IEIE Transactions on Smart Processing and Computing, vol. 6, no. 3, June 2017
https://doi.org/10.5573/IEIESPC.2017.6.3.210

Efficient Parallel Block-layered Nonbinary Quasi-cyclic Low-density Parity-check Decoding on a GPU

Huyen Pham Thi and Hanho Lee

Department of Information and Communication Engineering, Inha University, 100 Inha-ro, Nam-gu, Incheon, 22212, Korea
hhlee@inha.ac.kr

* Corresponding Author: Hanho Lee

Received April 6, 2017; Accepted May 10, 2017; Published June 30, 2017

* *Regular Paper*

***Abstract*:** This paper proposes a modified min-max algorithm (MMMA) for nonbinary quasi-cyclic low-density parity-check (NB-QC-LDPC) codes and an efficient parallel block-layered decoder architecture corresponding to the algorithm on a graphics processing unit (GPU) platform. The algorithm removes multiplications over the Galois field (GF) in the merger step to reduce decoding latency without any performance loss. The decoding implementation on a GPU for NB-QC-LDPC codes achieves improvements in both flexibility and scalability. To perform the decoding on the GPU, data and memory structures suitable for parallel computing are designed. The implementation results for NB-QC-LDPC codes over GF(32) and GF(64) demonstrate that the parallel block-layered decoding on a GPU accelerates the decoding process to provide a faster decoding runtime, and obtains a higher coding gain under a low 10^{-10} bit error rate and low 10^{-7} frame error rate, compared to existing methods.

***Keywords*:** NB-LDPC, Iterative decoding, GPU, Parallel computation, CUDA

1. Introduction

A binary low-density parity-check (LDPC) code, which provides performance close to that of the Shannon limit for long code lengths, was investigated by Gallager [1]. It was shown that nonbinary LDPC (NB-LDPC) codes outperform binary LDPC codes when the code lengths are short and moderate [2]. NB-LDPC codes [3-5] have attracted a tremendous amount of research interest because of their excellent error-correcting capabilities. However, the decoding algorithms for NB-LDPC codes require complex computations and large memories [5].

A belief propagation (BP) algorithm using fast Fourier transform (FFT) in the probability domain can be used for NB-LDPC codes to reduce the computational complexity from $O(q^{d_c})$ to $O(q\log_2 q)$ [6] . Nonetheless, numerous additions and multiplications lead to an increase in hardware complexity in probability domain algorithms. To address this problem, the extended min-sum (EMS) [7] and min-max algorithms [4] use log-likelihood ratio (LLR) values to decode channel messages; accordingly, multiplications are replaced by additions in the logarithm domain. Both algorithms have been attractive for very-large-scale integration (VLSI) implementations. Savin [4] applied a max operation instead of a sum operation for check node processing. This provides advantages for VLSI implementation because a max operation can easily be performed by using a comparator. Thus, the min-max algorithm has been widely used in NB-LDPC decoder architectures [8-10].

Recently, graphics processing units (GPUs) have been used for the high computational power by which they simultaneously execute numerous threads; furthermore, their peak performance can reach speeds up to tera floating–point operations per second. NVIDIA Corp. presented the Compute Unified Device Architecture (CUDA) [11] using the C high-level programming language, which offers a software environment to facilitate the development of high-performance applications. There has been growing interest in GPU-based implementations for binary LDPC decoding [12-16], since the GPU can push higher simulation speeds.

Because of the high complexity of NB-LDPC decoding algorithms, the simulation time on a central processing unit (CPU) is extremely slow in the higher-order Galois field GF(q). Therefore, it is impossible to show the error floor property of NB-LDPC codes using CPU-based simulations

at a low bit error rate (BER) and frame error rate (FER). The GPU can provide massively parallel computation threads with many-core architectures, which is suitable to accelerate simulations of NB-LDPC decoding. Currently, implementation of NB-LDPC decoders on a GPU device is actively researched. GPU-related works have focused on mapping the decoding algorithms on GPU devices using different optimization schemes, such as data-structure and memory-coalescing techniques. GPU-based decoding implementations [17-20] could accelerate the decoding process and obtain high throughput. In addition, a flood decoding scheme including two-phase check-node processing (CNP) and variable-node processing (VNP) has been applied. However, these works have not shown error correcting performance, such as BER and FER.

Recent work by Beermann et al. [20] applied a layered scheme for a belief propagation decoding algorithm on a GPU. Their results illustrated BER performance at around 10^{-5}, which is similar to C++ simulation results for VLSI-based decoders [8, 10]. A layered decoding scheme is well known for reducing the memory requirement and increasing the convergence speed of NB-LDPC decoding. Our work demonstrates that a horizontal-layered decoding scheme can be efficiently implemented with a min-max algorithm on a GPU.

In this paper, a modified min-max algorithm (MMMA) and a parallel block-layered nonbinary quasi-cyclic low-density parity-check (NB-QC-LDPC) decoding architecture corresponding to the algorithm on a GPU are proposed. The MMMA removes the multiplications over the Galois field in the merger step to reduce the number of computations, as well as the decoding latency, without any performance loss. The parallel block-layered NB-QC-LDPC decoding architecture is efficiently implemented on a GPU to accelerate the decoding process. The results show that the GPU-based parallel block-layered decoder using an MMMA achieves a much faster decoding runtime, compared to a single-core CPU implementation, and shows an error floor property for FERs of approximately 10^{-7} and a low 10^{-10} BER.

The remainder of this paper is organized as follows. In Section 2, NB-LDPC codes and decoding algorithms are briefly reviewed, and the modified min-max algorithm for the GPU-based implementation is provided. In Section 3, the parallel block-layered decoding architecture on a GPU using CUDA is described. Experimental results are presented in Section 4. Finally, conclusions are given in Section 5.

2. NB-LDPC Codes and Decoding Algorithms

2.1 NB-LDPC Codes

(N, K) NB-LDPC codes (N code symbols, K information symbols, and $M = N - K$ parity symbols) are defined as parity-check matrix **H** including a small proportion of Galois-field elements. The NB-LDPC codes are illustrated by a Tanner graph. Each row of the **H** matrix is connected with a check node, and each column of the **H** matrix connects with a variable node on the Tanner graph.

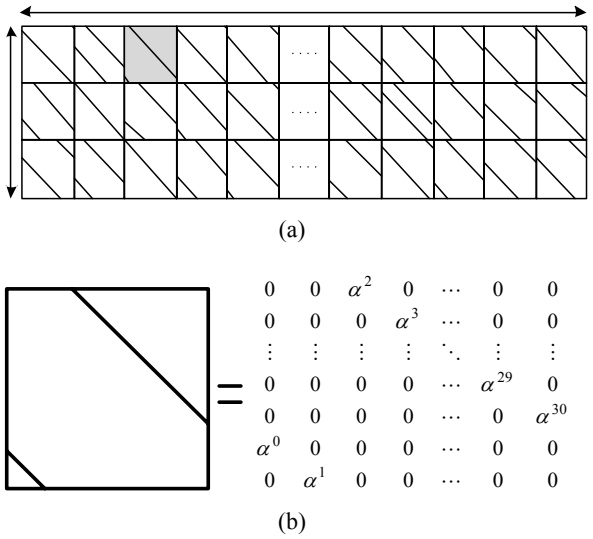


Fig. 1. (a) H matrix construction of (744, 653) NB-QC-LDPC code over GF(32), (b) an example of an α-multiplied CPM for α^2.

