Low-Latency Software Polar Decoders

Pascal Giard, Gabi Sarkis, Camille Leroux, Claude Thibeault, and Warren J. Gross

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.

Index Terms—Polar codes, successive-cancellation decoding, software decoders

I. 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 [1], [2] 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 [3]–[5]. 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 [6]. 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 [7]. Moreover they are well suited for software implementation, due to regular memory access patterns, on both x86 and embedded processors [8]–[10]. 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 [8] or inter-frame [9], [10], 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 [8]. We start Section II 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 III. Implementation and results on an embedded processor are discussed in Section IV. 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 V. Finally, Section VI compares the energy consumption of the different decoders and Section VIII concludes the paper.

This paper builds upon the work published in [8] and [11]. 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.

II. POLAR CODES

A. 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 ($N \rightarrow \infty$). A polarizing construction where $N = 2$ is shown in Fig. 1a. The probability of correctly estimating bit $u_1$ increases compared to when 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 \( W \) [7].

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 [12]. 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

\[
G_4 = F_2^{\otimes 2} = \begin{bmatrix} F_2 & 0 \\ F_2 & F_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}.
\]

In [7], 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 [8].

**B. Tree Representation of Polar Codes**

A polar code of length \( N \) is the concatenation of two constituent polar codes of length \( N/2 \) [7]. Therefore, binary trees are a natural representation of polar codes [13]. Fig. 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.

**C. 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 \( \alpha \) and \( \beta \), respectively. Messages to a left child \( l \) are calculated by the \( f \) operation using the min-sum algorithm:

\[
\alpha_i[l] = f(\alpha_i[l], \alpha_i[l + N/2]) = \text{sgn}(\alpha_i[l])\text{sgn}(\alpha_i[l + N/2]) \min(|\alpha_i[l]|, |\alpha_i[l + N/2]|),
\]

(1)

where \( N_r \) is the size of the corresponding constituent code and \( \alpha_i[l] \) the LLR input to the node.

Messages to a right child are calculated using the \( g \) operation

\[
\alpha_i[l] = g(\alpha_i[l], \alpha_i[l + N/2], \beta_i[l]) = \begin{cases} 
\alpha_i[l + N/2] + \alpha_i[l], & \text{when } \beta_i[l] = 0; \\
\alpha_i[l + N/2] - \alpha_i[l], & \text{otherwise},
\end{cases}
\]

(2)

where \( \beta_i[l] \) 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

\[
\beta_i[l] = \begin{cases} 
\beta_i[l] \oplus \beta_i[l], & \text{when } i < N_r/2; \\
\beta_i[l] - \beta_i[l - N_r/2], & \text{otherwise},
\end{cases}
\]

(3)

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

**D. 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 [13]. 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, \( \alpha_i[l] \). 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 Fig. 3b, respectively.

**E. 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:
SSC

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:

$$
\beta_i = \begin{cases} 
0, & \text{when } \sum_{i=0}^{N-1} \alpha_i \geq 0; \\
1, & \text{otherwise},
\end{cases}
$$

where \(N\) 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

$$
\beta_i = \begin{cases} 
0, & \text{when } \alpha_i \geq 0; \\
1, & \text{otherwise},
\end{cases}
$$

and then calculating the parity of these decisions

$$
\text{parity} = \bigoplus_{i=0}^{N-1} \beta_i.
$$

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

$$
\beta_i = \beta_i \oplus \text{parity}, \text{ where } i = \arg \min_i |\alpha_i|.
$$

**Repentin-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.

Fig. 3c shows the tree corresponding to a Fast-SSC decoder is will be described more thoroughly in Section III-B. Other types of operations are introduced in the Fast-SSC algorithm, we refer the reader to [14] for more details.

### III. 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. [14], [15], 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 [14]: \(F, G, G_0R\), Combine, Combine_0R, Repetition, 0SPC, RSPC, RepSPC and P_01. An Info function implementing eq. (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 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. 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 [1], [2], [16].

#### A. Instruction-based Decoder

The Fast-SSC decoder implemented on a field-programmable gate array (FPGA) in [14] 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) Data Representation: For the decoders using floating-point numbers, the representation of $\beta$ is changed to accelerate the execution of the $g$ operation on large vectors. Thus, when floating-point LLRs are used, $\beta_i[i] \in \{+1, -1\}$ instead of $\{0, 1\}$. As a result, (2) can be rewritten as

$$g(\alpha[i], \alpha[i + \frac{N}{2}], \beta[i]) = \alpha[i] \cdot \beta[i] + \alpha[i + \frac{N}{2}].$$

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 (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 $\beta[i] \in \{+1, -1\}$.

