FAST SOFTWARE POLAR DECODERS

Pascal Giard[†], Gabi Sarkis[*], Claude Thibeault[†], and Warren J. Gross[*]*

[*]McGill University, Montréal, Québec, Canada
[†]École de technologie supérieure, Montréal, Québec, Canada

ABSTRACT

Among error-correcting codes, polar codes are the first to provably achieve channel capacity with an explicit construction. In this work, we present software implementations of a polar decoder that leverage the capabilities of modern general-purpose processors to achieve an information throughput in excess of 200 Mbps, a throughput well suited for software-defined-radio applications. We also show that, for a similar error-correction performance, the throughput of polar decoders both surpasses that of LDPC decoders targeting general-purpose processors and is competitive with that of state-of-the-art software LDPC decoders running on graphic processing units.

Index Terms— Decoding, Polar Codes, Error-Correction, Software-Defined-Radio

1. INTRODUCTION

Being the first error-correcting codes with an explicit construction to provably achieve the symmetric channel-capacity, polar codes have drawn significant attention since their introduction in [1, 2]. Many hardware implementations of the successive-cancellation (SC) algorithm were able to exploit the regular structure of polar codes to reduce implementation complexity [3–5].

Recently, new decoding algorithms derived from SC were proposed with the explicit aim of increasing throughput without degrading error-correction performance: simplified successive-cancellation (SSC) [6], maximum-likelihood simplified successive-cancellation (ML-SSC) [7], two phase SC (TPSC) [8] and Fast-SSC [9].

Contribution and Outline: This paper presents fast software polar decoders with an information throughput that can exceed 200 Mbps using only one core of an Intel i7-2600 x86 CPU running at 3.4 GHz. To that end, Section 2 reviews polar codes, and Section 3 briefly reviews the Fast-SSC decoding algorithm. Then, Section 4 details different implementations of the algorithm on an x86 CPU featuring single-instruction-multiple-data (SIMD) capability. Section 5 provides throughput results. Finally, Section 6 concludes this paper.

2. POLAR CODES

Polar codes are constructed recursively by applying linear transformations that create N channels where, as $N \to \infty$, the probability of an error in transmission of a subset of the N channels tends to 0, and to 0.5 for the remaining ones [2].

In an (N, k) polar code, we use the k most reliable bits to transmit the information bits and set the remaining $N - k$ bits, called the frozen bits, to zero. The location of the information and frozen bits, for an additive white Gaussian noise (AWGN) channel, can be determined using the method described in [10]. To improve the bit error rate (BER), systematic encoding can used, [11], as is done in this work.

Polar codes are decoded using successive-cancellation decoding [2], which works by successively estimating a bit $\hat{u}_i$, $i = 0, ..., N-1$, using the channel output y and the previously estimated bits $\hat{u}_0$ to $\hat{u}_{i-1}$.

As shown in [3], this can be carried out without the use of multiplications or divisions by expressing probabilities as log-likelihood-ratios (LLRs), denoted λ, and applying the min-sum (MS) approximation. The decision rule for $\hat{u}_i$ becomes

$$\hat{u}_i = \begin{cases} 0, & \text{if } \lambda_{u_i} \geq 0; \\ 1, & \text{otherwise;} \end{cases} \tag{1}$$

where, for λ_{u_0} and λ_{u_1},

$$\lambda_{u_0} = f(\lambda_{v_0}, \lambda_{v_1}) = \text{sign}(\lambda_{v_0})\text{sign}(\lambda_{v_1}) \min(|\lambda_{v_0}|, |\lambda_{v_1}|); \tag{2}$$

and

$$\lambda_{u_1} = g(\lambda_{v_0}, \lambda_{v_1}, \hat{u}_0) = \begin{cases} \lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 0, \\ -\lambda_{v_0} + \lambda_{v_1} & \text{when } \hat{u}_0 = 1. \end{cases} \tag{3}$$

While the SC decoding algorithm of polar codes has been proven to achieve the channel capacity asymptotically in code length, it inherently has a low throughput due to the serial update of the decisions $\hat{u}_i$.

