A Parallel Decoding Algorithm of LDPC Codes using CUDA

Shuang Wang and Samuel Cheng
School of Electrical and Computer Engineering
University of Oklahoma-Tulsa
Tulsa, OK 74135
{shuangwang, samuel.cheng}@ou.edu

Qiang Wu
Soft Imaging, LLC
17000 El Camino Real
Houston, TX 77062
qiang.wu@softimagingllc.com

Abstract—A parallel belief propagation algorithm for decoding low-density parity-check (LDPC) Codes is presented in this paper based on Compute Unified Device Architecture (CUDA). As a new hardware and software architecture for addressing and managing computations, CUDA offers parallel data computing using the highly multithreaded coprocessor driven by very high memory bandwidth GPU. The parallel decoding algorithm, based on CUDA, allows that all bit-nodes or check-nodes work simultaneously, thus provides an efficient and fast way for implementing the decoder.

1. INTRODUCTION

Low-density parity-check (LDPC) code, as an error correcting code (ECC), was first invented by Gallager in 1960 [1][2]. However, researchers did not pay much attention on this kind of codes during the next several decades for the limited ability of computation. Until Mackay and Neal revived it in 1996 [3], LDPC code has rapidly become a popular research topic in telecommunication, because it offers remarkable performance to allow data transmission rates close to the Shannon Limit [4]. In 2003 LDPC code becomes the ECC in the Digital Video Broadcasting (DVB) standard and is also the prevailing competition in the 4th generation of mobile communication system.

LDPC codes are defined by a sparse parity-check matrix. The belief propagation algorithm is a powerful iterative algorithm for decoding the LDPC codes [1]-[3]. During decoding, the probability of each bit of a transmitted codeword being 1 is computed and sent between bit-nodes (representing the codeword bits) and check-nodes (representing the parity constraints). The time complexity of this algorithm increases linearly with the size of parity- check matrix.

Several parallel methods for decoding LDPC codes have been study in recent years [5]-[7]. A parallel architecture for decoding LDPC codes was studied by C. Howland and A. Blaiiksby. In their paper, a prototype soft decision decoder was implemented based on a 1024 bit, rate-1/2 LDPC code [5]. Shimizu *et al.* implemented a parallel LDPC decoder on an FPGA and simulated its decoding performance [6]. Daesun *et al.* presented an efficient highly-parallel decoder architecture using partially overlapped decoding scheme for quasi-cyclic (QC) LDPC codes, which leads to reduction in hardware complexity and power consumption [7]. However, most of these parallel decoding methods designed at the hardware level, thus decoding different codes may require complete changes of hardware architectures. As one of high performance computing platforms, the graphics processor Unit (GPU) offers highly parallel computation, very high memory bandwidth, and a flexible programmable environment. Thus, we can use GPUs to design a parallel decoder for any LDPC code efficiently.

In just a few years, GPUs have evolved into flexible platforms for general computing [9]. Initially, GPUs were programmed by low-level languages [10] which restricted its application as computing workhorses. The release of Cg, a high-level programming language for GPU, facilitated the application of GPU for a general purpose computation [11]. However, Cg is not user-friendly enough, because it requested programmers must have fundamental knowledge on computer graphics for using this high-level programming language. Until recently, NVIDIA releases the Compute Unified Device Architecture (CUDA) [9], programmers can write codes for both CPU and GPU in a similar way by using the instruction set of CUDA [12].

In this paper, we propose a parallel belief propagation algorithm for decoding LDPC Codes by using NVIDIA CUDA programming model. We show how the parallelism naturally appeared during message-passing between bit-nodes and check-nodes can be exploited using the CUDA model. A significant increase of performance is observed when the dimension of parity-check matrix is reasonably large.

This paper is organized as follows. Section 2 presents the basic concept of LDPC codes. In Section 3, the classic belief propagation algorithm for LDPC codes is reviewed. Section 4 describes the parallel belief propagation algorithm based on CUDA. Finally, section 5 and 6 present the performance results and concluding remarks.

2. Low Density Parity Check Codes

A LDPC codes can be defined by an $M \times N$ sparse parity-check matrix H, where M and N represent the numbers of check-nodes and bit-nodes respectively. H is sparse in the sense that the numbers of 1's in each row and in each column must far less than the numbers of rows and columns, respectively. Any codeword of a LDPC code $X = (x_1, x_2, \ldots x_N)$ must satisfy (1).

