GPU Accelerated Scalable Parallel Decoding of LDPC Codes

Guohui Wang, Michael Wu, Yang Sun, and Joseph R. Cavallaro

Department of Electrical and Computer Engineering
Rice University, Houston, Texas 77005
Email: {wgh, mbw2, ysun, cavallar}@rice.edu

Abstract— **This paper proposes a flexible low-density parity-check (LDPC) decoder which leverages graphic processor units (GPU) to provide high decoding throughput. LDPC codes are widely adopted by the new emerging standards for wireless communication systems and storage applications due to their near-capacity error correcting performance. To achieve high decoding throughput on GPU, we leverage the parallelism embedded in the check-node computation and variable-node computation and propose a parallel strategy of partitioning the decoding jobs among multi-processors in GPU. In addition, we propose a scalable multi-codeword decoding scheme to fully utilize the computation resources of GPU. Furthermore, we developed a novel adaptive performance-tuning method to make our decoder implementation more flexible and scalable. The experimental results show that our LDPC decoder is scalable and flexible, and the adaptive performance-tuning method can deliver the peak performance based on the GPU architecture.**

Keywords-GPGPU, parallel LDPC decoder, reconfigurable and scalable algorithms, adaptive performance-tuning

I. INTRODUCTION

Low-density parity-check (LDPC) codes are a class of powerful error correcting codes that can achieve near-capacity error correcting performance [1]. The LDPC codes are widely used in Ethernet standards such as 802.3an, high speed magnetic storage devices and many cellular standards such as WiMAX (802.11e) and WiFi (802.11n).

For cellular networks, cloud RAN (C-RAN) has been proposed to reduce the deployment cost and to support multiple wireless standards [2]. In this design, the base-station is a simple RF frontend that captures raw samples. The raw samples are forwarded to a central location where computational intensive baseband processing is performed. To achieve the performance as well as handle multiple standards, one possibility is to use multiple commodity x86 processors. An alternative is to employ graphics processors to perform baseband processing, which can achieve higher computational performance than x86 processors by using many more cores.

Although power and strict latency requirements of real communication systems continue to be the main challenges for a practical real-time GPU based platform, the flexibility, scalability and high performance of GPU-based systems make it a powerful platform for fast simulation of new algorithms and fast prototyping of a new system. Recently, researchers have started to use GPU to accelerate baseband signal processing algorithms in wireless communication systems. For instance, a soft MIMO detector is implemented on GPU and achieves very high throughput [3]. Reference [4] proposed a parallel turbo decoding accelerator implemented on GPU. Due to the inherently massively parallel nature and very complicated message passing decoding algorithm, the GPU-based implementation of LDPC decoder is another key communication component that has been widely studied. For example, the parallel implementations of high throughput LDPC decoder on GPU are discussed in [5-9], in which the authors improved the throughput performance according to the GPU's architecture by employing techniques such as workload kernel partitioning, shared and coalesced memory access optimization, and multiple-level parallelism degree exploration.

Since GPUs evolve very quickly, several generations of GPU co-exist in the market with different underlying architectures. Each of the previous works only focuses on a specific GPU architecture. Therefore, existing implementations lack scalability and flexibility and only can achieve high performance for only a small class of codes on specific GPUs.

This paper presents a scalable and flexible implementation of LDPC decoder on GPU. The novel adaptive tuning algorithm is proposed to map different LDPC decoding algorithms to different GPU architectures, which can generate a high performance LDPC decoder independent from the types of LDPC codes and the architectures of the GPU. In addition, this paper studies several implementation issues of GPU-based LDPC decoder according to extensive experiments and analysis.

II. LDPC CODES

Low-density parity-check (LDPC) codes are a class of error correcting codes. The binary LDPC codes can be represented by the following equation:

$$\mathbf{H} \cdot \mathbf{x}^T = 0, \qquad (1)$$

in which $\mathbf{x}$ is a codeword and $\mathbf{H}$ is an $\mathbf{M} \times \mathbf{N}$ sparse parity check matrix. The $\mathbf{H}$ matrix can be expressed by a bipartite graph in which each row represents a check node (CN) and each column represents a variable node (VN). $\mathbf{M}$ denotes the number of parity check nodes, and $\mathbf{N}$ is the number of variable nodes. A non-zero element in $\mathbf{H}$ is called an edge and represents a connection between a variable node and a check node.

