NR LDPC Decoder

Sebastian Wagner (TCL)

October 2, 2019

Currently Supported:

BG	Lifting Size Z	Code Rate R
1	all	1/3, 2/3, 8/9
2	all	1/5, 1/3, 2/3

Version 1.0:

- Initial version

Version 2.0:

- Enhancements in message passing:
  - LUTs replaced by smaller BG-specific parameters
  - Inefficient load/store replaced by circular memcpy
- Bug fixes:
  - Fixed bug in function `llr2CnProcBuf`
  - Corrected input LLR dynamic range in BLER simulations
- Results:
  - Size of LUTs reduced significantly (60MB to 200KB)
  - Siginifcantly enhances execution time (factor 3.5)
  - Improved BLER performance (all simulation results have been updated)

Contents

1 Introduction

Low Density Parity Check (LDPC) codes have been developed by Gallager in 1963 [1]. They are linear error correcting codes that are capacity-achieving for large block length and are completely described by their Parity Check Matrix (PCM) $\mathbf{H}^{M\times N}$. The PCM $\mathbf{H}$ defines M constraints on the codeword $\mathbf{c}$ of length N such that

$$\mathbf{Hc} = \mathbf{0}. \tag{1}$$

The number of information bits B that can be encoded with $\mathbf{H}$ is given by $B = N - M$. Hence the code rate R of $\mathbf{H}$ reads

$$R = \frac{B}{N} = 1 - \frac{M}{N}. \tag{2}$$

1.1 LDPC in NR

NR uses quasi-cyclic (QC) Protograph LDPC codes, i.e. a smaller graph, called Base Graph (BG), is defined and utilized to construct the larger PCM. This has the advantage that the large PCM does not have to be stored in memory and allows for a more efficient implementation while maintaining good decoding properties. Two BGs $\mathbf{H}_{\mathrm{BG}} \in \mathbb{N}^{M_b \times N_b}$ are defined in NR:

1. $\mathbf{H}_{\mathrm{BG1}} \in \mathbb{N}^{46\times 68}$
2. $\mathbf{H}_{\mathrm{BG2}} \in \mathbb{N}^{42\times 52}$

where $\mathbb{N}$ is the set of integers. For instance the first 3 rows and 13 columns of BG2 are given by

$$\mathbf{H}_{\mathrm{BG2}} = \begin{bmatrix} 9 & 117 & 204 & 26 & \emptyset & \emptyset & 189 & \emptyset & \emptyset & 205 & 0 & 0 & \emptyset & \emptyset \\ 127 & \emptyset & \emptyset & 166 & 253 & 125 & 226 & 156 & 224 & 252 & \emptyset & 0 & 0 & \emptyset \\ 81 & 114 & \emptyset & 44 & 52 & \emptyset & \emptyset & \emptyset & 240 & \emptyset & 1 & \emptyset & 0 & 0 \end{bmatrix}.$$

To obtain the PCM $\mathbf{H}$ from the BG $\mathbf{H}_{\mathrm{BG}}$, each element $\mathbf{H}_{\mathrm{BG}}(i,j)$ in the BG is replaced by a lifting matrix of size $Z_c \times Z_c$ according to

$$\mathbf{H}_{\mathrm{BG}}(i,j) = \begin{cases} \mathbf{0} & \text{if } \mathbf{H}_{\mathrm{BG}}(i,j) = \emptyset \\ \mathbf{I}_{P_{ij}} & \text{otherwise} \end{cases} \tag{3}$$

where $\mathbf{I}_{P_{ij}}$ is the identity matrix circularly shifted to the right by $P_{ij} = \mathbf{H}_{\mathrm{BG}}(i,j) \mod Z_c$. Hence, the resulting PCM $\mathbf{H}$ will be of size $M_b Z_c \times N_b Z_c$.

The lifting size Z_c depends on the number of bits to encode. To limit the complexity, a discrete set $\mathcal{Z}$ of possible values of Z_c has been defined in [2] and the optimal value Z_c is calculated according to

$$Z_c = \min_{\mathbf{Z}\in\mathcal{Z}} \left[Z \geq \frac{B}{N_b} \right]. \tag{4}$$

The base rate of the two BGs is 1/3 and 1/5 for BG1 and BG2, respectively. That is, BG1 encodes $K = 22Z_c$ bits and BG2 encodes $K = 10Z_c$ bits. Note that the first 2 columns of BG 1 and 2 are always punctured, that is after encoding, the first $2Z_c$ bits are discarded and not transmitted. For instance, consider $B = 500$ information bits to encode using BG2, (1.2) yields $Z_c = 64$ hence $K = 640$. Since $K > B$, $K - B = 140$ filler bits are appended to the information bits. The PCM $\mathbf{H}_{\mathrm{BG2}}$ is of size 2688×3328 and the 640 bits $\mathbf{b}$ are encoded according to (1) at a rate $R \approx 0.192$. To achieve the higher base rate of 0.2, the first 128 are punctured, i.e. instead of transmitting all 3328 bits, only 3200 are transmitted resulting in the desired rate $R = 640/3200 = 0.2$.

