Parallel LDPC Decoder Implementation on GPU Based on Unbalanced Memory Coalescing

Soonyoung Kang and Jaekyun Moon
Department of Electrical Engineering
Korea Advanced Institute of Science and Technology
Daejeon, 305-701, Republic of Korea
Email: soonyoung@kaist.ac.kr, jmoon@kaist.edu

***Abstract*—We consider flexible decoder implementation of low density parity check (LDPC) codes via compute-unified-device-architecture (CUDA) programming on graphics processing unit (GPU), a research subject of considerable recent interest. An important issue in LDPC decoder design based on CUDA-GPU is realizing coalesced memory access, a technique that reduces memory transaction time considerably. In previous works along this direction, it has not been possible to achieve coalesced memory access in both the read and write operations due to the asymmetric nature of the bipartite graph describing the LDPC code structure. In this paper, a new algorithm is proposed that enables coalesced memory access in both the read and write operations for one half of the decoding process – either the bit-to-check or the check-to-bit message passing. For the remaining half of the decoding step our scheme requires address transformation in both the read and write operations but one translating array is sufficient. We also describe the use of on-chip shared memory and texture cache. Overall, experimental results show that proposed GPU-based LDPC decoder achieves more than 234x-speedup compared to CPU-based LDPC decoders and also outperforms existing GPU-based decoders by a significant margin.**

I. Introduction

The low-density parity check (LDPC) code is a powerful error correction code [1] [2] with a wide range of applications including many communication system standards such as WiFi (IEEE 802.11n), 10 Gbit Ethernet (IEEE 802.3an), WiMAX (IEEE 802.16e), and DVB-S2. Although LDPC codes have excellent error correcting capability, to date there exist no known mathematical tools to accurately evaluate their performance. Thus, a resort is typically made to simulations using computers or dedicated hardware.

LDPC decoding algorithms are based on the message-passing algorithm that demands very intensive computation. A number of dedicated hardware implementations of the LDPC decoder have been proposed in the past few years [3]-[5]. However, dedicated-hardware-based implementations require high cost and considerable design effort. Also, verification and validation require a long time. Furthermore, since the structure of the LDPC decoder changes according to the parity check matrix, developing an LDPC decoder for a different code means a new design process.

Recently, the graphics processing unit (GPU) has evolved into flexible platforms for general computing. The GPU provides extremely high computational throughput by executing thousands of threads simultaneously. Compute-unified-device-architecture (CUDA) is an efficient programming language for GPU implementation provided by NVIDIA. By CUDA, programmers are able to develop applications on GPU in high-level languages like C with some extensions of instruction sets.

As LDPC decoding can be easily implemented in parallel by CUDA, a number of research works on LDPC decoding implementation on GPU have been conducted [13][14]. In [14], the concept of asynchronous data transaction is presented. This method can reduce the data transfer time. Falcão *et al.* proposed the method to access memory in continuous pattern which can reduce the memory transaction time [13]. In order to implement their method, they introduced a data address transformation technique. Although this technique enables memory access in continuous pattern, it increases the use of memory and requires additional memory transactions. Moreover, they focused only on off-chip memory, not on on-chip memory or texture cache.

The goal of this paper is to develop a highly parallel decoding program which can evaluate the error correcting performance of LDPC codes with a very large number of simulation runs. The three major techniques described in this paper to accelerate the program are: coalesced memory access, use of on-chip memory and use of texture cache. Coalesced memory access is a highly effective way to reduce memory transaction time. In [13], coalesced memory access is achieved only in the read or the write operation (but not both) in each direction of the message-passing (bit-to-check and check-to-bit). In either direction, one translating array is necessary to align the addresses of the bit-to-check and check-to-bit edges; this means two translating arrays are needed overall. In contrast, the scheme proposed here enables coalesced memory access in both read and write operations in one direction of message passing. For the other direction, we resort to address transformation in both the read and write operations, requiring only one translating array. Since fully coalesced memory access is achieved in one direction of the message passing whereas no memory coalescing is attempted in the other direction, we call this scheme "unbalanced" memory coalescing. We also load the translating array into the on-chip shared memory, enabling a significant reduction of memory transaction time. In addition, in an effort to minimize repetitive access to off-chip memory, we insert bit-to-check and check-to-bit messages in the texture cache. The overall throughput

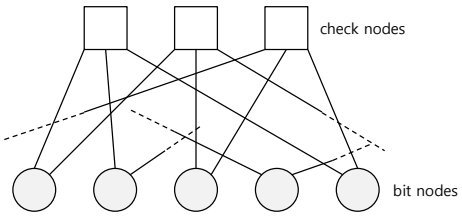


Fig. 1. Bipartite graph.

advantage of our scheme relative to existing schemes is shown to be significant.