Zhou et al. [3] presented two new algebraic constructions for NB-LDPC codes based on array dispersions of matrices. In their work [3], the parity check matrices of NB-QC-LDPC codes are row-column (RC) constrained arrays, which are extended as circulant permutation matrices (CPMs) over nonbinary Galois fields. The structural property of an RC constraint is a constraint on the rows and columns of the **H** matrix. Using this method, a (q-1) × (q-1) **H** submatrix is generated over GF(q). Let d_v and d_c be the column weight or variable node degree and the row weight or check node degree, respectively. Then, a submatrix with a size of (d_v, d_c) ($1 < d_v < (q\text{-}1)$ and $1 < d_c < (q\text{-}1)$) is selected from the **H** submatrix. Each element in the submatrix (d_v, d_c) is dispersed as either an all-zero matrix of size (q-1) × (q-1) or an α-multiplied CPM of size (q-1) × (q-1). As a result, the **H** matrix is generated with a size of (d_v × (q-1), d_c × (q-1)). Fig. 1(a) shows a (744, 653) NB-QC-LDPC code over GF(32), which is constructed by submatrix (3, 24) with $d_v = 3$ and $d_c = 24$. In α-multiplied CPMs, there is only one i-th nonzero entry in the first row of the matrix. It is generated by dispersing element α^i, and other entries are zero. Each of the other rows is a right cyclic-shift of the previous row multiplied by α. Fig. 1(b) shows the α-multiplied CPM for α^2.

2.2 Parallel Block-layered Min-max Algorithm

In this section, horizontal layered decoding is applied to decrease both the memory requirement and decoding iterations. The **H** matrix is divided into layers. Then, decoder iterations are sequentially performed with each layer's iteration. In this work, a parallel block-layered min-max decoding algorithm for implementation on a GPU is presented, as shown in *Algorithm* 1. It simultaneously

Algorithm 1: GPU-based parallel block-layered min-max decoding algorithm

Initialization: $L_n(a) = \ln(\Pr(x_n=s_n|\text{channel})/\Pr(x_n=a|\text{channel}))$;
$L_n^{1,0}(a) = L_n(a)$; $R_{mn}^{0}(a)=0$;
1: **for**(k = 1; $k \le I_{max}$; k++) **do**
2: **for**(l = 1; $l \le L$; l++) **do**
3: **for**(m = 0; $m < q$-1; m++) **do**
4: $\tilde{L}_{nm}^{k,l}(a) = L_n^{k,l-1}(a) - R_{mn}^{k-1,l}(a)$
5: $\tilde{L}_{nm}^{k,l} = \min_{a \in GF(q)} (\tilde{L}_{nm}^{k,l}(a))$
6: $L_{nm}^{k,l}(a) = \tilde{L}_{nm}^{k,l}(a) - \tilde{L}_{nm}^{k,l}$
7: $R_{mn}^{k,l}(a) = \min_{(a_{n'})_{n' \in N(m)} \in \gamma_{mn}(a)} (\max_{n' \in N(m) \setminus \{n\}} (L_{n'm}^{k,l}(a_{n'})))$

$$\gamma_{mn}(a) = \{a_{n'} \mid h_{mn} a + \sum_{n' \in N(m) \setminus \{n\}} h_{mn'} a_{n'}\}$$

8: $L_n^{k,l}(a) = \tilde{L}_{nm}^{k,l}(a) + R_{mn}^{k,l}(a)$
9: **end for**
10: **end for**
11: **end for**
12: **Decision**: $\tilde{c}_n = \arg\min(L_n^{k,l}(a))$

processes (q-1) check nodes in one block layer.

One block layer is constructed by non-overlapped (q-1) rows; each column of these block layers has a weight value of one.

Algorithm 1 is briefly summarized as follows. The layered decoding divides the **H** matrix rows into $L = d_v$ layers. The decoding processing for layer 1, layer 2,... , and layer d_v are sequentially performed to complete a single iteration; the extrinsic values are exchanged among the layers. This process is consecutively performed until the number of iterations reaches maximum value I_{max} or until the parity check is satisfied. The initialization of the algorithm is similar, as shown in *Algorithm* 1. In addition, the variable node (V2C) messages, $\tilde{L}_{nm}^{k,l}(a)$, of layer l in iteration k are computed based on $L_n^{k,l-1}(a)$ and $R_{mn}^{k-1,l}(a)$, which are the a posteriori messages of layer l-1 in iteration k and the check node (C2V) messages of layer l in iteration k-1, respectively. $\tilde{L}_{nm}^{k,l}(a)$ is expressed in Eq. (1) as follows:

$$\tilde{L}_{nm}^{k,l}(a) = L_n^{k,l-1}(a) - R_{mn}^{k-1,l}(a) \tag{1}$$

In the first layer of the first iteration, the V2C messages $L_n^{1,0}(a)$ are the reliability information from channel $L_n(a)$, and the check node memory values (CMEM) $R_{mn}^{0}(a)$ for all layers of the first iteration are equal to zero.

Let x_n be the n-th symbol in a received code word, and let s_n be the most likely symbol of x_n. The $L_n(a)$ vectors have q elements, including one zero element and (q-1) positive elements. The min-max decoding, which is implemented by the forward-backward algorithm (FBA) [4], is applied in the check node process.

2.3 Modified Min-max Algorithm

Assume that the finite-field elements of power representation are expressed as order {0, α^0, α^1,… , α^{q-2}}, where α is a primitive element of GF(q). The log-likelihood ratios are stored as a vector and are indexed in ascending order of α. Let a', a'', and a elements be Galois-field elements, which are used as indexes of the message vectors. Let m be a check node, and let $N(m) = \{n_0, n_1,\dots, n_{d_c-1}\}$ be the set of d_c variable nodes connected to check node m in the Tanner graph. The FBA [4] includes three steps to find the C2V messages: forward (FD), backward (BD), and merger (MG). For the FD and BD steps, forward metrics $(F_i)_{i=0,d_c-2}$ and backward metrics $(B_i)_{i=1,d_c-1}$ are sequentially calculated with a conditional equation, as seen in Eq. (2):

Conditional equation for forward step and backward step:

$$a' + \alpha^{h_i} a'' = a \tag{2}$$

where h_i is the nonzero element in the **H** matrix.