A Tanner graph as shown in Fig. 1 is an intuitive way to represent a LDPC code. A bit node j, corresponding to column j in H is connected to a check node c_i, corresponding to row i in H, if $H(i,j)=1$.

$$H \cdot X^T = 0 \tag{1}$$

3. The Belief Propagation Algorithm

As a subclass of message passing algorithms, a belief propagation (BP) algorithm for decoding LDPC codes was first presented in Gallager's work [2]. The essence of the BP algorithm is that the probabilities of bit nodes being 1 are exchanged between connected check nodes and bit nodes during each iteration cycle. The BP algorithm can be summarized as follows [13] and is shown in Fig. 1.

1. Denote y_j as the received bit at bit node j. A bit node j computes and sends the prior probabilities (beliefs) of x_j to be 1 ($q_{ji}(1)$) and 0 ($q_{ji}(0)$) according to the observed y_j.

2. A check node i computes the probabilities of x_j to be 1 ($r_{ij}(1)$) and 0 ($r_{ij}(0)$) according to (2) and (3):

$$r_{ij}(0) = \frac{1}{2}\left(1 + \prod_{\substack{H(i,k)=1 \\ k\in(0,N) \\ k\neq j}} \left(1 - 2q_{ki}(1)\right)\right), \tag{2}$$

$$r_{ij}(1) = 1 - r_{ij}(0) \tag{3}$$

These probabilities are then forwarded to the bit node j.

3. The probabilities $q_{ji}(0)$ and $q_{ji}(1)$ of bit node j are updated according to the follows,

$$q_{ji}(0) = K_{ji}\left(1 - P_j\right) \prod_{\substack{H(l,j)=1 \\ l\in(0,M) \\ l\neq i}} \left(r_{lj}(0)\right), \tag{4}$$

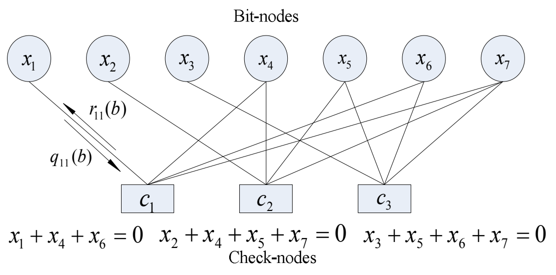


Fig. 1. Graphical representation for LDPC code and probability propagation.

$$q_{ji}(1) = K_{ji} P_j \prod_{\substack{H(l,j)=1 \\ l\in(0,M) \\ l\neq i}} r_{lj}(1), \tag{5}$$

where K_{ji} are constants to ensure that $q_{ji}(0) + q_{ji}(1) = 1$.

Finally, a new estimation $\hat{x}_j$ of bit node j are updated by using (6).

$$\hat{x}_j = \begin{cases} 1 & if\ Q_j(1) > Q_j(0) \\ 0 & otherwise \end{cases}, \tag{6}$$

where $Q_j(b)$ are defined as follows

$$Q_j(0) = K_j\left(1 - P_j\right) \prod_{\substack{H(i,j)=1 \\ i\in(0,M)}} r_{ij}(0), \tag{7}$$

$$Q_j(1) = K_j P_j \prod_{\substack{H(i,j)=1 \\ i\in(0,M)}} r_{ij}(1), \tag{8}$$

where K_j are constants to ensure that $Q_j(0) + Q_j(1) = 1$.

If the current estimate of the codeword $\hat{X} = \{\hat{x}_1, \hat{x}_2 \ldots \hat{x}_N\}$ satisfies (1), the algorithm stops and outputs the estimated codeword $\hat{X}$. Otherwise, go to step 2 unless the maximum number of iterations is reached.

The algorithm described above can be optimized by using log-domain to replace multiplications by additions.

4. CUDA Implement Of Parallel Belief Propagation Algorithm

4.1 General Framework of CUDA

CUDA is a new hardware and software architecture for parallel computing on GPU, which serves as a general computing device bypassing the need of direct access to low-level graphics API. When programmed through CUDA, the GPU is viewed as a computing device capable of executing a very high number of threads in parallel. It operates as a coprocessor to the main CPU (host), which means data-parallel, compute-intensive portions of applications running on the host are off-loaded onto the GPU (device). Both the host and device maintain their own DRAM, referred to as host-memory and device memory, respectively. One can copy data from host to device and vice versa through optimized API calls that utilize the device's high-performance direct memory access engines [9].