In this paper, the sum-product algorithm (SPA) is implemented for the (6, 32)-regular LDPC code with the codeword length 2048 and the code rate 0.84, which is the IEEE 802.3an standard LDPC code. In order to evaluate the throughput performance of the proposed method, comparison of execution time with CPU-based implementation is presented. Moreover, computational throughput comparisons with other related works are presented.

The outline of this paper is as follows. Section II briefly introduces LDPC codes and message passing algorithms for decoding LDPC codes. In Section III, features of GPU and CUDA programming model are presented. In Section IV, [13] is reviewed and modified method for coalesced memory access is described. In addition, methods for using on-chip memory and texture cache are proposed. Experimental results are shown in Section V. Finally, conclusions are given in Section VI.

II. LDPC Codes and Decoding Algorithm

LDPC codes are linear block codes specified by a parity check matrix containing mostly 0's and only a small number of 1's [1]. The number of ones in a row is defined as the row weight d_c, and that in a column is defined as the column weight d_v. An LDPC code is called (d_v, d_c)-regular if every column has the same d_v, and every row has the same d_c. Otherwise, it is called irregular. Irregular LDPC codes require a lower signal-to-noise ratio (SNR) to reach the 'waterfall' threshold, but suffer from higher error floors [6]. For this reason, certain applications requiring low error rates employ regular LDPC codes.

LDPC decoding is based on the message passing algorithm. Fig. 1 shows a bipartite graph illustrating how bits are grouped together through the checks. Bits that are in the same group are connected to a common node, called the check node. The lower nodes represent bits, both information and parity bits, and are called the bit nodes. The bits which are tied to the same check can help identify one another, in terms of the probability of a bit being one or zero. In this paper, the sum-product algorithm (SPA), a popular form of the message passing algorithm, is used to decode regular LDPC codes [7].

A. Sum-Product Algorithm

Let Λ_n denote the soft information for bit n in the form of log-probability ratio. s_{mn} represents the message that check m passes onto bit n. Define q_{mn} as the message that bit n sends to check m. The superscript indicates the iteration stage. A pseudo-code for the message passing algorithm described above is given in $Algorithm$ 1. Note that the set $M(n)$ is the group of checks tied to bit n and $M_m(n)$ is the same group excluding check m. The overall message passing algorithm is basically separated into two steps: the check-to-bit message passing and the bit-to-check message passing. In the bit-to-check message passing, a bit node sends information to a particular check node, as it collects information from all other check nodes it is connected to. In this way, when a bit node passes information to a check node, it is ensured that there is no information in that message that has come from the same check node.

Algorithm 1 SPA

Initialization: $s_{mn}^{(0)} = 0, 1 \le n \le N, 1 \le m \le M$
for i = 1 to max-iteration, **do**
Perform bit-to-check message passing:
For $1 \le n \le N, 1 \le m \le M$,

$$q_{mn}^{(i)} = \Lambda_n^{(0)} + \sum_{k \in M_m(n)} s_{kn}^{(i-1)} \quad (1)$$

Perform check-to-bit message passing:
For $1 \le n \le N, 1 \le m \le M$,

$$s_{mn}^{(i)} = 2 \times tanh^{-1}\{ \prod_{i \in N_n(m)} tanh(\Lambda_i/2)\} \quad (2)$$

Compute the overall information for each bit:
For $1 \le n \le N$,

$$\Lambda_n^{(i)} = \Lambda_n^{(0)} + \sum_{k \in M(n)} s_{kn}^{(i)} \quad (3)$$

Check the codeword:
If the hard-sliced $\Lambda_n^{(i)}$'s satisfy the parity condition $\mathbf{H}\hat{\mathbf{c}} = 0$, stop and release the codeword.
end for

III. Features of GPU

In this section, the concept of the manycore microprocessor is discussed briefly. In addition, the programming model of CUDA, which is used for GPU programming interface in this paper, is presented. More detailed descriptions of CUDA can be found elsewhere [9]. Finally, the memory hierarchy of CUDA-GPU and patterns of memory transactions are described.