4) Architecture-specific Optimizations: The decoders take advantage of the SSSE 3, SSE 4.1 and AVX instructions when available. Notably, the $\text{sign}$ and $\text{abs}$ instructions from SSSE 3 and the $\text{blend}$ 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.

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 I, 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 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 I 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 I also shows that vectorized implementations have 3.6 to 5.8 times lower latency than the floating-point decoder.

B. 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 indirects that were required to get good performance out of the instruction-based decoder.

1) Generating an Unrolled Decoder: The polar codes decoded by the instruction-based decoders presented in Section III-A can be specified at run-time. This flexibility comes at the cost of increased branches in the code due to conditionals,
TABLE I: Decoding polar codes with the instruction-based decoder.

| Code (\(N, k\)) | Implementation | Info T/P (Mbps) | Latency (\(\mu s\)) |
|------------------|----------------|----------------|---------------------|
| (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                 |

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

\[
\begin{align*}
\text{F}<8>(\alpha_1, \alpha_2); \\
\text{G}_0\text{R}<4>(\alpha_1, \alpha_2); \\
\text{Info}<2>(\alpha_2, \beta_1); \\
\text{Combine}_0\text{R}<4>(\beta_1, \beta_2); \\
\text{G}<8>(\alpha_1, \alpha_2, \beta_2); \\
\text{SPC}<4>(\alpha_2, \beta_2); \\
\text{Combine}<8>(\beta_2, \beta_3, \beta_4);
\end{align*}
\]

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 [11] 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.

2) Eliminating Superfluous Operations on \(\beta\)-Values: Every non-leaf node in the decoder performs the combine operation (3), rendering it the most common operation. In (3), half the \(\beta\) values are copied unchanged to \(\beta_i\). One method to significantly reduce decoding latency is to eliminate those superfluous copy operations by choosing an appropriate layout for \(\beta\) values in memory: Only \(N\) \(\beta\) values are stored in a contiguous array aligned to the SIMD vector size. When a combine operation is performed, only those values corresponding to \(\beta_i\) 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) Improved Layout of the \(\alpha\)-memory: Unlike in the case of \(\beta\) values, the operations producing \(\alpha\) 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 \(\alpha\) memory is negligible. As an example, for an \(N = 32,768\) floating-point polar decoder using AVX instructions, the size of the \(\alpha\) memory required by the proposed scheme is 262,208 bytes, including a 68-byte overhead.

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.

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

\[
\begin{align*}
\text{std::uint32_t ffindIdx}(\alpha, x, \beta, x_{max}) \{ \\
\quad \text{__mm256 minVec} = \text{__mm256 broadcastb_epi8}(x_{max}); \\
\quad \text{__mm256 mask} = \text{__mm256 cmpgeq_epi8(minVec, x}); \\
\quad \text{std::uint32_t mvMask} = \text{__mm256 movemask_epi8(mask)}; \\
\quad \text{return } \text{__tzcnt_u32(mvMask)};
\end{align*}
\]
Third, for the floating-point implementation, $\beta$ was changed to be in $[0, -128]$ instead of $[1, -1]$. In the floating-point representation [17], 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 $\beta$, multiplications are replaced by faster bitwise XOR operations. Similarly, for the 8-bit fixed-point implementation, $\beta$ 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.

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 $\alpha$ and $\beta$ values, respectively. The size of the $\beta$-memory is

$$M_\beta = NW_\beta,$$  
(5)

where $N$ is the frame length, $W_\beta$ is the number of bits used to store a $\beta$ value and $M_\beta$ is in bits.

The size of the $\alpha$-memory can be expressed as

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

where $N$ is the frame length, $W_\alpha$ is the number of bits used to store an $\alpha$ value, $A$ is the number of $\alpha$ values per SIMD vector and $M_\alpha$ is in bits. Note that the expression of $M_\alpha$ contains the overhead $M_{\alpha\text{OH}}$ due to tightly packing the $\alpha$ values as described in Section III-B3:

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

The memory footprint can thus be expressed as

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

The memory footprint in kilobytes can be approximated with

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

7) Implementation Comparison: We first compare the SIMD-float results for this implementation—the unrolled decoder—with those from Section III-A—the instruction-based decoder. Then we show SIMD-int8 results and compare them with that of the software decoder of Le Gal et al. [10].

