RESEARCH　　　　　　　　　　　　　　　　　　　　　　　　　　**Open Access**

Parallel LDPC decoding using CUDA and OpenMP

Joo-Yul Park and Ki-Seok Chung[*]

Abstract

Digital mobile communication technologies, such as next generation mobile communication and mobile TV, are rapidly advancing. Hardware designs to provide baseband processing of new protocol standards are being actively attempted, because of concurrently emerging multiple standards and diverse needs on device functions, hardware-only implementation may have reached a limit. To overcome this challenge, digital communication system designs are adopting software solutions that use central processing units or graphics processing units (GPUs) to implement communication protocols. In this article we propose a parallel software implementation of low density parity check decoding algorithms, and we use a multi-core processor and a GPU to achieve both flexibility and high performance. Specifically, we use OpenMP for parallelizing software on a multi-core processor and Compute Unified Device Architecture (CUDA) for parallel software running on a GPU. We process information on H-matrices using OpenMP pragmas on a multi-core processor and execute decoding algorithms in parallel using CUDA on a GPU. We evaluated the performance of the proposed implementation with respect to two different code rates for the China Multimedia Mobile Broadcasting (CMMB) standard, and we verified that the proposed implementation satisfies the CMMB bandwidth requirement.

Keywords: LDPC, decoder, parallel processing, CUDA, graphic processing unit

1. Introduction

Today, wireless devices transmit and receive high rate data in real-time. The need to provide high transmission rates with reliability is increasing, in order to offer various multimedia services with 4G mobile communication systems. Typical data transmission requirements of 4G mobile communication systems are for 100 Mbps in mobile circumstances and for 1 Gbps in a stationary state [1]. Therefore, powerful correcting codes are becoming indispensable.

The low density parity check (LDPC) code is one of the strongest error correcting codes; it is a linear block code originally devised by Gallager in 1960s [2]. However, it was impossible to implement the code in hardware that was available at that time. About 30 years later, the LDPC code was reviewed by Mackay and Neal [3,4]. They rediscovered the excellent properties of the code and suggested its current feasibility, thanks to the development of communication and integrated circuit technologies. Recently Chung and Richardson [5] showed that the LDPC code can approach the Shannon

limit to within 0.0045 dB. The LDPC code has a smaller minimum distance than the Turbo code, which was regarded as the best channel coding technique before the LDPC started to draw attention. Hence, with almost no error floor issues, it shows very good bit error rate curves. Furthermore, iterative LDPC decoding schemes based on the sum-product algorithm (SPA) can fully be parallelized, leading to high-speed decoding [5]. For these reasons, LDPC coding is widely regarded as a very attractive coding technique for high-speed 4G wireless communications.

LDPC codes are used in many standards, and they support multiple data rates for each standard. However, it is very challenging to design decoder hardware that supports various standards and multiple data rates. Recently, software defined radio (SDR) [6] baseband processing has emerged as a promising technology that can provide a cost-effective, flexible alternative by implementing a wide variety of wireless protocols in software. The physical layer baseband processing generally requires very high bandwidth and thus high processing power. Thus, multi-core processors are often employed in modern, embedded communication devices [7,8]. Also, GPU is often adopted to achieve high computational power [9]. Wide deployment of multi-core

* Correspondence: kchung@hanyang.ac.kr
Department of Electronics, Computer & Communication Engineering, Hanyang University, Seoul, Korea

processors and rapid advances in GPU performance has led to active studies in designing LDPC decoders using GPUs [10-14]. For example, a recent study proposed a technique to utilize GPU to run SPA and described a way to access LLR data [11]. A related study proposed a method for LDPC decoding using Compute Unified Device Architecture (CUDA) [13]. They showed that a GPU could reduce decoding time dramatically.

