Reduced-Complexity Low-Latency Logarithmic Successive Cancellation Stack Polar Decoding for 5G New Radio and Its Software Implementation

Luping Xiang, Shida Zhong, Robert G. Maunder, *Senior Member, IEEE* and Lajos Hanzo, *Fellow, IEEE*

***Abstract*—An efficient Fast logarithmic successive cancellation stack (Log-SCS) polar decoding algorithm is proposed along with its software implementation using single instruction multiple data (SIMD) style processing. Quantitatively, we reduce the decoding complexity by 50% on average, while simultaneously attaining a decoding latency that is only 21% of that of the state-of-the-art Fast successive cancellation list (SCL) polar decoder's software implementation. This is achieved without any loss of error correction performance by applying simplified path-metric (PM) computations for the rate-0, rate-1 and repetition sub-graphs of the proposed Fast Log-SCS decoder. Furthermore, a software implementation of the 32-bit fixed-point Fast Log-SCS polar decoder is conceived for x86 processors, which maintains the same block error ratio (BLER) as the floating-point Log-SCS polar decoder. Additionally, our software implementation is accelerated using SIMD instructions by relying on 512-bit Advanced Vector Extensions (AVX-512) and recursive template meta-programming for the first time, achieving a parallelism of 16, which makes it eminently suitable for the low-latency requirements of software-defined radio systems.**

***Index Terms*—Polar codes, Log-SCS decoder, fixed-point implementation, SIMD.**

I. Introduction

Polar codes have been selected for protecting the control channels of the 3rd Generation Partnership Project's (3GPP) new radio (NR) standard conceived for next generation communications [1–4]. They are capable of achieving the Shannon capacity, when using the successive cancellation (SC) decoding algorithm in the case of an idealised infinite code block length in binary memoryless channels [5]. However, the serial nature of the SC decoding algorithm imposes data dependencies, resulting in a high decoding latency and low throughput. Therefore, sophisticated techniques have been proposed in [6–11], for circumventing this disadvantage relative to other channel codes, such as low-density parity-check (LDPC) and turbo codes. For example, the authors of [6] proposed a simplified SC (SSC) polar decoder that considers the group decoding of successive frozen or information bits, referred to as rate-0 and rate-1 nodes. Following this, the group decoding of repetition nodes and single-parity-check nodes was proposed in [8], further simplifying the decoding operations and improving both the throughput and latency. This design was then employed in [12] for the implementation of the SC decoder in a software-defined radio (SDR) system employing x86 processors for supporting 256-bit Advanced Vector Extensions (AVX2) with the aid of single instruction multiple data (SIMD) parallel processing. Since the SIMD instruction set has been inherently embedded in x86 processors, a higher grade of parallelism can be achieved in the AVX2 mode, resulting in a significantly higher throughput.

However, the error correction performance of the SC polar decoder remains limited for realistic finite block lengths, especially in fading wireless vehicular channels. Therefore, more complex decoding algorithms have been proposed for further improving the block error ratio (BLER) in these scenarios [13]. In contrast to SC decoding, which only selects each successive bit value having the highest likelihood in a decoded bit sequence, the successive cancellation list (SCL) decoding algorithm of [13, 14] enhances the decoding reliability by considering a list of L candidate bit sequences at a time. The practical implementation of the SCL algorithm typically represents the likelihoods of candidate bit values using logarithmic likelihood ratios (LLRs), hence supporting an efficient and numerically stable implementation [15]. Further improvements that reduce the decoding latency and increase the throughput of the SCL decoder are proposed in [16–19], which belong to the family of the Fast simplified SCL (Fast-SSCL) decoders. The real-time software implementation of the SCL decoder was demonstrated in [17, 20]. Additionally, by appending a cyclic redundancy check (CRC), the BLER performance of the SCL may be enhanced [14]. More specifically, the CRC-aided SCL decoder discards any decoding candidate that does not satisfy the CRC, even if they would appear to offer the globally most-likely bit values.

By contrast, the successive cancellation stack (SCS) decoding of [21] uses a depth-first search, rather than the breath-first search of the SCL decoder. The SCS algorithm uses a stack of up to S decoding candidates in order to perform a directed search for the sequence of bit values having the globally highest likelihoods. The SCS algorithm achieves a lower decoding complexity than the SCL, and approaches that of SC when used in channels having high signal-to-noise ratios (SNRs). The logarithmic SCS (Log-SCS) decoder investigated in [22–24] operates on the basis of LLRs and benefits from an improved error correction capability and other practical

L. Xiang, R. G. Maunder and L. Hanzo are with Electronics and Computer Science, University of Southampton, SO17 1BJ, United Kingdom, e-mail: {lx1g15,rm,lh}@soton.ac.uk.

S. Zhong is with College of Electronics and Information Engineering, Shenzhen University, Shenzhen 518060, China, e-mail: shida.zhong@szu.edu.cn

L. Hanzo would like to gratefully acknowledge the financial support of the Engineering and Physical Sciences Research Council projects EP/Noo4558/1, EP/PO34284/1, COALESCE, of the Royal Society's Global Challenges Research Fund Grant as well as of the European Research Council's Advanced Fellow Grant QuantCom.

TABLE I: Contrasting the contributions of this work to the literature.

Contributions	**This work**	[16]	[17]	[18]	[19]	[21]	[22]	[23]
SCS decoder	✓					✓	✓	✓
Logarithmic domain	✓		✓	✓	✓		✓	✓
5G NR polar code	✓						✓	
Rate-0, Rate-1 and repetition nodes	✓	✓	✓	✓	✓			✓
Fixed-point implementation	✓		✓	✓	✓			✓
AVX-512 implementation	✓							
Recursive template meta-programming	✓							

advantages, despite its reduced complexity compared with the SCS decoders of [21, 25]. However, to the best of our knowledge, there exists no literature considering the real time software implementation of the Log-SCS decoder.

Motivated by this, we propose a novel CRC-aided Fast Log-SCS decoder to address these concerns. The comparison with related publications on polar decoders is summarized in Table I and the main contributions of this paper are summarised as follows.

- The proposed CRC-aided Fast Log-SCS decoder exploits the unique properties of Log-SCS decoding with the aid of several novel techniques that were previously only considered in the context of the Fast-SSCL decoder of [16–19]. More specifically, the simplified path-metric (PM) computation of the rate-0, rate-1 and repetition nodes originally proposed for the Fast-SSCL decoder are applied to the corresponding sub-graphs of the CRC-aided Fast Log-SCS decoder conceived, which reduces the decoding complexity by 50% on average, compared with the Log-SCS polar decoder of [22]. We performed a detailed comparison with several polar decoders in the literature and our simulation results include the BLER performance of all the standarized block lengths of the 3GPP NR uplink polar codes in the open literature for the first time.
- We also conceive its real-time software implementation for attaining a drastically reduced decoding latency that is only 21% of that imposed by the state-of-the-art Fast SCL polar decoder implementation of [17]. This is achieved by employing recursive template meta-programming [26] for the first time, where the compiler acts as an interpreter and generates optimised implementations. More specifically, the template meta-programming complies the legitimate block lengths in advance and performs an arbitrary-length polar decoding operation using a recursive pattern.
- Each LLR that is input to the proposed decoder is quantised using a 32-bit fixed point number representation. Additionally, the proposed software implementation exploits the inherent built-in SIMD of the Intel x86 processor with 512-bit AVX (AVX-512) extensions, creating the first software-based highly-parallel processing aided SCS polar decoder in the open literature. More specifically, in contrast to [20], which exploits the parallelism associated with the L decoding candidates of the SCL polar decoder, the proposed Fast Log-SCS decoder can only consider parallelism within the processing of a single decoding candidate, since the SCS algorithm considers decoding candidates in series. In this way, a parallelism degree up to 16 may be achieved.

