)} \neq 0 \\ C_j^{(t)}, & Sum0 = 1 \text{ and } M_{ij}^{(t)} = 0 \\ C_j^{(t)} \times M_{ij}^{(t)}, & \text{otherwise} \end{cases}$$

▷ **L3S2** - Stage S2: Variable node 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) : $\boldsymbol{x} = (x_1, x_1, ..., x_N) \in \{0,1\}^N$

Usually, the LDPC decoding process is performed thanks to a message passing (MP) approach where VNs and CNs exchange m messages. When the decoding process benefits from soft-input, the sum-product algorithm (SPA) approximations such as Min-Sum (MS) variants are applied [4,13].

Hard-input constraint involves another algorithmic choice. Gallager-B algorithm was initially proposed in [5] to process hard-input values. However, its decoding performance is relatively low due to binary values used to represent exchanged messages. This algorithm was recently improved in terms of error correction power by inserting for instance decoding noise (e.g. Probabilistic Gallager-B [19]) or using gradient descent based algorithm [7,20]. These algorithms (PGaB, GDBF and PGDBF) were developed for hardware decoders manipulating for short frame sizes. However, in the current 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 computa-

tion irregularity should be avoided. Consequently, they were discarded. Original Gallager B decoding algorithm [14] can be extended by considering erasures in message passing. This extended algorithm, called Gallager E algorithm in [17], manages ternary values $-1, 0, +1$ for exchanged messages instead of binary $-1, +1$ ones in the Gallager B algorithm. This third value that represents doubt during the decoding process, drastically improves error correction performance.

Gallager E algorithm is summarized in Algorithm 1. Horizontal layered scheduling was selected because it improves error correction performances whereas at the same time it reduces computation and memory complexities [16]. Algorithmic structure is closed to the one used for MS decoding [15]. However, contrary to MS algorithms that execute 8b arithmetic operations (addition, minimum and comparison), Gallager E algorithm needs on its side 1b or 2b logic operations and conditional branches to execute non-reversible operations such as voting.

Received bits Y_i come from the channel. The initial values of accumulator nodes V_0 should be null as presented in [16]. $C^{(tj}$ corresponds to the check node values during the iteration t, with t the current iteration and t_{max} the maximum number of decoding iterations. $V_{ij}^{(t)}$ represents the value of the vote for node VN_i. The possible value set for $V_{ij}^{(t)}$ message is in range $\{-1, 0, +1\}$. Message $c2v_{ji}^{(t+1)}$ from CN_j to VN_i is the product of incoming messages except $v2c_{ij}^{(t)}$. If one or more input messages equal zero then the output message equals the null value too. In this step, contrary to MS decoding algorithm, the $c2v_{ji}^{(t+1)}$ value cannot be easily deduced from $v2c_{ij}^{(t)}$ because of the vote operation that cannot be inverted. Moreover, as the vote operation could not be inverted, the Y_i values should be kept in memory during the overall decoding process. It increases the memory footprint of N elements in comparison to MS decoder implementations.

3 Parallelization Strategies

3.1 Targeted Multi-core System

INTEL x86 processors currently provide different parallelization features. At the higher level, the processor circuits include many independent physical cores that can be used to process tasks or sub-tasks in parallel (SIMT). The number of cores can reach up to dozens for server grade processors. At the same time, each physical processor core includes SIMD (Single Instruction Multiple Data) units that execute parallel computations. SIMD units are 128b up to 512b wide, authorizing up to 64× 8b computations per instruction. Finally, at the lower level, x86 processors are superscalar and thus implements Instruction-Level Parallelism (ILP) authorizing multiple instructions to be scheduled within a single clock cycle according to the resource availability. To reach efficiency, all these aspects should be addressed together. The LDPC decoding parallelization strategy and the optimizations are reported in this section.

3.2 SIMD Parallelization

Parallelization of message passing algorithm was widely studied in the context of traditional MS decoder implementations [1]. Two main SIMD parallelization strategies for multi-core targets where proposed [2,11]. These generic approaches that can also be applied for the Gallager-E algorithm provide different advantages and drawback effects.