Forward metrics:

First step:

$$F_0(a) = L_{0,c}(\alpha^{h_0^{-1}}(a)) \tag{3}$$

Recursive step: $1 \le i \le d_c$-2

$$F_i(a) = \min_{\substack{a',a'' \in GF(q) \\ a'+\alpha^{h_i}a''=a}} (\max(F_{i-1}(a'), L_{i,c}(a''))) \tag{4}$$

Backward metrics:

First step:

$$B_{d_c-1}(a) = L_{d_c-1,c}(\alpha^{h_{d_c-1}^{-1}}(a)) \tag{5}$$

Recursive step: d_c-2 $\ge i \ge 1$

$$B_i(a) = \min_{\substack{a',a'' \in GF(q) \\ a'+\alpha^{h_i}a''=a}} (\max(B_{i+1}(a'), L_{i,c}(a''))) \tag{6}$$

In the first step, the FD messages are generated by multiplying $L_{0,c}(a)$ with the inversion of the first nonzero entry of the m row, $\alpha^{h_0^{-1}}$, as shown in Eq. (3). Let a be an index of $F_i(a)$, and let a' and a'' be indexes of $F_{i-1}(a')$ and $L_{i,c}(a'')$, respectively. Then, the current forward vector $F_i(a)$ of the recursive step is calculated in Eq. (4) from the previous FD messages, $F_{i-1}(a')$, and the current V2C messages, $L_{i,c}(a'')$, using the conditional Eq. (2). Eq. (2) shows multiplication indexes a'' of the i-th current V2C vector with the nonzero elements of the **H** matrix, α^{h_i}. This ensures the FD messages are generated completely by the V2C vector multiplied with the nonzero elements of the **H** matrix. The computation of the BD messages is similar to the FD messages, as shown in Eqs. (5) and (6). In the FBA [4], merger messages $M_i(a)$ were constructed from the forward $F_{i-1}(a)$ and backward $B_{i+1}(a)$ messages following the condition equation, as seen in Eq. (7), where indexes a of merger vector $M_i(a)$ are calculated with one

Table 1. Decoding computational complexity of algorithms per iteration.

Algorithm		Addition	Multiplication	Comparison	Table lookup
SPA	VNP	$Md_c(q\text{-}1)$	$Md_v d_c q$	**0**	**0**
	CNP	$M(3d_c - 4)q(q\text{-}1)$	$M(3d_c - 4)q^2$	0	0
FFT-BP	VNP	0	$Nd_v^2 q$	0	0
	CNP	$(2qd_c + q/\log_2 q)M$	$q/\log_2 q(d_c^2 + 4d_c)M$	0	0
Log-FFT-BP	VNP	$Nd_v^2 q$	0	0	0
	CNP	$(2q + 4q/\log_2 q)d_c M$	0	0	$2Md_c q$
Proposed MMMA	VNP	$2Nd_v q$	0	0	0
	CNP	0	0	$3M(d_c - 2)q(2q - 1)$	0

addition operation and one division operation:

$$a' + a'' = -h_i a \tag{7}$$

Wang et al. [18] applied the min-max forward–backward algorithm [4] for a GPU implementation. In their work [18], the forward and backward vectors are repeatedly multiplied with the nonzero elements in the **H** matrix before computing the merger messages. For example, merger vector $M_i(a)$ is generated from the $F_{i-1}(a')$ and $B_{i+1}(a'')$ vectors. First, the $F_{i-1}(a')$ and the $B_{i+1}(a'')$ vectors are multiplied with h_{i-1} and h_{i+1} to obtain $F_{i-1}(h_{i-1}a')$ and $B_{i+1}(h_{i+1}a'')$, respectively. Then, merger messages $M_i(a)$ are constructed by using the conditional equation $a' + a'' = a$. We can see that two multiplication operations and one addition operation are needed to generate the merger messages [18].

In this paper, we keep the computation of the forward and backward messages similar to those by Savin [4], and the modified min-max algorithm is performed in the merger step to reduce the number of operations. As discussed above, the merger messages are constructed after finishing the forward and backward computations. Two merger vectors, $M_0(a)$ and $M_{d_c-1}(a)$, are directly derived from backward vector $B_1(a)$ and forward vector $F_{d_c-2}(a)$, respectively. For other merger vectors, we propose to remove multiplication with nonzero elements of the **H** matrix in Eq. (7) of the merger step [4], and directly use the forward and backward vectors for computing the merger vectors. This eliminates repeated multiplication [4, 18], because the forward and backward vectors are completely multiplied with the nonzero elements of the **H** matrix in the forward and backward steps. Therefore, the conditional equation for the merger step in the MMMA is shown as Eq. (8), and the computation of merger vectors in the MMMA is depicted in Eqs. (9) and (10). From Eq. (8), only one addition operation is needed to calculate merger messages $M_i(a)$, which is decreased in comparison with other methods [4, 18]. This allows reducing the calculation time in check node processing as well as delay in the decoding process. Furthermore, the MMMA does not affect the error correcting performance because multiplications with the nonzero elements of the **H** matrix are completely resolved in the forward and backward processing.

Proposed conditional equation for merger step:

$$a' + a'' = a \tag{8}$$

Proposed merger step:

$$M_{c,0}(a) = B_1(a)\,;\; M_{c,d_c-1}(a) = F_{d_c-2}(a) \tag{9}$$

$$M_{c,k}(a) = \min_{\substack{a',a''\in GF(q)\\ a'+a''=a}} (\max(F_{k-1}(a'), B_{k+1}(a''))) \tag{10}$$

3. Parallel Block-layered NB-QC-LDPC decoding on a GPU using CUDA

3.1 Complexity Analysis of Nonbinary LDPC Decoding Algorithms

Table 1 approximately summarizes the decoding computational complexity of NB-LDPC decoding algorithms. The total number of floating point operations per iteration is provided for the VNP and the CNP separately. We can see that the FFT algorithm can significantly reduce computational complexity in both the VNP and the CNP, compared to the BP algorithm [6]. The algorithms in the logarithm domain as log-FFT-BP [21, 22] and the proposed MMMA, which replace the multiplication with the addition, further decrease the decoding complexity. Table 1 shows that most of the computation in the algorithms is spent in the CNP. Therefore, to accelerate the decoding, the CNP needs to speed up.

NVIDIA GPUs are powerful arithmetic engines that can run thousands of lightweight threads in parallel. A GPU-based heterogeneous platform has one or more CPUs and GPUs that are well-suited to implementing NB-LDPC decoding algorithms. In addition, the NB-LDPC decoding algorithms have a high computation to memory access ratio (CMAR). The CMAR represents the complexity of operations that justify the cost of moving data to and from the device. To obtain a modern processor architecture integrated in GPUs, the application must first be assessed to identify the hotspots that can be parallelized. The runtime of main blocks in the min-max algorithm is measured by running a serial C code on a single-core CPU. A (744, 653) NB-LDPC code over GF(32) is used for the C simulation, and the execution time is achieved using the

