MASSIVELY PARALLEL IMPLEMENTATION OF CYCLIC LDPC CODES ON A GENERAL PURPOSE GRAPHICS PROCESSING UNIT

Hyunwoo Ji, Junho Cho, and Wonyong Sung

School of Electrical Engineering, Seoul National University
599 Gwanak-ro, Gwanak-gu, Seoul 151-742, Korea
E-mail: {hwji, juno} @dsp.snu.ac.kr; wysung@snu.ac.kr

ABSTRACT

Simulation of low-density parity-check (LDPC) codes frequently takes several days, thus the use of general purpose graphics processing units (GPGPUs) is very promising. However, GPGPUs are designed for compute-intensive applications, and they are not optimized for data caching or control management. In LDPC decoding, the parity check matrix **H** needs to be accessed at every node updating process, and the size of **H** matrix is often larger than that of GPU on-chip memory especially when the code-length is long or the weight is high. In this work, the parity check matrix of cyclic or quasi-cyclic LDPC codes is greatly compressed by exploiting the periodic property of the matrix. In our experiments, the Compute Unified Device Architecture (CUDA) of Nvidia is used. With the (1057, 813) and (4161, 3431) projective geometry (PG)–LDPC codes, the execution speed of the proposed method is more than twice of the reference implementations that do not exploit the cyclic property of the parity check matrices.

Index Terms— Low-density parity-check (LDPC) codes, Compute Unified Device Architecture (CUDA), general purpose graphics processing unit (GPGPU), parallel processing

1. INTRODUCTION

Low-density parity-check (LDPC) codes [1] show excellent error correcting performance that is close to the Shannon's theoretical limit. However it is not possible to predict the performance of various decoding methods without resorting to many simulations. Examples that need intensive simulations are researches on the construction of good LDPC codes, high-performance low-complexity decoding algorithms, fast converging scheduling of decoding algorithms, and error floor phenomena. Also, LDPC decoder chip implementation for communication standards such as DVB-S2 [2], WiMax (802.16e) [3] and WiFi (802.11n) [4] demands error performance estimation, especially in fixed-point arithmetic. Cyclic or quasi-cyclic codes are advantageous in implementation, thus they are very favored as communication standards. The encoders for cyclic codes can be designed using linear feedback shift registers (LFSRs). The decoding hardware also needs much low interconnection complexity.

Since LDPC decoding process contains a lot of parallelism, general purpose graphics processing units (GPGPUs) are very efficient for conducting this job. One example was recently reported by Falcão *et al.* [5]. They organized an efficient data structure to represent the Tanner graph [6] and modified the algorithm to perform the sum-product algorithm (SPA) based LDPC decoding. They showed 22 times of speedup compared to Intel CPU based simulations. However, they only dealt with half-rate random codes whose weights are small.

The aim of this paper is to develop highly parallel decoding programs for LDPC codes by using on-chip memory efficiently. By storing the parity-check matrix of cyclic or quasi-cyclic codes in a memory-efficient way, we can efficiently decode longer codes with a limited on-chip memory capacity.

In this study, two decoding algorithms, the sum-product and the normalized min-sum algorithms (MSA), are implemented for the (1053, 817) and (4161, 3431) projective geometry (PG)-LDPC codes, and their execution times are compared with those of CPU based implementations. Considering that CPU based simulations in high signal to noise (SNR) conditions tend to take a very long time, this work will be very useful for development of LDPC codes. Note that field programmable gate array (FPGA) based implementations [7] of the LDPC simulator can shorten the simulation time, but this requires longer time to design and verify the hardware.

This paper is organized as follows. In Section 2, the characteristics of the GPGPU and Compute Unified Device Architecture (CUDA) are briefly explained. Basic notations of LDPC codes and two decoding methods are described in Section 3. In Section 4, we illustrate how parallel processes are implemented in the GPGPU and Section 5 shows the experimental results. Finally, concluding remarks are given in Section 6.

2. CHARACTERISTICS OF THE GPGPU

Once specially designed for computer graphics and difficult to program, today's GPUs are general purpose parallel processors with support for accessible programming interfaces and industry-standard languages such as C. Thus it does not restrict its application to only graphics any more, but is widening its applicable fields to physical modeling, computational engineering, matrix algebra, sorting and so on.