LDPC codes are usually decoded by using the sum-product algorithm (SPA), which is based on the iterative message passing among CNs and VNs [1]. The log-domain SPA (log-SPA) is preferred due to the complexity and numerical stability issues of the probability-domain SPA. The log-SPA is described in detail in [11] and [12]. The decoding process contains an initiation stage, iterative decoding stages with

check-node to variable-node (CTV) update and variable-node to check-node (VTC) update, and a hard decision stage.

In this paper, we take quasi-cyclic LDPC (QC-LDPC) codes as examples since this class of codes has several good features. For example, they are very friendly to hardware implementations. In addition, QC-LDPC codes are widely adopted in current and next generation wireless communication standards such as IEEE 802.11n WiFi and IEEE 802.16e WiMAX, so studying QC-LDPC codes has a strong practical meaning.

III. CUDA PROGRAMMING MODEL

Computer Unified Device Architecture (CUDA) adopted in this work is widely used to program massively parallel computing applications [10]. CUDA exploits the computational power of a GPU by employing the Single Instruction Multiple Threads (SIMT) programming model. In this model, a kernel is executed by thousands of concurrent multiple threads over different data sets. The threads executing a kernel on a GPU are distributed on a grid. Each grid consists of thread blocks with adjustable dimensions. All threads inside the same block can share data through a shared memory mechanism. They can also synchronize execution at specific synchronization barriers inside the kernel, where all the threads in a block are suspended until they all reach that point.

Because of the large number of cores and an efficient SIMT architecture, GPU can achieve very high peak performance. However, it is still challenging to program a GPU to achieve peak performance due to the following reasons:

1. We have to provide a sufficient number of threads (typically at least thousands) to fully occupy the general-purpose processing cores in GPU.

2. We need to minimize the device memory access time since device memory access takes hundreds of clock cycles. In addition, non-coalesced memory access will be serialized which increases the number of device memory requests. Therefore, the programmer should carefully coalesce device memory access to reduce the number of device memory requests.

3. Although the shared memory can be accessed very fast, the size of the shared memory per thread is quite limited, which limits the achievable maximum parallelism.

In addition, there are differences among different generations of GPUs. The SIMD width of the stream multiprocessor (SM) has doubled and the number of registers available has quadrupled from G80 to Fermi. In addition, the amount of shared memory available per SM has increased from 32KB to 48KB. As a result, tuning is required to reach peak performance on different generations of GPUs. For example, the maximum number of threads per block depends on the number of registers available on the GPU. To make sure that the maximum numbers of threads are running concurrently to reach peak performance while keeping processing latency relatively low, the thread block configurations need to be updated according to the target GPU hardware.
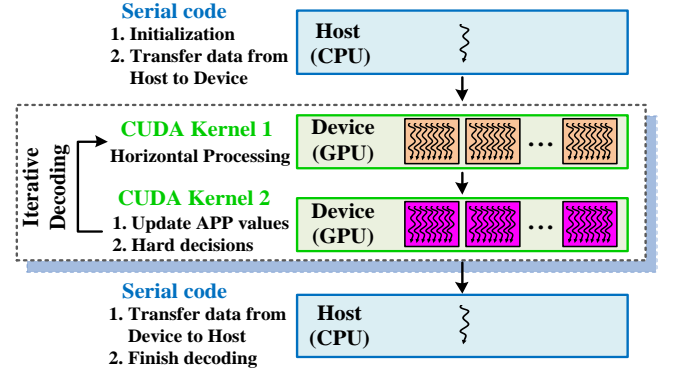

Figure 1: Code structure of the GPU implementation of LDPC decoder by using two CUDA kernels.

IV. ALGORITHM MAPPING AND IMPLEMENTATION

Because of the SIMT characteristic and the shared memory architecture employed by GPU, directly mapping the LDPC decoding algorithm will not deliver the peak performance of the GPU. Therefore, we first explore the methodology to map the LDPC decoding algorithm more efficiently.

