Herrmann and Wehn *J Wireless Com Network* (2022) 2022:90
https://doi.org/10.1186/s13638-022-02169-5

EURASIP Journal on Wireless Communications and Networking

RESEARCH Open Access

Beyond 100 Gbit/s Pipeline Decoders for Spatially Coupled LDPC Codes

Matthias Herrmann* and Norbert Wehn

*Correspondence:
herrmann@eit.uni-kl.de

Microelectronic Systems Design Research Group, Technische Universität Kaiserslautern, Kaiserslautern, Germany

Abstract

Low-density parity-check (LDPC) codes are a well-established class of forward error correction codes that provide excellent error correction performance for large code block sizes. However, for throughputs toward 1 Tbit/s, as expected for B5G systems, state-of-the-art LDPC soft decoders are restricted to short code block sizes of several hundred to thousands of bits due to routing congestion challenges, limiting the overall communications performance of the transmission system. Spatially coupled LDPC (SC-LDPC) codes and respective sliding window decoding methods show the potential to overcome these block size restrictions. However, in contrast to conventional LDPC codes little literature exists on the efficient hardware implementation of respective high-throughput decoders. In this work, we present the first in-depth investigation on the implementation of SC-LDPC decoders for throughputs beyond 100 Gbit/s. For an $N = 51328, R = 0.8$ terminated SC-LDPC code with sub-block size $c = 640$ and coupling width $m_s = 1$, we explore various design trade-offs, including row- and column-wise decoding, non-overlapping and overlapping window scheduling, and processor pipelining. To the best of our knowledge, this is the first description of a column-wise SC-LDPC decoding architecture in the literature. We complement the algorithmic investigation with the virtual silicon implementation of all presented decoders in a 22nm FD-SOI technology.

Keywords: Spatial coupling, LDPC codes and VLSI implementation, High-throughput decoding

1 Introduction

Forward error correction (FEC) is a key technology of modern communication systems. The capability of detecting and correcting transmission errors has largely contributed to the progress in data rates, roundtrip latencies, and number of connected devices in cellular mobile communication systems. And the demand does not cease. Beyond-5G systems target data rates toward 1 Tbit/s, posing new and fundamental challenges to the design and implementation of FEC systems [1]. Low-density parity-check (LDPC) codes are among the most powerful and widely used FEC schemes. Since their discovery by Gallager in 1962 [2], innovations in both the code and the decoder design have opened this code class for a wide range of practical applications. Today, LDPC codes are part of many modern communication standards, like DVB-S2x, Wi-Fi, and 3GPP 5G-NR.

However, achieving 1-2 orders of magnitude higher throughputs than today's fastest standards without sacrificing performance remains a great challenge, mainly due to area, power, and power density restrictions in the decoder implementation.

LDPC decoding is based on an iterative exchange of messages between the variable nodes and the check nodes in the Tanner graph of the code. To achieve good error correction performance, the Tanner graph must be large, of low density, and without short cycles causing the graph to be highly unstructured and without distinct regularity. These properties, however, challenge an efficient and high-throughput decoder implementation that requires locality to minimize the cost of data transfers and regularity to achieve large parallelism [3]. This fact manifests in particular in an increased wiring complexity and routing congestions for large code block sizes resulting in low area utilization, poor timing, and increased power consumption of the respective decoding hardware. As a consequence, state-of-the-art high-throughput decoders are limited to small code block sizes of 1000–2000 bits, e.g., [4–6], which limits the overall performance of the FEC system.

Spatial coupling of LDPC codes is a promising approach to overcome these block size limitations. Spatially coupled LDPC (SC-LDPC) codes, initially introduced as LDPC convolutional codes (LDPC-CC) [7], are constructed from a set of small LDPC codes that are coupled together to form a chain of local sub-codes. In this way, a code of almost any length can be generated. Similarly, a respective decoder can be constructed by chaining together multiple sub-decoders, of which each operates on a much smaller sub-block. In this way, SC-LDPC codes have the potential to combine good error correction performance with high-throughput decoding. This property is of particular importance in the context of beyond-5G/6G high-THz communications systems that aim at data rates of 1 Tbit/s. Respective use cases show significant variations in bit error rate (BER) requirements depending on whether they fall into infrastructure or end-user domains. For instance, IEEE 802.15.3d standard sets very stringent BER requirements of 10^{-12} for infrastructure-type use cases such as wireless backhaul/fronthaul and data centers, in contrast to relatively relaxed BER requirement of 10^{-6} for close-proximity communications with applications in personal-area networks [8]. SC-LDPC codes can cover this broad range of use cases as they exhibit good performance in both the waterfall and the error floor region of the BER curve [9].

In contrast to classical LDPC block codes, not so much research exists on the implementation of efficient, high-throughput SC-LDPC decoders. Essentially, we can distinguish three candidate architectures in the literature:

- The row-layered pipeline decoder (RLPD) [7],
- The row-compact pipeline decoder (RCPD) [10] and
- The full-parallel window decoder (FPWD) [11].

The RLPD achieves a fast convergence but supposedly suffers from a long initial decoding delay and high storage requirements [10], whereas the RCPD and the FPWD exhibit relatively smaller decoding delay and less storage requirements but also a slower convergence [10, 12]. However, these characterizations provide little information about the efficiency of respective decoding hardware, which is evaluated according to implementation

metrics like achievable frequency, latency, area, and power consumption. In particular, for data transfer dominated circuits with complex signal routing like highly parallel LDPC decoders, it is difficult to fully assess the implications of design decisions on the algorithmic level on the implementation efficiency.

In this work, we therefore provide the first comparative investigation of various high-throughput SC-LDPC decoding architectures down to the silicon level. Our investigation focuses on the $N = 51328$, $R = 0.8$ terminated EPIC SC-LDPC code with sub-block size $c = 640$ and coupling width $m_s = 1$ [13]. We explore various design trade-offs, including row- and column-wise decoding, non-overlapping and overlapping window scheduling, and processor pipelining. To the best of our knowledge, we present the first description of a column-wise SC-LDPC decoding architecture in the literature.

2 Preliminaries

2.1 Notation

In the following, we use italic letters, e.g., x, for the representation of scalars, bold lowercase letters, e.g., $\mathbf{x}$, for the representation of vectors, and bold uppercase letters, e.g., $\mathbf{X}$, for the representation of matrices. $\mathbb{N}_0$ denotes the set of positive integers, including the 0-element, $\mathbb{Z}$ the set of integers, $\mathbb{R}$ the set of real numbers, and $\mathbb{F}_2$ the Galois field 2. $\log_2(x)$ denotes the base 2 logarithm of variable x, and $\lceil x \rceil$ is the least integer greater than or equal to x.

2.2 System model

We define a system for the continuous transmission of data blocks, as depicted in Fig. 1. An information source generates a stream, or sequence, of information blocks $\mathbf{u}_{[-\infty,\infty]} = [\mathbf{u}_{-\infty}, ..., \mathbf{u}_{\infty}]$, with each information block being composed of b bits, i.e., $\mathbf{u}_t = [u_t(1), ..., u_t(b)], t \in \mathbb{Z}$ and $u_t(\cdot) \in \mathbb{F}_2$. An encoder maps this sequence of information blocks on a sequence of code blocks $\mathbf{v}_{[-\infty,\infty]} = [\mathbf{v}_{-\infty}, ..., \mathbf{v}_{\infty}]$ with $\mathbf{v}_t = [v_t(1), ..., v_t(c)]$, $c > b$ and $v_t(\cdot) \in \mathbb{F}_2$. After modulation, transmission over a noisy channel, and subsequent bit-level demodulation, the decoder receives a sequence of blocks $\mathbf{y}_{[-\infty,\infty]}$ each comprising c log-likelihood ratio (LLR) values, i.e., $\mathbf{y}_t = [y_t(1), ..., y_t(c)]$ and $y_t(\cdot) \in \mathbb{R}$. Finally, a decoder provides an estimation $\hat{\mathbf{u}}_{[-\infty,\infty]}$ on the initially transmitted information, with $\hat{\mathbf{u}}_t = [\hat{u}_t(1), ..., \hat{u}_t(b)]$ and $\hat{u}_t(\cdot) \in \mathbb{F}_2$.

