Parallel Nonbinary LDPC Decoding on GPU

Guohui Wang, Hao Shen, Bei Yin, Michael Wu, Yang Sun, and Joseph R. Cavallaro
Department of Electrical and Computer Engineering
Rice University, Houston, Texas 77005
Email: {wgh, hs9, by2, mbw2, ysun, cavallar}@rice.edu

***Abstract*—Nonbinary Low-Density Parity-Check (LDPC) codes are a class of error-correcting codes constructed over the Galois field $GF(q)$ for $q > 2$. As extensions of binary LDPC codes, nonbinary LDPC codes can provide better error-correcting performance when the code length is short or moderate, but at a cost of higher decoding complexity. This paper proposes a massively parallel implementation of a nonbinary LDPC decoding accelerator based on a graphics processing unit (GPU) to achieve both great flexibility and scalability. The implementation maps the Min-Max decoding algorithm to GPU's massively parallel architecture. We highlight the methodology to partition the decoding task to a heterogeneous platform consisting of the CPU and GPU. The experimental results show that our GPU-based implementation can achieve high throughput while still providing great flexibility and scalability.**

***Index Terms*—GPU, OpenCL, nonbinary LDPC, error correcting codes, parallel architecture.**

I. Introduction

Binary Low-Density Parity-Check (LDPC) codes have been proven to approach the Shannon limit performance for very long code lengths [1]. It is shown that nonbinary LDPC codes constructed over the Galois field $GF(q)$ $(q > 2)$ can improve the performance for short and moderate code lengths [2].

However, the performance gain of nonbinary LDPC codes is achieved at the expense of an increase in decoding complexity. Since the introduction of nonbinary LDPC codes, many efforts have been made to improve the nonbinary LDPC decoding performance. On one hand, many researchers are looking for encoding solutions to construct nonbinary LDPC codes with some good properties. On the other hand, many decoding algorithms and architectures have been proposed to reduce the complexity of nonbinary LDPC decoding algorithms [3, 4]. However, these implementations are usually designed for a specific code type or for a fixed codeword length, so they suffer from poor flexibility and scalability.

The demand for new codes and novel low-complexity decoding algorithms for nonbinary LDPC codes requires a huge amount of extensive simulations. The high complexity of nonbinary LDPC decoding algorithms indicates that the CPU-based simulation will be extremely slow in higher order $GF(q)$ fields, especially when people study the error floor property of the codes. A graphics processing unit (GPU) can provide massively parallel computation threads with a many-core architecture, which can accelerate the simulations of the LDPC decoding over $GF(q)$. Many GPU-based implementations have been proposed for binary LDPC decoding [5, 6]. However, due to the drastically increased complexity of the decoding algorithms at higher order fields, the implementation of nonbinary LDPC decoding on GPU is still very challenging.

In this paper, we present a GPU implementation of a nonbinary LDPC decoder. This paper is organized as follows. In section II, we briefly review the decoding algorithms for nonbinary LDPC codes. Section III introduces the OpenCL programming model. Then, the details concerning our parallel implementation of a nonbinary LDPC decoder are described in Section IV. Section V shows experimental results and Section VI concludes the paper.

II. Nonbinary LDPC Decoding Algorithms

A. Nonbinary LDPC Codes and Decoding Algorithm Review

A nonbinary LDPC code can be defined by a parity-check matrix $\mathbf{H}$, which is a q-ary sparse matrix with M rows and N columns, whose elements are defined in the Galois field consisting of q elements ($GF(q) = \{0, 1, \ldots, q-1\}$). Matrix $\mathbf{H}$ can be represented by a Tanner graph. Each row in $\mathbf{H}$ corresponds to a check node in the Tanner graph, and each column in $\mathbf{H}$ corresponds to a variable node in the Tanner graph. Let $M(n)$ denote the set of check nodes connected to variable node n. Let $N(m)$ denote the set of variable nodes connected to check node m. The row weight for a check node is denoted by d_c.