Inter-frame parallelization strategy [2] takes advantage of SIMD units to decode multiple frames in parallel. It eases the software description and provides regular computation parallelism at runtime. Indeed, when the number of frames processed in parallel ($Q \times 8$b) equals the SIMD width, the SIMD efficiency is constant to 100%. The inter-frame strategy drawback effect comes from the memory footprint (Δ_{inter}). Indeed, this footprint becomes quickly larger than L1, L2 and L3 memory caches. Its high memory bandwidth requirement limits the scalability of the decoder implementations but also provides high processing latency and impact on global system performance due.

$$\Delta_{\text{inter}} = Q \times (2 \times N + m) + m \tag{1}$$

At the opposite, intra-frame strategy [11] takes advantage of SIMD units to parallelize internal computations from a single frame. This strategy limits the memory footprint (Δ_{intra}) at runtime. However, it complexifies the software description of the decoder and slow down memory accesses. Indeed, **H** structure management is done at runtime and involves complex memory accesses. Moreover, depending on the LDPC code, a SIMD usage rate of 100% is rarely obtained. However, from a system point of view, the intra-frame implementation delivers low-latency feature and limits its impact on other processing elements.

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

Both SIMD parallelization strategies can be applied to speed-up the execution of loop 1 defined in Algorithm 1. In this work, both approaches are evaluated because they provide different features and thus different trade-offs.

3.3 ILP Improvement

An efficient implementation of loops 2 and 3 in Algorithm 1 is crucial because they are executed M times per decoding iteration. Consequently, a specific mapping of the algorithmic operations on available SIMD instruction is required. Gallager-E decoding algorithm mainly manipulates bit or ternary values and has many conditional instructions. So Gallager-E decoding is more challenging to efficiently implement on processor cores than MS ones [15].

First, to reduce the complexity of the M_{ij} computation (**L2S1**) that depends on the decoding iterations, an initialization of the messages to zero was done before the decoding. It avoids the comparison and conditional moves at runtime. Then for the computation of $v2c$ messages, as ω is in range $0, 1$, the multiplication operation is implemented thanks to a logical *and* instruction whose

second operand is a binary mask (0x00, 0xFF). Finally, the counter (Sum0) can be approximated: its value is increased by the result of the comparison instruction that is $\{0x00, 0xFF\}$ and nor by 1. This approximation is possible because in **L3S1**, the first part of the conditional structure can be reformulated as $Sum0 - (M_{ij} = 0) > 0$. This tricky optimization removes logical instructions and comparisons from the execution critical path. Moreover, it eases the implementation of the $c2v$ conditional computation making possible to describe it as a value selection in range $0, 1$, and then a conditional sign inversion to regenerate a message in range $\{-1, 0, +1\}$.

After these transformations, the number of instructions needed 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). Consequently, to improve the ILP thanks to instruction execution overlapping and also remove useless control instructions, specialized kernel codes are generated at compile time. To this end, the features of C++11 language (i.e., template specialization) are applied like in [8]. For each CN degree value, a dedicated and optimized kernel is generated. Consequently, the number of instructions is then minimized for each **L1** loop execution. At runtime, an array of function pointers is used to select the right binary code to execute.

3.4 Memory Compression

The memory bandwidth is a bottleneck for software implementation of LDPC decoders when long frame are processed. Decoder memory footprint becomes quickly higher than memory caches. Contrary to MS-based decoder implementations that manage 8b values internally for all data, Gallager-E decoding algorithm manipulate only bit or ternary values. Naively, all these values consume 8b in memory because they are involved in 8b arithmetic operations. However, as the memory bandwidth is a bottleneck, a memory compression technique was developed. It divides the memory footprint by 4 for exchanged messages ($c2v$) that are ternary values and divides by 8 the footprint for channel values (Y_i) that are binary values. Memory footprint reduction necessitates the execution of additional SIMD instructions at runtime. However, both compression and decompression tasks are executed with a low latency penalty on x86 architecture whereas a single memory cache miss can produce a penalty of some hundreds of clock cycles. The compression and the decompression tasks can be done easily and efficiently thanks to the source codes provided in Listing 1. Note that data compression does not impact on error correction performances because values that are stored on 8b are in reality 2b or 1b.

