J Sign Process Syst (2017) 89:209–224
DOI 10.1007/s11265-016-1184-8

Computational Complexity Analysis of FEC Decoding on SDR Platforms

Zhenzhi Wu[1,2] · Chen Gong[1] · Dake Liu[1,2]

Received: 22 October 2015 / Revised: 14 June 2016 / Accepted: 19 September 2016 / Published online: 15 October 2016
© The Author(s) 2016. This article is published with open access at Springerlink.com

Abstract The computational complexity evaluation is necessary for software defined Forward Error Correction (FEC) decoders. However, currently there are a limited number of literatures concerning on the FEC complexity evaluation using analytical methods. In this paper, three high efficient coding schemes including Turbo, QC-LDPC and Convolutional code (CC) are investigated. The hardware-friendly decoding pseudo-codes are provided with explicit parallel execution and memory access procedure. For each step of the pseudo-codes, the parallelism and the operations in each processing element are given. Based on it the total amount of operations is derived. The comparison of the decoding complexity among these FEC algorithms is presented, and the percentage of each computation step is illustrated. The requirements for attaining the evaluated results and reference hardware platforms are provided. The benchmarks of state-of-the-art SDR platforms are compared with the proposed evaluations. The analytical FEC complexity results are beneficial for the design and optimization of high throughput software defined FEC decoding platforms.

✉ Dake Liu
dake@bit.edu.cn

[1] School of Information and Electronics, Beijing Institute of Technology, 100081, Beijing, China

[2] Institute of Electrical Engineering, Linköping University, SE-581 83, Linköping, Sweden

Keywords SDR · FEC · GOPS · Computational complexity · Convolutional code · Turbo · LDPC · Viterbi · Layered decoding · BCJR

1 Introduction

High throughput Software Defined Radio (SDR) platform encounters extremely high computational complexity in FEC decoding. C.H. Kees [1] presented a roughly estimation on FEC workload for nowadays standards, and shows in graph that approximately 100-3000 Operations per Bit (OP/bit) is needed, occupying approximately 40 % of workload in baseband. The attainable throughputs for LDPC are revealed with GPU [2, 3], IBM Cell processor [4] and DSP [5]. Turbo decoding throughputs are provided with GPU platforms [6, 7] and DSP platform [8]. The cycle counts for Viterbi decoding are provided for general purpose processor [9, 10], SORA platform [9], SODA platform [10] and DVB-T receiver [11]. The arithmetic operations for Quasi-Cyclic Low-Density Parity-Check (LDPC) and Turbo are provided and compared in [12], which shows that QC-LDPC codes have significantly lower complexity than Turbo codes. However these results are provided without sufficient proof, especially for arbitrary configurations. Apart from SDR evaluations, the hardware based FEC complexity is given by [13] and [14], whereas the individual hardware result cases may be inaccurate for Software Defined Radio (SDR) platforms.

Among these proposals, few theoretical analysis on complexity is provided, therefore the relationship between the complexity and the algorithm is still unclear. Moreover, because most of cases are implementations with dedicated

configurations, the general complexity expression with arbitrary configurations (such as code length, parallelism) has not been exposed clearly, which hinders the designs and optimizations of the SDR platforms. For the algorithm researchers, the complexity results are also necessary for designing high efficient low complexity codes.

Based on these requirements, this research work aims to obtain a likely Giga Operations per Second (GOPS) or OP/bit low bound using analytical methods. The 'likely low bound' refers to an approximately low bound including the most probably memory access procedures and parallel execution procedures, which is more valuable for platform designers. The work consists of several aspects as following.

Firstly most of the papers addressing FEC decoding only concern on the algorithm level description which lacks of accuracy for the complexity evaluation. The addressing, memory access and shuffling network besides the arithmetic computation are all necessary to be considered. In addition, the parallel execution procedure for high throughput decoding needs to be taken into consideration as well. For these reasons, hardware-friendly highly accurate pseudo-codes are thus required and provided in this work.

Secondly there are many different decoding algorithm alternatives, and it is hard to cover all during evaluations. In this paper, the mainstream kernel functions and alternative decoding algorithms are listed and compared, and the ones with minimum operations are selected. Different from other papers providing the experimental results based on the hardware architecture details, this work tries to minimize the hardware specifics, and provides a platform independent result. The evaluation is performed on basic operation level (basic operations will be given in a following table) instead of the instruction level or hardware circuit level.

Thirdly, after the complexity is provided for each algorithm, different coding schemes are necessary to be compared using the parameters from wireless standards, which offer a general overview of the decoding efficiency. In addition, the operations for each decoding step are figured out as well.

The paper is organized as following. Section 2 lists all the pseudo-codes and the decoding procedure descriptions. Section 3 provides the operation analysis for each algorithm. Section 4 provides the comparison among the algorithms, and the operation cost on percentage for each decoding steps. Section 5 illustrates the reliable prototypes and assumptions for attaining the proposed results. Section 6 gives comparisons of proposed results with state-of-the-art SDR platforms.

2 Pseudo-Codes Analysis

To calculate the operations for each decoding step, the hardware friendly pseudo-codes are provided firstly. The parts of codes denoting with 'for x do in parallel' are able to be executed by independent Processing Elements (PEs) in parallel. The variables stored in memory are denoted with a 'bar' over the variable name, they need load and store procedure to access. Other variables without 'bar' are register variables which are only valid in the current parallel block and usually in small size. For a multiple PEs platform, except for the shared memory variable $\overline{L}(i, n)$ which is accessed by all PEs, other memory variables can be stored in each PE locally.

Let $\mathbf{x} = [x_0, x_1, \cdots, x_{N-1}]$ be the transmitted information codeword and $\mathbf{y} = [y_0, y_1, \cdots, y_{N-1}]$ be the received codeword. Let $LLR(n) = log(P(x_n = 1 \mid y_n)/P(x_n = 0 \mid y_n))$ be the likelihood-ratio (LLR) of each received message, which is the input of the algorithm. The decoded bits can be viewed as an estimation of $\mathbf{x}$ which is the output of the algorithm, and denoted as $\hat{\mathbf{x}} = [\hat{x}_0, \hat{x}_1, \cdots, \hat{x}_{N-1}]$.

2.1 LDPC Decoding

Many high-throughput wireless standards such as IEEE 802.16e and IEEE 802.11n apply Quasi-cyclic LDPC (QC-LDPC) because of its ability for intrinsic highly parallel execution. Hence we confine our research on QC-LDPC mainly. The efficient algorithms for QC-LDPC decoding are Belief-Propagation (also called Message Passing) and layered decoding [15]. Layered decoding method is proposed by updating the variable-node as soon as the check node updates for the current row are calculated instead of all the equations, and about half of the iterations can be saved. Hence the layered decoding is adopted for the following analysis, which is shown in Algorithm 1.

In Algorithm 1, A QC-LDPC code $QC(J, N)$ is processed, which has N decoded bits, J variable nodes (channel messages) and M check nodes ($M = J - N$). It can be described by a parity-check matrix $\mathbf{H}_{M\times J}$. The matrix $\mathbf{H}$ can be divided into $L \times C$ sub-matrices of degree Z, in which $L = M/Z$ and $C = J/Z$. $\mathbf{HB}_{L\times C}$ represents the base matrix with elements either zero or equals to the cyclic-shift value of the identity matrix. Let $\mathbf{S}_{r,k}$ represents the kth non-zero element (NZE) in row r of $\mathbf{HB}$, and the shift value can be obtained as $\mathbf{HB}_{r,\mathbf{S}_{r,k}}$, simplified as $\mathbf{R}_{r,k}$. S_r denotes the number of NZE in row r. A *p-layer* consists of Z layers which can be processed in parallel.

The decoding is performed in an iterative manner, and the maximum number of iterations is it_{max}. In each iteration

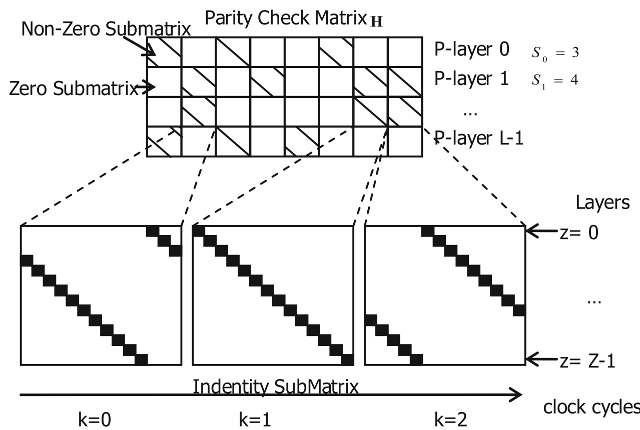


Figure 1 QC-LDPC parity check matrix and layered decoding routine.

L p-layers are processed one by one, where r denotes the index of the current p-layer being processed. In each p-layer, the NZEs are processed one by one sequentially, shown in Fig. 1.

The check-node update can be realized by Forward-Backward Recursion (FBR), and the principle can be briefly summarized as following. The LLR result of modulo-2 addition of two LLR values is given by $LLR(x \oplus y) = f(x, y) = log(1 + e^{x+y}) - log(e^x + e^y)$ [16]. The LLR for multiple input elements can be calculated using 2-input $f(x, y)$ function as $f(x_1, x_2, \cdots, x_n) = LLR(x_1 \oplus x_2 \oplus \cdots \oplus x_n) = LLR(((x_1 \oplus x_2) \oplus x_3) \oplus \cdots \oplus x_n) = f(f(f(x_1, x_2), x_3), \cdot, x_n)$. The check-node update can be represented by $\Lambda^z_{r,k} = LLR(\gamma^z_1 \oplus \gamma^z_2 \oplus \cdots \oplus \gamma^z_{k-1} \oplus \gamma^z_{k+1} \oplus \cdots \oplus \gamma^z_{S_r-1}) = f(\gamma^z_1, \gamma^z_2, \cdots, \gamma^z_{k-1}, \gamma^z_{k+1}, \cdots, \gamma^z_{S_r-1})$. The LLR value of the first k check nodes are calculated by $\alpha^z_k = f(f(f(\gamma^z_1, \gamma^z_2), \gamma^z_3), \cdot, \gamma^z_k)$. It can be calculated by Forward Recursion (FR) using $\alpha^z_k = f(\alpha^z_{k-1}, \gamma^z_k)$. The last k check nodes is originally calculated by $\beta^z_k = f(f(f(\gamma^z_{S_r-1}, \gamma^z_{S_r-2}), \gamma^z_{S_r-3})..., \gamma^z_k)$. They can be obtained by Backward Recursion (BR) following $\beta^z_k = f(\beta^z_{k+1}, \gamma^z_k)$, and finally the check node update process can be performed by $\Lambda^z_{r,k} = f(\alpha^z_k, \beta^z_{k+1})$. Further information related to FBR can be obtained from [17] and [16].