The belief propagation (BP) decoding algorithm can be extended to the $GF(q)$ field to decode nonbinary LDPC codes [2]. To reduce complexity, approximate algorithms have been proposed such as the extended min-sum (EMS) algorithm, the Min-Max algorithm [3, 7, 8] and iterative soft (hard) reliability-based majority logic decodable (ISRB-MLGD (IHRB-MLGD)) algorithms [4]. Among these algorithms, the EMS algorithm and the Min-Max algorithm have similar BER performance, but the Min-Max algorithm has lower complexity. The ISRB-MLGD (IHRB-MLGD) algorithm significantly simplifies the check node processing so it is efficient for VLSI implementations. However, the ISRB-MLGD and IHRB-MLGD algorithms suffer from BER performance loss. Therefore, taking into account the error-correcting performance and the decoding complexity, the Min-Max algorithm is the best choice for a GPU implementation. Moreover, we can easily extend the Min-Max computation kernel to support other algorithms such as the EMS algorithm, so the decoder also has great flexibility.

Algorithm 1 Min-Max decoding algorithm [3]

Initialization:

$L_n(a) = \ln(\Pr(c_n = s_n|channel)/\Pr(c_n = a|channel))$;
$Q_{m,n}(a) = L_n(a)$;

Iterations:

Check node processing

$R_{m,n}(a) = \min_{(a_{n'})_{n'\in N(m)}\in\mathbf{\Lambda}(a)} (\max_{n'\in N(m)\setminus\{n\}} Q_{m,n'})(a_{n'}))$;
$\mathbf{\Lambda}(a) \equiv \{a_{n'}|h_{mn}a + \sum_{n'\in N(m)\setminus\{n\}} h_{mn'}a_{n'} = 0\}$;

Variable node processing

$Q'_{m,n}(a) = L_n(a) + \sum_{m'\in M(n)\setminus\{m\}} R_{m',n}(a)$;
$Q'_{mn} = \min_{a\in GF(q)} Q'_{m,n}(a)$;
$Q_{m,n}(a) = Q'_{m,n}(a) - Q'_{m,n}$;

Tentative decoding:

$\widetilde{L}_n(a) = L_n(a) + \sum_{m\in M(n)} R_{m,n}(a)$;
$c_n = \arg\min_{a\in GF(q)}(\widetilde{L}_n(a))$.

If the check equation is satisfied or the max iteration number is reached, terminate the decoding; Otherwise, go back to iterations.

B. The Min-Max Decoding Algorithm

Let us first review the Min-Max decoding algorithm [3]. Denote $L_n(a)$ and $\widetilde{L}_n(a)$ as the *a priori* information and the *a posteriori* information of variable node n concerning a symbol a in $GF(q)$, respectively. Let $R_{m,n}(a)$ and $Q_{m,n}(a)$ denote the check node message and variable node message concerning a symbol a. Assume that x_n is the n-th symbol in a received codeword and s_n is the most likely symbol for x_n. The Min-Max algorithm is shown in Algorithm 1. The check node processing contains most of the computations of the Min-Max algorithm, which has a complexity of $\mathcal{O}(d_c \cdot q^2)$ for each check node. As is shown in Fig 1, $L_n(a)$ and $\widetilde{L}_n(a)$ can be represented by 2-D $a \times n$ arrays; $R_{m,n}(a)$ and $Q_{m,n}(a)$ can be represented by 3-D $a \times m \times n$ arrays. Due to the special 2-D and 3-D structure, the way to arrange these arrays in the memory will significantly affect performance. We will further discuss this in Section IV-E.

III. The OpenCL Programming Model

The goal of this work is to implement a highly parallel and flexible decoder that supports different code types, various code lengths and can run on different devices such as CPUs and GPUs. Therefore, we choose the Open Compute Language (OpenCL) programming model to implement a parallel nonbinary LDPC decoder. The OpenCL model is widely used to program heterogeneous platforms consisting of CPUs, GPUs and other devices [9]. For a massively parallel program developed for a GPU, data-parallel processing is exploited with the OpenCL by executing in parallel threads. The OpenCL model employs the Single Instruction Multiple Threads (SIMT) programming model. If a task is executed several times independently over different data, it can be mapped into a kernel, and executed in parallel on many threads.

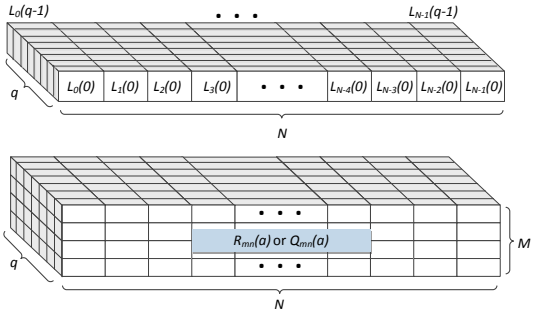


Fig. 1. Data structure of L_n, $\widetilde{L}_n(a)$, $R_{m,n}(a)$ and $Q_{m,n}(a)$.

