

Extremely Fast Simulator for Decoding LDPC Codes

S. F. Yau, T. L. Wong and F. C. M. Lau*

Department of Electronic and Information Engineering, Hong Kong Polytechnic University, Hong Kong

*Email: encmlau@polyu.edu.hk

Abstract—Decoding low-density parity-check (LDPC) codes requires a lot of computation time, particularly when bit error rates as low as 10^{-9} are needed. In this paper, we improve the simulation speed by making use of an inexpensive graphics processing unit (GPU). A dedicated program is written to utilize the hardware resources in the GPU to decode LDPC codes in a parallel manner. Codes with rate 1/2 and length 2,304 and 10,008 are simulated by both GPU and central processing unit (CPU). We also show the average iteration time when LDPC codes with length 15,000 and 20,000 are simulated.

Index Terms—Fast decoding, GPU computing, LDPC codes.

I. INTRODUCTION

Low-density parity-check (LDPC) codes form a type of forward error correction (FEC) codes with performance approaching the Shannon limits as the codeword length increases [1]. In a LDPC code, a set of bit nodes (also called variable nodes) is sparsely connected to a set of check nodes [2], [3], [4]. In other words, each bit node (BN) is usually connected to a few check nodes (CNs) although there may be thousands of BNs or CNs. Message passing algorithms are commonly used for the decoding of LDPC codes [5]. In these algorithms, messages are passed iteratively back and forth between the set of BNs and the set of CNs. The iterative process will stop when a valid codeword is found or when the maximum number of iterations is reached.

A well-designed LDPC code can produce a bit error rate (BER) of 10^{-9} at a reasonably small bit-energy-to-noise-power-spectral-density ratio (denoted by E_b/N_0). However, the simulation of such good codes will require a vast amount of computation time. For example, an over-the-counter computer can easily spend a few days to perform the simulations for a moderate length LDPC code (say 10,000 bits/codeword) before arriving at a BER of 10^{-9} . The simulation time can be shortened substantially if a supercomputer is used. Yet, the cost of a supercomputer is too high and not many research units/institutions can afford it. Recently, relatively inexpensive Graphic Processing Units (GPUs) have been gaining popularity in carrying out computation intensive tasks. It is because GPUs, when suitably programmed, can perform computations in parallel, resulting in enormous speed improvements when compared with the time taken by a central processing unit (CPU) to perform the same computation task.

In this paper, we build an inexpensive computer with a GPU. A dedicated program is written to utilize the hardware resources in the GPU to decode LDPC codes in a parallel

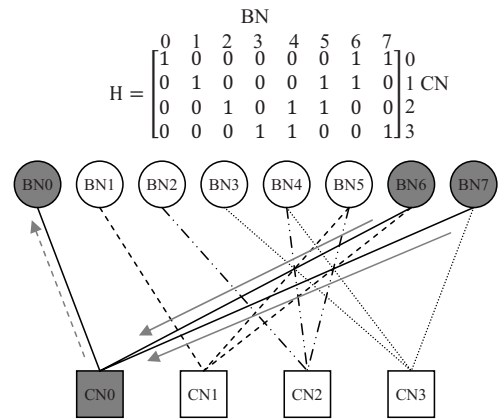


Fig. 1. A parity-check matrix and the corresponding bipartite graph.

manner. We compare the respective times taken by the GPU and CPU to decode LDPC codewords.

The organization of the paper is as follows. Section II reviews the LDPC codes and the belief propagation decoding algorithm. Section III describes the details of the proposed approach. In Section IV, we present our simulation results and Section V provides the conclusions.

II. LDPC CODES AND THE DECODING ALGORITHM

Low-density-parity-check codes are linear block codes. Associated with each LDPC code is a parity-check matrix (PCM) in which most of the entries are zeros and a few of the entries are ones. Each column corresponds to a bit node whereas each row corresponds to a check node. Moreover, the PCM can be represented by a bipartite graph (also called Tanner graph) consisting of bit nodes and check nodes. In the bipartite graph, a bit node is connected to a check node if the corresponding entry in the PCM is non-zero. Figure 1 shows an example of a PCM and its corresponding bipartite graph. In the PCM shown, both the row weights and the column weights are not regular. Moreover, the maximum row weight and maximum column weight are 3 and 2, respectively.

