Autogenerating Software Polar Decoders

Gabi Sarkis[⋆], Pascal Giard[⋆], Claude Thibeault[†], and Warren J. Gross[⋆]
[⋆]Department of Electrical and Computer Engineering, McGill University, Montréal, Québec, Canada
Email: gabi.sarkis@mail.mcgill.ca, pascal.giard@mail.mcgill.ca, and warren.gross@mcgill.ca
[†]Department of Electrical Engineering, École de technologie supérieure, Montréal, Québec, Canada
Email: claude.thibeault@etsmtl.ca

***Abstract*—Polar decoders are well suited for high-speed software implementations. In this work, we present a framework for generating fully-unrolled software polar decoders with branchless data flow. We discuss the memory layout of data in these decoders and show the optimization techniques used. At 335 Mbps, when decoding a (2048, 1707) polar code, the resulting decoder has more than twice the speed of the state of the art floating-point software polar decoder.**

***Index Terms*—polar codes, decoder, software**

I. Introduction

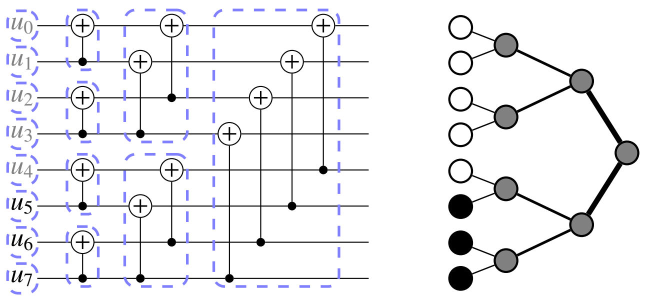
Polar codes [1] have been the topic of significant research recently, due to their ability to asymptotically achieve the symmetric channel capacity when decoded using the low-complexity successive-cancellation (SC) decoding algorithm.

Multiple hardware decoders have been presented in literature [2]–[5]; the fastest of which [5] achieved an information throughput of 1 Gbps when decoding a $(32768, 29492)$ polar code on an field-programmable gate-array (FPGA) running at 100 MHz.

Recently, software polar decoders have been shown to have high throughput when running on modern CPUs [6]. These decoders implement the fast simplified successive-cancellation (Fast-SSC) algorithm introduced in [5], exploiting CPU parallelism by using single-instruction multiple-data (SIMD) instructions. When decoding a $(32768, 29492)$ polar code on a single CPU core running at 3.4 GHz, the floating-point decoder and the 8-bit fixed-point decoders had information throughput of 156 and 226 Mbps, respectively.

The length of the polar codes used by the decoders of [6] is defined at compile-time. The rate of the codes and the location of frozen bits can be configured at run-time by changing a file containing instructions corresponding to the code. This flexibility comes at the cost of extra control logic and indirection in function calls, reducing the speed of the decoder. Knowing the location of the frozen bits at compile time enables us to further exploit SIMD parallalisation and write branchless decoder source code.

In this work, we present a framework for generating branchless compile-time-specified polar decoders that offer significant speed improvements over state of art software decoders and most hardware polar decoders. We start by reviewing decoder trees of polar codes and the Fast-SSC decoding algorithm in Section II. We then describe the framework generating the decoders in Section III and the decoder architecture in Section IV. Finally, the results of the proposed decoder are compared with others in literature in Section V.



(a) Graph (b) Decoder tree

Fig. 1: From a graph to a decoder tree, the case of a $(8, 3)$ polar code.

II. Background

A. Decoder Trees of Polar Codes

Every polar code of length N is the concatenation of two constituent polar codes of length $N/2$ [1]. As shown in [7], this structure leads to binary trees being a natural representation of polar codes. Fig. 1 shows how to represent an (8, 3) polar code as a binary tree. In Fig. 1a, the frozen—always set to zero—bits are labeled in gray while the information bits are in black. The corresponding tree, shown in Fig. 1b, uses white and black leaf nodes to denote frozen and information bits, respectively. The gray nodes in Fig. 1b correspond to the operations required to perform the concatenation shown in Fig. 1a, and each sub-tree corresponds to a constituent code.

