



# High-performance software implementations of SCAN decoder for polar codes

Bertrand Le Gal<sup>1</sup> · Camille Leroux<sup>1</sup> · Christophe Jego<sup>1</sup>

Received: 16 June 2016 / Accepted: 24 March 2018 / Published online: 11 April 2018  
© Institut Mines-Télécom and Springer International Publishing AG, part of Springer Nature 2018

## Abstract

This paper presents the first optimized software implementation of a SCAN decoder for Polar codes. Unlike SC and SC-List decoding algorithms, the SCAN decoding algorithm provides soft outputs (useful for, e.g., parallel concatenated decoders Zhang et al. IEEE Trans Commun 64(2):456–466 2016). Despite the strong data dependencies in the SCAN decoding, two highly parallel software implementations are devised for x86 processor target. Different parallelization strategies, algorithmic improvements, and source code optimizations were applied in order to enhance the throughput of the decoders. The impact of the parallelization approach, the code rate, and the code length on the throughput and the latency is investigated. Extensive experimentations demonstrate that the proposed software polar decoder can exceed 600 Mb/s on a single core and reaches multi-Gb/s when using four cores simultaneously. These decoders can then achieve real-time performance required in many applications such as software defined radio or cloud-RAN systems where network physical layer is implemented in software.

**Keywords** Polar codes · SCAN decoding · Soft-inputs and Soft-outputs · SIMD · Multi-core · x86 processor

## 1 Introduction

Polar codes [2] are of a high practical interest since they provide the possibility to implement efficient decoders with good error correction performance. As such, they were actually selected as one of the channel codes for the future 5G mobile communication standard. The original decoding algorithm for Polar codes is the Successive Cancellation decoding algorithm (SC). Three improved decoding techniques were then subsequently proposed: SC-List decoding algorithm [3, 4], the soft-output Belief Propagation (BP) algorithm [5, 6] and the Soft-CANcellation (SCAN) algorithm [7–9]. The SC-List decoding algorithm provides close-to-optimum decoding performance and was selected

for the future 5G mobile communication standard. The BP and SCAN algorithms have performance comparable to the SC decoder but have the advantage of providing a soft output. This feature can be exploited in a concatenated code structure [1, 10] or more generally in an iterative system where the decoder exchanges information with other components of the communication chain (the demodulator, the equalizer, ...). In the case of polar codes, to date, only the BP and SCAN decoding algorithms can be used in these so-called turbo-receivers. As an illustration, in [11–15], different channel coding/modulation schemes require a soft-output polar decoder.

On the implementation side, as an alternative to dedicated hardware, the optimization of software implementations is currently an active field for all error correction code (ECC) families, e.g., LDPC [16] and turbo codes [17]. Optimizing a software decoder allows to (i) quickly evaluate and compare new decoding algorithms or code families; and (ii) meet real-time execution constraint (throughput and latency) of software-defined radio (SDR) [18] or Cloud-RAN systems (CR) [19].

Efficient SC decoding algorithm software implementations have been investigated in [20, 21]. Several Gbps on an x86 single core was reached and up to 100 Mbps on a

---

✉ Bertrand Le Gal  
bertrand.legal@ims-bordeaux.fr  
Camille Leroux  
camille.leroux@ims-bordeaux.fr  
Christophe Jego  
christophe.jego@ims-bordeaux.fr

<sup>1</sup> IMS Laboratory, UMR CNRS 5218, Bordeaux-INP, Talence, France

low-power processor targets [22]. In [23] and [4], the authors have considered the SC-List decoding algorithm on account of its enhanced error correction performance compared with the SC decoding algorithm with drastically reduced throughput levels. However, to date, none of the soft output polar decoding algorithm (BP and SCAN) were optimized in software. The BP algorithm has a high computation parallelism. Yet, it has a high computational complexity while providing a moderate error correction performance which limits the interest of it from practical software implementation [24]. The SCAN algorithm, which is also iterative, exhibits not only a better correction performance than the BP algorithm, but also a lower computation complexity. Consequently, it has been subject to thorough algorithmic and hardware research works, e.g., in [24, 25]. In [25], a simplified SCAN algorithm named RC-SCAN is detailed. In this paper, we focus on the implementation of the RC-SCAN decoding algorithm on multi-core processors. This is a first step towards the inclusion of SCAN decoders in software defined “turbo-receivers”.

The remainder of the paper is organized as follows. Section 2 describes polar codes, the SCAN and RC-SCAN decoding algorithms. In Section 3, all the optimization techniques that contribute to the speedup of the decoding process are detailed. In Section 5, the experimental setup is introduced and experimental results are summarized. Section 6 concludes the paper.

## 2 Polar codes

### 2.1 Definition and encoding process

Polar codes are linear block codes of size  $N = 2^n$  with  $n$  a natural number. In [2], Arikan defined their construction based on the  $n$ th Kronecker power of a kernel matrix  $\kappa = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ . The encoding process consists in multiplying  $\kappa^{\otimes n}$  by a  $N$ -bit vector  $U$  that includes  $K$  information bits and  $N - K$  frozen bits which are set to a known value. This matrix multiplication can be efficiently implemented by using a recursive function. This allows a reduction of the encoding complexity to only  $N \log N$  operations. The location of the frozen bits in the vector  $U$  depends on both the considered channel type and its noise power [26].

### 2.2 Original SCAN algorithm

The SCAN decoding algorithm has a sequential nature and thus has strong data dependencies. This limits the amount of

parallelism that can be exploited. As suggested in [27] for the SC decoding algorithm, the SCAN decoding algorithm can be represented in the form of a tree that is recursively traversed in the following order: root node, left child node then right child node. In Fig. 1, a graph representation of an  $N = 8$  SCAN decoder is provided. Nodes are labeled  $\mathcal{N}_p^d$  with  $d$  the depth of the node ( $d = n$  for root node and  $d = 0$  for leaf nodes) whereas  $p$  provides the node position at the depth  $d$  (from left to right). Except for the leaf node, each node  $\mathcal{N}_p^d$  stores internal LLR values named  $\lambda_E^{d,p}$  and  $\beta_E^{d,p}$ . The data set  $E$  is composed of  $D = 2^{d-1}$  values.  $\lambda$  values are evaluated in the descending order (step ① then ②) whereas  $\beta$  are computed in the ascending order (step ③) according to child node results.

After being sent over the transmission channel, the noisy version of the codeword  $X$  is received in the form of log-likelihood ratios (LLRs) and denoted as  $Y$ .

