J Sign Process Syst (2018) 90:761–775
DOI 10.1007/s11265-016-1157-y

CrossMark

Low-Latency Software Polar Decoders

Pascal Giard[1] · Gabi Sarkis[1] · Camille Leroux[2] · Claude Thibeault[3] · Warren J. Gross[1]

Received: 30 October 2015 / Revised: 18 February 2016 / Accepted: 28 June 2016 / Published online: 11 July 2016
© Springer Science+Business Media New York 2016

Abstract Polar codes are a new class of capacity-achieving error-correcting codes with low encoding and decoding complexity. Their low-complexity decoding algorithms rendering them attractive for use in software-defined radio applications where computational resources are limited. In this work, we present low-latency software polar decoders that exploit modern processor capabilities. We show how adapting the algorithm at various levels can lead to significant improvements in latency and throughput, yielding polar decoders that are suitable for high-performance software-defined radio applications on modern desktop processors and embedded-platform processors. These proposed decoders have an order of magnitude lower latency and memory footprint compared to state-of-the-art decoders, while maintaining comparable throughput. In addition, we present strategies and results for implementing polar decoders on graphical processing units. Finally, we show that the energy efficiency of the proposed decoders is comparable to state-of-the-art software polar decoders.

Keywords Polar codes · Successive-cancellation decoding · Software decoders

✉ Pascal Giard
pascal.giard@mail.mcgill.ca

Gabi Sarkis
gabi.sarkis@mail.mcgill.ca

Camille Leroux
camille.leroux@ims-bordeaux.fr

Claude Thibeault
claude.thibeault@etsmtl.ca

Warren J. Gross
warren.gross@mcgill.ca

[1] Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada

[2] IMS Lab, Bordeaux-INP, Bordeaux, France

[3] Department of Electrical Engineering, École de Technologie Supérieure, Montréal, Québec, Canada

1 Introduction

In software-defined radio (SDR) applications, researchers and engineers have yet to fully harness the error-correction capability of modern codes due to their high computational complexity. Many are still using classical codes [7, 23] as implementing low-latency high-throughput—exceeding 10 Mbps of information throughput—software decoders for turbo or low-density parity-check (LDPC) codes is very challenging. The irregular data access patterns featured in decoders of modern error-correction codes make efficient use of single-instruction multiple-data (SIMD) extensions present on today's central processing units (CPUs) difficult. To overcome this difficulty and still achieve a good throughput, software decoders resorting to inter-frame parallelism (decoding multiple independent frames at the same time) are often proposed [12, 25, 26]. Inter-frame parallelism comes at the cost of higher latency, as many frames have to be buffered before decoding can be started. Even with a split layer approach to LDPC decoding where intra-frame parallelism can be applied, the latency remains high at multiple milliseconds on a recent desktop processor [10]. This work presents software polar decoders that enable SDR systems to utilize powerful *and* fast error-correction.

Polar codes provably achieve the symmetric capacity of memoryless channels [5]. Moreover they are well suited for

software implementation, due to regular memory access patterns, on both x86 and embedded processors [9, 13, 14]. To achieve higher throughput and lower latency on processors, software polar decoders can also exploit SIMD vector extensions present on today's CPUs. Vectorization can be performed intra-frame [9] or inter-frame [13, 14], with the former having lower decoding latency as it does not require multiple frames to start decoding.

In this work, we explore intra-frame vectorized polar decoders. We propose architectures and optimization strategies that lead to the implementation of high-performance software polar decoders tailored to different processor architectures with decoding latency of 26 μs for a (32768, 29492) polar code, a significant performance improvement compared to that of our previous work [9]. We start Section 2 with a review of the construction and decoding of polar codes. We then present two different software decoder architectures with varying degrees of specialization in Section 3. Implementation and results on an embedded processor are discussed in Section 4. We also adapt the decoder to suit graphical processing units (GPUs), an interesting target for applications where many hundreds of frames have to be decoded simultaneously, and present the results in Section 5. Finally, Section 6 compares the energy consumption of the different decoders and Section 8 concludes the paper.

This paper builds upon the work published in [9] and [19]. It provides additional details on the approach as well as more experimental results for modern desktop processors. Both floating- and fixed-point implementations for the final desktop CPU version—the unrolled decoder—were further optimized leading to an information throughput of up to 1.4 Gbps. It also adds results for the adaptation of our strategies to an embedded processor leading to a throughput and latency of up to 2.25 and 36 times better, respectively, compared to that of the state-of-the-art software implementation. Compared to the state of the art, both the desktop and embedded processor implementations are shown to have one to two orders of magnitude smaller memory footprint. Lastly, strategies and results for implementing polar decoders on a graphical processing unit (GPU) are presented for the first time.

2 Polar Codes

2.1 Construction of Polar Codes

Polar codes exploit the channel polarization phenomenon to achieve the symmetric capacity of a memoryless channel as the code length increases ($N \to \infty$). A polarizing construction where $N = 2$ is shown in Fig. 1a. The probability of correctly estimating bit u_1 increases compared to when

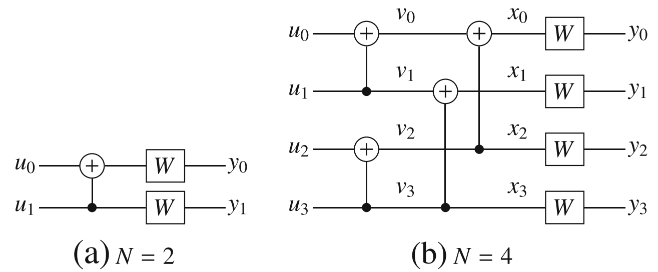


Figure 1 Construction of polar codes of lengths 2 and 4.

the bits are transmitted without any transformation over the channel W. Meanwhile, the probability of correctly estimating bit u_0 decreases. The polarizing transformation can be combined recursively to create longer codes, as shown in Fig. 1b for $N = 4$. As the $N \to \infty$, the probability of successfully estimating each bit approaches either 1 (perfectly reliable) or 0.5 (completely unreliable), and the proportion of reliable bits approaches the symmetric capacity of W [5].

To construct an (N, k) polar code, the $N - k$ least reliable bits, called the frozen bits, are set to zero and the remaining k bits are used to carry information. The frozen bits of an (8, 5) polar code are indicated in gray in Fig. 2a. The locations of the information and frozen bits are based on the type and conditions of W. In this work we use polar codes constructed according to [22]. The generator matrix, G_N, for a polar code of length N can be specified recursively so that $G_N = F_N = F_2^{\otimes \log_2 N}$, where $F_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$ and $^{\otimes}$ is the Kronecker power. For example, for $N = 4$, G_N is

$$G_4 = F_2^{\otimes 2} = \left[\begin{array}{c|c} F_2 & 0 \\ \hline F_2 & F_2 \end{array}\right] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}.$$

In [5], bit-reversed indexing is used, which changes the generator matrix by multiplying it with a bit-reversal operator B, so that $G = BF$. In this work, natural indexing is used as it yields more efficient software decoders [9].

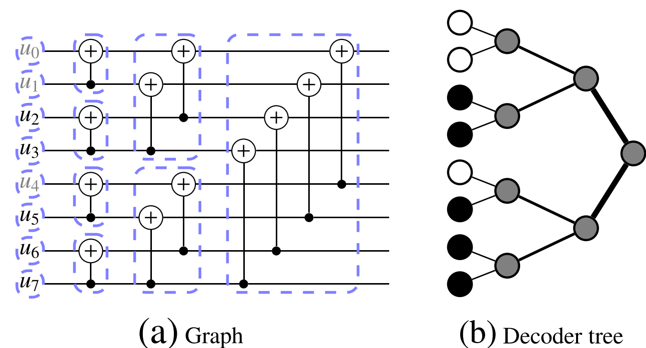


Figure 2 The graph and tree representation of an (8, 5) polar code.

2.2 Tree Representation of Polar Codes

A polar code of length N is the concatenation of two constituent polar codes of length $N/2$ [5]. Therefore, binary trees are a natural representation of polar codes [4]. Figure 2 illustrates the tree representation of an (8, 5) polar code. In Fig. 2a, the frozen bits are labeled in gray while the information bits are in black. The corresponding tree, shown in Fig. 2b, uses white and black leaf nodes to denote these bits, respectively. The gray nodes of Fig. 2b correspond to concatenation operations shown in Fig. 2a.