3. TREE STRUCTURE AND FAST-SSC DECODING

As mentioned in Section 2, a polar code is built recursively: a polar code of length N is the concatenation of two constituent polar codes of lengths $N/2$. Hence, this construction can be
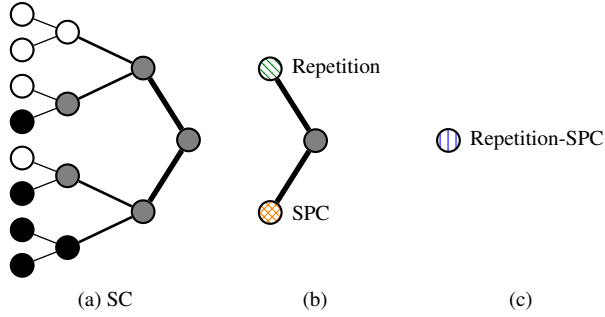
Fig. 1: Decoder trees for (a) the SC decoder, and the Fast-SSC decoder (b) without and (c) with Repetition-SPC nodes.

represented as a binary tree where each node corresponds to a constituent code [6, 7, 9]. Fig. 1a shows the tree representation of a polar code where every node is visited, be it an information node (black) or a frozen node (white).

3.1. Fast-SSC Decoding

The Fast-SSC algorithm consists of several operations and is thouroughly described in [9], we here focus on the key aspect of Fast-SSC decoding. It works by recognizing more types of constituent codes that, when decoded directly, significantly reduce the size and depth of the decoding tree without degrading error-correction performance [9]. In this section, we briefly review these codes.

Repetition codes: occur when only the last bit of a constituent code is an information bit. Shown as the node with a green striped pattern in Fig. 1b, repetition codes can be efficiently decoded by summing the input LLRs. Using threshold detection, the sign of the sum is used to determine the result, which is then replicated to form the vector of estimated bits.

Single-parity-check codes: when bits of a constituent code are all information bits except the first one, it is a single-parity-check code (SPC). An SPC node is shown as a cross-hatched orange node in Fig. 1b. Such codes are decoded by first calculating the hard decision of each LLR and by calculating the parity of these decisions. The estimate of the bit with the smallest LLR magnitude is flipped when the parity constraint is unsatisfied.

Repetition-SPC codes: correspond to nodes whose left child is a repetition code and the right an SPC one, shown as a node with blue vertical lines in Fig. 1c. The speculative nature of decoding such a code is beneficial in a hardware implementation, but not in software. Therefore, in this work, the calculations for the SPC code are delayed until the decision about the repetition code is taken.

If the Fast-SSC decoding algorithm was to only recognize the repetition and SPC codes, the code of Fig. 1a would be decoded in three steps as shown in Fig. 1b. Including the Repetition-SPC codes, the code is decoded in only one step as shown in Fig. 1c.
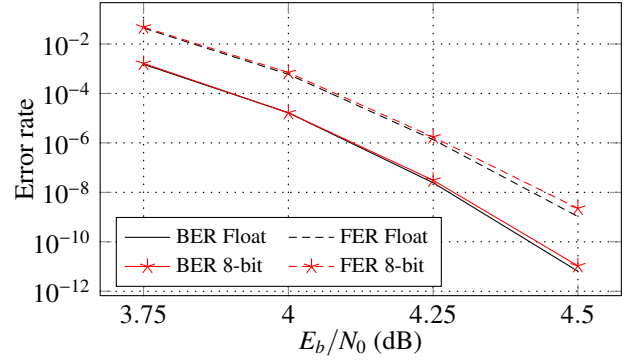

Fig. 2: Effect of quantization on error-correction performance of a $(32768, 27568)$ polar code.

