PAPER

Efficient Parallel Block-Layered Nonbinary LDPC Decoding on a GPU

Huyen Pham Thi[†] , **Sabooh Ajaz**[†], *Nonmember and* **Hanho Lee**[†], *Member*

SUMMARY This paper presents a novel modified Min-Max algorithm (MMMA) and an efficient implementation of a parallel, block-layered nonbinary low-density parity-check (NB-LDPC) decoding on a graphics processing unit (GPU) to achieve greater flexibility and scalability. An algorithm and data structures suitable for parallel computing are proposed in this paper to perform NB-LDPC decoding on a GPU. The MMMA for check node processing removes the multiplications over the Galois field in the merge step and significantly reduces decoding latency. The proposed MMMA provides better bit error rate (BER) and frame error rate (FER) performance than previous Min-Max algorithms. The experimental result shows that our GPU-based NB-LDPC decoder using MMMA provides faster run-time and higher coding gain under a low 10^{-10} BER and a low 10^{-7} FER compared to a CPU-based implementation.

.
key words: *Nonbinary LDPC, GPU, parallel computation, CUDA ..*

1. Introduction

Gallager investigated a binary low-density parity-check code that provided performance close to the Shannon limit for long code lengths [1]. Recently, nonbinary LDPC (NB-LDPC) codes [2-6] have attracted a tremendous amount of research interest owing to their excellent error correction capabilities. Matthew and MacKay [2] showed that NB-LDPC codes provide significant performance improvement when the code lengths are short and moderate. However, the decoding algorithms for NB-LDPC codes can require complex computations and a large memory [6].

The belief propagation (BP) algorithm using a fast Fourier transform (FFT) in the probability domain can be used for NB-LDPC codes, which reduces the computational complexity from $O(q^2)$ to $O(q\log_2 q)$. However, in the probability domain algorithms, a large number of additions and multiplications can lead to an increase in hardware complexity. To deal with this problem, the extended Min-Sum (EMS) [7] and Min-Max algorithms use log-likelihood ratio (LLR) values to decode channel messages, which multiplications are replaced by additions in the log-domain. Both of these algorithms have been attracted for very-large scale integration implementation. In [4] Savin used 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. Therefore, the Min-Max algorithm was used in some NB-LDPC decoder architectures [5, 13-16]. Also, it was demonstrated that a layered decoding can be used to reduce the memory requirement and increase the convergence speed.

Demand for new codes and a novel low-complexity decoding algorithm for NB-LDPC codes requires a huge number of extensive simulations. Due to the high complexity of NB-LDPC decoding algorithms, the simulation time on a central processing unit (CPU) is extremely slow in 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).

Recently, graphic processing units (GPUs) are widely used for their high computational power, which can execute numerous threads simultaneously and the peak performance can reach up to tetra floating operations per second. The NVIDIA corporation presented Compute Unified Device Architecture (CUDA) [8], using C as a high-level programming language, which offers a software environment that facilitates the development of high-performance applications. Recently, there has been growing interest in GPU-based implementations [9-12], which can push higher simulation speeds for LDPC decoding. GPU can provide massively parallel computation threads with a many-core architecture, which can accelerate simulations of NB-LDPC decoding. Even though many implementations of binary LDPC code on a GPU have been proposed [9, 10-12], the implementation of NB-LDPC codes is still very challenging.

In this paper, a novel modified Min-Max algorithm (MMMA) and an efficient implementation of parallel block-layered NB-LDPC decoding based on a GPU is presented. Algorithms and data structures suitable for parallel computing are proposed to perform NB-LDPC decoding on a GPU. The MMMA for check node processing removes the multiplications over the Galois field in the merge step and significantly reduces decoding latency.

This paper is organized as follows. In Section II, we briefly review the NB-LDPC codes and propose a modified Min-Max algorithm. Then, Section III

Manuscript received January xx, 20xx.
Manuscript revised March xx, 20xx.
† Authors are with Department of Information and Communication Engineering, Inha University, Incheon, 402-751, Korea. Email: hhlee@inha.ac.kr

describes the proposed parallel architecture and implementation on a GPU using the Compute Unified Device Architecture (CUDA). Experimental results are presented in Section IV. Finally, Section V concludes the paper.

2. NB-LDPC Codes and Decoding Algorithms

2.1 NB-LDPC Codes

An (N, K) NB-LDPC code (with N code symbols, K information symbols, and $M = N\text{-}K$ parity symbols) is defined by a parity-check matrix **H**, which includes a small proportion of Galois-field elements. An NB-LDPC code can be illustrated with a Tanner graph. Each row of the **H** matrix is connected with a check node, and each column of the **H** matrix is connected with a variable node on the Tanner graph. These codes introduce good bit error rate performance and efficient parallel processing. Zhou et al. [3] presented two new algebraic constructions for NB-LDPC codes based on array dispersions of matrices. The parity check matrices of quasi-cyclic (QC)-NB-LDPC codes are row–column (RC) constrained arrays, which can be extended by cyclically permutation matrices (CPM) over nonbinary Galois-fields, which provide high performance. The structural property of an RC-constraint is a constraint on the rows and columns of the **H** matrix. A (744, 653) NB-LDPC code over $GF(2^5)$ is generated by using RC-constrained arrays, which we used in our simulation. The construction process of RC-constraint code is described in detail by Zhou et al. [3]. By using their method [3], a 31×31 **H** matrix is generated. For a pair (γ=3, ρ=24), let **H** (3,24) be a 3×24 subarray of **H**. Each of elements in subarray **H**(3,24) is dispersed either an all-zero matrix of size (q-1)×(q-1) or an α-multiplied CPM of size (q-1)×(q-1). There is only one i-th nonzero entry in the first row of the matrix, which is generated by dispersing an element α^i, and other entries are zero. Each of other rows is a right cyclic-shift of the previous row multiplied by α. A (744,653) NB-LDPC code is constructed by subarray **H**(3,24) with column weight $d_v = 3$ and row weight $d_c = 24$. Figure 1(a) and (b) show the H-matrix of (744,653)NB-LDPC code over GF(2^5) and an an α-multiplied CPM for α^2.