This paper is structured as follows. Section II reviews polar encoding and the Log-SCS decoding process. Following this, a fixed-point Fast Log-SCS polar decoder is proposed in Section III, which is implemented and characterised for x86 processors relying on SIMD instructions in Section IV. Finally, the main conclusions of our work and directions for future research are summarised in Section V.

II. Overview of polar encoding and Log-SCS decoding

The encoding and decoding process of polar codes is discussed in this section, in the context of the 3GPP NR uplink. More specifically, Sections II-A and II-B detail the core polar encoding and Log-SCS decoding operations, respectively. Following this, the CRC generation and the CRC will be discussed in Section II-C.

A. Polar encoder

In an $[N, K]$ polar encoder, the K input bits are polar encoded into $N = 2^m$ bits, which are mapped to N polarised bit channels in a predefined pattern, where m is a positive integer [5]. To be more specific, the polarisation weight quantifies the reliability associated with each of the N bit channels and may be calculated using density evolution, as detailed in [27]. The resultant $K = A + C$ most polarised bit channels are employed for transmitting A information and C CRC bits, as it will be detailed in Section II-C, while the remaining $(N-K)$ bit channels are employed for transmitting 0-valued bits, referred to as frozen bits.

Then the resultant N-bit vector $\mathbf{u}$ comprising A information bits, C CRC bits and $(N - K)$ frozen bits may be polar encoded into a vector $\mathbf{x}$ of N encoded bits, according to the modulo-2 matrix multiplication

$$\mathbf{x} = \mathbf{u}\mathbf{F}_2^{\otimes m}, \tag{1}$$

where, $\mathbf{F}_2^{\otimes m}$ is the generator matrix, and the superscript $\otimes m$ indicates the m^{th} Kronecker power of the matrix $\mathbf{F}_2$, which is expressed as

$$\mathbf{F}_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}.$$

An example is shown in Fig. 1, where the core information block $\mathbf{u} = [00000100]$ is polar encoded to the encoded vector $\mathbf{x} = [11001100]$ after the successive stages of XOR operations. The encoded bits $\mathbf{x}$ are output from the encoder for subsequent operations, which typically includes rate matching, as in the

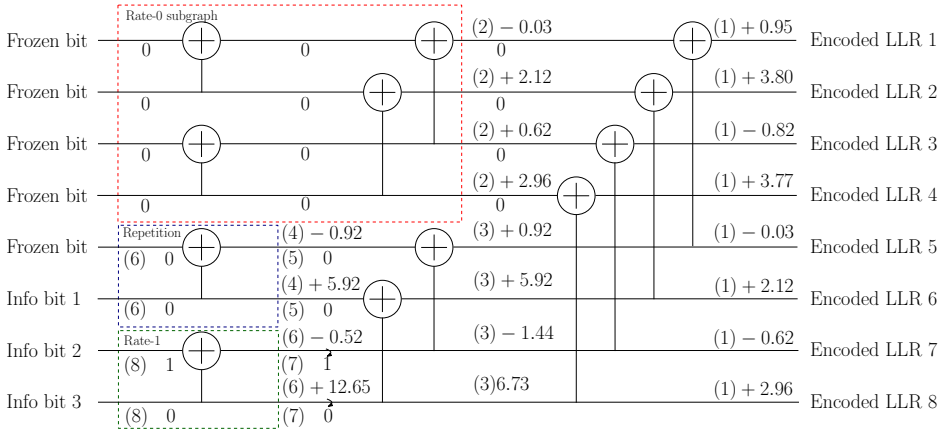


Fig. 1: The XOR operations performed in the polar encoder core, for an example where $A = 3$ information bits [010] are converted into $N = 8$ encoded bits with corresponding LLRs.

3GPP NR uplink polar code [22], giving a polar-coded block comprising G bits to be transmitted over the wireless channel, with the code rate R being defined as $R = A/G$.

B. Log-SCS polar decoder

Polar decoding can be thought of as a reverse process of encoding. Specifically, the channel LLRs input into the decoder are combined using a particular schedule of the soft XOR operations, using f, g and partial sum computations, as shown in Fig. 2.

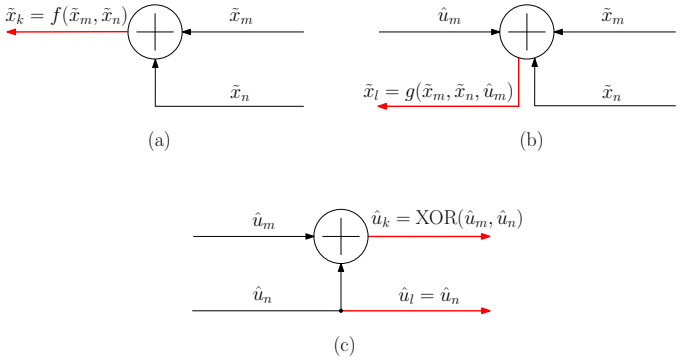


Fig. 2: The three computations that can be performed for a soft XOR in the polar code graph: (a) the f function, (b) the g function and (c) partial sum calculation.

As shown in Fig. 2(a), the first soft XOR operation, f function, for polar decoding is performed when two LLRs, $\tilde{x}_m$ and $\tilde{x}_n$, are provided on its connections at the right-hand side, respectively, which can be expressed as

$$\begin{aligned}\tilde{x}_k =& f(\tilde{x}_m, \tilde{x}_n)\\ =& 2\tanh^{-1}\left(\tanh\left(\frac{\tilde{x}_m}{2}\right)\tanh\left(\frac{\tilde{x}_n}{2}\right)\right) &\text{(2a)}\\ =& \tilde{f}(\tilde{x}_m, \tilde{x}_n) + \log(1+\exp(-|\tilde{x}_m + \tilde{x}_n|))\\ & - \log(1+\exp(-|\tilde{x}_m - \tilde{x}_n|)) &\text{(2b)}\\ \approx& \tilde{f}(\tilde{x}_m, \tilde{x}_n), &\text{(2c)}\end{aligned}$$

where $\tilde{f}(\tilde{x}_m, \tilde{x}_n) \triangleq \text{sign}(\tilde{x}_m)\,\text{sign}(\tilde{x}_n)\min(|\tilde{x}_m|, |\tilde{x}_n|)$.