4. THE FAST-SSC ALGORITHM ON AN X86 CPU

In software-defined-radio (SDR) applications, general purpose x86 CPUs are often used to carry out most of the signal processing (e.g. the Intel Core i7-2600 processor in the 2013 DARPA Spectrum Challenge). The same CPU is used to evaluate the performance of our software polar decoders based on the Fast-SSC algorithm in this work. This general-purpose x86 CPU provides support for the 128-bit Streaming SIMD Extensions (SSE) and the 256-bit Advanced Vector Extensions (AVX) and consists of four cores clocked at 3.4 GHz.

We present the results for three C++ implementations of the polar decoder that share the same memory management code, but differ in how the computational parts are implemented: The first implementation—referred to as Float in this work—uses single-precision floating-point values without any explicit attempts at vectorization. This decoder sets the baseline for the throughput comparison in Section 5. The second version, denoted SIMD-Float, uses the Vc C++ library [12] to perform vectorization. This enables the decoder to utilize the AVX instructions on the target platform and fall back to SSE instructions where AVX is not supported. The third decoder uses 8-bit signed integer data and explicitly uses the 8-bit integer operations provided by the SSE extensions by means of the compiler-provided SSE intrinsics. Support for integer operations in AVX instructions is not available prior to AVX2. AVX2 is not available on the targeted i7-2600 CPU. This decoder is referred to as SIMD-int8.

4.1. Quantization

It was shown in [9] that, for some codes of length 32768, using 7 bits of quantization for LLRs results in a negligible degradation of error-correction performance over a floating-point representation. Therefore we conclude that the 8-bit signed integer type used in the SIMD-int8 decoder is sufficient to achieve good error-correction performance for these codes. Figure 2 confirms this assumption. At a frame-error rate (FER) of 10^{-8} the performance loss over floating-point is less than 0.025 dB.

4.2. Software Mapping to a SIMD Architecture

The vector instructions added with SSE, up to version 4.1, support logic and arithmetic operations on vectors containing either 4 single-precision floating-point numbers or 16 8-bit integers. Additionally, our decoder uses AVX instructions, when available, to operate simultaneously on 8 packed single-precision floating-point numbers. This section lists the operations that benefited the most from explicit vectorization.

$f(\lambda_a, \lambda_b)$: (2) is often executed on large vectors of LLRs to prepare values for other processing nodes. The min operation and the sign calculation and assignment can all be vectorized to increase speed.

$g(\lambda_a, \lambda_b, \hat{u}_j)$: This operations is also often executed on large vectors. In the Float and SIMD-Float decoders, we use $\hat{u}_j \in \{+1, -1\}$ instead of $\{0, 1\}$. As a result, (3) can be rewritten as

$$g(\lambda_a, \lambda_b, \hat{u}_j) = \lambda_a * \hat{u}_j + \lambda_b.$$

This removes the conditional and turns $g(\cdot)$ into a multiply-accumulate operation, which can be performed efficiently in a vectorized manner on modern CPUs. In the SIMD-int8 decoder, multiplications cannot be carried out on 8-bit integers. Thus, both possibilities of (3) are calculated and are blended together with a mask to build the result.

COMBINE: The COMBINE operation combines two estimated bit-vectors using an XOR operation when $\hat{u}_j \in \{0, 1\}$, or a multiplication when $\hat{u}_j \in \{+1, -1\}$. The former is used in the SIMD-int8 decoder and the latter in the SIMD-Float decoder.

SPC Codes: Locating the LLR with the minimum magnitude is accelerated using SIMD instructions.

5. EXPERIMENTAL RESULTS

5.1. Methodology

In this section, we compare the throughput, in information bits per second, of the proposed software polar decoders with that of the fastest software decoders in literature. When available, latency is also compared. The software was compiled using the C++ compiler from GCC 4.8.1 using the flags "-march=native -funroll-loops -Ofast". Additionally, auto-vectorization and link-time optimization were also enabled for all versions. The decoder is inserted in a digital communication chain to measure its performance. We use binary phase shift keying (BPSK) over an AWGN channel with random codewords.