In FR part, Line 7 and Line 8 can be executed by a block data read operation and a block data permutation instead, where Z data are fetched together in normal sequence. The data are then shifted by a permutation network (cyclic-shifters) which is given by $I^z_{org} \leftarrow \overline{L}(z, \mathbf{S}_{r,k}), z \in [0, Z-1]$, and $I^z \leftarrow \Phi(I^z_{org}, \mathbf{R}_{r,k})$. Line 24 (along with 17) can be alternatively executed by a block transmission also, given by $O^z_{org} \leftarrow \Phi^{-1}(O^z, \mathbf{R}_{r,k})$, and $\overline{L}(z, \mathbf{S}_{r,k}) \leftarrow O^z_{org}$.

Algorithm 1 Parallel layered decoding for LDPC code

```
1  Initialization:
2  L̄(i,n) ≡ −LLR(nZ + i), i = 0,1,·,Z − 1
3  Iteration process:
4  for it = 0 to it_max − 1 do
5  |  for r ← 0 to L − 1 do
6  |  |  for k ← 0 to S_r − 1 do
7  |  |  |  for z ∈ [0, Z − 1] do in parallel
8  |  |  |  |  I^z ← L̄((z + R_{r,k})_{modZ}, S_{r,k})
9  |  |  |  |  ρ^z ← Λ̄^z_{r,k}
10 |  |  |  |  γ^z ← I^z − ρ^z
11 |  |  |  |  γ̄^z_k ← γ^z
12 |  |  |  |  ᾱ^z_k ← α^z
13 |  |  |  |  α^z ← f(α^z, γ^z)
14 |  |  |  end
15 |  |  end
16 |  |  for k ← S_r − 1 to 0 do
17 |  |  |  for z ∈ [0, Z − 1] do in parallel
18 |  |  |  |  γ^z ← γ̄^z_k
19 |  |  |  |  α^z ← ᾱ^z_k
20 |  |  |  |  Λ^z ← f(α^z, β^z)
21 |  |  |  |  Λ̄^z_{r,k} ← Λ^z
22 |  |  |  |  β^z ← f(β^z, γ^z)
23 |  |  |  |  O^z ← γ^z + Λ^z
24 |  |  |  |  L̄(z, S_{r,k}) ← O^{(z−R_{r,k})_{modZ}}
25 |  |  |  end
26 |  |  end
27 |  end
28 end
29 Hard output decision:
30 foreach n ∈ [0, N − 1] do
31 |  if LLR(n) ≥ 0 then
32 |  |  x̂_n ← 0
33 |  else
34 |  |  x̂_n ← 1
35 |  end
36 end
```

In Algorithm 1 and following Algorithms in this article, note that:

(1). The notation $\leftarrow$ represents 'assigned to', which means the calculation result on right side updates the variable on the left side.

(2). The register update procedure and memory access procedure are separately represented. For example, the forward path update $\overline{\alpha}^z_k \leftarrow f(\overline{\alpha}^z_k, \gamma^z)$ is split to $\overline{\alpha}^z_k \leftarrow \alpha^z$ and $\alpha^z \leftarrow f(\alpha^z, \gamma^z)$, where $\overline{\alpha}^z_k$ is a memory variable with k be the address. If there is no need to

store to memory (such as β), then no memory access procedure is needed.

(3). The foot mark k may be omitted for variables within the kth recursion body without confusion.

There is an alternative decoding approach named improved Min-Sum algorithm. In this method, the iteration process in Algorithm 1 is substituted by Algorithm 2. In the forward recursion, only five values are recorded, including the maximum and secondary maximum a-priori messages, the index of the maximum value, the sign of each message, and the product of all signs. The backward recursion part is redesigned in a non-recursion way. The a-posteriori messages are assigned as the maximum value or the secondary maximum value, with the product of signs from all other messages. It is a reduced complexity method and benefits for hardware design. However, for a programmable platform, the branch operation and sign operations may consume more resources. Therefore Algorithm 2 introduces more workload compared to Algorithm 1. Hence, Algorithm 1 is applied for the complexity evaluation.

Algorithm 2 Parallel layered decoding with improved min-sum method

```
1  Iteration process:
2  for it = 0 to it_max − 1 do
3    for r ← 0 to L − 1 do
4      for k ← 0 to S_r − 1 do
5        for z ∈ [0, Z − 1] do in parallel
6          m^z = ∞, n^z = ∞, p^z = 1
7          I^z ← L̄((z + R_{r,k})_{modZ}, S_{r,k})
8          ρ^z ← Λ̄^z_{r,k}
9          γ^z ← I^z − ρ^z
10         s^z ← sign(γ^z)
11         a^z ← abs(γ^z)
12         s̄_k^z ← s^z
13         p^z ← p^z · s^z
14         if a^z < m^z then
15           n^z ← m^z
16           m^z ← a^z
17           i^z ← k
18         else
19           if a^z < n^z then
20             n^z ← a^z
21         end
22       end
23     end
24     foreach k ∈ [0, S_r − 1] do
25       for z ∈ [0, Z − 1] do in parallel
26         s^z ← s̄_k^z
27         if k = i^z then
28           Λ^z ← μ · n^z · p^z · s^z
29         end
30         else
31           Λ^z ← μ · m^z · p^z · s^z
32         end
33         Λ̄^z_{r,k} ← Λ^z
34         O^z ← γ^z + Λ^z
35         L̄(z, S_{r,k}) ← O^{(z−R_{r,k})_{modZ}}
36       end
37     end
38   end
39 end
```

2.2 Turbo Decoding

Turbo is an efficient coding technique approaching to Shannon limit. Turbo meets the need of high-throughput wireless applications by its parallel decoding ability, and has been widely accepted in many standards such as 3GPP-LTE(A), HSPA, CDMA2000, and IEEE 802.16e. In this work, only 8-state Parallel Concatenated Convolutional Code (PCCC) Turbo is considered because most of the widely adopted standards (listed above) are based on it.

In the following, the BCJR (Bahl-Cocke-Jelinek-Raviv) algorithm with the parallel processing is investigated, shown in Algorithm 3. Let N denotes the information message length, which is encoded to $3N$ transmitted bits. Let $\overline{L}(i, n), i = 0, 1, 2$ denotes the $3n$th to $(3n + 2)$th data in $LLR(n)$. Because each iteration consists of two constituent maximum a-posteriori (MAP) decoding, we introduce a variable $r = 0, 1$ to distinguish the first half iteration (MAP1) and the second (MAP2). The first MAP decoding uses $\overline{L}(0, k)$, $\overline{L}(1, k)$ and a-posteriori messages from MAP2 $\overline{L}(i_0, k)$ as input, $\overline{L}(i_1, k)$ and $\overline{L}(i_L, k)$ as the output. Where k is the current FBR step. The second MAP decoding uses $\overline{L}(0, k_{inv})$, $\overline{L}(2, k)$ and $\overline{L}(i_1, k_{inv})$ as the input, $\overline{L}(i_0, k)$ and $\overline{L}(i_L, k)$ as the output. i_0 and i_1 are arbitrary constants used for bank labels, such as $i_0 = 3$, $i_1 = 4$. $\overline{L}(i_L, k)$ is only required at the last MAP2 procedure, in such occasion Line 37 is applied instead of Line 36 in Algorithm 3, otherwise Line 36 is skipped.

In parallel decoding, the received codewords are split into P slices with each length of L. P a-posteriori messages are read from a group of memory banks with both sequential and interleaved address. The data are arranged in a sequential pattern, referring that $\overline{L}(i, n)$ is stored in $\lfloor n/L \rfloor$th bank with address $mod(n, L)$. If accessed with interleaved address, the output should be reordered by a permuter. The interleaved address may also incur memory access conflict when more than one data being fetched are located in the