The execution of a kernel on a GPU is distributed according to a grid of work groups with adjustable dimensions. The number of work items per work group has to be programmed according to the number of registers available on the GPU, in order to guarantee that enough registers and local memories are allocated to each thread at compile time. All work items inside the same work group can share data through a shared local memory mechanism. Synchronizations across work items in a work group are necessary to guarantee the correctness of the parallel accesses to a shared local memory.

IV. Parallel Implementation of Nonbinary LDPC Decoder and Design Space Exploration

A. Complexity Analysis of Nonbinary LDPC Decoding

Given the properties of the algorithm, a GPU-based heterogeneous platform is very suitable to implement nonbinary LDPC decoding algorithms. To decode a binary LDPC code, more than hundreds of codewords are usually decoded simultaneously to fully utilize the GPU's computation resources to push the limit of decoding throughput [5, 6]. However, multi-codeword decoding suffers from long latency which prevent the GPU implementation from real-time applications. Being extended to higher $GF(q)$ fields, the computation kernels of nonbinary LDPC codes become more complex compared to the ones in the binary case ($\mathcal{O}(d_c \cdot q^2)$ vs. $\mathcal{O}(d_c)$ for check node processing; assume d_c is the number of non-zero elements connected to a check node in matrix H). The Min-Max kernel is more capable of providing enough computations to keep all the compute units busy. Moreover, the nonbinary LDPC decoding algorithm has a higher computation to memory access ratio. The higher this ratio is, the less time overhead is spent on the data transfer. These features make the nonbinary LDPC decoding algorithms a good candidate for a GPU implementation.

TABLE I
Breakdown of run time of the Min-Max algorithm on CPU.

Block name	Time	Percentage
Init LLR	0.353 ms	0.08%
CNP	431.336 ms	91.64%
VNP	30.462 ms	6.43%
Tentative dec	0.876 ms	1.86%

We measure the run time of major blocks in the Min-Max algorithm by running a serial C reference code on a CPU. The

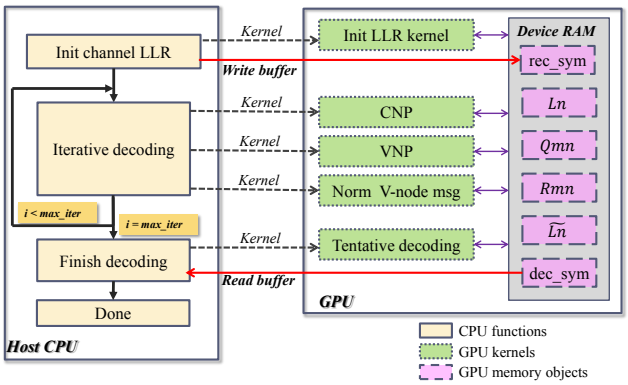


Fig. 2. Kernel partitioning and program flow.

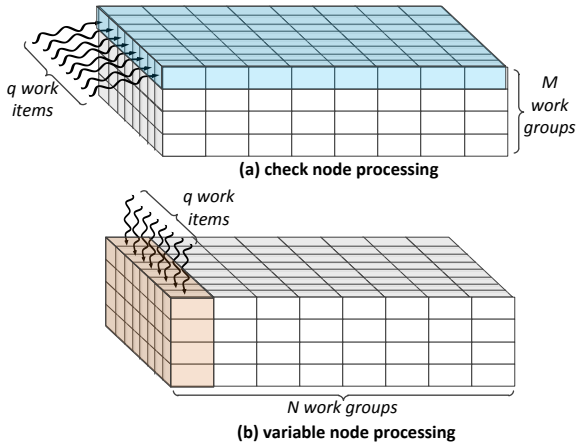


Fig. 3. Mapping CNP and VNP to OpenCL kernels.

profiling results in Table I show that check node processing (CNP) and variable node processing (VNP) occupy most of the processing time (91.64% and 6.43%, respectively).