The throughput is calculated using the time required to decode a frame averaged over 10 runs of 50,000 and 10,000 frames each for the $N = 2048$ and the $N > 2048$ codes, respectively. The time required to decode a frame, or latency, also includes the time required to copy a frame to decoder memory and copy back the estimated codeword, and is measured using the high precision clock provided by the Boost

Table 1: Throughput comparison of decoding polar codes of length $N = 32{,}768$ using Fast-SSC.

Code rate	Implementation	T/P (Mbps)		Latency (μs)
		Coded	Info	
0.84	Float	47.47	39.93	690
	SIMD-Float	147.01	123.68	223
	SIMD-int8	242.01	203.60	135
0.9	Float	54.04	48.64	606
	SIMD-Float	173.77	156.40	189
	SIMD-int8	252.06	226.86	130

Chrono library. Codeword generation, transmission over the channel, demodulation, and, for SIMD-int8, quantization are excluded from calculations. The copying of data to and from the decoder is included to facilitate future comparisons with non-CPU-based decoders.

5.2. Comparison of the Software Implementations

As shown in Table 1, the vectorized implementations are 3 to 5 times faster than the Float decoder for the $(32768, 27568)$ and $(32768, 29492)$ codes. Table 2 shows that for the shorter $N = 2048$ codes, the speedup factors are similar at approximately 3 for SIMD-float and greater than 4 for SIMD-int8.

5.3. Comparison with Software LDPC Decoders

Software LDPC decoders are presented in [13–17]. The decoders of [14, 15] target the Cell/BE multicore processor, an NVIDIA 8800 GTX GPU and an Intel x86 CPU, respectively. In [13], Falcão et al. cover more WiMAX LDPC codes with their implementation for the Cell/BE processor. Lastly, [16] and [17] are aimed at delivering very high throughput using modern GPUs. In all cases, these software LDPC decoders parallelize the decoding of multiple received frames whereas we parallelize the decoding of a single frame. If the proposed polar decoders use all four cores of the CPU to simultaneously decode 4 frames, the throughput approximately quadruples. However in typical SDR applications the remaining cores are used for other tasks such as demodulation.

We focus on the moderate length LDPC codes, with rates $1/2$ and $5/6$, from the WiMAX standard, which are used in [13–17]. We compare software LDPC and polar decoders at a similar error-correction performance for a given rate. Thus, the polar codes chosen for the comparison were selected accordingly. The $(2048, 1024)$ and $(2048, 1707)$ polar codes match the $(2304, 1152)$ and $(1248, 1040)$ LDPC codes, respectively, decoded using the min-sum algorithm with 10 iterations.

Fig. 3 shows the error-correction performance of these codes. It can be seen that the $(2048, 1707)$ polar code was about 0.05 dB away from the longest rate-$5/6$ code of [13].
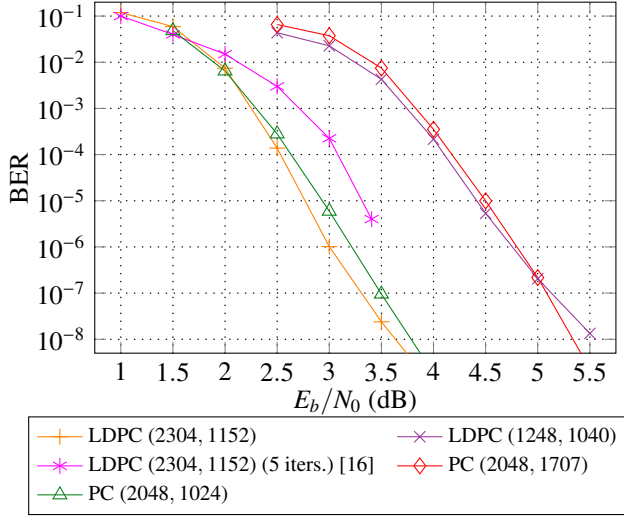
Fig. 3: Error-correction performance of polar codes compared with that of LDPC codes with the same rate and 10 iterations (5 where noted).