Algorithms based on belief propagation (BP) are widely used for decoding LDPC codes [6]. In the BP algorithm, the bit nodes and the check nodes exchange information iteratively. In each iteration, a bit node v evaluates the message sent to a neighboring check node c based on (i) the message from the channel and (ii) the incoming messages from all other

neighboring check nodes (all neighboring check nodes except c). Then, a check node c calculates the message sent to a bit node v based on the incoming messages from all other neighboring bit nodes. In Fig. 1, check node CN0 is connected to the bit nodes BN0, BN6 and BN7. Thus, the message from CN0 to BN0 is evaluated based on the incoming messages from BN6 and BN7.

We consider a LPDC code with N bit nodes and M check nodes. The transmitted signal corresponding to the i th code bit $\phi_i \in \{0, 1\}$ ($i = 1, 2, \dots, N$) is given by $(-1)^{\phi_i}$. Assuming an additive white Gaussian noise (AWGN) channel, the received signal is given by $y_i = (-1)^{\phi_i} + z_i$, where z_i is an independent and identically distributed zero-mean Gaussian random variable with variance (noise power) σ^2 .

For $b = 0, 1$, we define $q_{ij}(b)$ as the conditional posterior probability of the bit ϕ_i being equal to b ; and $r_{ji}(b)$ as the conditional posterior probability of the event that the j th check node being satisfied. Considering the bit node v_i , we denote its neighboring check-node set by C_i and the outgoing message to check node $c_{j'}$ by $q_{ij'}(b)$. Similarly, for the check node c_j , its neighboring bit-node set is denoted by V_j and the outgoing message to bit node $v_{i'}$ is given by $r_{ji'}(b)$. Assuming that the exchanged messages are independent random variables, the BP algorithm in the probability domain proceeds as follows.

- 1) Estimate the noise power σ^2 . For $i = 1, 2, \dots, N$ and $b = 0, 1$, initialize $P_i(b) := \Pr(\phi_i = b|y_i)$, where $\Pr(\phi_i = b|y_i)$ denotes the posterior probability that bit ϕ_i equals b given the received signal y_i . Set $q_{ij}(b) = P_i(b)$ if the i -th bit node v_i and the j -th check node c_j are connected.
- 2) Update $\{r_{ji}(b): j = 1, 2, \dots, M; i \in V_j \text{ and } b = 0, 1\}$ using

$$\begin{cases} r_{ji}(0) = \frac{1}{2} + \frac{1}{2} \prod_{i' \in V_j/i} (1 - 2q_{ji'}(1)) \\ r_{ji}(1) = 1 - r_{ji}(0). \end{cases} \quad (1)$$

- 3) Update $\{q_{ij}(b): i = 1, 2, \dots, N; j \in C_i \text{ and } b = 0, 1\}$ using

$$q_{ij}(b) = K_{ij} P_i(b) \prod_{j' \in C_i/j} r_{ji'}(b) \quad (2)$$

where K_{ij} is chosen to ensure that $q_{ij}(0) + q_{ij}(1) = 1$.

- 4) Compute the posteriori probability of bit $\phi_i = b$ for $b = 0, 1$ using

$$Q_i(b) = K_i P_i(b) \prod_{j \in C_i} r_{ji}(b) \quad (3)$$

where K_i is chosen to ensure that $Q_i(0) + Q_i(1) = 1$.

- 5) For $i = 1, 2, \dots, N$, set

$$Q_i(1) \stackrel{\hat{\phi}_i=1}{\geq} Q_i(0). \quad (4)$$

where $\hat{\phi}_i$ is the i -th estimated codeword bit by hard decision. If $\hat{\phi} \mathbf{H}^T = \mathbf{0}$ or the maximum number of iterations is reached, stop; else, go to Step 2.