A. GPU: Manycore Microprocessor

Recently in the semiconductor industry, there are two trends for designing microprocessors, multicore CPUs and manycore GPUs. While the multicore microprocessors focus on

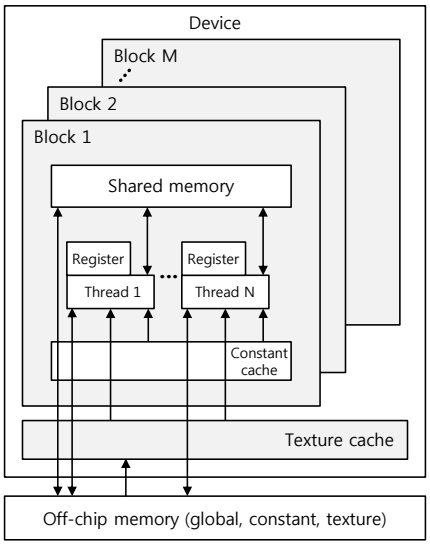


Fig. 2. Memory architecture of CUDA-GPU.

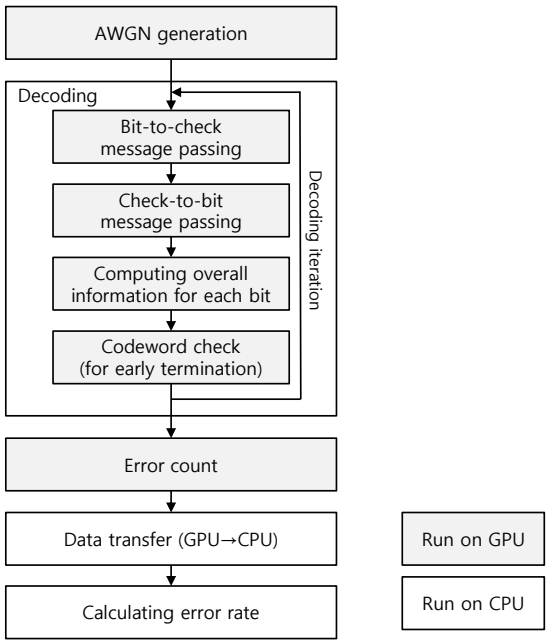


Fig. 3. Procedure of the simulation.

the execution speed of sequential programs, the manycore microprocessors focus more on the execution throughput of parallel applications. Manycore multiprocessors are composed of a large number of small cores and supports multithread programming. A multithreaded program is partitioned into blocks of threads that are executed independently from one another, so that a GPU with many cores can operate more programs in parallel than those with less cores.

B. CUDA Programming Model

The CUDA language defines a function called $kernel$, which is executed by multiple threads. Thread execution has a three-level hierarchy consisting of grid, block, and thread from the top. All threads form a grid and they all execute the same kernel. Each grid consists of 65535 blocks maximally while each block contains at most 512 threads, each of which is assigned to a streaming multiprocessor (SM) [9].

C. Characteristics of CUDA-GPU

The CUDA-GPU is comprised of several kinds of memories: global memory, local memory, shared memory, constant memory and texture memory. In order to maximize the operation speed of programs, the characteristics of all these memories should be understood. Fig. 2 illustrates the memory architecture of CUDA-GPU and patterns of memory transactions. This paper focuses specifically on global, shared and texture memories.

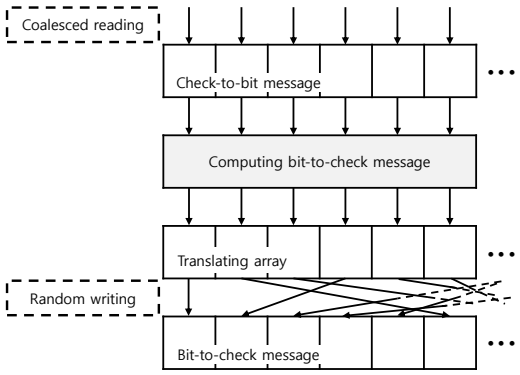
1) Global Memory: The global memory occupies the majority of space available on the off-chip memory. Because of its large capacity, it is easy to use. However, the global memory is an off-chip memory, which requires a longer access time than an on-chip memory. In order to minimize the access time to the global memory, data size and alignment should be considered. Any access to data in the global memory is done in a single instruction if and only if the size of the data type is 1, 2, 4, 8, or 16 bytes and the data is aligned. If size or alignment requirement is not satisfied, global memory access is carried out in multiple instructions, causing delayed access.