GPGPUs are specialized in highly parallel computation, and they are nicely matched with problems that can be modeled as data-parallel computations with high arithmetic intensity. Also the ratio of arithmetic to memory access has to be carefully considered in using GPGPUs. Even with the same computational complexity, efficient use of fast internal memory can lead to much shorter simulation time. Since transferring latency from host (PC) to global (GPU) memory and global to shared (thread) memory is very large, it is very desired to conduct computation with minimum access to the long-latency memory. The GPU architecture of Nvidia is depicted in Fig. 1. Serial or modestly parallel parts of application programs run in the host, while highly parallel parts operate in the device, which can be conveniently controlled using the CUDA (Compute Unified Device Architecture). A device is composed of many streaming multiprocessors (SMs), and each of which has 8 streaming processors (SPs). The device allocates thread blocks, whose size can be configured by a programmer, to SMs with their individual index. Each SP has its own registers, and each SM contains shared memory and constant cache memory

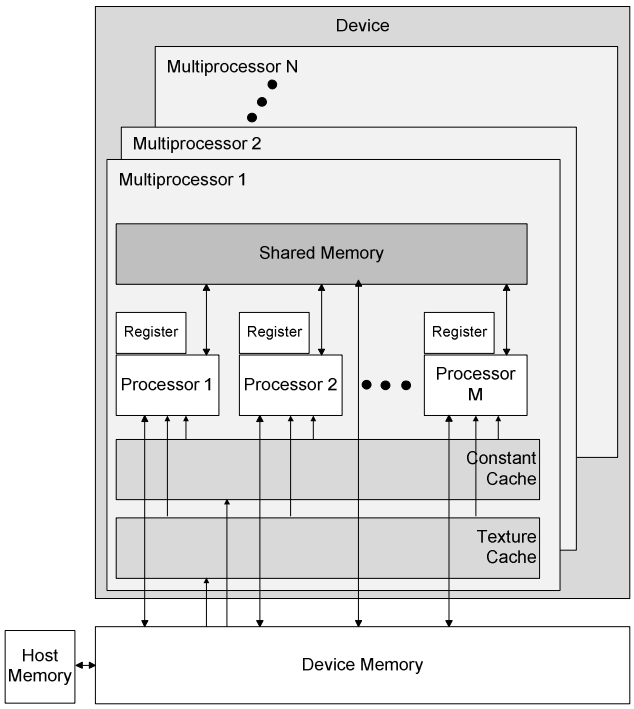

Figure 1. Nvidia GPU architecture [8]

blocks. The GPU based system contains its own GDDR (Graphics Double Data Rate) DRAM as the global memory [8].

3. LDPC CODES AND A BRIEF REVIEW OF SUM-PRODUCT AND MIN-SUM ALGORITHMS

LDPC codes deliver very good error correcting performance when decoded by the belief-propagation (BP), also known as the sum-product algorithm (SPA) [9]. Since the SPA has high computational complexity due to hyperbolic tangent computations, an approximation algorithm is devised, which is generally known as the min-sum algorithm (MSA) [10]. Additional researches have found that the use of normalization factor [11] or offset [12] in updating check nodes makes the performance of MSA almost comparable to that of the SPA.

An LDPC code is defined as the null space of a parity check matrix **H** with the following structural properties: (1) each row consists of ρ "ones"; (2) each column consists of γ "ones"; (3) the number of "ones" in common between any two columns is no greater than 1; both ρ and γ are small, compared to the length of the code and the number of rows in **H** [1]. Since ρ and γ are small, **H** has a small density of "ones" and hence is a sparse matrix. The LDPC code defined above is called a regular LDPC code. If the numbers of "ones" on rows or columns are unequal, that LDPC code is said to be irregular.

Regular PG-LDPC codes [13] are used in our simulation. They have good minimum distance, and the girth of their Tanner graph is proved to be at least 6 by construction. Furthermore, they can be represented in a cyclic form, which is very advantageous for hardware implementation.

