Memory Management in Successive-Cancellation based Decoders for Multi-Kernel Polar Codes

Valerio Bioglio, Carlo Condo, Ingmar Land
Mathematical and Algorithmic Sciences Lab
Huawei Technologies France SASU
Email: {valerio.bioglio,carlo.condo,ingmar.land}@huawei.com

***Abstract*—Multi-kernel polar codes have recently been proposed to construct polar codes of lengths different from powers of two. Decoder implementations for multi-kernel polar codes need to account for this feature, that becomes critical in memory management. We propose an efficient, generalized memory management framework for implementation of successive-cancellation decoding of multi-kernel polar codes. It can be used on many types of hardware architectures and different flavors of SC decoding algorithms. We illustrate the proposed solution for small kernel sizes, and give complexity estimates for various kernel combinations and code lengths.**

***Index Terms*—Polar Codes, Successive Cancellation Decoding, Decoder Architectures.**

***Topic Designation*–A. Communication Systems, 1. Modulation and Coding.**

I. Introduction

Polar codes [1] are a family of error correcting codes with capacity-achieving property over various classes of channels, providing excellent error rate performance for practical code lengths [2]. The construction of polar codes is based on the polarization effect of the Kronecker powers of the binary 2×2 kernel matrix $T_2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}$. A major drawback of this construction is the restriction of achievable block lengths to powers of 2. Puncturing and shortening techniques can be used to adjust the code length, at the cost of a reduced bit polarization [3]. To overcome this limitation, multi-kernel polar codes have been introduced in [4]. By mixing binary kernels of different sizes in the construction of the code, these codes prove that many block lengths can be achieved while keeping the polarization effect.

Many software and hardware implementations of polar code decoders have been proposed in literature. While software guarantees a higher degree of flexibility in terms of data structures, fast software decoders have to rely on efficient memory management [5], [6]. The importance of smart memory usage is even more evident in hardware implementations, where memory accounts for the majority of area occupation and power consumption, and heavily impacts decoder speed [7]–[9]. The memory structure first proposed in [10] for purely binary polar codes, and widely adopted in SC-based decoders [11], relies on the observation that memory requirements decrease as the decoding stage increases. We show how this trend continues in multi-kernel polar codes, proposing an efficient memory structure for SC-based polar decoders, and

$$G_{12} = \left(\begin{array}{ccc:ccc|ccc:ccc}
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hdashline
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
\hdashline
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 \\
0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 1 & 1
\end{array}\right)$$

Fig. 1: Transformation matrix $G_{12} = T_2 \otimes T_2 \otimes T_3$.

providing functions for the evaluation of the overall memory requirements. This structure supports the decoding of codes constructed with any combination of kernel sizes, making it an ideal framework for multi-kernel decoder hardware implementations [12].

II. Multi-Kernel Polar Codes

Multi-kernel polar codes generalize the construction of polar codes by mixing binary kernels of different sizes. Similarly to polar codes, an (N, K) multi-kernel polar code is completely defined by a $N \times N$ transformation matrix G_N and a frozen set $\mathcal{F}$, with $|\mathcal{F}| = N - K$. Transformation matrix has the form

$$G_N = T_{p_1} \otimes T_{p_2} \cdots \otimes T_{p_s}, \tag{1}$$

where T_{p_i} is a $p_i \times p_i$ binary matrix, $i = 1, 2, \ldots, s$, denoting a polarizing kernel of size p_i, and $N = p_1 \cdot p_2 \cdot \ldots \cdot p_s$. Binary kernels of different sizes can be found in [13]. Transformation matrix $G_{12} = T_2 \otimes T_2 \otimes T_3$ is shown in Figure 1, where

$$T_2 = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}, \qquad T_3 = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & 1 \end{pmatrix} \tag{2}$$

and the recursive structure of the matrix is highlighted. The frozen set $\mathcal{F}$ indicates the $N - K$ bits to be frozen in the code construction, and can generally be designed according to bit reliabilities [4] or minimum distance [14]. Finally, the encoder is defined by $x = u \cdot G_N$, mapping the input vector $u \in \mathbb{F}_2{}^N$ to the codeword $x \in \mathbb{F}_2{}^N$, where $u_i = 0$ for $i \in \mathcal{F}$, and u_i,