2) Shared Memory: The shared memory is an on-chip memory, which makes it much faster than the global memory. Since the shared memory is much smaller than the global memory, it is more difficult to use the shared memory in handling large data.

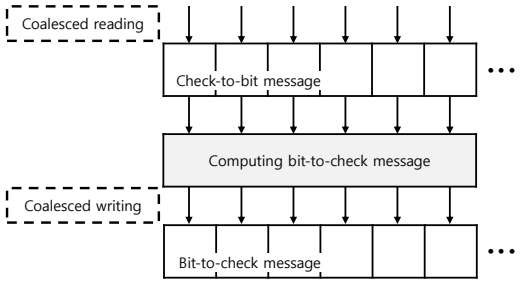
3) Texture Memory: CUDA supports a subset of texturing hardware that the GPU uses for graphics to access texture memory. The texture memory space resides in off-chip memory and is cached in the texture cache, so a texture fetch costs one memory read from off-chip memory only on a cache miss, otherwise it just costs one read from texture cache.

IV. Proposed LDPC Decoder on GPU

For the LDPC decoding implementation, noisy data is necessary. Fig. 3 describes the simulation procedure used in this paper. First, binary-phase-shift-keying (BPSK) channel outputs corrupted by additive white Gaussian noise (AWGN) are generated on the GPU. Then LDPC decoding begins with a bit-to-check message passing. There are three more steps in the decoding process: check-to-bit processing, calculating the overall soft-value of each bit, and checking the validity of the word. All these four steps are implemented on the GPU. In every process operated on the GPU, each thread operates one bit node (or check node) at a time. In order to calculate the error rates, an error counting kernel is run after the decoding process. To maximize the speed of counting errors, a parallel reduction algorithm is used [10]. After the error counting, its data is transferred to CPU for calculating error rates.



(a) Coalesced memory access of [13].



(b) Unbalanced memory coalescing.

Fig. 4. Memory access pattern for bit-to-check message passing in two different schemes of memory coalescing.

In order to evaluate the proposed algorithm, several terminologies are introduced in this section. Let T_{AWGN} denote the execution time for generating AWGN. In addition, T_{dec} represents the decoding time that is made up of T_{bc}, T_{cb}, T_{co} and T_{check}, denoting the execution times for bit-to-check message passing, check-to-bit message passing, computation of the overall soft-value of each bit and checking the validity of the word, respectively. Define N_{iter} as the iteration number of decoding. Moreover, while T_{count} denotes the execution time for counting errors, T_{trans} and T_{er} represents the execution time for transferring data (from GPU to CPU) and calculating error rate, respectively. Thus, total execution time, T_{total} can be written as:

$$T_{total} = T_{AWGN} + N_{iter} \times T_{dec} + T_{count} + T_{trans} + T_{er} \quad (4)$$

where

$$T_{dec} = T_{bc} + T_{cb} + T_{co} + T_{check} \quad (5)$$

A. Unbalanced Memory Coalescing

In order to reduce the access time to global memory while performing reading or writing operation on GPU, coalesced memory access should be considered. Unfortunately, due to the asymmetric nature of the bit node edge structure versus the check node edge structure, bit-to-check message passing and check-to-bit message passing cannot be simultaneously processed in coalesced fashion. In [13], two translating arrays are used. One translating array contains the direction information of bidirectional edges that link check nodes to bit nodes.

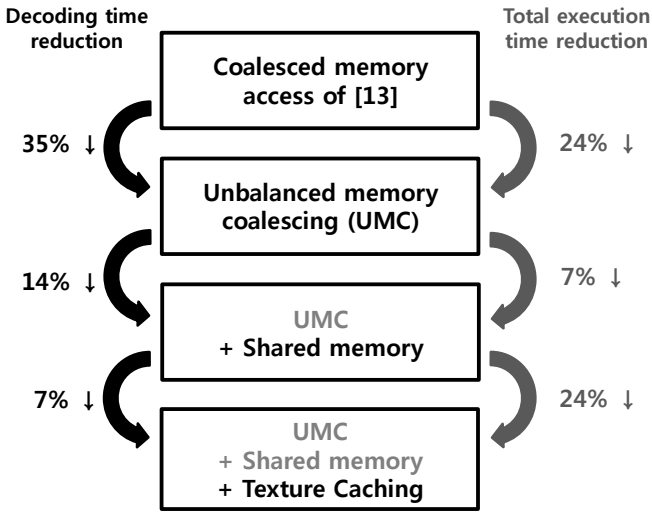


Fig. 5. Execution time reduction.