2.3 Successive-Cancellation Decoding

In successive-cancellation (SC) decoding, the decoder tree is traversed depth first, selecting left edges before backtracking to right ones, until the size-1 frozen and information leaf nodes. The messages passed to child nodes are log-likelihood ratios (LLRs); while those passed to parents are bit estimates. These messages are denoted α and β, respectively. Messages to a left child l are calculated by the f operation using the min-sum algorithm:

$$\begin{aligned}\alpha_l[i] &= f(\alpha_v[i], \alpha_v[i+N_v/2]) \\ &= \operatorname{sgn}(\alpha_v[i])\operatorname{sgn}(\alpha_v[i+N_v/2]) \\ &\quad \min(|\alpha_v[i]|, |\alpha_v[i+N_v/2]|),\end{aligned} \tag{1}$$

where N_v is the size of the corresponding constituent code and α_v the LLR input to the node.

Messages to a right child are calculated using the g operation

$$\begin{aligned}\alpha_r[i] &= g(\alpha_v[i], \alpha_v[i+N_v/2], \beta_l[i]) \\ &= \begin{cases}\alpha_v[i+N_v/2]+\alpha_v[i], & \text{when } \beta_l[i]=0; \\ \alpha_v[i+N_v/2]-\alpha_v[i], & \text{otherwise,}\end{cases}\end{aligned} \tag{2}$$

where β_l is the bit estimate from the left child.

Bit estimates at the leaf nodes are set to zero for frozen bits and are calculated by performing threshold detection for information ones. After a node has the bit estimates from both its children, they are combined to generate the node's estimate that is passed to its parent

$$\beta_v[i] = \begin{cases}\beta_l[i] \oplus \beta_r[i], & \text{when } i < N_v/2; \\ \beta_r[i-N_v/2], & \text{otherwise,}\end{cases} \tag{3}$$

where $\oplus$ is modulo-2 addition (XOR).

2.4 Simplified Successive-Cancellation Decoding

Instead of traversing a sub-tree whose leaves all correspond to frozen or information bits, simplified successive-cancellation (SSC) applies a decision rule immediately [4]. For frozen sub-trees, the output is set to the zero vector; while for information sub-tree the maximum-likelihood (ML) output is obtained by performing element-wise threshold detection on the soft-information input vector, α_v. This shrinks the decoder, reducing the number of calculations and increasing decoding speed. The SC and SSC pruned tree corresponding to an (8, 5) polar code are shown in Fig. 3a and b, respectively.

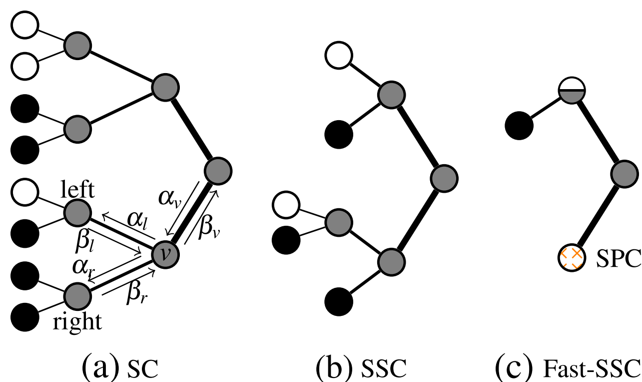


Figure 3 Decoder trees corresponding to the SC, SSC and Fast-SSC decoding algorithms.

2.5 The Fast-SSC Decoding Algorithm

The Fast-SSC decoding algorithm further prunes the decoder tree by applying low-complexity decoding rules when encountering certain types of constituent codes. These special cases are:

Repetition codes: are constituent codes where only the last bit is an information bit. These codes are efficiently decoded by calculating the sum of the input LLRs and using threshold detection to determine the result that is then replicated to form the estimated bits:

$$\beta_v[i] = \begin{cases}0, & \text{when } \left(\sum_{i=0}^{N_v-1}\alpha_v[i]\right) \geq 0; \\ 1, & \text{otherwise,}\end{cases}$$

where N_v is the number of leaf nodes.

Single-parity-check (SPC) codes: are constituent codes where only the first bit is frozen. The corresponding node is indicated by the cross-hatched orange pattern in Fig. 3c. The first step in decoding these codes is to calculate the hard decision of each LLR

$$\beta_v[i] = \begin{cases}0, & \text{when } \alpha_v[i] \geq 0; \\ 1, & \text{otherwise,}\end{cases} \tag{4}$$

and then calculating the parity of these decisions

$$\text{parity} = \bigoplus_{i=0}^{N_v-1} \beta_v[i]$$

If the parity constraint is unsatisfied, the estimate of the bit with the smallest LLR magnitude is flipped:

$$\beta_v[i] = \beta_v[i] \oplus \text{parity, where } i = \arg\min_i(|\alpha_v[i]|).$$

Repetition-SPC codes: are codes whose left constituent code is a repetition code and the right an SPC one. They can be speculatively decoded in hardware by simultaneously decoding the repetition code and two instances of the SPC code: one assuming the output of the repetition code is all 0's and the other all 1's. The correct result is selected once the output of the repetition code is available. This speculative decoding also provides speed gains in software.

Figure 3c shows the tree corresponding to a Fast-SSC decoder is will be described more thoroughly in Section 3.2. Other types of operations are introduced in the Fast-SSC algorithm, we refer the reader to [20] for more details.

3 Implementation on x86 Processors

In this section we present two different versions of the decoder in terms of increasing design specialization for software; whereas the first version—the instruction-based decoder—takes advantage of the processor architecture it remains configurable at run time and the second one—the unrolled decoder—presents a fully unrolled, branchless decoder fully exploiting SIMD vectorization. In the second version of the decoder, compile-time optimization plays a significant role in the performance improvements. Performance is evaluated for both the instruction-based and unrolled decoders.

It should be noted that, contrary to what is common in hardware implementations e.g. [15, 20], natural indexing is used for all software decoder implementations. While bit-reversed indexing is well-suited for hardware decoders, SIMD instructions operate on independent vectors, not adjacent values within a vector. Using bit-reverse indexing would have mandated data shuffling operations before any vectorized operation is performed.

Both versions, instruction-based decoders and unrolled decoders, use the following functions from the Fast-SSC algorithm [20]: F, G, G_0R, Combine, Combine_0R, Repetition, 0SPC, RSPC, RepSPC and P_01. An Info function implementing (4) is also added.

Methodology for the Experimental Results We discuss throughput in information bits per second as well as latency. Our software was compiled using the C++ compiler from GCC 4.9 using the flags "`-march=native -funroll-loops -Ofast`". Additionally, auto-vectorization is always kept enabled. The decoders are inserted in a digital communication chain to measure their speed and to ensure that optimizations, including those introduced by `-Ofast`, do not affect error-correction performance. In the simulations, 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 10000 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. Time is measured using the high precision clock provided by the Boost Chrono library.

In this work we focus on decoders running on one processor core only since the targeted application is SDR. Typically, an SDR system cannot afford to dedicate more than a single core to error-correction as it has to perform other functions simultaneously. For example, in SDR implementations of long term evolution (LTE) receivers, the orthogonal frequency-division multiplexing (OFDM) demodulation alone is approximately an order of magnitude more computationally demanding than the error-correction decoder [6, 7, 23].

3.1 Instruction-based Decoder

The Fast-SSC decoder implemented on a field-programmable gate array (FPGA) in [20] closely resembles a CPU with wide SIMD vector units and wide data buses. Therefore, it was natural to use the same design for a software decoder, leveraging SIMD instructions. This section describes how the algorithm was adapted for a software implementation. As fixed-point arithmetic can be used, the effect of quantization is shown.

3.1.1 Using Fixed-Point Numbers

On processors, fixed-point numbers are represented with at least 8 bits. As illustrated in Fig. 4, using 8 bits of quantization for LLRs results in a negligible degradation of error-correction performance over a floating-point representation. At a frame-error rate (FER) of 10^{-8} the performance loss compared to a floating-point implementation is less than 0.025 dB for the (32768, 27568) polar code. With custom hardware, it was shown in [20] that 6 bits are sufficient for that polar code. It should be noted that in Fast-SSC decoding, only the G function adds to the amplitude of LLRs and it is carried out with saturating adders.