1.2 LDPC Decoding

The decoding of codeword **c** can be achieved via the classical message passing algorithm. This algorithm can be illustrated best using the Tanner graph of the PCM. The rows of the PCM are called check nodes (CN) since they represent the parity check equations. The parity check equation of each of these check nodes involves various bits in the codeword. Similarly, every column of the PCM corresponds to a bit and each bit is involved in several parity check equations. In the Tanner graph representation, the bits are called bit nodes (BN). Let's go back to the previous example of BG2 and assume $Z_c = 2$, hence the first 3 rows and 13 columns of BG2 $\mathbf{H}_{\text{BG2}}$ read

$$\mathbf{H}_{\text{BG2}} = \begin{bmatrix} 1 & 1 & 0 & 0 & \emptyset & \emptyset & 1 & \emptyset & \emptyset & 1 & 0 & 0 & \emptyset & \emptyset \\ 1 & \emptyset & \emptyset & 0 & 1 & 1 & 0 & 0 & 0 & 0 & \emptyset & 0 & 0 & \emptyset \\ 1 & 0 & \emptyset & 0 & 0 & \emptyset & \emptyset & \emptyset & 0 & \emptyset & 1 & \emptyset & 0 & 0 \end{bmatrix}.$$

Replacing the elements according to (3), we obtain the first 6 rows and 26 columns of the PCM as

$$\mathbf{H} = \begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}.$$

The Tanner graph of the first 8 BNs is shown in Figure 1.2.

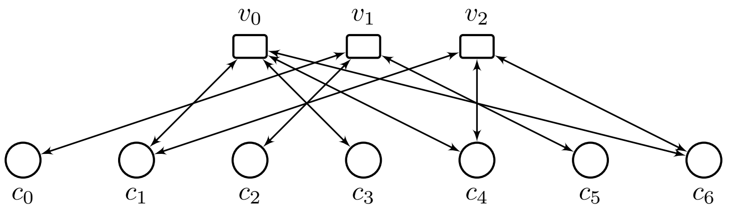

Figure 1: Tanner graph for first 7 bits nodes and 3 check nodes from (1.2).

The message passing algorithm is an iterative algorithm where probabilities of the bits (being either 0 or 1) are exchanged between the BNs and CNs. After sufficient iterations, the probabilities will have either converged to either 0 or 1 and the parity check equations will be satisfied, at this point, the codeword has been decoded correctly.

2 LDPC Decoder Implementation

The implementation on a general purpose processor (GPP) has to take advantage of potential instruction extension of the processor architecture. We focus on the Intel x86 instruction set architecture (ISA) and its advanced vector extension (AVX). In particular, we utilize AVX2 with its 256-bit single instruction multiple data (SIMD) format. In order to utilize AVX2 to speed up the processing at the CNs and BNs, the corresponding data has to be ordered/aligned in a specific way. The processing flow of the LDPC decoder is depicted in 2.

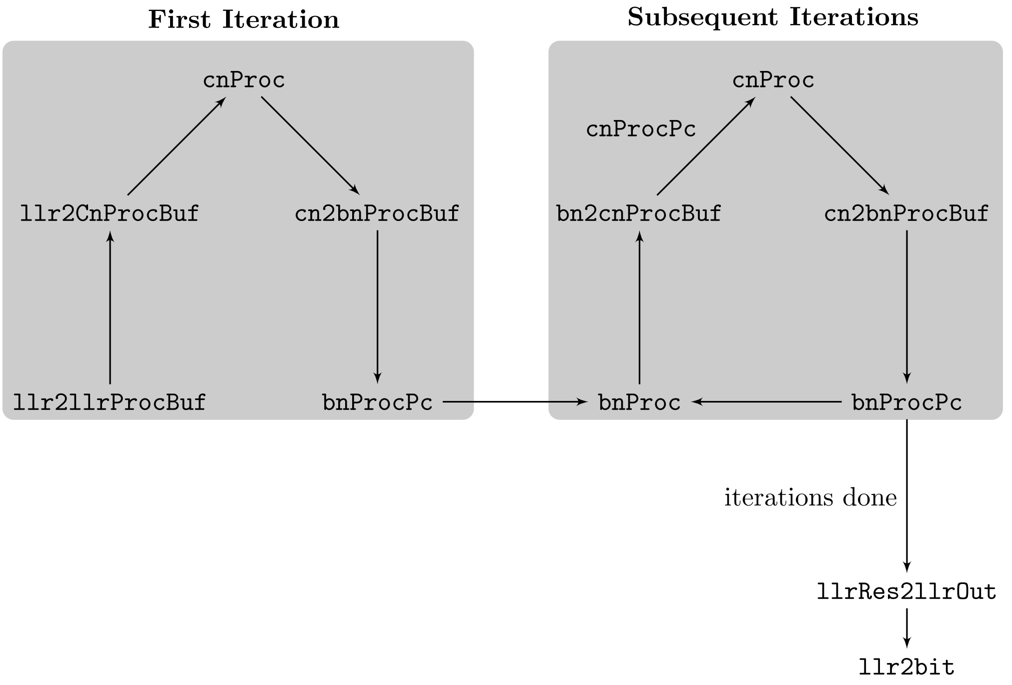

Figure 2: LDPC Decoder processing flow.