Fig. 2. A seed pool and a random variable pool. T denotes the number of threads and “Size” represents the number of random variables generated.

III. PROPOSED APPROACH

A. Coalesced Read and Bank Conflict

The GPU operates most efficiently when data of the threads are read from consecutive address locations. The event is called a *coalesced read*. In other words, thread 1 reads data at address #1, thread 2 reads data at address #2, ..., and thread N reads data at address $\#N$, in which the address $\#k$ equals address $\#(k-1)$ plus the size of the memory at address $\#(k-1)$. If coalesced read is not used, the GPU may need to go to a remote memory location to access the next data. For example if thread 2 reads the data at the address #3 and thread 3 reads the data at the address $\#N$, it will more time for the GPU to locate the destinations. Also, if two threads try to access the same memory address at the same time, the “late” thread has to wait for the “early” thread to finish reading before it can access the data. Such an event is called a *bank conflict*.

B. Noise Generator

In the GPU, we implement an AWGN generator by using a few linear congruential generators (LCG) and Tausworthe generators, and the Box-Muller transformation. The LCG equation is expressed as

$$X_{n+1} = (aX_n + c) \mod m \quad (5)$$

where X_n is a random number; a, c and m are constants selected to produce a random sequence with a long period. By combining a few LCG and Tausworthe generators, we generate a 2^{127} -period sequence with 32-bit resolution. For each “combined” uniform random generator, four seeds are needed. Since the GPU can perform the instructions in a parallel way, we create a pool of seeds such that all threads in the GPU can use different sets of seeds to generate uniformly distributed random numbers and transform them into Gaussian distributed random variables at the same time. The seed pool and the random variable pool are illustrated in Fig. 2. Note that we deliberately put the data in consecutive memory addresses such that coalesced read can be executed. Moreover, the transformation from uniformly distributed random numbers to Gaussian distributed random variables is achieved by using the Box-Muller equation, which is given by

$$Z_0 = \sqrt{-2 \ln(\alpha)} \times \cos(2\pi\beta) \quad (6)$$

where α and β are uniformly distributed random numbers.

Table I shows the rate of generating random numbers (in M samples/s) for different number of threads used and different size of random numbers generated. For example, to generate 1 M samples using 8096 threads, the rate of generation is 12.59 M Sa/s.

TABLE I

RATE OF GENERATING RANDOM NUMBERS (IN M SAMPLES/S) FOR DIFFERENT NUMBER OF THREADS USED AND DIFFERENT SIZE OF RANDOM NUMBERS GENERATED. $K = 1024$ AND $M = 1024K$.

Number of threads	16K Sa	128K Sa	1M Sa	8M Sa
16	0.207	1.25	N/A	N/A
128	0.173	1.38	N/A	N/A
1024	0.165	1.56	11.52	58.00
8096	0.197	1.73	12.59	70.94
65536	N/A	1.65	13.27	71.00

C. Compressed Parity-Check Matrices

The parity-check matrix (PCM) corresponding to a LDPC code is usually quite large. It would therefore be very memory-consuming to save all the entries in the PCM. Utilizing the sparse nature of the PCMs, the authors save PCMs in a compressed form, forming compressed PCMs (CPCMs) [7]. The CPCMs are further defined in the Check Node Domain (CND) and in the Bit Node Domain (BND), forming the CPCM-CND and CPCM-BND, respectively.

The size of a CPCM-CND is based on two factors: the number of check nodes and the maximum weight of the rows (i.e., maximum degree of the check nodes). In a CPCM-CND, each entry in a particular row stores the BN number that is connected to the CN corresponding to that row. We use the PCM in Fig. 1 as an example. Since the number of rows is 4 and the maximum row weight equals 3, the CPCM-CND will be of size 4×3 and the entries are shown as in Fig. 3. The formation of a CPCM-BND is similar and the CPCM-BND corresponding to the PCM in Fig. 1 is shown in Fig. 4. Note that in this case, we have padded the matrix with -1 because the column weights are not identical. The padding indicates that there are no more valid entries (connections) in this row and facilitates the CPU in accessing the memory.