With instructions that can work on registers of packed 8-bit integers, the SIMD extensions available on most general-purpose x86 and ARM processors are a good fit to implement a polar decoder.

3.1.2 Vectorizing the Decoding of Constituent Codes

On x86-64 processors, the vector instructions added with SSE support logic and arithmetic operations on vectors

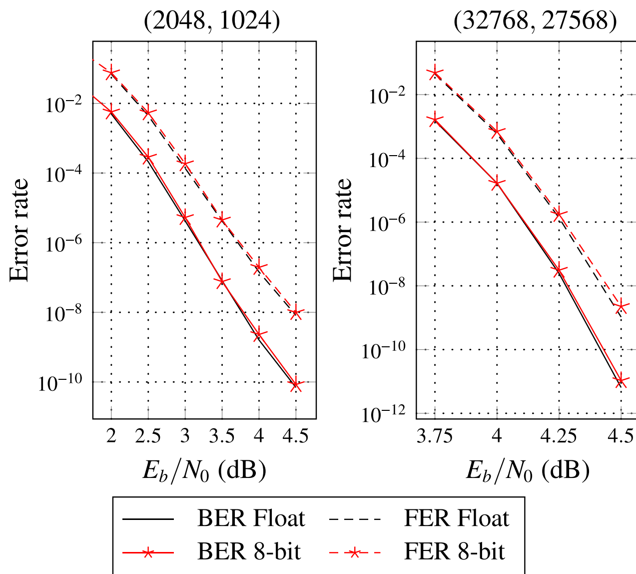


Figure 4 Effect of quantization on error-correction performance.

containing either 4 single-precision floating-point numbers or 16 8-bit integers. Additionally, x86-64 processors with AVX instructions can operate on data sets of twice that size. Below are the operations benefiting the most from explicit vectorization.

F: the f operation (1) is often executed on large vectors of LLRs to prepare values for other processing nodes. The min() operation and the sign calculation and assignment are all vectorized.

G and *G_0R*: the g operation is also frequently executed on large vectors. Both possibilities, the sum and the difference, of Eq. 2 are calculated and are blended together with a mask to build the result. The G_0R operation replaces the G operation when the left hand side of the tree is the all-zero vector.

Combine and *Combine_0R*: the Combine operation combines two estimated bit-vectors using an XOR operation in a vectorized manner. The Combine_0R operation is to Combine what G_0R is to G.

SPC decoding: locating the LLR with the minimum magnitude is accelerated using SIMD instructions.

3.1.3 Data Representation

For the decoders using floating-point numbers, the representation of β is changed to accelerate the execution of the g operation on large vectors. Thus, when floating-point LLRs are used, $\beta_l[i] \in \{+1, -1\}$ instead of $\{0, 1\}$. As a result, Eq. 2 can be rewritten as

$$g(\alpha_v[i], \alpha_v[i+N_v/2], \beta_l[i]) = \alpha_v[i] * \beta_l[i] + \alpha_v[i+N_v/2].$$

This removes the conditional assignment and turns $g()$ into a multiply-accumulate operation, which can be performed efficiently in a vectorized manner on modern CPUs. For integer LLRs, multiplications cannot be carried out on 8-bit integers. Thus, both possibilities of Eq. 2 are calculated and are blended together with a mask to build the result. The Combine operation is modified accordingly for the floating-point decoder and is computed using a multiplication with $\beta_l[i] \in \{+1, -1\}$.

3.1.4 Architecture-specific Optimizations

The decoders take advantage of the SSSE 3, SSE 4.1 and AVX instructions when available. Notably, the `sign` and `abs` instructions from SSSE 3 and the `blendv` instruction from SSE 4.1 are used. AVX, with instructions operating on vectors of 256 bits instead of the 128 bits, is only used for the floating-point implementation since it does not support integer operations. Data was aligned to the 128 (SSE) or 256-bit (AVX) boundaries for faster accesses.

3.1.5 Implementation Comparison

Here we compare the performance of three implementations. First, a non-explicitly vectorized version using floating-point numbers. Second an explicitly vectorized version using floating-point numbers. Third, the explicitly vectorized version using a fixed-point number representation. In Table 1, they are denoted as Float, SIMD-Float and SIMD-int8 respectively.

Results for decoders using the floating-point number representation are included as the efficient implementation makes the resulting throughput high enough for some

Table 1 Decoding polar codes with the instruction-based decoder.

Code (N, k)	Implementation	Info T/P (Mbps)	Latency (μs)
(2048, 1024)	Float	20.8	49
	SIMD-Float	75.6	14
	SIMD-int8	121.7	8
(2048, 1707)	Float	41.5	41
	SIMD-Float	173.9	10
	SIMD-int8	209.9	8
(32768, 27568)	Float	32.4	825
	SIMD-Float	124.3	222
	SIMD-int8	175.1	157
(32768, 29492)	Float	40.8	723
	SIMD-Float	160.1	184
	SIMD-int8	198.6	149

applications. The decoders ran on a single core of an Intel Core i7-4770S clocked at 3.1 GHz with Turbo disabled.

Comparing the throughput and latency of the Float and SIMD-Float implementations in Table 1 confirms the benefits of explicit vectorization in this decoder. The performance of the SIMD-Float implementation is only 21 % to 38 % slower than the SIMD-int8 implementation. This is not a surprising result considering that the SIMD-Float implementation uses the AVX instructions operating on vectors of 256 bits while the SIMD-int8 version is limited to vectors of 128 bits. Table 1 also shows that vectorized implementations have 3.6 to 5.8 times lower latency than the floating-point decoder.

3.2 Unrolled Decoder

The goal of this design is to increase vectorization and inlining and reduce branches in the resulting decoder by maximizing the information specified at compile-time. It also gets rid of the indirections that were required to get good performance out of the instruction-based decoder.

3.2.1 Generating an Unrolled Decoder

The polar codes decoded by the instruction-based decoders presented in Section 3.1 can be specified at run-time. This flexibility comes at the cost of increased branches in the code due to conditionals, indirections and loops. Creating a decoder dedicated to only one polar code enables the generation of a branchless fully-unrolled decoder. In other words, knowing in advance the dimensions of the polar code and the frozen bit locations removes the need for most of the control logic and eliminates branches there.

A tool was built to generate a list of function calls corresponding to the decoder tree traversal. It was first described in [19] and has been significantly improved since its initial publication notably to add support for other node types as well as to add support for GPU code generation. Listing 1 shows an example decoder that corresponds to the (8, 5) polar code whose dataflow graph is shown in Fig. 5. For brevity and clarity, in Fig. 5b, I and C_0R correspond to the Info and Combine_0R functions, respectively.

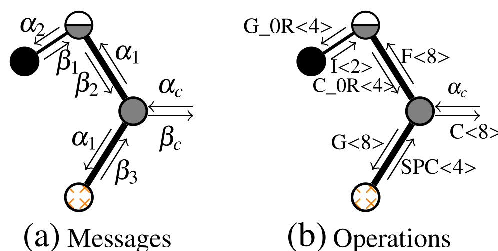


Figure 5 Dataflow graph of a (8, 5) polar decoder.

Listing 1 Unrolled (8, 5) Fast-SSC Decoder

```
F<8>(αc, α1);
G_0R<4>(α1, α2);
Info<2>(α2, β1);
Combine_0R<4>(β1, β2);
G<8>(αc, α2, β2);
SPC<4>(α2, β3);
Combine<8>(β2, β3, βc);
```

3.2.2 Eliminating Superfluous Operations on β-Values

Every non-leaf node in the decoder performs the combine operation (3), rendering it the most common operation. In Eq. 3, half the β values are copied unchanged to β_v. One method to significantly reduce decoding latency is to eliminate those superfluous copy operations by choosing an appropriate layout for β values in memory: Only N β values are stored in a contiguous array aligned to the SIMD vector size. When a combine operation is performed, only those values corresponding to β_l will be updated. Since the stage sizes are all powers of two, stages of sizes equal to or larger than the SIMD vector size will be implicitly aligned so that operations on them are vectorized.