Algorithm 3 Parallel turbo decoding algorithm

1 Initialization:
2 **for** $n = 0$ **to** $N - 1$ **do**
3 $\quad \overline{L}(i, n) = LLR(3n + i), i = 0, 1, 2$
4 **end**
5 Iteration process:
6 **for** $it \leftarrow 0$ **to** $it_{max} - 1$ **do**
7 $\quad$ **for** $r \leftarrow 0$ **to** 1 **do**
8 $\quad\quad$ **for** $p \in [0, P - 1]$ **do in parallel**
9 $\quad\quad\quad$ **for** $k \leftarrow 0$ **to** $L - 1$ **do**
10 $\quad\quad\quad\quad k^p \leftarrow k + pL$
11 $\quad\quad\quad\quad k^p_{inv} \leftarrow \prod_r k^p$
12 $\quad\quad\quad\quad k^p_{pmu} \leftarrow \coprod_r k^p$
13 $\quad\quad\quad\quad \{L_a, r_0, r_1\}^p \leftarrow \{\Phi(\overline{L}(i_r, k^p_{inv}), k^p_{pmu}), \Phi(\overline{L}(0, k^p_{inv}), k^p_{pmu}), \overline{L}(1 + r, k^p))\}$
14 $\quad\quad\quad\quad \overline{\{L_a, r_0, r_1\}}^p_k \leftarrow \{L_a, r_0, r_1\}^p$
15 $\quad\quad\quad\quad$ **for** $S \in [0, T_s - 1]$ **do in parallel**
16 $\quad\quad\quad\quad\quad S'_i \leftarrow S/2 + iT_s/2, i = 0, 1$
17 $\quad\quad\quad\quad\quad \{u, v_0, v_1\}^p \leftarrow Trellis(S'_i, S)$
18 $\quad\quad\quad\quad\quad \gamma^p(S'_i, S) \leftarrow \frac{1}{2}\,(u^p \cdot L^p_a + r^p_0 \cdot v^p_0 + r^p_1 \cdot v^p_1)$
19 $\quad\quad\quad\quad\quad \overline{\alpha}^p_k(S) \leftarrow \alpha^p(S)$
20 $\quad\quad\quad\quad\quad \alpha^p(S) \leftarrow f\left(\gamma^p(S'_0, S) + \alpha^p(S'_0), \gamma^p(S'_1, S) + \alpha^p(S'_1)\right)$
21 $\quad\quad\quad\quad$ **end**
22 $\quad\quad\quad$ **end**
23 $\quad\quad\quad$ **for** $k \leftarrow L - 1$ **to** 0 **do**
24 $\quad\quad\quad\quad k^p \leftarrow k + pL$
25 $\quad\quad\quad\quad \{L_a, r_0, r_1\}^p \leftarrow \overline{\{L_a, r_0, r_1\}}^p_k$
26 $\quad\quad\quad\quad$ **for** $S \in [0, T_s - 1]$ **do in parallel**
27 $\quad\quad\quad\quad\quad S'_i \leftarrow 2S + i, i = 0, 1$
28 $\quad\quad\quad\quad\quad \alpha^p(S) \leftarrow \overline{\alpha}^p_k(S)$
29 $\quad\quad\quad\quad\quad \{u, v_0, v_1\}^p \leftarrow Trellis(S, S'_i)$
30 $\quad\quad\quad\quad\quad \gamma^p(S, S'_i) \leftarrow \frac{1}{2}\,(u^p \cdot L^p_a + r^p_0 \cdot v^p_0 + r^p_1 \cdot v^p_1)$
31 $\quad\quad\quad\quad\quad \beta^p_{old}(S) \leftarrow \beta^p(S)$
32 $\quad\quad\quad\quad\quad \beta^p(S) \leftarrow f\left(\gamma^p(S, S'_0) + \beta^p(S'_0), \gamma^p(S, S'_1) + \beta^p(S'_1)\right)$
33 $\quad\quad\quad\quad\quad L^p_o \leftarrow f_{(S,S')\in\sum +}\left(\beta^p_{old}(S') + \gamma^p(S, S') + \alpha^p(S)\right) - f_{(S,S')\in\sum -}\left(\beta^p_{old}(S') + \gamma^p(S, S') + \alpha^p(S)\right)$
34 $\quad\quad\quad\quad$ **end**
35 $\quad\quad\quad\quad L^p_e \leftarrow L^p_o - L_c r^p_0 - L^p_a$
36 $\quad\quad\quad\quad \overline{L}(i_{1-r}, \prod_r k^p) \leftarrow \Phi^{-1}(L^p_e, k^p_{pmu})$
37 $\quad\quad\quad\quad (\overline{L}(i_L, \prod_r k^p) \leftarrow \Phi^{-1}(L^p_o, k^p_{pmu}))$
38 $\quad\quad\quad$ **end**
39 $\quad\quad$ **end**
40 $\quad$ **end**
41 **end**
42 Decision:
43 **foreach** $n \in [0, N - 1]$ **do**
44 $\quad$ **if** $\overline{L}(i_L, n) \geq 0$ **then**
45 $\quad\quad \hat{x}_n = 1$
46 $\quad$ **else**
47 $\quad\quad \hat{x}_n = 0$
48 $\quad$ **end**
49 **end**

same memory bank. A contention-free interleaver, such as Quadratic Permutation Polynomial (QPP)-interleaver applied in 3GPP-LTE(A), can avoid such occasions when N is divisible with the parallelism P [18, 19]. In this case, the interleaving for each memory data is $\prod_r(n) = mod(Q_r(n), L)$, and the permutation route for each data is $\coprod_r(n) = Q_r(n)/L$. Where $Q_1(n) = n$, and $Q_2(n)$ is the QPP-interleaving function. For the case of other interleavers, the data can be rearranged to achieve conflict-free using graph coloring algorithm [20] or annealing procedure [21]. From them the addresses $\prod_r(n)$ and permutation routes $\coprod_r(n)$ are obtained as well. The permutation routes are needed for both MAP1 and MAP2. The memory accesses and permutation procedures are still the same as conflict-free case.

The BCJR algorithm is applied for each MAP half iteration. It consists of three main steps: FR, BR and extrinsic a-posteriori calculation. In the forward recursion, the branch metrics are calculated, and previous α metrics are fetched. Then the α for each trellis step k is calculated using kernel function and then stored. In addition, the input data for all k are stored in local in order to be supplied in BR. BR begins at the end of FR and β is calculated. The a-posteriori and extrinsic messages for the next half iteration are obtained. In each step, S represents the current trellis state, and $S'_i, i = 0, 1$ represent the previous states connected to the current state. The function $Trellis(S'_i, S)$ outputs the information sequence u, the first output branch v_0 (v_0 equals to u when component code is a Systematic Convolutional Code) and second output branch v_1. $(S, S') \in \sum+$ denotes all the branches which output bit 1, and $(S, S') \in \sum-$ denotes all the branches which output bit 0.

For parallel decoding, α and β are discontinuous when separated into SISOs and decoded in parallel. Therefore Next Iteration Initialization (NII) [7] also called State Metric Propagation (SMP) [19] method is applied. The FR initial value $\overline{\alpha}_0^p$ is the final FR value $\overline{\alpha}_{L-1}^{p-1}$ of the previous iteration, and the backward initial value $\overline{\beta}_{L-1}^p$ is the final BR value $\overline{\beta}_0^{p+1}$ at the previous iteration, depicted in Fig. 2.

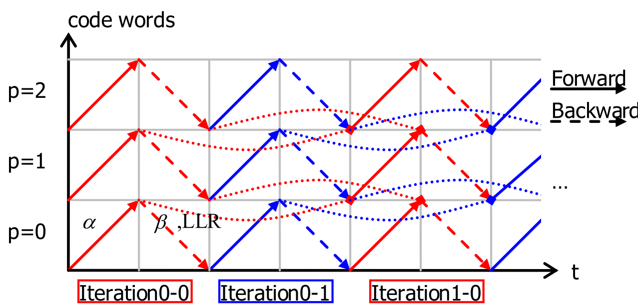


Figure 2 Turbo decoding NII message passing among processing elements.

If there is no previous iteration, the initial values are set to zeros. The messages may come from neighbour PEs, which lead to inter-PE data transfers or global memory accesses.

2.3 Convolutional Code Viterbi Decoding

Let $CC(m, 1, M)$ denote a CC with code rate $1/m$, information bits (decoded bits) N and constraint length M ($T_s = 2^{M-1}$ states). The algorithm description is shown in Algorithm 4. The channel messages are split to several blocks with overlapped area with the size of the traceback (TB) length, and decoded by separate PEs. The overlapped area is for traceback procedure for each PE.

Algorithm 4 Parallel viterbi algorithm

```
1  Initialization:
2  for n = 0 to N − 1 do
3  |   L̄(i, n) = LLR(mn + i), i = 0, 1, ·, m − 1
4  end
5  ACS:
6  for p ∈ [0, P − 1] do in parallel
7  |   for k ← 0 to L + L_TB − 1 do
8  |   |   k^p ← k + pL
9  |   |   l_i^p ← L̄(i, k^p), i = 0, ·, m − 1
10 |   |   for S ∈ [0, T_s − 1] do in parallel
11 |   |   |   S'_i ← S/2 + iT_s/2, i = 0, 1
12 |   |   |   v(S'_i, S) ← Trellis(S'_i, S)
13 |   |   |   γ^p(S'_i, S) ← dist(l^p, v(S'_i, S)), i = 0, 1
14 |   |   |   {α^p(S), sel^p(S)} ← f(γ^p(S'_0, S) + α^p(S'_0),
15 |   |   |   γ^p(S'_1, S) + α^p(S'_1))
16 |   |   |   TB̄_k^p(S) ← sel^p(S)
17 |   |   end
18 |   end
19 end
20 TB:
21 for p ∈ [0, P − 1] do in parallel
22 |   for k ← L + L_TB − 1 to 0 do
23 |   |   k^p ← k + pL
24 |   |   β^p ← TB̄_k^p(S^p)
25 |   |   if β^p = 1 then
26 |   |   |   S^p ← S^p/2
27 |   |   else
28 |   |   |   S^p ← S^p/2 + T_s/2
29 |   |   end
30 |   |   if k < L then
31 |   |   |   x̂_{k^p} ← LSB(S^p)
32 |   |   end
33 |   end
34 end
```

Branch Metric Update (BMU) and Add-Compare-Select (ACS) are the two main kernels in FR. In each FR step k, the BMU is performed to obtain the Euclidean Distance between the received messages and local trellis outputs. The distances (branch metrics) are added with the previous path metrics. Two branches S'_0 and S'_1 connecting to current state S are compared and the one has smaller metric

is selected (the transition bit is recorded as sel). After k reaches $L + L_{TB}$, the traceback procedure starts. The survival state calculated in previous step is applied to address the survival path memory and get the transition bit β. The survival state is updated by using the previous state and current β. This procedure performs recursively. Finally the Least Significant Bit (LSB) of S^p is stored back as the decoded bits.