Successive-cancellation (SC) decoding traverses the decoder tree depth-first, exploring left edges before backtracking to right ones, and progressing all the way to the size-1 frozen and information leaves. The decoder uses log-likelihood ratios (LLRs) for messages passed to children and bits for the messages passed to parents. 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 according to

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

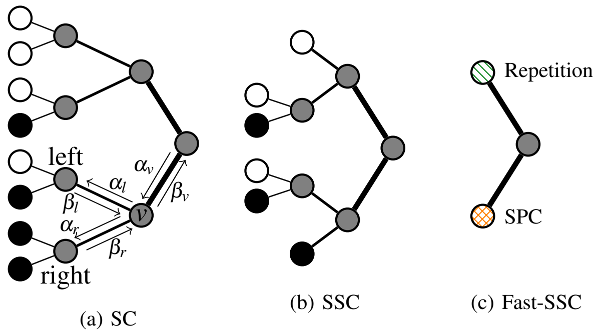


Fig. 2: Decoder trees corresponding to the SC, SSC and Fast-SSC decoding algorithms.

where N_v is the size of the corresponding constituent code and α_v the LLR input to the current node. Messages to a right child are calculated according to the g operation

$$\begin{aligned}\alpha_r[i] &= g(\alpha_v[i], \alpha_v[i + {}^{N_v}\!/2], \beta_l[i]) \\ &= \begin{cases}\alpha_v[i + {}^{N_v}\!/2] + \alpha_v[i] & \text{when } \beta_l[i] = 0, \\ \alpha_v[i + {}^{N_v}\!/2] - \alpha_v[i] & \text{when } \beta_l[i] = 1;\end{cases}\end{aligned} \tag{1}$$

where β_l is the bit estimate from the left child.

The bit estimates are calculated at the leaves either by setting them to zero for frozen bits or by performing threshold detection for information ones. Once a node has the bit estimates from both its children, it combines them to generate its own estimate, which is passed to its parent, according to

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

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

The simplified successive-cancellation (SSC) [7] algorithm does not traverse a sub-tree whose leaves are all frozen as its bit estimate is known a priori to be the all zero vector. Similarly, sub-trees where all leaves are information nodes need not to be traversed either as the maximum-likelihood (ML) output of such a tree can be obtained by performing element-wise threshold detection on the soft-information input vector, α_v. This reduces the number of nodes in the tree and the associated number of calculations, thereby increasing decoding speed. Fig. 2a shows the SC tree of a $(8, 4)$ polar code. Fig. 2b is the SSC pruned tree corresponding to the same code in which the white and black nodes correspond to the Info and Frozen node types respectively.

B. *The Fast-SSC Decoding Algorithm*

In [8], it was shown that further pruning of the decoder tree by using resource-constrained exhaustive-search ML decoding for all constituent codes of length 4 and rate $R \in (0, 1)$ improved decoding speed significantly compared to the SSC decoding algorithm.

More pruning opportunities were introduced in the Fast-SSC decoding algorithm [5]. Repetition codes were identified and decoded with an efficient ML decoding algorithm; and so were single-parity-check (SPC) codes. Other node mergers were introduced as well and are discussed in [5].

Fig. 2c illustrates the tree corresponding to a Fast-SSC decoder. It has significantly fewer nodes to visit and operations to perform than that of the SSC decoder. As such, the Fast-SSC decoder is faster than the SSC decoder [5].

III. Optimized Polar-Decoder Generator

In this section we describe the optimized polar-decoder generator (OPDG) we developed to generate polar decoders. It operates in three phases: generating the SC tree, pruning the tree to that of a Fast-SSC decoder, and generating the unrolled C++ code. A user provides OPDG with optimization parameters and a file containing the frozen-bit locations of the polar code to obtain a C++ file containing the decoder flow. This decoder file is compiled in combination with other files containing implementation details using a C++ compiler resulting in the decoder executable.