2.2 Parallel Block-Layered Min-Max Algorithm

Horizontal layered decoding was widely adopted to reduce the memory requirement and the number of decoding iterations [5, 13-15]. *Algorithm 1* shows the proposed parallel block-layered Min-Max decoding algorithm in which non-overlapped (q-1) rows are grouped into one block layer, and each column of these block layers has a weight value of one. The NB-LDPC decoding algorithm given in *Algorithm 1* can be briefly summarized as follow: the layered decoding divides the

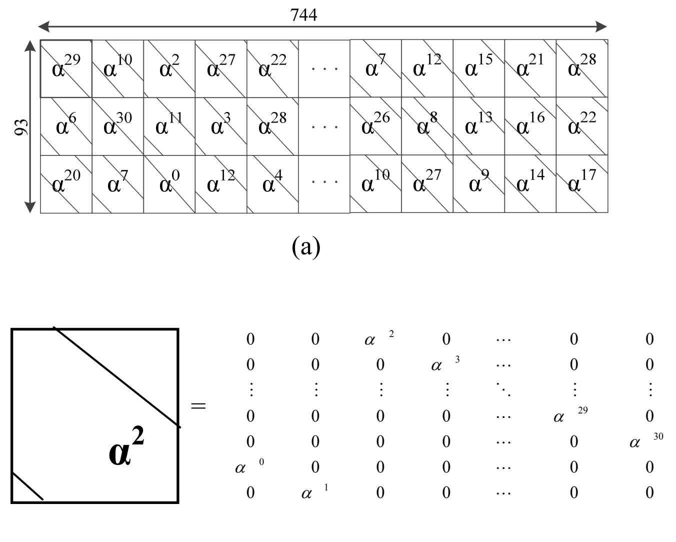

744
93
α29 α10 α2 α27 α22 · · · α7 α12 α15 α21 α28
α6 α30 α11 α3 α28 · · · α26 α8 α13 α16 α22
α20 α7 α0 α12 α4 · · · α10 α27 α9 α14 α17
(a)
α2
=
0 0 α 2 0 ··· 0 0
0 0 0 α 3 ··· 0 0
0 0 0 0 ··· α 29 0
0 0 0 0 ··· 0 α 30
α 0 0 0 0 ··· 0 0
0 α 1 0 0 ··· 0 0
(b)

Fig. 1 (a) H-matrix for a (744,653) QC-NB-LDPC code over GF(2^5). (b) Example of α-multiplied CPM for α^2.

rows of **H** matrix into d_v layers, where the size of each group is (q-1) rows (check nodes).

In the layered decoding, both check node processing and variable node processing for layer 0, layer 1, …, and layer $d_v - 1$ are sequentially performed to complete a single iteration, and the extrinsic values are exchanged among the layers. The initialization of the parallel block-layered Min-Max decoding algorithm is the same as that described in Algorithm 1. In addition, the variable node (V2C) messages of the l layer in iteration k are computed as follows:

$$\tilde{L}^{k,l}_{nm}(a) = L^{k,l-1}_{n}(a) - R^{k-1,l}_{mn}(a) \quad (1)$$

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

Let x_n be the n-th symbol in a received codeword, and let s_n be the most likely symbol of x_n. The $L_n(a)$ vector has q elements, including one zero element and (q-1) positive elements. This process is consecutively performed until the number of iterations reaches I_{max} or the parity check is satisfied. In the parallel block-layered decoding algorithm, iterations are sequentially implemented in L layers, and each layer simultaneously processes (q-1) check nodes. The Min-Max decoding, which is implemented by a well-known forward-backward algorithm, is applied in the check node process.

Algorithm 1: GPU-based parallel block-layered Min-Max decoding algorithm
Initialization:
$L_n(a) = \ln(\Pr(c_n = s_n|channel) / \Pr(c_n=a|channel))$;
$L_n^{1,0}(a) = L_n(a)$; $R_{mn}^{\ 0,1}(a)=0$;
Iterations:

```
For ($k$=1; $k$<=$I_{max}$; $k$++)
  For ($l$=1; $l$<=$L$; $l$++)
    For ($m$=0; $m$<$q$-1; $m$++)
      Step1:
        $\widetilde{L}_{nm}^{k,l}(a) = L_n^{k,l-1}(a) - R_{mn}^{k-1,l}(a)$
        $\widetilde{L}_{nm}^{k,l} = \min_{a \in GF(q)} \left(\widetilde{L}_{nm}^{k,l}(a)\right)$
        $L_{nm}^{k,l}(a) = \widetilde{L}_{nm}^{k,l}(a) - \widetilde{L}_{nm}^{k,l}$
      Step2:
        $R_{mn}^{k,l}(a) = \min_{(a_{n'})_{n' \in N(m)} \in \gamma_{mn}(a)} \left( \max_{n' \in N(m \setminus \{n\})} \left( L_{n'm}^{k,l}(a_{n'}) \right) \right)$
      Step3:
        $L_n^{k,l}(a) = \widetilde{L}_{nm}^{k,l}(a) + R_{mn}^{k,l}(a)$
    End for
  End for
Decision:
$\widetilde{C}_n = \arg\min\left(L_n^{k,l}(a)\right)$
End for
```