In the following, we assume binary phase-shift keying (BPSK) modulation, which maps a codeword $\mathbf{c} = (c_1, c_2, \ldots, c_N)$ into a transmitted sequence $\mathbf{s} = (s_1, s_2, \ldots, s_N)$, according to $s_n = 2c_n - 1$, for n = 1,2, …, N. Then, $\boldsymbol{s}$ is transmitted over a channel corrupted by additive white Gaussian noise (AWGN). The received value corresponding to s_n after demodulation is $y_n = s_n + v_n$, where v_n is a random variable with zero mean and the variance of $N_0/2$. We define the set of bits that participate in check m by $\mathcal{N}(m) = \{n: H_{mn} = 1\}$, and the set of checks that participate in bit n by $\mathcal{M}(n) = \{m: H_{mn} = 1\}$. $\mathcal{N}(m)\backslash n$ denotes the set $\mathcal{N}(m)$ excluding bit n, and $\mathcal{M}(n)\backslash m$ the set $\mathcal{M}(n)$ excluding check m.

3.1. Sum-Product Algorithm

The SPA is operated on the log-likelihood ratio (LLR) domain in order to replace expensive multiplications with cheap additions. An LLR transferred from bit node n to check node m provides the hard decision estimation of the n-th bit and the reliability of this estimation, and that passed from check node m to bit node n gives the same information

of the m-th check node. For simplicity of algorithm description, we define the following notations:

- F_n: *A priori* LLR of bit n. In SPA decoding, initially set $F_n = (\frac{4}{N_o})y_n$.
- L_{mn}: LLR of check m, sent from check m to bit n.
- Z_{mn}: LLR of bit n, sent from bit n to check m.
- Z_n: *A posteriori* LLR of bit n.

With these notations, the SPA proceeds the decoding as follows:

Initialization: For each m, n, set $Z_{mn} = F_n$.

Iterative processing: In every iteration, process the following three steps.

1) Check nodes update:

For each m and $n \in \mathcal{N}(m)$,

$$L_{mn} = \left(\prod_{n' \in N(m)\backslash n} sign\left(z_{mn'}\right)\right) \times 2\tanh^{-1}\left(\prod_{n' \in N(m)\backslash n} \tanh\left(\frac{|z_{mn'}|}{2}\right)\right). \quad (1)$$

2) Bit nodes update:

For each m and $n \in \mathcal{M}(m)$

$$z_{mn} = F_n + \sum_{m' \in M(n)\backslash m} L_{m'n}. \quad (2)$$

For each n,

$$z_n = F_n + \sum_{m \in M(n)} L_{mn}. \quad (3)$$

3) Decision

i) Quantize $\hat{\mathbf{c}} = (\hat{c}_1, \hat{c}_2, \ldots, \hat{c}_N)$ such that $\hat{c}_n = 1$ if $z_n \geq 0$, and $\hat{c}_n = 0$ otherwise.

ii) If $\hat{\mathbf{c}}\mathbf{H}^T = 0$, halt the decoding with the estimated codeword $\hat{\mathbf{c}}$ as the output; otherwise go back to step 1. If the iteration number exceeds the pre-designated limit, declare decoding failure.

3.2. Min-Sum Algorithm

Check node computation in (1) can be approximated by (4).

$$\hat{L}_{mn} = \left(\prod_{n' \in N(m)\backslash n} sign\left(z_{mn'}\right)\right) \times \min_{n' \in N(m)\backslash n}\left(|z_{mn'}|\right) \quad (4)$$

In order to compensate the performance degradation due to approximation, the check node update can be modified by normalization or offset constants [11][12]. In our experiments, the normalized MSA is used, in which the check node update is modified to (5).

$$\hat{L}_{mn} = \left(\prod_{n' \in N(m)\backslash n} sign\left(z_{mn'}\right)\right) \times \frac{\min_{n' \in N(m)\backslash n}\left(z_{mn'}\right)}{\alpha} \quad (5)$$

where α is a small positive number determined by density evolution.

4. DECODING PROCESS ON CUDA

There are two possible parallelization strategies, one of which is to process the decoding frame-by-frame by processing a single frame with the whole hardware and the other is to process multi-frames simultaneously by allocating exclusive blocks to each frame. For example, if there are 24 SMs in total, 6 frames can be simultaneously decoded with 4 blocks per each frame. Although the latter approach provides more parallelism, implementing this structure requires many conditional branches that are very expensive in the GPU. Also, in the PG-LDPC codes that have high weights, it is difficult to allocate many threads for each block due to the limited size of shared cache memory. In addition, unequal iterations needed for simultaneously executed frames might reduce the parallelization gain because the frames that finish their decoding earlier have to wait other frames.