In addition, two message matrices (MMs) are proposed for storing the messages passing from check nodes to bit nodes, and vice versa. One message matrix has the same size of CPCM-BND and the other the same size as CPCM-CND. They are called MM-BND and MM-CND, respectively. In the LDPC decoding process, the updated messages need to be converted to the other domain after each half-iteration. The conversion matrix (CM) for BND-to-CND (B2C) and for CND-to-BND (C2B) are illustrated in Fig. 5.

Note that all the aforementioned matrices are designed for CPU computation. Since the memory requirement is significantly reduced, most of the data can be found in the cache and the “cache missing” events can be decreased substantially. Consequently, the simulation speed can be improved. The results in [7] have shown computation time reduced by up to 75% when using the above techniques. The improvement is also more obvious when the matrix size is larger.

For GPU computation, similar CPCM matrices, messages matrices and conversion matrices are required. However, the format of them are slightly different from those for use in CPU computation such that we can make use of the coalesced read. Fig. 6, Fig. 7 and Fig. 8 illustrate the CPCM-CND, CPCM-

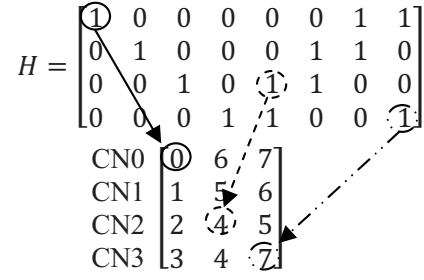


Fig. 3. Formation of the compressed parity-check matrix in the check-node domain (CPCM-CND) for use in CPU computation.

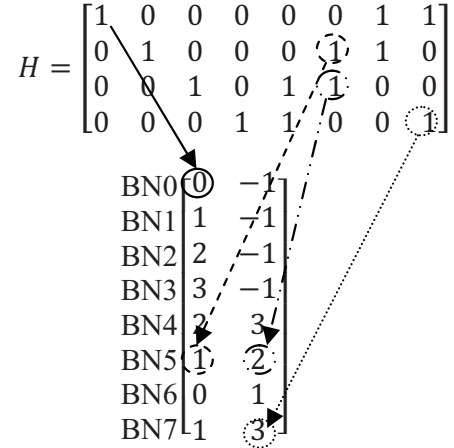


Fig. 4. Formation of the compressed parity-check matrix in the bit-node domain (CPCM-BND) for use in CPU computation.

BND and the conversion matrices for use in GPU computation, respectively.

D. Parallel Decoding

When LDPC decoding is performed by a CPU, nodes are processed one after another. Other nodes cannot be processed until the CPU has completed its computations for the current node. The flow of the LDPC decoding process in a CPU is shown in Fig. 9. On the other hand, GPU provides a platform on which nodes belonging to the same type (bit nodes or check nodes) can be processed simultaneously. The key concept of the proposed GPU LDPC decoder is “one thread one node”. In other words, the number of threads assigned for LDPC decoding is the same as the number of nodes and each thread deals with its corresponding edges. By re-packing the messages as described in the previous section, coalesced read can be fully utilized to accelerate the decoding process. Nonetheless, the second half-iteration cannot started until the first half-iteration is finished, and vice versa. The flow of the LDPC decoding process in a GPU is shown in Fig. 10.

