A Massively Parallel Implementation of QC-LDPC Decoder on GPU

Guohui Wang, Michael Wu, Yang Sun, and Joseph R. Cavallaro
Department of Electrical and Computer Engineering, Rice University, Houston, Texas 77005
{wgh, mbw2, ysun, cavallar}@rice.edu

***Abstract*—The graphics processor unit (GPU) is able to provide a low-cost and flexible software-based multi-core architecture for high performance computing. However, it is still very challenging to efficiently map the real-world applications to GPU and fully utilize the computational power of GPU. As a case study, we present a GPU-based implementation of a real-world digital signal processing (DSP) application: low-density parity-check (LDPC) decoder. The paper shows the efforts we made to map the algorithm onto the massively parallel architecture of GPU and fully utilize GPU's computational resources to significantly boost the performance. Moreover, several efficient data structures have been proposed to reduce the memory access latency and the memory bandwidth requirement. Experimental results show that the proposed GPU-based LDPC decoding accelerator can take advantage of the multi-core computational power provided by GPU and achieve high throughput up to 100.3Mbps.**

***Keywords*-GPU, parallel computing, CUDA, LDPC decoder**

I. Introduction

A graphics processing unit (GPU) provides a parallel architecture which combines raw computation power with programmability. GPU provides extremely high computational throughput by employing many cores working on a large set of data in parallel. In the field of wireless communication, although power and strict latency requirements of real communication systems continue to be the main challenges for a practical real-time GPU-based platform, GPU-based accelerators remain attractive due to their flexibility and scalability, especially in the realm of simulation acceleration and software-defined radio (SDR) test-beds. Recently, GPU-based implementations of several key components of communication systems have been studied. For instance, a soft information multiple-input multiple-output (MIMO) detector is implemented on GPU and achieves very high throughput [1]. In [2], a parallel turbo decoding accelerator implemented on GPU is studied for wireless channels.

Low-density parity-check (LDPC) decoder [3] is another key communication component and the GPU implementations of the LDPC decoder have drawn much attention recently, due to its high computational complexity. LDPC codes are a class of powerful error correcting codes that can achieve near-capacity error correcting performance. This class of codes are widely used in many wireless standards such as WiMax (IEEE 802.16e), WiFi (IEEE 802.11n) and high speed magnetic storage devices. The flexibility and scalability make GPU a good simulation platform to study the characteristics of different LDPC codes or to develop new LDPC codes. Recently, parallel implementations of high throughput LDPC decoders are studied in [4]. In [5], the researchers optimize the memory access and develop parallel decoding software for cyclic and quasi-cyclic LDPC (QC-LDPC) codes. However, there is still great potential to achieve higher performance by developing better algorithm mapping according to the GPU's architecture.

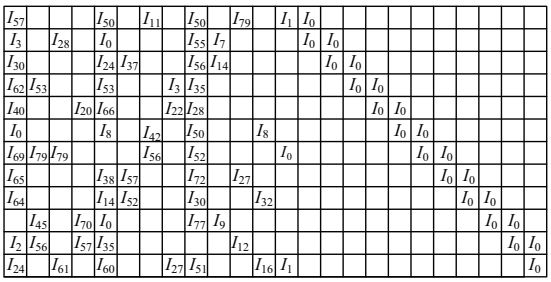


Fig. 1. Parity check matrix **H** for a block length of 1944 bits, code rate 1/2, IEEE 802.11n (1944, 972) LDPC code. **H** consists of $M_{sub} \times N_{sub}$ sub-matrices ($M_{sub} = 12, N_{sub} = 24$ in this example).

In this work, a highly-optimized and massively parallel LDPC decoder implementation on GPU is presented. This paper is organized as follows. Section II gives an overview of the LDPC decoding algorithm. Different aspects of the GPU implementation of the LDPC decoder and memory access optimization techniques are discussed in Section III. Section IV provides the experimental results for performance and throughput of the proposed implementation. Finally, Section V concludes this paper.