2.3 Modified Elementary Processing Step for the Min-max Algorithm

In the proposed algorithm, the finite-field elements of power representation are expressed following the order $\{0, \alpha^0, \alpha^1, \ldots \alpha^{q-2}\}$, where α is a primitive element of $GF(q)$. The log-likelihood ratios (LLRs) are stored as a vector, indexed in ascending order of α. Multiplying a vector, which includes all nonzero field elements $\{\alpha^0, \alpha^1, \ldots \alpha^{q-2}\}$ with a nonzero field element, is equivalent to cyclically shifting the vector by a power of α. In addition, a', a'' and a elements are Galois-field elements, which are used as an index of the message vector for variable node n_i. Let m be a check node and let $N(m) = \{n_0, n_1, \ldots, n_{dc-1}\}$ be the set of variable nodes connected to m in the Tanner graph. There are three steps to find the C2V messages, which include forward step, backward step and merge step [4]. Forward metrics $(F_i)_{i=0,dc-2}$ and backward metrics $(B_i)_{i=1,dc-1}$ are calculated sequentially with a conditional equation as follows:

Conditional equation for forward step and backward step:

$$a' + \alpha^{h_{v_i c}} a'' = a \tag{2}$$

Forward metrics:

First step:

$$F_0(a) = L_{v_1 c}(\alpha^{h^{-1}{}_{v_0 c}}(a)) \tag{3}$$

Recursive step: for i=1, d_c-2

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

Backward metric:

First step:

$$B_{d_c-1}(a) = L_{v_{d_c-1} c}(\alpha^{h^{-1}{}_{v_{d_c-1} c}}(a)) \tag{5}$$

Recursive step: for i= d_c-2, 1

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

In the first step, the forward messages are generated by multiplying with inversion of the first nonzero entry of the m row, as shown in equation (3). Then, the recursive steps are calculated from previous forward messages and current variable node messages, as shown in equation (4). In conditional equation (2), V2C messages a'' are multiplied with the nonzero elements $\alpha^{hvi,c}$ of the **H** matrix because V2C messages are directly sent to the check node. The previous forward messages a' are multiplied with the nonzero elements of the **H** matrix from the previous steps; hence, the previous forward messages are directed to the conditional equation, as shown in equation (2). This process is similar to backward processing.

In the merge step, the merge messages are C2V messages, which are updated for the a posteriori messages. When the check node degree is equal to d_c (that is, after finishing the forward and backward processing), two vectors of merge processing are found, such as $M_{cv0}(a)$ and $M_{c,v_{d_c-1}}(a)$. We note that $M_{cv0}(a)$ is equal to $B_1(a)$, and $M_{c,v_{d_c-1}}(a)$ is equal to $F_{dc-2}(a)$. The forward and backward messages, which are already multiplied with nonzero elements of the **H** matrix in forward-backward processing in equation (2), are used directly to generate other merge vectors. Therefore, new conditional equation (7) and computing merge metric (9) are proposed for the merge step in the decoding algorithm, as follows:

Conditional equation for merge step:

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

Merge metrics:

$$M_{c,v_0}(a) = B_1(a); M_{c,v_{d_c-1}}(a) = F_{d_c-2}(a) \tag{8}$$

$$M_{cv_k}(a) = \min_{\substack{a', a'' \in GF(q) \\ a' + a'' = a}} \left(\max\left(F_{k-1}(a'), B_{k+1}(a'') \right) \right) \tag{9}$$

3. Parallel Block-Layered NB-LDPC Decoding on GPU using CUDA

3.1 Data Flow of NB-LDPC Decoding

NVIDIA GPUs are powerful arithmetic engines capable of running thousands of lightweight threads in parallel. A GPU-based heterogeneous platform has one or more CPUs and GPUs well-suited to implementing NB-LDPC decoding algorithms. Furthermore, the NB-LDPC decoding algorithm has a high computation to memory access ratio (CMAR). The CMAR is the complexity of operations that justify the cost of moving

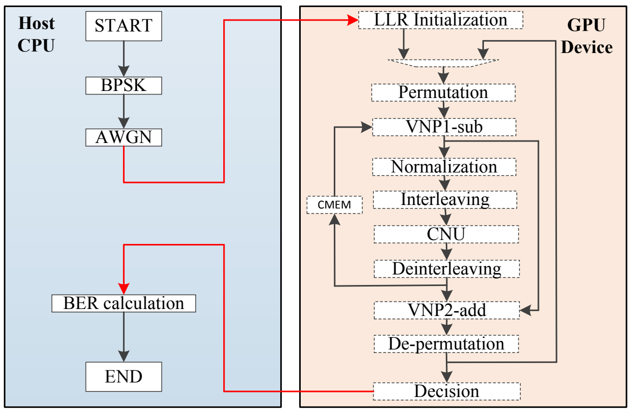


Fig. 2 Data flow of parallel block-layered NB-LDPC decoding on CPU and GPU platform.

data to and from the device. To profit from modern processor architecture–integrated GPUs, the first step is to assess the application 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 CPU, which shows that the check node processing is a bottleneck and accounts for 95.18% in the processing time.