In this article, we extend this parallelization further in such a way that various standards and code rates can be supported seamlessly. We propose a design which employs both a central processing unit (CPU) and a graphics processing unit (GPU). To support various code rates, the host multi-core CPU reads the H-matrix, and, using OpenMP, it generates address patterns which help the GPU to effectively execute the LDPC decoding in parallel. The LDPC decoding algorithm is written in CUDA [15], which is a parallel computing language developed by NVIDIA, and the CUDA program is executed by the GPU.

The rest of this article is organized as follows. In Section 2, we review LDPC decoding algorithms and parallelization techniques using CUDA. In Section 3, we present the memory structure for CUDA, an address generation method for LDPC decoding, and existing parallelization techniques. In Section 4, the proposed software implementation and performance evaluation results are presented. Section 5 concludes this article.

2. Background
2.1 Review of LDPC decoding algorithms
The LDPC code is a linear block code with a very sparse parity check matrix called H-matrix. The rows and columns of an H-matrix denote parity check codes and symbols, respectively. LDPC codes can be represented by a Tanner graph which is a bipartite graph in which the sides represent check nodes and bit nodes, respectively. Thus, check nodes correspond to the rows of the H-matrix, and bit nodes correspond to the columns of the H-matrix. For example, when the (i, j) element of an H-matrix is '1', the ith check node is connected to the jth bit node of the equivalent Tanner graph. Figures 1 and 2 illustrate an H-matrix and the equivalent Tanner graph for (8, 4) LDPC codes.

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

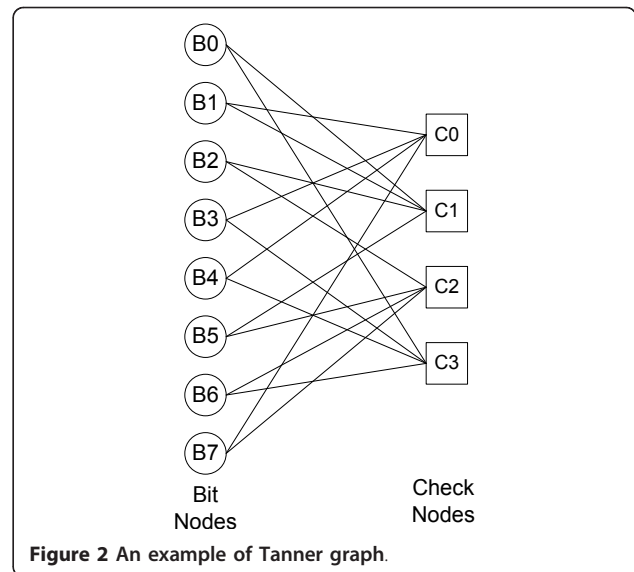**Figure 1 An example of LDPC H matrix**.

Figure 2 An example of Tanner graph.

Most practical LDPC decoders use soft-decisions, because soft-decision decoders typically outperform hard-decision ones. A soft-decision decoding scheme is carried out, based on the concept of belief propagation, by passing messages, which contain the amount of belief for a value being between 0 and 1, between adjacent check nodes and bit nodes. Based on the delivered messages, each node attempts to decode its own value. If the decoded value turns out to contain error, the decoding process is repeated for a predefined number of times. Typically, there are two ways to deliver messages in LDPC decoding. One is to use probabilities, and the other is to use log-likelihood ratios (LLRs). In general, using LLRs is favored since that allows us to replace expensive multiplication operations with inexpensive addition operations.

2.2 Parallelization of LDPC decoding
As explained in Section 2.1, an LDPC decoding algorithm is capable of correcting errors by repeatedly computing and exchanging messages. The amount of computation depends on the size of the H-matrix. However, recently published standards reveal a growing trend that the length of codewords is getting longer as the amount of data transfer is increasing [1]. By the same token, the size of the H-matrix is increasing. For a recent standard, DVB-T2 [16], the length of the codeword is 64,800 bits or 16,200 bits. For China Multimedia Mobile Broadcasting (CMMB) [17], the length is 9,126 bits. The huge size causes both decoding complexity and decoding time to increase. Therefore, it is crucial to distribute the computation load evenly to multiple cores and to parallelize the computation efficiently.