A. Mapping LDPC Decoder to Many-core Architecture of GPU

The LDPC decoding process can be split into two stages: horizontal CTV processing stage, and vertical VTC and APP update stage [12]. Therefore, we can create one computational kernel for each stage. We can partition the complete decoding process into CPU host code and two GPU computation kernels. The relationship between the CPU host code and the GPU kernel is shown in Fig. 1.

The LDPC decoding algorithm itself bears massive parallelism so that it is very suitable for a GPU implementation. During the horizontal processing stage, all the rows of the **H** matrix are processed from left to right horizontally. Since all the CTV messages are calculated independently and there is no data dependency among rows, we could use many parallel threads to compute these CTV messages. For an **M×N** parity-check matrix, **M** threads are spawned, and each thread processes one row. During the VTC and APP updating stage, there are **N** APP values to be updated. Similar to the CTV message processing stage, the APP value update is independent among variable nodes. We can spawn **N** threads to update all the APP values in parallel. Since all the threads in each kernel access device memory which is visible to all the threads and thread blocks, to avoid a memory conflict and RAW (read-after-write) data hazard, we need to synchronize all the threads explicitly at the end of each kernel, using the *__syncthreads()* function.

B. Performance Optimization Schemes

The goal to use GPU as an accelerator is to implement a highly scalable LDPC decoder with high throughput performance. To archive this goal, we have applied several optimization methods including using compact representation of the **H** matrix, optimizing the device memory access, using

an early termination algorithm, and so on. The facts that these performance optimization methods are highly related to the LDPC decoding algorithm and GPU's hardware architecture indicate that the programmer should be familiar with the GPU's architecture and programming model to achieve peak performance. Moreover, we have utilized different memories including constant memory and coalesced device memory to fully take advantage of the memory hierarchy. The details of these optimization schemes have been explained in our previous work [9].

C. Multi-codeword Decoding Scheme

In order to maximize the data throughput, we need to make sure the workload is large enough for all the SMs to work without idle cores because of the GPU's many-core architecture. Directly mapping the LDPC decoding algorithm based on Section IV-A cannot provide enough workload to hide the long latency and overhead of device memory access. We propose the following 3D-structure multi-codeword decoding scheme.

The parity check matrices of QC-LDPC codes in 802.11n WiFi and 802.16e WiMAX have 12 layers (one layer represents one row of $Z \times Z$ sub-matrices, in which Z is called expansion factor of the parity-check matrix). Since the threads to process one layer share the same parity-check matrix entry, they have very similar execution paths from which we can take advantage of multi-thread processing inside one thread block. Therefore, the natural idea is to use one thread block to process one layer, which has Z threads in each block. However, one layer has only Z rows (Z=81 for 802.11n WiFi and Z=96 for 802.16e WiMAX). Thus, the parallelism is too small to achieve peak performance.

To increase the amount of the workload, we add another dimension of parallelism by packing several independent codewords to form a macro-codeword. We use multiple thread blocks to process the same layer of parity-check matrix for different macro-codewords because they still share the same execution path. So far, each macro-codeword is processed by 12 thread blocks, each of which has $Z \times N_{CW}$ threads (N_{CW} represents the number of codewords in one macro-codeword).

To further increase the workload, we add the third dimension to the multi-codeword structure, in which we decode multiple (N_{MCW}) Macro-codewords in parallel. This 3D-structure multi-codeword decoding scheme not only provides higher parallelism but also enables the adaptive performance-tuning method described below.

D. Adaptive Performance-Tuning Method

Different GPUs have different architectures and hardware resources in terms of the number of stream multi-processors (SM) and maximum parallelism supported per SM, amount of shared memory and amount of registers [10]. The variations on these aspects are so big that the parallel programs tuned for certain GPUs will quite possibly perform badly on other GPUs in terms of flexibility, scalability and throughput performance. The mapping from algorithm to GPU architecture should be quite different according to the computation capability of the GPU [10]. Therefore, we propose an adaptive performance-tuning method to dynamically calculate the parameter

configurations for the decoder to achieve peak performance on different devices.