Secondly, by combining the hard bit decision $\hat{u}_m$ with the input LLRs $\tilde{x}_m$ and $\tilde{x}_n$, as shown in Fig. 2(b), the g function is performed to generate a new LLR $\tilde{x}_l$ to be forwarded, which can be expressed as

$$\tilde{x}_l = g(\tilde{x}_m, \tilde{x}_n, \hat{u}_m) = (-1)^{\hat{u}_m}\tilde{x}_m + \tilde{x}_n. \quad (3)$$

Then, another hard bit decision $\hat{u}_n$ will be provided, together with the sign of LLR of the corresponding decoding candidate bit, as shown in Fig. 2(c). Together with the input $\hat{u}_m$, the partial sum computation of the bits $\hat{u}_k$ and $\hat{u}_l$ is performed, which can be expressed as

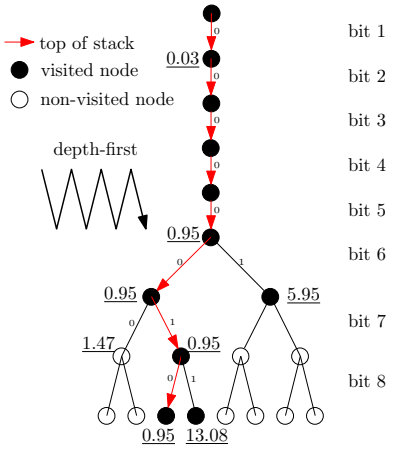
$$\hat{u}_k = \text{XOR}(\hat{u}_m, \hat{u}_n), \quad \hat{u}_l = \hat{u}_n. \quad (4)$$

By performing the three types of soft XOR computations in a prescribed successive cancellation schedule [15], an LLR may be obtained for each of the N connections on the left-hand edge of the polar code graph, one at a time in sequential order from top to bottom.

The Log-SCS algorithm uses a depth-first search of the code tree, as exemplified in Fig. 3, to develop a stack of up to S decoding candidates, with the processing switching to whichever candidate appears most likely, as the algorithm proceeds [24]. The likelihood of a decoding candidate is quantified by a PM, which is computed as a function of LLRs provided at the right-hand side of the graph [22]. To be more specific, the LLRs are first employed to compute the PMs that quantify the likelihood associated with a particular corresponding decoding candidate sequence $[\hat{\mathbf{u}}_i]_{i=1}^N$ of the N core information bits. When the i^{th} LLR $\tilde{x}_i$ in this sequence $[\hat{\mathbf{u}}_i]_{i=1}^N$ on the left-hand edge of the polar code graph is obtained, a PM ϕ_i may be updated for the decoding candidate according to [15]

$$\begin{aligned}\phi_i =& \phi_{i-1} + \ln\left(1+\exp\left[-\left(1-2\hat{u}_i\tilde{x}_i\right)\right]\right) &\text{(5a)}\\ \approx& \begin{cases}\phi_{i-1}, & \text{if } \hat{u}_i = \frac{1}{2}\left[1-\text{sign}(\tilde{x}_i)\right];\\ \phi_{i-1} + |\tilde{x}_i|, & \text{otherwise,}\end{cases} &\text{(5b)}\end{aligned}$$

where $\phi_0 = 0$. Here, the PM quantifies the likelihood of the decoding candidate, where lower PMs imply higher likelihoods.



PMs inside the stack	Step t					
	1	2···4	5	6	7	8
	0.03	0.03	0.95	0.95	0.95	0.95
	—	—	—	5.95	1.47	1.47

Fig. 3: Code trees in Log-SCS, when decoding the example $K = 3$ information or CRC bits from the $N = 8$ core encoded bits of Fig. 1.

The Log-SCS decoder's operation begins with only a single decoding candidate in the stack, which initially has undefined values for all N bits, and uses a stack to keep track of up to S decoding candidates at a time, where S is defined as the stack size. At each step of the Log-SCS algorithm, the decoding candidate at the top of the stack is selected and the value of its next undefined bit is considered. When either an information or a CRC bit is being decoded, the corresponding decoding candidate is updated to adopt a specific binary value for this bit and a replica is created that adopts the other binary value for this bit. The PMs for these two decoding candidates are updated and the replica is also inserted into the stack. The decoding candidates are sorted in the order of their PM, with the top element in the stack having the lowest PM and the bottom element having the highest. If the insertion of the replica into the stack has resulted in exceeding the stack size S, then the bottom element in the stack is eliminated. By contrast, if the next undefined bit in the top decoding candidate is a frozen bit, then this bit in the decoding candidate is set to zero and no replica is made. The PM of the decoding candidate is updated and the stack is sorted in the order of increasing PM.

C. CRC code appending and CRC

The error correction and detection performance of a polar code may be enhanced by appending a CRC code to the information bits, before polar encoding them. For example, a $C = 11$-bit CRC code relying on the generator polynomial $X^{11} + X^{10} + X^9 + X^5 + 1$ is appended to each block of A information bits, before polar encoding in the NR physical uplink control channel (PUCCH) for cases where we have $A \in [20, 1706]$ [1]. A false alarm occurs when an erroneous path passes the C-bit CRC. In the case of AWGN input, each of the C bits has a probability of 0.5 to pass the check. This gives a false alarm rate (FAR) of 2^{-C} for each path. Therefore, the FAR for S paths should be close to $S \cdot 2^{-C}$. However, this illustrates that when the stack size is large, the FAR will become excessively high. For example, for a Log-SCS decoder having an 11-bit CRC and a stack size of 128, the FAR becomes close to 2^{-5}. To maintain a reasonable FAR, we limit the number of CRC attempts to 8 and declare unsuccessful decoding, whenever 8 failing CRCs have been performed. This approach effectively uses $\log_2(8) = 3$ of the $C = 11$ CRC bits to improve the error correction capability of the polar decoder, while the remaining $C - 3 = 8$ CRC bits are used for error detection, for consistently maintaining FAR$= 2^{-8}$.

III. Fast Log-SCS decoder

This section proposes a Fast Log-SCS polar decoder and its software implementation that offers an improved decoding latency compared with the state-of-the-art Fast SSCL polar decoder. The proposed Fast Log-SCS decoder employs the fixed-point implementation detailed in Section III-A, as well as bespoke computations for rate-0, rate-1 and repetition sub-graphs, as detailed in Sections III-B to III-D, respectively. Note that we refer to the rate-0, rate-1 and repetition nodes of [6] as sub-graphs in this paper, in order to avoid confusion with the nodes of the tree exemplified in Fig. 5, which has a different structure from the tree used in [6]. Following the descriptions of our Fast Log-SCS decoder, we characterize its BLER performance and computational complexity in Section III-E and III-F, respectively.

A. Fixed-point implementation

As shown in Fig. 4, the proposed software implementation of the Fast Log-SCS decoder operates on the basis of fixed point arithmetic, where the classic scaling and quantisation operations are employed for preserving the error correction performance. Here, scaling is applied to the channel output LLRs $\boldsymbol{r}$ provided after the demodulation of the received signals, in order to match their range to that of the quantizer's dynamic range applied afterwards. Therefore, using an optimal scaling factor g strikes an attractive trade-off between the fixed-point overflow and underflow probabilities, in order to optimise the BLER performance. In the performance results of Section III-E, the value of the scaling factor g will be selected by maximizing the signal-to-distortion ratio at each SNR, according to the technique of [28].

Furthermore, the conversion from floating-point to the fixed-point arithmetic requires the selection of values for three parameters, namely the number of bits used for representing the channel LLRs, the internal LLRs and the PMs. In the proposed fixed-point software implementation, the channel LLRs are quantised using 24 bits, whereas the internal LLRs and PMs are quantised using 32 bits during the Log-SCS decoding process. In this way, a lower channel LLR width is employed so that an additional 8 bits become available for avoiding the overflow problem during the additions performed by summing the g functions and the PM computations. Further mitigation of overflow is achieved by considering that the

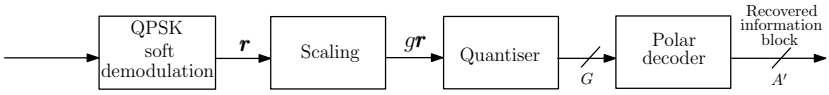


Fig. 4: Schematic of the fixed-point polar decoding.