Figure 3 Parallelization of LDPC decoding.

Figure 4 GPU architecture.
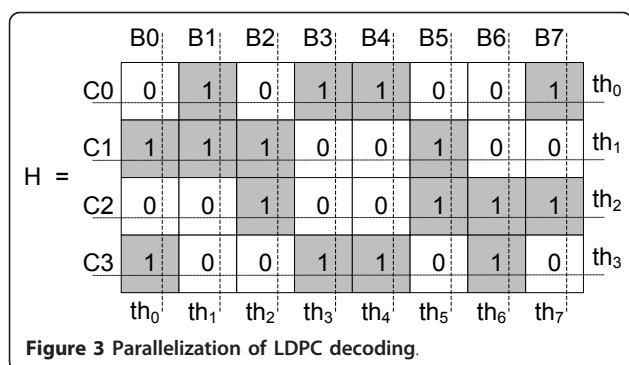
LDPC decoding consists of four general operations: initialization, check node update, bit node update, and parity check. Examining the algorithm reveals that the check node update can be done in parallel, since the rows are uncorrelated with each other. Also, the bit node update on each column can be processed in parallel, because an LDPC decoding algorithm has independent memory accesses among the four types of operations. For example, in the H-matrix in Figure 3, check node operations can process four rows in parallel, and bit node operations can process eight columns concurrently.

This article has three main technical contributions. First, we propose an efficient and flexible technique which can be applied to various protocols and target multi-core platforms, since we propose a solution which employs both CPUs and GPUs. We also introduce an efficient technique to reduce the memory requirement significantly. Next, we propose parallelization techniques for not only check and bit node update operations, but also parity check operations, which will be described in detail in Section 3.

2.3 CUDA programming
CUDA is a GPU software development kit proposed by David Kirk and Mark Harris. One major advantage of CUDA is that it is an extension of the standard C programming language. Hence, those who are familiar with the C/C++ programming language can learn how to program in CUDA relatively easily. Also, CUDA is capable of fully utilizing the fast-improving GPU processing power. Further, NVIDIA hardware engineers actively reflect scientists' opinions as they develop the next generation of CUDA and GPU. For instance, support for double precision computation, error correction capability, and increased shared memory may not be crucial for graphics processing in game applications, but they are important for many scientific and engineering applications. These features have been added in recent versions of CUDA and GPU.
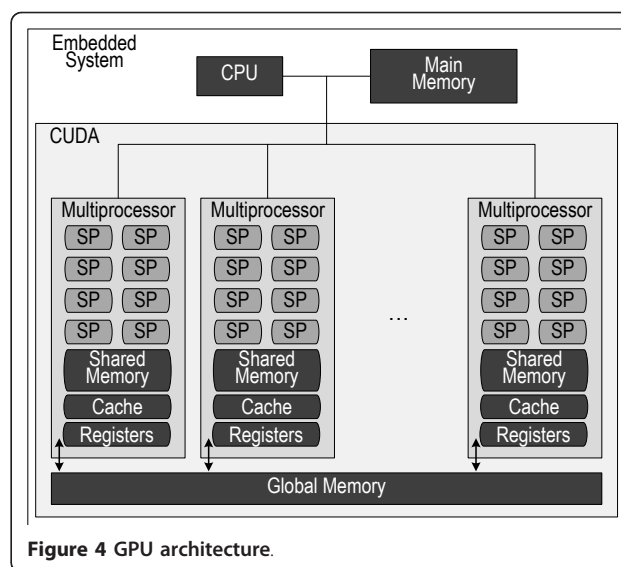
Figure 4 shows the architecture of NVIDIA's 8800GTX. There are 16 multiprocessors, and each multiprocessor has 8 single precision thread processors (SPs). Therefore, the total number of SPs is 128. Each SP can process a block of data with a thread allocation in parallel. However, it is not possible for the CPU and the GPU to share memory space. Thus, the GPU must make a copy of the shared data to its own memory space in advance. If the CPU wants data stored in the memory of the GPU, a similar copy operation must take place. These copy operations incur significant overhead.