3.2.3 Improved Layout of the α-memory

Unlike in the case of β values, the operations producing α values, f and g operations, do not copy data unchanged. Therefore, it is important to maximize the number of vectorized operations to increase decoding speed. To this end, contiguous memory is allocated for the $\log_2 N$ stages of the decoder. The overall memory and each stage is aligned to 16 or 32-byte boundaries when SSE or AVX instructions are used, respectively. As such, it becomes possible to also vectorize stages smaller than the SIMD vector size. The memory overhead due to not tightly packing the stages of α memory is negligible. As an example, for an $N = 32,768$ floating-point polar decoder using AVX instructions, the size of the α memory required by the proposed scheme is 262,208 bytes, including a 68-byte overhead.

3.2.4 Compile-Time Specialization

Since the sizes of the constituent codes are known at compile time, they are provided as template parameters to the functions as illustrated in Listing 1. Each function has two or three implementations. One is for stages smaller than the SIMD vector width where vectorization is not possible or straightforward. A second one is for stages that are equal or wider than the largest vectorization instruction set available. Finally, a third one provides SSE vectorization in an AVX or AVX2 decoder for stages that can be vectorized by the former, but are too small to be vectorized using AVX or

AVX2. The last specialization was noted to improve decoding speed in spite of the switch between the two SIMD extension types.

Furthermore, since the bounds of loops are compile-time constants, the compiler is able to unroll loops where it sees fit, eliminating the remaining branches in the decoder unless they help in increasing speed by resulting in a smaller executable.

3.2.5 Architecture-Specific Optimizations

First, the decoder was updated to take advantage of AVX2 instructions when available. These new instructions benefit the fixed-point implementation as they allow simultaneous operations on 32 8-bit integers.

Second, the implementation of some nodes were hand-optimized to better take advantage of the processor architecture. For example, the SPC node was mostly rewritten. Listing 2 shows a small but critical subsection of the SPC node calculations where the index within a SIMD vector corresponding to the specified value is returned. The reduction operation required by the Repetition node has also been optimized manually.

Listing 2 Finding the index of a given value in a vector

```
std::uint32_t findIdx(α* x, α x_min) {
    __mm256 minVec = _mm256_broadcastb_epi8(x_min);
    __mm256 mask = _mm256_cmpeq_epi8(minVec, x);
    std::uint32_t mvMask = _mm256_movemask_epi8(mask);
    return __tzcnt_u32(mvMask);
}
```

Third, for the floating-point implementation, β was changed to be in $\{+0, -0\}$ instead of $\{+1, -1\}$. In the floating-point representation [2], the most significant bit only carries the information about the sign. Flipping this bit effectively changes the sign of the number. By changing the mapping for β, multiplications are replaced by faster bitwise XOR operations. Similarly, for the 8-bit fixed-point implementation, β was changed to be in $\{0, -128\}$ to reduce the complexity of the Info and G functions.

Listings 3 and 4 show the resulting G functions for both the floating-point and fixed-point implementations as examples illustrating bottom-up optimizations used in our decoders.

Listing 3 Vectorized floating-point G function (*g* operation)

```
template<unsigned int N_v>
void G(α* α_in, α* α_out, β* β_in) {
    for (unsigned int i = 0; i < N_v/2; i += 8) {
        __m256 α_l = _mm256_load_ps(α_in + i);
        __m256 α_r = _mm256_load_ps(α_in + i + N_v/2);
        __m256 β_v = _mm256_load_ps(β_in + i);
        __m256 α'_l = _mm256_xor_ps(β_v, α_l);
        __m256 α_v = _mm256_add_ps(α_r, α'_l);
        __mm256_store_ps(α_out + i, α_v);
    }
}
```

Listing 4 Vectorized 8-bit fixed-point G function (*g* operation)

```
static const __m256i ONE = _mm256_set1_epi8(1);
static const __m256i M127 = _mm256_set1_epi8(−127);

template<unsigned int N_v>
void G(α* α_in, α* α_out, β* β_in) {
    for (unsigned int i = 0; i < N_v/2; i += 32) {
        __m256i α_l = _mm256_load_si256(α_in + i);
        __m256i α_r = _mm256_load_si256(α_in + i + N_v/2);
        __m256i β_v = _mm256_load_si256(β_in + i);
        __m256i β'_v = _mm256_or_si256(β_v, ONE);
        __m256i α'_l = _mm256_sign_epi8(α_l, β'_v);
        __m256i α_v = _mm256_add_ps(α_r, α'_l);
        __m256i α'_v = _mm256_max_epi8(M127, α_v);
        __mm256_store_si256(α_out + i, α'_v);
    }
}
```

3.2.6 Memory Footprint

The memory footprint is considered an important constraint for software applications. Our proposed implementations use 2 contiguous memory blocks that correspond to the α and β values, respectively. The size of the β-memory is

$$M_\beta = N W_\beta, \tag{5}$$

where N is the frame length, W_β is the number of bits used to store a β value and M_β is in bits.

The size of the α-memory can be expressed as

$$M_\alpha = \left[(2N-1) + A \log_2 A - \left(\sum_{i=0}^{\log_2(A)-1} 2^i \right) \right] W_\alpha, \tag{6}$$

where N is the frame length, W_α is the number of bits used to store an α value, A is the number of α values per SIMD vector and M_α is in bits. Note that the expression of M_α contains the expression for the overhead $M_{\alpha\mathrm{OH}}$ due to tightly packing the α values as described in Section 3.2.3:

$$M_{\alpha\mathrm{OH}} = \left[A \log_2 A - \left(\sum_{i=0}^{\log_2(A)-1} 2^i \right) \right] W_\alpha. \tag{7}$$

The memory footprint can thus be expressed as

$$\begin{aligned} M_{\text{total}} &= M_\beta + M_\alpha \\ &= N W_\beta + \left[(2N-1) + A \log_2 A - \left(\sum_{i=0}^{\log_2(A)-1} 2^i \right) \right] W_\alpha. \end{aligned} \tag{8}$$

The memory footprint in kilobytes can be approximated with

$$M_{\text{total (kbytes)}} \approx \frac{N(W_\beta + 2W_\alpha)}{8000}. \tag{9}$$

Table 2 Decoding polar codes with floating-point precision using SIMD, comparing the instruction-based decoder (ID) with the unrolled decoder (UD).

Code (N, k)	Info T/P (Mbps)		Latency (μs)	
	ID	UD	ID	UD
(2048, 1024)	75.6	229.8	14	4
(2048, 1707)	173.9	492.2	10	3
(32768, 27568)	124.3	271.3	222	102
(32768, 29492)	160.1	315.1	184	94

3.2.7 Implementation Comparison

We first compare the SIMD-float results for this implementation—the unrolled decoder—with those from Section 3.1—the instruction-based decoder. Then we show SIMD-int8 results and compare them with that of the software decoder of Le Gal et. al [14]. As in the previous sections, the results are for an Intel Core i7-4770S running at 3.1 GHz when Turbo is disabled and at up to 3.9 GHz otherwise. The decoders were limited to a single CPU core.

Table 2 shows the impact of the optimizations introduced in the unrolled version on the SIMD-float implementations. It resulted in the unrolled decoders being 2 to 3 times faster than the flexible, instruction-based, ones. Comparing Tables 1 and 2 shows an improvement factor from 3.3 to 5.7 for the SIMD-int8 implementations. It should be noted that some of the improvements introduced in the unrolled decoders could be backported to the instruction-based decoders, and is considered for future work.

Compared to the software polar decoders of [14], Table 3 shows that our throughput is lower for short frames but can be comparable for long frames. However, latency is an order of magnitude lower for all code lengths. This is to be expected as the decoders of [14] do inter-frame parallelism i.e. parallelize the decoding of independent frames while we parallelize the decoding of a frame. The memory footprint of our decoder is shown to be approximately 24 times lower than that of [14]. The results in [14] were presented with Turbo frequency boost enabled; therefore we present two sets of results for our proposed decoder: one with Turbo enabled, indicated by the asterisk (*) and the 3.1+ GHz frequency in the table, and one with Turbo disabled. The results with Turbo disabled are more indicative of a full SDR system as all CPU cores will be fully utilized, not leaving any thermal headroom to increase the frequency. The maximum Turbo frequencies are 3.8 GHz and 3.9 GHz for the i7-4960HQ and i7-4770S CPUs, respectively.

Looking at the first two, or last two rows of Table 2, it can be seen that for a fixed code length, the decoding latency is smaller for higher code rates. The tendency of decoding latency to decrease with increasing code rate and length was first discussed in [21]. It was noted that higher rate codes resulted in SSC decoder trees with fewer nodes and, therefore, lower latency. Increasing the code length was observed to have a similar, but lesser, effect. However, once the code becomes sufficiently long, the limited memory bandwidth and number of processing resources form bottlenecks that negate the speed gains.