The functions involved are described in more detail in Table 1.

The input LLRs are assumed to be 8-bit and aligned on 32 bytes. CN processing is carried out in 8-bit whereas BN processing is done in 16 bit. Subsequently, the processing tasks at the CNs and BNs are explained in more detail.

Function	Description
`llr2llrProcBuf`	Copies input LLRs to LLR processing buffer
`llr2CnProcBuf`	Copies input LLRs to CN processing buffer
`cnProc`	Performs CN signal processing
`cnProcPc`	Performs parity check
`cn2bnProcBuf`	Copies the CN results to the BN processing buffer
`bnProcPc`	Performs BN processing for parity check and/or hard-decision
`bnProc`	Utilizes the results of `bnProcPc` to compute LLRs for CN processing
`bn2cnProcBuf`	Copies the BN results to the CN processing buffer
`llrRes2llrOut`	Copies the results of `bnProcPc` to output LLRs
`llr2bit`	Performs hard-decision on the output LLRs

Table 1: Summary of the LDPC decoder functions.

2.1 Check Node Processing

Denote q_{ij} the value from BN j to CN i and let $\mathcal{B}_i$ be the set of connected BNs to the ith CN. Then, using the min-sum approximation, CN i has to carry out the following operation for each connected BN.

$$r_{ji} = \prod_{j' \in \mathcal{B}_i \setminus j} \operatorname{sgn} q_{ij'} \min_{j' \in \mathcal{B}_i \setminus j} |q_{ij'}| \tag{5}$$

where r_{ji} is the value returned to BN j from CN i. There are $M_b = \{46, 42\}$ CNs in BG 1 and BG 2, respectively. Each of these CNs is connected to only a small number of BNs. The number of connected BNs to CN i is $|\mathcal{B}_i|$. In BG1 and BG2, $|\mathcal{B}_i| = \{3, 4, 5, 6, 7, 8, 9, 10, 19\}$ and $|\mathcal{B}_i| = \{3, 4, 5, 6, 8, 10\}$, respectively. The following tables show the number of CNs $M_{|\mathcal{B}_i|}$ that are connected to the same number of BNs.

$\lvert\mathcal{B}_i\rvert$	3	4	5	6	7	8	9	10	19
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG1}}$	1	5	18	8	5	2	2	1	4
$M_{\lvert\mathcal{B}_i\rvert}^{\mathrm{BG2}}$	6	20	9	3	0	2	0	2	0

Table 2: Ceck node groups for BG1 and BG2.

It can be observed that each CN is at least connected to 3 BNs and there are 9 groups and 5 groups in BG1 and BG2, respectively. Denote the set of CN groups as $\mathcal{G}$ and M_k the number of CNs in group $k \in \mathcal{G}$, e.g. for BG2 $M_4 = 20 Z_c$. Each CN group will be processed separately. The CN processing buffer p_C^k of group k is defined as

$$p_C^k = \{\underbrace{q_{11} q_{21} \dots q_{M_k 1}}_{1.BN}, \underbrace{q_{12} q_{22} \dots q_{M_k 2}}_{2.BN}, \dots, \underbrace{q_{12} q_{22} \dots q_{M_k k}}_{lastBN}\} \tag{6}$$

Hence, $|p_C^k| = k M_k$, e.g, $Z_c = 128$, $|p_C^4| = 4 \cdot 20 \cdot 128 = 10240$.

Listing 1: Example of CN processing for group 3 from `cnProc`.

```
const uint8_t lut_idxCnProcG3[3][2] = {{72,144}, {0,144}, {0,72}};
```

```
  // ======================================================================
  // Process group with 3 BNs

  // Number of groups of 32 CNs for parallel processing
  M = (lut_numCnInCnGroups[0]*Z)>>5;
  // Set the offset to each bit within a group in terms of 32 Byte
  bitOffsetInGroup = (lut_numCnInCnGroups_BG2_R15[0]*NR_LDPC_ZMAX)>>5;

  // Set pointers to start of group 3
  p_cnProcBuf    = (__m256i*) &cnProcBuf   [lut_startAddrCnGroups[0]];
  p_cnProcBufRes = (__m256i*) &cnProcBufRes[lut_startAddrCnGroups[0]];

  // Loop over every BN
  for (j=0; j<3; j++)
  {
    // Set of results pointer to correct BN address
    p_cnProcBufResBit = p_cnProcBufRes + (j*bitOffsetInGroup);

    // Loop over CNs
    for (i=0; i<M; i++)
    {
      // Abs and sign of 32 CNs (first BN)
      ymm0 = p_cnProcBuf[lut_idxCnProcG3[j][0] + i];
      sgn  = _mm256_sign_epi8(*p_ones, ymm0);
      min  = _mm256_abs_epi8(ymm0);

      // 32 CNs of second BN
      ymm0 = p_cnProcBuf[lut_idxCnProcG3[j][1] + i];
      min  = _mm256_min_epu8(min, _mm256_abs_epi8(ymm0));
      sgn  = _mm256_sign_epi8(sgn, ymm0);

      // Store result
      min = _mm256_min_epu8(min, *p_maxLLR); // 128 in epi8 is -127
      *p_cnProcBufResBit = _mm256_sign_epi8(min, sgn);
      p_cnProcBufResBit++;
    }
  }

}
```