B. Mapping Algorithm onto Parallel Architecture

By carefully analyzing the Min-Max decoding algorithm, we develop a work flow of the iterative decoding process, as is shown in Figure 2, including the CPU-GPU task partitioning, kernel launching and memory transfer operations. The main program is running on a host CPU, which handles the OpenCL context initialization, the kernel scheduling and synchronization, the control of decoding iterations, memory management and so on. To reduce the memory transfer overhead between CPU and GPU, we put most of the computations on the GPU and keep all the intermediate messages in the device memory. Therefore, we only need two memory transfers: one is to transfer the received symbol data into the device RAM of GPU in the beginning, and the other is to get the decoded symbols back at the end of the decoding process.

It is worth mentioning that the proposed work flow represents a general architecture which can be used to implement different nonbinary LDPC decoding algorithms, including the Min-Max algorithm. Only some small changes are needed in the CNP kernel to support other algorithms.

We partition the decoding algorithm into five OpenCL kernels, which are listed in Table II. The method to distribute the tasks into work groups and to fully explore the parallelism is important for a high performance OpenCL implementation.

TABLE II
MAPPING KEY ALGORITHM BLOCKS ONTO OPENCL KERNEL.

Function	# of workgroups	# work items per group	Total # of work items
Init LLR	N	q	$N \times q$
CNP	M	q	$M \times q$
VNP	N	q	$N \times q$
Norm VNP msg	N	dc	$N \times dc$
Tentative dec	$\lceil M/32 \rceil$	32	M

As examples, Figure 3 shows the details of mapping CNP and VNP kernels onto GPU's parallel architecture. Since all the messages are vectors in the nonbinary field, we can spawn q work items ("thread" in CUDA) per work group ("thread block" in CUDA) to compute each CNP or VNP message, so that these q work items can have exactly the same computation path and memory access pattern. This could help the compiler combine operations for work items into SIMT instructions to take advantage of GPU's parallel architecture.

To further increase the parallelism in a work group to fully utilize a compute unit, we can keep the total number of work items unchanged, but assign the work groups in a different way. For example, to launch a CNP kernel, we still spawn $M \times q$ work items in total. But we can assign $C \cdot q$ work items for each work group and use $\lceil M/C \rceil$ work groups (assume C is a chosen integer).

C. GPU Implementation of Nonbinary Arithmetic

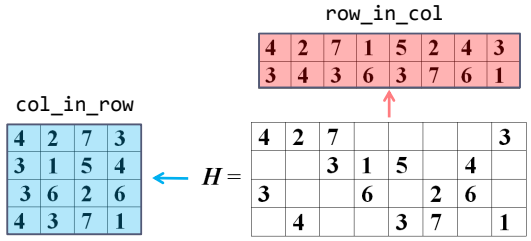
Nonbinary arithmetic is necessary for a nonbinary LDPC decoder. Addition and subtraction in $GF(q)$ can be implemented by XOR operations. However, multiplication and division are non-trivial to implement. We can use look-up tables (LUT) to implement the nonbinary multiplication and division. Two LUTs holding expq and logq values are used, each of which contains q elements. For instance, a multiplication can be calculated as follows: $a \times b = \mathrm{expq}[\mathrm{logq}[a] + \mathrm{logq}[b])\%(q-1)]$. The division can be computed in a similar way: $a/b = \mathrm{expq}[(\mathrm{logq}[a] - \mathrm{logq}[b] + q - 1)\%(q-1)]$. Since these two LUTs are used frequently by all the work items, we propose to put them into the shared local memory to enable quick memory access. Since q ranges from 2 to 256, the LUTs can be easily fit into local memory and can be efficiently loaded into the local memory in parallel.

D. Efficient Data Structures

Since matrix H of an LDPC code is sparse, we can reduce the storage requirement and enable fast memory access by using compressed representations shown in Figure 4. Similar to the method used in our previous work, horizontal and vertical compression of matrix H generates very efficient data structures [6]. By utilizing the vector data type in OpenCL as is shown in the figure, we can further increase the efficiency of the compressed representations. Since $R_{mn}(a)$ and $Q_{mn}(a)$ messages also have sparse structures, they can also be compressed in a similar way.