The batch of threads that executes a kernel is organized as a grid of thread blocks as illustrated in Fig. 2. A batch of threads can cooperate by sharing data through the fast shared memory ($16K$) and synchronizing executions efficiently. Each thread is identified by its thread ID in each block, and each block is identified by its block ID in each grid. The thread ID is arranged sequentially. For example, the thread ID of a thread with index $(td.x, td.y, td.z)$ in a three dimensional block of size (D_x, D_y, D_z) is $(td.x + td.yD_x + td.zD_xD_y)$. Similarly, for a two-dimensional block of size (D_x, D_y), the block ID of a block of index $(bk.x, bk, y)$ is $(bk.x + bk, yD_x)$ [9].

A grid of thread blocks is executed on device by scheduling blocks for execution on the multiprocessors. The number of blocks that each multiprocessor can process in one batch depends on the property of the device [9].

Device memory can be sorted as read-write per-thread registers, read-write per-thread local memory, read-write per-block shared memory, read-write per-grid global memory, read-only per-grid constant memory and read-only per-grid texture memory. Since the shared memory is embedded on the multiprocessor, it provides a very fast read and write access for threads.

4.2 Parallel decoding on CUDA

In our CUDA implementation of the BP algorithm, a thread is assigned to either a bit node or a check node. The workflow of the algorithm is illustrated in Fig. 4 and is summarized as follows:

1. *Copy the data required for GPU computation from host memory in CPU to global memory in GPU*, so that all threads can access the data in the global memory. These data include a structure array S containing probabilities $q_{ji}(1)$ and $r_{ij}(1)$, two mapping arrays indicating the positions where bit node j and check node i can access $q_{ji}(1)$ and $r_{ij}(1)$ in structure array S, and one array for storing the codeword.

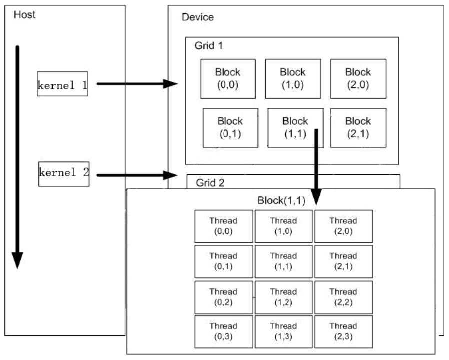


Fig. 2. Thread batching: the host issues a succession of kernel invocations to the device. Each kernel is executed as a batch of threads organized as a grid of thread blocks [9].

2. *Initialize probabilities $q_{ji}(b)$ in parallel using GPU.* The algorithm initializes the probabilities according to the order as shown in Fig. 3 (b). Each thread is assigned to a bit node j and calculates all probabilities $q_{ji}(1)$ with the help of the mapping arrays. For instance, for a binary symmetric channel (BSC) with error probability p, the probability $q_{ji}(1)$ is $1-p$ if the received y_j is 1 and $q_{ji}(1) = p$ otherwise.

3. *Compute current estimated codeword.* Following the order as shown in Fig. 3 (b), N threads are assigned to compute the codeword. If the estimated codeword satisfies (1), go to step 6. Otherwise, proceed to the step 4.

4. *Compute and exchange bit node and check node probabilities*

   4.1 Calculate probabilities $r_{ij}(b)$ at check nodes in the order as shown in Fig. 3(c). A thread is responsible for a particular check node i, and calculates all respective probabilities $r_{ij}(b)$. Moreover, the shared memory on device is used to accelerate the massive number of multiplications in (2).

   4.2 Update the probabilities $q_{ji}(b)$ at bit nodes and estimate $\hat{x}_i$ using $Q_j(b)$ as computed in (7) and (8). The basic idea is the same as step 4.1 except each bit node is assigned to one thread. All threads run and calculate probabilities $q_{ji}(b)$ and $Q_j(b)$ simultaneously.

5. *If doesn't reach the maximum iterations, go to step 3. Otherwise, stop.*

6. *Output codewords.* The decoded codewords are copied from the global memory back to the host memory.

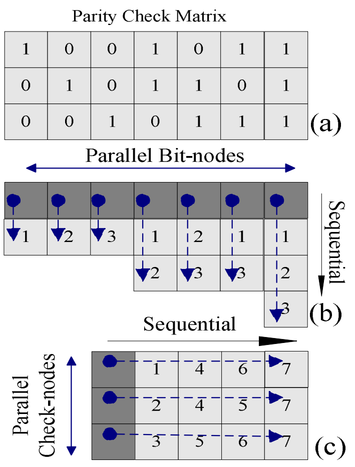
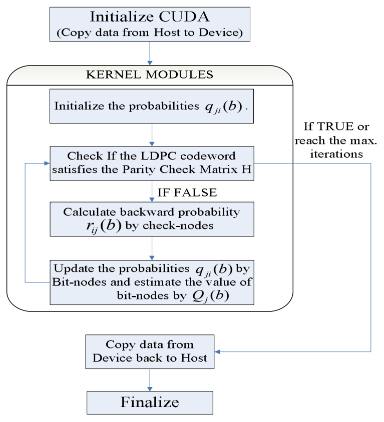