In Fig. 1, the root node  $\mathcal{N}_0^3$  receives the channel information  $Y$  composed of 8 LLR values, and it successively exchanges data with its left child node (①) and its right child node (②) named  $\mathcal{N}_0^2$  and  $\mathcal{N}_1^2$  respectively. Assuming that a non-leaf/non-root node  $\mathcal{N}_p^d$  receives  $\lambda_{[0,D-1]}^{d,p}$ . In order to compute  $\beta_{[0,D-1]}^{d,p}$  (③), it should first recursively evaluate its child nodes as described in Algorithm 1. The  $f_1$  and  $f_2$  functions used in Algorithm 1 are defined as follows:

$$f_1(a, b, c) = f(a, b + c) \tag{1}$$

$$f_2(a, b, c) = f(a, b) + c \tag{2}$$

with:

$$f(a, b) = \text{sgn}(a.b). \min(|a|, |b|) \tag{3}$$

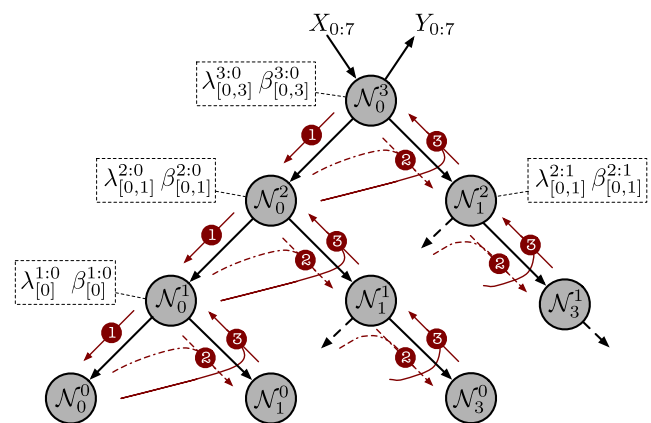


Fig. 1 Recursive tree representation of a  $N = 8$  SCAN decoder

**Algorithm 1** SCAN decoding algorithm.

---

```

1: function PROCESSNODE( $\lambda_{[0,D]}$ ,  $d$ ,  $p$ )
2:   Step 1: Calculates  $D/2$   $\lambda$  values for left child (❶)
3:   for all  $i \in [0, D/2[$  do
4:      $\lambda_i^{d-1,2p} = f_1(\lambda_i^{d,p}; \lambda_{i+\frac{D}{2}}^{d,p}; \beta_i^{d-1,2p+1})$ 
5:   end for
6:   Step 2: Recursively evaluates the left child
7:    $\beta_{[0,\frac{D}{2}]^k} = \text{ProcessNode}(\lambda_{[0,D/2]}^{d-1,2p}, d-1, 2p)$ 
8:   Step 3: Calculates  $D/2$   $\lambda$  values for right child (❷)
9:   for all  $i \in [0, D/2[$  do
10:     $\lambda_i^{d-1,2p+1} = f_2(\beta_i^{d-1,2p+1}; \lambda_i^{d,p}; \lambda_{i+\frac{D}{2}}^{d,p})$ 
11:  end for
12:  Step 4: Recursively evaluates the right child
13:   $\beta_{[0,\frac{D}{2}]^{d-1,2p+1}} = \text{ProcessNode}(\lambda_{[0,\frac{D}{2}]^{d-1,2p+1}}, d-1, 2p+1)$ 
14:  Step 5: Calculates  $\frac{D}{2}$   $\beta$  values (❸)
15:  for all  $i \in [0, D/2[$  do
16:     $\beta_i^{d,p} = f_1(\beta_i^{d-1,2p}; \beta_i^{d-1,2p+1}; \lambda_{i+\frac{D}{2}}^{d,p})$ 
17:     $\beta_{i+\frac{D}{2}}^{d,p} = f_2(\beta_i^{d-1,2p}; \lambda_i^{d,p}; \beta_i^{d-1,2p+1})$ 
18:  end for
19:  Return  $\beta_{[0,D]}^{d,p}$ 
20: end function

```

---

For leaf nodes, the processing is easier:  $\beta_i^{0,i} = 0$  if the leaf node corresponds to an information bit and  $\beta_i^{0,i} = +\infty$  if the leaf node is a frozen bit.

Finally, for the root node, few modifications are needed with regard to the description provided in Algorithm 1. Firstly, the  $\lambda^{0,n}$  LLRs are replaced by channel LLRs ( $Y$ ). Secondly, the  $\beta^{0,n}$  LLRs are replaced by the decoder soft outputs. Finally, one should notice that the whole tree can be processed several times in an iterative manner.

### 2.3 Reduced complexity SCAN algorithm

The SCAN algorithm provides soft output information. However, the memory and computation complexities of its original description are higher than the SC decoding algorithm (e.g., all  $\beta$  values have to be stored due to the iterative decoding process which increases the memory cost to  $N \times \log N$ ). To solve these issues, a reduced complexity SCAN decoding algorithm named RC-SCAN was proposed in [25] providing equivalent correction performance. Authors show that some of the computations in SCAN decoding can be saved. For instance, the  $\lambda$  computations for left child nodes are simplified by removing the  $\beta$  term from  $f_1$  computation only for left child node computations. The  $\lambda$  computation expression for left child nodes becomes

$$f_1^{\text{zero}}(a, b) = f_1(a, b, 0) = f(a, b) \quad (4)$$

In addition, they demonstrate that some tree pruning optimizations proposed to reduce SC computation complexity are transposable to SCAN decoding [25].

The RC-SCAN algorithm allows a higher throughput than the regular SCAN algorithm as shown in [25]. Moreover, the decoding performance of RC-SCAN algorithm is slightly better than the regular SCAN when the same amount of decoding iterations is executed. The lower complexity and slightly improved decoding performance of the RC-SCAN compared to the regular SCAN pushed us to select it for our optimized software implementation.

## 3 Speeding up the software SCAN decoder

As a result of programmable architectures progress, high-throughput software implementations of FEC decoders became feasible for applications that were long considered practical only on ASIC devices [16, 17, 21, 28, 29]. Software implementations provide some clear benefits to FEC decoders. At first, their high-throughput implementation enables to rapidly evaluate the decoding performance of a given error correction code/algorithm. Moreover, they are usually a viable solution to the real-time implementation of FEC features on embedded systems. The challenging task in a software implementation consists in mapping efficiently the algorithm parallelism on an existing parallel architecture designed to support different application types. Current microprocessors have inherent built-in parallelism features (SIMD, multi-core, etc) that may be considered and used to efficiently execute concurrent tasks.

Polar codes have algorithmic characteristics such as their memory access regularity which can lead to very high throughput implementations even on embedded processors [4, 21, 30] compared with other FEC families such as LDPC and turbo code at similar FER performance [16, 17]. To date, only SC and SC-List decoding algorithm were optimized in software. In this paper, the performance of the first software SCAN polar decoder on x86 multi-core platform is evaluated. This section describes (i) the targeted device and the available programming models, (ii) the analysis of the parallelism in the SCAN algorithm, and (iii) the different optimization techniques that were applied to the software SCAN decoder to maximize its throughput.

### 3.1 Selected multi-core architecture

x86 processors have multi-core architectures and were initially designed to support general purpose computations. Currently, they have high clock frequencies and they offer

multi-level caches to minimize memory access latency. Intel added complex SIMD instruction sets to its processors. These instructions [31] can execute a single operation type on multiple data sets simultaneously. In the most advantageous cases, an SIMD unit performs up to 32 8-bit parallel computations. SIMD parallelism mainly depends on data format. However, using the SIMD feature is not always easy and cost-free due to the algorithm computation scheduling and/or the data alignment in the memory.