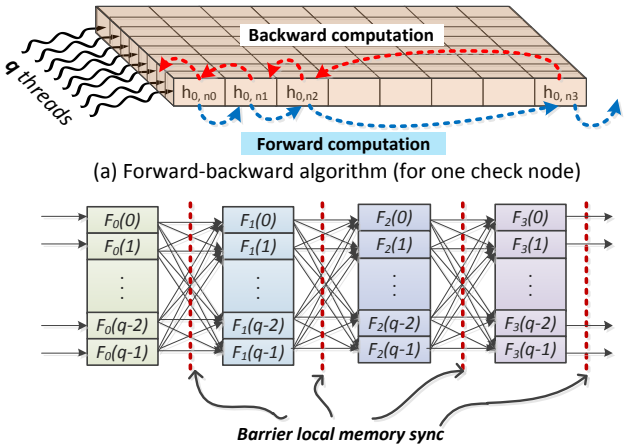
E. Accelerate The Forward-backward Algorithm in CNP

The original Min-Max algorithm has a complexity of $\mathcal{O}(q^{d_c})$ in the check node processing (CNP). As is shown




```
cl_short2 col_in_row[M][dv];
        col_in_row[row][index].s[0]= col number;
        col_in_row[row][index].s[1]= H[row][col];
cl_short2 row_in_col[N][dc];
        row_in_col [col][index].s[0]= row number;
        row_in_col [col][index].s[1]= H[row][col];
```

Fig. 4. Efficient representation of H matrix.



(a) Forward-backward algorithm (for one check node)

(b) Use local memory to speedup the FBA algorithm

Fig. 5. Forward-backward algorithm and GPU implementation.

in Algorithm 2, a forward-backward algorithm (FBA) is able to reduce the complexity to $\mathcal{O}(d_c \cdot q^2)$ [3]. Let $N(m) = \{n_0, n_1, \cdots, n_{(dc-1)}\}$ be the set of variable nodes connected to check node m.

Algorithm 2 Forward-backward Algorithm (FBA) [3]

For check node m, compute forward metrics

$F_0(a) = Q_{m,n_0}(h^{-1}_{m,n_0} a)$;

$F_i(a) = \min_{a'+h_{m,n_i}\cdot a''=a} (\max(F_{i-1}(a'),\ Q_{m,n_i}(a'')))$;

Backward metrics

$B_{(dc-1)}(a) = Q_{m,n_{(dc-1)}}(h^{-1}_{m,n_{(dc-1)}} a)$;

$B_i(a) = \min_{a'+h_{m,n_i}\cdot a''=a} (\max(B_{i+1}(a'),\ Q_{m,n_i}(a'')))$;

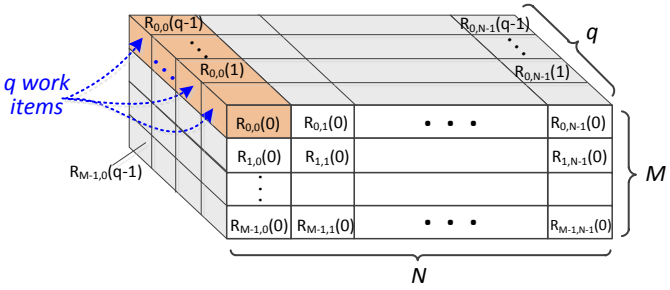
Check node message computation

$R_{m,n_0}(a) = B_1(h_{m,n_0} a)$;

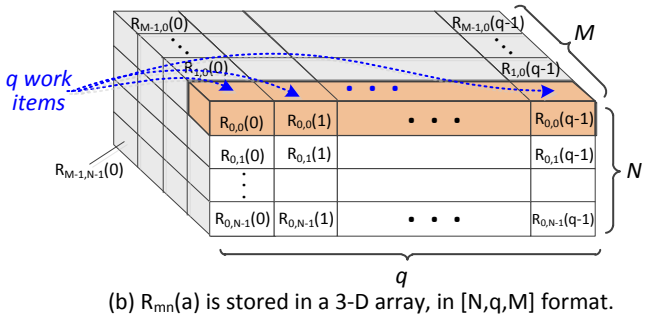
$R_{m,n_{(dc-1)}}(a) = F_{(dc-2)}(h_{m,n_{(dc-1)}} a)$;

$R_{m,n_i}(a) = \min_{a'+a''=a} (\max(F_{i-1}(h_{m,n_{i-1}} a'), B_{i+1}(h_{m,n_{i+1}} a'')))$.