2.3 SC-LDPC codes

We define an $R = b/c$ SC-LDPC code as the set of code sequences $\mathbf{v}_{[-\infty,\infty]}$ satisfying

$$\mathbf{v}_{[-\infty,\infty]} \cdot \mathbf{H}^T_{[-\infty,\infty]} = \mathbf{0}, \tag{1}$$

where

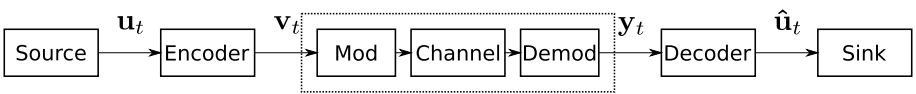


Fig. 1 System model

$$\mathbf{H}_{[-\infty,\infty]} = \begin{bmatrix} \ddots & & & & & & \\ & \mathbf{H}_0(t-1) & & & & & \\ \ddots & \vdots & \mathbf{H}_0(t) & & & & \\ & \mathbf{H}_{m_s}(t-1) & \vdots & \ddots & & & \\ & & \mathbf{H}_{m_s}(t) & \dots & \mathbf{H}_0(t+m_s) & & \\ & & & \ddots & \vdots & \ddots & \\ & & & & \mathbf{H}_{m_s}(t+m_s) & & \\ & & & & & \ddots & \end{bmatrix}$$

is also denoted as the parity-check matrix or syndrome former matrix of the SC-LDPC code. The elements $\mathbf{H}_i(t) \neq \mathbf{0}$ are binary, full rank sub-matrices of dimension $(c-b) \times c$. From Eq. 1, it follows that each $\mathbf{v}_t$ satisfies

$$[\mathbf{v}_{t-i}, \dots, \mathbf{v}_t, \dots, \mathbf{v}_{t+j}] \cdot [\mathbf{H}_{m_s}(t-i), \dots, \mathbf{H}_0(t+j)]^T = 0, \{i,j \in \mathbb{N}_0 \mid i+j = m_s\} \quad (2)$$

and is thus coupled to m_s preceding and ascending code blocks. Consequently, m_s is also referred to as the coupling width or the memory of the code, (m_s+1) as the constraint length.

We define the parity-check matrix of a terminated SC-LDPC code of finite length L as $\mathbf{H}_{[1,L]}$. For simplification, we consider in this work only time-invariant codes, i.e., $\mathbf{H}_i(t) = \mathbf{H}_i(t')$ for $i = 0, \dots, m_s$ and $t, t' \in \mathbb{Z}$. For a detailed description of SC-LDPC codes, we refer to [7, 10] and [9].

2.4 Window decoding of SC-LDPC codes

SC-LDPC codes are decoded with message-passing algorithms, i.e., an exchange of probabilistic messages between variable and check nodes in the corresponding Tanner graph of the code. The Tanner graph can be directly derived from $\mathbf{H}_{[1,L]}$, and the decoding can be performed similarly to a conventional LDPC block code by repeatedly updating the respective variable nodes and check nodes. This block scheduling, however, does not take advantage of the limited memory of the code and the consequent diagonal band structure of $\mathbf{H}_{[1,L]}$. Instead, it is more efficient to use a sliding window scheduling. Here, the decoding is not performed simultaneously on the full graph corresponding to $\mathbf{H}_{[1,L]}$, but time-delayed on multiple sub-graphs each defined by a decoding window $\mathcal{W}(t)$ of length W. The decoding window can be considered as the Tanner graph corresponding to a sub-matrix of $\mathbf{H}_{[1,L]}$ comprising W rows and $W + m_s$ columns. Throughout the decoding process, the window traverses $\mathbf{H}_{[1,L]}$ from the upper left to the lower right, moving by one sub-block each time step t. The advantage of a window decoding schedule is a significantly lower structural latency than a block schedule, as the decoder can start the decoding right after receiving the first sub-block $\mathbf{y}_1$. Furthermore, since the window length is typically much smaller than the block length, i.e., $W \ll L$, the memory requirements are far less compared to a conventional LDPC block decoder of similar length.

3 State-of-the-art high-throughput decoders

From an implementation perspective, a sliding window decoder is subject to similar constraints as a conventional LDPC block decoder. For a large window size W, which in this analogy corresponds to a large block size N, routing congestions limit the achievable throughput. Reducing W, on the other hand, reduces the performance of the decoder. This performance loss can be partially counterbalanced by increasing the number of iterations on the window, but this again increases the logic critical path.

In [7], the authors propose to split the window into multiple sub-windows, each comprising only a single row of $\mathbf{H}_{[1,L]}$, in the following denoted as a row-layer. These sub-windows are then processed individually on multiple processors in parallel. Each processor performs only a single iteration involving $(c - b)$ check nodes and $(m_s + 1) \cdot c$ variable nodes. Moving the window to the next position is similar to shifting the processed data from one processor to the next. In this way, the processors form a pipeline. It is hence commonly referred to as pipeline decoder. The initially proposed architecture requires a spacing between simultaneously processed layers of m_s to avoid memory hazards. To better differentiate it from other decoding architectures, we will refer to it in the following as RLPD. An application-specific integrated circuit (ASIC) implementation of an RLPD was presented in [14]. For a (491,3,6) LDPC-CC, the decoder achieves a throughput of 2.37 Gbit/s in 90 nm CMOS technology. Note that the achievable throughput for a LDPC-CC is much lower due to code's serial structure, that limits the achievable hardware parallelism.

The structural latency, which also impacts the memory requirements of the RLPD, is proportional to the number of processors (iterations) I and the constraint length $(m_s + 1)$. This can become a significant issue for codes with large m_s and that require many iterations. In [10], the authors propose a so-called compact pipeline decoder with overlapping row-layers. The overlapping regions reduce the decoding window's size and thus the decoder's latency and memory. In analogy to the RLPD, we refer to this architecture as RCPD. However, a drawback of this architecture is a slower convergence of the decoding. The reason for this is a simultaneous update of the variable node in the overlapping regions, similar to a flooding schedule. With the size of the overlap, latency and convergence of the decoder can be weighed against one another. In [15], the authors implemented an RCPD for a (215,3,6) LDPC-CC in a 65 nm technology. The decoder achieves a throughput of 7.72 Gbit/s.

Another approach for low decoding latency in combination with high throughput is the FPWD [11]. Like the RLPD and RCPD, the FPWD comprises multiple processors arranged in a pipeline. However, here the processors operate on small overlapping sub-windows of at least $W = m_s + 1$ using a flooding schedule. In the course of the decoding, the processors exchange extrinsic messages. An implementation of a FPWD was presented in [12]. The decoder achieves a throughput of 336 Gbit/s in a 22 nm technology and is currently the fastest SC-LDPC decoder in the literature.