Current general purpose processors contain several processor cores on a single chip. For instance, Intel currently integrates up to 18 processor cores in its Xeon E5 chip. Multiple cores enable to exploit the SPMD (Single Program, Multiple Data) programming model. If properly exploited, SPMD can provide an important speedup. However, the speed does not scale linearly with the number of cores because of the cache coherence issues. Moreover, from a software point of view, synchronization barriers are required, e.g., for data exchange. Consequently, SPMD provides interesting performance improvement when the thread runtime is important enough to hide thread start/join penalties.

To reach high performance, both the memory footprint and the number of instructions have to be reduced. Moreover, the usage rate of SIMD units has to be increased. In the next subsections, we detail the different parallelism levels of the SCAN decoding algorithm. Then, the implementation choices and the applied optimizations are presented.

### 3.2 Parallelism levels in the SCAN algorithm

To achieve high-throughput performance on a programmable device, a software SCAN decoder has to exploit the overall computation parallelism provided by the target. Three parallelism levels were identified in the SCAN decoding algorithm:

**First parallelism level** lies in the  $\mathcal{N}^{d,p}$  nodes (Fig. 1). As shown in Algorithm 1, each node requires a large set of computations to process the  $\lambda$  and  $\beta$  LLRs according to upper and lower node LLRs. The amount of parallelizable  $f_1$  and  $f_2$  computations depends on node level in the graph. For a node  $\mathcal{N}_p^d$ ,  $2^{d-1}$  computations of type  $f_1$  are first performed to obtain the  $\lambda$  LLRs required by the left child node execution. Then,  $2^{d-1}$   $f_2$  computations are required for the right children node. Then,  $2^{d-1}$  computations of type  $f_2$  are executed to generate the  $\lambda$  LLRs that are inputs of right children node. Finally,  $2^{d-2}$  computations of type  $f_1$  and  $2^{d-2}$  computations of type  $f_2$  are required to produce the  $\beta$  LLRs that are returned to mother node ( $\mathcal{N}_{p/2}^{d+1}$ ). At higher graph levels, when  $2^d$  is higher or equal to the SIMD data width, SIMD parallelization is straightforward. It becomes however less efficient at lower graph levels

where the amount of computation to perform is less than the SIMD data width. Therefore, these levels need complex data padding. This parallelization approach without data padding was selected for SC decoding algorithm implementation in [32]. Note that due to the low computation complexity of  $f_1$  and  $f_2$  functions and to the thread start & join overheads, MPMD (Multiple programs, Multiple Data streams) programming model feature is inefficient.

**Second parallelism level** exists between connected nodes in the graph (descending order), e.g.,  $\mathcal{N}_p^d$  and  $\mathcal{N}_{2,p}^{d-1}$  nodes. In the SCAN algorithm,  $f_1$  computations of  $\mathcal{N}_{2,p}^{d-1}$  node can start before the end of the  $f_1$  computations of  $\mathcal{N}_p^d$  node. Considering this, it is possible to save a small amount of memory accesses at  $\mathcal{N}_{d-1}$  level by reusing the calculated LLRs immediately instead of storing them back into the memory. However, this parallelism is mainly limited by the left edge processing. It also increases the algorithm description complexity that is why it is discarded in real software implementation.

**Third parallelism level** is located at the frame level. Two parallelism approaches can be explored. The first one takes advantage of the iterative nature of the algorithm. A pipelined decoder can execute the decoding iteration  $i + 1$  for frame  $q$  and iteration  $i$  for frame  $q + 1$ . However, when an early termination criterion is applied to save useless decoding iterations as shown in [25], the parallelism becomes irregular and thus inefficient. The second approach consists in taking advantage of inter-frame parallelism. Several consecutive frames are simultaneously decoded at once. Indeed, the same computation sequence is executed over the different data sets. This approach is an efficient parallel processing method but it requires data interleaving and deinterleaving steps before and after the decoding process to align the data in memory. It was demonstrated to be the most advantageous way to achieve high-throughput performance for SC decoding algorithm the [21] at the cost of an increased latency.

In this work, two parallelization schemes taking advantage of massively parallel devices but providing different implementation tradeoff are explored. Indeed, they seem to answer different use cases.

A first approach called intra-frame parallelization focuses on the first parallelism level. Computation set in graph nodes is optimized to benefit from SIMD features. The  $P$  processor cores are then used to decode  $P$  distinct frames in parallel. This way should provide low processing latency in applications like turbo-based equalization where a single information frame is received at once [9].

A second approach called inter-frame parallelization focuses on the third parallelism level. It decodes a set

of  $F \times P$  frames in parallel.  $F$  frames are processed in parallel by the SIMD units of one processor core while the  $P$  processor cores are used in an identical way. This way should provide higher processing regularity in systems implementing, e.g., parallel concatenated decoders [1] where multiple information frames are available.

## 4 Algorithm architecture matching

In the following subsections, we present different optimization techniques applied to both SC and RC-SCAN decoders using intra- and inter-frame parallelism to achieve high throughputs for SCAN decoders.

### 4.1 Tree cut

In [27], it was shown that some of the computations in the SC decoding are not necessary. Depending on the frozen bit positions in the code, some parts of the graph become useless and the associated computations are either simplified or simply discarded. In [25], an identical demonstration was made for the SCAN algorithm though single parity check and repetition subcodes were not simplified. Let us consider a node  $\mathcal{N}_p^d$  corresponding to the decoding of a subcode of size  $D = 2^d$ . Assuming that the considered subcode has a code rate 0, the node  $\mathcal{N}_p^d$  always returns  $\beta_{[0,D]}^{d,p} = +\infty$  regardless of the provided  $\lambda_{[0,D]}^{d,p}$ . Indeed, all bits in this subcodes are frozen. On the contrary, when none of the bits is frozen (rate-1 subcode), the  $\mathcal{N}_p^d$  always returns  $\beta_{[0,D]}^{d,p} = 0$ . Removing these parts from the graph significantly reduces the amount of computations in the SCAN decoding algorithm. Each node in the tree is characterized by its code rate which can be  $R = 0$ ,  $R = 1$  or  $0 < R < 1$ . This information is stored in a static vector for each node as it is constant for a given polar code. When a node  $\mathcal{N}_p^d$  is called, it retrieves its associated code rate then it performs the processing accordingly as shown in Algorithm 2.

---

#### Algorithm 2 Simplified ProcessNode function.

---

```

1: LLRs and child node are computed only if necessary
2: if  $R(\mathcal{N}_j) == 0$  then
3:    $\beta_{[1..l]}^k = +\infty$ 
4: else if  $R(\mathcal{N}_j) == 1$  then
5:    $\beta_{[1..l]}^k = 0$ 
6: else
7:   for all  $i \in [0, J/2[$  do
8:      $\lambda_i^k = f_x(\dots; \dots; \dots)$ 
9:   end for
10:   $\beta_{[1..l]}^k = \text{ProcessNode}(\lambda^k, \frac{J}{2})$ 
11: end if

```

---

### 4.2 $\mathcal{N}_p^2$ node unrolling

The singular tree-based representation of the SCAN decoding algorithm motivates decoder implementations from a recursive-based software description. It actually simplifies the algorithm's description but generates many recursive function calls which are time-consuming. Indeed, a function call needs register saving on the memory stack. At a higher level of the graph, where a large set of  $f_1$  and  $f_2$  computations are performed, this runtime overhead is negligible. However, at lower graph levels, such as for node  $\mathcal{N}_p^d$  with  $d < 3$ , the additional runtime cost is non-negligible.