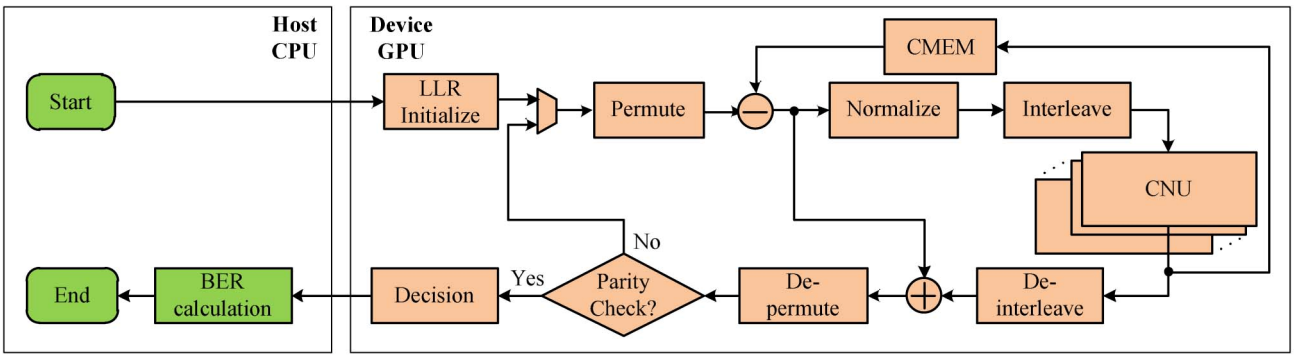


Fig. 2. Data flow of parallel block-layered NB-QC-LDPC decoding on CPU and GPU platforms.

CPU timer. It has been shown that check node processing is a bottleneck in the decoding algorithm and accounts for 95.2% of the processing time. Hence, the computations of the CNP can be parallelized on the GPU platform to accelerate the CNP.

3.2 Data Flow of NB-QC-LDPC Decoding

This section presents an efficient parallel block-layered decoder architecture for NB-QC-LDPC codes based on the GPU platform to accelerate the decoding process. Fig. 2 shows data flow for the parallel block-layered decoder based on both the host CPU and device GPU platforms. The CUDA program is separated into a host CPU and a device GPU. The host CPU handles the kernel scheduling, control of the decoding iterations, and computing of BER performance. In addition, the host CPU must transfer the symbols of the received code word to the GPU; it then receives the decoded symbols from the GPU. Most of the decoding computations are implemented on the GPU. All intermediate messages are stored in the device memory to restrict data transfer between host and device. Each of the modules in Fig. 2 corresponds to a kernel mapped to the GPU platform.

At the beginning of the decoding process, the $L_n(a)$ channel LLRs are directed to check nodes according to the locations of the nonzero entries in the **H** matrix by a permutation block. After subtracting, the variable node messages are used as input messages for the normalization block. Normalization is necessary to ensure numerical stability and to ensure that the smallest LLR in a vector is always zero. The normalized output messages for consecutive check nodes are not aligned in GPU memory. Therefore, interleaving must be performed before the check-node processing to align the V2C messages for each check node. Similarly, de-interleaving is performed after check node processing is finished.

3.3 Data and Memory Structure

As mentioned in Section 2.1, the **H** matrix is constituted from the (d_v, d_c) submatrix, where the elements of the submatrix are extended by α-multiplied CPM (q-1) × (q-1). To take advantage of α-multiplied CPM, a submatrix is stored in memory instead of the full **H** matrix. This method is called the compress technique, which

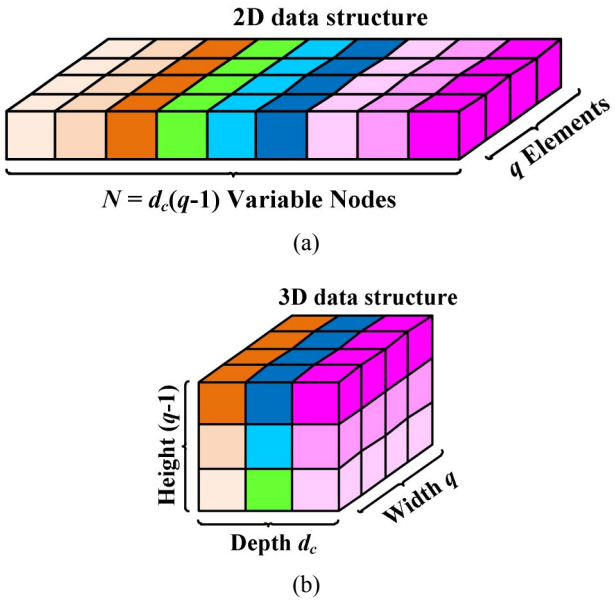


Fig. 3. Data structure for coalescing memory access in variable node processing (a) 2D data structure of LLRs and V2C messages, (b) 3D data structure of LLRs and V2C messages.

reduces the storage memory for the **H** matrix and enables faster memory access.

In this work, we propose a layered decoding scheme on the GPU platform in which (q-1) check nodes in one block row are simultaneously processed. Therefore, in this section, we describe the data and memory structures for single-layer processing. Fig. 3(a) shows the two-dimensional (2D) data structure of [q, (q-1) × d_c] for the LLR messages, as well as V2C messages $L_{nm}(a)$ corresponding to q entries in one vector and (q-1) × d_c vectors for variable node processing. A total of (q-1) × d_c V2C vectors are distributed for (q-1) check nodes; therefore, each check node has d_c connected LLR vectors. To process (q-1) check nodes in parallel, the 2D data structure is re-ordered to generate a three-dimensional (3D) data structure with a size of [q, q-1, d_c] for the LLR vectors, as shown in Fig. 3(b). The three dimensions of the LLR message vectors are expressed as follows: the width, q, corresponds to q entries in one vector; height q-1 corresponds to q-1 LLR message vectors in one block

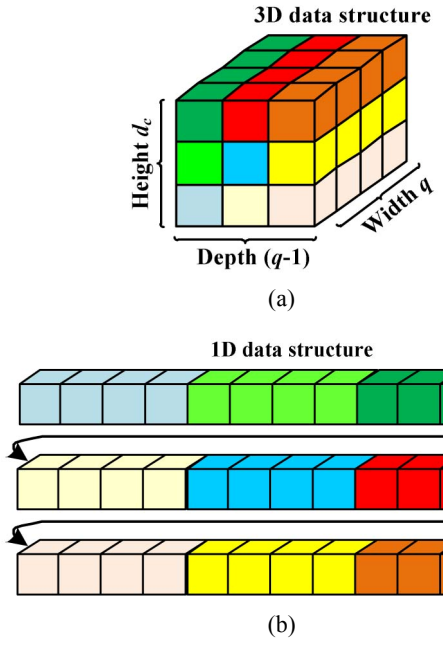


Fig. 4. (a) Data structure for coalescing memory access in check node processing (a) 3D data structure of C2V messages, (b) 1D data structure of C2V messages.

column; and depth d_c corresponds to d_c block columns. Because $(q\text{-}1) \times d_c \times q$ LLR messages are required to process one layer, the size of temporary memory for variable node processing is only $(q\text{-}1) \times d_c \times q$. This reduces the memory requirement for VNP by a factor of the number of layers, d_v, compared to the flooding scheme.