Fig. 3. Schematic figures of a computational block. Fig. 3(a) shows the parity check matrix of Fig. 1. Fig. 3(b) illustrates the parallel computation of bit node, while the parallel computation of check-nodes is shown in Fig. 3(c). In Fig. 3(a) and Fig. 3(b), the numbers indicate the position of 1's in the corresponding parity-check matrix.



Initialize CUDA (Copy data from Host to Device)
KERNEL MODULES
Initialize the probabilities $q_{ji}(b)$.
Check If the LDPC codeword satisfies the Parity Check Matrix H
If TRUE or reach the max. iterations
IF FALSE
Calculate backward probability $r_{ij}(b)$ by check-nodes
Update the probabilities $q_{ji}(b)$ by Bit-nodes and estimate the value of bit-nodes by $Q_j(b)$
Copy data from Device back to Host
Finalize

Fig. 4. The workflow for decoding LDPC codes with CUDA, where the kernel modules means the program executed on GPU

5. Performance Results

In this section, we present the performance results of parallel BP algorithm for decoding LDPC codes based on CUDA. The experiment is performed under an Intel Core Duo 1.6 GHz PC with 2 GB 667MHz DDR2 Memory, and a GPU NVIDIA 8800GT with 512 MB memory installed. The specifications of software are CUDA Toolkit 1.1, CUDA SDK 1.1 and CUDA Driver 169.21 (for Windows XP). In our simulation, 100 different codewords are employed to test performances of C++ codes on CPU and CUDA codes on GPU in decoding LDPC codes. The average iterations used in this paper are defined as divide total iterations for decoding all the codes by the number of codes.

Table I and table II show the decoding performance for code with different parity check matrices. In Table I and II, the sizes of parity check matrix are 2048×4096 and 1024×2048 respectively. For each size, the matrices with 9, 6 and 3 check nodes per row are used for simulation. A significant gain in performance of GPU versus CPU is observed. Moreover, for a given parity check matrix in table I and II, the decoding times are shorter, when larger block sizes are used.

Fig. 5 shows the speedup GPU vs. CPU for three different matrices by using 64 threads per block. In Fig. 5, the speedups increase as the size of check matrix increases for a given number of nodes per row, while the speedups decrease as the number of nodes per row decreases for a given parity check matrix. We can also see that GPU using CUDA is 9 times faster than CPU at the size of 2048×4096 with 9 nodes per row.

The average running time in this paper is defined as $t = T/(N \cdot A)$, where T is running time of GPU or CPU, N is the number of codes and A is the average iterations. Fig. 6 shows the average running time for four different matrices with 6 nodes per row by using 64 threads per block. With increase of the matrices size, we can see that the increase of running time for GPU is linear with a small slope, while the increase for CPU is rapidly. However, as shown in Fig. 6, for small code size such as 256, the performance of GPU is almost the same as CPU. The explanation for this "low" performance is that the GPU needs 400 to 600 clock cycles to read a float number from global memory. However, if there are sufficient independent arithmetic instructions that can be issued while the GPU is waiting for the global memory access, much of this global memory latency can be hidden by the thread scheduler. This fact also verified that CUDA is more suitable for compute-intensive, highly parallel computation [9].

6. Conclusion

This paper proposed a parallel BP algorithm for decoding LDPC codes based on CUDA. CUDA is a new architecture for high-performance computation by using massively multi-threaded GPU with high memory bandwidth. For the decoding of LDPC codes, CUDA offers a highly parallel architecture and significant increase of performance in contrast with traditional computation on CPU. With CUDA, we do not need specific hardware design knowledge to accomplish parallel decoding for LDPC codes. Finally, we conclude that GPU based parallel programming is a very efficient way for the intensive decoding of LDPC codes.

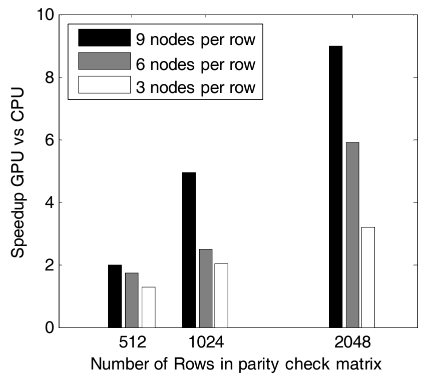


Fig. 5. Speedup GPU versus CPU for three different matrices by using 64 threads per block.