II. Introduction to LDPC Decoding Algorithm

A. QC-LDPC Codes

The binary LDPC codes can be defined by the equation $\mathbf{H} \cdot \mathbf{x}^T = 0$, in which $\mathbf{x}$ is a codeword and $\mathbf{H}$ is an $M \times N$ sparse parity check matrix. Quasi-Cyclic LDPC (QC-LDPC) codes are a special class of LDPC codes with a structured $\mathbf{H}$ matrix, which can be generated by the expansion of a $Z \times Z$ base matrix. As an example, Fig. 1 shows the parity check matrix for the (1944, 972) 802.11n LDPC code with sub-matrix size $Z = 81$. In this matrix representation, each square box with a label I_x represents an 81×81 circularly right-shifted identity matrix with a shifted value of x, and each empty box represents an 81×81 zero matrix.

B. Sum-product Algorithm for LDPC Decoder

The sum-product algorithm (SPA) is based on iterative message passing among check-nodes (CNs) and variable-

nodes (VNs) [3]. The SPA has a computational complexity of $O(N^3)$, in which N is normally very large. The SPA is usually performed in the log-domain (log-SPA).

Let c_n denote the n-th bit of a codeword, and let x_n denote the n-th bit of a decoded codeword. The *a posteriori* probability (APP) log-likelihood ratio (LLR) is soft information for c_n and can be defined as $L_n = log((Pr(c_n = 0)/Pr(c_n = 1))$.

1) Initialization:

L_n is initialized to be the input channel LLR. The VN-to-CN (VTC) message Q_{mn} and the CN-to-VN (CTV) message R_{mn} are initialized to 0.

2) Iterative Decoding:

For each VN n, calculate Q_{mn} by

$$Q_{mn} = L_n + \sum_{m' \in \{M_n \setminus m\}} R_{m'n}, \quad (1)$$

where $M_n \setminus m$ denotes the set of all the CNs connected with VN n except CN m. Then, for each CN m, compute the new CTV message R'_{mn} and Δ_{mn} by

$$R'_{mn} = Q_{mn_1} \boxplus Q_{mn_2} \boxplus \cdots \boxplus Q_{mn_k}, \quad (2)$$

$$\Delta_{mn} = R'_{mn} - R_{mn}, \quad (3)$$

where $n_1, n_2, \cdots, n_k \in \{N_m \setminus n\}$ and $N_m \setminus n$ denotes the set of all the CNs connected with VN n except CN m. The $\boxplus$ operation is defined as below:

$$x \boxplus y = \text{sign}(x)\text{sign}(y)\min(|x|, |y|) + \text{S}(x, y), \quad (4)$$

$$\text{S}(x, y) = \log(1 + e^{-|x+y|}) - \log(1 + e^{-|x-y|}). \quad (5)$$

3) Update the APP values and make hard decisions

$$L'_n = L_n + \sum_m \Delta_{mn}. \quad (6)$$

The decoder makes a hard decision to get the decoded bit x_n by quantizing the APP value L'_n into 1 and 0, that is, if $L'_n < 0$ then $x_n = 1$, otherwise $x_n = 0$. The decoding process terminates when the codeword $\mathbf{x}$ satisfies $\mathbf{H} \cdot \mathbf{x}^T = 0$, or the pre-set maximum number of iterations is reached. Otherwise, go back to step 2 and start a new iteration of decoding.

C. Scaled Min-Sum Algorithm

The min-sum algorithm (MSA) reduces the decoding complexity of the SPA with minor performance loss [6][7]. The R_{mn} calculation in the scaled MSA can be expressed as below:

$$R'_{mn} = \alpha \cdot \prod_{n' \in \{N_m \setminus n\}} sign(Q_{mn'}) \cdot \min_{n' \in \{N_m \setminus n\}} | Q_{mn'} |, \quad (7)$$

where α is the scaling factor to compensate for the performance loss in the min-sum algorithm ($\alpha = 0.75$ is used) [7].