The steps of this adaptive scheduling and tuning algorithm are as follows. First, the program enquiries the architecture information of the GPU using API functions. In this step, we can get several important parameters of GPU such as number of usable parallel threads, maximum active thread blocks and the amount of shared memories and registers. Then according to the properties of the LDPC codes, such as expansion factor Z of a certain code matrix and the length of the codeword, the program calculates the parameters N_{CW} and N_{MCW} to configure the multi-codeword decoding structure and to determine the size of the computation kernels.

TABLE I
NOTATIONS FOR ADAPTIVE PERFORMANCE TUNING

Parameter	Explanation
Z	Expansion factor. For 802.11n, Z=81; for WiMax, Z=96.
N_{CW}	Number of codewords per macro-codeword
N_{MCW}	Number of macro-codewords
$N_{threads/block}$	Allocated number of threads per thread block
$N_{active\_threads/SM}$	Number of active threads per SM
$N_{active\_TB/SM}$	Number of active thread blocks per SM
$N_{register/threads}$	Amount of registers used per thread
$N_{shared\ memory/threads}$	Amount of shared memory used per thread
$N_{codeword}$	Total number of codewords to be processed

The proposed adaptive performance-tuning method can be summarized as an optimization problem, in which the notations are defined in Table I. Our goal is to maximize the number of active threads per SM ($N_{active\_threads/SM}$) with the smallest $N_{codeword}$, given the following four constraints:

- $N_{threads/block}=Z \times N_{cw} \leq$ max number of threads per thread block;

- $N_{active\_threads/SM}=Z \times N_{cw} \times N_{active\_TB/SM} \leq$ max number of active threads per SM;

- $N_{active\_threads/SM} \times N_{registers/thread} \leq$ amount of available registers per SM;

- $N_{active\_threads/SM} \times N_{shared\_memory/thread} \leq$ amount of available shared memory per SM.

After solving the above optimization problem based on the given constraints, we can get $N_{active\_threads/SM}$, $N_{threads/block}$ and N_{CW}. We can then calculate the number of macro-codewords by using the equation $N_{MCW}=N_{active\_threads/SM}/N_{threads/block} \times N_{SM}$. Then the total amount of codewords is $N_{codeword}=N_{MCW} \times M_{CW}$.

Let us take the WiMAX 1152×2304 LDPC code with expansion factor Z=96 as an example to show how this adaptive scheme works. A CUDA compute capability 2.0 device is used in this example, so the maximum number of threads per block is 1024, the number of active threads per SM is 1536 and the total number of SMs is 14 [10]. We try to maximize $N_{active\_threads/SM}$ under the following constraints:

- $N_{threads/block}=Z \times N_{cw} \leq 1024$;

- $N_{active\_threads/SM}=Z \times N_{cw} \times N_{active\_TB} \leq 1536$.

Solving this problem, we get $N_{active\_TB}=2$, $N_{threads/block}=768$, $N_{cw}=8$. Then, the number of macro-codeword can be calculated via equation $N_{MCW}=N_{active\_threads/SM}/N_{threads/block}\times N_{SM}=28$. The total number of codewords for the peak performance should be $N_{codeword}=N_{MCW}\times M_{CW}=28\times 8=224$. By profiling our CUDA code, we know that 17 registers are used per thread in the first kernel and 11 registers are used per thread in the second kernel. None of the kernels use any shared memory. Based on the configurations, we have 32 available shared memory slots per thread (48 K/1536=32) and 21 available registers per thread (32 K/1536=21). The amount of required shared memory and registers in our kernels is within the limit, so the shared memory and registers are not the bottleneck for this decoding system. The peak performance is bounded by the amount of computational resources (SMs).

V. IMPLEMENTATION RESULTS AND DISCUSSIONS

We conducted experiments to evaluate the performance of the proposed algorithm on the GPU consisting of an NVIDIA GTX470 GPU with 448 stream processors running at 1.215GHz and with 1280MB DDR5 device memory. We also run experiments on an NVIDIA 8600GT GPU with 32 stream processors running at 1.18GHz and with 256MB DDR3 device memory. We use C language and the CUDA programming interface (version 4.0). The graphics card is installed on a PC with an Intel i5-750 CPU and 8GB DDR3 memory.

