

# Programming Graphics Processing Units for the Decoding of Low-Density Parity-Check Codes

F. C. M. Lau\* and L. Shi

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

\*Email: encmlau@polyu.edu.hk

**Abstract**—Simulating the error performance of low-density parity-check (LDPC) codes usually takes a lot of computation time. Inexpensive graphics processing units (GPUs) have recently been used to accelerate the decoding process by allowing the smaller and identical tasks to be performed in a highly parallel manner. In this paper, we propose decoding a number of LDPC codes at the same time using GPUs. We show that the decoding speed is improved with our proposed method.

**Index Terms**—Error-correction code, GPU computing, LDPC code.

## I. INTRODUCTION

Low-density parity-check (LDPC) codes are a kind of error-correction code with excellent error performance [1], [2]. The decoding algorithm, which involves the iterative updating of bit-node-to-check-node (B2C) messages and check-node-to-bit-node (C2B) messages, is relatively complex [3], [4]. In particular, when a large number of codewords need to be decoded, a CPU requires much time to complete the decoding process. Sometimes a few weeks are needed to evaluate the error performance of a LDPC code with an extremely low error rate [5], [6].

Computer Unified Device Architecture (CUDA) [7] is a computing architecture that makes use of Graphic Processing Units (GPUs) [8] to perform computations in a highly parallel manner. In the updating of B2C (or C2B) messages, the messages are independent of one another. Hence we can divide the updating process into many smaller and identical tasks. Moreover, we can ensure that such tasks can be carried out in parallel and are highly suitable to be implemented by using CUDA. In [9], such a CUDA program that decodes LDPC codes using GPUs has been designed. In the program, the message updating processes are accomplished efficiently in parallel by the hundreds of cores in a GPU. The program, moreover, has been found to run much faster than a corresponding decoder program run in a CPU. The drawback of the program is that it decodes only one LDPC codeword at a time.

In this paper, we design a CUDA program which decodes multiple LDPC codewords at the same time. Instead of computing the smaller tasks in parallel for the same codeword, we compute the same task for different codewords in parallel. By decoding a large number of codewords simultaneously, we aim to improve the decoding speed by fully utilize all the resources available. We organize the paper as follows. To make the paper self-contained, we briefly review the LDPC codes

$$H = \begin{array}{c} \text{BN} \\ \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \left[ \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{array} \right] \end{array} \begin{array}{l} 0 \\ 1 \\ 2 \\ 3 \end{array} \text{CN} \end{array}$$

Fig. 1. A parity-check matrix with a maximum row weight of 3 and a maximum column weight of 2. BN: bit node; CN: check node.

and the decoding algorithm in Section II. In Section III, we describe the details of our proposed program architecture. In Section IV, we present our simulation results.

## II. REVIEW OF LDPC CODES

Each LDPC code is associated with a parity-check matrix (PCM) usually denoted by  $\mathbf{H}$ . In  $\mathbf{H}$ , most of the entries are zeros and the number of entries with ones is linearly proportional to the number of columns. Figure 1 shows an example of a PCM with a maximum row weight of 3 and a maximum column weight of 2. In addition, each column corresponds to a bit node and each row corresponds to a check node. Each non-zero element in  $\mathbf{H}$  further indicates that the corresponding bit node and check node are connected. Consequently, associated with the non-zero element are a B2C message and a C2B message used in the iterative decoding algorithm [9], [10].

In the decoding algorithm, messages are first received from the channel. We assume that an additive white Gaussian noise channel is used. Supposing there are  $N$  bit nodes (equals the number of columns in  $\mathbf{H}$ ),  $N$  channel messages, denoted by  $\lambda_i$  ( $i = 1, \dots, N$ ), are therefore received for each codeword. The B2C messages in the  $i$ th column are then all initialized to  $\lambda_i$ , i.e.,  $q_{ij} = \lambda_i$  where  $q_{ij}$  denotes the B2C message from bit node  $i$  to check node  $j$ . Assume that there are  $M$  check nodes. In each iteration, the following steps are performed. (For details of the algorithm, please refer to [9], [10].)