III. Implementation of the LDPC Decoder on GPU

In this work, we use the Computer Unified Device Architecture (CUDA) programming model to implement the LDPC decoder. In order to reduce the complexity of the LDPC decoder, loosely-coupled algorithm [8] and forward-backward traversal scheme [9] are employed. Due to the space limit, the details of these two algorithms are not discussed here.

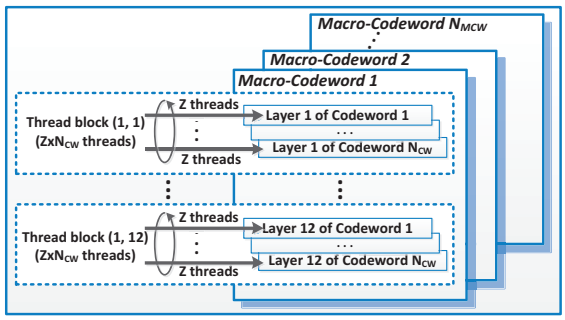


Fig. 2. Multi-codeword parallel decoding algorithm. The 802.11n (1944, 972) code is assumed. N_{CW} represents the number of codewords in one Macro-codeword (MCW). N_{MCW} represents the number of MCWs. Total number of thread blocks: $12 \cdot N_{MCW}$; total number of threads: $12 \cdot N_{MCW} \cdot N_{CW} \cdot Z$.

A. Mapping LDPC Decoding Algorithm to GPU Kernels

According to Equations (2), (3) and (6), the decoding process can be split into two stages: the horizontal processing stage and the APP update stage. We can create one computational kernel for each stage, which runs in the GPU. The host code running in the CPU takes charge of the CUDA initialization and memory copy between host and device.

1) CUDA Kernel 1: Horizontal Processing: During the horizontal processing stage, since all the CTV messages are calculated independently, we could use many parallel threads to process these CTV messages. For an $M \times N$ **H** matrix, M threads are spawned, and each thread processes a row. Since all non-zero entries in a sub-matrix of **H** have the same shift value (one square box in Fig. 1), threads processing the same layer (a row of square boxes in Fig. 1) have almost exactly the same operations when calculating the CTV messages. M_{sub} thread blocks are used and each consists of Z threads. Taking the 802.11n (1944, 972) LDPC code as an example, 12 thread blocks are generated, and each contains 81 threads, so there are a total of 972 threads used to calculate the CTV messages.

2) CUDA Kernel 2: APP value update: During the APP update stage, there are N APP values to be updated. Similarly, the APP value update is independent among variable nodes. Thus, N_{sub} thread blocks are used, with Z threads in each thread block. In the APP update stage, there are 1944 threads which are grouped into 24 thread blocks working concurrently for the 802.11n (1944, 972) LDPC code. Kernel 2 finally makes a hard decision for each bit.

B. Multi-codeword Parallel Decoding

Since the number of threads and thread blocks are limited by the dimensions of the **H** matrix, multi-codeword decoding is needed to further increase the parallelism of the workload. A two-level multi-codeword scheme is designed. N_{CW} codewords are first packed into one macro-codeword (MCW). Each MCW is decoded by a thread block and N_{MCW} MCWs are decoded by a group of thread blocks. The multi-codeword parallel decoding algorithm is described in Fig. 2. Since multiple codewords in one MCW are decoded by the threads within the same thread block, all the threads follow the same execution path. Moreover, the latency of read-after-write dependencies and memory bank conflicts can be completely hidden by a sufficient number of active threads.