TABLE II
DATA SETS AND ADAPTIVE PERFORMANCE-TUNING PARAMETER CONFIGURATIONS (FOR NVIDIA GTX470 GPU)

Code type	Code Length	Edges	Threads Per block	Active Threads Per SM	N_{CW}	N_{MCW}
WiFi	1944	6804	729	1458	9	28
WiMAX	2304	8064	768	1536	8	28
Matrix A	3072	15360	768	1536	6	28
Matrix B	6144	30720	768	1536	3	28
Matrix C	12288	55296	512	1536	1	42
Matrix D	24576	98304	1024	1024	1	14

Our experiments include different data sets, which are defined in Table II. The data sets consist of 802.11n WiFi codes, 802.16e WiMAX codes. Matrix A, B, C and D are generated based on WiMAX **H** matrix by increasing the expansion factor. All the codes in the table are irregular LDPC codes, which can provide better error-correcting performance but are challenging to decode due to workload imbalance. The average row weights (number of non-zero elements per row) are seven for all the data sets in this experiment. Number of edges indicates the computational complexity of the LDPC decoding kernel. The number of edges increases in Table II, so the computational complexity from WiFi to matrix D increases.

A. Throughput Performance Analysis

Table II also shows the parameter configurations for each code based on the adaptive tuning scheme. Given code information, the parameters can be easily generated. Table III shows the throughput results of our LDPC decoder implementation to decode different types of codes on GPU. The throughput values are calculated by using the total processing time that includes all kernel execution time on GPU,

TABLE III
THROUGHPUT PERFORMANCE SIMULATION RESULTS (ON GTX470 GPU)

Code type	Code Length	$N_{Code\text{-}word}$	Iterations	Decoding throughput
WiFi	1944	252	10	39.01 Mbps
WiMAX	2304	224	10	48.74 Mbps
Matrix A	3072	168	10	48.96 Mbps
Matrix B	6144	84	10	47.27 Mbps
Matrix C	12288	42	10	49.35 Mbps
Matrix D	24576	14	10	48.21 Mbps

run time of host code, and the time to transfer data between host and device. We can see that the decoder can achieve high throughput, which can greatly speed up the simulation of LDPC decoders.

The performance of WiFi codes and WiMAX codes are different because the properties of each **H** matrix are very different. The distribution of the non-zero entries in the sparse matrix is different. Matrix A, B, C and D have similar performance as WiMAX codes, since their H matrices have similar structures as the WiMAX code. We can also notice that for different kinds of codes, the performance is almost the same. Moreover, experiments show that even if we further increase the workload for all the codes, the throughput does not increase. This fact indicates that we have fully utilized the SMs and achieved peak performance for all the code types by using the configuration parameters generated by our adaptive performance-tuning scheme.

B. Configurability and Scalability

Our massively parallel LDPC decoding implementation has great configurability and scalability. First, the proposed parallel LDPC decoder supports different types of codes, for example, codes with different parity-check matrices and codes with different codeword sizes. Second, our LDPC decoder supports both the Min-Sum algorithm and log-SPA algorithm. Furthermore, thanks to the adaptive performance-tuning scheme, our decoder is able to support devices with different compute capabilities.