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 execution of the loop 1 (**L1**) defined in Algorithm 1. However, this

```
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, r_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 1.1. SIMD functions used for (de)compression of binary and ternary values

approach is inefficient because: (1) loop elements are not independent when horizontal layered based decoding algorithm is applied due to the fact that memory access conflicts can occur, and (2) the time spend in forking and joining tasks is not negligible compared to **L1** execution time. The best way to increase the performance level is to allocate **P** independent LDPC decoders to process **P** distinct frames. It avoids L1/L2 memory sharing at runtime between the cores but increase L3 cache usage. It also increases the pressure on the memory bandwidth by a factor **P**. In the current work to enable an asynchronous behavior of the decoders, the LDPC decoders are encapsulated in C++11 threads.

4 Experimentation Results

4.1 Experimentation Setup

The software-based LDPC decoder implementations were developed in C++ 11 language. The targeted device was an INTEL Xeon Gold processor. Consequently, the AVX512 instruction subset was selected. To benefits from INTEL SIMD features, INTEL intrinsics which are C-style functions were applied. 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 multi-core system composed of a dual socket INTEL Xeon Gold 6148 CPU. Each Xeon processor has 20 physical processor cores. The overall processor cores shares a 28160K L3 memory cache and 256 GB of RAM. A working frequency up to 3, 70 GHz is achievable on this platform thanks

Table 1. Selected QC-LDPC codes for benchmarking purposes

Code	C_1 [6]	C_2	C_3
(N, K)	$(1296, 648)$	$(32768, 16384)$	$(20480, 16384)$
Z	54	256	256
d_c	$\{8\}$	$\{7, 8, 9\}$	$\{19, 20\}$
# msgs	5184	131072	81664

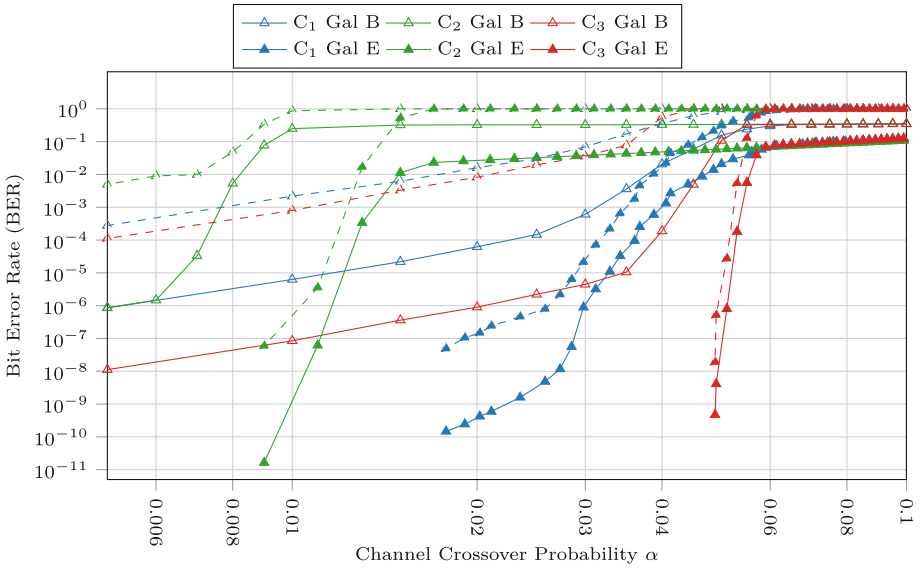

Fig. 1. Error correction performance comparison of Gallager-B and Gallager-E algorithms when 10 layered decoding iterations are executed.

to turbo boost 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.

4.2 Error Correction Performance

Before benchmarking the throughput efficiency of the proposed Gallager-E parallelization schemes, a validation of its BER and FER performances was done. To check its behavior, three different QC-LDPC codes were used. The first code comes from related works on hard-input LDPC decoding [6]. The two others are custom LDPC codes developed specifically for spatial optical communications [16]. The main characteristics of the codes are summarized in Table 1.

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

		Γ in Mbit/s			Δ in μs			**P** in Watts			**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

The bit error rate (BER) and frame error rate (FER) performances for C_1 to C_3 LDPC codes are reported in Fig. 1 when a Monte-Carlo simulation is built on a BSC channel and a OOK modulation. As expected, the curves show that Gallager-E algorithm outperforms the Gallager-B ones in the overall use-cases. These results are consistent with the published literature. They demonstrate the correct functionality of the LDPC decoders implemented in this work.

4.3 Absolute Performances

For benchmarking purpose, a communication system simulation runs during a period of 120 seconds to avoid working frequency scaling impact on average values. The throughput (Γ) and the latency (Δ) results are reported in Table 2 when 10 decoding iterations are executed. The following setups are evaluated:

- **Inter-frame setup** (d_1) - Each physical processor core decodes $Q = 64$ LDPC 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 process $Q = 64$ 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 ($Q = 1$). All the values (channel, accumulators and messages) are stored on 8 bits.

Results presented in Table 2 show that in single core configuration, throughput from 77 Mbps up to 294 Mbps was measured for the d_1 implementation. The highest throughput was obtained when C_1 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 throughput is divided up to 4× due to a memory footprint that grows up to 12125 KB. The d_2 implementation gives higher throughput for C_2 and C_3 codes due to memory compression that decreases the memory footprint to 4375 KB. However, the additional arithmetic and logical computations used to compress the information at runtime