Figure 5 shows the relation between a block and a thread in the GPU. A kernel function is executed for one thread at a time. For example, if there were 12 threads in Block (1,1), and there were 6 blocks in a grid, then the kernel function would be executed 72 times. When a function is invoked, the thread and the block index are identified by the thread_idx and block_idx variables, respectively [18,19].

2.4 OpenMP
OpenMP is a set of application program interfaces (APIs) for parallelization of C/C++ or Fortran programs in a shared memory multiprocessing environment. OpenMP has gained lots of attention lately as multi-core systems are being widely deployed in many embedded platforms. Recently, version 2.5 was released. OpenMP is a parallelization method based on compiler directives, in which a directive will tell the compiler which part of the program should be parallelized, by generating multiple threads. Many commercial and noncommercial compilers support OpenMP directives, and thus, we can utilize OpenMP in many existing platforms [20].

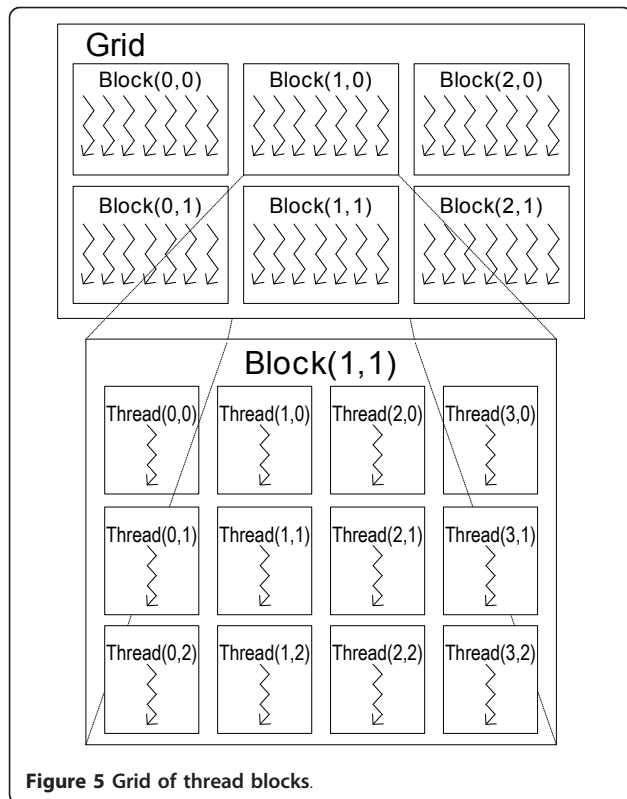OpenMP's parallelization model is based on a fork-join model. Starting with only the master thread,

Figure 5 Grid of thread blocks.

additional slave threads are forked on demand. All threads except for the master one are terminated when execution for a parallel region ends. In this article, we use OpenMP pragmas to parallelize address generation computations. Since only the new address is transferred to the CUDA memory, the memory copy overhead is minimal.(Figure 6)

3. Proposed LDPC decoder

As described above, when the size of H-matrices increases, the amount of computation grows rapidly. This makes it difficult to achieve satisfactory performance in either software- or hardware-only implementations that attempt to support multiple standards and data rates. Therefore, we propose a novel parallel software implementation of LDPC decoding algorithms that are based on OpenMP and CUDA programming. We
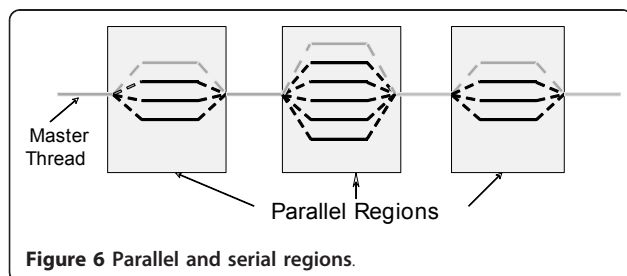