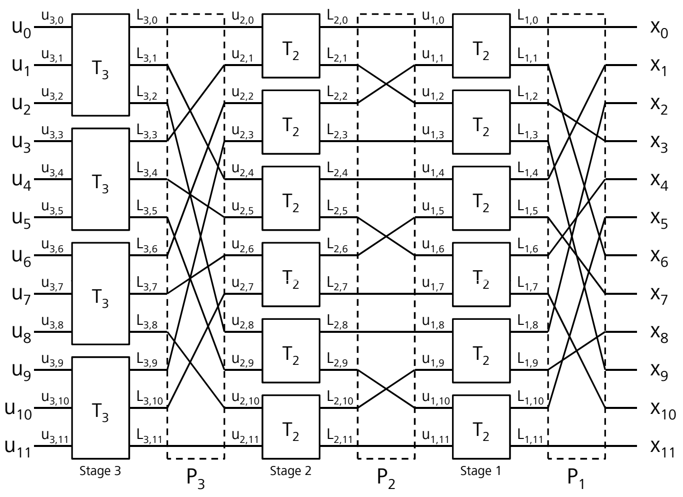
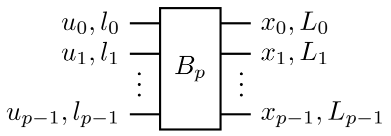


Fig. 2: Tanner graph defined by $G_{12} = T_2 \otimes T_2 \otimes T_3$.

$i \notin \mathcal{F}$, stores the information bits. We recall the set $\mathcal{I} = \mathcal{F}^c$ to be termed as information set.

The structure of multi-kernel polar codes can be better understood through the Tanner graph of the code; this consists of various $p_i \times p_i$ blocks B_{p_i}, corresponding to the different T_{p_i} kernels used in the construction of the transformation matrix, connecting input vector and codeword. Each of the s stages composing the graph is formed by $N_i = N/p_i$ kernel blocks B_{p_i}, performing the operations involving kernel T_{p_i}. Permutations P_i between stages are described in [4]; an example of Tanner graph for a G_{12} is given in Figure 2.

Multi-kernel polar codes can be decoded through successive cancellation (SC) decoding on the Tanner graph of the code, where log-likelihood ratios (LLRs) [15] are passed from the right to the left, while partial sums (PSs) based on hard decisions on the decoded bits are passed from the left to the right. LLRs and PSs are calculated in the kernel blocks, depicted as



Blocks in the same column belong to the same stage and can perform decoding operations in parallel. Roughly speaking, L_i and l_i represent the LLRs of the partial sums u_i and x_i respectively. However, PSs are calculated on the basis of the previously decoded bit, hence they may not match with the connected LLRs. We indicate with $L_{i,(j-1)p_i}, \dots, L_{i,jp_i-1}$ and $u_{i,(j-1)p_i}, \dots, u_{i,jp_i-1}$ the LLRs and PSs input of the j-th block of stage i respectively, with $j \leq N_i = N/p_i$. LLRs $L_{1,0}, \dots, L_{1,N-1}$ correspond to channel LLRs, while $u_{s,0}, \dots, u_{s,N-1}$ correspond to the decoded bits. An example of this labeling is given in Figure 2.

Given the binary input vector $u = (u_0, u_1, \dots, u_{p-1})$, corresponding to the partial sums calculated from the decoded bits, the output vector $x = (x_0, x_1, \dots, x_{p-1})$ is calculated as $x = u \cdot T_p$. If we call T_p^i the i-th column of the kernel matrix T_p, the update rule for the PSs can be written as

$$x_i = u \cdot T_p^i. \tag{3}$$

The vector x corresponds to the partial sums calculated by the kernel T_p, that will be used as input for the LLRs calculations of other blocks. This update rule is performed from left to right, and can be used also for the encoding.