4 Proposed pipeline decoder architectures

The decoders presented in the previous chapter, i.e., the RLPD, the RCPD, and the FPWD, are individual decoding solutions for SC-LDPC codes with the advantages and disadvantages discussed. In this chapter, we generalize several of the presented

state-of-the-art concepts, like the layered/compact window schedules [10] and overlapping decoding windows [12], and combine them with row- and column-wise decoding algorithms. This systematic approach leads to a new, more abstract perspective on state-of-the-art decoders. For example, the FPWD can then be viewed, as a particular column-wise decoder that uses a compact processing schedule.

Based on our methodology, we propose new pipeline decoder architectures, which are described in the following. The description resembles a bottom-up approach starting with the description of the row and column processors that constitute the fundamental building blocks of the respective decoders. We then show how the different window schedules can be implemented by the different interconnection of these elementary components. For simplicity, we assume in the following a fixed coupling width of $m_s = 1$. Furthermore, we focus on Min-Sum (MS) decoding. However, the presented concepts can be also applied to larger values of m_s and other decoding algorithms.

4.1 Processors

For the processor design, we utilize the node-splitting concept [12] that was initially introduced for the check nodes in the FPWD and apply it to the variable nodes of a row-wise decoder and the check nodes of a column-wise decoder. Furthermore, the on-demand variable node activation (OVA) schedule for the row-wise decoding is extended to column-wise decoding.

4.1.1 Node splitting

For $m_s = 1$, a row-layer of $\mathbf{H}_{[1,L]}$ at time instance t is

$$\mathbf{H}_{\mathrm{r}}(t) = \left[\mathbf{H}_1(t-1)\ \ \mathbf{H}_0(t)\right], \tag{3}$$

and the corresponding sub-graph is composed of $N_{\mathrm{r}} = 2 \cdot c$ variable nodes and $M_{\mathrm{r}} = c - b$ check nodes. By transforming the SC-LDPC factor graph [16], the variable nodes corresponding to a row-layer can be considered the partial variable nodes of a group of coupling nodes. For layer t, the respective coupling nodes connect the variable nodes corresponding to $\mathbf{H}_0(t)$ to the variable nodes corresponding to $\mathbf{H}_1(t)$ in layer $t+1$. This principle is illustrated in Fig. 2. Similarly, we can merge the coupling nodes of layer t with the partial nodes of $\mathbf{H}_0(t)$ such that the c leftmost nodes of layer t are directly

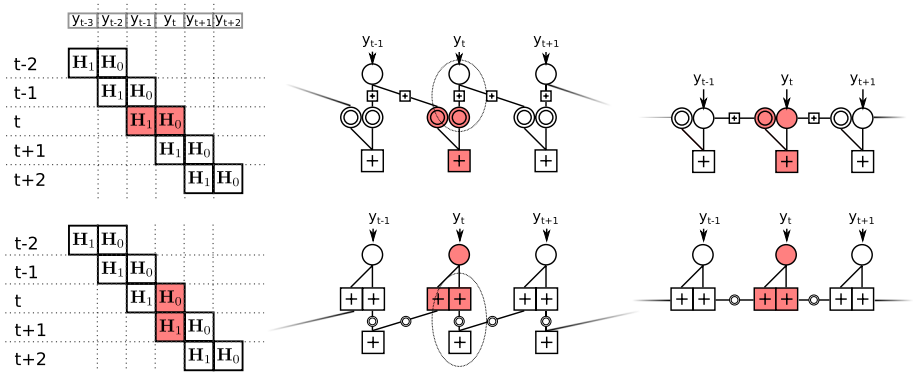


Fig. 2 Illustration of the node splitting for row- and column-wise decoding

connected via a single edge to the c rightmost nodes of layer $t+1$. Likewise, the c rightmost nodes of layer t are connected to the c leftmost nodes of layer $t-1$. This concept can be similarly applied to a column of $\mathbf{H}_{[1,L]}$

$$\mathbf{H}_{\mathrm{c}}(t) = \begin{bmatrix} \mathbf{H_0}(t) \\ \mathbf{H_1}(t) \end{bmatrix}, \tag{4}$$

corresponding to $N_{\mathrm{c}} = c$ variable and $M_{\mathrm{c}} = 2 \cdot (c-b)$ check nodes. For simplicity, we assume that the SC-LDPC is regular with d_v and d_c and that $d_v(\mathbf{H}_{\mathrm{r}}) = d_v/2$ and $d_c(\mathbf{H}_{\mathrm{c}}) = d_c/2$, i.e., the nodes are split exactly in half.

4.1.2 On-demand variable and check node activation

In the standard flooding schedule commonly used in the decoding of LDPC block codes, an iteration starts with the update of the check nodes (CNs) and ends with the update of the variable nodes (VNs). We denote this in the following as CN–VN iteration, see Algorithm 1. It was shown in [10] that when employing the flooding schedule in a row-wise pipeline decoder, the processors do not make use of the most recent information in the processing pipeline. The authors, therefore, propose an OVA schedule that provides faster convergence for row-wise pipeline decoders. This OVA schedule resembles a sub-iteration in the layered decoding schedule for LDPC block codes [17]. Here, the iteration, in the following denoted as VN–VN iteration, starts with the calculation of new extrinsic messages for the check nodes (line 8, Alg. 1) and ends with the calculation of the a posteriori probability (APP) values (line 7, Alg. 1).

Similarly, we propose an on-demand check node activation (OCA) schedule for column-wise pipeline decoders. The corresponding CN–CN iteration starts at line 4 of Alg. 1 and ends at line 3. The first step in the CN–CN iteration is thus to compute the extrinsic messages for the VNs based on the respective signs and minimum values. Then, the VNs are updated and new signs and minimum values are eventually calculated. Between two decoding iterations, the extrinsic messages are represented only using the first and second absolute minimum min_0 and min_1, the edge index of the first minimum idx_0, and the $d_c/2$ signs of the output messages.

The node splitting requires an exchange of messages between neighboring windows/processors. Therefore, we extend the VN–VN and CN–CN iterations by additional steps for the message exchange.

- Before the iteration on the respective sub-code, the local variable, respectively, check nodes are updated with the incoming messages from the neighboring windows/processors. For the row-wise decoder by adding the exchange messages to the respective APP value, for the column-wise decoder by sorting the exchange messages into the list of minima and updating the sign of the respective check node.
- After the VN–VN/CN–CN iteration, the outgoing exchange messages for the neighboring windows/processors are generated. For the row-wise decoder by subtracting the incoming exchange message from the updated APP values, and for the column-wise decoder by multiplication of the minimum value with the check node sign.

Algorithm 2 and Algorithm 3 summarize the resulting processing algorithms for a row and a column layer.