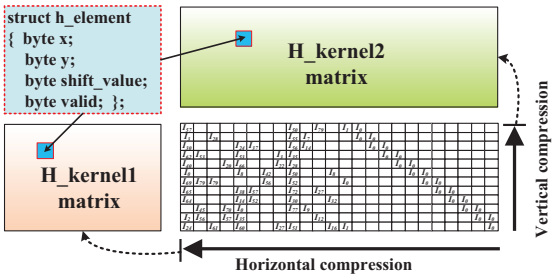


Fig. 3. The compact representation for **H** matrix. The **H** matrix is the same as in Fig. 1. After the horizontal compression and vertical compression, we get $\mathbf{H}_{kernel1}$ and $\mathbf{H}_{kernel2}$, respectively. Each entry of the compressed **H** matrix contains 4 8-bit data indicating the row and column index of the element in the original **H** matrix, the shift value and a valid flag which shows whether the current entry is empty or not.

C. Implementation of Early Termination Scheme

The early termination (ET) algorithm is used to avoid unnecessary computations when the decoder already converges to the correct codeword. For the LDPC codes, the parity check equations $\mathbf{H} \cdot \mathbf{x}^T = 0$ can be used to verify the correctness of the decoded codeword. A new CUDA kernel with M threads is launched and each thread calculates one parity check equation independently. Since the decoded codeword **x**, compact **H** matrix and parity check results are used by all the threads, on-chip shared memory is used to speed up the memory access. After the concurrent threads finish computing the parity check equations, we reuse these threads to perform a reduction operation on all the parity check results to generate the final ET check result, which indicates the correctness of the codeword. For multi-codeword parallel decoding, we propose a tag-based ET algorithm. We assign one tag per codeword and mark the tag once the corresponding parity check equation is satisfied. Once the tags for all the codewords are marked, the iterative decoding process is terminated.

D. Optimizing Memory Access on GPU

The latency of memory access is one of the major bottlenecks which limits the performance of the LDPC decoder. Several memory access optimization techniques are employed to further increase the throughput.

*1) Memory Optimization for **H** Matrix:* Reading from the constant memory is as fast as reading from a register as long as all the threads within a half-warp read the same address. Since all the Z threads in one thread block access the same entry of the **H** matrix simultaneously, we can store the **H** matrix in the constant memory and take advantage of the broadcasting mode of the constant memory. Simulation shows that constant memory increases the throughput by about 8%.

The quasi-cyclic characteristic of the QC-LDPC code allows us to efficiently store the sparse **H** matrix. We regard the cyclic **H** matrix in Fig. 1 as a 12×24 matrix $\bar{\mathbf{H}}$. As is shown in Fig. 3, we can get the compact matrices $\mathbf{H}_{kernel1}$ and $\mathbf{H}_{kernel2}$ by compressing $\bar{\mathbf{H}}$ horizontally and vertically, respectively. The compact representations of **H** reduces the device memory usage, therefore, the time spent on reading the **H** matrix from device memory is reduced. Moreover, the number of branch instructions which may cause throughput degradation are also reduced since there is no need to check whether an entry of **H** is empty. Taking the 802.11n (1944, 972) **H** matrix as an example, 40% of memory access and branch instructions are reduced by using the compressed $\mathbf{H}_{kernel1}$ and $\mathbf{H}_{kernel2}$.

2) Coalescing Device Memory Access: In CUDA kernel 1, R_{mn} and Δ_{mn} values are stored in the device memory. Since there is only one R_{mn} value and one Δ_{mn} value per row in each sub-matrix of **H**, the compressed format can be used to store R_{mn} and Δ_{mn}. Two $M \times \omega_r$ matrices are used to store R_{mn} and Δ_{mn}. In total, memory saving for R_{mn} and Δ_{mn} is more than halved. More importantly, the GPU supports very efficient coalesced access if all threads in a warp access the memory locations which have contiguous addresses. By writing the compressed R_{mn} and Δ_{mn} matrices column-wise, all memory accesses to R_{mn} and Δ_{mn} are coalesced. Simulation shows that 20% throughput improvement is achieved by coalescing device memory access for R_{mn} and Δ_{mn}.

TABLE I
DECODING THROUGHPUT ON GPU.