The effects of unrolling and using the Fast-SSC algorithm instead of SC are illustrated in Table 4. It can be observed that unrolling the Fast-SSC decoder results in a 5 time decrease in latency. Using the Fast-SSC instead of SC decoding algorithm decreased the latency of the unrolled decoder by 3 times.

Table 3 Comparison of the proposed software decoder with that of [14].

Decoder	Target	L3 Cache	f (GHz)	Code (N, k)	Mem. footprint (kbytes)	Info T/P (Mbps)	Latency (μs)
[14]*	Intel Core i7-4960HQ	6MB	3.6+	(2048, 1024)	144	1,320	25
				(2048, 1707)	144	2,172	26
				(32768, 27568)	2304	1,232	714
				(32768, 29492)	2304	1,557	605
this work	Intel Core i7-4770S	8MB	3.1	(2048, 1024)	6	398	3
				(2048, 1707)	6	1,041	2
				(32768, 27568)	98	886	31
				(32768, 29492)	98	1,131	26
this work*	Intel Core i7-4770S	8MB	3.1+	(2048, 1024)	6	502	2
				(2048, 1707)	6	1,293	1
				(32768, 27568)	98	1,104	25
				(32768, 29492)	98	1,412	21

*Results with Turbo enabled.

Table 4 Effect of unrolling and algorithm choice on decoding speed of the (2048, 1707) code on the Intel Core i7-4770S.

Decoder	Info T/P (Mbps)	Latency (μs)
ID	210	8.1
UD SC	363	4.7
UD Fast-SSC	1041	1.6

4 Implementation on Embedded Processors

Many of the current embedded processors used in SDR applications also offer SIMD extensions, e.g. NEON for ARM processors. All the strategies used to develop an efficient x86 implementation can be applied to the ARM architecture with changes to accommodate differences in extensions. For example, on ARM, there is no equivalent to the `movemask` SSE/AVX x86 instruction.

The equations for the memory footprint provided in Section 3.2.6 also apply to our decoder implementation for embedded processors.

Comparison with Similar Works Results were obtained using the ODROID-U3 board, which features a Samsung Exynos 4412 system on chip (SoC) implementing an ARM Cortex A9 clocked at 1.7 GHz. Like in the previous sections, the decoders were only allowed to use one core. Table 5 shows the results for the proposed unrolled decoders and provides a comparison with [13]. As with their desktop CPU implementation of [14], inter-frame parallelism is used in the latter.

It can be seen that the proposed implementations provide better latency and greater throughput at native frequencies. Since the ARM CPU in the Samsung Exynos 4412 is clocked at 1.7 GHz while that in the NVIDIA Tegra 3 used in [13] is clocked at 1.4 GHz, we also provide linearly scaled throughput and latency numbers for the latter work, indicated by an asterisk (*) in the table. Compared to the scaled results of [13], the proposed decoder has 1.4–2.25 times the throughput and its latency is 25–36 times lower. The memory footprint of our proposed decoder is approximately 12 times lower than that of [13]. Both implementations are using 8-bit fixed-point values.

Table 5 Decoding polar codes with 8-bit fixed-point numbers on an ARM Cortex A9 using NEON.

Code (N, k)	Decoder	Mem. Footprint (kBytes)	T/P (Mbps) Coded	T/P (Mbps) Info	Latency (μs)
(1024, 512)	[13]	38	70.5	35.3	232
	[13]*	38	80.6	42.9	191
	this work	3	113.1	56.6	9
(32768, 29492)	[13]	1,216	33.1	29.8	15,844
	[13]*	1,216	40.2	36.2	13,048
	this work	98	90.8	81.7	361

*Results linearly scaled for the clock frequency difference.

5 Implementation on Graphical Processing Units

Most recent graphical processing units (GPU) have the capability to do calculations that are not related to graphics. These GPUs are often called general purpose GPUs (GPGPU). In this section, we describe our approach to implement software polar decoders in CUDA C [18] and present results for these decoders running on a NVIDIA Tesla K20c.

Most of the optimization strategies cited above could be applied or adapted to the GPU. However, there are noteworthy differences. Note that, when latency is mentioned below we refer to the decoding latency including the delay required to copy the data in and out of the GPU.

5.1 Overview of the GPU Architecture and Terminology

A NVIDIA GPU has multiple microprocessors with 32 cores each. Cores within the same microprocessor may communicate and share a local memory. However, synchronized communication between cores located in different microprocessors often has to go through the CPU and is thus costly and discouraged [8].

GPUs expose a different parallel programming model than general purpose processors. Instead of SIMD, the GPU model is single-instruction-multiple-threads (SIMT). Each core is capable of running a thread. A computational kernel performing a specific task is instantiated as a block. Each block is mapped to a microprocessor and is assigned one thread or more.

As it will be shown in Section 5.3, the latency induced by transferring data in and out of a GPU is high. To minimize decoding latency and maximize throughput, a combination of intra- and inter-frame parallelism is used for the GPU contrary to the CPUs where only the former was applied. We implemented a kernel that decodes a single frame. Thus, a block corresponds to a frame and attributing e.g. 10 blocks to a kernel translates into the decoding of 10 frames in parallel.

5.2 Choosing an Appropriate Number of Threads per Block

As stated above, a block can only be executed on one microprocessor but can be assigned many threads. However, when more than 32 threads are assigned to a block, the threads

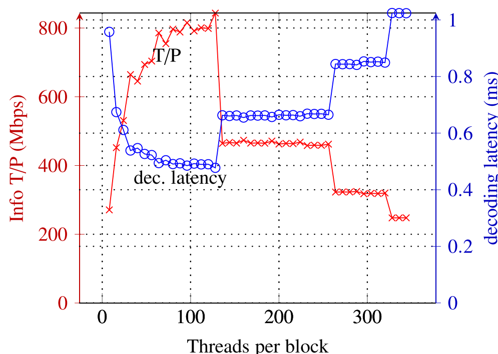


Figure 6 Effect of the number of threads per block on the information throughput and decoding latency for a (1024, 922) polar code where the number of blocks per kernel is 208.

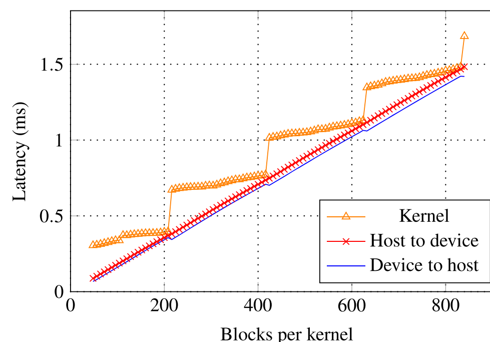


Figure 7 Effect of the number of blocks per kernel on the data transfer and kernel execution latencies for a (2048, 1707) polar code where the number of threads per block is 128.

starting at 33 are queued for execution. Queued threads are executed as soon as a core is free.

Figure 6 shows that increasing the number of threads assigned to a block is beneficial only until a certain point is reached. For the particular case of a (1024, 922) code, associating more than 128 threads to a block negatively affects performance. This is not surprising as the average node width for that code is low at 52.

5.3 Choosing an Appropriate Number of Blocks per Kernel

Memory transfers from the host to the GPU device are of high throughput but initiating them induces a great latency. The same is also true for transfers in the other direction, from the device to the host. Thus, the number of distinct transfers have to be minimized. The easiest way to do so is to run a kernel on multiple blocks. For our application, it translates to decoding multiple frames in parallel as a kernel decodes one frame.

Yet, there s a limit to the number of resources that can be used to execute a kernel i.e. decode a frame. At some point, there will not be enough computing resources to do the work in one pass and many passes will be required. The NVIDIA Tesla K20c card features the Kepler GK110 GPU that has 13 microprocessors with 32 cores and 16 load and store units each [16]. In total, 416 arithmetic or logic operations and 208 load or store operations can occur simultaneously.

Yet, there is a limit to the number of resources that can be used to execute a kernel i.e. decode a frame. At some point, there will not be enough computing resources to do the work in one pass and many passes will be required. The NVIDIA Tesla K20c card features the Kepler GK110 GPU that has 13 microprocessors with 32 cores and 16 load and store units each [16]. In total, 416 arithmetic or logic operations and 208 load or store operations can occur simultaneously.