Another translating array contains the similar information for the opposite direction. As depicted in Fig. 4(a), the reading operation can be coalesced in performing the bit-to-check message passing, but the writing operation may not be coalesced because data addresses for writing have been shuffled by the translating array. The same argument applies to the check-to-bit message passing. Note that in this algorithm, two different translating arrays are need to be constructed as the array for the read operation is different from that of the write operation.

In this paper, numbering of the check-to-bit edges has been rearranged so that, as shown in Fig. 4(b) the bit-to-check message passing process can be coalesced during both reading and writing without a translating array. However, using this method, the translating array should be used in both reading and writing for the check-to-bit message passing, although the translation operation is identical between reading and writing. The identical translating array operation means that the array needs to be constructed only once, which can save the memory usage and the data transaction time. Moreover, the translating array is stored in local memory when it is read for the first time, so the total number of off-chip memory access is reduced by half, compared to the coalesced memory access scheme of [13]. Since fully coalesced memory access is achieved in one direction of the message passing whereas no memory coalescing is attempted in the other direction, we call this scheme "unbalanced" memory coalescing (UMC). As shown in Fig. 5, which summarizes the result of execution time reduction by the proposed method, UMC reduces T_{dec} by 35%, compare to the coalesced memory access scheme of [13].

B. Using Shared Memory

As the shared memory is small and restricted, data should be divided into a number of smaller partitions before being stored. Also the kernel which uses the same data should be divided into the same number of sub-kernels. In this paper, the translating array is stored in the shared memory because

TABLE I
DEVICES FOR EXPERIMENTS

	CPU	GPU
Platform	Intel i7-980X	NVIDIA GTX 480
Number of Cores	6 (used only single-core)	480
Clock Rate	3.33 GHz	1.51 GHz
Memory	12 GB	1.5 GB

its size is much smaller than bit-to-check (or check-to-bit) message data and its value is fixed in the whole decoding process. In contrast, message data – both bit-to-check and check-to-bit messages – change in every decoding iteration, which would cause overload of computation had they been stored in shared memory.

Before dividing the data, the number of partitions should be predetermined. Dividing the data into a smaller number of partitions would result in many threads per block but lower the number of simultaneous active blocks. In order to find the optimum number of partitions, the amount of shared memory per block, the total number of available registers per block and the architecture of GPU need to be considered [12]. Fig. 5 shows that dividing the translating array and using the shared memory reduce T_{dec} by 14%.

C. Texture Caching

Another method to reduce the memory access time to off-chip memory is to use cached memory. Although the state of the art GPUs (compatibility version 2.x) support cache for global memory, earlier versions of GPUs do not have cache for the global memory. Accessing global memory without cache increases the memory transaction time. In this paper we consider using the texture cache that can be applied to any version of GPU. As mentioned in Section III, reading from the texture cache can efficiently reduce the memory transaction time by obviating the need for off-chip memory access. In every iteration, all three critical decoding steps – bit-to-check message passing, check-to-bit message passing and computation of overall soft information for each bit – normally require constant access to off-chip memory. In addition, since retrieving AWGN samples, parity check matrix and hard sliced decision data require off-chip memory access in every codeword simulation, the area of loading these data should also be considered to reduce the data transaction time for multiple codewords simulation. In our design, in order to minimize repetitive access to off-chip memory, bit-to-check message, check-to-bit message, AWGN samples, parity check matrix for LDPC code and hard sliced decision data are loaded into the texture cache. Thus, T_{AWGN}, T_{bc}, T_{cb}, T_{co}, T_{check} and T_{count} can be decreased. Consequently, T_{total} is reduced by 24% with texture caching. This measurement was done on TESLA C1060 (compatibility version 1.3).