OPDG parses the polar code description into a binary tree comprising three types of nodes: the leaf nodes corresponding to frozen bits are rate-0 nodes, those corresponding to information bits are rate-1 nodes, and all the non-leaf nodes are treated as rate-R nodes. This is the SC decoder tree shown in Fig. 2a.

A. *Tree Optimization*

The optimizer is provided with a list of available node types. It traverses the decoder tree top-to-bottom starting with the root node. At each node, the optimizer tests whether the node can be replaced by another node type with a lower cost. The cost is defined according to the targeted implementation, e.g. in terms of latency or hardware resources used. For the software decoder, the cost function is directly proportional to the number of memory accesses. When multiple nodes can be used, the one with the lowest cost is chosen. If the replacement node function does not require sub-tree traversal in a direction, the optimizer only traverses the other direction. When a node is a leaf node, e.g. SPC or Repetition, the optimizer stops and moves back up to the parent node. Fig. 2b is the optimized decoder tree for an (8, 4) code when Info and Frozen node types are enabled. While, Fig. 2c shows the decoder tree built with cost measured using latency and with the SPC and Repetition nodes enabled as well. It can be observed that Repetition and SPC nodes replaced higher-latency sub-trees.

B. *C++ Code Generation*

The final stage in the compilation process is to generate the decoder C++ source code. The C++ code generator traverses the optimized decoder tree starting with the root node. For each node, the following occurs

1) If the node has a left child, generate code to calculate α_l and move to the left child.
2) If the node has a right child, generate code to calculate α_r and move to the right
3) When messages from both children are available, or the node is a leaf, generate code to calculate β_v and move to the parent.

Algorithm 1 printNode(node)

```
if node.hasLeftChild() then
    print function-call to calculate $\alpha_l$
    printNode(node.leftChild())
end if
if node.hasRightChild() then
    print function-call to calculate $\alpha_r$
    printNode(node.rightChild())
end if
print function-call to calculate $\beta_v$
```

Listing 2 Unrolled (8, 4) Fast-SSC Decoder

```
F<8>(αc, α1);
Repetition<4>(α1, β1);
G<8>(αc, α2, β1);
SPC<4>(α2, β2);
Combine<8>(β1, β2, βc);
```

This process is summarized in Algorithm 1 and is applied recursively until the code to calculate the root node's β_v output has been generated.

The unrolled function calls corresponding to the decoder in Fig. 2c are shown in Listing 2 and further illustrated in Fig. 3. The decoder receives the channel information α_c and applies the f operation to the 8-LLR input, generating the 4-LLR output α_1 by calling the F function. The left child of the root is a repetition node of size 4 that takes α_1 and generates β_1. Once β_1 is computed, the decoder is back at the root node and applies the g operation to α_c and β_1 to calculate the input to the root's right child, α_2. The SPC node of size 4 uses α_2 to calculate β_2, which is then combined with β_1 to generate the estimated codeword β_c. From this example, it is observed that data flows sequentially from one decoder function to the next, without any conditionals or loops. The node sizes are passed as template arguments, indicated by < and >, to enable better exploitation of compile-time vectorization as will be shown in Section IV.

IV. The Decoder

A. Memory Layout

There are two types of information to store in this decoder: the LLR values and the bit values. They are respectively denoted as α and β.

α values are represented using single-precision floating-point numbers. A contiguous section of memory is allocated for the $\log_2 N$ stages of α values. This memory is aligned to a 16- or 32-byte boundary when SSE or AVX instructions are used, respectively. In addition to aligning the entire block, the memory for each stage is also aligned to the same boundary. This allows for vectorization even when the stage size N_v is smaller than the CPU vector size. The memory wasted by not tightly packing the stages of α memory is small. For example, for an $N = 32768$ polar decoder using the AVX instruction set, the size of the α memory required by the proposed scheme is 262,160 bytes. The memory required by a tightly-packed

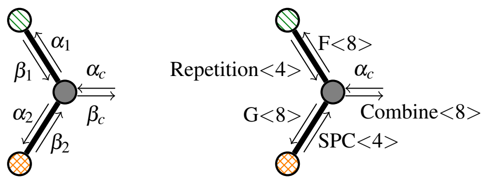


Fig. 3: Dataflow graph of a $(8, 4)$ polar decoder.

scheme amounts to 262,140 bytes. Thus our scheme results in a mere 20-byte overhead.