The error-correction performance of the $(2048, 1024)$ polar code was 0.2 dB worse than that of the $(2304, 1152)$ LDPC code when decoded with 10 iterations, but better than the LDPC decoder using only 5 iterations.

Table 2 shows the information throughput and latency corresponding to the proposed polar decoders in comparison with that of 10 decoding iterations of the software LDPC decoders of [13–15, 17]. The LDPC decoder of [16] is also shown but only uses 5 iterations. For the highest rate $5/6$ code, both of our proposed decoders using SIMD instructions have a better throughput than the software LDPC decoder even though we only use one CPU core. For the codes with rate $1/2$, we would need to use all four cores of our CPU in order to surpass the throughput of the fastest LDPC decoder on GPU.

As shown in Table 2, the latency of the SIMD polar decoders, at 10–14 μs, is an order of magnitude smaller than that of the GPU and Cell/BE implementations of [14, 15, 17]. This is due in part to the LDPC decoders buffering multiple frames, e.g. 16 for [14] and 50 for [17].

It should be noted that the implementations of [13–17] trade error-correction performance for throughput and that these LDPC codes can perform better when more iterations are used. For example, the use of 5 decoding iterations in [16] instead of 10 leads to an error-correction performance degradation of 0.5 dB, as shown in Fig. 3.

5.4. A Note About Hardware SC Polar Decoders

In terms of throughput, the proposed software decoders are competitive with all hardware decoders in literature, with the exception of the hardware implementation of the same (Fast-SSC) algorithm [9]. For example, despite using a smaller number of quantization bits, the semi-parallel polar decoder of [3] offers an inferior throughput for the $(2048, 1707)$

Table 2: Comparison with software LDPC decoders for codes of rates 1/2 and 5/6 and 10 iterations (5 where noted).

Decoder	(N, k)	Target	Info T/P (Mbps)	Latency (μs)
LDPC [15]	$(1024, 512)$	x86 CPU	1.04	N/A
LDPC [14]	$(1024, 512)$	GPU	20.9	393
LDPC [15]	$(1024, 512)$	Cell/BE	34.8	354
LDPC [17]	$(2304, 1152)$	GPU	152.1	1266
LDPC [16] (5 iters.)	$(2304, 1152)$	GPU	355.0	N/A
PC Float	$(2048, 1024)$	x86 CPU	23.2	44
PC SIMD-Float			71.5	14
PC SIMD-int8			101.7	10
LDPC [13]	$(1248, 1040)$	Cell/BE	65.3	N/A
PC Float	$(2048, 1707)$	x86 CPU	45.4	38
PC SIMD-Float			154.1	12
PC SIMD-int8			198.7	9

and $(32768, 29462)$ codes, achieving only 69.2 Mbps and 27.6 Mbps respectively, whereas the SIMD-int8 decoder reaches 198.7 Mbps and 226.9 Mbps. The decoder is also faster than the hardware polar decoders of [5, 8]. In the case of the latter, the two phase SC decoder [8], the fastest non-Fast-SSC decoder in literature, has an information throughput of 102.6 Mbps for a $(16384, 14746)$ code. Our 8-bit SIMD decoder achieves 242.3 Mbps for the same code.

6. CONCLUSION

In this work, we presented fast software implementations of polar decoders. By exploiting SIMD extensions, our int8 implementation was able to achieve an information throughput greater than 200 Mbps using one 3.4 GHz x86 CPU core. Moreover, for codes with similar error-correction performance, the polar decoders achieved a greater throughput than LDPC decoders targeting x86 CPUs and the Cell/BE processor; and were competitive with the fastest software GPU-based LDPC decoders. In addition, these SIMD software decoders are faster than any hardware polar decoder with exception of the one implementing the same algorithm.