in d_2 makes it less efficient for C_1 code. At the opposite, the d_3 implementation provides the highest decoding throughput for C_2 and C_3 (from $\approx 1.2\times$ up to $\approx 2.4\times$ higher than d_2) with a maximum memory footprint of 194 KB. In parallel, the processing latency of d_3 implementation is about 99% shorter than d_2 ones. However, the usage rate of the SIMD units for d_3 is lower than 100% at runtime for C_1 LDPC code ($Z = 54$). Complex memory accesses and SIMD inefficiency in this case is not compensated by L1/L2 cache efficiency.

Experimentations were done when 20 or 40 processor cores are activated to check the scalability of the decoder implementations. The results show that the d_1 implementation is better in terms of throughput for C_1 code in all setups with speedup factors of $15\times$ and $25\times$. However, its performances fall down for C_2 and C_3 codes where the speedup factors are limited to $7\times$ to $10\times$. Indeed, even if the number of cores is increased from 20 to 40, a performance floor due to the maximal memory bandwidth appears. Memory bottleneck assertion is validated by the results obtained for d_2 implementation. Indeed, the d_2 implementation that executes memory compression over-classed d_1 one for the long frames (C_2 and C_3) codes. The throughput grows with the number of activated cores. However, this the performance improvement is not linear due to working frequency scaling. For d_2 decoder implementation, the measured speedups are $17\times$ and $24\times$ when 20 cores and 40 cores are activated, respectively. This phenomenon is due to the turbo-boost frequency scaling feature. In 20 and 40 core setups, the d_3 implementation offers the better performances and achieved up to 5446 Mbps. The speedup factors obtained in comparison with single core experiments are $14\times$ and $24\times$. At the same time, the working frequency reduction increases the processing latency by a factor of 2.

In parallel to the throughput and latency evaluation of the implementations, power and energy per bit comparisons were also done. Energy measurements are provided in Table 2. The reported energy consumption values include the CPU and the RAM consumptions. These values were obtained at runtime with the *turbostat* tool that captures the power consumption of sensors. The power consumption depends on the number of activated cores and the RAM usage rate. The power consumption results are equivalent to C_1 code for all implementations. However, for C_2 and C_3 codes, the d_1 and d_2 implementations have a higher power consumption ($\approx 40\%$) than d_3 due to their high usage rate of the RAM. Indeed, 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.

4.4 Comparison with FPGA-Based Decoder Implementations

Finally, the proposed Gallager-E LDPC decoder implementations on multi-core devices were compared with FPGA-based ones [16] to estimate the feature differences. Indeed, works presented in [16] details hardware optimized Gallager-E architectures implemented in a Zynq Ultrascale+ FPGA (xczu9eg-3ffvb1156e) based on architecture described in [12]. The number of hardware decoders allocated in the FPGA device varies according to the LDPC code concerned as

reported in Table 3. 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. Table 3 summarized for performance levels reached in terms of throughput, latency and energy.

The throughput and latency measurements first demonstrate that the FPGA solution provides 2× to 16× higher decoding throughput 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 computation requires on several clock cycles on the Xeon processor whereas in hardware they can be trivially implemented in one clock cycle. The observations related to the decoding latency between the two types of systems is in line with the observations made on throughput. The difference in terms of energy consumption per bit is more important. Factors are in the range 315× up to 830×. This is due to the high-power consumption of the Xeon processor when one core is active.

Table 3. Comparisons of software Gallager-E decoders with FPGA ones [16].

LDPC	Xeon implementation				FPGA implementation [16]			
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