In order to avoid multiple recursive calls, the description of Algorithm 1 was completely unrolled for the level  $d = 2$  nodes. Optimizing these lower level nodes, execution reduces the overall decoding time. Subsequently, the number of clock cycles spent in the control of function calls is reduced which decreases the amount of the executed instructions. Moreover, some control structures can be removed, e.g., loop structures required to process  $\lambda$  and  $\beta$  values ( $f_1$  and  $f_2$  computations) inside nodes.

Unlike the SC optimization proposed in [32], program unrolling is only applied to  $\mathcal{N}_p^2$  nodes. In fact, this choice has three main advantages compared to [32]:

1. It limits the increase in program size and thus reduces the number of cache misses at runtime.
2. A single program description can process any code length and code rate without requiring neither regeneration nor recompilation of the source code.
3. It insures the decoder's scalability for long-length codes as stated in [32] and [33].

### 4.3 Memory mapping

The amount of memory that is necessary during the algorithm execution and its access types have a large impact on the decoder throughput. This is mainly due to the increasing cache misses. Memory access cost at runtime mainly depends on (a) the information location in the cache memories and (b) the access patterns regularity. Fortunately, the SCAN algorithm processes the data set linearly without random memory access, unlike LDPC codes [16] or turbo codes [17]. This characteristic enables efficient memory bandwidth usage. However, to reach the highest performance, it is necessary to minimize the decoder's memory footprint so as to limit the cache deficiency probability.

The first descriptions of the SCAN-based algorithm [7, 8] and its first hardware implementation [24] consumed a large amount of memory to store the  $\lambda$  and the  $\beta$  values. During the decoding of one frame, the SCAN decoder has to store the following:  $N$  channel values;  $N \log_2 N$  LLRs ( $\lambda_{[0,D]}^{d,p}$ )



and  $N \log_2 N$  partial sums ( $\beta_{[0,D_L]}^{d,p}$ ). The memory footprint of the decoder is

$$\mu_A = 2N \log_2 N + N \quad (\text{Elements}) \quad (5)$$

For fixed-point data format used in hardware implementations [24], the memory complexity is also a limiting factor when  $N$  increases.

In [25], the proposed algorithmic transformations reduce the memory requirements for the  $\beta$  values while for the hardware architecture, the amount of  $\beta$  values to store is reduced from  $N(\log_2 N)$  to  $3 \times N$ . Consequently, the memory footprint can be further reduced:

$$\mu_C = 5 \times N \quad (\text{Elements}) \quad (6)$$

For software implementation, the technique proposed in [34] aiming at reducing the amount of  $\lambda$  LLRs to  $N$  can also be applied to  $\beta$  LLR values. Indeed, in the RC-SCAN algorithm, the  $\beta$  LLRs are not alive during the same decoding time period. The memory mapping proposed in [21] for  $\lambda$  values can then be reused also for  $\beta$  values. The memory footprint of  $\beta$  values also decreases to  $N$  elements. Finally, the memory footprint of the proposed RC-SCAN decoder is reduced to

$$\mu_D = 3 \times N \quad (\text{Elements}) \quad (7)$$

This memory mapping is  $1.6\times$  less memory consuming than the one used in hardware RC-SCAN decoder implementation.

#### 4.4 LLR data packing

In the previous section, we mentioned that LLR elements and computations are performed using the 32-bit floating-point format. It was demonstrated in [25] that 7-bit fixed-point representation is sufficient for  $\lambda$  and  $\beta$  LLRs to achieve decoding performance close to a floating-point format representation even at FER= $10^{-7}$  ( $< 0.1$  dB). An 8-bit fixed-point representation is selected. Indeed, it is a natural width of the processor data path. Moreover, it reduces the correction performance penalty (close to 0 dB) as shown in the experimental section (Figs. 4 and 5). The decoding performance of several other polar codes were simulated but not reported in the figure. In all of the simulated codes, 8-bit quantization was sufficient to get very close to floating point decoding performance.

In the fixed-point description of the RC-SCAN decoder, the overall LLR elements ( $Y$ ,  $\lambda$ , and  $\beta$ ) in the decoding algorithm are converted to 8-bit signed integers. This data format transformation impacts on the memory footprint of the software SCAN decoder, reducing it by a factor of four.

This memory saving helps to reduce the runtime cache deficiency probability and thus to improve the decoding

throughput, especially for long frames. In the following subsection, it will be shown that packing data into 8-bit integers also increases the SIMD usage rate (up to 32 computations in parallel can be performed on 8-bit fixed-point values whereas 8 floating point computations can only be performed simultaneously).

#### 4.5 SIMD parallelization

Today's multi-core processors include SIMD technology [35, 36]. The targeted Core-i7 processor supports up to AVX2 instruction-set. It can perform 32 8-bit computations simultaneously.

In the current work, two orthogonal approaches are used to take advantage of computation parallelism:

The first approach explores the intra-frame parallelism. One frame is decoded at a time and loop computations in nodes are vectorized. Speedup can reach up to  $32\times$  with the 8-bit fixed-point format when nodes have more than  $2\times$  SIMD  $\lambda$  and  $\beta$  LLRs. For lower level nodes in the graph, the SIMD units are not fully assigned and the acceleration factors are lower.

The second approach takes advantage of the inter-frame parallelism.  $F$  independent frames are decoded in parallel. Consequently, with regard to a non-vectorized implementation, the speedup is constant to  $32\times$  for a 8-bit fixed-point version.

Inter-frame approach is more attractive, yet it suffers from massive misaligned memory accesses. Indeed, to process the same  $X$  elements for the different frames in a SIMD way at the same time, non-contiguous read/write accesses are necessary. To solve this issue, a data interleaving process [21] has to be performed before the decoding can start. Similarly, a deinterleaving step has to be performed after the decoding process. In other words, channel information  $X$  coming from the  $F$  frames are first transformed to obtain an aligned memory data structure.