Algorithm 1 CN-VN decoding iteration

for every check node **do**
  find (min_0, min_1, idx_0) of incoming edges
  sign := $\prod$ sign(incoming edges)
  map (min_0, min_1, idx_0, sign) to outgoing edges
end for
for every variable node **do**
  APP := $\sum$ incoming edges + channel value
  outgoing edge i := APP - incoming edge i
end for

Algorithm 2 Process row layer

Input: $(\text{APP}, \text{extr\_msgs}[d_v/2], \text{exch\_msg})[c]$
Output: $(\text{APP}', \text{extr\_msgs}'[d_v/2], \text{exch\_msg}')[c]$
APP += exch_msg
Perform VN to VN decoding iteration:
$(\text{APP}, \text{extr\_msgs}) \rightarrow (\text{APP}', \text{extr\_msgs}')$
$\text{exch\_msg}' := \text{APP}' - \text{exch\_msg}$
$\text{APP}' -= \text{exch\_msg}$

Algorithm 3 Process column layer

Input: $(\text{min}_0, \text{min}_1, \text{idx}_0, \text{sign}, \text{signs}[d_c/2], \text{exch\_msg})[c-b]$
Output: $(\text{min}'_0, \text{min}'_1, \text{idx}'_0, \text{sign}', \text{signs}'[d_c/2], \text{exch\_msg}')[c-b]$
Update all $(c-b)$ check node input messages using Algorithm 4
Perform CN to CN decoding iteration:
$(\text{min}_0, \text{min}_1, \text{idx}_0, \text{sign}) \rightarrow (\text{min}'_0, \text{min}'_1, \text{idx}'_0, \text{sign}')$
$(\text{signs}) \rightarrow (\text{signs}')$
$\text{exch\_msg}' := \text{sign}' \cdot \text{min}'_0$

Algorithm 4 Update Minima

Input: min_0, min_1, idx_0, $\text{signs}[d_c/2]$, exch_msg
Output: min'_0, min'_1, idx'_0, $\text{signs}'[d_c/2]$
Update minima :
if $(\text{abs}(\text{exch\_msg}) \leq \text{min}_0)$ **then**
  $\text{min}'_0 := \text{abs}(\text{exch\_msg})$; $\text{min}'_1 := \text{abs}(\text{exch\_msg})$
else if $(\text{min}_0 < \text{abs}(\text{exch\_msg}) \leq \text{min}_1)$ **then**
  $\text{min}'_0 := \text{min}_0$; $\text{min}'_1 := \text{abs}(\text{exch\_msg})$
else
  $\text{min}'_0 := \text{min}_0$; $\text{min}'_1 := \text{min}_1$
end if
Update signs and min index :
for $i = 0$ to $(d_c/2 - 1)$ **do**
  $\text{signs}'[i] := \text{signs}[i] \cdot \text{sign}(\text{exch\_msg})$
end for
$\text{idx}'_0 := \text{idx}_0$

4.1.3 Processor architecture

The row and column processors apply full parallelism on the node and edge level. For a description of respective variable node functional units (VFUs) and check node functional units (CFUs), we refer to [18] and [19]. The architecture of a full-parallel row processor that implements Algorithm 2 is depicted in Fig. 3a. The incoming right-to-left (R2L) and left-to-right (L2R) exchange messages are first added to the left and right local APP values, which are then passed to the VN–VN processor. The VN–VN processor comprises three stages, each implementing different parts of Algorithm 1: the VFU out (VFUo) stage line 8, the CFU stage lines 1–5, and the VFU in (VFUi) stage line 7. Eventually, the received exchange messages are subtracted from the left and right APP values. Note that the respective channel values are implicitly contained in the APP values, which is a common practice in row-layered decoding architectures [19].

The column processor has a similar structure and is depicted in Fig. 3b. The update minima unit (UMU) stage updates the minima for all $2 \cdot (c - b)$ check nodes with the exchange messages according to Algorithm 4. Inside the CN–CN processor, the CFU out (CFUo) stage implements line 4, the VFU stage lines 6–9, and the CFU in (CFUi) stage lines 2–3 of Algorithm 1. In the split stage, the new exchange messages are generated by concatenating the sign bit from the check node computation with the new minimum value $\min_0$ (sign-magnitude representation). Note that no logic is required to perform this step.

The first and the last sub-block of the code require special processing. If a processed sub-block is the first of a block, it must not interact with the previous, if it is last, with the subsequent sub-block in the pipeline. This is realized with the respective multiplexers that, depending on the case, pass a neutral element or the termination sequence to the respective VN–VN or CN–CN processor.

The outputs are stored in a register stage. Let Q_{chv}, Q_{ext} and Q_{app} denote the bitwidths for the channel values, the extrinsic messages, and the APP values. The memory requirement for a row processor is then

$$\text{Mem}_{\text{row}} = 2 \cdot c \cdot Q_{\text{app}} + c \cdot d_v \cdot Q_{\text{ext}} \tag{5}$$

and for the column processor

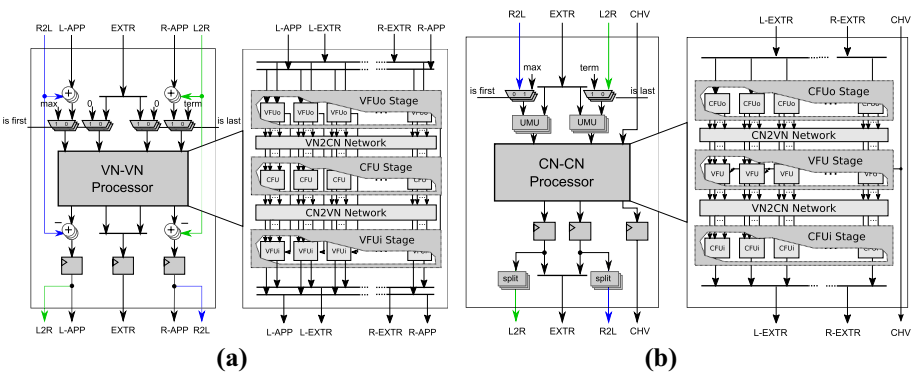


Fig. 3 Architectures for row- and column-wise iteration processors

$$\mathrm{Mem}_{\mathrm{col}} = 2 \cdot (c - b) \cdot [2 \cdot (Q_{\mathrm{ext}} - 1) + \lceil \log_2(d_c/2) \rceil + d_c/2] + [c \cdot Q_{\mathrm{chv}}]. \quad (6)$$

4.2 Window schedules

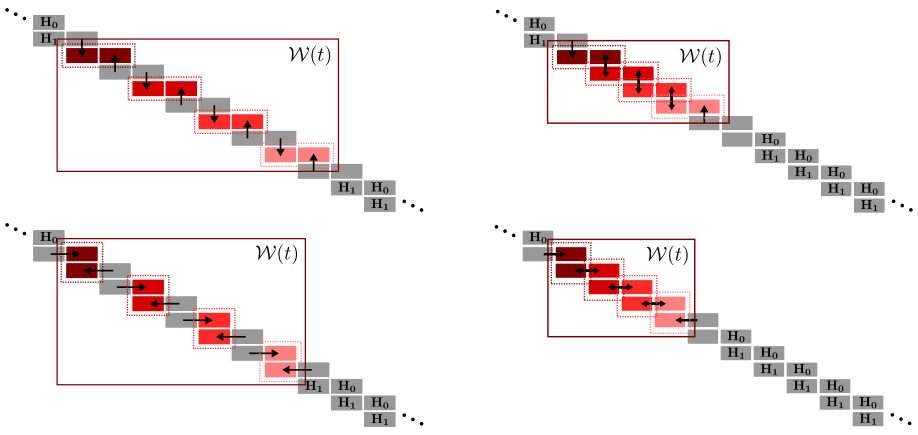
With the described processors, different window processing schedules can be implemented by differently interconnecting the processors.

4.2.1 Layered schedule

The layered schedule resembles the layer-by-layer schedule for LDPC block codes [17], except that multiple layers are processed in parallel. It was already stated in the introduction of the RLPD that the spacing between active layers must be at least m_s. Respective processing windows for 4 iterations of row- and column-layered decoding are illustrated in Fig. 4a.