Fig. 2 shows a data flow for parallel block-layered NB-LDPC decoding on CPU and GPU platforms. The CUDA program is divided into two tasks: one is for the CPU and the other for the GPU. The host system (CPU) is in charge of kernel scheduling, control of the decoding iterations, computing BER performance, and so on. The host CPU needs to transfer the symbols of the received codeword to the GPU, and receives the decoded symbols from the GPU. Most of the decoding computations are implemented on the GPU, and all the intermediate messages are stored in the device memory to restrict data transfer between host and device. Each of the modules in Fig. 2 responds to a kernel that we map on the GPU. 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 subtraction, the messages are used as input messages for the normalization block. Normalized output messages for consecutive check nodes are not aligned in GPU memory, so interleaving has to be performed before check node processing. Similarly, de-interleaving is performed after check node processing is finished. The first layer in the first iteration, which is the reliability information from the channel, is V2C messages, and CMEM values are equal to zero.

3.2 Data and Memory Structure

As described in Section 2.1, the **H** matrix is constituted from [d_v, d_c] submatrices, where the elements are extended by α-multiplied CPM size (q-1)×(q-1).

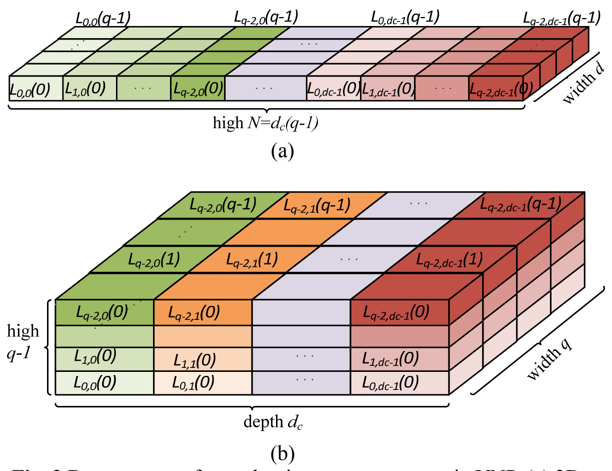


Fig. 3 Data structure for coalescing memory access in VNP. (a) 2D structure of LLRs and VN messages, (b) 3D structure of LLRs and VN messages.

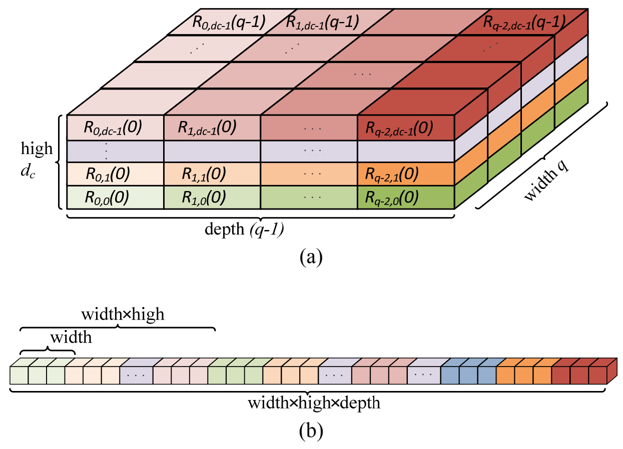
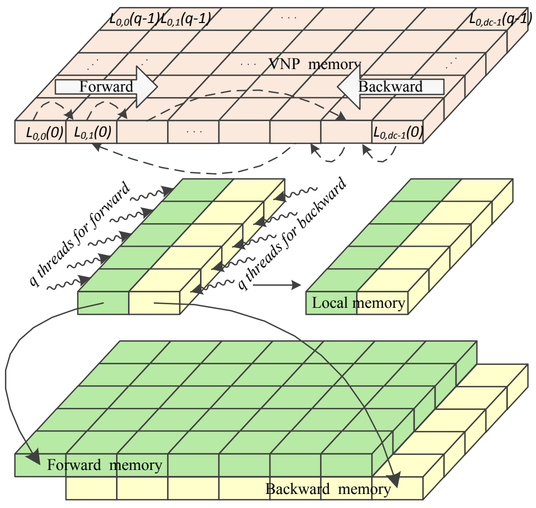


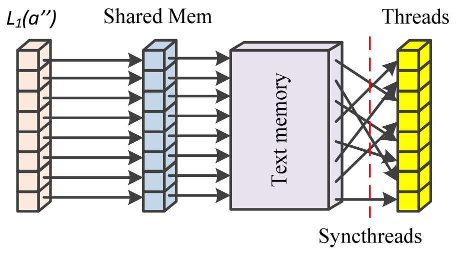
Fig. 4 Data structure for coalescing memory access in CNP. (a) 3D structure of CN messages, (b) 1D structure of CN messages.