Output LLRs $l_0, \dots, l_{p-1}$ are calculated sequentially using the input LLRs $L_0, \dots, L_{p-1}$ coming from the previous stage and the PSs corresponding to the previously decoded bit, i.e.,

$$l_i = f_i^p(L_0, \dots, L_{p-1}, u_0, \dots, u_{i-1}), \tag{4}$$

with $l_0 = f_0^p(L_0, \dots, L_{p-1})$. This update is performed from the right to the left, and corresponds to the successive cancellation principle. Rules for the derivation of LLR update functions for arbitrary binary kernels can be found in [16].

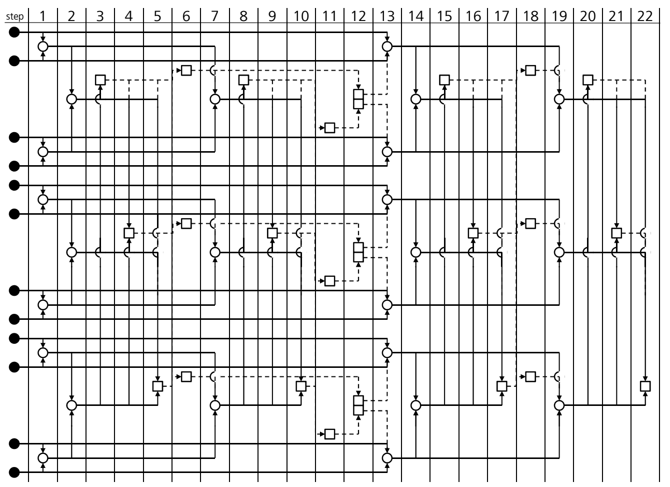


Fig. 3: Data flow graph of the SC decoder for the multi-kernel polar code defined by $G_{12} = T_2 \otimes T_2 \otimes T_3$. Circles represent LLR updates, squares represent PS updates.

III. Memory Management

Similarly to polar codes, it is possible to describe the SC decoding process of a multi-kernel polar code using a data flow graph, depicted in Figure 3 for the code generated by G_{12}. The data flow graph represents memory dependencies arising during the decoding, where circles and squares represent memory needed to store LLRs and PSs, respectively, and black circles represent channel LLRs. In particular, circles and squares identify the need for new memory allocation, while horizontal lines determine the number of time steps for which the values need to be stored. Thick lines represent LLR updates, while dotted lines identify operations involving partial sums, i.e. LLR updates when they merge with thick lines and PSs updates when they connect squares. The study of the data flow graph highlights the strong dependencies among data, along with a repetitive structure in the LLR update functions, and gives a precise order in the scheduling of the decoding operations. In general, the hardest constraint for the LLR update functions is given by the calculation of the necessary PSs. The memory usage patterns observed in Figure 3 can be found for code of any length, and can be exploited to develop a memory management framework as follows.

A. Memory Structure

The memory structures for a generic SC decoder for the multi-kernel polar code defined by G_{12} is presented in Figure 4, along with memory dependencies. We call Λ, Π and Υ the data structures used to respectively store LLRs, PSs and decoded bits. We define as Q the number of bits assigned to the representation of each internal LLR, while a partial sum and a decoded bit are, by definition, single-bit values. The proposed memory structure relies on the observation made in in [10] that memory requirements for polar codes decoding decrease as the stage index increases; we show that this phenomenon can be extended to multi-kernel polar codes.