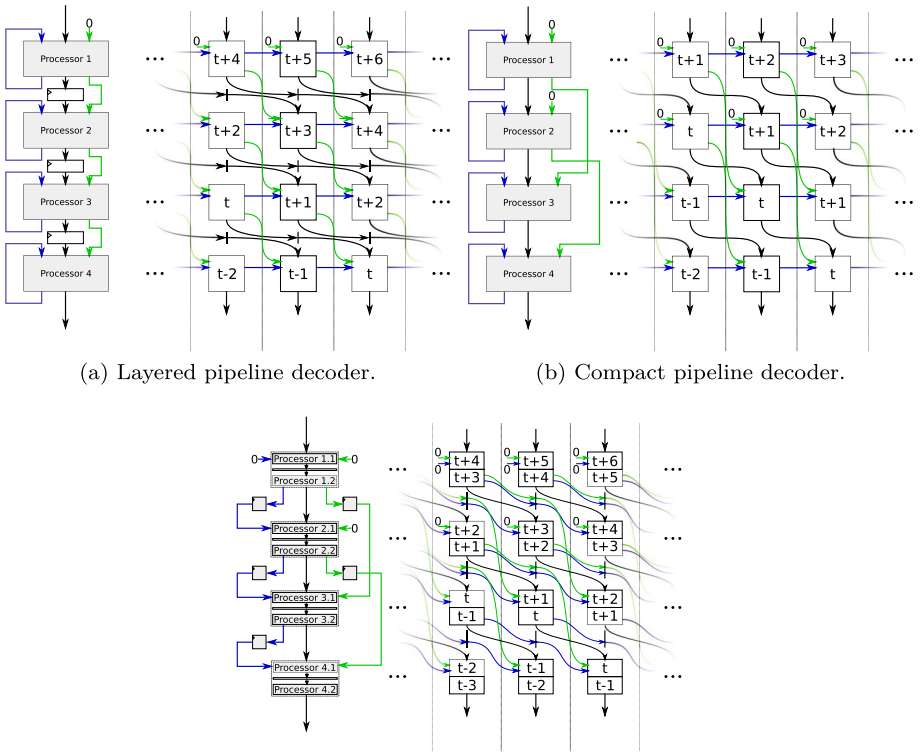
Figure 5a shows the corresponding processor arrangement. Note that the processors can be interchangeably row or column processors. The APP and extrinsic messages are passed from one processor to the next via an intermediate pipeline stage. Consequently, the respective rows/columns are updated only every second clock cycle, which creates the alternating layer processing depicted in Fig. 4a. To better understand the exchange message handling, Fig. 5a also shows the respective pipeline diagram. If we consider a sub-block t currently processed inside the pipeline (shown at the bottom right of the pipeline diagram), it must receive a R2L message from sub-block $t - 1$ and a L2R message from sub-block $t + 1$. With the sub-blocks moving through the pipeline, the R2L output of a processor i is connected to its R2L input and the L2R output to the L2R input of processor $i + 1$. In the first stage of the pipeline, L2R messages are not yet available and must be initialized with '0'.

It should be noted that this processor arrangement results in redundant computations in the row processors as each processor updates its right and left APP values in every



(a) Layered schedule. (b) Compact schedule.

Fig. 4 Illustration of different window schedules (4 iterations)



(a) Layered pipeline decoder. (b) Compact pipeline decoder.

(c) Double Compact pipeline decoder.

Fig. 5 Pipeline decoder architectures and pipeline diagrams

clock cycle, although the respective R2L and L2R messages are only updated every second clock cycle. To improve the efficiency of the architecture, we can directly connect the left APP input to the right APP output of the same processor and the right APP input to the left APP output of the previous processor. The R2L and L2R inputs are then set to zero. In this way, the processors share the APP values. This is not possible for the column-wise decoder as there is no distinction between intrinsic and extrinsic messages in the list representation.

4.2.2 Compact schedule

In the compact schedule, the layers are overlapping (see Fig. 4b). The respective processor arrangement is shown in Fig. 5b. Due to the overlaps, the sub-codes are updated in every clock cycle and the stored data passed from one processor to the next. This modification of the processing also affects the message exchange, which is again illustrated in the pipeline diagram. Accordingly, the R2L exchange messages must be propagated to the input of the same processor or, in the case of L2R messages, two stages ahead.

4.2.3 Multi-compact schedule

The decoding throughput for a given code is mainly determined by the achievable operating frequency and thus by the critical path of the processors. The critical path of both the row and column processors comprises a complete VN–VN or CN–CN

iteration. Unrolled block decoders introduce additional pipeline stages in the decoding iterations to achieve higher clock frequencies [5, 18]. For pipelined SC-LDPC decoders, this is more challenging due to the data dependencies between neighboring sub-blocks. With the proposed split node architecture, however, each processor operates on its local data set, which allows us to introduce additional pipeline stages in the processors and delay the message exchange. The corresponding arrangement for one additional pipeline stage in the processors is shown in Fig. 5c. The connections between processors for the L2R and R2L messages are again derived from the pipeline diagram. For each additional pipeline stage, the exchange message must be delayed by one clock cycle. We denote this schedule in the following as multi-compact, or compact (i), where i denotes the number of pipeline stages inside the processors.

4.3 Input/output stages

The column-wise decoders require an input stage to generate the initial list for the first processor in the pipeline from the channel values. This input stage is equivalent to a CFUi stage in the CN–CN processor. The row-wise decoders do not require a designated input stage, as the right APP values can be initialized directly with the channel values and the left APP values with zero messages. Instead, the row-wise decoders require an output stage to combine the left and right partial APP values. Therefore, the right APP messages of the last processor are stored in a register stage and then added to the current left APP messages.

4.4 Computational complexity analysis

We estimate the computational complexity of the decoders based on the number of edge messages that are computed and transferred in every clock cycle [20]. Let E_{row} denote the number of edges corresponding to a row-layer and E_{col} the number of edges corresponding to a column layer of $\mathbf{H}_{[1,L]}$. For a (d_v,d_c) regular code, the number of edges can be expressed as $E_{\text{row}} = d_c \cdot (c - b)$ and $E_{\text{col}} = d_v \cdot c$, which is equivalent to the number of ones in $\mathbf{H}_r(t)$ and $\mathbf{H}_c(t)$, respectively. Since $\mathbf{H}_r(t)$ and $\mathbf{H}_c(t)$ of a time-invariant code are composed of the same sub-matrices $\mathbf{H}_1$ and $\mathbf{H}_0$, the number of edges in one row-layer equals the number of edges in one column layer, i.e., $E_{\text{row}} = E_{\text{col}}$. From this, it follows that the VN–VN processor and the CN–CN processor compute the same number of edge messages in each clock cycle and thus exhibit similar computational complexity. However, the node splitting introduces additional edges for the exchange of messages between processors. The resulting number of edges for the row processor is then $\hat{E}_{\text{row}} = E_{\text{row}} + 2 \cdot c$ and for the column processor $\hat{E}_{\text{col}} = E_{\text{col}} + 2 \cdot (c - b)$. Taking into account the number of iterations (processors) I, we can define the complexity of the row-wise decoders as

$$C_{\text{row}} = \hat{E}_{\text{row}} \cdot I = [d_c \cdot (c - b) + 2 \cdot c] \cdot I \tag{7}$$