Accordingly, it can be divided into d_v layers and the computations for (q-1) rows of **H** are carried out at a time. Fig. 3(a) shows a 2D structure of [q, (q-1)×d_c], corresponding to q (index: 0, 1, 2…, q-1) entries in a vector and (q-1)×d_c vectors of LLRs or VN messages ((q-1) (index: 0, 1, 2,…, q-2) vectors in a block, d_c (index: 0, 1, 2,…, d_c-1) blocks). Since a total of d_c × (q-1) variable node (VN) messages are distributed within d_c blocks operate separately, the 2D structure is reordered to generate a 3D-structure size [q, q-1, d_c], as shown in Fig. 3(b) (the width q corresponds to the q entries in a vector, the height (q-1) corresponds to (q-1) vectors in one block, and the depth is d_c blocks). Because (q-1) rows correspond to (q-1) check nodes of **H** operate separately, Fig. 4(a) depicts a 3D structure of check node (CN) messages [q, d_c, q-1] (the width q corresponds to q entries in a vector, the height d_c is d_c LLR vectors connect to one check node, and the depth is (q-1) check nodes).

If a 3D array is formatted by [*width, height, depth*],



(a)



(b)

Fig. 5 (a) Forward-backward kernel implementation of the forward-backward algorithm on a GPU. (b) Using local memory for random memory access.

Table 1 Kernel architecture of main blocks.

Function		No. thread blocks	No. threads	Total No. threads
Init LLR		d_c	$q\times(q\text{-}1)$	$d_c\times q\times(q\text{-}1)$
CNP	Forward-backward	$q\text{-}1$	$q+q$	$(q\text{-}1)\times(q+q)$
	Merge	$q\text{-}1$	$q\times d_c$	$(q\text{-}1)\times q\times d_c$
VNP		d_c	$q\times(q\text{-}1)$	$d_c\times q\times(q\text{-}1)$
Decision		1	$d_c\times(q\text{-}1)$	$d_c\times(q\text{-}1)$

then each element $[x, y, z]$ of an array is uniquely indexed by $[x + y\times width + z\times width\times height]$ in 1D array Fig. 4(b). By arranging $L_{nm}^{k,l}(a)$ and $R_{mn}^{k,l}(a)$ in this format, the q adjacent data entries can be accessed by q adjacent threads, so coalesced memory access is enabled, which achieves high throughput. In $GF(q)$, addition and subtraction can be implemented by exclusive or (XOR) operations, and division by α^a can be computed by multiplication with $\alpha^{(31-a)\%31}$. Therefore, to implement nonbinary arithmetic in $GF(q)$, the GPU's texture memory is employed, which is available to all kernels. There are two 2D lookup tables of size $q\times q$ for multiplication and addition, in addition to two 1D lookup tables of size q for conversion between exponential and decimal representation. We have also used a 64 KB

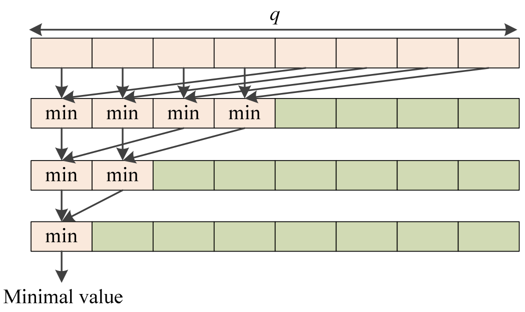


Fig. 6 Execution of a min reduction algorithm.

constant memory to store values from the parity check matrix, which are actually the values and the indices of bit nodes connected to each check node.

3.3 Parallel Forward-Backward Scheme in CNP

The proposed decoding algorithm is partitioned into four kernels, which are listed in Table I. Note that each check node sequentially computes forward, backward and merge messages in the MMMA. However, forward and backward messages are processed simultaneously by using q threads for forward and q threads for backward. After forward-backward messages are available, the merge computation is started. Fig. 5 (a) shows the architecture of a detailed kernel for one check node. Input messages for the kernel are stored in variable node processing (VNP) memory and output forward-backward messages are kept in both forward and backward memories to continue computing merge messages and an on-chip local memory to compute next forward-backward messages.

Since the $F_i(a)$ and $B_j(a)$ messages are used to compute $F_{i+1}(a)$ and $B_{j-1}(a)$ in the next step, the on-chip local memory which has high bandwidth and low latency to store output elementary messages $F_i(a)$ and $B_j(a)$ of each check node is used to speed up the elementary steps. The amount of memory required for $F_i(a)$ and $B_j(a)$ in $(q\text{-}1)$ check nodes is $2\times q\times sizeof(float)\times(q\text{-}1)$. For example, 7.75KB of local memory is required for 31 check nodes in $GF(2^5)$. A barrier synchronization function, __syncthreads(), is performed after each forward or backward step, $F_i(a)$ or $B_j(a)$, to ensure that threads are synchronized.