TABLE II
EXECUTION TIME (ms) COMPARISON

Number of Iterations	CPU	GPU
10	20.6	0.085
20	41	0.164
30	61.5	0.262
40	81.9	0.348
50	102.3	0.426

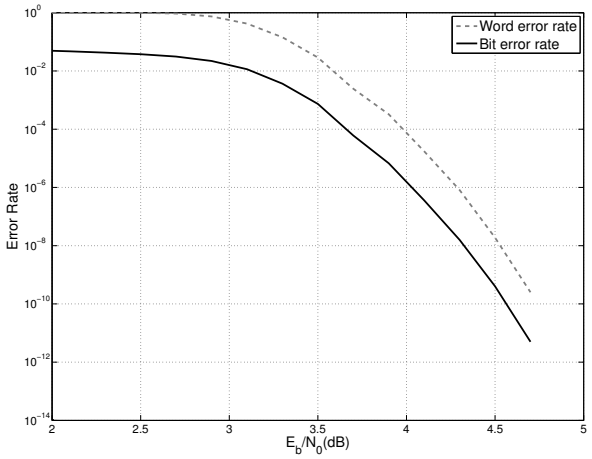


Fig. 6. Error rate performance of (2048, 1723) LDPC code.

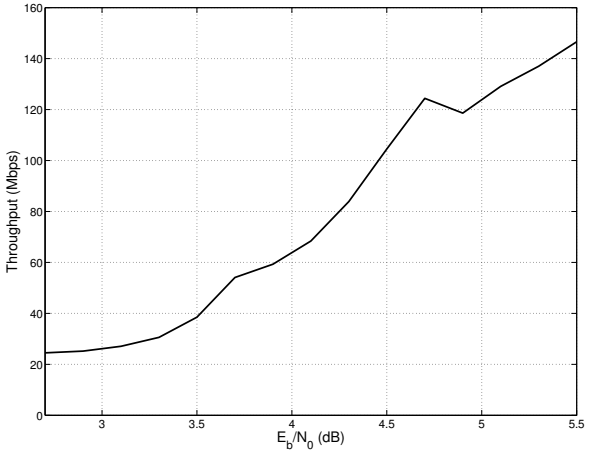


Fig. 7. Throughput of proposed scheme.

V. EXPERIMENTAL RESULTS

In this section, experimental results for decoding IEEE 802.3an LDPC code are presented. This LDPC code is (6, 32)-regular and has a total length of 2048 bits with rate 0.84 [8]. The experiments are performed using two types of devices, Intel i7-980X for CPU, which is the state-of-the-art machine, and NVIDIA GTX480 for GPU. Specifications of devices are presented in Table I. All simulations used 32-bit data precision. Comparisons of speed performance between the CPU and

TABLE III
COMPARISON WITH OTHER WORKS

Work	[13]		[14]	[15]	[16]	This Paper
Platform	8800 GTX		TESLA C1060	GTX 285	GTX 470	GTX 480
LDPC code (n, k)	(1024, 512)	(4896, 2448)	(4000, 2000)	(1944, 972)	(1944, 972)	(2048, 1723)
Code type (Column weight, Row weight)	(3, 6)	(3, 6)	(3, 6)	(3.6, 7.2) on average (irregular)	(3.6, 7.2) on average (irregular)	(6, 32)
Number of edges	3072	14688	12000	6966	6966	12288
Max iteration	10	10	10	10	10	10
Decoding algorithm	SPA	SPA	SPA	SPA	SPA	SPA
Early termination	No	No	No	Yes	Yes	Yes
Throughput (Mbps)	10.0	17.9	2.39	0.75	22.5-100.3	24.5-146.6

GPU are shown in Table II. In this experiment, a fixed number of iterations are run even if the decoded codeword after a given iteration turns out to be a valid codeword. The GPU-based decoder based on the proposed algorithm shows about 250x speedup compared to the CPU-based decoder for 20-iteration runs. For 30 iterations, the result shows the lowest speedup factor, although it is still more than 234x.

Table III presents comparison of the proposed method with other works. In [13], coalesced memory access was considered and, in [14], an asynchronous data transfer technique was applied. The code with low column and row weights are used (3 and 6, respectively) in both cases. In [15] and [16], an IEEE 802.11n (1944,972) LDPC code is implemented. This LDPC code is irregular with 6966 edges. Both works use an early termination scheme and the decoding stops when a valid codeword is detected. Our work also uses an early termination scheme. In this work, the throughput is measured with the various E_b/N_0, from 2.7 to 5.5 dB. The results show that the proposed method clearly outperforms the existing methods in terms of the throughput. We emphasize that the code considered in this paper is considerably more complex than those used in other works being compared.