- 1) Check node  $j$  ( $j = 1, 2, \dots, M$ ), based on all the incoming B2C messages  $q_{ij}$  ( $i = 1, 2, \dots$ ), computes/updates the C2B message (denoted by  $r_{ji}$ ) to bit node  $i$  ( $i = 1, 2, \dots$ ).
- 2) Bit node  $i$  ( $i = 1, 2, \dots, N$ ), based on the incoming C2B messages  $r_{ji}$  ( $j = 1, 2, \dots$ ) and the channel message  $\lambda_i$ , computes/updates the B2C message  $q_{ij}$  to check node  $j$  ( $j = 1, 2, \dots$ ).

| Offset  | Codeword 0             | Codeword 1             | ... | Codeword K             |
|---------|------------------------|------------------------|-----|------------------------|
| 0       | Channel Message #0     | Channel Message #0     | ... | Channel Message #0     |
| K       | Channel Message #1     | Channel Message #1     | ... | Channel Message #1     |
| 2K      | Channel Message #2     | Channel Message #2     | ... | Channel Message #2     |
| ...     | ...                    | ...                    | ... | ...                    |
| (N-1) K | Channel Message #(N-1) | Channel Message #(N-1) | ... | Channel Message #(N-1) |

Fig. 2. Alignment of the channel messages for  $K$  codewords. Each codeword has  $N$  channel messages.

| Offset | CN0 | CN1 | CN2 | CN3 |
|--------|-----|-----|-----|-----|
| 0      | 0+0 | 0+1 | 0+2 | 0+3 |
| 4      | 0+6 | 0+5 | 0+4 | 8+4 |
| 8      | 0+7 | 8+6 | 8+5 | 8+7 |

(a)

| Offset | BN0 | BN1 | BN2 | BN3 | BN4 | BN5 | BN6 | BN7 |
|--------|-----|-----|-----|-----|-----|-----|-----|-----|
| 0      | 0+0 | 0+1 | 0+2 | 0+3 | 4+2 | 4+1 | 4+0 | 8+0 |
| 8      | -1  | -1  | -1  | -1  | 4+3 | 8+2 | 8+1 | 8+3 |

(b)

Fig. 3. Domain conversion matrices corresponding to the PCM in Fig. 1. (a) Check-node-domain-to-bit-node-domain (CND-to-BND); (b) bit-node-domain-to-check-node-domain (BND-to-CND).

- 3) Bit node  $i$  ( $i = 1, 2, \dots, N$ ), based on the incoming C2B messages  $r_{ji}$  ( $j = 1, 2, \dots$ ) and the channel message  $\lambda_i$ , decodes its bit value. Then the decoded codeword is checked against its validity. If the decoded codeword is a valid codeword, the decoding process terminates; otherwise, the next iteration begins. The iteration also stops when the number of iterations exceeds a preset maximum number.

### III. PROPOSED CUDA PROGRAM

#### A. Memory allocation

In our proposed CUDA program, we decode  $K$  codewords at the same time using  $K$  separate threads. In order to access the data from the memory efficiently, we have to align the messages in such a way that coalesced read/write can be performed. In Fig. 2, we show the arrangement of the channel messages for the  $K$  codewords in the memory. We can see that such an arrangement can facilitate the use of coalesced read. Note that there are  $N$  channel messages for each codeword.

In many cases, the parity-check matrices (PCMs) used are very large in size. A lot of memory will be required if we are to store the whole PCM, including the non-zero elements. It is therefore more efficient to store only the information of the non-zero elements in the PCM. Here, we compress the PCM and store it in two different domains — one in the check-node

|      | Offset | Thread 0 | Thread 1 | ... | Thread K-1 |
|------|--------|----------|----------|-----|------------|
| BN 0 | 0      | Edge #0  | Edge #0  | ... | Edge #0    |
|      | K      | Edge #1  | Edge #1  | ... | Edge #1    |
| BN 1 | 2K+0   | Edge #0  | Edge #0  | ... | Edge #0    |
|      | 2K+K   | Edge #1  | Edge #1  | ... | Edge #1    |
| ...  | ...    | ...      | ...      | ... | ...        |
| BN 7 | 7*2K+0 | Edge #0  | Edge #0  | ... | Edge #0    |
|      | 7*2K+K | Edge #1  | Edge #1  | ... | Edge #1    |

(a)

|      | Offset  | Thread 0 | Thread 1 | ... | Thread K-1 |
|------|---------|----------|----------|-----|------------|
| CN 0 | 0       | Edge #0  | Edge #0  | ... | Edge #0    |
|      | K       | Edge #1  | Edge #1  | ... | Edge #1    |
|      | 2K      | Edge #2  | Edge #2  | ... | Edge #2    |
| CN 1 | 3K+0    | Edge #0  | Edge #0  | ... | Edge #0    |
|      | 3K+K    | Edge #1  | Edge #1  | ... | Edge #1    |
|      | 3K+2K   | Edge #2  | Edge #2  | ... | Edge #2    |
| ...  | ...     | ...      | ...      | ... | ...        |
| CN 3 | 3*3K+0  | Edge #0  | Edge #0  | ... | Edge #0    |
|      | 3*3K+K  | Edge #1  | Edge #1  | ... | Edge #1    |
|      | 3*3K+2K | Edge #2  | Edge #2  | ... | Edge #2    |

(b)

Fig. 4. Alignment of (a) B2C messages and (b) C2B messages for  $K$  codewords.