Therefore, the frame-by-frame scheme with maximum utilization of shared memory is chosen in this paper. The LDPC decoders are simulated using the Nvidia GTX 285 graphics processor that contains 30 SMs. In the CUDA programming model, a thread block implies a group of threads to be processed by a single SM. Thus the number of thread blocks needs to be larger than that of SMs in order not to waste hardware resources. Also, the number of

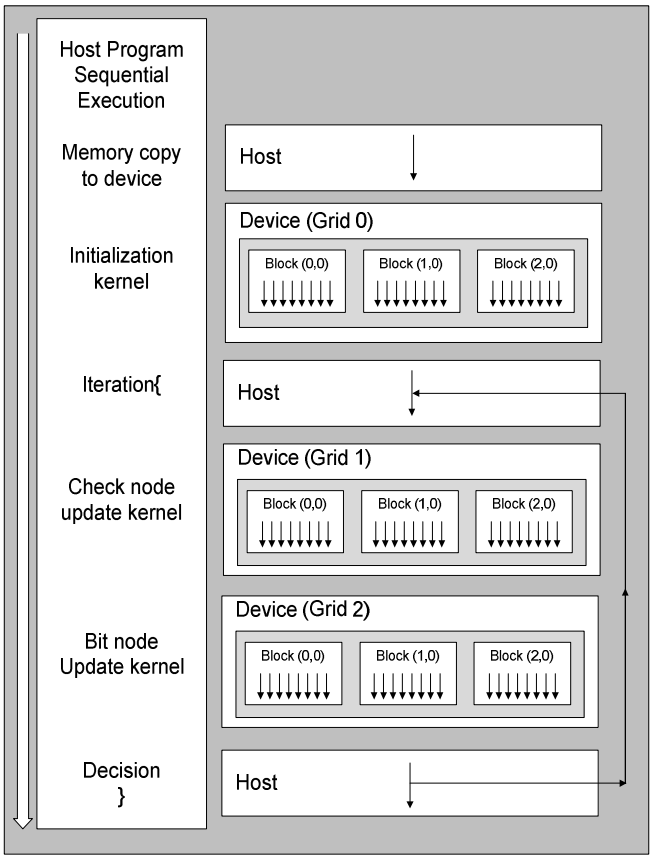

Figure 2. Devised heterogeneous programming model for SPA and MSA decoding

threads per thread block is recommended to be multiples of warps. Note that the warp is a group of 32 threads scheduled simultaneously in the GPU.

Fig. 2 depicts the proposed decoding procedure for SPA and MSA implemented on CUDA. Initially, the LLR of the received frame and the parity-check matrix **H** are copied to the constant cache memory by the host CPU. Then, at the initialization kernel of the GPGPU, $N \times \gamma$ threads are assigned to store the LLR of the received frame into Z[n][m], which is a two-dimensional $N \times M$ array of bit-to-check messages, where N is the code length. Thread i stores the LLR of i-th bit into Z[n][m]. Consecutive γ threads store the same value, F_n, to Z[n][m], whose indices can be accessed immediately from the parity-check matrix **H** stored in the constant cache memory. Both F_n and the indices can be broadcasted from the constant cache memory when those threads are allocated in the same warp.

In the following check node update kernel, M threads compute the two-dimensional $M \times N$ array of check-to-bit messages L[m][n], as defined in (1), where M denotes the number of check nodes. At first, each thread loads the required data with the size of ρ from Z[n][m] using the indices stored in **H**, and then stores them into the shared memory. At the same time, it performs the following two computations in (1):

$$\prod_{n \in N(m)} sign(z_{mn}) \quad \text{and} \quad \prod_{n \in N(m)} \tanh\left(\frac{|z_{mn}|}{2}\right).$$

After that, every thread runs ρ iterations again. At each iteration, the product of all $\tanh\left(\frac{|z_{mn}|}{2}\right)$ elements is divided by each element, then the $2 \times \tanh^{-1}$ of the quotient is stored in L[m][n] with the corresponding sign. In this step, since all the z_{mn} values have already been stored in the shared memory, they can be fetched very fast. If z_{mn} values are stored in the global memory that takes many cycles to access, the execution time is nearly doubled.