SCS algorithm does not depend on the absolute values of the PMs, but rather it depends on the difference between PMs, when determining which path is most likely. Hence, only the difference between the PM and the minimum comparable PM are stored. The simulations of Section III-E will show that the fixed-point operation has similar error correction performance to that of a floating-point implementation.

B. Rate-0 sub-graph computation

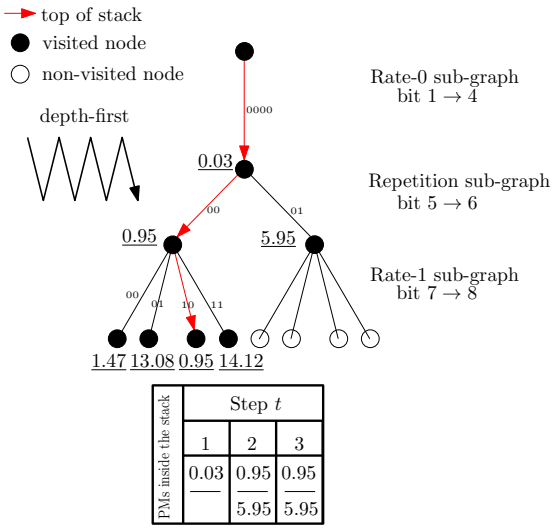


Fig. 5: Code trees in Log-SCS, when decoding the example $A = 3$ information bits from the $N = 8$ core encoded bits of Fig. 1, using the exact f and ϕ functions of (2a) and (5a).

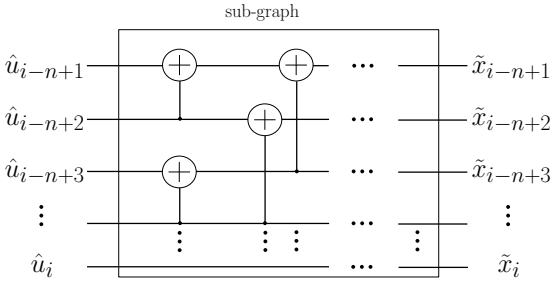


Fig. 6: Notation used for n-length rate-0, rate-1 and repetition sub-graphs.

A rate-0 sub-graph in the XOR graph is one which contains only frozen bits, as exemplified in Figs. 5 and 1. Since the decoder has *a priori* knowledge that all frozen bits have the value 0, and hence all encoded bits produced by the sub-graph also have a value of 0, the decoding of a rate-0 sub-graph may be carried out immediately, without propagating LLRs into the sub-graph. Using the notation of Fig. 6, the decoded bits $[\hat{u}_j]_{j=i-n+1}^{i}$ of a rate-0 sub-graph in any valid decoding candidate are zeros and the corresponding PM $\phi_i[s_1]$ may be updated as [18]

$$\phi_i[s_1] = \phi_{i-n} + \sum_{j=i-n+1}^{i} \left|\min\{0, \tilde{x}_j\}\right|. \tag{6}$$

Note that this calculation can be completed without the computation of any f or g functions within the sub-graph.

C. Rate-1 sub-graph computation

In contrast to the rate-0 sub-graph, a rate-1 sub-graph contains only information bits, as exemplified in Figs. 5 and 1. For a sub-graph comprising n information bits, 2^n decoding candidates can be produced considering each possible combination of the n bits. However, this leads to the exponential expansion of the candidate set, and it is impractical to consider all candidates, when n is not trivially small. Therefore, instead of considering all the 2^n decoding candidates, we adopt the most likely hard decision values for the $(n-2)$ most reliable bits and consider the four possible decoding candidates that are obtained by varying the values of the two least-reliable bits [6]. For example, when the decoder encounters a sub-graph, comprising $n = 4$ consecutive information bits, $\hat{u}_{i-3}, \hat{u}_{i-2}, \hat{u}_{i-1}, \hat{u}_i$, hard decisions are made for the two bits corresponding to the two LLRs from the set $\{\tilde{x}_{i-3}, \tilde{x}_{i-2}, \tilde{x}_{i-1}, \tilde{x}_i\}$ having the highest magnitudes, according to the signs of these two LLRs. Following this, the LLRs $\tilde{x}_{\min 1}$ and $\tilde{x}_{\min 2}$ of the two least-reliable bits in the sub-graph, where $\tilde{x}_{\min 1} \leq \tilde{x}_{\min 2}$, are used for computing the PMs for the four legitimate decoding candidates, and then generate four PMs, $\phi_i[s_1], \phi_i[s_2], \phi_i[s_3]$ and $\phi_i[s_4]$, according to the following rules,

$$\phi_i[s_1] = \phi_{i-4}, \tag{7a}$$
$$\phi_i[s_2] = \phi_{i-4} + \left|\tilde{x}_{\min 1}\right|, \tag{7b}$$
$$\phi_i[s_3] = \phi_{i-4} + \left|\tilde{x}_{\min 2}\right|, \tag{7c}$$
$$\phi_i[s_4] = \phi_{i-4} + \left|\tilde{x}_{\min 1}\right| + \left|\tilde{x}_{\min 2}\right|. \tag{7d}$$

In this case, only three additional decoding candidates are generated and stored in the stack, rather than the 15 that would be required, if all possible candidates were computed.

D. Repetition sub-graph computation

When only the last bit u_i in a group of $n = 2^v, v \geq 1$ bits is an information bit, with the rest being frozen bits, then the sub-graph is referred to as repetition sub-graph, as exemplified in Figs. 5 and 1. The corresponding n encoded bits have only two possible hard decisions, either n zeros or n ones, depending on whether the decoding candidate uses $\hat{u}_i = 0$ or $\hat{u}_i = 1$, respectively. In order to consider both options, we store both

possible decoding candidates in the stack and update the PM according to the following rules [18],

$$\phi_i[s_1] = \phi_{i-n} + \sum_{j=i-n+1}^{i} |\min\{0, \tilde{x}_j\}|\,, \tag{8a}$$

$$\phi_i[s_2] = \phi_{i-n} + \sum_{j=i-n+1}^{i} |\max\{0, \tilde{x}_j\}|\,, \tag{8b}$$

where the updated $\phi_i[s_1]$ stores the PM when $\hat{u}_i = 0$ and $\phi_i[s_2]$ stores the PM when $\hat{u}_i = 1$. In this way, these PM calculations are performed without the computation of f or g functions, as that in the rate-0 or rate-1 sub-graph.