Figure 7 shows the latency to execute a kernel, to transfer memory from the host to the GPU and vice versa for a given number of blocks per kernel. The number of threads assigned per block is fixed to 128 and the decoder is built for a (2048, 1707) polar code. It can be seen that the latency of memory transfers grows linearly with the number of blocks per kernel. The kernel latency however has local minimums at multiples of 208. We conclude that the minimal decoding latency, the sum of all three latencies illustrated in Fig. 7, is bounded by the number of load and store units.

5.4 On the Constituent Codes Implemented

Not all the constituent codes supported by the general purpose processors are beneficial to a GPU implementation. In a SIMT model, reduction operations are costly. Moreover, if a conditional execution leads to unbalanced threads, performance suffers. Consequently, all nodes based on the single-parity-check (SPC) codes, that features both characteristics, are not used in the GPU implementation.

Experiments have shown that implementing the SPC node results in a throughput reduction by a factor of 2 or more.

5.5 Shared Memory and Memory Coalescing

Each microprocessor contains shared memory that can be used by all threads in the same block. The NVIDIA Tesla K20c has 48 kB of shared memory per block. Individual reads and writes to the shared memory are much faster than accessing the global memory. Thus, intuitively, when conducting the calculations within a kernel, it seems preferable to use the shared memory as much as possible in place of the global memory.

However, as shown by Fig. 8, it is not always the case. When the number of blocks per kernel is small, using the shared memory provides a significant speedup. In fact, with 64 blocks per kernel, using shared memory results in a decoder that has more than twice the throughput compared to a kernel that only uses the global memory. Past a certain

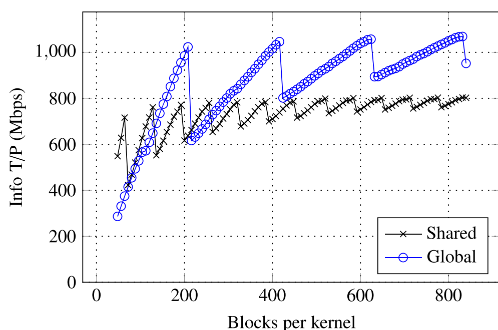


Figure 8 Information throughput comparison for a (1024, 922) polar code where intermediate results are stored in shared or global memory. The number of threads per block is 128.

value of blocks per kernel though, solely using the global memory is clearly advantageous for our application.

These results suggest that the GPU is able to efficiently schedule memory transfers when the number of blocks per kernel is sufficiently high.

5.6 Asynchronous Memory Transfers and Multiple Streams

Transferring memory from the host to the device and vice versa induces a latency that can be equal to the execution of a kernel. Fortunately, that latency can be first reduced by allocating pinned or page-locked host memory. As page-locked memory can be mapped into the address space of the device, the need for a staging memory is eliminated [18].

More significantly, NVIDIA GPUs with compute capability of 2.0 or above are able to transfer memory in and out of the device asynchronously. By creating three streams—sequences of operations that get executed in issue-order on the GPU—memory transfers and execution of the kernel can be overlapped, effectively multiplying throughput by a factor of 3.

This also increases the memory footprint by a factor of three. On the GPU, the memory footprint is

$$M_{\text{total (kbytes)}} = \frac{N(W_\beta + W_\alpha)BS}{8000}, \quad (10)$$

where B is the number of blocks per kernel—i.e. the number of frames being decoded simultaneously—, S is the number of streams, and where W_β and W_α are the number of bits required to store a β and an α value, respectively. For best performance, as detailed in the next section, both β and α values are represented with floating-point values and thus $W_\beta = W_\alpha = 32$.

5.7 On the Use of Fixed-Point Numbers on a GPU

It is tempting to move calculations to 8-bit fixed-point numbers in order to speedup performance, just like we did with the other processors. However, GPUs are not optimized for calculations with integers. Current GPUs only support 32-bit integers. Even so, the maximum number of operations per clock cycle per multiprocessor as documented by NVIDIA [18] clearly shows that integers are third class citizens behind single- and double-precision floating-point numbers. As an example, Table 2 of [18] shows that GPUs with compute capability 3.5—like the Tesla K20c—can execute twice as many double-precision floating-point multiplications in a given time than it can with 32-bit integers. The same GPU can carry on 6 times more floating-point precision multiplications than its 32-bit integer counterpart.

5.8 Results

Table 6 shows the estimated information throughput and measured latency obtained by decoding various polar codes on a GPU. The throughput is estimated by assuming that the total memory transfer latencies are twice the latency of the decoding. This has been verified to be a reasonable assumption, using NVIDIA's profiler tool, when the number of blocks maximizes throughput.

Performing linear regression on the results of Table 6 indicates that the latency scales linearly with the number of blocks, leading to standard error values of 0.04, 0.04 and 0.14 for the (1024, 922), (2048, 1707) and (4096, 3686) polar codes, respectively. In our decoder, a block corresponds to the decoding a single frame. The frames are independent of each other, and so are blocks. Thus, our decoder scales well with the number of available cores.

Furthermore, looking at Table 6 it can be seen that the information throughput is in the vicinity of a gigabit per second. Experiments have shown that the execution of two kernels can slightly overlap, making our throughput results of Table 6 worst-case estimations. For example,

Table 6 Decoding polar codes on an NVIDIA Tesla K20c.

Code (N, k)	Nbr of Blocks	Info T/P (Mbps)	Latency (ms)
(1024, 922)	208	1,022	0.6
	416	1,046	1.1
	624	1,060	1.6
	832	1,070	2.2
(2048, 1707)	208	915	1.1
	416	936	2.2
	624	953	3.3
	832	964	4.5
(4096, 3686)	208	959	2.6
	416	1,002	4.9
	624	1,026	6.9
	832	1,043	9.4

while the information throughput to decode 832 frames of a (4096, 3686) polar code is estimated at 1,043 Mbps in Table 6, the measured average value in NVIDIA's profiler tool was 1,228 Mbps, a 18 % improvement over the estimated throughput.

Our experiments have also shown that our decoders are bound by the data transfer speed that this test system is capable of. The PCIe 2.0 standard [1] specifies a peak data throughput of 64 Gbps when 16 lanes are used and once 8b10b encoding is accounted for. Decoding 832 frames of a polar code of length $N = 4096$ requires the transfer of 3,407,872 LLRs expressed as 32-bit floating-point numbers for a total of approximately 109 Mbits. Without doing any computation on the GPU, our benchmarks measured an average PCIe throughput of 45 Gbps to transfer blocks of data of that size from the host to the device and back. Running multiple streams and performing calculations on the GPU caused the PCIe throughput to drop to 40 Gbps. This corresponds to 1.25 Gbps when 32-bit floats are used to represent LLR inputs and estimated-bit outputs of the decoder. In light of these results, we conjecture that the coded throughput will remain approximately the same for any polar code as the PCIe link is saturated and data transfer is the bottleneck.

6 Energy Consumption Comparison

In this section the energy consumption is compared for all three processor types: the desktop processor, the embedded processor and the GPU. Unfortunately the Samsung Exynos 4412 SoC does not feature sensors allowing for power usage measurements of the ARM processor cores. The energy consumption of the ARM processor was estimated from board-level measurements. An Agilent E3631A DC power supply was used to provide the 5V input to the ODROID-U3 board and the current as reported by the power supply was used to calculated the power usage when the processor was idle and under load.

On recent Intel processors, power usage can be calculated by accessing the Running Average Power Limit (RAPL) counters. The LIKWID tool suite [24] is used to measure the power usage of the processor. Numbers are for the whole processor including the DRAM package. Recent NVIDIA GPUs also feature on-chip sensors enabling power usage measurement. Steady state values are read in real-time using the NVIDIA Management Libray (NVML) [17].

Table 7 compares the energy per information bit required to decode the (2048, 1707) polar code. The SIMD-int8 implementation of our unrolled decoder is compared with that of the implementation in [14]. The former uses an Intel Core i7-4770S clocked at 3.1 GHz. The latter uses an Intel Core i7-4960HQ clocked at 3.6 GHz with Turbo enabled. The results for the ARM Cortex A9 embedded processor and NVIDIA Tesla K20c GPU are also included for comparison. Note that the GPU represents LLRs with floating-point numbers.