Once all results of the check node processing r_{ji} have been calculated, they are copied to the bit node processing buffer.

2.2 Bit Node Processing

Denote r_{ji} the value from CN i to BN j and let $\mathcal{C}_j$ be the set of connected CNs to the jth BN. Each BN j has to carry out the following operation for every connected CN $i \in \mathcal{C}_j$.

$$q_{ij} = \Lambda_j + \sum_{i' \in \mathcal{C}_j \setminus i} r_{ji'} \tag{7}$$

There are $N_b = \{68, 52\}$ BNs in BG 1 and BG 2, respectively. Each of these BNs is connected to only a small number of CNs. The number of connected CNs to BN j is $|\mathcal{C}_j|$. In BG1 and BG2, $|\mathcal{C}_j| = \{1, 4, 7, 8, 9, 10, 11, 12, 28, 30\}$ and $|\mathcal{C}_j| = \{1, 5, 6, 7, 8, 9, 10, 12, 13, 14, 16, 22, 23\}$, respectively. The following tables show the number of BNs $K_{|\mathcal{C}_j|}$ that are connected to the same number of CNs.

The BNs that are connected to a single CN do not need to be considered in the BN processing since (7) yields $q_{ij} = \Lambda_j$. It can be observed that the grouping is less compact, i.e. there are many

$\lvert\mathcal{C}_j\rvert$	1	4	5	6	7	8	9	10	11	12	13	14	15	16	22	23	28	30
$K^{\mathrm{BG1}}_{\lvert\mathcal{C}_j\rvert}$	42	1	1	2	4	3	1	4	3	4	1	0	0	0	0	0	1	1
$K^{\mathrm{BG2}}_{\lvert\mathcal{C}_j\rvert}$	38	0	2	1	1	1	2	1	0	1	1	1	0	1	1	1	0	0

Table 3: Bit node groups for BG1 and BG2 for base rates 1/3 and 1/5, respectively.

groups with only a small number of elements.

Denote the set of BN groups as $\mathcal{B}$ and K_k the number of BNs in group $k \in \mathcal{B}$, e.g. for BG2 $K_5 = 2Z_c$. Each BN group will be processed separately. The BN processing buffer p_B^k of group k is defined as

$$p_B^k = \{\underbrace{r_{11}r_{21}\dots r_{K_k1}}_{1.CN}, \underbrace{r_{12}r_{22}\dots r_{K_k2}}_{2.CN}, \dots, \underbrace{r_{12}r_{22}\dots r_{K_kk}}_{lastCN}\} \tag{8}$$

Hence, $|p_B^k| = kK_k$, e.g, $Z_c = 128$, $|p_B^5| = 5 \cdot 2 \cdot 128 = 1024$.

Depending on the code rate, some parity bits are not being transmitted. For instance, for BG2 with code rate $R = 1/3$ the last $20Z_c$ bits are discarded. Therefore, the last 20 columns or the last $20Z_c$ parity check equation are not required for decoding. This means that the BN groups shown in table 3 are depending on the rate.