E. Error correction performance

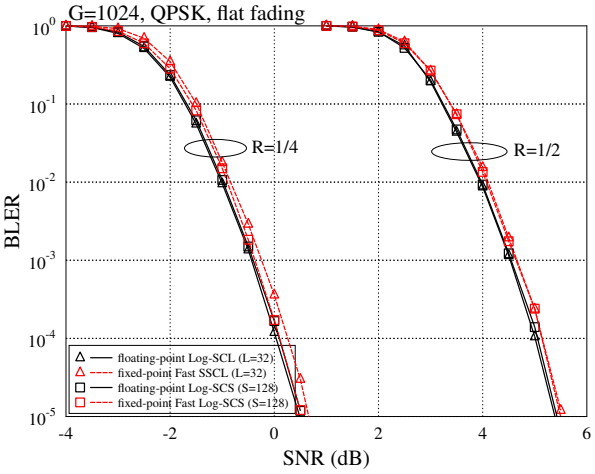


Fig. 7: BLER performance of the floating-point Log-SCL decoder of [15], the floating-point Log-SCS polar decoder of [22], 32-bit fixed-point Fast SSCL decoder of [17] and the proposed 32-bit fixed-point Fast Log-SCS polar decoder for $G = 1024$ polar codes of different coding rates R over a flat uncorrelated Rayleigh fading channel, where QPSK modulation is employed.

When a scaling factor g optimised for the particular channel SNR E_s/N_0 and 32-bit fixed-point quantization is applied to the channel LLRs, the BLER performance of the proposed 32-bit fixed-point Fast Log-SCS decoder becomes indistinguishable from that of the floating-point Log-SCS decoder of [22], especially for higher coding rates. This may be observed in Fig. 7 for the case of $G = 1024$-bit polar codes of various coding rates R ranging from $1/8$ to $1/2$ transmitting over a flat uncorrelated Rayleigh fading channel, where QPSK modulation is employed. Additionally, the BLER performance of the floating-point Log-SCL decoder of [15] and Fast SSCL decoder of [17] is also included in Fig. 7 as the benchmarks of our proposed 32-bit fixed-point Fast Log-SCS decoder.

Furthermore, Figs. 8 and 9 demonstrate the SNR required for the $S = 128$ floating-point Log-SCS decoder of [22] and the proposed $S = 128$ fixed-point Fast Log-SCS decoder to achieve a BLER of 10^{-3} for different combinations of

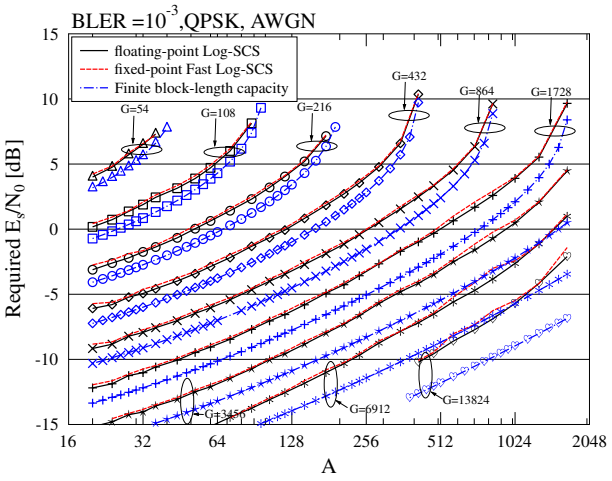


Fig. 8: The SNR required for the $S = 128$ floating-point Log-SCS decoder of [22] and the proposed 32-bit fixed-point $S = 128$ Fast Log-SCS decoder to achieve a BLER of 10^{-3} for different combinations of information block length A and encoded block length G of the 3GPP NR PUCCH polar code, where QPSK modulation is used for communication over an AWGN channel.

information block length A and encoded block length G of the 3GPP NR PUCCH polar code over an AWGN channel and a flat uncorrelated Rayleigh fading channel, respectively. The finite block length capacity shown in Fig. 8 is calculated according to the $\mathcal{O}(n^{-2})$ meta-converse Polyanskyi-Poor-Verdù (PPV) upper bound of [29]. We can see that when further combining the techniques discussed in Sections III-A to III-D together, the BLER performance of the 32-bit fixed-point Fast Log-SCS decoder remains similar to that of the floating-point Log-SCS decoder, while enabling a significantly reduced decoding complexity and latency, as it will be discussed in Sections III-F and IV-B, respectively.

F. Computational complexity

The overall decoding complexity of the proposed Fast Log-SCS decoder may be quantified in terms of the number of f, g and ϕ function evaluation performed in the decoder at the specific channel SNR, where a BLER of 10^{-3} is achieved. Fig. 10 compares the overall complexity of the $L = 32$ Log-SCL decoder of [15], of the $L = 32$ Fast 32-bit SSCL decoder of [17], of the $S = 128$ floating-point Log-SCS decoder of [22] and of the proposed $S = 128$ Fast 32-bit fixed-point Log-SCS decoder for $G = 1024$-bit polar codes having various information block lengths A. In all scenarios, QPSK modulation is used for communication over a flat uncorrelated Rayleigh fading channel. Here, it may be observed that exploiting the simplified rate-0, rate-1 and repetition sub-graph computations of the proposed Fast Log-SCS polar decoder yields a 50% reduction in the complexity, compared to the $S = 128$ floating-point Log-SCS decoder of [22].

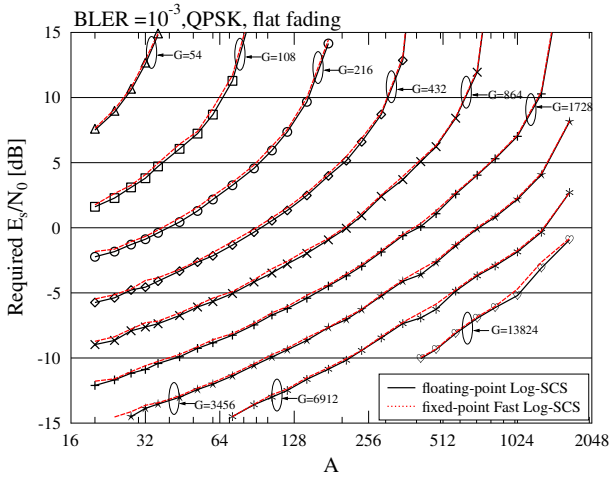


Fig. 9: The SNR required for the $S = 128$ floating-point Log-SCS decoder of [22] and the proposed 32-bit fixed-point $S = 128$ Fast Log-SCS decoder to achieve a BLER of 10^{-3} for different combinations of information block length A and encoded block length G of the 3GPP NR PUCCH polar code, where QPSK modulation is used for communication over a flat uncorrelated Rayleigh fading channel.

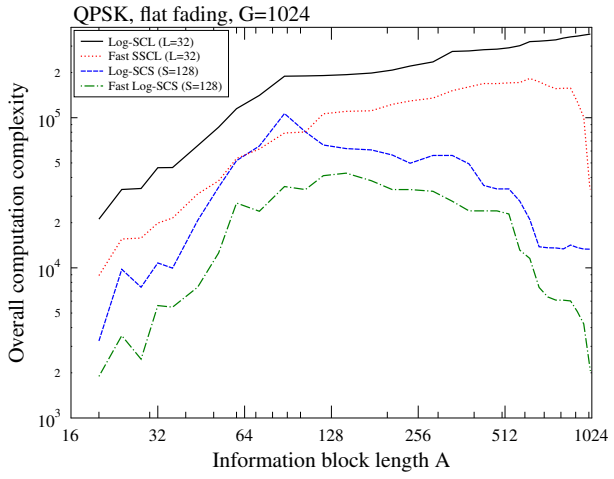


Fig. 10: Overall computational complexity of the $L = 32$ Log-SCL decoder of [15], $L = 32$ Fast 32-bit SSCL decoder of [17], $S = 128$ floating-point Log-SCS decoder of [22] and the proposed $S = 128$ Fast 32-bit fixed-point Log-SCS decoder for $G = 1024$-bit polar codes with various information block lengths A, where QPSK modulation is used for communication over a flat uncorrelated Rayleigh fading channel.