The energy per information bit is calculated with

$$\mathrm{E}\ (J/\text{info. bit}) = \frac{\mathrm{P}\ (W)}{\text{info. T/P}\ (bits/s)}.$$

It can be seen that the proposed decoder is slightly more energy efficient on a desktop processor compared to that of [14]. For that polar code, the latter offers twice the throughput but at the cost of a latency that is at least 13 times greater. However, the latter is twice as fast for that polar code. Decoding on the embedded processor offers very similar energy efficiency compared to the Intel processor although the data throughput is an order of magnitude slower. However, decoding on a GPU is significantly less energy efficient than any of the decoders running on a desktop processor.

The power consumption on the embedded platform was measured to be fairly stable with only a 0.1 W difference between the decoding of polar codes of lengths 1024 or 32,768.

Table 7 Comparison of the power consumption and energy per information bit for the (2048, 1707) polar code.

Decoder	Target	Mem. Footprint (kbytes)	Info. T/P (Gbps)	Latency (μs)	Power (W)	Energy (nJ/info. bit)
[14]	Intel Core i7-4960HQ*	144	2.2	26	13	6
this work	Intel Core i7-4770S	6	1.0	2	3	3
	Intel Core i7-4770S*	6	1.3	1	5	4
	ARM Cortex A9	6	0.1	14	0.8	7
	NVIDIA Tesla K20c	3,408†	0.9	1100	108	118

*Results with Turbo enabled.

†Amount required per stream. Three streams are required to sustain this throughput.

7 Further Discussion

7.1 On the Relevance of the Instruction-Based Decoders

Some applications require excellent error-correction performance that necessitates the use of polar codes much longer than $N = 32,768$. For example, Quantum Key Distribution benefits from frames of 2^{21} to 2^{24} bits [11]. At such lengths, current compilers fail to compile an unrolled decoder. However, the instruction-based decoders are very suitable and are capable of throughput greater than 100 Mbps with a code of length 1 million.

7.2 On the relevance of software decoders in comparison to hardware decoders

The software decoders we have presented are good for systems that require moderate throughput without incurring the cost of dedicated hardware solutions. For example, in a software-defined radio communication chain based on USRP radios and the GNU Radio software framework, a forward error-correction (FEC) solution using our proposed decoders only consumes 5 % of the total execution time on the receiver. Thus, freeing FPGA resources to implement functions other than FEC, e.g. synchronization and demodulation.

7.3 Comparison with LDPC codes

LDPC codes are in widespread use in wireless communication systems. In this section, the error-correction performance of moderate-length polar codes is compared against that of standard LDPC codes [3]. Similarly, the performance of the state-of-the-art software LDPC decoders is compared against that of our proposed unrolled decoders for polar codes.

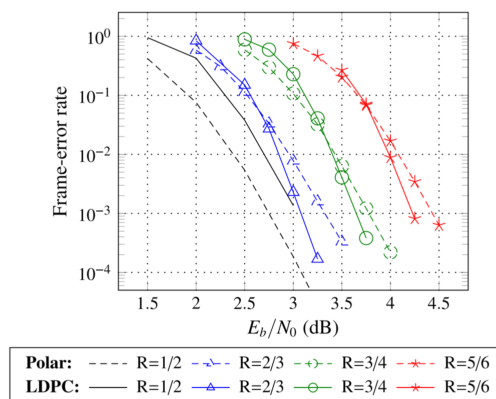


Figure 9 Error-correction performance of the polar codes of length 2048 compared with the LDPC codes of length 1944 from the 802.11n standard.

Table 8 Information throughput and latency of the polar decoders compared with the LDPC decoders of [10] when estimating 524,280 information bits on a Intel Core i7-2600.

Decoder	N	Rate	Latency		Info. T/P (Mbps)
			total (ms)	per frame (μs)	
[10]	1944	1/2	17.4	N/A	30.1
		2/3	12.7	N/A	41.0
		3/4	11.2	N/A	46.6
		5/6	9.3	N/A	56.4
this work	2048	1/2	2.0	3.83	267.4
		2/3	1.0	2.69	507.4
		3/4	0.8	2.48	619.4
		5/6	0.6	2.03	840.9

The fastest software LDPC decoders in literature are those of [10], which implements decoders for the 802.11n standard and present results for the Intel Core i7-2600 x86 processor. That wireless communication standard defines three code lengths: 1944, 1296, 648; and four code rates: 1/2, 2/3, 3/4, 5/6. In [10], LDPC decoders are implemented for all four codes rates with a code length of 1944. A layered offset-min-sum decoding algorithm with five iterations is used and early-termination is not supported.

Figure 9 shows the frame-error rate (FER) of these codes using 10 iterations of a flooding-schedule offset min-sum floating-point decoding algorithm which yields slightly better results than the five iteration layered algorithm used in [10]. The FER of polar codes with a slightly longer length of 2048 and matching code rates are also shown in Fig. 9.

Table 8 that provides the latency and information throughput for decoding 524,280 information bits using the state-of-the-art software LDPC decoders of [10] compared to our proposed polar decoders. To remain consistent with the result presented in [10], which used the Intel Core i7-2600 processor, the results in Table 8 use that processor as well.

While the polar code with rate 1/2 offers a better coding gain than its LDPC counterpart, all other polar codes in Fig. 9 are shown to suffer a coding loss close to 0.25 dB at a FER of 10^{-3}. However, as Table 8 shows, there is approximately an order of magnitude advantage for the proposed unrolled polar decoders in terms of both latency and throughput compared to the LDPC decoders of [10].

8 Conclusion

In this work, we presented low-latency software polar decoders adapted to different processor architectures. The decoding algorithm is adapted to exploit different SIMD instruction sets for the desktop and embedded processors

(SSE, AVX and NEON) or to the SIMT model inherent to GPU processors. The optimization strategies go beyond parallelisation with SIMD or SIMT. Most notably, we proposed to generate a branchless fully unrolled decoder, to use compile-time specialization, and adopt a bottom-up approach by adapting the decoding algorithm and data representation to features offered by processor architectures. For desktop processors, we have shown that intra-frame parallelism can be exploited to get a very low-latency while achieving information throughputs greater than 1 Gbps using a single core. For embedded processors, the principle remains but the achievable information throughputs are more modest at 80 Mbps. On the GPU we showed that inter-frame parallelism could be successfully used in addition to intra-frame parallelism to reach better speed, and the impact of two critical parameters on the performance of the decoders was explored. We showed that given the right set of parameters, GPU decoders are able to sustain an information throughput around 1 Gbps while simultaneously decoding hundreds of frames. Finally, we showed that the memory footprint of our proposed decoder is at least an order of magnitude lower than that our the state-of-the-art polar decoder while being slightly more energy efficient. These results indicate that the proposed software decoders make polar codes interesting candidates for software-defined radio applications.

Acknowledgments The authors wish to thank Samuel Gagné of École de technologie supérieure and CMC Microsystems for providing access to the Intel Core i7-4770S processor and NVIDIA Tesla K20c graphical processing unit, respectively. Claude Thibeault is a member of ReSMiQ. Warren J. Gross is a member of ReSMiQ and SYTACom.