Code type	N_{iter}	Throughput (Mbps) log-SPA	min-sum
802.11n (1944, 972)	5	74.85	74.65
	10	39.98	39.82
	15	27.25	27.18
WiMAX (2304, 1152)	5	95.8	96.12
	10	52.15	52.31
	15	35.84	35.98

IV. EXPERIMENT RESULTS

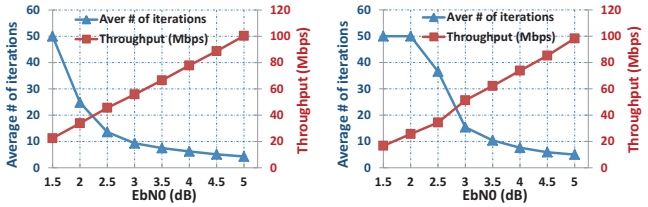
The experimental setup to evaluate the performance of the proposed architecture on the GPU consists of an NVIDIA GTX470 GPU with 448 stream processors, running at 1.215GHz and with 1280MB of GDDR5 device memory. We implement both the log-SPA and the min-sum algorithm.

A. Throughput Results

Assume the codeword length is N_{bits}, the total number of codewords is $N_{codeword}$, the simulation number is N_{Sim}, and the running time is T_{total}, which contains both the decoding time and the time for memory copy between host and device. The throughput can be calculated by: $Throughput = (N_{bits} \times N_{Sim} \times N_{codeword})/T_{total}$. According to the capacity of GTX470 GPU, around 300 codewords are processed in parallel in the multi-codeword decoding scheme ($N_{codeword} = 300$).

Table I shows the throughput of our implementation for both the 802.11n code and WiMAX code with different number of iterations (N_{iter}). The throughput for the log-SPA algorithm is comparable to the min-sum algorithm. The reason is that GPU implementation employs very efficient intrinsic functions __logf() and __expf(). And the bottleneck for GPU implementation is in the long latency of the device memory access, therefore, the run time for the extra instructions in the log-SPA is hidden behind the memory access latency.

Furthermore, the results also show that the decoder for the WiMAX code has higher throughput compared to the 802.11n code. The reason is that the row weights (ω_r) for



(a) Log-SPA algorithm (b) Scaled min-sum algorithm

Fig. 4. Experiment results for LDPC decoder with early termination scheme for the 802.11n (1944, 972) codes. The max number of iterations is set to 50.

TABLE II
DECODING THROUGHPUT COMPARISON WITH OTHER WORK.

Work	GPU	Code	Throughput
[10]	8800GT	(2048, 1024)[a]	2.95∼8.0 Kbps (ET[d])
[11]	Tesla C1060	(4000, 2000)[a]	2.34 Mbps
[4]	8800 GTX	(1024, 512)[a]	10.0 Mbps
	8800 GTX	(4896, 2448)[a]	17.9 Mbps
[5]	GTX 285	(1944, 972)[b]	0.75 Mbps (ET[d])
This work	GTX 470	(1944, 972)[b]	39.98 Mbps
		(1944, 972)[b]	22.5∼100.3 Mbps (ET[d])
		(2304, 1152)[c]	52.15 Mbps

[a] Regular codes.
[b] 802.11n codes, irregular codes.
[c] WiMAX codes, irregular codes.
[d] Early termination scheme is used. For others, max $N_{iter} = 10$. In this work, the throughput with ET is measured with the EbN0=1.5 $\sim$ 5dB.

these two codes are similar, which means that the computational workload is comparable. Therefore, the WiMAX code which has longer codewords tends to have higher throughput according to the throughput equation. Furthermore, there are more arithmetic instructions per memory access for a longer codeword, which can hide the memory access overhead.

Fig. 4 shows the throughput results and the average number of iterations with the parallel early termination (ET) scheme. As the SNR (represented by EbN0) increases, the average number of iterations decreases and the decoding throughput increases. Fig. 4 shows that the parallel early termination scheme significantly speeds up the simulation for the high SNR. For low SNR, the ET version may be slower than the non-ET version due to overhead of the ET kernel. Therefore, an adaptive scheme can be used to speed up the simulation for the whole SNR range – the ET kernel launches only when the simulation SNR is higher than a specific threshold.