It is also possible to allocate $M \times \rho$ threads, each of which is responsible for computing one edge, for the check node update kernel. This increases the number of threads allocated for a block, and hence leads to higher warp occupancy. But this approach did not show better performance because of duplicated arithmetic and a limited number of SMs. The bit node update kernel that performs (2) is executed with N threads in a similar way of the check node update kernel. With the same strategy, the CUDA based MSA decoding can be easily implemented.

The large size bit-to-check and check-to-bit messages are stored in the large but slow global memory between the bit and check node update kernels. The host calls the next update kernel after completing all the current threads, and thereby preserves the memory coherency. As for the storage of the received frame and **H** matrix, the constant cache memory is used, which is a 64KB read-only memory block of the GPU while only write-enabled to the CPU. In the conventional method that records only the positions of non-zero elements in the **H** matrix, $2N\gamma$ $(= (N \times \gamma) + (M \times \rho))$ indices need to be stored. When the size of the **H** matrix exceeds the capacity of the constant cache memory, the overhead of accessing the **H** matrix in the global memory causes significant performance degradation.

$$H = \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

Figure 3. (8,8) cyclic code

In this paper, the **H** matrices of cyclic PG-LDPC codes are stored by their first row and column indices. The whole matrix can be simply reconstructed by them; e.g., if the indices of the ones in the first row of **H** are stored in r[γ], the k-th index of the n-th row is computed by '(r[k] + n) mod N' using the cyclic property. Thus, in this case, only $\rho + \gamma$ indices are required. As a comparison, the cyclic matrix shown in Fig. 3 needs to store 64 (= (8 columns × 4 elements in a column) + (8 rows × 4 elements in a row)) indices using a conventional method applied to sparse matrices, whereas only 8 (= 4 elements in the first column + 4 elements in the first row) indices are enough by using the cyclic property. Note that the check node update uses the row data, while the bit node update consults the column data. Also, with this arithmetic calculation of indices, all threads in a warp access the same address so that a read operation from the constant memory becomes as fast as that from the registers. If **H** is not saved in a cyclic manner, all threads in a warp access different addresses, in which case the read operations are serialized.

This method can be easily expanded to the quasi-cyclic (QC)-LDPC codes, for which the **H** matrix is composed of γ rows and ρ columns of $\delta \times \delta$ permutation submatrices. Since the permutation submatrices are cyclic shifts of the $\delta \times \delta$ identity matrix, only two indices are sufficient for

Table 1. Memory requirement for various LDPC codes

Codes			Memory (KB)			
			Constant			Global
Cons-truc-tion	(N, K)	(γ, ρ)	Frame	H	Total	Messa-ges
Ran-dom	(1008, 504)	(3, 6)	4	24	28	24
	(2640, 1320)	(3, 6)	10	63	74	63
PG	(1057, 813)	(33, 33)	4	0.2	4.2	280
	(4161, 3431)	(65, 65)	16	0.5	16.5	2164
	(16513, 14326)	(129, 129)	65	1	66	17173
QC	(1944, 972)	(12, 24)	8	23	31	192

each sub-matrix. Therefore, the memory requirement for the QC-LDPC codes are reduced from $2N\gamma\ (=2(\delta\rho)\gamma = 2\delta\rho\gamma)$ to $2\rho\gamma$, which corresponds to δ times saving. Memory requirement for various LDPC codes is listed in Table 1. The PG-LDPC codes require quite smaller memory compared to the random codes, even with a very long codeword length N and large weights ρ and γ. Note that in case of the (2640, 1320) random code, the frame array cannot be stored in the fast constant memory and, as a result, its access should be slow.

In our method, all computation intensive tasks are performed in the GPU, while the host CPU just copies the needed data, calls the kernels, and gathers statistics for analysis of simulation results.

5. EXPERIMENTAL RESULTS AND ANALYSIS

In order to compare the performance of CPU and GPU-based LDPC decoding methods, we developed C programs for them. Both codes use single-precision floating-point arithmetic. Table 2 depicts experimental environments in detail. In the following, the (1057, 813) and (4161, 3431) codes were decoded at 3.0dB and 4.0dB E_b/N_0, respectively. Note that CPU-based implementation does not utilize SIMD (single instruction multiple data) instructions and only one core is used. Note that SIMD based decoding of LDPC codes is also a challenge [15][16].