Figure 6 Parallel and serial regions.

will show that the proposed design is a cost-effective and flexible LDPC decoder which satisfies the throughput requirement for various H-matrices and multiple code rates. First, we will show the overall software structure, and next we will explain the parallelization techniques that we propose. (Figure 7)

3.1 Architecture of the proposed LDPC decoder

The overall structure of the proposed LDPC decoder is as follows. We assume that the target platform consists of a single host multi-core processor which can run C codes with OpenMP pragmas and a GPU which can run CUDA codes. To support multiple standards and data rates, multiple H-matrices are stored as files. The host CPU reads the H-matrix for a given standard and signal-to-noise ratio (SNR) constraint. The host CPU then generates an address table of data processed in parallel by the GPU. Generation of the address table is parallelized by OpenMP pragmas. Next, generated address information is transferred to the memory in the GPU. This copy operation takes place only if there is a change in standard, SNR constraint, or code rate.

When signals are received, the host CPU delivers them to the GPU. The GPU executes the proposed LDPC decoding software in parallel. Upon completion of the decoding, decoded bits are transferred to the host CPU. A CUDA API called "CUDA Copy" is used to exchange data between the host and the GPU. The copy
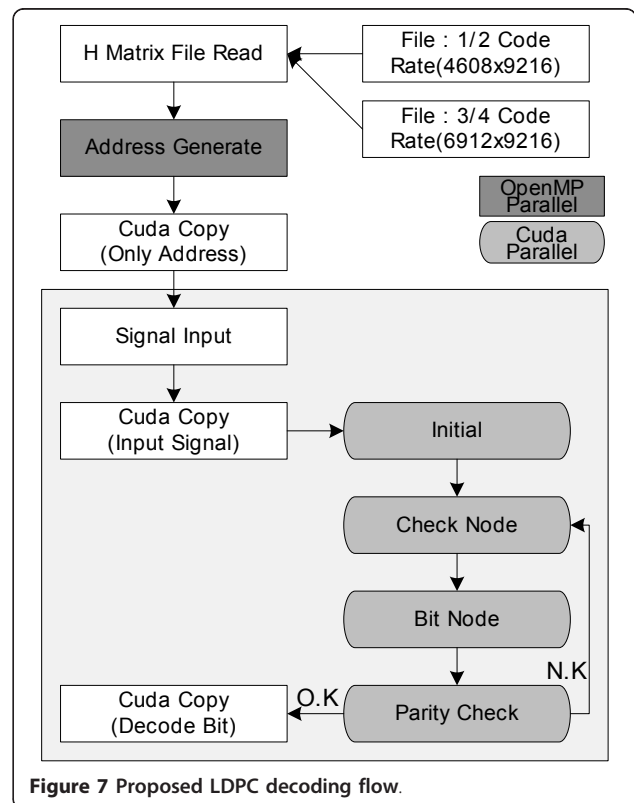

Figure 7 Proposed LDPC decoding flow.

overhead may be significant, so it is crucial to minimize it. It should be noted that in our implementation this copy operation takes place only for generated address transfers, received signal transfers, decoded bit transfers, and configuration (standard, code rates, etc.) changes. Therefore, the copy overhead is not large in our implementation.