In order to maximize the efficiency of the proposed SCAN decoder implementation, some parts of the source code use SIMD primitives. Figure 3 shows a code snippet of the  $f_1$  function implementation optimized for AVX2 instruction set. Figure 2 shows a naive non-optimized implementation of the  $f_1$  function together with a SIMD-optimized version both in floating-point format. An analysis of the characteristics of the two compiled functions was done with Intel Architecture Code Analyzer (IACA). On an Intel Core-i7 processor, the naive floating-point  $f_1$  function uses 10 instructions and takes 18 clock cycles to execute. In fact, the optimized function is composed of the same number of instructions but the processing latency is only 16 clock cycles. The performance improvement comes from the fact that the SIMD  $f_1$  function processes 8 LLR values in 16 cycles. The computation time per LLR is thus 2 cycles

**Fig. 2** Initial and AVX2-optimized of  $f_1$  function source codes (floating-point data format)

```
float f1(float a, float b, float c)
{
    const float d = b + c;
    const float mA = std::fabs(a);
    const float mD = std::fabs(d);
    const float sA = std::fabs(a);
    const float sD = std::fabs(d);
    const float mR = std::fmin(mA, mD);
    return (sA == sD) ? mR : -mR;
}

__m256 f1(__m256 a, __m256 b, __m256 c)
{
    const __m256 d = _mm256_add_ps(b, c);
    const __m256 mask = _mm256_set1_epi32(0x80000000);
    const __m256 MASK = _mm256_castsi256_ps( mask );
    const __m256 mA = _mm256_andnot_ps(MASK, a);
    const __m256 mD = _mm256_andnot_ps(MASK, b);
    const __m256 mR = _mm256_min_ps(mA, mD);
    const __m256 sign = _mm256_xor_ps(a, d);
    const __m256 sR = _mm256_and_ps(sign, sM);
    return _mm256_xor_ps(mR, sR);
}
```

while the naive  $f_1$  function description processes only one LLR value per execution.

The fixed-point version of the  $f_1$  function provided in Fig. 3 is more complex. Its higher description complexity comes from the processing of the  $+\infty$  value that is not defined in 8-bit fixed-point format. To solve this special case, we have taken advantage of the symmetry of the signed 8-bit fixed-point format and used the  $-128$  value to encode  $+\infty$ . Runtime tests are performed to ensure that, e.g.,  $-128 + 127 = -128$ . The latency of the fixed-point version of the  $f_1$  function which generates 32 LLR values simultaneously is estimated to 21 clock cycles by

the IACA tool. It is  $1.3\times$  slower than the floating point implementation but it processes  $4\times$  more LLRs. Compared with  $f$  and  $g$  functions that are the edge processing functions in the SC decoding algorithm [21], the execution time of the  $f_1$  and  $f_2$  functions are about twice slower.

## 5 Experimental results

The software implementations of the SCAN decoder are described in standard C++11 language. The genericity of our C++ source code enables a single compiled source

**Fig. 3** AVX2-optimized of  $f_1$  function source code (fixed-point data format)

```
__m256i f1(__m256i a, __m256i b, __m256i c)
{
    // loading saturation and +inf constants
    __m256i inf = _mm256_set1_epi8(-128);
    __m256i sat = _mm256_set1_epi8(-127);
    // looking for +inf value
    __m256i test_1 = _mm256_cmpeq_epi8(b, inf);
    __m256i test_2 = _mm256_cmpeq_epi8(c, inf);
    // Add and saturate
    __m256i add = AVX_32S_ADD( b, c);
    __m256i res = AVX_32S_MAX(add, sat);
    // right result selection (+inf support)
    __m256i reS = _mm256_blendv_epi8(res, inf, test_1);
    __m256i d = _mm256_blendv_epi8(reS, inf, test_2);
    // looking for +inf value
    __m256i test_1 = _mm256_cmpeq_epi8(a, inf);
    __m256i test_2 = _mm256_cmpeq_epi8(d, inf);
    // performing (sA xor sB) * min( abs(a), abs(d) )
    __m256i mA = _mm256_abs_epi8(a);
    __m256i mB = _mm256_abs_epi8(d);
    __m256i mR = _mm256_min_epi8(mA, mB);
    __m256i sS = _mm256_xor_si256(a, d);
    __m256i sR = _mm256_or_si256( sS, mask);
    __m256i Re = _mm256_sign_epi8(mR, sR);
    // right result selection (+inf support)
    __m256i RE = _mm256_blendv_epi8( Re, d, test_1);
    return _mm256_blendv_epi8(RE, a, test_2);
}
```

code to decode any polar code (different frame lengths and code rates) without neither regeneration nor recompilation. The source codes were compiled with INTEL C++ compiler 16 toolchain. The following compilation flags were added: `-march=native -fast -fopenmp -opt-prefetch -unroll-aggressive`.

The evaluation platform used for measuring the characteristics of the SCAN decoders is composed of an INTEL Haswell Core-i7 4960HQ CPU. The turbo-boost option was switched on. Hence, the operating clock frequency of the processor reaches 3.6 GHz when a single processor core is used and 3.4 GHz when the four cores are switched on.

A complete evaluation of the proposed parallel processing SCAN decoders is carried out in the following subsections.

### 5.1 Error correction performance

Before analyzing the optimized decoder throughput, the error correction performance was investigated in order to verify the correctness of the RC-SCAN decoder implementations. The decoding performance is actually equivalent to the one reported in [24]. Figure 4 provides the frame error rate (FER) for the  $2^{15}$  Polar code with floating-point and fixed-point decoders. SCAN-x refers to the decoding performance of a SCAN decoder with x iterations. Random codewords and AWGN channel are considered for simulation. The frozen bits are selected for  $\sigma = 0.419$ . Figure 5 provides the same curves for a  $2^{11}$  polar code. In both cases, the fixed-point format reaches decoding performance very close to floating-point decoders.

The selected fixed-point format representation enables to achieve similar performance to a floating point representation

as demonstrated by the FER performance shown in Figs. 4 and 5.

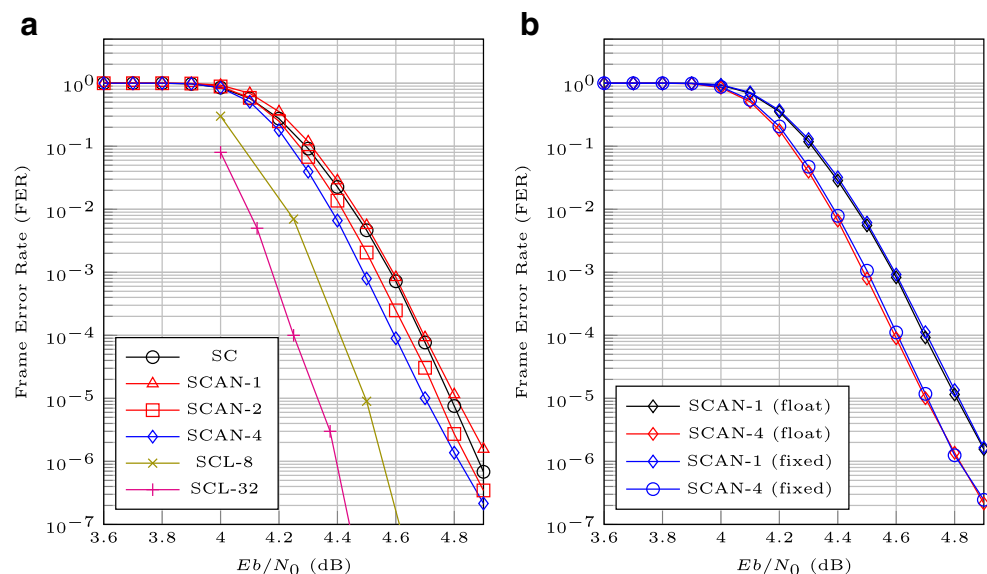
### 5.2 Throughput performance

Two different variants of the SCAN decoders are compared by considering the intra- and inter-frame parallelizations in fixed-point format. The time required to decode  $F$  frames includes (i) the writing of the  $F$  frames in the processor memory, (ii) the writing of the  $F$  estimated codewords, (iii) the decoding of  $F$  frames with the SIMD decoder, and (iv) the ordering ( $\Pi$ )/reordering ( $\Pi^{-1}$ ) functions (only for inter-frame implementations). The intra-frame decoder processes only  $F = 1$  frame at the time. For the inter-frame decoders, the processor core processes 32 frames in parallel. For the experimentation, the air throughput called  $\Gamma$  is estimated thanks to the Chrono API from Boost library.