An elementary step has two input message vectors, which are denoted by $F_i(a')$ and $L_{i+1}(a'')$. One output message vector, $F_{i+1}(a)$, consists of q messages generated from the elementary step. Using equations (2) and (7), there are q different pairs of a' and a'' to satisfy $a' + ha'' = a$ in the forward-backward step, and $a' + a'' = a$ in the merge step. Suppose that $F_1(a) = \{ F_1(0), F_1(\alpha^0),\ldots F_1(\alpha^{30})\}$ are computed when $h = \alpha^2$, variable node messages $L_1(a'') = \{ L_1(0), L_1(\alpha^0),\ldots L_1(\alpha^{30})\}$ and previous forward messages $F_0(a') = \{ F_0(0), F_0(\alpha^0),\ldots F_0(\alpha^{30})\}$ are known. To compute a message of the $F_1(a)$

vector, q combinations of $F_0(a')$ and $L_1(a'')$ have to be determined by substituting the symbols into a' and a''. After that q messages are generated first by selecting the larger ones in each of the q pairs $F_0(a')$ - $L_1(a'')$. Then the minimal value of the q messages is found and defaulted as an output message of the forward messages $F_1(a)$. A parallel reduction algorithm [17] is applied to extract the minimal value from an array of q messages, as shown in Fig .6. In this way, the min is processed in $\log_2 q$ steps, and after the necessary iterations, a min reduction algorithm returns 0 in its first index. For example, with $h = \alpha^2$, $a = \alpha$, there are 32 pairs that satisfy: $a' + \alpha^2 a''=\alpha$ as follows: a'- a''= {0 - α^{30}, α^0- $\alpha^{16,}$ α^1- 0, …, α ^{30}- α ^{2}}. As shown above, variable node messages $L_1(a'')$ are stored to access in order of a linear memory. However, to compute forward messages $F_1(a)$ in elementary step, variable node messages $L_1(a'')$ are accessed in an arbitrary order. In this case the order to access the $L_1(a'')$ messages follows { α^{30}, α^{16}, 0, …, α ^{2} }. To deal with this problem, variable node messages are copied to an on-chip share-memory, which has high bandwidth and low latency before beginning the computation. Moreover, the additions and divisions are usefully implemented by lookup tables in the text memory. In this way, firstly variable node messages are copied directly to the shared memory. Then the output is written using the indexes are computed in the text memory as shown in Fig. 5 (b). Thus bank conflicts are not generated and memory access is speeded up.

Given a' values, a'' values are depicted by $a''= (a'+ \alpha)/ \alpha^2$. The calculation needs one addition and one division to determine a'' because of equation (2). However, the merge step of the proposed MMMA algorithm requires only one addition to calculate $a'' = a + a'$. It is obvious that we can reduce computation complexity for a merge step in comparison with the original forward-backward Min-Max algorithm [4], which is extremely useful when the decoder is implemented on a GPU.

4. Experimental Results

The experimental setup to evaluate the performance of the proposed NB-LDPC decoder consists of an NVIDIA GTX650Ti GPU and an Intel Core i7-4770 CPU. The CPU platform of an Intel Core i7-4770 CPU at 3.4 GHz with 16 GB RAM was used to simulate serial C code, and the NVIDIA GTX650Ti GPU with 768 CUDA cores at 0.97 GHz and with 1024 MB of GDDR5 device memory was used to implement CUDA C code. Moreover, a NVIDIA GTX TITAN Black graphics card is applied to perform CUDA C code. This work used CUDA toolkit v5.5 for the implementation. A regular (744, 653) NB-LDPC code constructed over $GF(2^5)$ with an 0.877 code rate was used in this simulation.

Decoding performance of (744, 653) NB-LDPC code

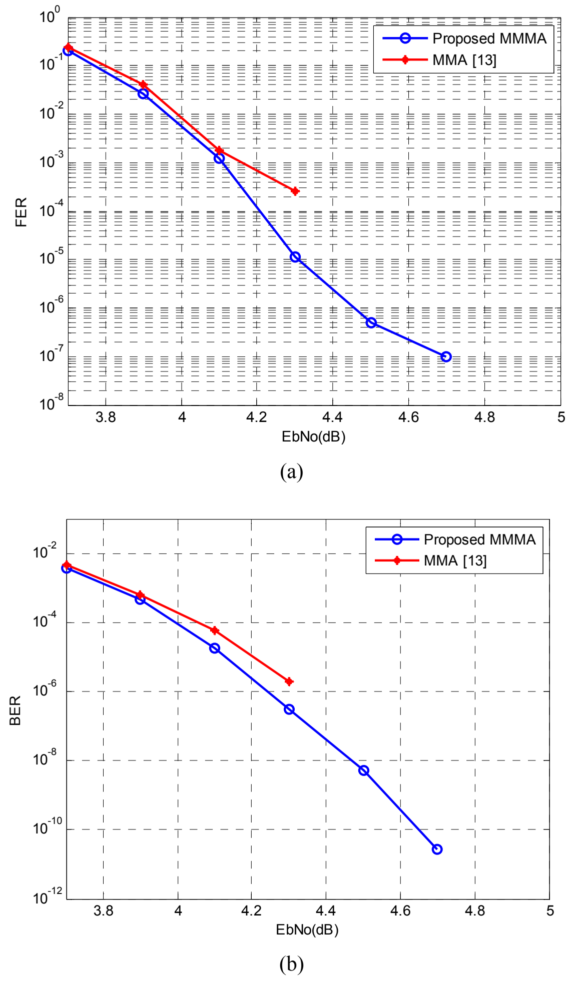


Fig. 7 (a) FERs of a (744, 653) NB-LDPC code over $GF(2^5)$. (b) BERs of a (744, 653) NB-LDPC code over $GF(2^5)$ using GPU.