Figure 5-(a) shows how the forward-backward algorithm is mapped to a OpenCL implementation. Computations for one check node are shown. All the q work items in a work group still follow the same execution path, which is very efficient for GPU implementation. Figure 5-(b) shows details of a trellis structure of the forward computing steps. The forward messages $F_i(a)$ (for $a \in GF(q)$ and $i = 0, 1, \cdots, d_c - 1$) at stage i always need to read $F_{i-1}(a)$ messages from stage $i-1$ after interleaving them. This interleaving operation causes significant performance degradation due to the access conflicts in the global memory. To solve this problem, we propose to put the forward messages $F_i(a)$ into on-chip local memory of GPU. Since the trellis in Figure 5-(b) is traversed by work items in the same work group, $F_i(a)$ can be shared by all work items. At first, all q work items work in parallel to initialize $F_0(a)$ and then update $F_i(a)$ in parallel for each stage i. A local memory barrier synchronization operation **barrier(CLK_LOCAL_MEM_FENCE)** is performed after each stage to keep work items synchronized. The backward messages can be computed in a similar way. The amount of local memory required for $F_i(a)$ and $B_i(a)$ is $2 \cdot sizeof(cl\_float) \cdot q \cdot d_c$. For example, 1.5KB of local memory is required for a $(3, 6)$-regular $GF(32)$ code. Experimental results show that this optimization gives us $2\times$ speedup.



(a) $R_{mn}(a)$ is stored in a 3-D array, in [M, N, q] format.



(b) $R_{mn}(a)$ is stored in a 3-D array, in [N,q,M] format.

Fig. 6. Coalescing memory access by adjusting the data structure.

F. Coalescing Global Memory Accesses

The data structures stored in the global memory should be carefully designed due to the long latency of global memory accesses. If global memory accesses can not be grouped in a coalesced way by the compiler, they will be compiled into multiple instructions and will significantly hurt the overall performance.

$R_{mn}(a)$ and $Q_{mn}(a)$ are stored in the global memory and they have complicated 3-D structures. Figure 6 shows two alternative ways to store $R_{mn}(a)$. If we define a 3-D array A in a format $[A, B, C]$, then an entry $[x, y, z]$ of A can be accessed as $A[z \cdot A \cdot B + y \cdot A + x]$. The original format of $[M, N, q]$ shown in 6-(a) is the most straightforward way to arrange a 3-D array of $R_{mn}(a)$. However, for q work items to access $\{R_{0,0}(0), R_{0,0}(1), \cdots, R_{0,0}(q-1)\}$ in parallel, the data are loaded from memory locations far away from each other. This results in multiple memory access instructions and

increases the memory access time. In contrast, if we arrange $R_{mn}(a)$ in the format of $[N, q, M]$ depicted in Figure 6-(b), q work items always access $R_{mn}(a)$ data stored contiguously. By doing so, coalesced memory access is enabled, and we observed a $4 \sim 5\times$ speedup in our experiments.

V. Experimental Results

A. Experimental Setup

We implemented the proposed architecture using OpenCL. This implementation is flexible and can be easily configured by adjusting parameters to support different code types, code lengths, and various devices such as CPUs and GPUs. The implementation is evaluated on two CPU platforms: an Intel i7-640LM dual-core CPU running at 2.93GHz and an AMD Phenom II X4-940 quad-core CPU running at 2.9GHz. We also ran our experiments on an NVIDIA GTX470 GPU with 448 stream processors running at 1.215GHz and with 1280MB of GDDR5 device memory. The corresponding OpenCL SDK is installed for each platform. We use a $1/2$ $(620, 310)$ $(3, 6)$-regular GF(32) LDPC code, which is widely used in related research and shows good error-correcting performance [7]. OpenCL events and functions such as ***clFinish()***, ***clWaitForEvents()***, and ***clGetEventProfilingInfo()*** are used to measure the run time.

B. Experimental Results and Discussion

Experimental results are shown in Table III. On the i7 CPU, the OpenCL implementation shows a $2.47\times$ speedup compared to a serial C reference program. On the AMD X4-940 CPU, the OpenCL accelerated version results in a $6.67\times$ speedup over the C reference code. When executing on the NVIDIA GTX-470 GPU, we achieve 693.5 Kbps throughput for 10 iterations; if early termination is enabled, the throughput can be further improved to 1260 Kbps.