Finally, our initial experiments with an Intel Haswell processor core, featuring the AVX2 supplementary instructions, gave us an information throughput greater than 300 Mbps for a core clocked at 3.4 GHz by using the new instructions operating simultaneously over 32 8-bit integers packed in a 256-bit register. Hence, our results indicate that polar codes are promising candidates for software-defined-radio applications.

7. REFERENCES

[1] E. Arıkan, "Channel polarization: A method for constructing capacity-achieving codes," in *Inf. Theory (ISIT). IEEE Intl. Symp. on*, 2008, pp. 1173–1177.

[2] ——, "Channel polarization: A method for constructing capacity-achieving codes for symmetric binary-input memoryless channels," *IEEE Trans. Inf. Theory*, vol. 55, no. 7, pp. 3051–3073, 2009.

[3] C. Leroux, A. J. Raymond, G. Sarkis, and W. J. Gross, "A semi-parallel successive-cancellation decoder for polar codes," *IEEE Trans. Signal Process.*, vol. 61, no. 2, pp. 289–299, 2013.

[4] A. Mishra, A. J. Raymond, L. G. Amaru, G. Sarkis, C. Leroux, P. Meinerzhagen, A. Burg, and W. J. Gross, "A successive cancellation decoder ASIC for a 1024-bit polar code in 180nm CMOS," in *Proc. IEEE Asian Solid-State Circuits Conf. (A-SSCC)*, 2012.

[5] A. J. Raymond and W. J. Gross, "Scalable successive-cancellation hardware decoder for polar codes," in *Sign. and Inf. Proc. (GlobalSIP), 1st IEEE Glob. Conf.*, Dec 2013, pp. 1282–1285.

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

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

[8] A. Pamuk and E. Arıkan, "A two phase successive cancellation decoder architecture for polar codes," in *Inf. Theory (ISIT). IEEE Intl. Symp. on*, 2013.

[9] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, "Fast polar decoders: Algorithm and implementation," *IEEE J. Sel. Areas Commun.*, *to appear*. [Online]. Available: http://arxiv.org/abs/1307.7154

[10] I. Tal and A. Vardy, "How to construct polar codes," *CoRR*, vol. abs/1105.6164, 2011. [Online]. Available: http://arxiv.org/abs/1105.6164

[11] E. Arıkan, "Systematic polar coding," *IEEE Commun. Lett.*, vol. 15, no. 8, pp. 860–862, 2011.

[12] M. Kretz and V. Lindenstruth, "Vc: A C++ library for explicit vectorization," *Software: Practice and Experience*, vol. 42, no. 11, pp. 1409–1430, 2012.

[13] G. Falcão, V. Silva, J. Marinho, and L. Sousa, "LDPC decoders for the WiMAX (IEEE 802.16e) based on multicore architectures," in *WIMAX New Developments*. Upena D Dalal and Y P Kosta (Ed.), 2009.

[14] G. Falcão, V. Silva, and L. Sousa, "How GPUs can outperform ASICs for fast LDPC decoding," in *Supercomputing (ICS). ACM Proc. of the 23rd Intl. Conf. on*, 2009, pp. 390–399.

[15] G. Falcão, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 309–322, 2011.

[16] R. Li, J. Zhou, Y. Dou, S. Guo, D. Zou, and S. Wang, "A multi-standard efficient column-layered LDPC decoder for software defined radio on GPUs," in *Sign. Proc. Advances in Wireless Commun. (SPAWC), IEEE 14th Workshop on*, 2013, pp. 724–728.

[17] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *Sign. and Inf. Proc. (GlobalSIP), 1st IEEE Glob. Conf.*, Dec 2013, pp. 1258–1261.