Copying β_r to β_v unchanged constitutes half the operations in (2) and a significant amount of all calculations involving β values. Therefore, the β memory layout was designed to eliminate the need for copy operations in such cases. Only N β values are stored, each as a single-precision floating-point number. Thus, unlike the α memory, there is no padding or explicit alignment except for the overall memory. However, since the stage sizes are all powers of two, stages of sizes equal to or larger than the CPU vector size will be implicitly appropriately aligned so that efficient vector operations can be used.

B. Computation

While the compiler generates the decoder code calling each function, the functions themselves were manually implemented in a different file using intrinsics targeting the SSE or AVX instruction set. In this work, we only detail the AVX implementation since the SSE one can be similarly implemented. To enable vectorization when implementing the g operation (1) in the form of the G functions, β values were implemented using the single-precision floating-point values -1.0 and 1.0. This can be further optimized as will be shown in Section IV-C.

Since the sizes of the constituent codes are known at compile time and provided as template parameters, each function had two implementations: one for stages smaller than the AVX vector size—8 single-precision floating-point values—where vectorization is not possible or straightforward, and one for the larger stages. For example, Listings 3 and 4 show the vectorized and non-vectorized implementations of the g operation where α^* and β^* are pointers to α and β values respectively. P is the number of elements per SIMD vector—8 for AVX. When the size N_v of a node is greater than P, the vectorized function is used, the non-vectorized one is used otherwise. Those listings also illustrate how branching is avoided for the g function by using multiplication by elements of β_{in} to change the sign of elements of α_{in}.

By providing the function implementation as template specializations, we allow the C++ compiler to unroll loops, which all have compile-time known bounds, and inline functions as it deems fit.

C. Other Optimization

The goal of the multiplication in Listings 3 and 4 is to change the sign of $\alpha_{in}[i]$ when $\beta_{in}[i]$ is negative. However,

Listing 3 Vectorized G function (g operation)

```
template<unsigned int $N_v$>
void G($\alpha^*$ $\alpha_{in}$, $\alpha^*$ $\alpha_{out}$, $\beta^*$ $\beta_{in}$) {
  for (unsigned int i = 0; i < $N_v/2$; i += $P$) {
    __m256 $\alpha_l$ = _mm256_load_ps($\alpha_{in}$ + i);
    __m256 $\alpha_r$ = _mm256_load_ps($\alpha_{in}$ + i + $N_v/2$);
    __m256 $\beta_v$ = _mm256_load_ps($\beta_{in}$ + i);
    __m256 $\alpha_l'$ = _mm256_mul_ps($\beta_v$, $\alpha_l$);
    __m256 $\alpha_v$ = _mm256_add_ps($\alpha_r$, $\alpha_l'$);
    __mm256_store_ps($\alpha_{out}$ + i, $\alpha_v$);
  }
}
```

Listing 4 Non-vectorized G function (g operation)

```
template<unsigned int $N_v$>
void G($\alpha^*$ $\alpha_{in}$, $\alpha^*$ $\alpha_{out}$, $\beta^*$ $\beta_{in}$) {
  for (unsigned int i = 0; i < $N_v/2$; ++i) {
    $\alpha_{out}$[i] = $\alpha_{in}$[i + $N_v/2$] + ($\beta_{in}$[i] * $\alpha_{in}$[i]);
  }
}
```

floating-point multiplication, used as well when combining β_l and β_r to calculate β_v, is a slow operation. To eliminate it, we take advantage of the IEEE-754 standard representation for floating-point numbers used in modern CPUs. Under this standard, a number's most-significant bit represents the sign and we can perform the sign-change operation using a bit-wise XOR operation where all bits in the β values are always zero with the exception of the sign bit. Therefore, we use $\beta \in \{0.0, -0.0\}$ and XOR operations instead of $\beta \in \{1.0, -1.0\}$ and multiplication to significantly increase the speed of the decoder.