2.4 Low Complexity Kernel Functions

The original kernel function $f(x, y)$ for LDPC and Turbo are hard to be implemented for the fixed data format and sensitive to the quantization error. Therefore several frequently used low complexity approximations are discussed here in detail. For LDPC, $f(x, y)$ can be the addition of a linear part and a correction part, given by $f(x, y) = f_b(x, y) + f_c(x, y)$. Following Jacobian approach, $f_b(x, y) = sign(x) \cdot sign(y) \cdot min(|x|, |y|) = max(x, y) - max(x + y, 0)$, and $f_c(x, y) = log(1 + e^{-|x+y|}) - log(1 + e^{-|x-y|})$. $f_c(x, y)$ can be implemented by a Look-Up-Table (LUT) or polynomial approximation, shown as f_{cL1} and f_{cL2} in Table 2. Turbo kernel function is part of the LDPC kernel, and the low complexity methods applied for LDPC is also suitable for Turbo.

If only the linear part is used with no correction part, it is called min-sum algorithm. A refined version of min-sum is the offset-min-sum algorithm, which is defined as

$$\Lambda_k = \prod_{i \neq k, i \in 0:S_r-1} sign(\gamma_i) \cdot \max \left\{ \min_{i \neq k, i \in 0:S_r-1} |\gamma_i| - \mu, 0 \right\}$$

Where μ is a positive small constant. Similar to it, scaled-min-sum algorithm is an alternative approach, which is given by

$$\Lambda_k = \prod_{i \neq k, i \in 0:S_r-1} \mu \cdot sign(\gamma_i) \cdot \min_{i \neq k, i \in 0:S_r-1} |\gamma_i|$$

Where μ is a scaling factor (less than 1). With these methods, the extrinsic messages Λ are updated with offset-/scaled-min-sum function, whereas FR messages α and BR messages β are updated with the original min-sum function.

In summary, the base part $f_b(x, y)$ for the three algorithms can be chosen from Table 1, and the correction part $f_c(x, y)$ can be chosen from Table 2.

3 Computational Analysis

3.1 Proposed Evaluation Method and Platform Independent Assumptions

The computational complexity is evaluated by Giga Operations per Second (GOPS) or Operations per decoded bit (OP/bit), where the 'Operation' refers to the basic operations listed in Table 3. They are the hardware architecture unrelated basic computation units for constructing the pseudo-codes, and all of them cost one operation. Therefore the number of operations can be derived from the pseudo-codes based on these unit operations. One operation may need several instructions to be performed according to the instruction-set specification. The 'Complex arithmetic' category in the table is a special class of computations which requires much more computational resources and floating point support. They are applied for comparing kernel function alternatives only.

There are several assumptions needed to achieve a platform independent evaluation.

(1). To unify the evaluation of branch cost, it is assumed that all the loops are unrolled and the loop branch overhead is zero;
(2). The memory capacity is assumed to be sufficient;
(3). The permutation can be realized by a likely 'load' instruction with the permute route as the offset address, which cost one operation. The permutation for a set of data may also be executed by a dedicated hardware such as crossbar network;
(4). The data are assumed in floating format or fixed format with enough datawidth, hence no overflow protection is included.

In the following analysis, the number of operations of kernel function $f(x, y)$ alternatives is analysed at first, then

Table 1 Base part alternatives in kernel function.

Algorithm	Name	Function
Turbo	f_{bT}	$max(x, y)$
LDPC	f_{bL1}	$max(x + y, 0) - max(x, y)$ [15]
	f_{bL2}	$sign(x) \cdot sign(y) \cdot min(\|x\|, \|y\|)$ [22]
	f_{bL3}	$sign(x) \cdot sign(y) \cdot$ $max\ (min(\|x\|, \|y\|) - \mu, 0)$
	f_{bL4}	$\mu \cdot sign(x) \cdot sign(y) \cdot min(\|x\|, \|y\|)$
Viterbi	f_{bV}	$min(x, y)$

Table 2 Correction part alternatives in kernel function.

Algorithm	Name	Function
Turbo	f_{cT1}	$max(\frac{5}{8} - \frac{\|x+y\|}{4}, 0)$
	f_{cT2}	$log(1 + e^{-\|x-y\|})$ [23]
	f_{cT3}	0
LDPC	f_{cL1}	$max(\frac{5}{8} - \frac{\|x+y\|}{4}, 0) - max(\frac{5}{8} - \frac{\|x-y\|}{4}, 0)$ [15]
	f_{cL2}	$log(1 + e^{-\|x+y\|}) - log(1 + e^{-\|x-y\|})$ [22]
	f_{cL3}	0
Viterbi	f_{cV}	0

Table 3 List of unit operations for the computational complexity evaluation.

Category	Operation definition
Arithmetic computation	$c = a + b, c = a - b, c = a \cdot b, c = \|a\|$
	$\{c, idx\} = max(a, b), \{c, idx\} = min(a, b)$
	$c = -a, c = sign(a), c = a \cdot const^*$
	$c = a/2, c = a/4$
Complex arithmetic	$c = log(x), c = e^x$
LUT access	$c = LUT(a, b), c = LUT(a)$
Memory access	load a datum, store a datum, permute a datum
Branch	if-else

Note*: Although const multiplication may require multiple pipeline stages, when data dependency is proper sheduled, one equivalent clock cycle (one operation) processing is attainable.

Table 5 Operations for LDPC forward recursion kernel.

Type	Equation	Operations
Address LUT	$\mathbf{R}_{r,k}, \mathbf{S}_{r,k}$	2
Address comp.	-	0
Memory read	$I^z_{org} \leftarrow \overline{L}(z, \mathbf{S}_{r,k}), \rho^z \leftarrow \overline{\Lambda}^z_{r,k}$	$Z \times 2$
Memory write	$\overline{\gamma}^z_k \leftarrow \gamma^z, \overline{\alpha}^z_k \leftarrow \alpha^z$	$Z \times 2$
Permutation	$I^z \leftarrow \Phi(I^z_{org}, \mathbf{R}_{r,k})$	$1 \times Z$
Kernel	$\alpha^z \leftarrow f(\alpha^z, \gamma^z)$	$Z \times f$
Other arith. comp.	$\gamma^z \leftarrow I^z - \rho^z$	$Z \times 1$

Note: (1). In LDPC decoding, the minimum kernel complexity is $f = 4$; (2). The operations is represented as the number of PEs times the operations in each PE.

the number of operations of single FR/BR step is evaluated. With the number of FR/BR iterations, the total decoding complexity is derived at last.

3.2 LDPC Decoding Computational Complexity Analysis

(1). Kernel operation

The operations for the possible kernel functions are listed in Table 4. It shows that the minimum number of operations is 4, whereas all the kernels can be calculated within 9 operations. The original function needs exponential and logarithm calculation, hence approximation kernels with simple operations and LUTs are recommended to be applied. For calculating f_{cL1} and f_{cL2}, LUTs for function $max(\frac{5}{8} - \frac{|x+y|}{4}, 0)$, $max(\frac{5}{8} - \frac{|x-y|}{4}, 0)$, $log(1+e^{-|x+y|})$, $log(1+e^{-|x-y|})$ are assumed to be provided. The addition operation for connecting f_b (linear part) and f_c (correction part) are counted in. In following, the 'min-sum' solution with $f = 4$ is chosen for the overall decoding procedure complexity estimation.

(2). Recursion kernel

We divide the Algorithm into FR part and BR part, and then estimate them separately. In Algorithm 1, Line 7 to Line 15 is the FR part. In Table 5, the operations for FR part are summarized, where block transmission is applied. The total operations in FR part are calculated as $FR = 2 + (6 + f) \cdot Z$.

From Line 16 to Line 26 in Algorithm 1 is the BR part. Table 6 shows the summary of operations, and the number of operations in BR part is $BR = (6 + 2f) \cdot Z$.

(3). Loop structure and total operations

There are up to it_{max} iterations. In each iteration L p-layers (rows in **HB**) are processed sequentially. In each row, there are totally S_r NZEs. For an irregular code, S_r is aligned to the maximum number of NZEs among all p-layers. This approximation is beneficial for parallel FR/BR alignment, and also with

Table 4 Operations for calculating LDPC $f(x, y)$ kernels.

Name	Kernel	Summary	Total
Original	$f(x, y)$	add/sub=4, exp=3, log=2	9
Min-sum	$f_{bL1}(+f_{cL3})$	max=2, add/sub=2	4
LUT-min-sum	$f_{bL2}(+f_{cL3})$	abs=2, sign=2, min=1, mul=2	7
	$f_{bL1} + f_{cL1}$	base=4, LUT=2, add/sub=2	8
	$f_{bL1} + f_{cL2}$	base=4, LUT=2, add/sub=2	8
Offset -min-sum	f_{bL3}	abs=2, sign=2, max=1 min=1, sub=1, mul=2	9
Scaled -min-sum	f_{bL4}	abs=2, sign=2, min=1 mul=3	8

Table 6 Operations for LDPC backward recursion kernel.

Type	Equation	Operations
Address LUT	-	0
Address comp.	-	0
Memory read	$\gamma^z \leftarrow \overline{\gamma}_k^z, \alpha^z \leftarrow \overline{\alpha}_k^z$	$Z \times 2$
Memory write	$\overline{\Lambda}_{r,k}^z \leftarrow \Lambda^z$	$Z \times 2$
	$\overline{L}(z, \mathbf{S}_{r,k}) \leftarrow O_{org}^z$	
Permutation	$O_{org}^z \leftarrow \Phi^{-1}(O^z, \mathbf{R}_{r,k})$	$1 \times Z$
$f(x, y)$ Kernel	$\beta^z \leftarrow f(\beta^z, \gamma^z)$	$Z \times 2f$
	$\Lambda^z \leftarrow f(\alpha^z, \beta^z)$	
Other arith. comp.	$O^z \leftarrow \gamma^z + \Lambda^z$	$Z \times 1$

the consideration that the S_r difference among p-layers is at most one for wireless standards IEEE 802.11n and IEEE 802.16e. Therefore the total decoding operations is

$$\begin{aligned} OP_{LDPC} &= it_{max} \cdot L \cdot S_r \cdot (FR + BR) \\ &= it_{max} \cdot L \cdot S_r \cdot (2 + (12 + 3f) \cdot Z) \end{aligned} \quad (1)$$

Note that the statistics include the decoding kernel only. Early termination and correction check are not counted in. The data input and output procedure is assumed to be finished by DMA instead of processors. Therefore the data load/store operation cost is not included. It is assumed in the deduction that Z PEs are available. However as long as the number of PEs P is less than or equal to Z, the total number of operations is the same, whereas the Z layers may be updated in partially parallel if PEs are not abundant.