Algorithm 1 Main function for the Fast Log-SCS polar decoder

Global: int stack_index, bit_index;
function main()
1: PolarDecoder<1024> polar_decoder; /* Initialize an stack decoder with $N = 1024$ */
2: **Initialize** llrs, bits, PM, decod_bit and stack_index=1; /* Initialize for decoding candidates in the stack decoder. */
/* 3-6: Complete the decoding when the length of the top element in the stack reaches N */
3: **while** decod_bit[stack_index]<N **do**
4: bit_index=0;
5: polar_decoder.decode(llrs+stack_index*N, bits+stack_index*N, PM, decod_bit);
6: **end while**
end function

IV. SIMD IMPLEMENTATION OF THE PROPOSED FAST LOG-SCS DECODER

In this section, we introduce an implementation of the proposed fixed-point Fast Log-SCS decoder using SIMD instructions on an x86 CPU that supports AVX-512. The implementation is detailed in Section IV-A and characterised in Section IV-B.

A. SIMD implementation

A number of techniques are adopted in order to optimise the throughput and latency of the proposed implementation of the Fast Log-SCS decoder. Firstly, we employ recursive template meta-programming [26], as shown in Algorithms 1 and 2, with the special case of the code length $Nsc = 1$ shown in Algorithm 3. This technique supports a recursive programming style that is easy to maintain, while also enabling the computer to optimize and pre-process as much of the processing as possible at compile time. The summary of the functions employed in the algorithms of this paper are shown in Table II. To be more specific, in the process of meta-programming, polar decoders each having an integer power-of-2 block length N up to the maximum used in 3GPP NR, namely, 1024 are instantiated recursively during the compilation process. The recursive structure of the polar codes allows the recursive computations of the f and g functions to be unrolled at compile time, so that the computations of the f and g functions do not have to be repeatedly invoked every time the programme is running, producing highly optimised compiled code.

The AVX-512 implementation of the f and g functions is detailed in the C++ code of Algorithms 4 and 5, respectively. Here, the **__m512i** registers have 512 bits and the LLRs are quantised using 32 bits as discussed in Section III-A, resulting in a maximum parallelism degree of $P_1 = 16$. Note that in this paper, the parallelism of the SIMD instructions is achieved by performing parallel computations for up to 16 bits within the same block, rather than performing parallel computations for bits from 16 separate blocks, as discussed for the SCL

TABLE II: Summary of the functions used in Algorithms 1-5.

Name	Abbreviation	Function
_mm512_set1_epi32(int a)	set(a)	broadcast the 32-bit integer a to all elements of dst
_mm512_load_epi32(void const *mem_addr)	load(mem_addr)	load 512 bits from memory into dst
_mm512_abs_epi32(Is32vec16 a)	abs(a)	compute the absolute value of a
_mm512_min_epi32(Is32vec16 a, Is32vec16 b)	min(a,b)	compute the minimum value of a and b
_mm512_cmpgt_epi32_mask(Is32vec16 a, Is32vec16 b)	mask_cmpgt(a,b)	compare a and b, and store in mask vector
_mm512_mask_set1_epi32 (Is32vec16 src,__mmask16 mask,int a)	mask_set(src,mask,a)	broadcast 32-bit integer a to dst using writemask mask
_mm512_storeu_epi32(voild* mem_addr,Is32vec16 a)	store(a,b)	store 512 bits from a into memory address mem_addr.

Algorithm 2 Template meta-programming for the Fast Log-SCS polar decoder

Global:

```
1: template <int Nsc>; /* Class for a polar decoder having a length of Nsc */
2: class PolarDecoder{
     /* A polar decoder of length Nsc recursively contains a polar decoder of length Nsc/2 */
3:   PolarDecoder<Nsc/2> next;
4:   public:
5:   void decode(int *llrs, int *bits, int *decod_bit, int *PM) {
6:     if is_rate0_subgraph(decod_bit, Nsc) then
7:       decode_rate0(llrs, bits, decod_bit, PM);
8:     else if is_rate1_subgraph(decod_bit, Nsc) then
9:       decode_rate1(llrs, bits, decod_bit, PM);
10:    else if is_rep_subgraph(decod_bit,Nsc) then
11:      decode_rep(llrs, bits, decod_bit, PM);
12:    else
13:      if decod_bit[stack_index]==bit_index then
14:        this → f_function(llrs);
15:      end if
         /* Use the Nsc/2-length polar decoder to decode the top half */
16:      next.decode(llrs, bits, decod_bit,PM);
17:      if decod_bit[stack_index]==bit_index then
18:        this → g_function(llrs+Nsc/2, bits+Nsc/2);
19:      end if
         /* Use the Nsc/2-length polar decoder to decode the bottom half */
20:      next.decode(llrs+Nsc/2, bits+Nsc/2, decod_bit, PM);
21:    end if
22:  }
23: };
```

Algorithm 3 Template meta-programming special case when Nsc=1

Global:

```
1: class PolarDecoder<1>{
2:   void decode(int *llrs, int *bits, int *decod_bit, int *PM) {
3:     if decod_bit[stack_index]==bit_index then
4:       PM_calculate(PM,llrs,bits); /* Calculate the PM of the top element in the stack */
5:       decod_bit[stack_index]++; /* Extend the length of top candidate by 1 */
6:       expand_stack(llrs,bits,decod_bit,PM); /* Expand the current stack size or remove the bottom element to insert a new element based on PM */
7:       stack_index=sort_stack(PM); /* Enable stack_index to point to the top element with the highest PM in the stack [24] */
8:     end if
9:   bit_index++; /* Indicate calculations required for decoding the next bit */
10:  }
11: };
```

decoder of [20]. In this way, our approach provides a benefit even when 16 blocks having the same information and encoded block lengths are not available for decoding at the same time. Furthermore, the approach to parallelism used in [20] cannot be applied to SCS decoding, since it is unlikely that the decoding of the 16 blocks would stay synchronised throughout the dynamic SCS decoding process.

B. Latency, Throughput and Memory requirement

We have compiled our AVX-512-enabled C++ code using the Intel C++ Compiler with the aid of -Ofast optimisation, and characterised it on an Intel(R) Xeon(R) Gold 6138 CPU operating at $f_1 = 2.00$ GHz. The latency D averaged over 100 runs and required to perform the $S = 128$ Fast Log-SCS decoding of each block is characterised as a function of the number of information bits A and the resultant encoded bits G in each block.