3.2 One-dimensional address generation for parallelization

Function 1. New address generator

```
1: {New address :}
#pragma omp parallel for private(i, j)
shared(v_nodes, c_nodes) schedule(static)
2: for i Check Node Num
3: for j weight of check Node
4: for k weight of bit Node

  if(v_nodes[c_nodes[i].index[j]].index[k] = = i
  {
      c_nodes[i].order[j] = k;
      break;
  }

5: end for
6: end for
7: end for
```

We will explain how we generate addresses for CUDA parallelization, using the H-matrix in Figure 1. The H-matrix is stored in a file as a two-dimensional array which contains bit node positions that are necessary for the check node update operation. The first table in Figure 8 shows an example. Since the positions of the LLR values of Bit Node 1 for Check Node 0 and Check Node 1 are different, the bit node order is determined by reading the H-matrix information. We minimize the execution time for this by parallelizing the operation using OpenMP. We use an Intel Quad-Core processor

as the host CPU, and the following algorithm with four threads may be used.

The position of an LLR value is stored in the form of (x, y) where x is the position of a bit node and y indicates that it is the $(y + 1)$th 1 $(0 \leq y \leq (degree - 1))$ of the same bit node. To make it more convenient to parallelize the execution and reduce the memory access time, this (x, y) information is rearranged as a one-dimensional array, as shown at the bottom of Figure 8. The position of the LLR value for (x, y) in the one-dimensional array which is the address for check node computation is easily computed as follows:

$$Laddr = x \times Wb + y \tag{1}$$

where Wb is the degree of bit nodes.

By using this method, when CUDA parallelizes the decoding process, the position of LLR values is obtained by reading memory instead of computing a new address. This improves execution time.

Figure 9 shows the positions of check nodes which are necessary for bit node update operations. Using a similar method to compute $Laddr$, $Zaddr$, which is the position of bit node (x, y) in the one-dimensional array, is computed as follows:

$$Zaddr = x \times Wc + y \tag{2}$$

where x is the position of a check node, and y indicates that it is the $(y + 1)$th 1 $(0 \leq y \leq (degree - 1))$ of the same check node and Wc is the degree of check nodes. Using this one-dimensional address arrangement, the number of memory accesses is minimized in all of the operations of check node updates, bit node updates, initialization, and parity checks.

3.3 Parallel LDPC decoding by GPU

LDPC decoding consists of four parts as shown in Figure 10. The first part is that received LLR values are copied into the location of 1's in the H-matrix. Then,
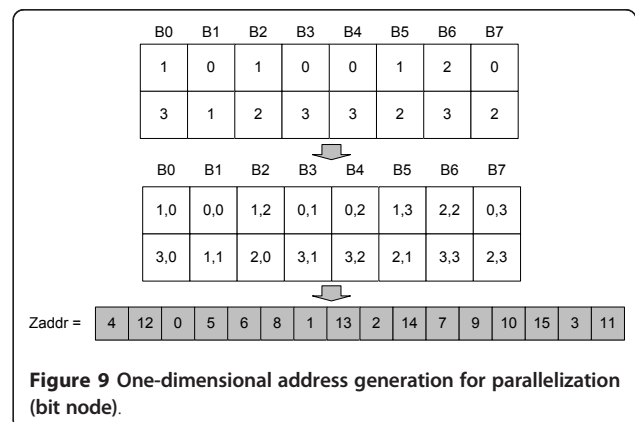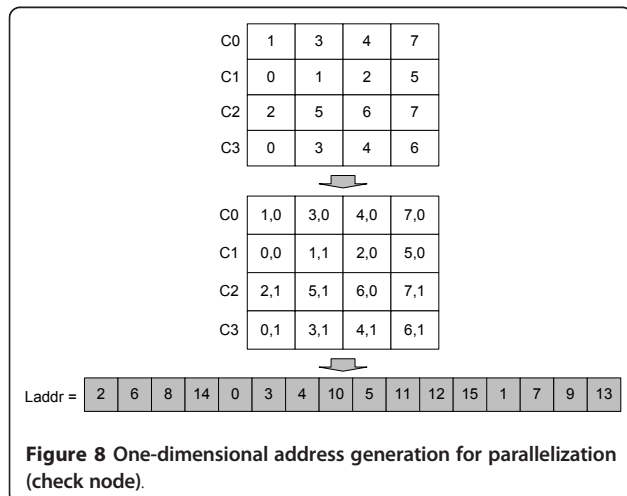

Figure 8 One-dimensional address generation for parallelization (check node).