Multi-core execution modifies the previous acknowledgment. The throughput difference is in this setup in range 5× to 11×. Indeed, the number of decoding cores that can be instantiated in the FPGA 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 because 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 difference in terms of power consumption between the FPGA implementation and the multi-core implementation varies finally from 52× to 82×.

This comparative section highlights the differences in terms of performance between dedicated architectures on FPGA targets 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 parallelized software LDPC decoder implementations are detailed. These software implementations implement the Gallager-E decoding algorithm which is efficient for hard input decoding of LDPC codes contrary to related work that manage soft-input values. The parallelization scheme applied and arithmetic optimizations used to implement this algorithm on INTEL Xeon multi-core target are detailed. Throughput up to 7,5 GBps is reported when 10 decoding iterations are executed. Finally, a comparison of throughput, latency and power measurements is done with FPGA-based implementation to highlight the efficiency of the proposer decoder implementations.

References

1. Andrade, J., Falcao, G., Silva, V., Sousa, L.: A survey on programmable LDPC decoders. IEEE Access **4**, 6704–6718 (2016)
2. Le Gal, B., Jego, C.: High-throughput multi-core LDPC decoders based on x86 processor. IEEE Trans. Parallel Distrib. Syst. **27**(5), 1373–1386 (2016)
3. Checko, A., et al.: Cloud RAN for mobile networks - a technology overview. IEEE Commun. Surv. Tutorials **17**(1), 405–426 (2015)
4. Chen, J., Fossorier, M.: Near optimum universal belief propagation based decoding of low-density parity check codes. IEEE Trans. Commun. **50**(3), 406–414 (2002)
5. Gallager, R.: Low density parity-check codes. IRE Trans. Inform. Theory **8**, 21–28 (1962)
6. Ghaffari, F., et al.: Efficient FPGA implementation of probabilistic Gallager B LDPC decoder. In: Proceedings of ICECS, pp. 178–181, December 2017
7. Ghaffari, F., Vasic, B.: Probabilistic gradient descent bit-flipping decoders for flash memory channels. In: Proceedings of ISCAS, pp. 1–5, May 2018
8. Giard, P., Sarkis, G., Leroux, C., Thibeault, C., Gross, W.J.: Low-latency software polar decoders. J. Sig. Process. Syst. **90**, 761–775 (2016)
9. Grayver, E.: Implementing Software Defined Radio. Springer, New York (2013). https://doi.org/10.1007/978-1-4419-9332-8
10. Le, K., Ghaffari, F., Kessal, L., Declercq, D., Boutillon, E., Winstead, C.: A probabilistic parallel bit-flipping decoder for low-density parity-check codes. IEEE Trans. Circuits Syst. I Regul. Pap. **66**(1), 403–416 (2018)
11. Le Gal, B., Jego, C.: Low-latency software LDPC decoders for x86 multi-core devices. In: Proceedings of SiPS (2017)
12. Le Gal, B., Jego, C., Leroux, C.: A flexible NISC-based LDPC decoder. IEEE Trans. Sig. Process. **62**(10), 2469–2479 (2014)
13. Marchand, C., Boutillon, E.: LDPC decoder architecture for DVB-S2 and DVB-S2x standards. In: Proceedings of SiPS, pp. 1–5, October 2015
14. Mitzenmacher, M.: A note on low density parity check codes for erasures and errors. SRC Technical Note 1998-017 (1998)
15. Pignoly, V., et al.: High data rate and flexible hardware QC-LDPC decoder for satellite optical communications. In: Proceedings of ISTC, pp. 1–5, December 2018
16. Pignoly, V., Le Gal, B., Jégo, C., Gadat, B.: Horizontal layered Gallager decoding of low-density parity-check codes for wireless up-link optical space communication. In: Proceedings of the ICECS, Glasgow, Scotland, 23–25 November 2020

17. Richardson, T.J., Urbanke, R.L.: The capacity of low-density parity-check codes under message-passing decoding. IEEE Trans. Inf. Theory **47**, 599–618 (2001)
18. Roberts, M.K., Anguraj, P.: A comparative review of recent advances in decoding algorithms for Low-Density Parity-Check (LDPC) codes and their applications. Arch. Comput. Methods Eng. **28**, 2225–2251 (2020)
19. Unal, B., Ghaffari, F., Akoglu, A., Declercq, D., Vasić, B.: Analysis and implementation of resource efficient probabilistic Gallager B LDPC decoder. In: Proceedings of NEWCAS, pp. 333–336, June 2017
20. Wadayama, T., Nakamura, K., Yagita, M., Funahashi, Y., Usami, S., Takumi, I.: Gradient descent bit flipping algorithms for decoding LDPC codes. IEEE Trans. Commun. **58**(6), 1610–1614 (2010)
21. Wubben, D., et al.: Benefits and impact of cloud computing on 5G signal processing: flexible centralization through cloud-RAN. IEEE Sig. Process. Mag. **31**(6), 35–44 (2014)