For the case of the maximum throughput in IEEE 802.11n standard, the configuration is $L = 4$, $S_r = 20$, and $Z = 81$. Along with the selected parameters $f = 4$ and $it_{max} = 6$, there are 934,080 operations for the 1620 decoded bits (1944 channel bits). Hence the computational complexity is 577 OP/bit. For a throughput of 450 Mb/s, 259 GOPS would be consumed.

Following similar way the computational complexity for Algorithm 2 can be calculated. It shows that Algorithm 2 consumes $6Z$ additional operations than Algorithm 1 in the Check Node Update kernel (FBR kernel). Therefore we recommend Algorithm 1 for software defined (SD) decoding.

3.3 Turbo Decoding Computational Complexity Analysis

Table 7 shows the alternatives of Turbo kernel function $f(x, y)$. In this table, we assume that LUTs are available for $max(\frac{5}{8} - \frac{|x+y|}{4}, 0)$ and $log(1 + e^{-|x-y|})$. In most of Turbo implementations the max-log-MAP algorithm is applied, and is therefore selected for the following decoding complexity evaluation.

Table 7 Operations for calculating turbo $f(x, y)$ kernels.

Name	Kernel	Summary	Total
log-MAP	$f(x, y)$	exp=2, add=1, log=1	4
Max-log-MAP	$f_{bT}(+f_{cT3})$	max=1	1
LUT-log-MAP	$f_{bT} + f_{cT1/2}$	base=1, LUT=1, add=1	3

In Algorithm 3, forward recursion is in Line 9 to Line 22. It contains the accesses of memory with permutation, branch metric update and forward recursions. Table 8 summarized the operations in detail.

When the constituent code is systematic such as in 3GPP standard, where $u = v_0$, a simplified solution for BM calculation is shown following.

Step 1. Calculating the trellis constraints, following $t_0[0] = S[0] \oplus S[2]$, $t_0[1] = S[0] \oplus S[1]$, $t_1[0] = S[0] \oplus S[2] \oplus 1$, and $t_1[1] = S[0] \oplus S[1] \oplus 1$. Where $\oplus$ represents XOR logic.

Step 2. Obtaining g_0 and g_1 by $p = L_a + r_0$, $g_0 = (p + r_1)/2$, and $g_1 = (-p + r_1)/2$.

Step 3. Output the branch messages following the Table 10.

Although this method has the similar number of operations as the original method, for all 16 branches ($T_s = 8$) there are only four independent branch metric values associated with all possible t_i. In addition, two of them are the negative numbers of the other two. Therefore the BM calculation in FR/BR consumes 8 operations for each step in each PE.

It is derived from Tables 8 and 9 that the total number of FR operations is $(19 + (3 + f) \cdot T_s) \cdot P$, and BR is $(17 + (7 + 3f) \cdot T_s) \cdot P$. With the number of half iterations ($2it_{max}$), the recursion window length L, and the parallelism P (where $L \cdot P = N$), the total operations are calculated by

$$\begin{aligned} OP_{Turbo} &= 2it_{max} \cdot L \cdot (FR + BR) \\ &= 2it_{max} \cdot N \cdot (36 + (10 + 4f) \cdot T_s) \end{aligned} \quad (2)$$

Take 3GPP-LTE(A) Turbo ($T_s = 8$) with $N = 6144$ decoded bits as an example. When max-log-MAP kernel function ($f = 1$) and $it_{max} = 6$ are selected, the total operations for the decoding are 10,911,744. Therefore the computational complexity is 1776 OP/bit (Table 10). For a decoding throughput of 150 Mbit/s, the computation would be 266 GOPS. Note that in this evaluation, no pipeline stall caused by data dependency or memory access conflict is counted in. The Inter-PE message passing only happens at the border of the recursion shown in Fig. 2. The complexity contribution of it is negligible, and therefore not included.

Table 8 Operations for turbo forward recursion kernel.

Type	Equation	PNum. × OP/Par.
Address LUT	$k_{inv}^p \leftarrow \prod_r k^p, k_{pmu}^p \leftarrow \coprod_r k^p$	$P \times 2$
Addr. Comp.	$k^p \leftarrow k + pL$	$P \times 1$
Memory read	$M_1^p \leftarrow \overline{L}(i_r, k_{inv}^p), M_2^p \leftarrow \overline{L}(0, k_{inv}^p),$ $M_3^p \leftarrow \overline{L}(1+r, k^p)$	$P \times 3$
Memory write	$\overline{\{L_a, r_0, r_1\}}_k^p \leftarrow \{L_a, r_0, r_1\}^p$	$P \times 3$
	$\overline{\alpha}_k^p(S) \leftarrow \alpha^p(S)$	$P \cdot T_s \times 1$
Permutation	$\{L_a, r_0, r_1\}^p \leftarrow \{\Phi(M_1^p, k_{pmu}^p), \Phi(M_2^p, k_{pmu}^p), M_3^p)\}$	$2 \times P$
Kernel BM	$S_0' \leftarrow S/2, S_1' \leftarrow S/2 + T_s/2$	0
	$\{u, v_0, v_1\}^p \leftarrow Trellis(S_i', S)$	0
	$\gamma^p(S_i', S) \leftarrow \frac{1}{2}\left(u^p \cdot L_a^p + r_0^p \cdot v_0^p + r_1^p \cdot v_1^p\right)$	$P \times 8$
Kernel FR	$b^p(S_i', S) \leftarrow \gamma^p(S_i', S) + \alpha^p(S_i'), i = 0, 1$	$2P \cdot T_s \times 1$
	$\alpha^p(S) \leftarrow f\left(b^p(S_0', S), b^p(S_1', S)\right)$	$P \cdot T_s \times f$

Note: (1). PNum. refers to the number of PEs, OP/Par. refers to the operations in each PE;
(2). In Turbo decoding, the minimum kernel complexity is $f = 1$.

If parallelism P increases, the decoding latency would be reduced. In such case the amount of complexity increasing is negligible whereas the bit error rate performance may degrade due to the discontinuity of recursion messages. More iterations can be applied to reduce the degradation which results in the linear increasing of total operations.

3.4 CC Decoding Computational Complexity Analysis

The distance function $dist(r, v)$ is originally implemented by Euclidean distance function $dist(r, v) = \sum_i (r_i - v_i)^2, i = 0, \cdots, m-1$. Because $\sum_i r_i^2$ and $\sum_i v_i^2$ are the same for all transition branches. After dropping these terms, the remaining part is $-2\sum_i r_i \cdot v_i$, and -2 is a constant which can be avoided without changing the relative value. In addition, v_i is the m local trellis transition bits containing at most 2^m combinations, and half of them can be obtained by a negative operation from the other half (consumes totally 2^{m-1} operations). Therefore the 2^{m-1} possible metrics are calculated in advance. For each metric calculation, $m-1$ additions/subtractions are required. Therefore the $dist(r, v)$ calculation for all branches in a trellis step consumes $(m-1) \cdot 2^{m-1} + 2^{m-1} = m \cdot 2^{m-1}$ in total. If the pre-calculation is not applied, m multiplications (or selections) and $m-1$ additions are needed for each branch metric calculation.

Table 9 Operations for turbo backward recursion kernel.

Type	Equation	PNum. × OP/Par.
Addr. Comp.	$k^p \leftarrow k + pL$	$P \times 1$
Memory read	$\{L_a, r_0, r_1\}^p \leftarrow \overline{\{L_a, r_0, r_1\}}_k^p$	$P \times 3$
	$\alpha^p(S) \leftarrow \overline{\alpha}_k^p(S)$	$P \cdot T_s \times 1$
Memory write	$\overline{L}(i_{1-r}, k_{inv}^p) \leftarrow L_w^p$, or $\overline{L}(i_L, k_{inv}^p) \leftarrow L_w^p$	$P \times 1$
Permutation	$L_w^p \leftarrow \Phi^{-1}(L_e^p, k_{pmu}^p)$, or $L_w^p \leftarrow \Phi^{-1}(L_o^p, k_{pmu}^p)$	$1 \times P$
Kernel BM	$S_0' \leftarrow 2S, S_1' \leftarrow 2S + 1$	0
	$\{u, v_0, v_1\}^p \leftarrow Trellis(S, S_i')$	0
	$\gamma^p(S, S_i') \leftarrow \frac{1}{2}\left(u^p \cdot L_a^p + r_0^p \cdot v_0^p + r_1^p \cdot v_1^p\right)$	$P \times 8$
Kernel BR	$b^p(S, S_i') \leftarrow \gamma^p(S, S_i') + \beta^p(S_i'), i = 0, 1$	$2P \cdot T_s \times 1$
	$\beta^p(S) \leftarrow f\left(b^p(S, S_0'), b^p(S, S_1')\right)$	$P \cdot T_s \times f$
Kernel LLR	$L_o^p \leftarrow f_{(S,S') \in \sum^+} W^p(S', S) - f_{(S,S') \in \sum^-} W^p(S', S)$	$2P \cdot T_s \times f$, $P \times 1$
	$W^p(S', S) \leftarrow \beta_{old}^p(S') + \gamma^p(S, S') + \alpha^p(S)$	$2P \cdot T_s \times 2$
	$L_e^p \leftarrow L_o^p - L_c r_0^p - L_a^p$, with $L_c = 1$	$P \times 2$

Table 10 Complexity reduced turbo branch metric calculation.

$t_i[0]$	$t_i[1]$	$b(S_i', S)$
0	0	$-g_0$
0	1	g_1
1	0	$-g_1$
1	1	g_0

Table 12 Operations for CC backward path.