The error correction performance increases along with the code length  $N$  and it is likewise for both computation complexity and memory footprint. Hence, we found relevant to evaluate the impact of the code length on the decoder throughput. The throughput was measured for one decoding iteration for each decoder (intra-frame and inter-frame). The results are reported in Fig. 6 for two different code rates.

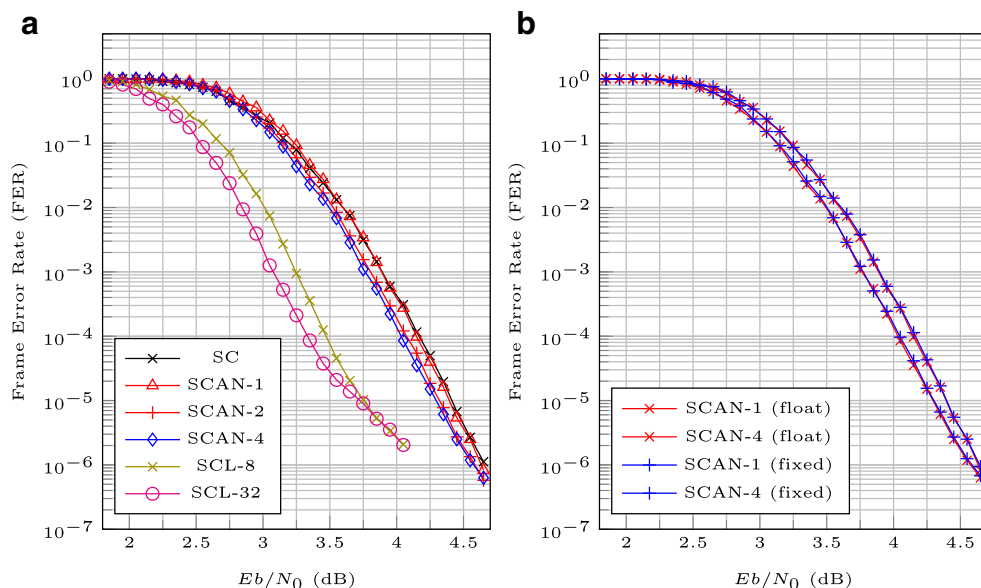
These results demonstrate that the proposed optimized software SCAN decoders reach high throughputs. For a code rate  $1/2$ , the throughputs of the intra-frame decoder vary from 387 Mbps ( $n = 2^{20}$ ) to 529 Mbps ( $N = 2^{12}$ ) while for the inter-frame decoder throughputs range from 324 to 1004 Mbps. For a code rate  $9/10$ , the air throughputs of the two decoders are higher than their rate- $1/2$  counterpart. This is also the case for floating-point

**Fig. 4** Decoding performance of the floating-point SCAN decoders for **a** (32768, 29504) Polar code using floating-point data format and **b** (32768, 29504) Polar code using 6b quantization format for channel and 8b quantization format for LLRs





**Fig. 5** Decoding performance of the floating-point SCAN decoders for **a** (2048, 1723) Polar code using floating-point data format and **b** (2048, 1723) Polar code using 6b quantization format for channel and 8b quantization format for LLRs



versions. Regarding the code rate 9/10 configuration, the throughputs of the intra-frame decoder vary from 523 Mbps ( $n = 2^{20}$ ) to 838 Mbps ( $n = 2^{12}$ ) while for the inter-frame decoder throughput range from 337 Mbps ( $n = 2^{20}$ ) to 1421 Mbps ( $n = 2^{10}$ ) Mbps.

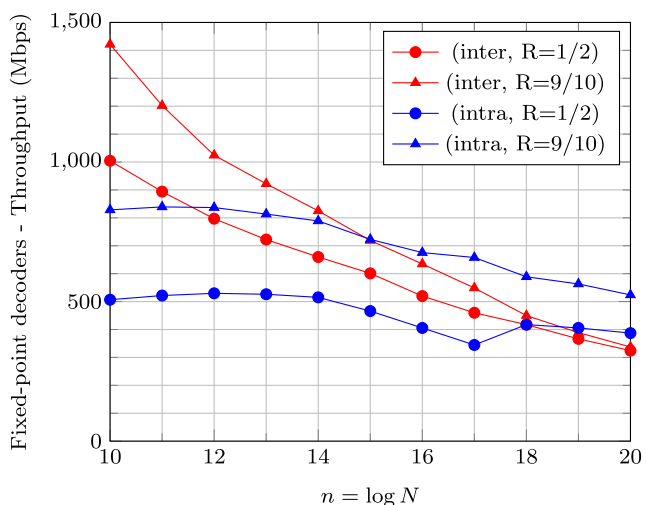
The intra-frame parallelization approach does not always provide the best performance. For a code rate 1/2, the inter-frame decoder is more efficient when  $n \leq 18$ , while for a code rate 9/10, it is faster when  $n \leq 15$ . The lower throughput of the intra-frame decoders is due to the data padding required for the lower level nodes  $d < 6$ . It drastically reduces the efficiency of the intra-frame decoder because the SIMD units have to perform a large set of useless computations and data padding. It can be noticed

that the code length and code rate have also an impact on the throughput performance. Besides, the constant SIMD parallelization leads to a high speed for the inter-frame decoder which are less important when  $N$  increases. Indeed, a constant parallelism is achieved at the cost of increased memory requirements. When  $N$  increases, it leads to high memory footprint which gives rise to a large set of cache misses that counterbalances the processing efficiency.

The SCAN decoding algorithm is based on an iterative process. As shown previously, more than one decoding iteration can be executed on the data set to improve the decoding performance. The decoder’s throughput decreases along with the number of iterations. Since the amount of computation is the same at each iteration, the throughput decreases linearly with the number of iterations. This was actually verified experimentally for a large set of code lengths and code rates. To facilitate the experimentations understanding, all the throughput evaluations are provided for one decoding iteration. When several iterations are performed, the decoder throughput can be safely approximated by the throughput of a single iteration decoder divided by the number of iterations.