and for the column-wise decoders as

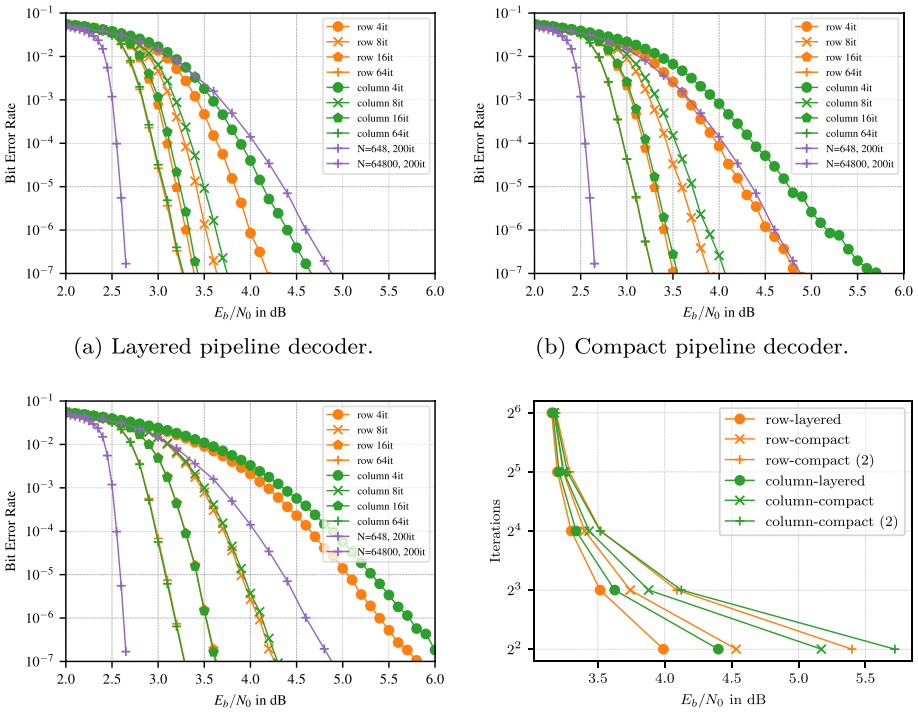
$$C_{\text{col}} = \hat{E}_{\text{col}} \cdot I = [d_v \cdot c + 2 \cdot (c - b)] \cdot I. \tag{8}$$

5 Results and discussion

In this section, we present FEC performance and implementation results of the presented pipeline decoders. The performance evaluation and the implementation were performed for the EPIC SC-LDPC code with $L = 80$, $m_s = 1$, $b = 512$ and $c = 640$ [1]. The code is regular with $d_v = 4$ and $d_c = 20$. The code rate is $R = 0.798$ and the total block length $N = 51238$ bit. The choice of code is based on the fact that the associated parity-check matrices and performance data are publicly available [13]. Furthermore, the sub-block size $c = 640$ allowed for a fully parallel decoder implementation, other than the larger EPIC codes with $c = 960$ and $c = 1280$. For the latter, not all decoder architectures could successfully pass the placement and routing process in the integrated circuit design due to routing congestions. The smaller sub-block size results in a lower performance of the code, for the quantification of which we again refer to [13]. A detailed description of the experimental setup is provided in Sect. 7

5.1 FEC Performance

Figure 6a shows a side-by-side performance comparison of the row- and the column-layered decoder with 4, 8, 16, and 64 iterations (floating point). For better classification of the results, performance curves of the (64800, 51804) DVB-S2 code and the (648, 540) Wi-Fi code are also shown, each decoded with 200 iterations Normalized MS (NMS) ($\gamma = 0.75$, flooding schedule). Both codes are comparable to the SC-LDPC code in terms of code rate ($R = 0.8$ and $R = 0.83$). The DVB-S2 serves as a reference LDPC code, that is similar in length to a full code block (64800 bit vs. 51238 bit). Conversely, the Wi-Fi



(a) Layered pipeline decoder.

(b) Compact pipeline decoder.

(c) Double compact pipeline decoder.

(d) Comparison for BER 10^{-6}.

Fig. 6 Simulation results

code can be considered an uncoupled counterpart to the SC-LDPC code with a block length similar to one sub-block (648 bit vs. 640 bit). We see that for the same number of iterations, the row-layered decoder outperforms the column-layered decoder. This observation is also made for the compact decoders in Fig. 6b and the compact decoders with one additional pipeline stage in the processor in Fig. 6c. We also see that the performance decreases from the layered to the compact and the double-compact decoders. For better visualization, Fig. 6d shows the signal-to-noise ratio (SNR) at BER 10^{-6} for all presented decoders over the iterations. This representation helps to illustrate the convergence behavior of the decoders, where a strong curvature toward the origin of the diagram indicates a faster convergence. The fastest convergence is achieved with layered decoding, the slowest with double-compact decoding. Furthermore, we see that all decoders asymptotically approach an SNR of approximately 3.1 dB. This behavior is expected as the exchanged messages in the compact, particularly in the double-compact decoder, are less recent, which affects the convergence speed of the decoder.

5.2 Implementation results

To compare the efficiency of different pipeline decoders, we implemented

- A 4-iteration row-layered decoder,
- A 5-iteration column-layered decoder,
- A 6-iteration row-compact decoder,
- A 7-iteration column-compact decoder,
- And two 9-iteration compact decoders with one additional pipeline stage in the processor, with row- and column-wise processing.

The pipeline stage was inserted in the minimum search tree of the CFU stage for the row-wise processor and after the VFU stage for the column-wise processor. The respective number of iterations was selected such that all decoders achieve the same FEC performance (fixed point) of BER $= 10^{-6}$ at 4.2 dB, which is around 1 dB from the maximum MS performance of the code. All decoders use 4 bits to represent the channel values and the extrinsic messages. The row processors' APP, R2L, and L2R values are quantized with 6 bits. The performance loss compared to floating point for this configuration is less than 0.2 dB up to a BER of 10^{-6}.

The layouts of the implemented decoders are depicted in Fig. 7, and Table 1 summarizes the results. When comparing the respective row- and column-wise decoders, we see that the column-wise decoders outperform the respective row-wise decoders in terms of throughput, energy, and area efficiency. For the layered architecture, the column-wise decoder achieves 25% higher clock frequency (throughput), around 19% better energy efficiency and 22% better area efficiency. For the compact and double-compact architectures, the improvements are even more profound with 50% (freq.), 59% (energy eff.), and 135% (area eff.), and with 38% (freq.), 24% (energy eff.) and 38% (area eff.), respectively.

At first glance, these results seem unexpected, especially considering the fact that the column-wise decoders are composed of an equal or even larger number of processors. However, an important fact to consider is the lower number of edge message