C. Discussion of Adaptive Performance-Tuning Scheme

In this section, we evaluate the adaptive performance-tuning scheme by experiments. We use the WiMAX 1152×2304 LDPC code as an example. Since we have already calculated the configuration parameters for peak performance, here we change the workload assigned to GPU by adjusting the parameters N_{CW} and N_{MCW}. The program records the kernel run time and calculates the decoding throughput. Fig.2 shows the relationship between throughput performance and the amount of workload (in the unit of "number of codewords"). Fig.2 shows that the throughput increases very quickly in the beginning; however, after a certain transition point, the throughput almost does not change any more. If we zoom in to take a closer look at the range 0~500 codewords, we can see that when the workload is larger than 200~300 codewords, the throughput curve tends to be flat. Section IV-D shows that decoding more than 224 codewords in parallel could deliver the peak performance. Furthermore, the calculated configurations from Section IV-D and Table II match the simulation results in Fig.2 very well.
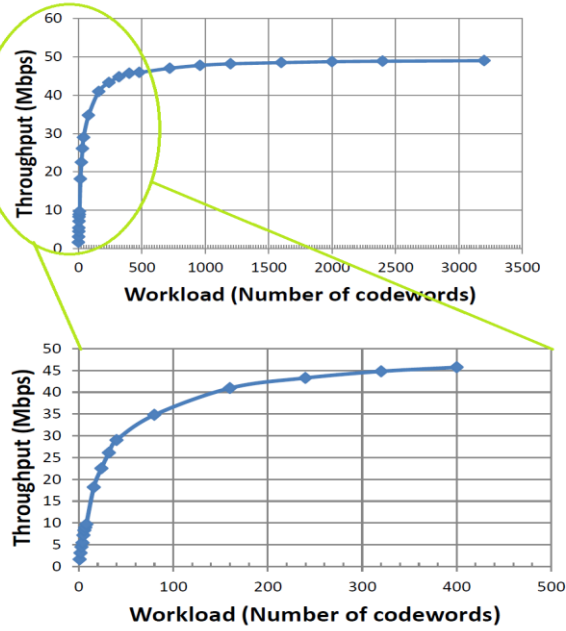
Figure 2: The trend of throughput performace as the workload increases (On an NVIDIA **GTX470** graphics card wich CUDA compute capability 2.0).

Fig.3 shows experimental results on 8600GT GPU. Using our adaptive performance-tuning algorithm, we get the following parameters: N_{CW}=4, N_{MCW}=6 and $N_{codeword}$=24. Fig.3 shows that when we decode 24 codewords, the throughput is very close to the peak performance. This result indicates that our adaptive performance-tuning algorithm works well for older GPUs such as 8600GT whose CUDA compute capability is 1.1 [10].
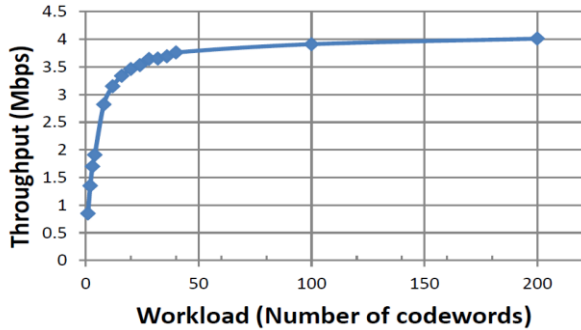

Figure 3: The trend of throughput performace as the workload increases (On an NVIDIA **8600GT** graphics card wich CUDA compute capability 1.1).

The above experimental results prove that our adaptive performance-tuning scheme can successfully calculate and predict the parameters for different GPU devices to achieve peak performance. Since the GPU computation involves other complicated overheads, when practically applying the adaptive performance tuning algorithm to the implementation, we can assign a little more workload than the calculated number to guarantee that we can achieve near-peak performance.

VI. CONCLUSION

Flexibility and scalability are two major advantages of the GPU accelerator for parallel signal processing. However, due to the diversity of LDPC codes and the different hardware architecture of a GPU, it is very challenging to design an adaptive LDPC decoding accelerator on GPU. In this paper, we propose a novel multi-codeword decoding method and an adaptive performance-tuning scheme, which can not only provide high decoding throughput but also deliver good flexibility and scalability. Experimental results show that the implemented LDPC decoding accelerator is able to support different codes with different codeword sizes. With the help of the adaptive performance-turning scheme, the decoder can efficiently achieve peak performance while keeping the decoding latency relatively low. The experimental results also prove the correctness and efficacy of the configuration parameters generated by the adaptive performance-tuning scheme.

REFERENCES

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

[2] INTEL Solutions for Next Generation Multi-Radio Basestation. [Online]. Availible: ftp://download.intel.com/design/intarch/applnots/30745002.pdf

[3] M. Wu, Y. Sun, S. Gupta, and J. R. 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

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

[5] 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.

[6] H. Ji, J. Cho, and W. Sung, "Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPU," 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

[7] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in IEEE 42nd Asilomar Conference on Signals, Systems and Computers, 2008, pp. 171 –175.

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

[9] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A Massively Parallel Implementation of LDPC Decoder on GPU," in Preceding of IEEE Symposium on Application Specific Processors, 2011, pp. 82 -85.

[10] NVIDIA CUDA C programming guide Version 4.0. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[11] 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.

[12] 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.