Table 2 Coding gain comparison of various algorithms on different devices.

Designs		This work		[13]	This work	[16]
Code		(744,653)			(837,726)	
Algorithm		MMMA		MMA	MMMA	MMA
No. iter		15		15	15	15
Program		C++	CUDA C	C++	CUDA C	C++
Coding gain	at 10^{-5} BER	4.13	4.13	4.3	-	-
	at 10^{-10} BER	-	4.64	-	-	-
	at 10^{-5} FER	4.3	4.3	-	4.38	4.47
	at 10^{-7} FER	-	4.7	-	-	-

and its random counterpart over an additive white Gaussian noise (AWGN) channel with binary phase shift keying (BPSK) is illustrated in Fig. 7. Figs. 7 (a) and (b) show FER and BER, respectively, of the proposed MMMA and a conventional MMA [13] for a (744, 653) NB-LDPC code over $GF(2^5)$ with 15 decoding iterations. Performance results in Table 2 show that the proposed NB-LDPC decoding algorithm with 15 iterations can obtain 4.3 dB coding gain at a 10^{-5} FER, and a 4.13 dB coding gain at a 10^{-5} BER, which is approximately 0.17 dB higher at a 10^{-5} BER compared to Zhang and Cai's result [13] for the same code length over $GF(2^5)$.

Table 3 Decoding time at various iterations on different devices.

No. iteration	Decoding time (s)		
	CPU (Intel i7)	GTX 650 Ti	GTX TITAN Black
5	1.063	0.235	0.143
10	2.053	0.470	0.273
15	3.155	0.707	0.400

Moreover, the GPU accelerates to achieve the coding gain under a low 10^{-10} BER and a low 10^{-7} FER within days, instead of weeks of computation in C++.In Fig. 7(a), we can see that compared with conventional MMA using C++, the proposed MMMA using CUDA shows coding gain under a low 10^{-7} FER. In [16], a relaxed Min-Max check node processing is applied in the check node processing, which significantly reduced the computation complexity in comparison with forward-backward check node processing and provide the same performance. To compare with proposed MMMA algorithm, a (837,726) NB-LDPC decoding is performed on GPU platform with the proposed MMMA algorithm, which reached around 0.1dB better coding gain at 10^{-5} FER than that in [16].

Table 3 shows the decoding runtime using the CPU platform and various GPU devices. Execution times were obtained with CPU timers. In (744, 653) NB-LDPC decoding on the GTX TITAN Black, the decoding runtime is 0.273 second, which is 7.5 times faster than a CPU-based implementation. On the other hand, different GPU devices are set up on different CPU platform give varying decoding runtime. The GTX TITAN Black graphic card has more advantage than GTX 650 Ti, thus, the decoding runtime is almost double faster than the GTX 650 Ti. By using the multithreaded data-parallel processing units of GPUs, software LDPC decoders using a GPU can expect to achieve performance an order of magnitude higher than modern multi-core CPUs. Moreover, since general-purpose GPUs are able to perform floating-point arithmetic operations, better accuracy and a lower BER can be expected with very large-scale integration LDPC decoders.

Fig. 8 shows the average runtime of (744,653) layered NB-LDPC decoding with three layers per iteration (d_v=3) that is implemented on the CPU platform and GPU platform with maximum of 10 iterations. Two schemes are almost similar the BER results. However, the speeds of two schemes are different, and are measured by the average runtime per decoding iteration with different E_b/N_0 values. The average runtime values decrease with increasing of E_b/N_0 since the fewer decoding iterations have to be processed when channel performance is good. As can be seen from the Fig.9, from 1dB to 3dB the runtime of CPU platform and GPU platform remained fairly static at the highest runtime. This is due to the bad channel performance in low E_b/N_0 values, and the decoding need to be executed at maximum of 10 iterations. Similar with this case, the runtime of CPU platform and GPU platform was relatively stable when E_b/N_0 values are more than 5 dB. However, the runtimes

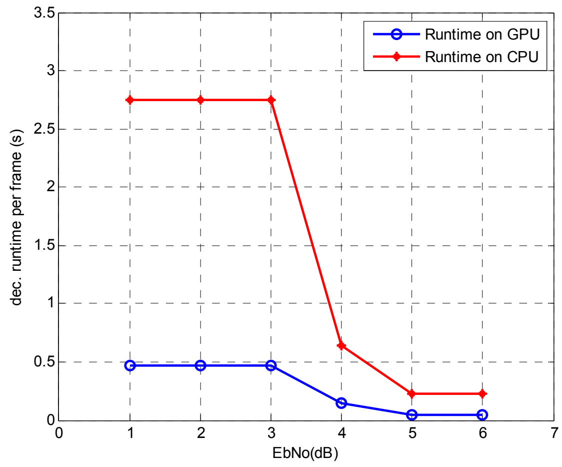


Fig. 8 Average runtime on GPU and CPU at a maximum of 10 iterations.

are the lowest, since the channel performance in this period is good, and the smallest iteration needed to be executed. Moreover, in period 3dB to 5dB the total runtime steadily decreases, since fewer decoding iterations have to be executed until a correct codeword is recovered. The decoding runtime in this scheme depends on the number of variable nodes (d_c) connect to each of check node, since the forward-backward algorithm is used in the check node processing which forward-backward messages are implemented sequentially (d_c -1) steps. Furthermore, a layered scheme introduces the decoding runtime is depend on the number of layers in decoding.