References

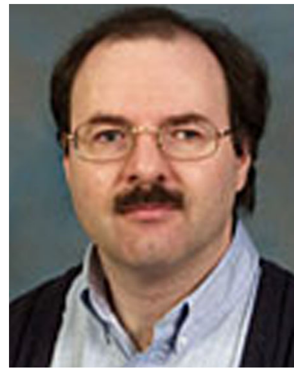
1. PCI (2006). Express base specification revision 2.0. PCI-SIG.
2. IEEE (2008). Standard for floating-point arithmetic. IEEE Std 754–2008 pp. 1–70. doi:10.1109/IEEESTD.2008.4610935.
3. IEEE (2012). Standard for information technology–telecommunications and information exchange between systems local and metropolitan area networks–specific requirements part 11: wireless LAN medium access control (MAC) and physical layer (PHY) specifications. *IEEE Std 802.11-2012 (Revision of IEEE Std 802.11–2007)*, 1–2793. doi:10.1109/IEEESTD.2012.6178212.
4. Alamdar-Yazdi, A., & Kschischang, F.R. (2011). A simplified successive-cancellation decoder for polar codes. *IEEE Communications Letters*, *15*(12), 1378–1380. doi:10.1109/LCOMM.2011.101811.111480.
5. Arıkan, E. (2009). Channel polarization: a method for constructing capacity-achieving codes for symmetric binary-input memoryless channels. *IEEE Transactions on Information Theory*, *55*(7), 3051–3073. doi:10.1109/TIT.2009.2021379.
6. Bang, S., Ahn, C., Jin, Y., Choi, S., Glossner, J., & Ahn, S. (2014). Implementation of LTE system on an SDR platform using CUDA and UHD. *Analog Integrated Circuits and Signal Processing*, *78*(3), 599–610. doi:10.1007/s10470-013-0229-1.
7. Demel, J., Koslowski, S., & Jondral, F. (2015). A LTE receiver framework using GNU Radio. *Journal of Signal Processing System*, *78*(3), 313–320. doi:10.1007/s11265-014-0959-z.
8. Feng, W.C., & Xiao, S. (2010). To GPU synchronize or not GPU synchronize?. In *IEEE international symposium on circuits and system. (ISCAS)* (pp. 3801–3804). doi:10.1109/ISCAS.2010.5537722.
9. Giard, P., Sarkis, G., Thibeault, C., & Gross, W.J. (2014). Fast software polar decoders. In *IEEE international conference on acoustic, speech, and signal process. (ICASSP)* (pp. 7555–7559). doi:10.1109/ICASSP.2014.6855069.
10. Han, X., Niu, K., & He, Z. (2013). Implementation of IEEE 802.11n LDPC codes based on general purpose processors. In *IEEE international conference on communication technology. (ICCT)* (pp. 218–222). doi:10.1109/ICCT.2013.6820375.
11. Jouguet, P., & Kunz-Jacques, S. (2014). High performance error correction for quantum key distribution using polar codes. *Quantum Information and Computation*, *14*(3-4), 329–338.
12. Le Gal, B., Jego, C., & Crenne, J. (2014). A high throughput efficient approach for decoding LDPC codes onto GPU devices. *IEEE Embedded Systems Letters*, *6*(2), 29–32. doi:10.1109/LES.2014.2311317.
13. Le Gal, B., Leroux, C., & Jego, C. (2014). Software polar decoder on an embedded processor. In *IEEE international workshop on signal processing system. (SiPS)*. doi:10.1109/SiPS.2014.6986083.
14. Le Gal, B., Leroux, C., & Jego, C. (2015). Multi-Gb/s software decoding of polar codes. *IEEE Transactions on Signal Processing*, *63*(2), 349–359. doi:10.1109/TSP.2014.2371781.
15. Leroux, C., Raymond, A., Sarkis, G., & Gross, W. (2013). A semi-parallel successive-cancellation decoder for polar codes. *IEEE Transactions on Signal Processing*, *61*(2), 289–299. doi:10.1109/TSP.2012.2223693.
16. NVIDIA (2012). Kepler GK110 - the fastest, most efficient HPC architecture ever built. NVIDIA's Next Generation CUDA Computer Architecture: Kepler GK110.
17. NVIDIA (2014). NVIDIA management library (NVML), NVML API Reference Guide.
18. NVIDIA (2014). Performance guidelines. CUDA C Programming Guide.
19. Sarkis, G., Giard, P., Thibeault, C., & Gross, W.J. (2014). Autogenerating software polar decoders. In *IEEE global conference on signal and information processing. (GlobalSIP)* (pp. 6–10). doi:10.1109/GlobalSIP.2014.7032067.
20. Sarkis, G., Giard, P., Vardy, A., Thibeault, C., & Gross, W.J. (2014). Fast polar decoders: Algorithm and implementation. *IEEE Journal on Selected Areas in Communications*, *32*(5), 946–957. doi:10.1109/JSAC.2014.140514.
21. Sarkis, G., & Gross, W.J. (2013). Increasing the throughput of polar decoders. *IEEE Communications Letters*, *17*(4), 725–728. doi:10.1109/LCOMM.2013.021213.121633.
22. Tal, I., & Vardy, A. (2013). How to construct polar codes. *IEEE Transactions on Information Theory*, *59*(10), 6562–6582. doi:10.1109/TIT.2013.2272694.
23. Tan, K., Liu, H., Zhang, J., Zhang, Y., Fang, J., & Voelker, G.M. (2011). Sora: high-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, *54*(1), 99–107. doi:10.1145/1866739.1866760.
24. Treibig, J., Hager, G., & Wellein, G. (2010). LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *International conference on parallel process. Workshops (ICPPW)* (pp. 207–216). doi:10.1109/ICPPW.2010.38.
25. Wang, G., Wu, M., Yin, B., & Cavallaro, J.R. (2013). High throughput low latency LDPC decoding on GPU for SDR systems. In *IEEE global conference on signal and information processing. (GlobalSIP)* (pp. 1258–1261). doi:10.1109/GlobalSIP.2013.6737137.

26. Xianjun, J., Canfeng, C., Jaaskelainen, P., Guzma, V., & Berg, H. (2013). A 122 Mb/s turbo decoder using a mid-range GPU. In *International wireless communication and mobile comput. Conference. (IWCMC)* (pp. 1090–1094). doi:10.1109/IWCMC.2013.6583709.

Pascal Giard received the B.Eng. and M.Eng. degree in electrical engineering from École de technologie superieure (ÉTS), Montreal, QC, Canada, in 2006 and 2009, respectively. From 2009 to 2010, he worked as a research professional in the NSERC-Ultra Electronics Chair on 'Wireless Emergency and Tactical Communication' at ÉTS. He is currently working toward the Ph.D. degree at McGill University. His research interests are in the design and implementation of signal processing systems with a focus on modern error-correcting codes.

Gabi Sarkis received the B.Sc. degree in electrical engineering (summa cum laude) from Purdue University, West Lafayette, Indiana, United States, in 2006 and the M.Eng. and Ph.D. degrees from McGill University, Montreal, Quebec, Canada, in 2009 and 2016, respectively. His research interests are in the design of efficient algorithms and implementations for decoding error-correcting codes, in particular non-binary LDPC and polar codes.

Camille Leroux received his M.Sc. degree in Electronics Engineering from the University of South Brittany, Lorient, France, in 2005. He received his Ph.D. degree in Electronics Engineering from TELECOM Bretagne, Brest, France, in 2008. From 2008 to 2011 he worked as a Post Doctoral Research Associate in the Electrical and Computer Engineering Department at McGill University, Montreal, QC, Canada. He is an Associate Professor at Bordeaux INP since 2011. His research interests are in the design and hardware implementation of telecommunication systems and computer architecture.



Claude Thibeault received his Ph.D. from Ecole Polytechnique de Montreal, Canada. He is now with the Electrical Engineering department of Ecole de technologie superieure, where he serves as full professor. His research interests include design and verification methodologies targeting ASICs and FPGAs, defect and fault tolerance, radiation effects, as well as IC and PCB test and diagnosis. He holds 13 US patents and has published more than 140 journal and conference papers, which were cited more than 850 times. He co-authored the best paper award at DVCON'05, verification category. He has been a member of different conference program committees, including the VLSI Test Symposium, for which he was program chair in 2010–2012, and general chair in 2014 and 2015.

Warren J. Gross received the B.A.Sc. degree in electrical engineering from the University of Waterloo, Waterloo, Ontario, Canada, in 1996, and the M.A.Sc. and Ph.D. degrees from the University of Toronto, Toronto, Ontario, Canada, in 1999 and 2003, respectively. Currently, he is a Professor with the Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada. His research interests are in the design and implementation of signal processing systems and custom computer architectures.

Dr. Gross served as Chair of the IEEE Signal Processing Society Technical Committee on Design and Implementation of Signal Processing Systems. He has served as Technical Program Co-Chair of the IEEE Workshop on Signal Processing Systems (SiPS 2012) and as Chair of the IEEE ICC 2012 Workshop on Emerging Data Storage Technologies. Dr. Gross served as Associate Editor for the IEEE Transactions on Signal Processing and currently is a Senior Area Editor. He has served on the Program Committees of the IEEE Workshop on Signal Processing Systems, the IEEE Symposium on Field-Programmable Custom Computing Machines, the International Conference on Field-Programmable Logic and Applications and as the General Chair of the 6th Annual Analog Decoding Workshop. Dr. Gross is a Senior Member of the IEEE and a licensed Professional Engineer in the Province of Ontario.