The memory structure of multi-kernel polar codes decoder depends on the order of the kernels defining the transformation matrix $G_N = T_{p_1} \otimes \ldots \otimes T_{p_s}$, where s is the number of stages, i.e., the number of factors in the Kronecker product. LLRs can be stored in $s+1$ Q-bits vectors $\Lambda_0, \ldots, \Lambda_s$ of different lengths, i.e. with a different number of elements. The length of vector Λ_s is always 1, and stores the LLR of the currently decoded bit. The length of vector Λ_i is given by the product of the last $s-i$ kernel sizes, i.e., Λ_i has $p_{i+1} \cdot \ldots \cdot p_s$ entries. PSs are stored in s binary matrices $\Pi_1, \ldots, \Pi_s$ of different width and depth, depending on the decoding stage. The width of Π_i is given by the size of kernel p_i, while its depth is given by the product of the last $s-i$ kernel sizes, i.e., is given by $p_{i+1} \cdot \ldots \cdot p_s$, similarly to LLR vectors. The first matrix Π_s is an exception, since it has width $p_s - 1$. This is due to the fact that the last column of Π_s would be updated during the PS update phase of the decoding of the last bit. Since the PS update is executed right after the bit estimation, we skip this last PS update and we do not need to store the last column of Π_s. Finally, the decoded bits are stored in the binary vector Υ of length N.

B. Memory Update

```
Algorithm 1 SC Algorithm
1: Initialize $\Lambda, \Pi, \Upsilon$
2: for $i = 0 \ldots N-1$ do
3:     LLR update
4:     $u_i$ calculation
5:     PS update
6: end for
7: return $u$
```

Algorithm 1 depicts the logical flow of operations required by SC decoding. In this Section, we follow its schedule and first describe the update operations for the LLRs, then for the decoded bits u, and finally for the PSs. The memory update operations are performed by the kernel block. LLR vector Λ_0 is initially filled with the N LLRs extracted from the received symbols, while the rest of the memory is initialized to zero; we recall that the LLRs have to be permuted according to P_1 before the insertion in Λ_0. According to the SC algorithm, bits

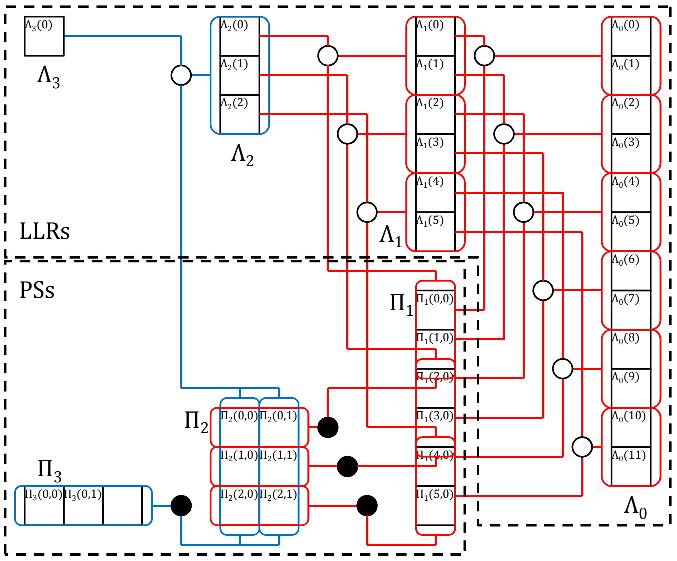


Fig. 4: Memory structure for $G_{12} = T_2 \otimes T_2 \otimes T_3$. White dots represent LLR updates, black dots represent PS updates. Red lines represent B_2 blocks, blue lines represent B_3 blocks.

are decoded sequentially, hence some of the memory structures are updated at every bit estimation; this update process is illustrated for the decoding of generic input bit u_i.

LLR update:
The update function (4) to be used in this phase depends on the index i of the decoded bit u_i, and it is selected using the mixed radix representation of i based on the kernels composing the transformation matrix of the code.