Type	Equation	PNum. × OP/Par.
Address comp.	$k^p \leftarrow k + pL$	$P \times 1$
Memory read	$\beta^p \leftarrow \overline{TB}_k^p(S^p)$	$P \times 1$
Memory write	$\hat{x}(k^p) \leftarrow LSB(S^p)$ when $k < L$	$P \times 1$
Kernel TB	if $\beta^p = 1$ then $S^p \leftarrow S^p/2$	$P \times 3.5$
	else $S^p \leftarrow S^p/2 + T_s/2$	

From Tables 11 and 12, we can conclude that for each step, there are $P \cdot (1 + m + m \cdot 2^{m-1} + (3 + f) \cdot T_s)$ operations to be applied in ACS and $6.5P$ operations in TB. For a codeword with length $N = L \cdot P$, The decoding procedure contains $L + L_{TB}$ ACS and TB steps, therefore the overall operations are

$$OP_{CC} = (7.5 + m + m \cdot 2^{m-1} + (3 + f) \cdot T_s) \cdot P \cdot (L + L_{TB})$$
$$= (7.5 + m + m \cdot 2^{m-1} + 4T_s) \cdot P \cdot (L + L_{TB}) \quad (3)$$

For a CC(2, 1, 7) ($m = 2$, $T_s = 64$), with decoded bits $N = 2048$, traceback length $L_{TB} = 35$ and parallelism $P = 8$, the total operations are 627,396, equivalent to 306 OP/bit. In the case of larger trellis constraint length code such as CC(3, 1, 9) ($m = 3$, $T_s = 256$), with $N = 2048$, $L_{TB} = 45$ and $P = 8$, the total operations are 2,519,972, and equivalent to 1230 OP/bit.

Because there is an overlapped area in each PE for the traceback computing which costs redundant computations, the parallelism P impacts on the computational complexity obviously. For the previous CC(3,1,9) example, the decoding needs extra 15 % of operations than the single PE ($P = 1$) configuration.

4 Complexity Comparison

To reveal the relationship of computational complexity among these algorithms, the number of operations is shown in Fig. 3, wherein all the LDPC configurations in IEEE 802.11n and IEEE 802.16e standards (R=1/2 to R=5/6), 8-state Turbo (3GPP-LTE Turbo R=1/3) and CC are compared. Six iterations are applied for LDPC and Turbo decoding. Several conclusions are summarized from the comparison: (1). For all modes, the decoding complexity is approximately proportional to the number of the decoded bits; (2). When the cost of operations is the same, LDPC may offer even 2-3 times of throughput than that of Turbo; (3). The complexity of CC(3, 1, 9) is slightly higher than LDPC with 6 iterations; (4). IEEE 802.11n and IEEE 802.16e LDPC have the similar complexity; (5). CC with small constraint length has the minimum complexity, whereas Turbo decoding consumes much more operations than other coding types. In Fig. 4, the relationship of throughput and computing cost as GOPS is provided for several typical codes. The 1 Gbps Turbo consumes approximately 2000 GOPS, along with other auxiliary workload it would be 2-3 times more, which is currently difficult to be performed in a single chip. However 1Gbps LDPC requires less than 600 GOPS which is easier to be realized in SDR platforms. Less iterations can linearly reduce the complexity, which can be achieved with the early termination.

The computation is composed by several kernel tasks, and the operations for each step are summarized, listed in Table 13. CC(2, 1, 7), 3GPP-LTE Turbo and $Z = 96$ QC-LDPC codes are shown as examples. It is concluded that for Turbo and CC, arithmetic computation occupies

Table 11 Operations for CC forward path.

Type	Equation	PNum. × OP/Par.
Address comp.	$k^p \leftarrow k + pL$	$P \times 1$
Memory read	$r_i^p \leftarrow \overline{L}(i, k^p), i = 0, \cdot, m-1$	$P \cdot m \times 1$
Memory write	$\overline{TB}_k^p(S) \leftarrow sel^p(S)$	$P \cdot T_s \times 1$
Kernel BMU	$\mathbf{v}(S_i', S) \leftarrow Trellis(S_i', S)$	0
	$\gamma^p(S_i', S) \leftarrow dist(\mathbf{l}^p, \mathbf{v}(S_i', S)), i = 0, 1$	$P \times m \cdot 2^{m-1}$
kernel ACS	$S_i' \leftarrow S/2 + iT_s/2, i = 0, 1$	0
	$b^p(S_i', S) \leftarrow \gamma^p(S_i', S) + \alpha^p(S_i'), i = 0, 1$	$P \cdot 2T_s \times 1$
	$\{\alpha^p(S), sel^p(S)\} \leftarrow f(b^p(S_0', S), b^p(S_1', S))$	$P \cdot T_s \times f$

Note: In CC decoding, the minimum f is 1.

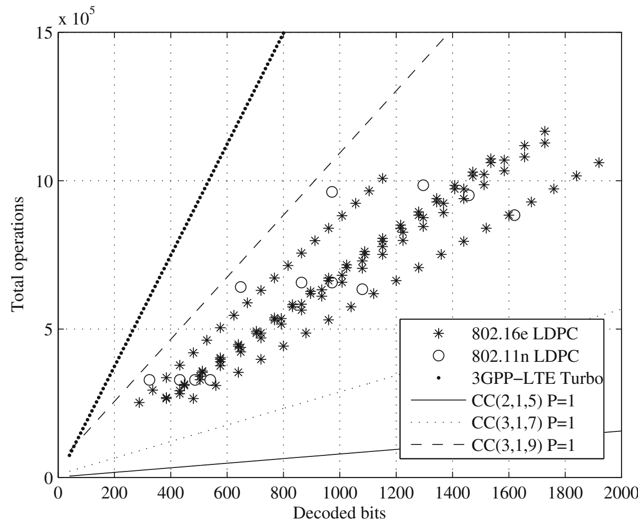


Figure 3 Computational complexity comparison among FEC algorithms with different decoded codeword length.

75 % around of workload, whereas memory access consumes 25 % around. For LDPC, 42 % of workload belongs to memory access, and only 58 % belongs to arithmetic computation. One of the reasons is that LDPC outputs Z a-posteriori messages in a recursion step, whereas Viterbi or (Binary) Turbo only outputs one bit. Considered that LDPC Layered decoding consumes approximately 1/3 of total computation to Turbo BCJR, Layered Decoding requires only 25 % message update computations per bit than that of BCJR (with the same number of iterations). Viterbi algorithm consumes most of the computation on ACS because T_s states need to be processed one by one. For an even larger T_s such as in CC(2, 1, 9), the percentage of computation for ACS is even larger.

It is revealed from Table 13 that several kernel functions consume a large portion of computing resources. The

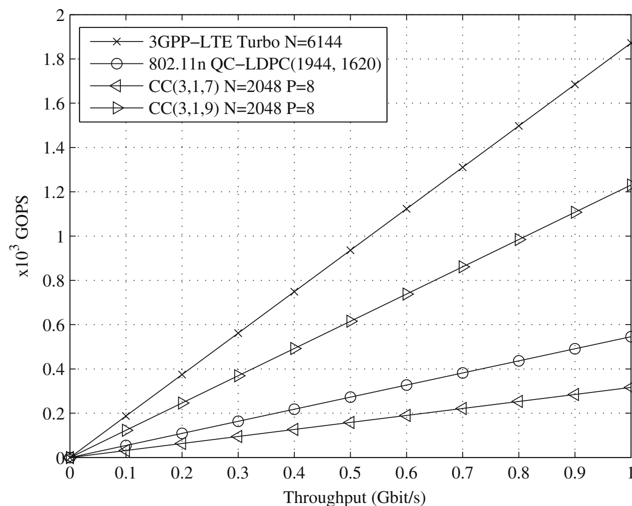


Figure 4 Evaluated total computing cost with respect to throughput.

Table 13 Percentage of component computations.

Kernel	Operations	Percentage
Convolutional code		
Address transform	2	0.7 %
Memory read	3	1.1 %
Memory write	65	24.1 %
Kernel BMU	4	1.5 %
Kernel ACS	192	71.2 %
Kernel TB	3.5	1.3 %
Sum	269.5	100.0 %
Turbo (Half iteration)		
Address LUT	2	1.4 %
Address transform	2	1.4 %
Memory read	14	9.5 %
Memory write	12	8.1 %
Permutation	3	2.0 %
Kernel BM	16	10.8 %
Kernel FR	24	16.2 %
Kernel BR	24	16.2 %
Kernel LLR	51	34.5 %
Sum	148	100.0 %
LDPC (One iteration)		
Address LUT	2	0.1 %
Memory read	384	16.7 %
Memory write	384	16.7 %
Permutation	192	8.3 %
$f(x, y)$ kernel	1152	50.0 %
Arithmetic	192	8.3 %
Sum	2306	100.0 %

operations would be reduced dramatically if the kernels are accelerated by hardware circuits and operation-fusion instructions. Such as the $f(x, y)$ function for LDPC layered decoding, the LLR calculation in Turbo decoding, and the ACS kernel in Viterbi algorithm.

5 Specific Platform Design with the Proposed Evaluations

For highly parallel platforms, the platform related overhead needs to be considered in, which includes the inter-core communication, the core workload balancing, and the synchronization between cores. The decoding algorithms also require for a conflict-free or conflict-minimum memory access. It is advisable to construct a many core platform based on the ASIC FEC implementations. In ASIC (Application Specific Integrated Circuit) or ASIP (Application Specific Instruction-set Processor) implementations of the proposed pseudo-codes [19, 24–26], the inter-core data passing are operation-free, the core tasks are well balanced

and the synchronization between cores are not necessary. The memory banks are small sized on-chip scratch-pad modules, therefore no cache is needed. All the memory access conflicts can be avoided and therefore not considered in. These optimizations for ASIC decoders can be applied for designing an SD FEC platform.