Fig. 4(a) depicts 3D structure [q, d_c, q-1] for the C2V vectors, $R_{mn}(a)$. The three dimensions of the C2V vectors are as follows: width q corresponds to q entries in one vector; height d_c corresponds to d_c connected V2C message vectors; and depth (q-1) corresponds to (q-1) check nodes. This data structure allows (q-1) check nodes to operate in parallel, and lets each check node access d_c V2C message vectors in alignment. If a 3D data structure is formatted by [*width*, *height*, *depth*], each element [x, y, z] of the data structure is uniquely indexed by [$x + y \times width + z \times width \times height$] in the one-dimensional (1D) data structure, as shown in Fig. 4(b). By arranging $L_{nm}(a)$ V2C message vectors and $R_{mn}(a)$ C2V message vectors in this format, the q adjacent data entries are accessed by q adjacent threads; thus, coalesced memory access is enabled, which achieves high throughput.

The texture memory of a GPU is employed to implement the nonbinary arithmetic in GF(q), which is available to all kernels. Additions and subtractions in GF(q) are implemented as exclusive OR (XOR) operations, and divisions by α^a are computed by multiplication with $\alpha^{(31-a)\%31}$. Therefore, two 2D lookup tables of size $q \times q$ exist for multiplication and addition; two 1D lookup tables of size q exist for conversion between exponential and decimal representation. A 64 KB constant memory is used to store values from the parity-check matrix, which are actually the values and indices of bit nodes connected to each check node.

Table 2. Kernel architecture for main blocks of the decoder over GF(32).

Function		No. of blocks	No. of threads	Total No. of threads
Initial LLR		d_c	$q\times(q\text{-}1)$	$d_c\times q\times(q\text{-}1)$
CNP	FD-BD	q-1	$q+q$	$(q\text{-}1)\times(q+q)$
	MG	q-1	$q\times d_c$	$(q\text{-}1)\times q\times d_c$
VNP		d_c	$q\times(q\text{-}1)$	$d_c\times q\times(q\text{-}1)$
Decision		1	$d_c\times(q\text{-}1)$	$d_c\times(q\text{-}1)$

3.4 Parallel Forward-backward Scheme in Check Node Processing

The proposed decoding algorithm is partitioned into four kernels corresponding to four main functions, and the kernel sizes are listed in Table 2. Note that the configuration parameters of kernels are flexibly changed depending on the parameters of each GPU card used. The initial LLR kernel, which has d_c thread blocks and $q \times (q\text{-}1)$ threads per thread block corresponding to d_c block columns and $q\times(q\text{-}1)$ messages in each block column, is responsible for conversion of the channel messages from the probability domain to the logarithm domain. For each check node, the FD and BD messages are first simultaneously calculated using q threads for the forward step and q threads for the backward step. Once the forward–backward messages are available, the d_c MG vectors will be computed in a parallel way by $q \times d_c$ threads. Because (q-1) check nodes are processed in parallel, (q-1) thread blocks are required for the FD, BD, and MG steps. The VNP kernel architecture is similar to the initial LLR kernel to perform addition, subtraction, and normalization. Finally, the decision kernel is used to find the most reliable symbols of $d_c \times (q\text{-}1)$ a posteriori vectors.

Fig. 5 shows the detailed kernel architecture for one check node. The input messages for the CNP kernel are stored in VNP temporary memory. The FD and BD messages are simultaneously computed using q threads for the forward process and q threads for the backward process in the CNP kernel; these messages are kept in the forward and backward memories to continue computing the merger messages. On the other hand, the FD and BD messages are also stored in on-chip local memory to compute the next forward and backward messages. Assuming that the $F_i(a)$ and $B_j(a)$ message vectors are concurrent forward and backward vectors, these vectors are immediately used to compute $F_{i+1}(a)$ and $B_{j-1}(a)$ vectors in the next step. So, these messages need to be stored in memory that can be accessed rapidly. Therefore, on-chip local memory with high bandwidth and low latency is appropriate for storing message vectors $F_i(a)$ and $B_j(a)$ and to speed up the forward and backward steps. The memory requirement for $F_i(a)$ and $B_j(a)$ in (q-1) check nodes is $2 \times q \times$ sizeof(float) $\times$ (q-1). For example, 7.75 KB of local memory is required for 31 check nodes in GF(32). A barrier synchronization function, __syncthreads(), is performed after each forward $F_i(a)$ or backward step $B_j(a)$ to ensure that threads are synchronized.

Let the computation step of forward vector (backward

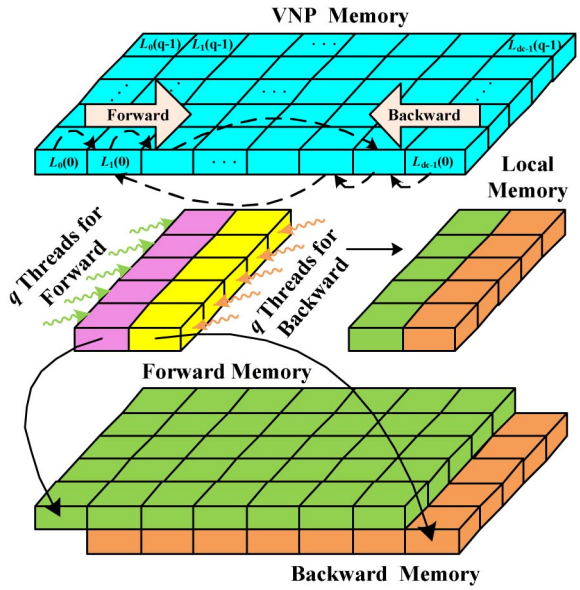


Fig. 5. Forward–backward kernel implementation of the FBA on a GPU.

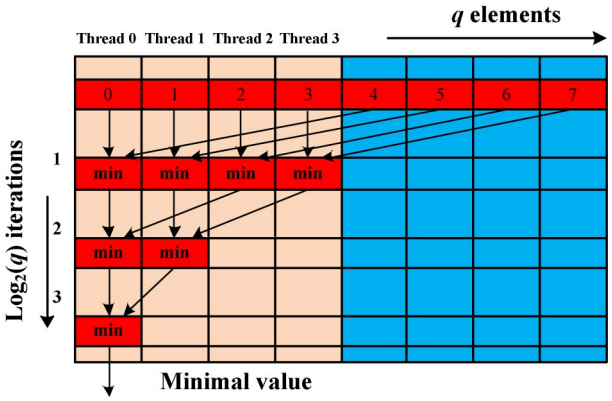


Fig. 6. Execution of the min-reduction algorithm.

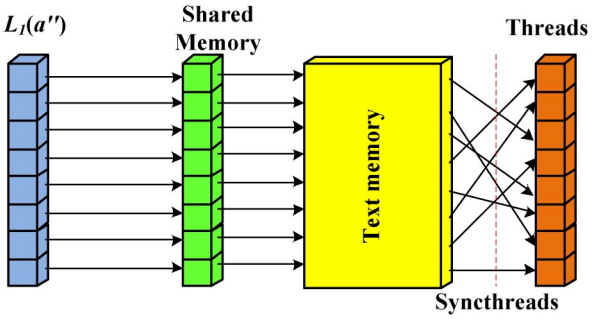


Fig. 7. Shared memory for random memory access.