Fig. 11 characterises the average latency D of the Fast Log-SCS polar decoder and AVX-512 Fast Log-SCS polar decoder for encoded block lengths of $G = 512$, 1024 and 2048 at various information block lengths A for the case, where the channel SNR is such that a BLER of 10^{-3} is achieved. Note that in contrast to the Fast SSCL decoder, whose latency is independent of the channel SNR, the average latency of the Fast Log-SCS polar decoder decreases as the SNR increases. Note that the encoded block lengths of $G = 512$ and 1024 are obtained using the 3GPP NR PUCCH polar codes having core block lengths of $N = 512$ and 1024, respectively. By contrast, density evolution with the Gaussian approximation (DE-GA) algorithm is employed for parameterizing an $N = 2048-$bit polar code, which is used for generating the encoded block

Algorithm 4 f function calculation using AVX-512 instructions

```
Global: static const Is32vec16 const_0 = set(0);
        /* constant-zero AVX512 vector */
function f_function(int *llrs)
 1: int i=0;
    /* Use different masks if the parallelism is less 16 */
 2: if Nsc>=16 then
 3:    while i<Nsc/2 do
 4:       Is32vec16 x_m = load(llrs+i); /* load llrs of x_m to
          the AVX512 vector */
 5:       Is32vec16 x_n = load(llrs+i + Nsc/2); /* load llrs of
          x_n to AVX512 vector */
 6:       Is32vec16 min_a_b = min(abs(x_m),abs(x_n));
 7:       Is32vec16 sign_x_m1 = mask_set(const_0,
          mask_cmpgt(x_m, const_0), 1);
 8:       Is32vec16 sign_x_m2 =mask_set(const_0,
          mask_cmpgt(x_m, const_0), -1);
 9:       Is32vec16 sign_x_m = sign_x_m1 | sign_x_m2;
10:       Is32vec16 sign_x_n1 = mask_set(const_0,
          mask_cmpgt(x_n, const_0), 1);
11:       Is32vec16 sign_x_n2 =mask_set(const_0,
          mask_cmpgt(x_n, const_0), -1);
12:       Is32vec16 sign_x_n = sign_x_n1 | sign_x_n2;
13:       Is32vec16 result = sign_x_m*sign_x_n* min_a_b;
14:       store(llrs+i, result); /* store the AVX-512 vector to
          the corresponding llrs memory address */
15:       i+=16;
16:    end while
17: end if
end function
```

Algorithm 5 g function calculation using AVX-512 instructions

```
Global: static const Is32vec16 const_1 = set(1);
        static const Is32vec16 const_2 = set(2) ;
function g_function(int *llrs, int *bits)
 1: int i=0;
    /* Use different masks if the parallelism is less 16 */
 2: if Nsc>=16 then
 3:    while i<Nsc/2 do
 4:       Is32vec16 x_m = load(llrs+i-Nsc/2);
 5:       Is32vec16 x_n = load(llrs+i);
 6:       Is32vec16 u_m = load(bits+i-Nsc/2);
 7:       Is32vec16 temp = const_1-u_m*const_2;
 8:       Is32vec16 result = temp*x_m+x_n;
 9:       store(llrs+i, result);
10:       i+=16;
11:    end while
12: end if
end function
```

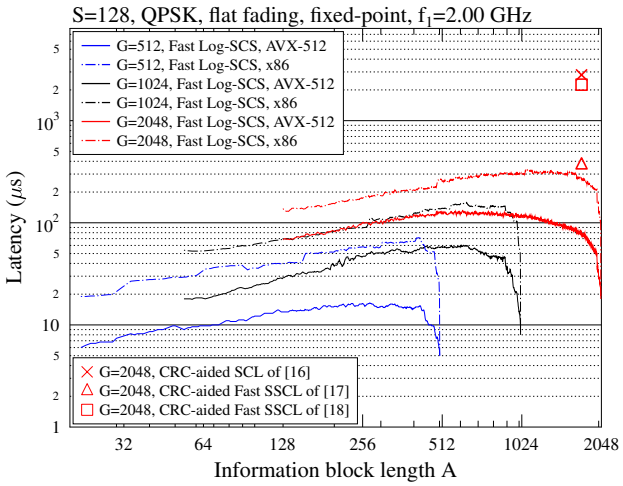


Fig. 11: Average latency D of the proposed $S = 128$ AVX-512 fixed-point Fast Log-SCS decoder operating at $f_1 = 2.00$ GHz for $G = 512, 1024$ and 2048 bits with various information block lengths A, as well as the normalised latency $\bar{D}$ of the works in [16–18] for $G = 2048$ and $A = 1723$ bits, where QPSK modulation is employed for communication over a flat uncorrelated Rayleigh fading channel in all cases.

length of $G = 2048$, since the 3GPP NR PUCCH polar codes only support a maximum core block length of $N = 1024$ bits. Fig. 11 also includes benchmark results of the works in [16–18] for $G = 2048$ and $A = 1723$ bits.

Furthermore, Table III compares the latency, throughput and memory requirement of different polar decoders when achieving a BLER of 10^{-3} for block lengths of $A = 1723$ and $G = 2048$. More specifically, the proposed CRC-aided Fast Log-SCS decoder is compared with the AVX2 Fast SCL decoders of [16], [17] and [18], which were characterised using clock frequencies of $f_2 = 3.90$ and $f_3 = f_4 = 3.40$ GHz, respectively. These implementations use LLR widths of 8 or 32 bits and since AVX2 has register widths of 256 bits, a parallelism order of $P_2 = 32$ and $P_3 = P_4 = 8$ is achieved. Note that while the polar decoders of [16, 17] employ 32-bit quantization of the LLRs, which is the same as that of our proposed CRC-aided Fast Log-SCS decoder, the Fast SSCL implementation of [18] uses 8-bit fixed-point LLRs. For the sake of fair comparison, we define the normalised latency $\bar{D}$ as

$$\bar{D} = \frac{P_i \times f_i \times D}{P_1 \times f_1}, \quad i = 2, 3, 4, \tag{9}$$

where $P_1 = 16$ and $f_1 = 2.00$ GHz represent the order of parallelism and the clock frequency of the proposed CRC-aided Fast Log-SCS decoder,respectively. Furthermore, P_i and f_i represent the order of parallelism and the clock frequency of the decoder under consideration. As shown in Table III and plotted in Fig. 11, our proposed AVX-512 Fast Log-SCS decoder has a normalised decoding latency $\bar{D}$ that is as low as 21% that of the best-performing benchmarker, namely, the CRC-aided Fast SSCL decoder of [17]. The throughput T of a software-based polar decoder is proportional to the reciprocal

TABLE III: Latency, throughput and memory requirement comparisons of different polar decoders when achieving a BLER of 10^{-3} for block lengths of $A = 1723$, $G = 2048$.

Platform	Intel(R) Xeon(R) Gold 6138		I5-6600K	I7-2600	I7-2600
f_i (GHz)	$f_1 = 2.00$		$f_2 = 3.90$	$f_3 = 3.40$	$f_4 = 3.40$
Algorithm	CRC-aided Fast Log-SCS		CRC-aided Fast SSCL [18]	CRC-aided Fast SSCL [17]	CRC-aided SCL [16]
Required SNR (dB)	11.90		11.98	11.98	11.75
LLR bit-width	32		8	32	32
AVX register bit-width	32	512	256		
List size L or stack size S	S =128		L =32		
D (μs)	274	79	577	433	3300
$\bar{D}$ (μs)	-	79	2252	368	2805
T (Mbps)	6.29	21.81	2.99	3.98	0.52
T (Mbps)	-	21.81	0.77	4.68	0.61
Memory (KB)	1096		82	280	280

of its latency, according to $T = A/D$, whereas the normalised throughput $\bar{T}$ can be defined as $\bar{T} = A/\bar{D}$. Table III shows that the normalised throughput of our proposed AVX-512 CRC-aided Fast Log-SCS decoder is in excess of 6 times higher than that of the Fast SSCL decoder of [17]. The performance of the proposed Fast Log-SCS decoder run on a 32-bit x86 CPU is also characterised in Table III, which demonstrates that AVX-512 offers a 4.66-fold throughput improvement, in the case of our CRC-aided Fast Log-SCS polar decoder. Note that the throughput considered in Table III is the throughput per CPU core. For Intel(R) Xeon(R) Gold 6138 equipped with 20 cores, the overall throughput is 416.20 Mbps, which is sufficient for the uplink polar decoding in the PUCCH of 5G NR.