We also constructed a tri-mode unified ASIP decoder [27] following these pseudo-codes. (Additional sliding windows are added for reducing the buffer size). The interconnection network and memory subsystem can be borrowed to construct the fully programmable platform. The difference is that the processor cores are introduced to substitute the arithmetic circuits in the ASIP prototype. Higher flexible inter-core network with more redundancy can be introduced without performance degradation if the network and memory structure in the ASIP prototype are included.

6 Compare the Evaluated Results with General Software Defined Decoding Platforms

Apart from the theoretical complexity results and the supporting hardware platforms which can reach these low bounds, the FEC benchmarks are provided for revealing the attainable complexity in feasible processors. Currently, high throughput decoding mainly relies on the General Purpose GPU (GPGPU) platforms because of its highly parallel architecture. In such platforms, the peak floating point operations (GFLOPS) are accessible by manufactures. An alternative choice is general DSP platforms with Very Long Instruction Word (VLIW) architecture, where several instructions are possibly processed simultaneously, such as Texas Instrument (TI) TMS320Cx series DSPs. The peak MIPS can be derived from the device datasheets. Apart from the peak performance, the decoding throughput, iteration numbers, and code length are available in the reference papers, therefore the operations per bit per iteration can be obtained following $OP/N/it_{max}$. Meanwhile the evaluated results are shown with OP given by Eqs. 1, 2 and 3.

Figure 5 shows the software defined (SD) Turbo in general purpose platforms. The proposed guideline is approximately 312 for all sizes of decoded bits. Most of the reference approaches are in GPU platform. Wolf et al. [28] proposed a Design Space Exploration method for SD Turbo and four platforms are tested. With the codeword size of 5000, the complexity is 10 KOP/bit - 100 KOP/bit around. Other proposals target on $N = 6144$ 3GPP-LTE Turbo, which reveal a complexity of 7 KOP/bit - 45 KOP/bit. Proposal [6] reported that the efficiency can be further improved with multi-codeword parallel decoding, which can fully utilize the GPU resources, and the efficiency improves from 9 KOP/bit to 1.8 KOP/bit. It also introduces highly parallel number to make full use of core resources, and

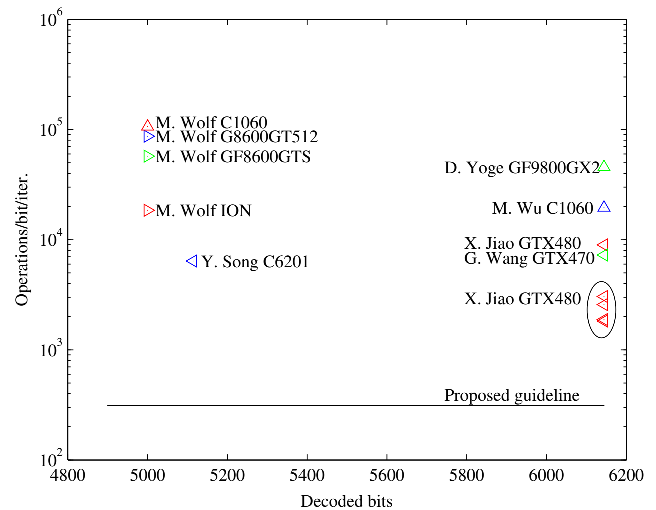


Figure 5 Computational complexity of SD Turbo implementations. The solution references are M. Wolf [28], D. Yoge [29], M. Wu [30], Y. Song [8], X. Jiao [6], and G. Wang [7].

finally 122 Mb/s throughput is derived. Apart from that, a TMS320C6201 DSP approach [8] reveal that approximately 6400 OP/bit is required for HSPA N=5114 code, which is similar to GPU approaches. For this TMS320C6201 DSP platform and following TMS320C64x platform, the peak operations are evaluated by eight times of its peak Million Instruction Per Second (MIPS) because the processor has eight processing units. The actual workload derived from these implementations is higher than the proposed results because of several reasons. (1). An operation defined in Table 3 may be mapped into several processor operations (instructions). (2). In our evaluation, extra tasks such as

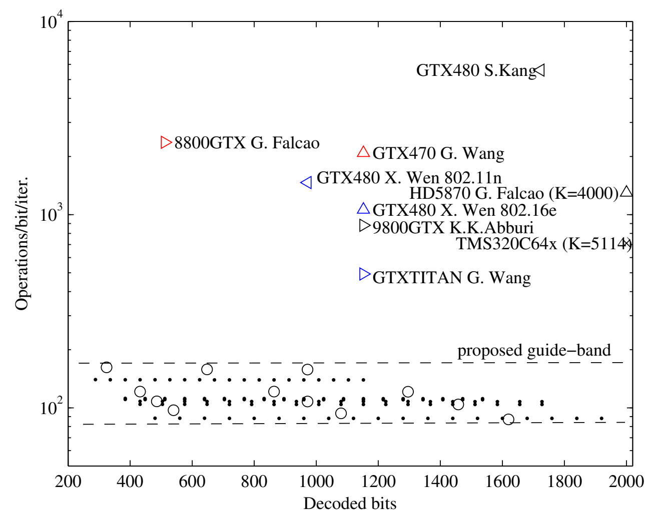


Figure 6 Computational complexity of SD LDPC implementations. The reference implementations are G. Falcao [4], GTX 470 G. Wang [32] [33], K.K.Abburi [2], S. Kang [34], G. Falcao (K=4000) [35], GTXTITAN G. Wang [3], X. Wen [31], and TMS320C64x [5].

Table 14 Viterbi computational complexity benchmarks.

Code type	Platform	Peak performance	Benchmark	Equivalent complexity
CC(2, 1, 5) N=189 for GSM	TMS320C62x DSP [36]	1600 MIPS, 8 PEs.	38.3 Inst/bit	306 OP/bit (8-parallel VLIW)
CC(2, 1, 5) N=189 for GSM	Proposed guideline			87 OP/bit
CC(2, 1, 7)	Cell Processor [37]	102.4 GOPS	30 Mbit/s	3413 Inst/bit
CC(2, 1, 7) N=2048 SPIRAL	Core i7 2600	128,300 MIPS (16037/core), without SSE	4.1 Mbit/s	3887 Inst/bit
CC(2, 1, 7) N=2048 SPIRAL	Core i7 2600	128,300 MIPS (16037/core), with SSE	26.4 Mbit/s	607 Inst/bit (16-parallel SSE)
CC(2, 1, 7) N=2048 SPIRAL	ARM Cortex A7	2850 MIPS/core	786 Kbit/s	3627 Inst/bit
CC(2, 1, 7)	Proposed guideline			1252 OP/bit

data management, memory conflict management, controlling overhead, and thread synchronization are not taken into account. (3). The GPU device peak GFLOPS is assumed as the cost used for the decoding procedure, however making full use of all the computation resources on a chip is unrealistic. Nevertheless, the reference designs reveal the nowadays attainable complexity. It is also hopeful that using an alternative hardware architecture the decoding efficiency can be improved further.

According to the code rates and how sparse the base matrices are, the LDPC decoding complexity varies in different configurations. However they most locate in a 'band', proposed in Fig. 6. It shows the complexity ranging from 88 OP/bit to 162 OP/bit for all the configurations in IEEE 802.11n and IEEE 802.16e. Most of proposed solutions realized a complexity of 500 OP/bit to 2300 OP/bit. Among them, Wen et al. [31] proposed a min-sum layered decoder reaching up to 507 Mbit/s (2 iterations) with early termination, which is the highest throughput among all SD LDPC. It reaches a complexity of 1062 OP/bit. G. Wang et al. proposed a 304 Mbps (10 iterations, 50 codewords in GTX TITAN GPU) solution which reaches the lowest computational complexity (493 OP/bit). K.K.Abburi et al. [2] proposed another high efficient solution which reaches a complexity of 881 OP/bit. The TI TMS320C64x DSP solution [5] shows its similar complexity to GPU solutions.

In case of Viterbi decoding, the benchmarks for DSP, ARM and Intel Processor are summarized. For TMS320C62x series DSP, the number of instructions for GSM CC(2, 1, 5) decoding is given by $(38{\cdot}N+12+N/4)/N$ [36]. For a larger N, approximately 38 instructions per bit are required, which equivalent to 306 operations/bit due to the parallel architecture with eight processing units. The SPIRAL Viterbi decoding code [38] is applied for evaluating the overall decoding complexity in ARM processor (ARM Cortex A7) and Intel processor (Intel Core i7-2600). The code is complied by GNU GCC with '-O3' optimization level. For Core i7 implementation, only one of the 8 cores is utilized. The evaluation results are shown in Table 14, wherein the peak MIPS is derived in [39]. Because 16-way SSE (Streaming SIMD Extensions) vector instructions can perform up to 16 calculations per instruction, therefore the number of instructions is lower than the proposed guideline. It reveals that VLIW DSP is efficient for Viterbi processing. For high throughput decoding in Intel processors, a much higher decoding throughput is achieved by enabling SSE acceleration [9]. The complexity of reference implementations are approximately 2-3 times higher than the proposed complexity guideline.

7 Conclusion

In this work the complexity evaluations for LDPC Layered Decoding, Turbo BCJR decoding and CC Viterbi decoding are provided. Closed expressions for these coding types are offered with variety of configurable parameters. The complexity of these algorithms are compared with the configurations in wireless standards. The reference implementations are compared with the proposed results, which shows that current SDR platforms still have possibilities for achieving higher decoding efficiency. The proposed pseudocodes, parallel schemes and opeartion results may promote the architecture selection and software design for further software defined FEC platforms.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

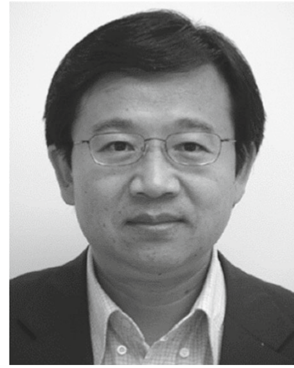
References

1. Van Berkel, C.H. (2009). Multi-core for mobile phones. In *Design, automation test in Europe conference exhibition, 2009. DATE '09* (pp. 1260–1265).