An important parameter in most of the digital communication systems is the decoding latency ( $L$ ), which is the time required by the decoder to obtain a decoded data. It is reported in Fig. 6 by using Eq. 8 with  $N$  the frame length and  $F$  the SIMD computation parallelism level ( $F = 1$  for intra-frame decoders).

$$L = \frac{N \times F}{\text{throughput}} \tag{8}$$



**Fig. 6** Measured coded throughputs on a single core of the Core-i7 processor with fixed-point data formats and AVX2 instruction set

The processing latency depends on the considered parallelism (intra- or inter-). Indeed, the inter-frame

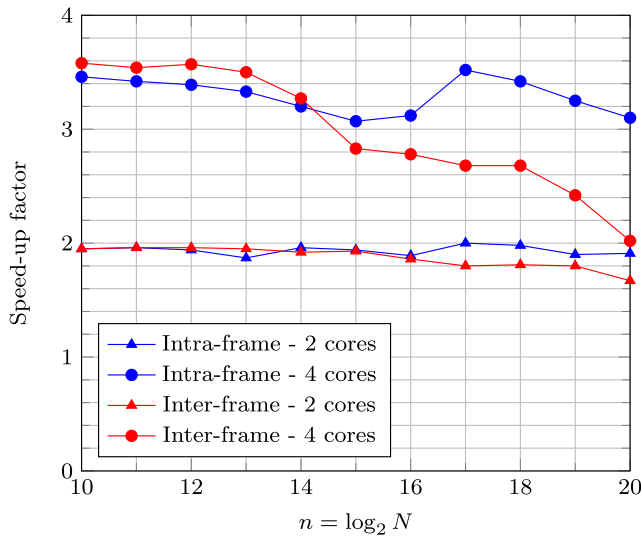


Fig. 7 Speedup factor achieved using multiple processor cores

decoder processes  $F$  frames. This requires that the system buffers the data which generates latency penalty before beginning of the decoding process. Consequently, inter-frame decoders processing latency is at least  $F$  times higher than intra-frame decoders when an equivalent throughput is considered. For instance, for the fixed-point decoders with  $N = 2^{15}$  and a code rate  $9/10$ , the intra-frame decoder latency is about  $50\mu s$  whereas the latency of the inter-frame decoder is  $1574\mu s$  ( $32\times$  higher).

### 5.3 Impact of the multi-threading

Current embedded CPUs include multiple processing cores. The performance of computation intensive algorithms running on a multi-core processor depends on the number of available cores. The speed is also highly impacted by the processor’s memory architecture. It generates more cache

deficiencies and limits the speedup offered by the multi-core processing.

OpenMP directives [37] were included in the decoder descriptions to perform multi-core execution. One thread is used per available core. Assuming  $P$  cores are available, the  $P$  threads are first created. Then, each core processes  $F = 32$  frames in parallel for inter-frame decoders and  $F = 1$  for intra-frame decoders. Finally, threads are synchronized before a new set of  $F$  frames is launched. Figure 7 shows the measured acceleration of the fixed-point decoders for one iteration,  $P = \{2, 4\}$  cores and a code rate  $R = 1/2$ .

A maximum speedup of  $\sim 3.6$  is reached for  $1024 \leq N \leq 4096$  and  $M = 4$  for inter-frame decoders. The improvement is slightly lower ( $\sim 3.4$ ) for the intra-frame decoders. The maximum speedup decreases with  $N$  for the three aforementioned reasons. Note that for the inter-frame decoder, the speedup significantly decreases when  $N > 15$ . This due to its memory footprint that is  $4\times$  higher than in a single-core scenario. The intra-frame decoder provides a more constant speedup factor as  $N$  increases.

Despite the decreasing speedup, the intra-frame decoders reach throughputs of 3600 Mbps when  $N = 2^{10}$  and 654 Mbps with  $N = 2^{20}$ , if four processor cores are switched on. They provide throughputs of 1753 and 1199 Mbps in similar conditions. These results show that the two decoder implementations on a multi-core processor can reach throughputs higher than 1 Gbps.

Compared with hardware solutions [24, 25], the proposed software implementations offers a higher flexibility at equal throughput performance. Indeed, compared with an FPGA prototype [24], where the throughput reaches 17 Mbps for a (1024, 512) code, the software implementation delivers more than 1 Gbps on a single Core-i7 processor core. In comparison with ASIC circuit result which reports a throughput of 2208 Mbps for a  $2^{15}$  code [25], the performance level achieved by the proposed software solution

Table 1 Throughput and latency comparison of different software implementations of polar decoding algorithms for (2048, 1723) code

Related work decoders						Proposed SCAN decoders				
Algo.	Par.	Targ.	List	$\mu s$	Mbps	Par.	Targ.	Iter.	$\mu s$	Mbps
SC [21]	Inter	$C_1$	–	30	2172	Inter	$C_1$	1	54	1202
								2	607	107
								4	306	211
SC [30]	Intra	$C_2$	–	1	1293	Intra	$C_1$	1	2	839
								2	4	421
								4	8	211
SCL [39]	Intra	$C_3$	8	490	2.09	Intra	$C_1$	1	2	839
SCL [40]		$C_4$	8	392	2.61			2	4	421
SCL [40]		$C_4$	32	1531	0.67			4	8	211

$C_1$  Core-i7 4960HQ,  $C_2$  Core-i7 4770S,  $C_3$  Core-i7 2600,  $C_4$  Core-i7 4790k

when four processor cores are switched on is slightly higher (about 2.4 Gbps).

#### 5.4 State-of-the-art software polar decoders

Among all existing polar decoding algorithms, one should keep in mind that only the BP decoder actually delivers a soft output. However, in order to have an idea of the speed of the proposed SCAN decoder in comparison with other software polar decoders, Table 1 shows the throughput and latency of SC and SCL optimized software polar decoders.

From Table 1, one can notice that the SC decoder has the lowest complexity and then provides the highest throughput. The SCAN is always slower than SC software decoders but can have slightly improved decoding performance if the number of iterations is higher than 1.

The SCL decoding clearly outperforms the SCAN in terms of decoding performance at the cost of an increased complexity. The throughputs of the state-of-the-art SCL software decoders is at least 100 times lower than SC or SCAN software decoders.

The only fully relevant algorithm to compare with is the BP decoding implementation [38] because it does provide soft outputs. However, this algorithm hardly reaches the decoding performance of SC and SCAN decoders while requiring a large number of iterations (at least 20). In terms of throughput, this large number of iterations limits the throughput to a few megabits per second. In [38], throughputs of 6.04 and 2.11 Mbps are reported for (1024, 512) and (2048, 1024) polar codes, respectively. The implementation platform was a NVIDIA GTX 560 Ti GPU device. For the same polar codes, the proposed inter-frame parallelized decoders deliver higher decoding throughputs: 1004 and 796 Mbps whereas intra-frame parallelized decoders reach 506 and 529 Mbps. It shows the interest of the proposed x86 implementation for soft output polar code decoding.

## 6 Conclusion

In this paper, two optimized software implementations of the SCAN decoding for Polar codes are detailed. The intra-frame and inter-frame parallelization strategies are exploited in order to efficiently map the decoding process to a multi-core processor. In addition to SIMD parallelization, the decoder description was optimized at algorithmic and source code levels. The enhancements lead to improved results which reach multi-Gbps for the software decoders. Reported throughputs are higher than the ones reported for ASIC and FPGA implementations at the cost an increased power consumption. However, these performance

levels obtained on a laptop CPU target associated with software flexibility should meet with current and future communication networks or Cloud-RAN systems where most of the processing function would be implemented in software.