Figure 9 One-dimensional address generation for parallelization (bit node).
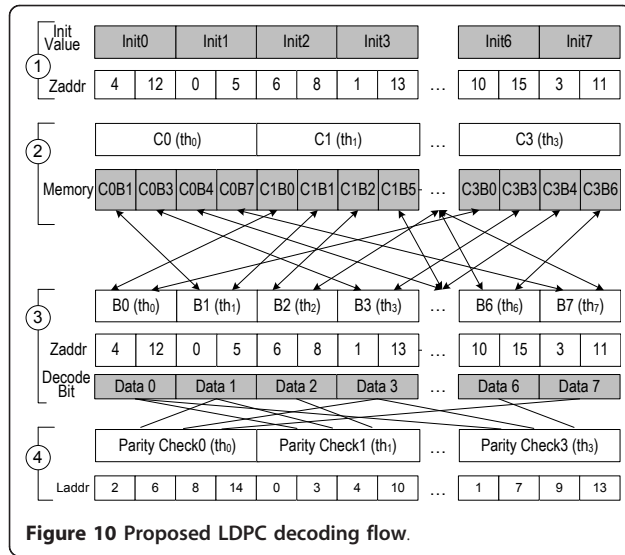
Figure 10 Proposed LDPC decoding flow.

check node update operations and bit node update operations are carried out. Lastly, parity check operations are conducted.

Kernel 1. Initialization kernel

1: {Initilization :}

$$xIndex = blockIdx.x \times blocksize \times Wb + threadIdx.x \times Wb$$
$$Index = blockIdx.x \times blocksize + threadIdx.x$$

2: for i weight of bit Node

$$Memory[Zaddr[xIndex + i]] = Init[Index]$$

3: end for

Kernel 2. Check node update kernel

1: {Check Node Update :}

$$xIndex = blockIdx.x \times blocksize \times Wc + threadIdx.x \times Wc$$

2: for i weight of Check Node

$$Memory[xIndex + i] = CheckNode\_Comp[xIndex + i]$$

3: end for

Kernel 3. Bit node update kernel

1: {Bit Node Update :}

$$xIndex = blockIdx.x \times blocksize \times Wb + threadIdx.x \times Wb$$
$$Index = blockIdx.x \times blocksize + threadIdx.x$$

2: for i weight of bit Node

$$Memory[Zaddr[xIndex + i]] = BitNode\_Comp[Zaddr[xIndex + i]]$$

3: end for

4: $Decode[Index] = BitNode\_Comp$

Kernel 4. Parity check kernel

1: {Parity Check :}

$$xIndex = blockIdx.x \times blocksize \times Wc + threadIdx.x \times Wc$$
$$Index = blockIdx.x \times blocksize + threadIdx.x$$

2: for i weight of Check Node

$$check = Decode[int\ (Laddr[xIndex + i]/Wb)] + check$$

3: end for

$$check = int\ (check\%2)$$

The first initialization step is carried out with a pre-generated *Zaddr*. When the number of signals received equals the number of bit nodes, each received value is copied into the position indicated by *Zaddr*. This task for the H-matrix in Figure 1 can be processed in parallel using eight threads, if we use the GPU in Figure 4.

Second, a check node update operation is conducted after generating as many threads as the number of check nodes. Each thread sequentially reads values from the memory as many times as the degree of check nodes from the memory, and it updates the values and stores them back to the same locations from which they were read.

Third, the bit node update is conducted after generating as many threads as the number of bit nodes. The stored data in memory are arranged in such a way that a check node update operation can effectively be carried out. Therefore, for bit node updates, each thread reads as many values as the degree of bit nodes, using *Zaddr*. Using the input values, bit node updates and determination of a decode bit are conducted. Updated values are stored back to the same locations from which they were read.