B. Comparison with Related Work

It is difficult to use massive threads to fully occupy the computation resources of the GPU when decoding the irregular LDPC codes. When processing an irregular LDPC code, imbalanced workloads cause the threads on GPU to complete the computations at different times and runtime is bounded by the threads with the most amount of work. Table II compares our work with the related work. Table II shows that although the irregular codes we used are theoretically harder to get higher throughput than the ones in the related work, our decoder still outperforms the related work with significant improvement, especially when the parallel ET scheme is used. Our work is directly comparable to [5] since they also implemented a decoder for 802.11n (1944, 972) QC-LDPC code. Although the GPU used in this work has approximately twice the amount of computation resource as in [5], our decoder achieves more than 50 times throughput compared to their work. This huge improvement can be attributed to our highly optimized algorithm mappings, efficient data structures and the memory access optimizations.

V. CONCLUSION

This paper presents the techniques and design methodology to fully utilize a GPU's computational resources to accelerate a computation-intensive DSP algorithm. As a case study, a massively parallel implementation of LDPC decoder on GPU is presented. To achieve high decoding throughput, several techniques including efficient algorithm mapping, compact data structures and memory access optimizations are employed. We take the LDPC decoder for the IEEE 802.11n WiFi LDPC code and 802.16e WiMAX LDPC code as examples to demonstrate the performance of our GPU-based implementation. The simulation results exhibit that our LDPC decoder can achieve high throughput around up to 100.3Mbps.

ACKNOWLEDGMENTS

This work was supported in part by Renesas Mobile, Texas Instruments, Xilinx, and by the US National Science Foundation under grants CNS-0551692, CNS-0619767, EECS-0925942 and CNS-0923479.

REFERENCES

[1] M. Wu, Y. Sun, S. Gupta, and J. Cavallaro, "Implementation of a high throughput soft MIMO detector on GPU," *Journal of Signal Processing Systems*, pp. 1–14, 2010, 10.1007/s11265-010-0523-4. [Online]. Available: http://dx.doi.org/10.1007/s11265-010-0523-4

[2] M. Wu, Y. Sun, and J. Cavallaro, "Implementation of a 3GPP LTE turbo decoder accelerator on GPU," in *IEEE Workshop on Signal Processing Systems (SIPS)*, 2010, pp. 192 –197.

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

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

[5] 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. 1–11, 2010, 10.1007/s11265-010-0547-9. [Online]. Available: http://dx.doi.org/10.1007/s11265-010-0547-9

[6] M. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Transactions on Communications*, vol. 47, no. 5, pp. 673 –680, May 1999.

[7] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, and X.-Y. Hu, "Reduced-complexity decoding of LDPC codes," *IEEE Transactions on Communications*, vol. 53, no. 8, pp. 1288 – 1299, 2005.

[8] S.-H. Kang and I.-C. Park, "Loosely coupled memory-based decoding architecture for low density parity check codes," in *IEEE Custom Integrated Circuits Conference (CICC)*, 2005, pp. 703 – 706.

[9] X.-Y. Hu, E. Eleftheriou, D.-M. Arnold, and A. Dholakia, "Efficient implementations of the sum-product algorithm for decoding LDPC codes," in *IEEE Global Telecommunications Conference*, 2001, pp. 1036 –1036E.

[10] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *IEEE Asilomar Conference on Signals, Systems and Computers*, 2008, pp. 171 –175.

[11] Y.-L. Chang, C.-C. Chang, M.-Y. Huang, and B. Huang, "High-throughput GPU-based LDPC decoding," vol. 7810, no. 1. SPIE, 2010, p. 781008. [Online]. Available: http://link.aip.org/link/?PSI/7810/781008/1