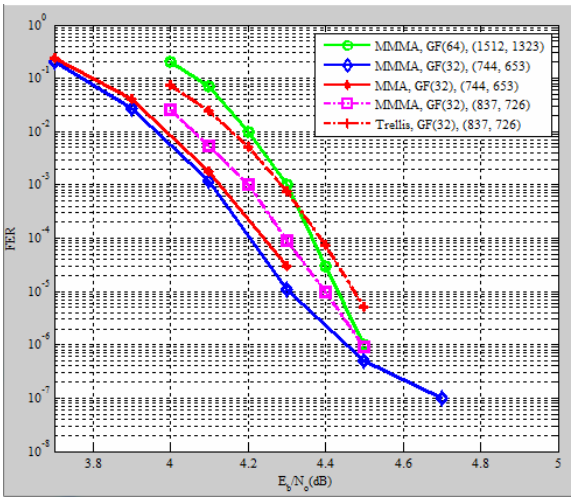
or merger vector) be an elementary step. To compute the output forward vector $F_{i+1}(a)$, the elementary step has two input message vectors: $F_i(a')$ and $L_{i+1}(a'')$. Using Eq. (2), there are q different pairs of $a' - a''$, which satisfy $a' + ha'' = a$ to find one message with index a of vector $F_{i+1}(a)$. Assume that q indexes of a' are in the order $\{0, \alpha^0, \ldots, \alpha^{30}\}$. Then, the q indexes of a'' that pair to a' are calculated, such that $a'' = (a' + a)/h$. After obtaining q pairs of $a' - a''$, the q messages are first generated by selecting the larger ones in each of the q pairs, $F_i(a') - L_{i+1}(a'')$. Then, the minimal value of the q messages is found and defaulted as an output message of the forward vector $F_{i+1}(a)$. A parallel reduction algorithm is applied to extract the minimal value from an array of q messages, as shown in Fig. 6. In this way, the min operation is processed in $\log_2(q)$ steps; after the necessary iterations, a min reduction algorithm returns the minimum value in its first index. The max and min operation in the forward computation are performed by q threads for the forward step in the CNP kernel. Similarly, the backward computations are performed by q threads for the backward step in the CNP kernel. As mentioned above, variable node messages $L_{i+1}(a')$ are stored for access in the order of linear memory. However, to compute forward messages $F_{i+1}(a)$, V2C messages $L_{i+1}(a'')$ are accessed in an arbitrary order, such as $a'' = (a' + a)/h$. To address this problem, the V2C messages are copied to on-chip shared memory, which has high bandwidth and low latency before beginning the computation. Then, the output messages are written using the indexes, which are computed by the lookup tables for additions and multiplications in text memory, as shown in Fig. 7. Thus, bank conflicts are not generated, and memory access is speeded up. Note that one addition is required to calculate a'' values as $a'' = a + a'$ from Eq. (8) for the merger messages of the proposed MMMA algorithm.

For example, vector $F_1(a) = \{F_1(0), F_1(\alpha^0), \ldots, F_1(\alpha^{30})\}$ needs to be computed if conditions such as $h = \alpha^2$, the current V2C vector $L_1(a'') = \{L_1(0), L_1(\alpha^0), \ldots, L_1(\alpha^{30})\}$, and the previous forward vector $F_0(a') = \{F_0(0), F_0(\alpha^0), \ldots, F_0(\alpha^{30})\}$ are known. To compute one message of the $F_1(a)$ output vector with index $a = \alpha^1$, there are 32 pairs for $a' - a''$ that satisfy $a' + \alpha^2 a'' = \alpha^1$, such as $\{0 - \alpha^{30}, \alpha^0 - \alpha^{16}, \alpha^1 - 0, \ldots, \alpha^{30} - \alpha^2\}$. Now, the indexes of a'' are in the order $\{\alpha^{30}, \alpha^{16}, 0, \ldots, \alpha^2\}$. After applying the max and min operation to these pairs $a' - a''$, the value of $F_1(\alpha^1)$ is found.

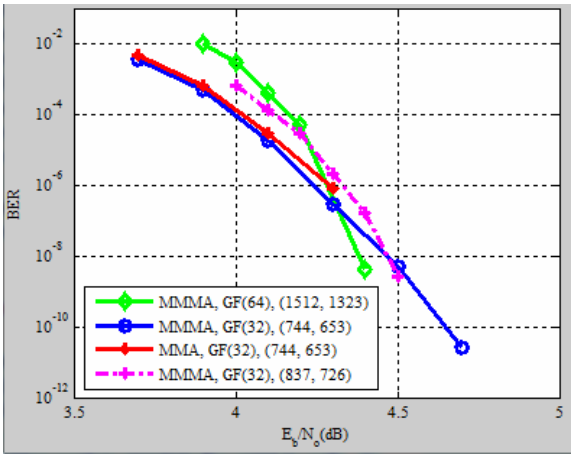
4. Experimental Results

The experimental setup to evaluate the performance of the proposed NB-QC-LDPC decoder consists of an NVIDIA GTX 650Ti GPU and an Intel Core i7-4770 CPU. A platform with the Intel Core i7-4770 CPU at 3.4 GHz and 16 GB RAM was used to simulate the serial C code. An NVIDIA GTX 650Ti GPU with 768 CUDA cores at 0.97 GHz and 1024 MB of GDDR5 device memory was used to implement the CUDA C code. Moreover, an NVIDIA GTX TITAN Black graphics card was applied to perform the CUDA C code. This work used the CUDA toolkit v5.5 for the implementation. Two NB-QC-LDPC codes over GF(32), such as (744, 653) with $(d_v, d_c) = (3, 24)$ and (837, 726) with $(d_v, d_c) = (4, 27)$, were used in this simulation. Furthermore, a GPU-based (1512, 1323) NB-QC-LDPC decoder with $(d_v, d_c) = (3, 24)$ over GF(64) was also investigated.

The decoding performance of NB-QC-LDPC code and its random counterpart over an additive white Gaussian noise (AWGN) channel with binary phase shift keying



(a)



(b)

Fig. 8. (a) FERs of NB-QC-LDPC codes over GF(32) and GF(64), (b) BERs of NB-QC-LDPC codes over GF(32) and GF(64).

(BPSK) is illustrated in Fig. 8. Figs. 8(a) and (b), respectively, show FERs and BERs of the proposed MMMA and previous works [10, 23] with I_{max}=15. We can see that the GPU accelerates processing to enable the detection of error floors of about 10^{-7} FER and 10^{-10} BER within days, instead of weeks, of computation in C++. Fig. 8(a) demonstrates clearly that the coding gains are in the waterfall type from 4.1 dB to 4.5 dB. These results show that GPU processing led to superior FER–BER performance, compared to CPU solutions [10, 23].

Table 3 shows the decoding runtime for (744, 653) NB-QC-LDPC code based on an MMMA using a single-core CPU platform and various GPU devices. Execution times were obtained with CPU timers. Using the GTX TITAN Black, the decoding runtime is 71.25 *ms* at 4 dB, which is 7.5 times faster than a single-core CPU-based implementation. Note that different GPU devices, which were set up on different CPU platforms, produced varying decoding runtimes. The GTX TITAN Black graphics card has more advantages over the GTX 650 Ti. Thus, the decoding runtime is approximately twice as fast as that of the GTX 650Ti.