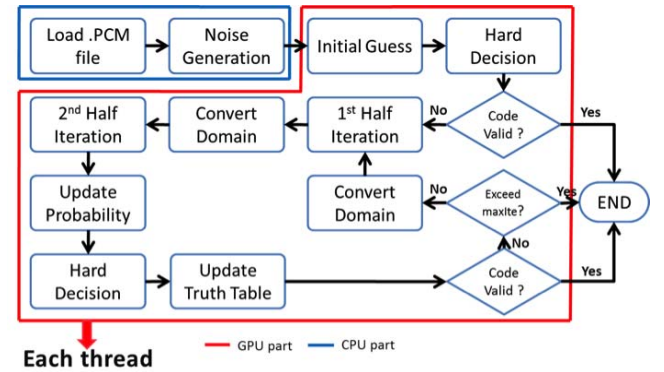


Fig. 5. Flow chart of the main program.

domain (CND) and the other in the bit-node domain (BND), as has been proposed in [9]. In addition, the B2C messages and the C2B messages are in compressed forms, i.e., in BND and in CND. Figure 3 further illustrates the conversion matrices [9] used for converting the B2C messages or C2B messages from one domain to another. Finally, for  $K$  codewords, we align the B2C messages and the C2B messages as in Fig. 4. Again, the reason for doing so is to enable the efficient use of coalesced read/write.

| Platform One |               |                                |
|--------------|---------------|--------------------------------|
| Hardware     | CPU           | Intel Core2 Duo E8400 @3.68GHz |
|              | GPU           | Inno3D GeForce GTS 450 1024MB  |
|              | Mainboard     | ASUS P5Q Pro                   |
|              | RAM           | A-DATA 2GB DDR2 @800 x 2       |
| Software     | OS            | Window7 Ultimate 32bit         |
|              | VS Version    | Microsoft Visual Studio 2008   |
|              | CUDA tool kit | V3.2 32-bit                    |
|              | CUDA SDK      | V3.2 32-bit                    |

(a)

| Platform Two |               |                                    |
|--------------|---------------|------------------------------------|
| Hardware     | CPU           | Intel Core i7 920 @ 2.66GHz        |
|              | GPU           | Inno3D GeForce GTX 295 1792MB      |
|              | Mainboard     | ASUS P6T6 X58                      |
|              | RAM           | Corsair DDR3 @1600 6GB kit (3x2GB) |
| Software     | OS            | Window7 Enterprise 64bit           |
|              | VS Version    | Microsoft Visual Studio 2008       |
|              | CUDA tool kit | V3.0 64-bit                        |
|              | CUDA SDK      | V3.0 64-bit                        |

(b)

Fig. 6. Hardware and software platforms used in the simulation. (a) Platform one; (b) platform two.

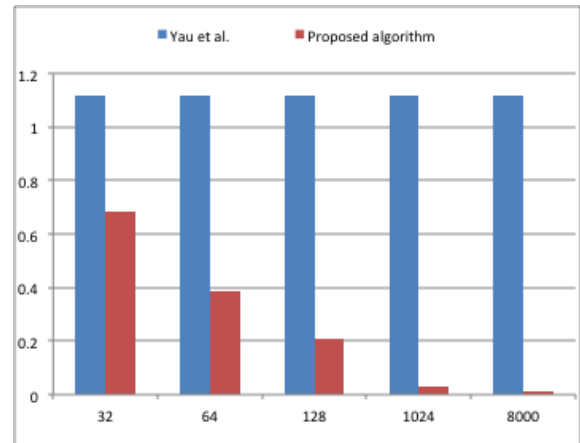
TABLE I  
DETAILS OF THE LDPC CODES USED IN THE SIMULATIONS.

| LDPC Code    | Number of Bit Nodes | Number of Check Nodes | Max. Column (Bit-Node) Weight | Max. Row (Check-Node) Weight | Number of Edges |
|--------------|---------------------|-----------------------|-------------------------------|------------------------------|-----------------|
| 1008x504pchk | 1008                | 504                   | 8                             | 20                           | 3716            |
| Wimax        | 2304                | 1152                  | 6                             | 7                            | 7200            |