Listing 2: Example of BN processing for group 3 from `bnProcPc`.

```
// If elements in group move to next address
idxBnGroup++;

// Number of groups of 32 BNs for parallel processing
M = (lut_numBnInBnGroups[2]*Z)>>5;

// Set the offset to each CN within a group in terms of 16 Byte
cnOffsetInGroup = (lut_numBnInBnGroups[2]*NR_LDPC_ZMAX)>>4;

// Set pointers to start of group 3
p_bnProcBuf   = (__m128i*) &bnProcBuf  [lut_startAddrBnGroups   [idxBnGroup]];
p_llrProcBuf  = (__m128i*) &llrProcBuf [lut_startAddrBnGroupsLlr[idxBnGroup]];
p_llrRes      = (__m256i*) &llrRes     [lut_startAddrBnGroupsLlr[idxBnGroup]];

// Loop over BNs
for (i=0,j=0; i<M; i++,j+=2)
{
  // First 16 LLRs of first CN
  ymmRes0 = _mm256_cvtepi8_epi16(p_bnProcBuf[j]);
  ymmRes1 = _mm256_cvtepi8_epi16(p_bnProcBuf[j+1]);

  // Loop over CNs
  for (k=1; k<3; k++)
  {
    ymm0 = _mm256_cvtepi8_epi16(p_bnProcBuf[k*cnOffsetInGroup + j]);
    ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);

    ymm1 = _mm256_cvtepi8_epi16(p_bnProcBuf[k*cnOffsetInGroup + j+1]);
    ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);
  }

  // Add LLR from receiver input
  ymm0    = _mm256_cvtepi8_epi16(p_llrProcBuf[j]);
```

```
    ymmRes0 = _mm256_adds_epi16(ymmRes0, ymm0);

    ymm1    = _mm256_cvtepi8_epi16(p_llrProcBuf[j+1]);
    ymmRes1 = _mm256_adds_epi16(ymmRes1, ymm1);

    // Pack results back to epi8
    ymm0 = _mm256_packs_epi16(ymmRes0, ymmRes1);
    // ymm0     = [ymmRes1[255:128] ymmRes0[255:128] ymmRes1[127:0] ymmRes0
        [127:0]]
    // p_llrRes = [ymmRes1[255:128] ymmRes1[127:0] ymmRes0[255:128] ymmRes0
        [127:0]]
    *p_llrRes = _mm256_permute4x64_epi64(ymm0, 0xD8);

    // Next result
    p_llrRes++;
  }
}
```

The sum of the LLRs is carried out in 16 bit for accuracy and is then saturated to 8 bit for CN processing. Saturation after each addition results in significant loss of sensitivity for low code rates.

2.3 Mapping to the Processing Buffers

For efficient processing with the AVX instructions, the data is required to be aligned in a certain manner. That is the reason why processing buffers have been introduced. The drawback is that the results of the processing need to copied every time to the processing buffer of the next task. However, the speed up in computation with AVX more than makes up for the time wasted in copying data. The copying is implemented as a circular memcpy because every edge in the BG is a circular shift of a $Z \times Z$ identity matrix. Hence, a circular mempcy consists of two regular memcpys each copying a part of the Z values depending on the circular shift in the BG definition. The circular shifts are stored in `nrLDPC_lut.h` in arrays `circShift_BGX_ZX_CNGX`. In the specification there are only 8 sets of cirular shifts defined. However, the applied circular shift depends on Z, i.e. modulo Z. To avoid inefficient modulo operations in loops, we store the the circular shift values for every Z. Moreover, for convinience the arrays are already arranged depending on the CN group (CNG).

3 Performance Results

In this section, the performance in terms of BLER and decoding latency of the current LDPC decoder implementation is verified.

3.1 BLER Performance

In all simulations, we assume AWGN, QPSK modulation and 8-bit input LLRs, i.e. −127 until +127. The DLSCH coding procedure in 38.212 is used to encode/decode the TB and an error is declared if the TB CRC check failed. Results are averaged over at least 10 000 channel realizations.

The first set of simulations in Figure 3 compares the current LDPC decoder implementation to the reference implementation developed by Kien. This reference implementation is called *LDPC Ref* and uses the min-sum algorithm with 2 layers and 16 bit for processing. Our current optimized decoder implementation is referred to as *LDPC OAI*. Moreover, reference results provided by Huawei are also shown.

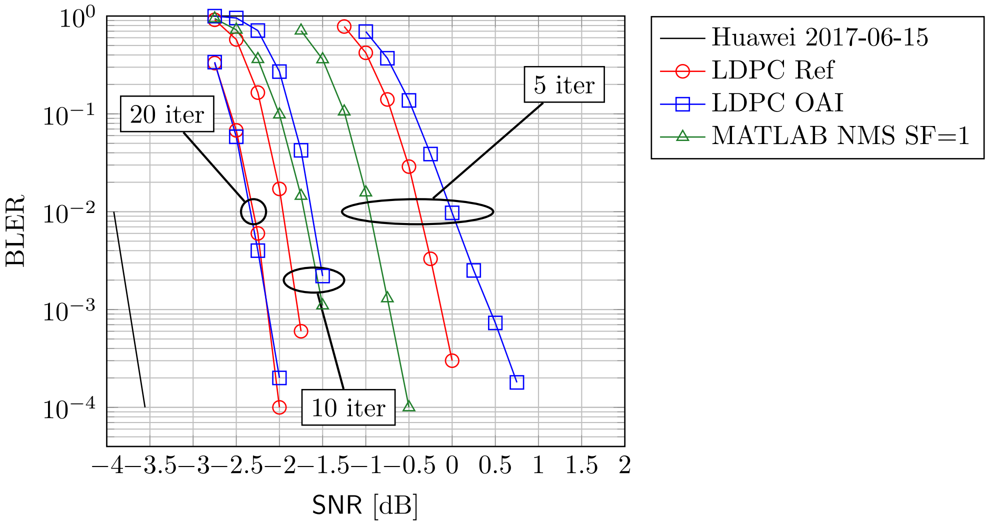

Figure 3: BLER vs. SNR, BG2, Rate=1/5, {5,10,20} Iterations, B=1280.