V. Experimental Results

A. Methodology

In this section, we compare the proposed software polar decoder with the fastest software polar decoders in literature [6] and to some hardware polar decoders. For that purpose, all software were compiled using the C++ compiler from GCC 4.8 using the flags "`-march=native -funroll-loops -Ofast`". Auto-vectorization and link-time optimization were enabled as per default. Decoders are inserted in a digital communication chain to measure their performance. We use binary phase shift keying (BPSK) over an AWGN channel with random codewords.

The time required to decode a frame, or latency, includes the time required to copy a frame to decoder memory and copy back the estimated codeword, and is measured using the high precision clock provided by the Boost Chrono library. Codeword generation, transmission over the channel, and demodulation are excluded from calculations. The copying of data to and from the decoder is included to facilitate comparisons with non-CPU-based decoders.

TABLE I: Comparing software polar decoders for codes of rates 1/2, 5/6, 0.84 and 0.9.

Decoder	(N, k)	Info T/P (Mbps)	Latency (μs)
[6]	$(2048, 1024)$	71.50	14
this work		147.34	7
[6]	$(2048, 1707)$	154.06	11
this work		335.17	5
[6]	$(16384, 14746)$	151.23	98
this work		292.43	50
[6]	$(32768, 27568)$	123.68	223
this work		219.77	125
[6]	$(32768, 29492)$	156.40	189
this work		260.81	113

Decoders were allowed to use only one core of an Intel i7-2600 x86 CPU running at 3.4 GHz, with turbo-boosting disabled.

B. Comparison with Previous Works

In this subsection we compare the information throughput and latency of decoders generated with the proposed framework against the single-precision floating point version of the decoders presented in [6]. To the best of our knowledge, the SIMD decoders of [6] are the fastest software polar decoders in the literature.

Table I shows that our proposed decoder has a lower latency and greater throughput for all polar codes and every code rate. The improvement is close to a factor of 2 for almost all codes. The $(2048, 1707)$ polar code shows the greatest improvement at 2.18 times the throughput. On the opposite side of the spectrum, is the $(32768, 29492)$ code with a 1.67 time speedup.

C. A Note About Hardware Polar Decoders

Even though the proposed decoder uses the single-precision floating-point number representation instead of quantized integers, this work is competitive with most hardware implementations with the exception of the tree-based 2b-SC decoder of [4] and the hardware implementation of the Fast-SSC algorithm [5]. For example, the semi-parallel polar decoder of [2] is nearly 5 and 10 times slower for the $(2048, 1707)$ and $(32768, 29462)$ codes, achieving only 69.2 Mbps and 27.6 Mbps respectively. The proposed decoder is slower than the fastest non-Fast-SSC decoder in literature [4] which has an estimated information throughput of 250 Mbps at 750 MHz for a $(1024, 512)$ code. This work is 1.7 times slower at 144.3 Mbps for the same code.

VI. Conclusion

In this paper we have presented a software framework to autogenerate polar decoders. Its use results in fully-unrolled decoders with a branchless data flow that can be better optimized at compile time. As a consequence, these decoders present a much lower latency and greater throughput than our previous works.

ACKNOWLEDGEMENT

Claude Thibeault is a member of ReSMiQ. Warren J. Gross is a member of ReSMiQ and SYTACom. The authors would like to thank Alexandre J. Raymond for helpful discussions.

References

[1] 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.

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

[3] A. Pamuk and E. Arıkan, "A two phase successive cancellation decoder architecture for polar codes," in *IEEE Int. Symp. on Inf. Theory (ISIT)*, Jul. 2013, pp. 1–5.

[4] B. Yuan and K. Parhi, "Low-latency successive-cancellation polar decoder architectures using 2-bit decoding," *IEEE Trans. Circuits Syst. I*, vol. 61, no. 4, pp. 1241–1254, April 2014.

[5] 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.

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

[7] 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.

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