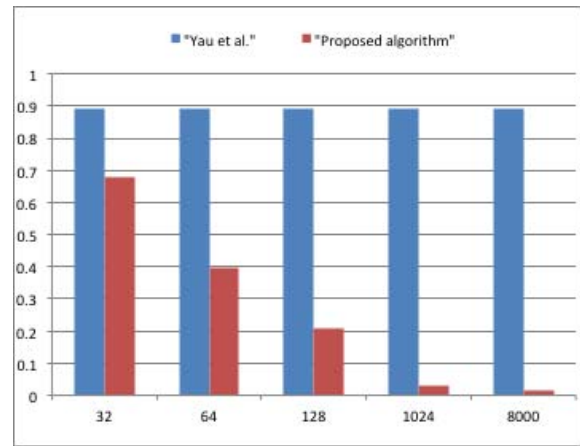
### B. Program flow

Figure 5 shows the flow diagram of the CUDA program. The program will first read a .PCM file which contains the details of the parity-check matrix. Based on the information, the compressed PCMs in the BND and CND are formed. Moreover, the domain conversion matrices are created. Then the  $K$  codewords, each of length  $N$ , are generated and mixed with noise. All the aforementioned data are subsequently loaded into the GPU memory. The CUDA kernel is called and all  $K$  threads are instructed to do the same calculations, with one thread assigned to decode one codeword.

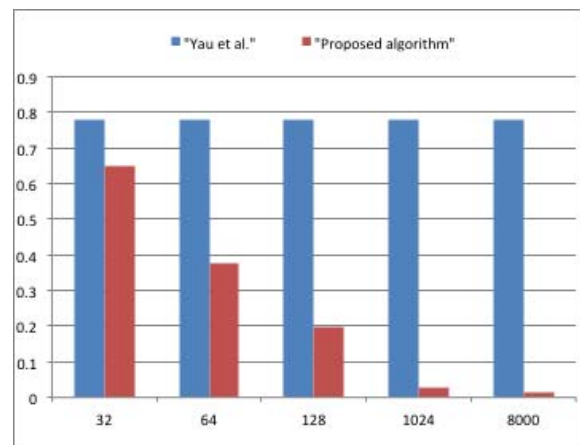
Referring to Fig. 5, an initial guess on each received codeword is first made based on the channel messages. If some decoded codewords (out of the  $K$  codewords) are valid, the corresponding entries in the truth table (of size  $K$ ) in the GPU memory will be set to "0"; otherwise, they will be set to "1". For codewords that are not valid, they will go through the iterative process — C2B message updating (first-half iteration), C2B messages converted from CND to BND, B2C message



(a)



(b)



(c)

Fig. 7. Relative simulation time taken to decode a fixed number of "1000x504pchk" LDPC code. Platform one is used and the number of codewords  $K$  being decoded simultaneously varies from 32 to 8000 in the proposed approach. The signal-to-noise ratio used is (a) 0.5 dB, (b) 1.2 dB and (c) 1.5 dB.

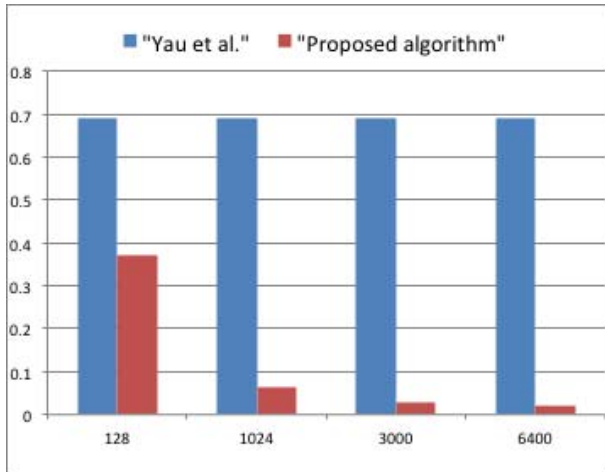


Fig. 8. Relative simulation time taken to decode a fixed number of “Wimax” LDPC code. Platform two is used and the number of codewords  $K$  being decoded simultaneously varies from 128 to 6400 in the proposed approach. The signal-to-noise ratio used is SNR = 1.2 dB.

updating (second-half iteration), bit-node decision, and validity check of decoded codeword. Again, the corresponding entries in the truth table will be updated (set to “0”) when valid codewords are decoded. The remaining codewords will then proceed to the next iteration — B2C messages converted from BND to CND, C2B message updating (first-half iteration), C2B messages converted from CND to BND, B2C message updating (second-half iteration), bit-node decision, and validity check of decoded codeword. The process continues until all the codewords have been decoded or when a preset maximum number of iterations has been executed.