From Figure 3 it can be observed that the reference decoder outperforms the current implementation significantly for low to medium number of iterations. The reason is the implementation of 2 layers in the reference decoder, which results in faster convergence for punctured codes and hence requires less iterations to achieve a given BLER target. Note that there is a large performance loss of about 4 dB at BLER 10^{-2} between the Huawei reference and the current optimized decoder implementation with 5 iterations.

Moreover, there is a gap of about 1.5 dB between the results provided by Huawei and the current decoder with 20 iterations. The reason is the min-sum approximation algorithm used in both the reference decoder and the current implementation. The gap can be closed by using a tighter approximation like the min-sum with normalization or the lambda-min approach. Moreover, the gap closes for higher code rates which can be observed from Figure 4. The gap is only about 0.6 dB for 50 iterations.

The Matlab results denoted `MATLAB NMS` are obtained with the function `nrLDPCDecode` provided by the MATLAB 5G Toolbox R2019b. The following options are provided to the function: `'Termination','max','Algorithm','Normalized min-sum','ScalingFactor',1`. Furthermore, the 8-bit input LLRs are adapted to fit the dynamic range of `nrLDPCDecode` which is shown in Listing 3.

Listing 3: Input adaptation for MATLAB LDPC Decoder

```
maxLLR = max(abs(softbits));
rxLLRs = round((softbits/maxLLR)*127);
// adjust range to fit tanh use in decoder code
softbits = rxLLRs/3.4;
```

A scaling factor (SF) of 1 has been chosen to compare the results more easily with the *LDPC OAI* since the resulting check node processing is the same. However, the Matlab normelized min-sum algorithm uses layered processing and floating point operations. Thus, for the same number of iterations, the performance is significantly better than *LDPC OAI*, especially for small a number of iterations.

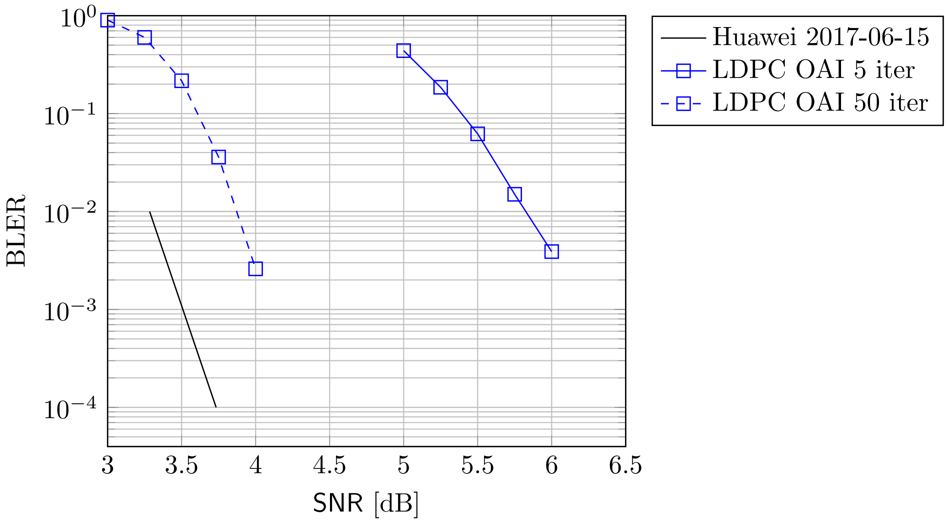

Figure 4: BLER vs. SNR, BG2, Rate=2/3, {5,50} Iterations, B=1280.

In Figure 5 we compare the performance of different algorithms using at most 50 iterations with early stopping if the parity check passes. The Matlab layered believe propagation (LBP) is used with unquantized input LLRs and performs the best since no approximation is done in the processing. Both NMS and offset min-sum (OMS) use a scaling factor and offset, respectively, that has been empirically found to perform best in this simulation setting. Theirs performance is very close to the BLP and OMS is slightly better than NMS. The performance of *LDPC OAI* is more than 1 dB worse mainly because of the looser approximation. Moreover, the NMS algorithm with SF=1 performs worst probably because the SF is not optimized for the input LLRs. From the results in Figure 5 we can conclude that the performance of the *LDPC OAI* can be significantly improved by adopting an offset min-sum approximation improving the performance to within 0.3dB of the Huawei reference curve.