Table 3. Decoding time of (744, 653) NB-LDPC code over GF(32) on different devices based on MMMA at I_{max} = 10.

E_b / N_0	Decoding time (ms)		
	CPU (Intel i7)	GTX 650Ti	GTX TITAN black
0 dB – 3 dB	2126	470	285
4 dB	531.5	117.5	71.25
5 dB	170.08	37.6	22.8
6 dB	127.56	28.2	17.1
7 dB	85.04	18.8	11.4
8 dB	63.78	14.1	8

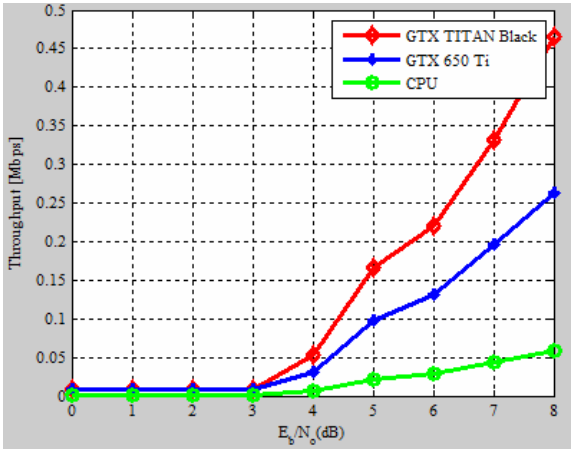


Fig. 9. Average decoder throughput on GPUs and a CPU at a maximum of 10 iterations.

Fig. 9 shows the average total throughput of the decoder under different channel qualities processed on the CPU platform and various GPU platforms. The two platforms are similar in terms of BER results. However, the speeds of the two platforms are different. The speeds were measured as the average runtime per decoding iteration with different E_b/N_0 values. In Fig. 9, from 1 dB to 3 dB, throughput remains fairly static at the lowest level. This is due to the bad channel performance in low E_b/N_0 values, and decoding has to be executed at a maximum of 10 iterations. The throughput increases with increasing E_b/N_0 because fewer decoding iterations have to be executed until the correct code word is recovered. The throughput of the GPU-based decoder is presented in Eq. (11), where $T_{host\rightarrow gpu}$, $T_{gpu\text{-}proc}$, and $T_{gpu\rightarrow host}$ are the time to copy data from the host CPU to the device GPU, the processing time on the GPU, and the time to copy data from the device GPU to the host CPU, respectively. N_{itr} is the number of average iterations simulated.

$$T_P = \frac{(q-1)\times d_c \times \log_2 q}{(T_{host\rightarrow gpu} + T_{gpu-proc} + T_{gpu\rightarrow host})\times N_{itr}}[bps] \quad (11)$$

Two factors affect the decoding runtime of the layered scheme in this study. First, it is dependent on the number of layers, because the decoding has to be sequentially performed on each layer to finish one iteration, instead of

one time per iteration in the flooding scheme. Thus, the decoding time of the layered scheme can be estimated to be d_v times higher than in the flooding scheme. However, the layered scheme doubly increases the convergence speed of the iterative decoder. This means that the number of required decoding iterations can be cut in half, compared to the flooding scheme. Secondly, the check node degree, or d_c, also impacts decoding runtime, because the FBA, used in the CNP, is sequentially implemented in (d_c-1) steps. It was concluded that the decoding time is a balance of achieving the same decoding performance between the layered and flooding schemes. Nonetheless, the layered scheme is much more efficient in storage memory than the flooding scheme. In addition, decoding runtime is also affected by the different algorithms used and the construction of the **H** matrix, such as (d_v, d_c). Beermann et al. [19] proposed implementation of an FFT-BP algorithm on a GPU; however, the construction of the **H** matrix was not given. Therefore, it is difficult to compare this work [19]. Andrade et al. [17] introduced FFT-SPA NB-LDPC decoding on a GPU, and binary images were used to construct the **H** matrix. The algorithm and the parameters of NB-LDPC codes in their work [17] are different from the proposed work. However, a fair comparison between the proposed work and that of Andrade et al. [17] in terms of execution time is given in the following analysis. First, execution time for Andrade et al. [17] only refers to the soft-decoding kernels, such as the VNP and the CNP, without mentioning the data copy time from/to CPU to/from GPU and the execution time in the hard-decoding kernels, such as FFT execution on V2C messages and the decision. By contrast, the execution time in the proposed work includes data copy time from/to CPU to/from GPU and the execution time of all kernels on the GPU platform, as shown in Fig. 2. In addition, the execution time in the proposed work is (d_c-1) × d_v times as high as that for Andrade et al. [17] because the FBA algorithm takes (d_c-1) steps for the CNP, and d_v layers are sequentially processed to complete one decoding iteration. For a maximum of 10 iterations, the proposed decoding time is 470 *ms* on the GTX 650 Ti, which is 23 × 3 (d_c = 24 and d_v = 3) times as high as the 14.93 *ms* decoding time for Andrade et al. [17] on a Tesla C1060 NVIDIA GPU. In fact, this number is reduced to only 31 (470/14.93) times because the complexity of the FBA min-max is smaller than that of the FFT-SPA algorithm. This result demonstrates that the decoding time on a GPU for both algorithms is appropriate. The advantage of the proposed work is that the error flow property of NB-LDPC codes with a higher Galois-field order is shown by simulations on the GPU, and the FBA min-max algorithm with less hardware complexity and good error-correcting performance can be designed for future applications, compared to the FFT-SPA algorithm. Wang et al. [18] executed a flooding scheme for a GPU-based min-max NB-QC-LDPC decoder using the FBA; the CNP kernel used q threads for each check node. In this approach, to take advantage of independent computation of the forward and backward steps, we use q threads for the forward step and q threads for the backward step to simultaneously process each step. As a result, the running time for the CNP kernel is estimated to be doubly reduced, compared to that achieved by Wang et al. [18]. Our work also adopts a layered scheme on a GPU, which significantly reduces the memory requirement, compared to Wang et al. [18].

Moreover, compared to the millisecond decoding runtime reached by the GPU-based binary LDPC decoders reported in previous work [15, 16], the decoding runtime measured in this experiment justifies the following complexity analysis. The complexity of check-node processing in a binary decoder is $O(d_c)$, whereas the one in a min-max nonbinary decoder is $O(d_c q^2)$. In addition, the nonbinary arithmetic needs four look-up tables for computations and conversions.

5. Conclusions

In this paper, we proposed a modified min-max decoding algorithm for NB-QC-LDPC codes and an efficient parallel block-layered decoder architecture corresponding to the algorithm on a GPU platform. This algorithm removes the multiplications over the Galois field in the merger step to provide a reduction in the number of computations without any performance loss. Owing to its inherently massive parallelism, NB-QC-LDPC decoding is suitable for implementation on a GPU. The experimental results show that the GPU-based implementation of the parallel block-layered decoder accelerates the decoding process to obtain a faster decoding runtime and higher coding gain under a low 10^{-10} BER and a low 10^{-7} FER. A new solution is thereby provided for NB-QC-LDPC decoding on a device GPU, which provides greater efficiency than on a host CPU platform.