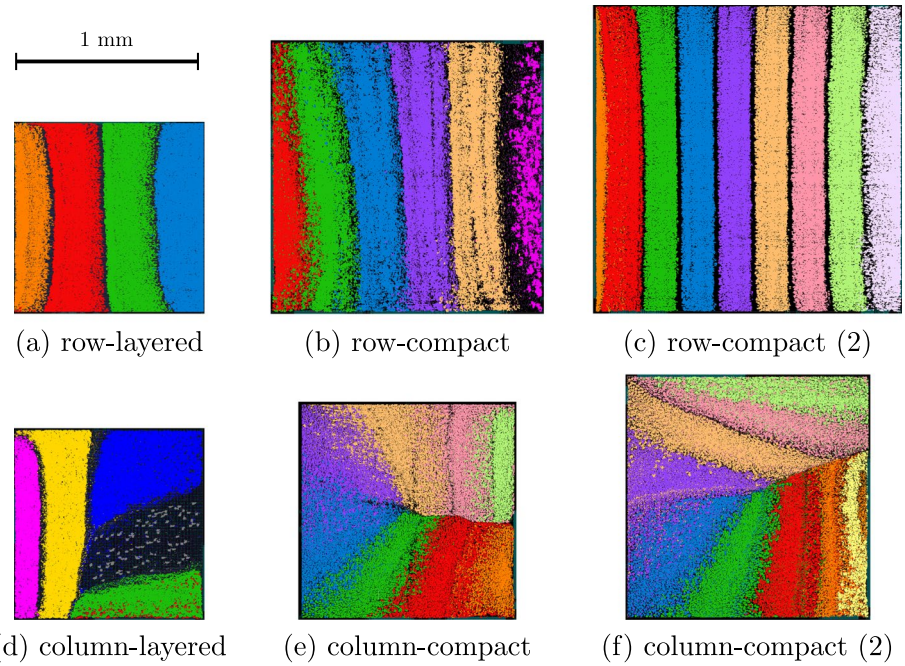


Fig. 7 Layouts of the presented decoders in 22nm

Table 1 Implementation results in 22nm FD-SOI technology

Code (*c*, *b*, *L*)	**(640,512,80)**					
Architecture	Layered		Compact		Compact (2)	
Processing	Row	Column	Row	Column	Row	Column
#Processors	4	5	6	7	9	9
Comp. Compl.	10240[a]	14080[b]	23040[c]	19712[b]	34560[c]	25344[b]
#Pipeline Stages	10	13	8	10	20	21
Frequency [MHz]	438	549	351	525	500	693
Core Area [mm^2]	1.02	1.04	2.11	1.33	2.69	1.68
Utilization [%]	82	76	77	77	80	77
Power Total [W]	2.38	2.42	5.13	3.16	4.42	3.78
Power Density [W/mm^2]	2.35	2.33	2.43	2.37	1.65	2.24
Throughput [Gbps]	280	351	224	336	320	443
Latency [ns]	22.8	23.7	22.8	19.0	40.0	30.3
Area Eff. [Gb/s/mm^2]	276	338	107	252	119	263
Energy Eff. [pJ/bit]	8.5	6.9	22.8	9.4	13.8	8.5
Energy Eff. per proc. [pJ/bit]	2.1	1.4	3.8	1.3	1.5	0.94

a Calculated using $E_{\text{row}} \cdot I$ since no exchange messages are processed.

b Calculated using Eq. 8.

c Calculated using Eq. 7

computations of the column-wise processors ($\hat{E}_{\text{col}} = 2816$ vs. $\hat{E}_{\text{row}} = 3840$). Relating the calculated computational complexity values to implementation metrics such as energy and area efficiency, we observe a large discrepancy. We attribute this to the data transfer dominance of the decoder. It was already shown in [21] that for data

transfer dominated applications, like LDPC decoding, the number of operations has little impact on area and energy efficiency. Another important aspect to consider for the better efficiency of the column-wise decoders is that the data transfer between the processors is much lower ($256 \cdot Q_{\text{ext}} = 1024$ vs. $1296 \cdot Q_{\text{app}} = 7776$) and, consequently, the wiring load for the logic circuit resulting in smaller logic cells. Furthermore, the column processor has 57% fewer registers (Eq. 5 and Eq. 6), and thus a smaller clock tree.

A similar observation is made when comparing the compact and the double-compact decoders. Despite a higher computational complexity of the latter, the energy efficiency improves. Here, the introduction of an additional pipeline stage reduces the wiring load and toggling rates. Consequently, the energy efficiency per processor improves by 28% for the column-wise decoder and 60% for the row-wise decoder.

It should be noted that there is a large drop in frequency and degradation in energy efficiency from the row-layered to the row-compact decoder. The reason for this is that the row-layered decoder does not process exchange messages (see Sect. 4.2.1). Consequently, the adders in the row processors and the corresponding wiring were not synthesized, which contributes to the better efficiency of the row-layered decoder.

Finally, Table 2 shows the results for the column double-compact decoder in comparison to data from the fastest LDPC Block Code (LDPC-BC) and LDPC-CC decoders in the literature. The column double-compact decoder was selected for the comparison as it achieves the highest throughput of the presented decoders. The FPWD in [12] is not listed in Table 2, as the results are similar to the column-compact decoder in Table 1.

The LDPC-BC decoder in [8] is implemented in the same 22 nm technology and features a similar code rate and (sub-)block size and therefore allows for a detailed comparison. The throughput of the SC-LDPC decoder is about 20% lower than the throughput of the unrolled block decoder. This results from the fewer pipeline stages per iteration

Table 2 Comparison with state-of-the-art high-throughput LDPC decoders

Decoder	**This work**	**[15]**	**[22]**	**[5]**	**[8]**
Code	SC-LDPC	LDPC-CC	LDPC-CC	LDPC-BC	LDPC-BC
(Sub-)block size	640	n/a	n/a	2048	648
Code Rate	0.798	1/2–4/5	1/2	0.84	0.83
CMOS Technology	22nm	65nm	90nm	28nm	22nm
Supply Voltage [V]	0.8	1.20	0.8	1.0	0.8
Processing	Column	Row	Row	Block	Block
Architecture	Compact	Compact	Layered	Unrolled	Unrolled
# Processors	9	6	4	5	8
Quantization [bit]	4	6	6	3	4
Eb/N_0 at BER 10^{-6} [dB]	4.2	n/a	~ 3	~ 4.7	4.9
Frequency [MHz]	693	322	305	862	837
Decoding Latency [ns]	30.3	n/a	n/a	69.6	31.0
Post P &R Area [mm^2]	1.68	1.6	2.18	16.2	1.75
Throughput [Gbit/s]	443	7.72	3.66	588	542
Energy Eff. [pJ/bit]	8.5	53.4	75.2	22.7	5.8
Energy Eff. per proc. [pJ/bit]	0.94	8.9	18.8	4.5	0.73
Area Eff. [Gbit/s/mm^2]	263	4.8	1.7	36.3	311

of the SC-LDPC decoder (2 vs. 3). As a result, the LDPC-BC decoder achieves a higher maximum frequency. The area of both decoders is similar. However, the SC-LDPC decoder achieves a gain in error correction performance of 0.7 dB over the LDPC-BC decoder. With an E_b/N_0 of 4.2 dB the SC-LDPC decoder surpasses the maximum performance of the Wi-Fi code at 200 iterations (see Fig. 6c). This illustrates the great potential of SC-LDPC codes for high-throughput decoding beyond 5G.

6 Conclusions

In this paper, we presented the first in-depth investigation on the implementation of SC-LDPC decoders for throughputs beyond 100 Gbit/s. For a $N = 51328$, $R = 0.8$ terminated SC-LDPC code with sub-block size $c = 640$ and coupling width $m_s = 1$, we explored various design trade-offs, including row- and column-wise decoding, non-overlapping and overlapping window scheduling, and processor pipelining. In this context, we provided the first description of a column-wise SC-LDPC decoding architecture. We have shown that the column-wise decoding of SC-LDPC codes is a promising approach, which, despite poorer convergence behavior, offers advantages in the implementation efficiency.

7 Methods

Performance evaluation and implementation were performed for the EPIC SC-LDPC code with $L = 80$, $m_s = 1$, $b = 512$ and $c = 640$ [13].

Performance is evaluated with BER over SNR simulations for the transmission scenario depicted in Fig. 1. Each SNR point was simulated with 10^6 blocks (corresponding to $80 \cdot 10^6$ sub-blocks) transmitted over an additive white Gaussian noise (AWGN) channel with binary phase-shift keying (BPSK) modulation. All decoders use the NMS algorithm with $\gamma = 0.75$.