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 II 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 I and II 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 [10], Table III 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 [10] 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 [10]. The results in [10] 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

---

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

```c
template<unsigned int N>
void G(float* a, float* a_out, float* b, float* b_out) {
    for (unsigned int i = 0; i < N; i += 8) {
        _mm256_a = _mm256_load_ps(a + i);
        _mm256_b = _mm256_load_ps(b + i);
        _mm256_c = _mm256_load_ps(a_out + i);
    }
}
```

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

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

template<unsigned int N>
void G(unsigned* a, unsigned* a_out, unsigned* b, unsigned* b_out) {
    for (unsigned int i = 0; i < N; i += 32) {
        _mm256_a = _mm256_load_si256(a + i);
        _mm256_b = _mm256_load_si256(b + i);
        _mm256_c = _mm256_load_si256(a_out + i);
    }
}
```

**Table II:** 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 (\mu s) |
|------------|----------------|-----------------|
| 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 |
decoding speed of the (2048, 1707) code on the Intel Core
a similar, but lesser, eff
lower latency. Increasing the code length was observed to have
resulted in SSC decoder trees with fewer nodes and, therefore,
first discussed in [18]. It was noted that higher rate codes
latency to decrease with increasing code rate and length was
is smaller for higher code rates. The tendency of decoding
can be seen that for a fixed code length, the decoding latency
3.9 GHz for the i7-4960HQ and i7-4770S CPUs, respectively.

Looking at the first two, or last two rows of Table II, it
Looking at the first two, or last two rows of Table II, it
Comparison with Similar Works:
Comparison with Similar Works: Results were obtained
processors. All the strategies used to develop an e
er SIMD extensions, e.g. NEON for ARM

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 II, 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 [18]. 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 IV. 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.

IV. 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 III-B6 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 V shows the results for the proposed unrolled decoders and provides a comparison with [9]. As with their desktop CPU implementation of [10], 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 [9] 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 [9], 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 [9]. Both implementations are using 8-bit fixed-point values.

V. 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 [19] 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

\[ \text{TABLE IV: Effect of unrolling and algorithm choice on decoding speed of the } (2048, 1707) \text{ code on the Intel Core i7-4770S} \]

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

\[ \text{TABLE V: Decoding polar codes with 8-bit fixed-point numbers on an ARM Cortex A9 using NEON.} \]

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

\[ \text{*Results linearly scaled for the clock frequency difference.} \]

\[ \text{TABLE III: Comparison of the proposed software decoder with that of [10].} \]

| Decoder | Target | L3 Cache | \( f \) (GHz) | Code \((N, k)\) | Mem. footprint (kbytes) | Info T/P (Mbps) | Latency (µs) |
|---------|--------|----------|---------------|----------------|------------------------|----------------|--------------|
| [10]*   | 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           |

\[ \text{*Results with Turbo enabled.} \]
refer to the decoding latency including the delay required to copy the data in and out of the GPU.

A. 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 [20].

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 Sect. V-C, 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.

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

Fig. 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.

D. 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 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 [21]. 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 [21]. In total, 416 arithmetic or logic operations and 208 load or store operations can occur simultaneously.

Fig. 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.

D. 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.

E. 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 value of blocks per kernel though, solely using the shared 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.

F. 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 [19].

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

\[ M_{\text{total}} \text{(kbytes)} = \frac{N(W_\beta + W_\alpha)BS}{8000}, \]  

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 \( W_\beta \) and \( W_\alpha \) are the number of bits required to store a \( \beta \) and an \( \alpha \) value, respectively. For best performance, as detailed in the next section, both \( \beta \) and \( \alpha \) values are represented with floating-point values and thus \( W_\beta = W_\alpha = 32 \).

G. 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 [19] clearly shows that integers are third class citizens behind single- and double-precision floating-point numbers. As an example, Table 2 of [19] 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.

H. Results

Table VI 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 VI 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.
On recent Intel processors, power usage can be calculated by accessing the Running Average Power Limit (RAPL) counters. 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 calculate 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 [23] 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) [24].

Table VII 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 [10]. 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