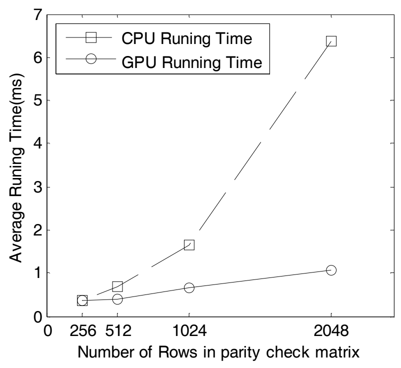


Fig. 6. Average running time for four different matrices with 6 nodes per row by using 64 threads per block.

TABLE I

DECODING PERFORMANCE FOR CODE WITH PARITY CHECK MATRIX SIZE OF 2048 BY 4096

Nodes per Row	Number of Edges	CPU running Time (ms)	GPU running time for different block sizes (ms)				Average Iterations
			8	16	32	64	
9	18432	10002	1517	1444	1138	1113	11.9
6	12288	3059	621	549	528	518	4.8
3	6144	1328	467	432	414	415	5.8

TABLE II.

DECODING PERFORMANCE FOR CODE WITH PARITY CHECK MATRIX SIZE OF 1024 BY 2048

Nodes per Row	Number of Edges	CPU running Time (ms)	GPU running time for different block sizes (ms)				Average Iterations
			8	16	32	64	
9	9216	3428	966	899	894	693	10.7
6	6144	764	360	334	321	307	4.6
3	3072	513	303	285	257	256	5.2

REFERENCES

[1] R. G. Gallager, "Low-Density Parity-Check Codes," IRE Trans. Inform. Theory, vol. IT-8, pp. 21–28, Jan. 1962.

[2] R. G. Gallager, "Low-Density Parity-Check Codes," MIT Press, Cambridge, MA, 1963.

[3] D. J. C. MacKay and Radford M. Neal, Near "Shannon Limit Performance of Low Density Parity Check Codes", Electronics Letters, vol. 32, pp. 1645-1646, July 1996.

[4] F. R. Kschischang, B. J. Frey, and H. A. Loeliger, "Factor graphs and the sum-product algorithm", IEEE Transactions on Information Theory, vol. 47, pp. 498-519, Feb 2001.

[5] C. Howland, A. Blanksby, "Parallel decoding architectures for low density parity check codes", The 2001 IEEE International Symposium on Circuits and Systems, 2001. ISCAS 2001. vol. 4, pp. 742 – 745, May 2001

[6] K. Shimizu, T. Ishikawa, et al, "A parallel LSI architecture for LDPC decoder improving message-passing schedule", Proceedings. 2006 IEEE International Symposium on Circuits and Systems, 2006. ISCAS 2006. pp. 5099-5102, May 2006

[7] Oh. Daesun, K.K Parhi, "Efficient Highly-Parallel Decoder Architecture for Quasi-Cyclic Low-Density Parity-Check Codes", IEEE International Symposium on Circuits and Systems, 2007. ISCAS 2007. pp. 1855 – 1858, May 2007.

[8] Nyland, Lars, Mark Harris, and Jan Prins. 2004. "The Rapid Evaluation of Potential Fields Using Programmable Graphics Hardware." Poster presentation at GP2, the ACM Workshop on General Purpose Computing on Graphics Hardware.

[9] NVIDIA Corporation. 2007. NVIDIA CUDA Compute Unified Device Architecture Programming Guide. Version 1.1. http://www.nvidia.com/ object/cuda_develop.html

[10] P. Warden. Pete's GPU Notes, 2005. http://petewarden.com/notes/archives/ 2005/05/

[11] William R. Mark, R. Steven Glanville, Kurt Akeley, Mark J. Kilgard, "Cg: A System for Programming Graphics Hardware in a C-like Language", Proceedings of SIGGRAPH 2003.

[12] Shams, Ramtin; Barnes, Nick, "Speeding up Mutual Information Computation Using NVIDIA CUDA Hardware" 9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications, pp. 555–560, Dec. 2007

[13] M.J. Bernhard "LDPC Codes – a brief Tutorial", http://users.tkk.fi/~pat/ coding/essays/ ldpc.pdf