#### IV. RESULTS AND DISCUSSIONS

We use two different platforms to test our program. The first one uses a GTS450 GPU while the second one uses a GTX295 GPU [8]. Details of the two platforms are shown in Fig. 6. We compare the simulation times of our approach and the one proposed by Yau *et al.* [9]. In running the GPU program proposed by Yau *et al.*, we set the number of threads per block to be 64 in the decoding process.

First, we simulate the “1000x504pchk” LDPC code which contains 1008 bit nodes and 504 check nodes. It is a relatively simple code and we run it on platform one. The signal-to-noise ratios (SNRs) used are 0.5 dB, 1.2 dB and 1.5 dB. In our approach, the number of codewords being decoded at the same time varies from 32 to 8000. The relative simulation times taken to decode a fixed number of codewords are shown in Fig. 7. We observe that our proposed approach requires less time to decode the same number of codewords compared with the algorithm used by Yau *et al.* Further, the simulation time reduces as the number of codewords being decoded at the same time increases in our proposed approach. The reason is that with the new approach, more resources (i.e., memory and cores) in the GPU can be utilized for the decoding process at the same time. The effect is more prominent if more codewords are being decoded simultaneously.

Next, we simulate the “Wimax” LDPC code [11], [12], [13]. It has twice the number of edges compared with the “1000x504pchk” code. In our approach, the number of codewords being decoded at the same time varies from 128 to 64000. The SNR used is 1.2 dB. This time, we run it using platform two and we plot the relative simulation times in Fig. 8. We arrive at the same conclusions as above, i.e., the simulation time is reduced compared with the method used by Yau *et al.* and the time decreases with the number of threads.

#### V. CONCLUSIONS

This paper proposes a new approach based on GPU to decode LDPC codes. The proposed approach decodes multiple codes at the same time and utilizes more resources available in the GPU. Simulation results have shown that, compared with the method used by Yau *et al.* which decodes one codeword at a time, our proposed approach can improve the time taken to decode the LDPC codes. Moreover, the speed-up improvement of our approach becomes more remarkable as the number of codewords being decoded at the same time increases.

#### 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).

#### REFERENCES

- [1] D. J. C. MacKay and R. M. Neal, “Near Shannon limit performance of low density parity check codes,” *Electronics Letters*, vol. 32, pp. 1645–1646, August 1996.
- [2] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, “Design of capacity-approaching irregular low-density parity-check codes,” *IEEE Trans. Inform. Theory*, vol. 47, no. 2, pp. 619–637, 2001.
- [3] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, “Reduced complexity iterative decoding of low-density parity check codes based on belief propagation,” *IEEE Trans. Commun.*, vol. 47, no. 5, pp. 673–680, 1999.
- [4] M. P. C. Fossorier, “Iterative reliability-based decoding of low-density parity check codes,” *IEEE Journal on Selected Areas in Communications*, vol. 19, no. 5, pp. 908–917, 2001.
- [5] L. Dini, F. Sottile, S. Benedetto, I. Superiore, M. Boella, and I. Torino, “Design of variable-rate irregular LDPC codes with low error floor,” in *Proc. IEEE Int. Conf. Commun.*, pp. 647–651, Seoul, Korea, 2005.
- [6] X. Zheng, F. C. M. Lau, and C. K. Tse, “Constructing Short-Length Irregular LDPC Codes with Low Error Floor,” *IEEE Transactions on Communications*, vol. 58, no. 10, pp. 2823–2834, Oct. 2010.
- [7] C. Nvidia, “Nvidia CUDA C programming guide version 3.2,” tech. rep., NVIDIA Corporation, 2010.
- [8] *The Current Generation CUDA Architecture, Code Named Fermi*. Available at [http://www.nvidia.com/object/fermi\\_architecture.html](http://www.nvidia.com/object/fermi_architecture.html).
- [9] S. F. Yau, T. L. Wong, and F. C. M. Lau, “Extremely fast simulator for decoding LDPC codes,” in *Proc. The 13th International Conference on Advanced Communication Technology*, Phoenix Park, Korea, Feb. 2011.
- [10] 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.
- [11] T. Brack, M. Alles, F. Kienle, and N. Wehn, “A synthesizable IP Core for WiMax 802.16e LDPC code decoding,” in *Proc. Personal Indoor and Radio Communications Conference*, pp. 1–5, Helsinki, Finland, 2006.
- [12] 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, no. 2, pp. 169–171, 2010.
- [13] IEEE, “IEEE standard for local metropolitan area networks. Part 16: Air interface for fixed and mobile broadband wireless access systems, Amendment 2: Physical and medium access control layers for combined fixed and mobile operation in licensed bands,” *IEEE Standard 802.16e-2005*, 28 Feb. 2006.