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

It can be seen that the proposed decoder is slightly more energy efficient on a desktop processor compared to that of [10]. 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.

### VI. 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 calculate the power usage when the processor was idle and under load.

| Decoder         | Target               | Mem. Footprint (kbytes) | Info. T/P (Gbps) | Latency (µs) | Power (W) | Energy (nJ/info. bit) |
|-----------------|----------------------|-------------------------|------------------|--------------|-----------|-----------------------|
| [10]            | 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.

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 [25]. 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.

### VII. Further Discussion

#### A. 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 [25]. 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.
B. 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.

C. 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 [26]. Similarly, the performance of the state-of-the-art software LDPC decoders is compared against that of our proposed unrolled decoders for polar codes.

The fastest software LDPC decoders in literature are those of [6], which implements decoders for the 802.11n standard. 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 specialisation, 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 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 of 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.

VIII. 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 specialisation, 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 of 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.

| Decoder | N/M | Rate | Latency total (ms) per frame (µs) | Info. T/P (Mbps) |
|---------|-----|------|-----------------------------------|-----------------|
| [6]     | 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           |
ACKNOWLEDGEMENT

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] K. Tan, H. Liu, J. Zhang, Y. Zhan, Y. Fang, and G. M. Voelker, “Sora: High-performance software radio using general-purpose multi-core processors,” Commun. ACM, vol. 54, no. 1, pp. 99–107, Jan. 2011.

[2] J. Demmel, S. Koslowski, and F. Jondral, “A LTE receiver framework using GNU Radio,” J. Signal Process. Syst., vol. 78, no. 3, pp. 313–320, 2015.

[3] J. Xianjun, C. Canfeng, P. Jaaskelainen, V. Guzman, and H. Berg, “A 122M/s turbo decoder using a mid-range GPU,” in Int. Wireless Commun. and Mobile Comput. Conf. (IWCMC), July 2013, pp. 1090–1094.

[4] G. Sarkis and W. J. Gross, “Increasing the throughput of polar decoders,” in IEEE J. Sel. Areas Commun. (GlobalSIP), Dec 2013, pp. 1258–1261.

[5] B. Le Gal, C. Thibeault, and C. Jego, “A high throughput efficient approach for decoding LDPC codes onto GPU devices,” IEEE Embedded Syst. Lett., vol. 6, no. 2, pp. 29–32, June 2014.

[6] X. Han, K. Niu, and Z. He, “Implementation of 802.11n LDPC codes based on general purpose processors,” in IEEE Int. Conf. on Commun. Technol. (ICCCT), Nov 2013, pp. 218–222.

[7] E. Arıkan, “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.

[8] P. Giard, G. Sarkis, C. Thibeault, and W. J. Gross, “Fast software polar decoders,” in IEEE Int. Conf. on Acoast., Speech, and Signal Process. (ICASSP), May 2014, pp. 7555–7559.

[9] B. Le Gal, C. Leroux, and C. Jego, “Software polar decoder on an embedded processor,” in IEEE Int. Workshop on Signal Process. Syst. (SIPS), Oct 20–22 2014.

[10] ——, “Multi-Gb/s software decoding of polar codes,” IEEE Trans. Signal Process., vol. 63, no. 2, pp. 349–359, Jan 2015.

[11] G. Sarkis, P. Giard, C. Thibeault, and W. J. Gross, “Autogenerating software polar decoders,” in IEEE Global Conf. on Signal and Inf. Process. (GlobalSIP), Dec 2014, pp. 6–10.

[12] I. Tal and A. Vardy, “How to construct polar codes,” IEEE Trans. Inf. Theory, vol. 59, no. 10, pp. 6562–6582, Oct 2013.

[13] 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.

[14] G. Sarkis, P. Giard, A. Vardy, C. Thibeault, and W. J. Gross, “Fast polar decoders: Algorithm and implementation,” IEEE J. Sel. Areas Commun., vol. 32, no. 5, pp. 946–957, May 2014.

[15] C. Leroux, A. Raymond, G. Sarkis, and W. Gross, “A semi-parallel successive-cancellation decoder for polar codes,” IEEE Trans. Signal Process. (GlobalSIP), vol. 61, no. 2, pp. 289–299, 2013.

[16] S. Bang, C. Ahn, Y. Jin, S. Choi, J. Glossner, and S. Ahn, “Implementation of LTE system on an SDR platform using CUDA and UHD,” Analog Integr. Circuits and Signal Process., vol. 78, no. 3, pp. 599–610, 2014.

[17] “IEEE standard for floating-point arithmetic,” IEEE Std 754-2008, pp. 1–70, Aug 2008.

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

[19] NVIDIA. “Performance guidelines,” CUDA C Programming Guide, Aug 2014.

[20] W.-C. Feng and S. Xiao, “To GPU synchronize or not GPU synchronize?” in in IEEE Int. Symp. on Circuits and Syst. (ISCAS), May 2010, pp. 3801–3804.

[21] NVIDIA. “Kepler GK110 - the fastest, most efficient HPC architecture ever built,” NVIDIA’s Next Generation CUDA Computer Architecture: Kepler GK110, Dec 2012.

[22] “PCI express base specification revision 2.0,” PCI-SIG, Dec 2006.

[23] J. Treibig, G. Hager, and G. Wellein, “LIKwid: A lightweight performance-oriented tool suite for x86 multicore environments,” in Int. Conf. on Parallel Process. Workshops (ICPPW), Sept 2010, pp. 207–216.

[24] NVIDIA, “NVIDIA management library (NVML),” NVML API Reference Guide, March 2014.

[25] P. Jouguet and S. Kunz-Jacques, “High performance error correction for quantum key distribution using polar codes,” Quantum Inf. & Computation, vol. 14, no. 3–4, pp. 329–338, 2014.

[26] “IEEE 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), pp. 1–2793, Mar 2012.
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 an Associate 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 is currently 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. 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.