## References

- Zhang Q, Liu A, Zhang Y, Liang X (2016) Practical design and decoding of parallel concatenated structure for systematic polar codes. *IEEE Trans Commun* 64(2):456–466
- Arikan E (2009) Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Trans Inf Theory* 55(7):3051–3073
- Tal I, Vardy A (2015) List decoding of polar codes. *IEEE Trans Inf Theory* 61(5):2213–2226
- Sarkis G, Giard P, Vardy A, Thibeault C, Gross WJ (2015) Unrolled polar decoders, part ii: Fast list decoders. *IEEE J Sel Areas Commun - Special Issue on Recent Advances In Capacity Approaching Codes* (submitted)
- Arikan E (2008) A performance comparison of polar codes and reed-muller codes. *IEEE Commun Lett* 12(6):447–449
- Guo J, Qin M, Fabregas G, Siegel PH (2014) Enhanced belief propagation decoding of polar codes through concatenation. In: *Proceedings of the IEEE international symposium on information theory (ISIT)*, pp 2987–2991
- Fayyaz UU, Barry JR (2013) A low-complexity soft-output decoder for polar codes. In: *Proceedings of the IEEE global communications conference (GLOBECOM)*, pp 2692–2697
- Fayyaz UU, Barry JR (2013) Polar codes for partial response channels. In: *Proceedings of the IEEE international conference on communications (ICC)*, pp 4337–4341
- Fayyaz UU (2014) Polar code design and decoding for magnetic recording, Georgia Institute of Technology, PhD thesis
- Zhao S-M, Xu SP, Xing C (2015) Concatenated polar-coded multilevel modulation. In: *Proceedings of the 10th international conference on communications and networking in China (ChinaCom)*, pp 153–157
- Wu D, Liu A, Zhang Q, Zhang Y (2015) Concatenated polar codes based on selective polarization. In: *Proceedings of the 12th international computer conference on wavelet active media technology and information processing (ICCWAMTIP)*, pp 436–442
- Sha J, Liu J, Lin J, Wang Z (2016) A stage-combined belief propagation decoder for polar codes. *J Signal Process Syst*, 1–8
- Wang Y-CHY, Narayanan KR (2016) Interleaved concatenations of polar codes with bch and convolutional codes. *IEEE J Selected Areas Commun* 34:267–277
- Li G, Jianjun M, Jiao X, Guo J, Liu X (2017) Enhanced belief propagation decoding of polar codes by adapting the parity-check matrix. *EURASIP J Wirel Commun Netw* 40:9
- Zhang Q, Liu A, Pan X, Zhang Y (2017) Symbol-based belief propagation decoder for multilevel polar coded modulation. *IEEE Commun Lett* 21:24–27
- Le Gal B, Jego C (2015) High-throughput multi-core LDPC decoders based on x86 processor. *IEEE Trans Parallel Distrib Syst (TPDS)* 27(5):1373–1386
- Wu M, Sun Y, Wang G, Cavallaro JR (2011) Implementation of a high throughput 3GPP turbo decoder on GPU. *J Signal Process Sys Springer*. 65(171)
- Grayver E (2013) *Implementing software defined radio*. Springer, New York

19. Checko A, Christiansen HL, Yan Y, Scolari L, Kardaras G, Berger MS, Dittmann L (2015) Cloud RAN for mobile networks—a technology overview. *IEEE Commun Surv Tutor* 17(1):405–426
20. Giard P, Sarkis G, Thibeault C, Gross WJ (2014) A fast software polar decoder. In: *Proceedings of the IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pp 7555–7559
21. Le Gal B, Leroux C, Jego C (2015) Multi-Gb/s software decoding of polar codes. *IEEE Trans Signal Process* 63(2):349–359
22. Le Gal B, Leroux C, Jego C (2014) Software polar decoder on an embedded processor. In: *Proceedings of the IEEE workshop on signal processing systems (SiPS)*, pp 1–6
23. Sarkis G, Giard P, Vardy A, Thibeault C, Gross W (2014) Increasing the speed of polar list decoders. In: *Proceedings of the IEEE workshop on signal processing systems (SiPS)*, pp 1–6
24. Berhault G, Leroux C, Jego C, Dallet D (2015) Hardware implementation of a soft cancellation decoder for polar codes. In: *Proceedings of the IEEE conference on design & architectures for signal & image processing*, pp 1–8
25. Lin J, Xiong C, Yan Z (2015) Reduced complexity belief propagation decoders for polar codes. In: *Proceedings of the IEEE workshop on signal processing systems (SiPS)*, pp 1–6
26. Tal I, Vardy A (2013) How to construct polar codes. *IEEE Trans Inf Theory* 59(10):6562–6582
27. Alamdar-Yazdi A, Kschischang FR (2011) A simplified successive-cancellation decoder for polar codes. *IEEE Commun Lett* 15(12):1378–1380
28. Andrade J, Falcao G, Silva V (2014) Optimized fast Walsh-Hadamard transform on GPUs for non-binary LDPC decoding. *Parallel Comput Elsevier* 40(9):449–453
29. Hou Y, Liu R, Peng H, Zhao L (2015) High throughput pipeline decoder for LDPC convolutional codes on GPU. *IEEE Commun Lett* 19(12):2066–2069
30. Giard P, Sarkis G, Leroux C, Thibeault C, Gross WJ (2016) Low-latency software polar decoders. *Journal of Signal Processing Systems*
31. Deilmann M (2012) A guide to vectorization with intel C++ compilers. Intel Corporation
32. Sarkis G, Giard P, Thibeault C, Gross WJ (2014) Autogenerating software polar decoders. In: *Proceedings of the IEEE Global conference on signal and information processing (GlobalSIP)*, pp 6–10
33. Sarkis G, Tal I, Giard P, Vardy A, Thibeault C, Gross WJ (2016) Flexible and low-complexity encoding and decoding of systematic polar codes. *IEEE Trans Commun* 64(7):2732–2745
34. Leroux C, Tal I, Vardy A, Gross WJ (2011) Hardware architectures for successive cancellation decoding of polar codes. In: *Proceedings of the IEEE International conference on acoustics, speech and signal processing (ICASSP)*, pp 1665–1668
35. Bjerke H (2008) SIMD tutorial. CERN openlab
36. Intel corporation (2014) Intel 64 and IA-32 architectures optimization reference manual order number: 248966-029 edition
37. Chapman B, Jost G, Van Der Pas R (2008) *Using OpenMP: portable shared memory parallel programming*. The MIT Press
38. Reddy BK, Nitin C (2012) GPU implementation of belief propagation decoder for polar codes. In: *Proceedings of the forty sixth Asilomar conference on signals, systems and computers (ASILOMAR)*, pp 1272–1276
39. Sarkis G, Giard P, Vardy A, Thibeault C, Gross WJ (2016) Fast list decoders for polar codes. *IEEE J Selected Areas Commun* 34:318–328
40. Shen Y, Zhang C, Yang J, Zhang S, You X (2016) Low-latency software successive cancellation list polar decoder using stage-located copy. In: *Proceedings of the IEEE International conference on digital signal processing (DSP)*, pp 84–88