Compared to the 50∼100 Mbps throughput achieved by the GPU-based binary LDPC decoders reported in previous work, the throughput measured in this experiment justifies the following complexity analysis. The complexity of the CNP in a binary case is $\mathcal{O}(d_c)$, while the one in a nonbinary case is $\mathcal{O}(d_c \cdot q^2)$. Additionally, the nonbinary arithmetic uses $2 \sim 3$ computations and 3 table look-up operations, which again adds at least $2 \sim 3\times$ non-trivial overhead. The overall time complexity of the nonbinary decoder can be estimated to be $2q^2 \sim 3q^2$ higher than the binary case. Taking the code used in this paper as an example, q is equal to 32, so we can expect a $2000 \sim 3000\times$ increase in complexity when comparing a nonbinary decoder to a binary one. However, thanks to the massive parallelism in the decoding algorithm and our optimizations, the gap between the binary and the nonbinary implementation is reduced to around $50\times$.

It is worth mentioning that we choose a short code $(620, 310)$ on purpose since the nonbinary codes show performance gain over the binary codes for short codewords. Researcher are more interested in short codewords and higher GF fields, so accelerating the decoders supporting short codes and high GF fields such as $GF(32)$ are of great importance for nonbinary LDPC codes research [2, 3, 7]. However, we can always achieve higher throughput by decoding a longer code in lower GF fields such as $GF(4)$, $GF(8)$ and $GF(16)$ with a small row weight d_c, if a higher throughput is the goal. For example, the projected throughput of a (8000, 4000) $GF(16)$ code is around 20 Mbps (@10 iterations) on a GTX-470 GPU based on the complexity analysis and our results in Table III.

TABLE III
Experimental results. (Max # of iterations=10).

Processor	Program	Time	Throughput
Intel i7-640LM (Intel OpenCL SDK 2012)	Serial C code	410.5 ms	7.55 Kbps
	OpenCL	172 ms	18.7 Kbps
AMD X4-940 (AMD APP SDK v2.7)	Serial C code	563 ms	5.46 Kbps
	OpenCL	82.3 ms	36.57 Kbps
NVIDIA GTX-470 (NVIDIA SDK v4.2)	OpenCL	4.47 ms	693.5 Kbps
	OpenCL	2.46 ms	1260 Kbps*

* GPU implementation with early termination, $E_bN_0 = 3.0$, SER=2.1×10^{-5}, FER=3.3×10^{-3} (SER: symbol error rate; FER: frame error rate).

VI. Conclusion

This paper presents a novel parallel implementation of nonbinary LDPC decoder on GPU. Due to its inherently massive parallelism, a nonbinary LDPC decoder is more suitable for a GPU implementation than for binary LDPC codes. We demonstrate our method to take full advantage of the GPU's compute power to accelerate the nonbinary LDPC decoding algorithms. The experimental results show that the proposed GPU-based implementation of the nonbinary LDPC decoder can achieve great performance, flexibility, and scalability.

ACKNOWLEDGMENTS

This work was supported by the US National Science Foundation under grants EECS-1232274, EECS-0925942 and CNS-0923479.

References

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

[2] M. Davey and D. MacKay, "Low-density parity check codes over GF(q)," *IEEE Communications Letters*, vol. 2, no. 6, pp. 165 – 167, June 1998.

[3] V. Savin, "Min-Max decoding for non binary LDPC codes," in *IEEE International Symposium on Information Theory, 2008.*, July 2008, pp. 960 –964.

[4] X. Zhang, F. Cai, and S. Lin, "Low-complexity reliability-based message-passing decoder architectures for non-binary LDPC codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 11, pp. 1938 –1950, Nov. 2012.

[5] G. Falcao, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 2, pp. 309 –322, 2011.

[6] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "GPGPU accelerated scalable parallel decoding of LDPC codes," in *the 45th Asilomar Conference on Signals, Systems and Computers*, Nov. 2011, pp. 2053 –2057.

[7] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient decoder design for nonbinary quasicyclic LDPC codes," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 57, no. 5, pp. 1071 –1082, May 2010.

[8] 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 Transactions on Circuits and Systems I*, vol. 59, no. 2, pp. 385 –398, Feb. 2012.

[9] Khronos OpenCL Working Group, *The OpenCL Specification*. [Online]. Available: http://www.khronos.org/opencl