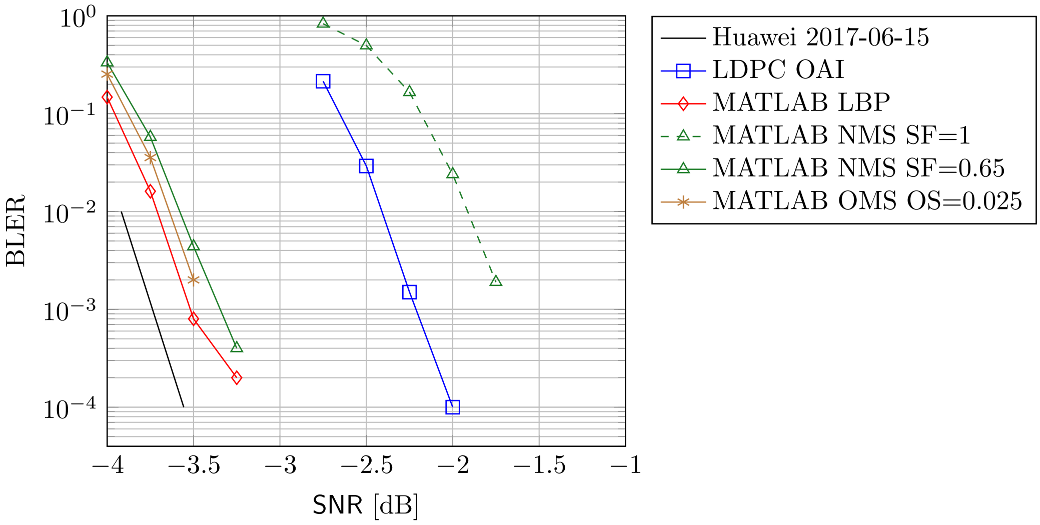

Figure 5: BLER vs. SNR, BG2, Rate=1/5, max iterations = 50, B=1280.

Figure 6 shows the performance of BG1 with largest block size of $B = 8448$ and highest code rate $R = 8/9$. From Figure 6 it can be observed that the performance gap is only about 0.3 dB if 50 iterations are used. However, for 5 iterations there is still a significant performance loss of about 2.3 dB at BLER 10^{-2}.

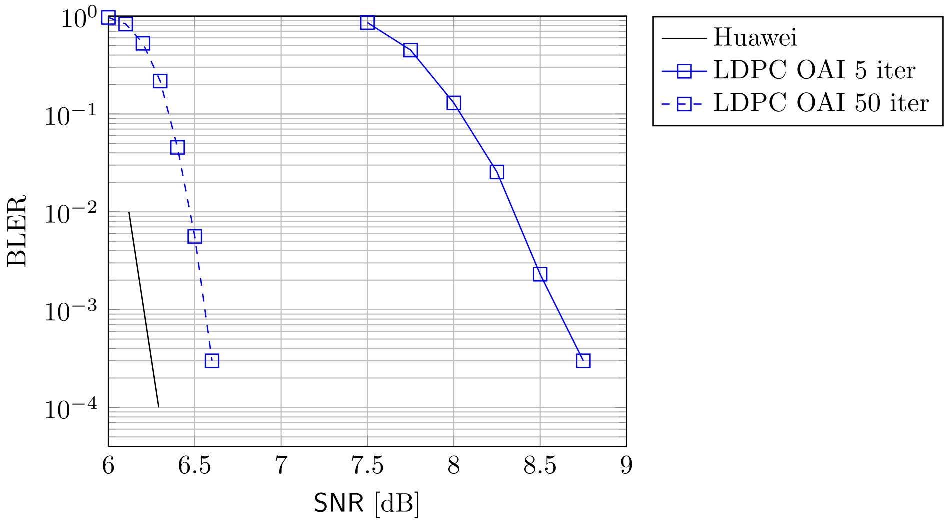

Figure 6: BLER vs. SNR, BG1, Rate=8/9 {5,50} Iterations, B=8448.

3.2 Decoding Latency

This section provides results in terms of decoding latency. That is, the time it takes the decoder to to finish decoding for a given number of iterations. To measure the run time of the decoder we use the OAI tool `time_meas.h`. The clock frequency is about 2.9 GHZ, decoder is run on a single core and the results are averaged over 10 000 blocks.

The results in Table 4 show the impact of the number of iterations on the decoding latency. It can be observed that the latency roughly doubles if the number of iterations are doubled.

Function	**Time [μs] (5 it)**	**Time [μs] (10 it)**	**Time [μs] (20 it)**
`llr2llrProcBuf`	0.5	0.5	0.5
`llr2CnProcBuf`	5.0	4.8	4.9
`cnProc`	12.4	23.0	42.7
`bnProcPc`	8.4	14.8	27.0
`bnProc`	5.5	10.1	19.0
`cn2bnProcBuf`	14.9	24.4	44.0
`bn2cnProcBuf`	10.5	17.8	31.8
`llrRes2llrOut`	0.3	0.3	0.3
`llr2bit`	0.2	0.2	0.2
Total	**58.5**	**97.1**	**172.6**

Table 4: BG2, Z=128, R=1/5, B=1280, LDPC OAI

Table 5 shows the impact of the code rate on the latency for a given block size and 5 iterations. It can be observed that the performance gain from code rate 1/3 to 2/3 is about a factor 2.