Acknowledgement

This work was supported by MSIP, Korea, under the ITRC support program (IITP-2017-2014-0-00729) supervised by the IITP and in part by the Basic Science Research Program through the NRF funded by the Ministry of Science, ICT and Future Planning (2016R1A2B4015421).

References

[1] R.G. Gallager, Low-density parity-check codes, Information Theory, IRE Transactions on, vol.8, no.1, pp.21-28, 1962. Article (CrossRef Link)

[2] H.C. Davey, D.J. MacKay, Low density parity check codes over GF (q), in: Information Theory Workshop, pp. 70-71, 1998. Article (CrossRef Link)

[3] B. Zhou, J. Kang, S. Song, S. Lin, K. Abdel-Ghaffar, M. Xu, Construction of non-binary quasi-cyclic LDPC codes by arrays and array dispersions, IEEE Transactions on Communications, vol.57, pp.1652-1662, 2009. Article (CrossRef Link)

[4] V. Savin, Min-Max decoding for non binary LDPC codes, IEEE International Symposium on Information Theory (ISIT), pp. 960-964, 2008. Article (CrossRef

Link)
[5] F. Cai, Low-complexity Decoding Algorithms and Architectures for Non-binary LDPC Codes, Ph.D. thesis, Case Western Reserve University, 2013. Article (CrossRef Link)
[6] L. Barnault, D. Declercq, Fast decoding algorithm for LDPC over GF (2^q), in: IEEE Information Theory Workshop, pp. 70-73, 2003. Article (CrossRef Link)
[7] D. Declercq, M. Fossorier, Decoding algorithms for nonbinary LDPC codes over GF, IEEE Transactions on Communications, vol.55, pp.633-643, 2007. Article (CrossRef Link)
[8] X. Chen, S. Lin, V. Akella, Efficient configurable decoder architecture for nonbinary Quasi-Cyclic LDPC codes, IEEE Transactions on Circuits and Systems I: Regular Papers, vol.59, pp.188-197, 2012. Article (CrossRef Link)
[9] C.-S. Choi, H. Lee, Block-Layered Decoder Architecture for Quasi-Cyclic Nonbinary LDPC Codes, Journal of Signal Processing Systems, vol.78, no.2, pp.209-222, Feb. 2015. Article (CrossRef Link)
[10] X. Zhang, F. Cai, Efficient partial-parallel decoder architecture for quasi-cyclic nonbinary LDPC codes, IEEE Transactions on Circuits and Systems I: Regular Papers, vol.58, pp.402-414, 2011. Article (CrossRef Link)
[11] D. Kirk, W. mei w. hwu, Programming Massively Parallel Processors: A Hands-on Approach, Burlington, MA, USA, 2010.
[12] S. Wang, S. Cheng, Q. Wu, A parallel decoding algorithm of LDPC codes using CUDA, in: Signals, systems and computers, 42nd asilomar conference on, pp. 171-175, 2008. Article (CrossRef Link)
[13] G. Falcão, L. Sousa, V. Silva, Massive parallel LDPC decoding on GPU, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pp. 83-90, 2008. Article (CrossRef Link)
[14] Y. Zhao, X. Chen, C.-W. Sham, W.M. Tam, F.C. Lau, Efficient decoding of QC-LDPC codes using GPUs, in: International Conference on Algorithms and Architectures for Parallel Processing, Springer, pp. 294-305, 2011. Article (CrossRef Link)
[15] G. Falcao, L. Sousa, V. Silva, Massively LDPC decoding on multicore architectures, IEEE Transactions on Parallel and Distributed Systems, vol.22, pp.309-322, 2011. Article (CrossRef Link)
[16] B. Le Gal, C. Jego, J. Crenne, A high throughput efficient approach for decoding LDPC codes onto GPU devices, IEEE Embedded Systems Letters, vol.6, pp.29-32, 2014. Article (CrossRef Link)
[17] J. Andrade, G. Falcao, V. Silva, K. Kasai, FFT-SPA non-binary LDPC decoding on GPU, in: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 5099-5103, 2013. Article (CrossRef Link)
[18] G. Wang, H. Shen, B. Yin, M. Wu, Y. Sun, J.R. Cavallaro, Parallel nonbinary LDPC decoding on GPU, in: Asilomar Conference on Signals, Systems and Computers (ASILOMAR), pp. 1277-1281, 2012. Article (CrossRef Link)
[19] M. Beermann, E. Monró, L. Schmalen, P. Vary, High speed decoding of non-binary irregular LDPC codes using GPUs, in: IEEE Workshop on Signal Processing Systems (SiPS), pp. 36-41, 2013. Article (CrossRef Link)
[20] M. Beermann, E. Monzó, L. Schmalen, P. Vary, GPU accelerated belief propagation decoding of non-binary LDPC codes with parallel and sequential scheduling, Journal of Signal Processing Systems, vol.78, pp.21-34, Jan. 2015. Article (CrossRef Link)
[21] H. Song, J. Cruz, Reduced-complexity decoding of Q-ary LDPC codes for magnetic recording, IEEE Transactions on Magnetics, vol.39, pp.1081-1087, 2003. Article (CrossRef Link)
[22] H. Wymeersch, H. Steendam, M. Moeneclaey, Log-domain decoding of LDPC codes over GF (q), in: IEEE International Conference on Communications, pp. 772-776, 2004. Article (CrossRef Link)
[23] F. Cai, X. Zhang, Relaxed min-max decoder architectures for nonbinary low-density parity-check codes, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol.21, no.11, pp.2010-2023, 2013. Article (CrossRef Link)

Huyen Pham Thi received a B.S. degree in Information and Communication Engineering in 2011 from Military Technical Academe in VietNam. She is currently working toward her M.S and Ph.D integrated degree in Information and Communication Engineering in Inha University, Korea. Her research interests include forward error correction algorithms, and VLSI architecture design and implementation for communications.

Hanho Lee received the M.S. and Ph.D. degrees in Electrical & Computer Engineering from the University of Minnesota, Minneapolis, in 1996 and 2000. In 1999, he was a Member of Technical Staff-1 at Lucent Technologies, Bell Labs, Holmdel, New Jersey. From April 2000 to August 2002, he was a Member of Technical Staff at the Lucent Technologies (Bell Labs Innovations), Allentown. From August 2002 to August 2004, he was an Assistant Professor at the Department of Electrical and Computer Engineering, University of Connecticut, USA. Since August 2004, he has been with the Department of Information and Communication Engineering, Inha University, where he is currently Professor. From August 2010 to August 2011, he was a visiting scholar at Bell Labs, Alcatel-Lucent, Murray Hill, New Jersey, USA. His research interest includes VLSI architectures design for digital signal processing, forward error correction architectures, cryptographic systems, and communications.