IV. RESULTS AND DISCUSSIONS

To demonstrate the performance of the proposed method, we decode LDPC codes and record the number of iterations

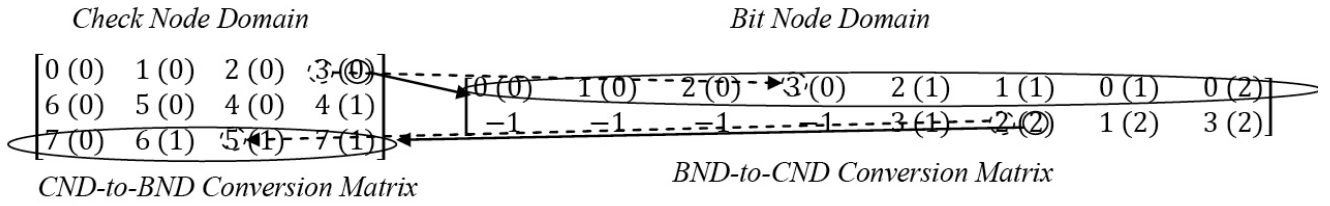


Fig. 8. Formation of the CND-to-BND conversion matrix and the BND-to-CND conversion matrix for use in GPU computation.

TABLE II
TYPES OF RATE-0.5 LDPC CODES USED IN THE SIMULATIONS AND THE AVERAGE TIME CONSUMED PER ITERATION.

Code	Type	Size	No. of edges	Time per iteration by GPU	Time per iteration by CPU
I	Irregular	$10,008 \times 5,004$	37,214	$61\mu s$ (0.27)	$229\mu s$ (1.00)
II	Regular	$10^5 \times 5 \cdot 10^4$	$3 \cdot 10^5$	$170\mu s$ (0.14)	$1191\mu s$ (1.00)

TABLE III
QC-LDPC CODES USED IN THE SIMULATIONS AND THE RESULTS.

Code length	Column weight	Row weight	No. of blocks simulated	Total time	Time per iteration	Decoding rate
15,000	4	24	31,930,000	201,534s	$0.21\mu s$	158.4 blocks/s
18,000	4	24	225,670,000	1,445,957s	$0.22\mu s$	156.1 blocks/s

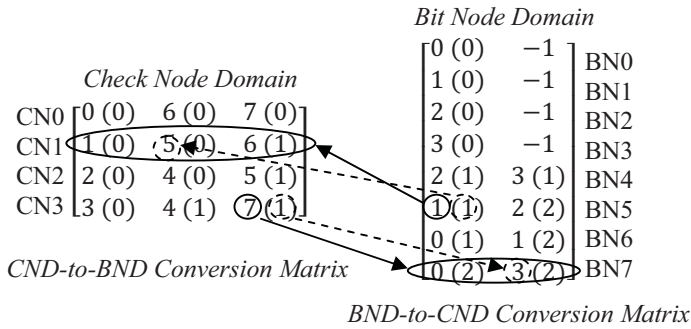


Fig. 5. Formation of the CND-to-BND conversion matrix and the BND-to-CND conversion matrix for use in CPU computation.

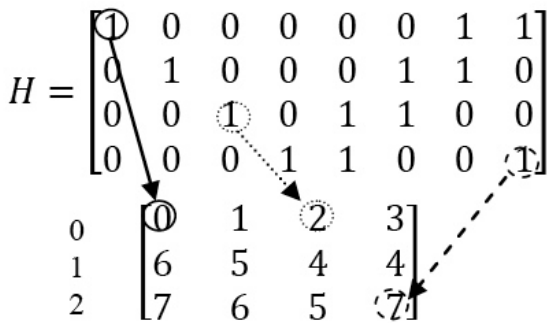


Fig. 6. Formation of the compressed parity-check matrix in the check-node domain (CPCM-CND) for use in GPU computation.

and the time taken. Two different LDPC codes — one irregular code with matrix size $10,008 \times 5,004$ and 37,214 edges; and one regular code with matrix size $10^5 \times 5 \cdot 10^4$ and $3 \cdot 10^5$

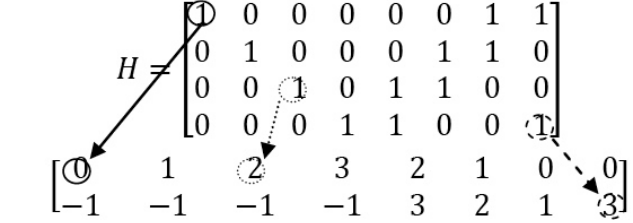


Fig. 7. Formation of the compressed parity-check matrix in the bit-node domain (CPCM-BND) for use in GPU computation.