Table 2. Simulation environments

	CPU	GPGPU
Platform	Intel Core 2 Quad	Nvidia GTX 285
Number of cores	4 (only 1 core used)	240
Clock speed	2.42 GHz	1.48 GHz
Memory	2 GB	1 GB
OS	Linux(Fedora 9)	
Compiler	Gcc	Nvcc

Table 3 shows the execution time to decode 1,000 frames of several LDPC codes, where the maximum iteration number is forced to 15. The BER (bit error rate) performances of both CPU and GPU implementations are the same.

For decoding of the (1057, 813) code, the GPGPU based implementations of the SPA and the normalized MSA show 30 and 23 times faster execution speed, respectively. A longer code tends to exhibit better performance than a shorter one because a larger number of threads can be created, thereby allowing an enough number of warps assigned to each SM and hiding long latency of accessing the global memory. Ideally, it is recommended to have more than 13 warps for every SM to perfectly hide global memory access. Therefore, it is expected that longer codes yield higher performance gain as long as their frame LLR and **H** matrix do not cause the capacity overflow of the constant

Table 3. Decoding time (sec) for several LDPC codes with 1,000 frames

Code	Algorithm	CPU	GPGPU	Speed up
(1057, 813)	SPA	50.9	1.68	30.3
	Normalized MSA	47.9	2.09	22.9
(4161, 3431)	SPA	267.0	6.60	40.45
	Normalized MSA	376.7	15.25	24.7

Table 4. Decoding time (sec) when GPGPU programming is conducted in a non-cyclic manner

Algorithm	Codes	GPGPU	Speed up
SPA	(1057, 813)	3.79	13.4
	(4161, 3431)	23.76	19.5

cache (see Section 4). Because of the different convergence speed and error rate of two algorithms, the normalized MSA iterates more times than the SPA. In case of the (4161, 3431) code, the SPA iterates 1491 times whereas the normalized MSA iterates 2964 times. That is why the normalized MSA takes longer time than the SPA even though it is an approximation of the SPA.

If the time to obtain 100 frame errors is extrapolated to E_b/N_0 of 3.9dB, which shows the frame error rate (FER) of 10^{-6} for the (1057, 813) PG-LDPC code, approximately 60 days are required for the single core non-SIMD CPU-based implementation, while only 2 days (=48 hours) for the GPGPU-based one. Note that the fast convergence speed due to the increased E_b/N_0 is also considered in this estimation.

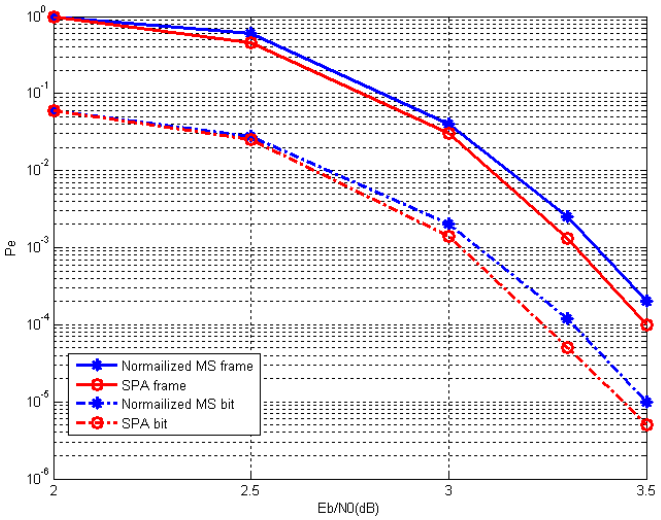

Figure 4. Error performance of the (1057, 813) PG-LDPC code

Table 4 shows the results when the GPGPU programs are developed without utilizing the cyclic characteristic of **H** matrix. Because the PG codes have large weights, the total required memory space for matrix exceeds 64Kbytes and the matrix has to be stored in the global memory. The comparison of the results in Table 3 and 4 shows that the speed-up exceeding 200% can be obtained by utilizing the cyclic property for storing the matrix.