In a base p radix system, positive integers are represented as a finite sequence of digits smaller than p. A mixed radix system is a non-standard positional numeral system, generalizing classic radix system, in which the numerical base depends on the digit position. A well known example of a mixed radix numeral system is the one used to measure time in hours, minutes and seconds. We use the sequence $\langle p_1, \ldots p_s \rangle$ of the sizes of the kernels constructing the transformation matrix as the base of a finite mixed radix system representing the index i of the decoded bit u_i. According to this representation, any integer $i < N$ can be expressed as a vector of s digits $i = b_1^{(i)} \ldots b_s^{(i)}$, with $0 \leq b_j^{(i)} < p_j$ and

$$i = b_s^{(i)} + \sum_{j=1}^{s-1} b_j^{(i)} \cdot (p_{j+1} \cdot \ldots \cdot p_s). \tag{5}$$

The mixed radix representation of the decoded bits indices for $G_{12} = T_2 \otimes T_2 \otimes T_3$ is given by:

i	0	1	2	3	4	5	6	7	8	9	10	11
$b_1^{(i)}$	0	0	0	0	0	0	1	1	1	1	1	1
$b_2^{(i)}$	0	0	0	1	1	1	0	0	0	1	1	1
$b_3^{(i)}$	0	1	2	0	1	2	0	1	2	0	1	2

The update of LLR vectors proceeds from right to left in Figure 4. Starting from Λ_1, all the vectors are updated using the previous LLR vector and the present PS matrix as input; in general, vector Λ_j is updated using vector Λ_{j-1} and the partial sums stored in Π_j. The LLR update rule to be used in the update of Λ_j is selected using the mixed radix representation of i, and more precisely the LLR update rule $f^{p_j}_{b^{(i)}_j}$ is used. This method is an extension of the method proposed in [10] for the scheduling of f and g functions in the decoding of polar codes. Each entry $\Lambda_j(k)$ of the LLR vector is calculated as

$$\begin{aligned}\Lambda_j(k) = f^{p_j}_{b^{(i)}_j}(&\Lambda_{j-1}(k\cdot p_j),\dots,\Lambda_{j-1}((k+1)\cdot p_j-1),\\ &\Pi_j(k,0),\dots,\Pi_j(k,b^{(i)}_j)).\end{aligned} \tag{6}$$

The update operations of a vector Λ_j can be run in parallel to reduce latency using up to $p_{j+1}\cdot\ldots\cdot p_s$ kernel blocks.

Using the proposed LLR update algorithm, s LLR vectors, i.e. from Λ_1 to Λ_s, are updated for every decoded bit u_i. However, the data flow presented in Figure 3 shows that the number of vectors to be updated actually depends on the index i of the decoded bit. A closer look to the mixed radix representation table suggests the reason of this scheduling: in fact, the mixed radix representations of two consecutive numbers differ only on the right of the position of the rightmost nonzero element of the second number. In practice, given $i-1 = b^{(i-1)}_1 \dots b^{(i-1)}_s$, if it exists an index z such that $b^{(i)}_z \neq 0$ and $b^{(i)}_j = 0$ for all $j > z$, we have that $b^{(i)}_j = b^{(i-1)}_j$ for all $j < z$. As a consequence, to decode the bit u_i it is not necessary to update the vectors Λ_j with indices $j < z$, and the update can be run starting from Λ_z. Of course, for the case $i = 0$ all the vectors have to be updated. This acceleration technique is a generalization of the one proposed in [15] for polar codes, and halves the number of vectors updates.

This property allows a further simplification of the LLR update algorithm. We have seen that the LLR update is run starting from Λ_z with z such that $b^{(i)}_z \neq 0$ and $b^{(i)}_j = 0$ for all $j > z$. This means that the vector Λ_z is updated using function $f^{p_j}_{b^{(i)}_z}$, while all the other vectors are updated using a function of the form $f^{p_j}_0$. This means that it is only necessary to find the subscript of the first LLR update function, while the other ones all have subscript 0.