edges — are simulated. Both codes have a code rate of 0.5 and they are constructed with the progressive-edge-growth (PEG) algorithm [8]. The maximum number of iterations taken to decode each codeword equals 200. If a valid codeword is not found at the end of 200 iterations, a decoding failure (block error) will be declared. Table II compares the time consumed per iteration for the GPU approach and the CPU approach. The results show that the GPU can reduce the simulation time by over 85% when compared with the time taken by the CPU. Moreover, the simulated block error rates (BLERs) and bit error rates (BERs) are found to be identical regardless of using GPU or CPU for computation.

Finally, we simulate two quasi-cyclic LDPC (QC-LDPC) codes [9] using the proposed GPU approach. 30 iterations are performed for each received QC-LDPC codeword. The properties of the two QC-LDPC codes, together with the simulated results, are listed in Table III. The results show that the GPU program can be run with long periods and decodes at a rate of about 160 blocks/s under the above conditions.



Fig. 9. Flow of the LDPC decoding process in a CPU.

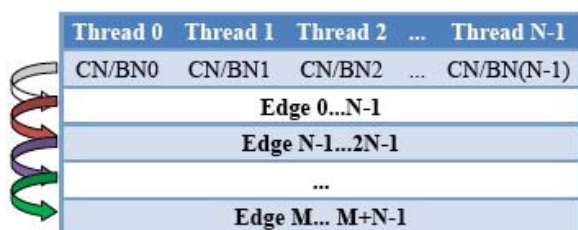


Fig. 10. Flow of the LDPC decoding process in a GPU.

V. CONCLUSIONS

This paper proposes a new approach based on GPU to decode LDPC codes. Modified compressed parity-check matrices (CPCMs), message matrices and conversion matrices have been proposed for GPU decoding. The results show that the GPU can complete each iteration with only 27% of the time taken by the CPU in the case of a rate-1/2 code with length 10,008. The time is further reduced to 14% of that taken by the CPU for the length 10^5 codeword. For a longer code or a code with more connections, the time reduction using GPU will be more significant.

ACKNOWLEDGEMENTS

The work described in this paper was partially supported by a grant from the RGC of the Hong Kong SAR, China (Project No. PolyU 521809) and by the National Natural Science Foundation of China (Grant No. 60972037).

REFERENCES

- [1] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters*, vol. 32, no. 18, pp. 1645–1645, 1996.
- [2] W. E. Ryan, "An Introduction to LDPC codes," in *CRC Handbook for Coding and Signal Processing for Recording Systems*, 2004.

- [3] A. Shokrollahi, "LDPC codes: An introduction," in *Coding, Cryptography and Combinatorics*, (K. Feng, H. Niederreiter and C. Xing, eds.) Birkhauser Basel, 2004.
- [4] T. J. Richardson and R. L. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 599–618, 2001.
- [5] T. J. Richardson and R. L. Urbanke, *Modern Coding Theory*. Cambridge University Press, 2008.
- [6] X. Zheng, F. C. M. Lau, C. K. Tse, and S. C. Wong, "Study of nonlinear dynamics of LDPC decoders," in *Proc. European Conference on Circuit Theory and Design*, pp. 157–160, Cork, Ireland, 2005.
- [7] S. F. Yau, F. C. M. L. T. L. Wong, and Y. He, "A Fast LDPC-Code Simulator Based on Compressed Parity-Check Matrices," *submitted*.
- [8] X. Y. Hu, E. Eleftheriou, and D. M. Arnold, "Regular and irregular progressive edge-growth tanner graphs," *IEEE Trans. Inform. Theory*, vol. 51, no. 1, pp. 386–398, 2005.
- [9] W. M. Tam, F. C. M. Lau, and C. K. Tse, "A Class of QC-LDPC Codes with Low Encoding Complexity and Good Error Performance," *IEEE Communications Letters*, vol. 14, pp. 169–171, Feb. 2010.