Implementation was performed in a 22 nm fully depleted silicon-on-insulator (FD-SOI) technology under worst-case process, voltage, and temperature (PVT) conditions (125°, 0.72 V for timing, 0.80 V for power). For synthesis, we used the design compiler, for placement and routing the IC-compiler, both from Synopsys. Power numbers are calculated with back-annotated wiring data using Synopsys Primetime and Siemens Modelsim. The stimuli for the power simulations were obtained at a fixed SNR of 4 dB. Measurements started after an initialization phase of 100 clock cycles to ensure that the pipelines are filled.

Abbreviations

APP	A posteriori probability
AWGN	Additive white Gaussian noise
ASIC	Application-specific integrated circuit
BER	Bit error rate
BPSK	Binary phase shift keying
BSMC	Binary symmetric memoryless channel
CCPD	Column-compact pipeline decoder
CFU	Check node functional unit
CFUi	CFU in
CFUo	CFU out
CLPD	Column-layered pipeline decoder
FD-SOI	Fully depleted silicon on insulator
FEC	Forward error correction
FER	Frame error rate

FPWD	Full-parallel window decoder
L2R	Left-to-right
LDPC	Low-density parity-check
LDPC-BC	LDPC block code
LDPC-CC	LDPC convolutional codes
LEX	Left exchange
LLR	Log-likelihood ratio
MS	Min-sum
NMS	Normalized MS
OCA	On-demand check node activation
OVA	On-demand variable node activation
PVT	Process, voltage and temperature
R2L	Right-to-left
RCPD	Row-compact pipeline decoder
REX	Right exchange
RLPD	Row-layered pipeline decoder
SC-LDPC	Spatially coupled LDPC
SNR	Signal-to-noise ratio
SOA	State-of-the-art
UMU	Update minima unit
VFU	Variable node functional unit
VFUi	VFU in
VFUo	VFU out

Author Contributions
MH and NW contributed to the main idea. MH performed the numerical simulations and hardware implementation. MH and NW wrote the paper. All authors read and approved the final manuscript.

Funding
Open Access funding enabled and organized by Projekt DEAL. This work was financially supported by the EU (project-ID: 760150-EPIC) and the German ministry of education and research (project: FACTOR).

Availability of data and materials
EPIC SC-LDPC codes: https://www.uni-kl.de/en/channel-codes/channel-codes-database/spatially-coupled-ldpc/

Declarations

Competing interests
The authors declare that they have no competing interests.

Received: 2 November 2021 Accepted: 1 September 2022
Published online: 22 September 2022

References

1. EPIC - Enabling practical wireless Tb/s communications with next generation channel coding - https://epic-h2020.eu/results
2. R.G. Gallager, Low-density parity-check codes. IRE Trans. Inf. Theory **8**(1), 21–28 (1962)
3. C. Kestel, M. Herrmann, N. Wehn, When channel coding hits the implementation wall, in 2018 IEEE 10th International Symposium on Turbo Codes Iterative Information Processing (ISTC), pp. 1–6 (2018). https://doi.org/10.1109/ISTC.2018.8625324
4. M. Li, V. Derudder, K. Bertrand, C. Desset, A. Bourdoux, High-speed LDPC decoders towards 1 Tb/s. IEEE Transactions on Circuits and Systems I: Regular Papers, pp. 1–10 (2021). https://doi.org/10.1109/TCSI.2021.3060880
5. R. Ghanaatian, A. Balatsoukas-Stimming, T.C. Müller, M. Meidlinger, G. Matz, A. Teman, A. Burg, A 588-Gb/s LDPC decoder based on finite-alphabet message passing. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 26(2), 329–340 (2018). https://doi.org/10.1109/TVLSI.2017.2766925
6. M. Herrmann, C. Kestel, N. Wehn, Energy efficient FEC decoders. In: 2021 IEEE International Symposium on Topics in Coding (ISTC) (2021)
7. A. Jimenez Felstrom, K.S. Zigangirov, Time-varying periodic convolutional codes with low-density parity-check matrix. IEEE Trans. Inf. Theory **45**(6), 2181–2191 (1999)
8. N. Wehn, O. Sahin, M. Herrmann, Forward-error-correction for beyond-5G ultra-high throughput communications. 2021 IEEE International Symposium on Topics in Coding (ISTC) (2021)
9. D.J. Costello, L. Dolecek, T.E. Fuja, J. Kliewer, D.G.M. Mitchell, R. Smarandache, Spatially coupled sparse codes on graphs: theory and practice. IEEE Commun. Mag. **52**(7), 168–176 (2014)
10. A.E. Pusane, A.J. Feltstrom, A. Sridharan, M. Lentmaier, K.S. Zigangirov, D.J. Costello, Implementation aspects of LDPC convolutional codes. IEEE Trans. Commun. **56**(7), 1060–1069 (2008)
11. N.U. Hassan, M. Schluter, G.P. Fettweis, Fully parallel window decoder architecture for spatially-coupled IDPC codes. 2016 IEEE International Conference on Communications (ICC), pp. 1–6 (2016)

12. M. Herrmann, N. Wehn, M. Thalmaier, M. Fehrenz, T. Lehnigk-Emden, M. Alles, A 336 gbit/s full-parallel window decoder for spatially coupled ldpc codes. Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit) (2021)
13. M. Helmling, S. Scholl, F. Gensheimer, T. Dietz, K. Kraft, S. Ruzika, N. Wehn, Database of Channel Codes and ML Simulation Results. (2022). https://www.uni-kl.de/channel-codes
14. C. Chen, Y. Lin, H. Chang, C. Lee, A 2.37-gb/s 284.8 mw rate-compatible (491, 3, 6) ldpc-cc decoder. IEEE J Solid-State Circuits **47**(4), 817–831 (2012)
15. C. Lin, R. Liu, C. Chen, H. Chang, C. Lee, A 7.72 gb/s ldpc-cc decoder with overlapped architecture for pre-5g wireless communications. 2016 IEEE Asian Solid-State Circuits Conference (A-SSCC), pp. 337–340 (2016)
16. F.R. Kschischang, B.J. Frey, H.-A. Loeliger, Factor graphs and the sum-product algorithm. IEEE Trans. Inf. Theory **47**(2), 498–519 (2001). https://doi.org/10.1109/18.910572
17. D.E. Hocevar, A reduced complexity decoder architecture via layered decoding of IDPC codes. IEEE Workshop on Signal Processing Systems, 2004. SIPS 2004., pp. 107–112 (2004)
18. P. Schläfer, N. Wehn, M. Alles, T. Lehnigk-Emden, A new dimension of parallelism in ultra high throughput LDPC decoding. SiPS 2013 Proceedings, pp. 153–158 (2013). https://doi.org/10.1109/SiPS.2013.6674497
19. O. Boncalo, A. Amaricai, Ultra high throughput unrolled layered architecture for qc-ldpc decoders. 2017 IEEE Computer Society Annual Symposium on VLSI (ISVLSI), pp. 225–230 (2017)
20. M. Baldi, *Low-Density Parity-Check Codes* (Springer, Cham, 2014), pp.5–21. https://doi.org/10.1007/978-3-319-02556-8_2
21. F. Kienle, N. Wehn, H. Meyr, On complexity, energy- and implementation-efficiency of channel decoders. IEEE Trans. Commun. **59**(12), 3301–3310 (2011). https://doi.org/10.1109/TCOMM.2011.092011.100157
22. C. Chen, Y. Lan, H. Chang, C. Lee, A 3.66gb/s 275mw tb-ldpc-cc decoder chip for mimo broadcasting communications. 2013 IEEE Asian Solid-State Circuits Conference (A-SSCC), pp. 153–156 (2013)

Publisher's Note