Table III also quantifies the maximum amount of memory required for storing the partial sum bits and the LLRs generated by the f, g and ϕ functions in the various implementations considered. These comparisons reveal that the proposed AVX implementation of the CRC-aided Fast Log-SCS decoder requires 13 times higher memory than the Fast SSCL decoder of [18], and 3 times higher memory than the CRC-aided Fast SSCL decoder of [17]. Hence, the improved latency and throughput of our Fast Log-SCS polar decoder is achieved at the cost of a higher memory requirement. However, it may be argued that the 1096 KB memory required is insignificant in the context of a high-performance CPU.

V. Conclusions

In this paper, we have proposed a novel Fast Log-SCS decoder for the 3GPP NR uplink control channel's polar code, which reduces the complexity by 50%, compared with the state-of-the-art CRC-aided Fast polar SSCL decoder. We have also proposed an AVX-512 software implementation, which increases the normalised throughput by a factor of 6 and reduces the normalised latency to 21% of the existing state-of-the-art AVX Fast SSCL polar decoder.

References

[1] 3GPP TS 38.212 V15.1.1, "NR Multiplexing and channel coding," *3rd Generation Partnership Project Std. 3GPP*, 2018.

[2] D. Hui, S. Sandberg, Y. Blankenship, M. Andersson, and L. Grosjean, "Channel coding in 5G new radio: A tutorial overview and performance comparison with 4G LTE," *IEEE Vehicular Technology Magazine*, vol. 13, no. 4, pp. 60–69, 2018.

[3] Z. B. K. Egilmez, L. Xiang, R. G. Maunder, and L. Hanzo, "The development, operation and performance of the 5G polar codes," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 96–122, 2019.

[4] Z. Babar, Z. B. K. Egilmez, L. Xiang, D. Chandra, R. G. Maunder, S. X. Ng, and L. Hanzo, "Polar codes and their quantum-domain counterparts," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 1, pp. 123–155, 2019.

[5] E. Arikan, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[6] A. Alamdar-Yazdi and F. R. Kschischang, "A simplified successive-cancellation decoder for polar codes," *IEEE Communications Letters*, vol. 15, no. 12, pp. 1378–1380, 2011.

[7] G. Sarkis and W. J. Gross, "Increasing the throughput of polar decoders," *IEEE Communications Letters*, vol. 17, no. 4, pp. 725–728, 2013.

[8] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast polar decoders: Algorithm and implementation," *IEEE Journal on Selected Areas in Communications*, vol. 32, no. 5, pp. 946–957, 2014.

[9] C. Husmann, P. C. Nikolaou, and K. Nikitopoulos, "Reduced latency ML polar decoding via multiple sphere-decoding tree searches," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 2, pp. 1835–1839, 2017.

[10] H. Zhou, W. Song, W. J. Gross, Z. Zhang, X. You, and C. Zhang, "An efficient software stack sphere decoder for polar codes," *IEEE Transactions on Vehicular Technology*, vol. 69, no. 2, pp. 1257–1266, 2019.

[11] H. Zhou, X. Tan, W. J. Gross, Z. Zhang, X. You, and C. Zhang, "An improved software list sphere polar decoder with synchronous determination," *IEEE Transactions on Vehicular Technology*, vol. 68, pp. 5236–5245, June 2019.

[12] P. Giard, G. Sarkis, C. Thibeault, and W. J. Gross, "Fast software polar decoders," in *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pp. 7555–7559, IEEE, 2014.

[13] I. Tal and A. Vardy, "List decoding of polar codes," in *Information Theory Proceedings (ISIT), 2011 IEEE International Symposium on*, pp. 1–5, IEEE, 2011.

[14] I. Tal and A. Vardy, "List decoding of polar codes," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2213–2226, 2015.

[15] A. Balatsoukas-Stimming, M. B. Parizi, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," *IEEE Transactions on Signal Processing*, vol. 63, no. 19, pp. 5165–5179, 2015.

[16] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Increasing the speed of polar list decoders," in *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pp. 1–6, IEEE, 2014.

[17] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast list decoders for polar codes," *IEEE Journal on Selected Areas in Communications*, vol. 34, no. 2, pp. 318–328, 2016.

[18] M. Léonardon, A. Cassagne, C. Leroux, C. Jégo, L.-P. Hamelin, and Y. Savaria, "Fast and flexible software polar list decoders," *Springer Journal of Signal Processing Systems (JSPS)*, pp. 1–16, January 2019.

[19] S. A. Hashemi, C. Condo, and W. J. Gross, "Fast simplified successive-cancellation list decoding of polar codes," in *Wireless Communications and Networking Conference Workshops (WCNCW), 2017 IEEE*, pp. 1–6, IEEE, 2017.

[20] B. Le Gal, C. Leroux, and C. Jego, "Multi-Gb/s software decoding of polar codes," *IEEE Transactions on Signal Processing*, vol. 63, no. 2, pp. 349–359, 2015.

[21] K. Niu and K. Chen, "Stack decoding of polar codes," *Electronics Letters*, vol. 48, no. 12, pp. 695–697, 2012.

[22] L. Xiang, Z. B. K. Egilmez, R. G. Maunder, and L. Hanzo, "CRC-aided logarithmic stack decoding of polar codes for ultra reliable low latency communication in 3GPP new radio," *IEEE Access*, vol. 7, pp. 28559–28573, 2019.

[23] H. Aurora and W. J. Gross, "Towards practical software stack decoding of polar codes," *arXiv preprint arXiv:1809.03606*, 2018.

[24] H. Aurora, C. Condo, and W. J. Gross, "Low-complexity software stack decoding of polar codes," in *Circuits and Systems (ISCAS), 2018 IEEE International Symposium on*, pp. 1–5, IEEE, 2018.

[25] K. Niu and K. Chen, "CRC-aided decoding of polar codes," *IEEE Communications Letters*, vol. 16, no. 10, pp. 1668–1671, 2012.

[26] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*. Addison-Wesley Professional, 2004.

[27] R. Mori and T. Tanaka, "Performance of polar codes with the construction using density evolution," *IEEE Communications Letters*, vol. 13, no. 7, pp. 519–521, 2009.

[28] Y. Wu and B. D. Woerner, "The influence of quantization and fixed point arithmetic upon the BER performance of turbo codes," in *Vehicular Technology Conference, 1999 IEEE 49th*, vol. 2, pp. 1683–1687, IEEE, 1999.

[29] T. Erseghe, "Coding in the finite-blocklength regime: Bounds based on laplace integrals and their asymptotic approximations," *IEEE Transactions on Information Theory*, vol. 62, no. 12, pp. 6854–6883, 2016.