u_i estimation:
If $i \in \mathcal{F}$, i.e. it belongs to the frozen set, its value is known to be zero, hence $\Upsilon(i) = 0$. Otherwise, i.e. if $i \notin \mathcal{F}$, the value decoded bit is decided by hard decision on its LLR. After the LLR update phase, the LLR of the bit u_i will be copied in Λ_s as explained in next paragraph. In our implementation, negative LLRs represent the bit 1, while positive LLRs represent the bit 0. Through hard decision, we set $\Upsilon(i) = \frac{\mathrm{sgn}(\Lambda_s(0))+1}{2}$. To sum up, we have that

$$\Upsilon(i) = \begin{cases} 0 & \text{if } i \in \mathcal{F} \\ \frac{\mathrm{sgn}(\Lambda_s(0))+1}{2} & \text{if } i \notin \mathcal{F} \end{cases} \tag{7}$$

PS update:
PS matrices are updated in decreasing order starting from Π_s. Inside each matrix, the entries update is performed per columns, in increasing order starting from the first column. When the last column of a matrix is filled, a column of the next matrix is updated. Similarly to LLRs, the update function depends on the mixed radix representation of index i; in particular, the number of matrices to be updated is given by the number of consecutive digits of the mixed radix representation of i with the highest symbol admitted by the radix, counting from the last digit.

Algorithm 2 LLR update

```
1:  $n = 1$
2:  if i = 0 then
3:      break
4:  else
5:      for $z = s \dots 1$ do
6:          if $i \mod p_z \neq 0$ then
7:              break
8:          end if
9:          $i = \frac{i}{p_z}$
10:         $n = n \cdot p_z$
11:     end for
12: end if
13: $b = i \mod p_z$
14: for $k = 0 \dots n-1$ do
15:     $\Lambda_z(k) = f^{p_z}_b(\Lambda_{z-1}(k\cdot p_z),\dots,\Lambda_{z-1}((k+1)\cdot p_z - 1),\Pi_z(k,0),\dots,\Pi_z(k,b))$
16: end for
17: for $j = z+1 \dots z$ do
18:     $n = n \cdot p_j$
19:     for $k = 0 \dots n-1$ do
20:         $\Lambda_j(k) = f^{p_j}_0(\Lambda_{j-1}(k\cdot p_j),\dots,\Lambda_{j-1}((k+1)\cdot p_j - 1))$
21:     end for
22: end for
```

Algorithm 3 u_i calculation

```
1: if $i \in \mathcal{F}$ then
2:     $\Upsilon(i) = 0$
3: else
4:     $\Upsilon(i) = \frac{\mathrm{sgn}(\Lambda_s(0))+1}{2}$
5: end if
```

Update always starts from the last PS matrix Π_s, that is a row vector of width p_s. The value of the decoded bit u_i is copied in the column $b^{(i)}_s$ of the matrix, i.e., $\Pi_s(0, b^{(i)}_s) = \Upsilon(i)$. When $b_s = p_s - 1$, the last column of the matrix has been filled, and the column $b^{(i)}_{s-1}$ of the matrix Π_{s-1} is updated, otherwise the update process ends. In general, if $b^{(i)}_j = p_j - 1$ for all $j > z$ and $b^{(i)}_z < p_z - 1$, the matrices $\Pi_s, \Pi_{s-1}, \dots, \Pi_z$ are going to be updated. When the last column of matrix Π_j is filled, i.e. when $b^{(i)}_j = p_j - 1$, then the column b_{j-1} of matrix Π_{j-1} has to be updated. In this case, each row of Π_j is used to update the column $b^{(i)}_{j-1}$ of Π_{j-1} as $[\Pi_{j-1}(k\cdot p_j, b^{(i)}_{j-1}),\dots,\Pi_{j-1}((k+1)\cdot p_j - 1, b^{(i)}_{j-1})] = [\Pi_j(k,0),\dots,\Pi_j(k,p_j-1)]\cdot T_{p_j}$ for $k = 0,\dots,p_{j+1}\cdot\ldots\cdot p_s - 1$. If we call T^k_p the vector formed by the k-th column of the kernel matrix T_p, the update rule for the PSs can be rewritten as