Fig. 6 shows the error rate performance of the IEEE 802.3an LDPC code reflecting the proposed decoding method. The x-axis is for the E_b/N_0. As an example, the maximum number of iteration is set to 50 and 10 codeword errors were obtained at E_b/N_0 of 4.7 dB, at which the word error rate (WER) is 2.5×10^{-10}. This particular simulation run took 9 days by the GPU. For CPU-based implementation using a single core, more than 6 years would be required to obtain the same result.

Fig. 7 shows the E_b/N_0-dependent throughput of the proposed scheme with IEEE 802.3an LDPC code. For each E_b/N_0 point simulated, more than 40,000 codewords were implemented. The results in particular show that the achieved throughput is 124.4 Mbps at a 4.7 dB E_b/N_0, which is the point that shows the bit error rate of 4.9×10^{-12} (see Fig. 6).

VI. CONCLUSION

Flexible LDPC decoder implementation based on GPU is gaining popularity. In this paper, methods for parallel implementation of an LDPC decoder on GPU were presented. In particular, a new method for aligning data was proposed for coalesced memory access. Specifically, full memory coalescing is induced in one direction of the message passing while only one translating array is utilized with no memory coalescing in the other direction. In addition, on-chip memory access and data cache techniques were utilized to maximize the throughput. Simulation results indicate that the proposed method accelerates the speed of the LDPC decoder and provides significant throughput improvement compared to existing methods.

ACKNOWLEDGMENT

This work was supported by MKE under Grant 10035202-2011-02 and NRF of Korea under Grant 2011-0029854.

REFERENCES

[1] R. G. Gallager, "Low-Density Parity-Check Codes," IRE Trans. Inform. Theory, vol. IT-8, pp. 21-28, Jan. 1962.
[2] D.J.C.MacKay and R.M.Neal, Near Shannon limit performance of low density parity check codes, Electron Lett., vol.32, pp.1645-1646.1996.
[3] S. Muller, M. Schreger, M. Kabutz, M. Alles, F. Kienle, and N. Wehn, A novel LDPC decoder for DVB-S2 IP, in Proceedings of Design, Automation and Test in Europe, 2009 (DATE09), April 2009, pp. 13081313.
[4] Y.-L. Ueng, C.-J. Yang, Z.-C. Wu, C.-E. Wu, and Y.-L. Wang, VLSI decoding architecture with improved convergence speed and reduced decoding latency for irregular LDPC codes in WiMAX, in IEEE International Symposium on Circuits and Systems, 2008, pp. 520523.
[5] Guido Masera, Federico Quaglio, and Fabrizio Vacca, "Implementation of a Flexible LDPC Decoder," IEEE Transactions on Circuits and Systems II: Express Briefs, vol. 54, no. 6, June 2007.
[6] William E. Ryan and Shu Lin, "Channel Codes: Classical and Modern," Cambridge University Press, 2009.
[7] Shu Lin and D. J. Costello, "Error Control Coding: 2nd Edition," Prentice Hall, 2004.
[8] IEEE Standard 802.3an, Available: http://www.ieee802.org/3/an
[9] "NVIDIA CUDA C Programming Guide Version 3.2," NVIDIA, 2010.
[10] Mark Harris, "Optimizing Parallel Reduction in CUDA," NVIDIA, 2010.
[11] David B. Kirk and Wen-mei W. Hwu, "Programming Massively Parallel Processors," NVIDIA, 2009.
[12] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA," in proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP), 2008.
[13] G. Falcao, L. sousa, and V. Silva, "Massively LDPC Decoding on Multicore Architectures," IEEE Transactions on Parallel and Distributed Systems, vol. 2, no. 2, pp. 309-322, 2011.
[14] Cheng-Chun Chang, Yang-Lang Chang, Min-Yu Huang, and Bormin Huang, "Accelerating Regular LDPC Code Decoders on GPUs," IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 4, no. 3, September, 2011.
[15] H. Ji, J. Cho, and W. Sung, Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU, Journal of Signal Processing Systems, pp. 111, 2010, 10.1007/s11265-010-0547-9. [Online]. Available: http://dx.doi.org/10.1007/s11265-010-0547-9
[16] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, A Massively Parallel Implementation of LDPC Decoder on GPU, in Proceedings of IEEE Symposium on Application Specific Processors, 2011, pp. 82-85