2. Abburi, K.K. (2011). A scalable LDPC decoder on GPU. In *24th international conference on VLSI design (VLSI Design), 2011* (pp. 183–188).
3. Wang, G., Wu, M., Yin, B., & Cavallaro, J.R. (2013). High throughput low latency LDPC decoding on GPU for SDR systems. In *Proceedings of the IEEE global conference on signal and information processing (GlobalSIP)*.
4. Falcao, G., Sousa, L., & Silva, V. (2011). Massively LDPC decoding on multicore architectures. *IEEE Transactions on Parallel and Distributed Systems*, *22*(2), 309–322.
5. Lechner, G.ottfried., Sayir, J., & Rupp, M. (2004). Efficient DSP implementation of an LDPC decoder. In *Proceedings of IEEE international conference on acoustics, speech, and signal processing, 2004. (ICASSP '04)*, (Vol. 4 pp. iv–665–iv–668).
6. Xianjun, J.iao., Canfeng, C.hen., Jaaskelainen, P., Guzma, V., & Berg, H. (2013). A 122mb/s turbo decoder using a mid-range GPU. In *9th international wireless communications and mobile computing conference (IWCMC), 2013* (pp. 1090–1094).
7. Wu, M., Sun, Y., Wang, G., & Cavallaro, J.R. (2011). Implementation of a high throughput 3GPP turbo decoder on GPU. *Journal of Signal Processing System*, *65*(2), 171–183.
8. Song, Y., Liu, G., et al. (2005). *The implementation of turbo decoder on DSP in W-CDMA system*.
9. Tan, K., He, L., Zhang, J., Zhang, Y., Ji, F., & Voelker, G.M. (2011). Sora: high-performance software radio using general-purpose multi-core processors. *Communications of the ACM*, *54*(1), 99–107.
10. Lin, Y.uan., Lee, H.yunseok., Woh, M., Harel, Y., Mahlke, S., Mudge, T., Chakrabarti, C., & SODA, K.Flautner. (2006). A low-power architecture for software radio. In *33rd international symposium on computer architecture, 2006. ISCA '06* (pp. 89–101).
11. Chengzhi, P., Bagherzadeh, N., Kamalizad, A.H., & Koohi, A. (2003). Design and analysis of a programmable single-chip architecture for DVB-T base-band receiver. In *Design, automation and test in Europe conference and exhibition, 2003* (pp. 468–473).
12. Ohkubo, N., Miki, N., Kishiyama, Y., Higuchi, K., & Sawahashi, M. (2006). Performance comparison between turbo code and rate-compatible LDPC code for evolved utra downlink OFDM radio access. In *Military communications conference, 2006. MILCOM 2006* (pp. 1–7): IEEE.
13. Kienle, F., Wehn, N., & Meyr, H. (2011). On complexity, energy- and implementation-efficiency of channel decoders. *IEEE Transactions on Communications*, *59*(12), 3301–3310.
14. Dielissen, J., Engin, N., Sawitzki, S., & van Berkel, K. (2008). Multistandard FEC decoders for wireless devices. *IEEE Transactions on Circuits and Systems* 284–288.
15. Mansour, M.M. (2006). A turbo-decoding message-passing algorithm for sparse parity-check matrix codes. *IEEE Transactions on Signal Processing*, *54*(11), 4376–4392.
16. Hu, X.-Y., Eleftheriou, E., Arnold, D.-M., & Dholakia, A. (2001). Efficient implementations of the sum-product algorithm for decoding LDPC codes. In *Global telecommunications conference, 2001. GLOBECOM '01*, (Vol. 2 pp. 1036–1036E): IEEE.
17. Wang, Z., & Cui, Z. (2007). Low-complexity high-speed decoder design for quasi-cyclic LDPC codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, *15*(1), 104–114.
18. Takeshita, O.Y. (2006). On maximum contention-free interleavers and permutation polynomials over integer rings. *IEEE Transactions on Information Theory*, *52*(3), 1249–1253.
19. Sun, Y., & Cavallaro, J.R. (2011). Efficient hardware implementation of a highly-parallel 3GPP LTE/LTE-advance turbo decoder. *Integration, the VLSI Journal*, *44*(4), 305–315.
20. Sani, A.H., Coussy, P., & Chavet, C. (2013). A first step toward on-chip memory mapping for parallel turbo and LDPC decoders: a polynomial time mapping algorithm. *IEEE Transactions on Signal Processing*, *61*(16), 4127–4140.
21. Tarable, A., Benedetto, S., & Montorsi, G. (2004). Mapping interleaving laws to parallel turbo and LDPC decoder architectures. *IEEE Transactions on Information Theory*, *50*(9), 2002–2009.
22. Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M.P.C., & Hu, X.-Y. (2005). Reduced-complexity decoding of LDPC codes. *IEEE Transactions on Communications*, *53*(8), 1288–1299.
23. Lin, S., & Costello, D.J. (2004). *Error Control Coding* Vol. 123. Englewood Cliffs: Prentice-hall.
24. Sun, Y., & Cavallaro, J.R. (2008). A low-power 1-Gbps reconfigurable LDPC decoder design for multiple 4G wireless standards. In *IEEE international SOC conference, 2008* (pp. 367–370).
25. Cavallaro, J.R., & Vaya, M. (2003). *Viturbo: a reconfigurable architecture for Viterbi and turbo decoding*.
26. Gentile, G., Rovini, M., & Fanucci, L. (2010). A multi-standard flexible turbo/LDPC decoder via ASIC design. In *6th international symposium on turbo codes and iterative information processing (ISTC), 2010* (pp. 294–298).
27. Zhenzhi, W., & Liu, D. (June 2014). Flexible multistandard FEC processor design with ASIP methodology. In *IEEE 25th international conference on application-specific systems, architectures and processors (ASAP), 2014* (pp. 210–218).
28. Lee, D., Wolf, M., & Kim, H. (2010). Design space exploration of the turbo decoding algorithm on GPUs. In *Proceedings of the 2010 international conference on compilers, architectures and synthesis for embedded systems, CASES '10* (pp. 217–226). New York: ACM.
29. Reddy, D., YOge, N., & Chandrachoodan, N. (2012). GPU implementation of a programmable turbo decoder for software defined radio applications. In *VLSI Design'12* (pp. 149–154).
30. Wu, M., Sun, Y., & Cavallaro, J.R. (2010). Implementation of a 3GPP LTE turbo decoder accelerator on GPU (pp. 192–197).
31. Wen, X., Xianjun, J., Jaaskelainen, P., Kultala, H., Canfeng, C., Berg, H., & Zhisong, B. (2014). A high throughput LDPC decoder using a mid-range GPU. In *IEEE international conference on acoustics, speech and signal processing (ICASSP), 2014* (pp. 7515–7519).
32. Wang, G., Wu, M., Sun, Y., & Cavallaro, J.R. (2011). A massively parallel implementation of QC-LDPC decoder on GPU. In *SASP'11* (pp. 82–85).
33. Wang, G., Wu, M., Sun, Y., & Cavallaro, J.R. (2011). GPU accelerated scalable parallel decoding of LDPC codes. In *Conference record of the 45th Asilomar conference on signals, systems and computers (ASILOMAR), 2011* (pp. 2053–2057).
34. Kang, S., & Moon, J. (2012). Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing. In *IEEE international conference on communications (ICC), 2012* (pp. 3692–3697).
35. Falcao, G., Silva, V., Sousa, L., & Andrade, J. (2012). Portable LDPC decoding on multicores using OpenCL [applications corner]. *IEEE Signal Processing Magazine*, *29*(4), 81–109.
36. TEXAS INSTRUMENTS. C6000 Benchmarks., http://www.ti.com/sc/docs/products/dsp/c6000/62bench.htm. Accessed: 2016-6-10.
37. Lai, J., & Chen, J. (2008). High performance viterbi decoder on cell BE. In *Proceedings of the 1st international workshop on software radio technology (SRT2008)*.
38. SPIRAL. Viterbi decoder software generator., http://www.spiral.net/software/viterbi.html. Accessed: 2016-6-10.
39. Wikipedia. Instructions per second., http://en.wikipedia.org/wiki/Instructions_per_second. Accessed: 2016-6-10.

Zhenzhi Wu received his B.S. degree in electronic science and technology from Beijing Institute of Technology (BIT) in 2008 and his Ph.D. degree in information and communication engineering from BIT in 2015. He was a visiting researcher with the division of Computer Engineering from Sep. 2012 to Sep. 2014, Linköping University, Linköping, Sweden. Currently, he is an assistant researcher in Center for Brain-Inspired Computing Research, Tsinghua University, Beijing, China. Zhenzhi's research interests are high performance hardware for signal processing algorithms, artificial intelligence hardware, and wireless baseband accelerator designs.

Chen Gong received his B.S. degree in electrical engineering from the Dalian University of Technology, China, in 2012. He is currently pursuing his Ph.D. degree in electrical engineering at the Beijing Institute of Technology, China. His research interests include inductive near-field communication, high-performance application specific instruction set processors (ASIP), and low-power mixed-signal circuit design for advanced medical devices.



Dake Liu is professor and the director of ASIP Lab, Beijing Institute of Technology, China, and also professor of Computer Engineering Division at the Department of Electrical Engineering of Linköping University, Sweden. He got technology doctor degree from Linköping University Sweden in 1995. Dake published more than 150 papers on journals and international conferences. Dake's research interests are high-performance low-power ASIP (application specific instruction set processors), integration of onchip multi-processors for communications and media digital signal processing. Dake has experiences also in design of communication systems and Radio frequency CMOS integrated circuits. Dake Liu is the cofounder and CTO of FreehandDSP AB, Stockholm Sweden, and the co-founder of Coresonic AB, Linköping, Sweden. Coresonic was acquired by MediaTek. Currently he is enrolled in the China Recruitment Program of Global Expert (Thousand Talents Plan Project) since 2010.