Table 6 shows the results for BG1, larges block size and different code rates. The latency difference betwee code rate 1/3 and code rate 2/3 is less than half because upper left corner of the

Function	Time [μs] (R=1/5)	Time [μs] (R=1/3)	Time [μs] (R=2/3)
`llr2llrProcBuf`	1.5	0.9	0.5
`llr2CnProcBuf`	6.0	4.1	2.2
`cnProc`	32.2	23.7	14.4
`bnProcPc`	21.2	12.1	5.5
`bnProc`	9.8	5.9	2.9
`cn2bnProcBuf`	23.3	13.9	6.8
`bn2cnProcBuf`	14.8	9.7	5.0
`llrRes2llrOut`	0.6	0.4	0.3
`llr2bit`	0.7	0.4	0.2
Total	**111.0**	**71.8**	**38.5**

Table 5: BG2, Z=384, B=3840, LDPC OAI, 5 iterations

PCM is more dense than the rest of the PCM.

Function	Time [μs] (R=1/3)	Time [μs] (R=2/3)	Time [μs] (R=8/9)
`llr2llrProcBuf`	2.1	1.2	0.9
`llr2CnProcBuf`	10.6	5.4	2.9
`cnProc`	89.8	66.3	50.0
`bnProcPc`	28.1	12.4	7.1
`bnProc`	17.1	8.1	4.8
`cn2bnProcBuf`	38.7	17.1	9.3
`bn2cnProcBuf`	25.6	12.7	7.2
`llrRes2llrOut`	0.8	0.4	0.3
`llr2bit`	0.9	0.4	0.3
Total	**214.6**	**124.6**	**83.6**

Table 6: BG1, Z=384, B=8448, LDPC OAI, 5 iterations

From the above results it can be observed that the data transfer between CNs and BNs takes up a significant amount of the run time. However, the performance gain due to AVX instructions in both CN and BN processing is significantly larger than the penalty incurred by the data transfers.

4 Parity Check and Early Stopping Criteria

It is often unnecessary to carry out the maximum number of iterations. After each iteration a parity check (PC) (1) can be computed and if a valid code word is found the decoder can stop. This functionality has been implemented and the additional overhead is reasonable. The PC is carried out in the CN processing buffer and the calculation complexity itself is negligible. However, for the processing it is necessary to move the BN results to the CN buffer which takes time, the overall overhead is at most 10% compared to an algorithm without early stopping criteria with the same number of iterations. The PC has to be activated via the define `NR_LDPC_ENABLE_PARITY_CHECK`.

5 Conclusion

The results in the previous sections show that the current optimized LDPC implementation full-fills the requirements in terms of decoding latency for low to medium number of iterations at the expense of a loss in BLER performance. To improve BLER performance, it is recommended to implement a layered algorithm and a min-sum algorithm with normalization. Further improvements upon the current implementation are detailed in the next section.

6 Future Work

The improvements upon the current LDPC decoder implementation can be divided into two categories:

1. Improved BLER performance
2. Reduced decoding latency

6.1 Improved BLER Performance

The BLER performance can be improved by using a tighter approximation than the min-sum approximation. For instance, the min-sum algorithm can be improved by adding a correction factor in the CN processing . The min-sum approximation in (5) is modified as

$$r_{ji} = \prod_{j' \in \mathcal{B}_i \setminus j} \operatorname{sgn} q_{ij'} \min_{j' \in \mathcal{B}_i \setminus j} |q_{ij'}| + w(q_{ij'}) \tag{9}$$

The correction term $w(q_{ij'})$ is defined as

$$w(q_{ij'}) = \begin{cases} c & \text{if} \\ -c & \text{if} \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where the constant c is of order 0.5 typically.

6.2 Reduced Decoding Latency

The following improvements will reduce the decoding latency:

- Adapt to AVX512
- Optimization of CN processing
- Implement 2/3-layers for faster convergence

AVX512: The computations in the CN and BN processing can be further accelerated by using AVX512 instructions. This improvement will speed-up the CN and BN processing by a approximately a factor of 2.

Optimization of CN Processing: It can be investigated if CN processing can be improved by computing two minima regardless of the number of BNs. Susequently, the (absolute) value fed back to the BN is one of those minima.

Layered processing: The LDPC code in NR always punctures the first 2 columns of the base graph. Hence, the decoder inserts LLRs with value 0 at their place and needs to retrieve those bits during the decoding process. Instead of computing all the parity equations and then passing the results to the BN processing, it is beneficial to first compute parity equations where at most one punctured BN is connected to that CN. If two punctured BNs are connected than according to (5), the result will be again 0. Thus in a first sub-iteration those parity equation are computed and the results are send to BN processing which calculates the results using only those rows in the PCM. In the second sub-iteration the remaining check equation are used. The convergence of this layered approach is much fast since the bit can be retrieved more quickly while the decoding complexity remains the same. Therefore, for a fixed number of iterations the layered algorithm will have a significantly better performance.

References

[1] R. Gallager, “Low-density parity-check codes,” *IRE Transactions on information theory*, vol. 8, no. 1, pp. 21–28, 1962.

[2] 3rd Generation Partnership Project, “Multiplexing and channel coding (Release 15),” 3GPP TS 38.212 V15.0.1, Tech. Rep., Mar. 2018.