$$\Pi_j(k, b^{(i)}_j) = \Pi_j\left(\left\lfloor\frac{k}{p_{j-1}}\right\rfloor, -\right)\cdot T^c_{p_j} \tag{8}$$

for $k = 0, \ldots, p_{j+1} \cdot \ldots \cdot p_s$, where $\Pi_j(k,-)$ represents the k-th row of Π_j and $c = (k \mod p_{j+1}) + 1$. As an exception, the PS update step is not executed for the last decoded bit u_{N-1}, since this phase would have been executed after the decoding of the last bit and it would be pointless.

Algorithm 4 PS update

```
1:  $n = 1$
2:  **if** $i = N - 1$ **then**
3:      return
4:  **end if**
5:  **for** $j = s - 1 \ldots 1$ **do**
6:      **if** $i + 1 \mod p_{j+1} \neq 0$ **then**
7:          return
8:      **end if**
9:      $i = \frac{i+1}{p_{j+1}} - 1$
10:     $n = n \cdot p_{j+1}$
11:     $b = i \mod p_j$
12:     **for** $k = 0 \ldots n - 1$ **do**
13:         $c = (k \mod p_{j+1}) + 1$
14:         $\Pi_j(k,b) = \Pi_{j+1}\left(\left\lfloor \frac{k}{p_{j-1}} \right\rfloor, -\right) \cdot T^c_{p_j}$
15:     **end for**
16: **end for**
```

IV. Analysis and Conclusions

The proposed memory structure allows to limit the memory requirement of a multi-kernel polar decoder. In fact, a naïve memory management of the SC decoder for a multi-kernel polar codes with transformation matrix $G_N = T_{p_1} \otimes \ldots \otimes T_{p_s}$ requires to store all the LLRs and the PSs depicted in the Tanner graph of the code. As a consequence, $\mathcal{M}^{LLR} = N \cdot (s+1)$ LLRs and $\mathcal{M}^{PS} = N \cdot s$ PSs, with $N = p_1 \cdot \ldots \cdot p_s$, have to be stored, with space complexity $O(sN)$. The memory requirement is hence linearly dependent on both the code length N and the number of kernels s.

In the proposed memory structure, every LLR vector Λ_i with $i \leq 1$ stores $\frac{N}{p_1 \cdot \ldots \cdot p_i}$ LLRs, while the first vector Λ_0 stores the N LLRs derived from the received signals. In total, for the proposed memory framework

$$\begin{aligned} \mathcal{M}^{LLR}_{prop} &= N + \frac{N}{p_1} + \frac{N}{p_1 \cdot p_2} + \ldots + 1 = \\ &= (\ldots((p_1 + 1) \cdot p_2 + 1) \cdot \ldots) \cdot p_s + 1 \end{aligned} \tag{9}$$

LLRs have to be stored. Similarly, every PS matrix Π_i with $i > 1$ stores $\frac{N}{p_1 \cdot \ldots \cdot p_i} \cdot p_i = \frac{N}{p_1 \cdot \ldots \cdot p_{i-1}}$ partial sums, while Π_1 stores $\frac{N}{p_1} \cdot (p_s - 1)$ PSs. Then, the total number of PSs is

$$\begin{aligned} \mathcal{M}^{PS}_{prop} &= \frac{N}{p_1} \cdot (p_s - 1) + \frac{N}{p_1 \cdot p_2} + \ldots + p_s = \\ &= (\ldots((p_1 \cdot p_2 + 1) \cdot p_3 + 1) \cdot \ldots) \cdot p_s. \end{aligned} \tag{10}$$

By construction, we have that $\mathcal{M}^{PS}_{prop} \leq N \leq \mathcal{M}^{LLR}_{prop} < 2N$, hence the space complexity for both LLRs and PSs is reduced to $O(N)$. A comparison between the memory requirements for the proposed memory structure and the naïve one involving only kernels of sizes 2 and 3 is presented here:

N	12	72	144	384	972
$\mathcal{M}^{LLR}_{prop}$	22	139	283	766	1822
$\mathcal{M}^{LLR}$	48	432	1008	3456	7776
$\mathcal{M}^{PS}_{prop}$	15	102	210	573	1335
$\mathcal{M}^{PS}$	36	360	864	3072	6804

The memory requirement reduction enabled by the proposed memory structure is remarkable. This proves that multi-kernel polar codes can be used as a valid alternative to punctured polar codes in terms of memory complexity. Given the similarities between polar codes and multi-kernel polar codes, it is straightforward to apply the proposed memory structure to list or simplified SC decoders. Finally, the proposed implementation can be easily transposed to hardware, reducing the complexity of an ASIC or FPGA dedicated architecture.

References

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

[2] G. Liva, L. Gaudio, T. Ninacs, and T. Jerkovits, "Code design for short blocks: A survey," in *arXiv preprint*, arXiv:1610.00873, Oct. 2016.

[3] V. Bioglio, F. Gabry, and I. Land, "Low-complexity puncturing and shortening of polar codes," in *IEEE Wireless Communications and Networking Conference (WCNC)*, San Francisco, CA, USA, March 2017.

[4] F. Gabry, V. Bioglio, I. Land, and J.-C. Belfiore, "Multi-kernel construction of polar codes," in *IEEE International Conference on Communications (ICC)*, Paris, France, May 2017.

[5] B. L. Gal, C. Leroux, and C. Jego, "Software polar decoder on an embedded processor," in *IEEE Workshop on Signal Processing Systems (SiPS)*, Belfast, UK, Oct 2014.

[6] Y. Shen, C. Zhang, J. Yang, S. Zhang, and X. You, "Low-latency software successive cancellation list polar decoder using stage-located copy," in *IEEE International Conference on Digital Signal Processing (DSP)*, Beijing, China, Oct 2016.

[7] C. Leroux, I. Tal, A. Vardy, and W. J. Gross, "Hardware architectures for successive cancellation decoding of polar codes," in *Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on*, Prague, Czech Republic, May 2011.

[8] S. A. Hashemi, C. Condo, F. Ercan, and W. J. Gross, "Memory-efficient polar decoders," *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, vol. 7, no. 4, pp. 604–615, Dec 2017.

[9] F. Ercan, C. Condo, S. A. Hashemi, and W. J. Gross, "On error-correction performance and implementation of polar code list decoders for 5G," in *Allerton Conference on Communication, Control, and Computing*, Monticello, IL, USA, Oct 2017.

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

[11] A. Balatsoukas-Stimming, A. J. Raymond, W. J. Gross, and A. Burg, "Hardware architecture for list successive cancellation decoding of polar codes," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 61, no. 8, pp. 609–613, 2014.

[12] G. Coppolino, C. Condo, G. Masera, and W. J. Gross, "A multi-kernel multi-code polar decoder architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, pp. 1–10, 2018.

[13] N. Presman, O. Shapira, S. Litsyn, T. Etzion, and A. Vardy, "Binary polarization kernels from code decompositions," *IEEE Transactions on Information Theory*, vol. 61, no. 5, pp. 2227–2239, May 2015.

[14] V. Bioglio, F. Gabry, I. Land, and J.-C. Belfiore, "Minimum-distance based construction of multi-kernel polar codes," in *IEEE Global Communications Conference (GLOBECOM)*, Singapore, Dec. 2017.

[15] A. Balatsoukas-Stimming, M. Bastani Parizi, and A. Burg, "LLR-based successive cancellation list decoding of polar codes," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Florence, Italy, May 2014.

[16] V. Bioglio and I. Land, "On the marginalization of polarizing kernels," in *International Symposium on Turbo Codes & Iterative Information*, Hong Kong, December 2018.