We also implemented a (1944, 972) QC-LDPC code using the proposed approach. This code is irregular because the column weights are between 2 and 11. Decoding of irregular QC-LDPC codes can be inefficient because the difference of the number of ones in each column incurs the load imbalance problem. The implementation of (1944, 972) QC-LDPC code shows 12 times of speed-up when storing H matrix in the constant cache by utilizing the periodic property, while only 8 times speed-up when placing it in the global memory.

6. CONCLUDING REMARKS

We have developed efficient GPGPU-based programs for SPA (Sum-Product Algorithm) and MSA (Min-Sum Algorithm) based decoding of LDPC codes. Although longer LDPC codes are good for increasing the number of threads, their parity-check matrix size can be too large to store in the on-chip memory. This problem is especially critical when the weights of the codes are high. In order to alleviate this problem, the parity-check matrix of cyclic or quasi-cyclic codes is stored in a memory efficient way, and by which it was possible to obtain additional speed-up of larger than 200% for SPA decoding of cyclic codes and 150% for SPA decoding of a QC-LDPC code. Parallelization strategies for SPA and MSA based decoding on GPGPUs are also discussed.

7. ACKNOWLEDGEMENTS

This work was supported in part by the Ministry of Education and Human Resources Development (MOEHRD), Republic of Korea, under the Brain Korea 21 Project, and in part by the MIC/IITA/ETRI SoC Industry Promotion Center under the Human Resource Development. The authors would like to thank L. Sousa and M. Wu and the other reviewers for their valuable suggestions.

8. REFERENCES

[1] R.G. Gallager, *Low Density Parity Check Codes*, Cambridge, MA: MIT Press, 1963.

[2] The Digital Video Broadcasting Standard [Online]. Available: www.dvb.org

[3] The IEEE 802.16 Working Group [Online]. Available: http://www.ieee802.org/16/

[4] The IEEE 802.11n Working Group [Online]. Available: http://www.ieee802.org/11/

[5] G. Falcão, L. Sousa, and V. Silva, "Massive Parallel LDPC Decoding on GPU," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP 2008),* Salt Lake City, Utah, USA, pp. 83–90, Feb. 2008.

[6] R. M. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Trans. Inform. Theory*, vol. IT-27, pp. 533–547, Sept. 1981.

[7] T. Zhang and K. K. Parhi, "A 54 Mbps (3, 6)-regular FPGA LDPC decoder," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, pp. 127-132, 2001.

[8] URL - http://developer.nvidia.com/object/cuda.html

[9] D.J.C. MacKay, "Good Error-Correcting Codes Based on Very Sparse Matrices," *IEEE Trans. Info. Theory*, vol.45, pp. 399-431, Mar. 1999.

[10] 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, pp. 673-680, May 1999.

[11] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.Y. Hu, "Near Optimal Reduced-Complexity Decoding Algorithms for LDPC codes," in *Proc. IEEE Int. Symp. Information Theory*, Lausanne, Switzerland, p. 455, 2002.

[12] J. Zhao, F. Zarkeshvari, and A. Banihashemi, "On Implementation on Min-Sum Algorithm and Its Modifications of Decoding Low-Density Parity-Check (LDPC) Codes," *IEEE Trans. on Communication*, vol. 53, pp. 549–554, 2005.

[13] T. Richardson, M. Shokrollahi, and R. Urbanke, "Design of Capacity-Approaching Irregular Low-Density Parity-Check Codes," *IEEE Trans. Inform. Theory*, vol. 47, pp. 638–656, Feb. 2001.

[14] Y. Kou, S. Lin, and M. Fossorier, "Low Density Parity Check Codes Based on Finite Geometries: A Rediscovery and More," *IEEE Trans. Inform. Theory*, vol. 47, pp. 2711–2736, Nov. 2001.

[15] G. Falcão, V. Silva, M. Gomes, and L. Sousa, "Edge Stream Oriented LDPC Decoding," in *16th Euromicro International Conference on Parallel, Distributed and network-based Processing* (PDP), France, Feb. 2008.

[16] S. Seo, T. Mudge, Y. Zhu, and C. Chakrabarti, "Design and Analysis of LDPC Decoders for Software Defined Radio," in *Proc. IEEE Workshop on Signal Processing Systems (SiPS)*, Shanghai, China, pp. 210–215, Oct. 2007.