Compared to the millisecond decoding runtime reached by the GPU-based binary LDPC decoders reported in previous work, the decoding runtime measured in this experiment justifies the following complexity analysis. The complexity of check node processing in a binary decoder is $O(d_c)$, while the one in a nonbinary decoder is $O(d_c \times q^2)$. Additionally, the nonbinary arithmetic uses 4 table look-up operations for computations and conversions between exponential and decimal represents to find out indexes before the min and max operations are processed.

5. Conclusions

This paper presents a novel MMMA and an efficient implementation of parallel block-layered NB-LDPC decoding on a GPU. Due to its inherently massive parallelism, NB-LDPC decoding is more suited to GPU implementation than binary LDPC codes. The proposed MMMA provides better BER and FER performance than previous MMA algorithms. The experimental results show that GPU-based implementation of the proposed NB-LDPC decoder provides faster runtime and higher coding gain under a low 10^{-10} BER and a low 10^{-7} FER compared to a

CPU-based implementation.

Acknowledgments

This research was supported by Basic Science Research Program through the NRF funded by the Ministry of Science, ICT and Future Planning (2013R1A2A2A01068628).

References

[1] R. G. Gallager, "Low density parity check codes," IRE Trans. on Information Theory, vol. 8, no. 1, pp. 21-28, 1962.

[2] C. D. Matthew, and D. MacKay, "Low-Density Parity Check Codes over GF(q)," IEEE Communications Letters, vol. 2, no. 6, pp. 165-167, Jun. 1998.

[3] B. Zhou, J. Kang, S. Song, S. Lin, K. A. Ghaffar, and M. Xu, "Construction of non-binary Quasic-cyclic LDPC codes by arrays and array dispersions," IEEE Trans. on Communications, vol. 57, no. 6, pp. 1652-1662, Jun. 2009.

[4] V. Savin, "Min-Max decoding for nonbinary LDPC codes," In Proc. IEEE. Int. Symp. Inf. Theory, Toronto Canada, pp. 960-964, Jul. 2008.

[5] C. S. Choi and H. Lee, "A Block-Layered Decoder Architecture for Quasi-Cyclic Non-Binary LDPC codes," Journal of Signal Processing Systems, 2014. (online publication).

[6] F. Cai, "Low-Complexity decoding algorithms and architectures for non-binary LDPC codes," PhD. dissertation, Dept. Elect. Eng., Case Western Reserve University, Cleveland, United States, Aug. 2013.

[7] D. Declercq, M. Fossorier, "Decoding Algorithms for Nonbinary LDPC Codes Over GF(q)," IEEE Trans. on communications, vol.55, no.4, pp. 633-643, April. 2007.

[8] NVIDIA Corporation, "CUDA C Programming Guide version 5.5." NVIDIA Corporation, http://www.nividia.com, accessed May. 2013.

[9] G. Falcao, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," IEEE Trans. on Parallel and Distributed Systems, vol. 22, no. 2, p. 309-322, Feb. 2011.

[10] G. Falcao, J. Andrade, V. Silva, and L. Sousa, "Real-time DVB-S2 LDPC decoding on many-core GPU accelerators," in Proceedings of the International Conference on Acoustics, Speech and Signal Processing (ICASSP), Prague, May 22-27, 2011, pp. 1685-1688.

[11] J. Andrade, G. Falcao, and V. Silva, "FFT-SPA Non-binary LDPC decoding on GPU," IEEE International Conference on Speech and Signal Processing, Vancouver, BC, May 26-31, 2013, pp. 5099-5103.

[12] B. Legal, C. Jego and J. Crenne, "A high throughput efficient approach for decoding LDPC codes onto GPU devices," IEEE Embedded Systems Letters, vol. 6, no. 2, pp. 29-32, Jun. 2014.

[13] X. Zhang, and F. Cai, "Efficient Partial-Parallel Decoder Architecture for Quasi-Cyclic Nonbinary LDPC Codes," IEEE Trans. on Circuits and Systems I, vol. 58, no. 2, pp. 402-414, Feb. 2011.

[14] Y. L. Ueng, C. Y. Leong, C. J. Yang, C. C. Cheng, K. H. Liao, and S.W. Chen, "An efficient Layered Decoding Architecture for Nonbinary QC-LDPC Codes," IEEE Trans. on Circuits and Systems-I: Regular papers, vol. 59, no. 2, pp. 385-398, Feb. 2012.

[15] X. Chen, V. Akella, "Efficient Configurable Decoder Architecture for Nonbinary Quasi-Cyclic LDPC Codes," IEEE Trans. on circuits and systems-I: Regular papers, vol. 59, no. 1, pp. 188-197, Jan. 2012.

[16] F. Cai, and X. Zhang, "Relaxed Min-Max Decoder Architectures for Nonbinary Low-Density Parity-Check Codes," IEEE Trans. on very large scale integration systems, vol. 21, no. 11, pp. 2010-2023, Nov. 2013.

[17] D. B. Kirk, W. W. Hwu, "Programming Massively Parallel Processors: A Hands-on Approach", Morgan Kaufmann, 30 Corporate Drive, Suite 400, Burlington, MA 01803, USA, 2010.