Last, parity check operations are parallelized for each check node. A parity check operation is intended to check that all the checking results are 0; this is done using the addresses in *Laddr*. When an address from *Laddr* is divided by the degree of the bit node, we obtain the position of the decode bit for parity check operations.

4. Performance results

Performance evaluation results of the proposed LDPC decoding implementation are presented in this section. H-matrices for the CMMB standard, which is currently used in Digital Multimedia Broadcasting in China, are used for performance evaluation. The length of the codeword in the CMMB standard is 9,216 bits, and two code rates, 1/2 and 3/4, are supported [17].

The execution platform was composed of Intel i5 750 (a Quad-Core CPU with 2.6 GHz) as the host CPU, with 4 GB of DDR3 RAM, Windows XP ServicePack 3 (32 bit). MS Visual Studio 2005 was used as the C

Table 1 LDPC decoding time (second) versus block size (CMMB 1/2, frame 10,000)

SNR	Block size						
	8	16	32	64	128	256	Avg. iter. num
1	224.5	170.0	146.3	*142.3*	142.6	149.2	93.3
1.5	45.8	37.1	30.8	*29.6*	30.5	31.5	18.0
2	2.7	22.4	18.9	*18.0*	18.8	19.5	10.3
2.5	21.3	17.5	14.8	*14.2*	14.8	15.2	7.5
3	17.6	14.5	12.5	*12.0*	12.5	12.8	5.9
3.5	15.3	12.8	10.9	*10.6*	11.0	11.3	4.9
4	13.5	11.3	9.8	*9.6*	9.8	10.0	4.1
Total	365.9	285.5	244.0	*236.3*	240.0	249.6	

Table 3 LDPC decoding time (second) versus SNR value (CMMB 3/4, frame 10,000)

SNR	CUDA & OpenMP	Normal	Only OpenMP	Avg. iter. num
2.5	98.9	2907.1	850.0	29.4
3	23.3	428.9	130.0	18.4
3.5	17.1	270.2	87.2	15.8
4	15.6	197.7	68.2	12.7
4.5	15.2	153.8	57.0	10.1
5	13.1	123.4	45.7	9.4
5.5	12.6	102.0	39.2	8.1
6	14.4	91.0	37.9	6.3
6.5	11.5	70.1	30.5	6.1
7	9.7	55.2	26.3	5.7

compiler. The GPU consisted of NVIDIA GT 8800, 512 MB of memory, and CUDA v2.3.

To optimize the GPU performance, the best block size must be determined. Table 1 summarizes the performance evaluation results for various block sizes. From this evaluation, we determined that the optimal block size was 64 (threads per block).

To evaluate the performance of the proposed design, we compared the performance of three cases: (1) where no parallelization technique was applied, (2) where only parallelization utilizing OpenMP was applied, and (3) where both OpenMP and CUDA were utilized. Report times are the latencies to take decoding 10,000 frames for each SNR value.

Table 2 shows that when both OpenMP and CUDA were utilized, we achieved a speedup of 22 over the case in which no parallelization was applied. When the iteration count increased with low SNR values, the speedup became greater. This is mainly because of the fact that as the iteration count increases, the amount of check and bit nodes' operation will increase. Thus, more parallelization can be done, and accordingly the speedup will become bigger. The effective SNR must be greater than 1.5 dB for the CMMB. Hence, the iteration count is typically greater than 20. When we iterated 20 times,

the decoding performance was 2.046 Mbps, which satisfies the CMMB standard.

Table 3 summarizes the performance for a code rate 3/4 of the CMMB. As the code rate increased, LDPC decoding was successfully finished with at least 2.5 dB. To satisfy the CMMB performance requirement, more than 300 frames per second must be processed. Our results verify that signal reception with at least 3 dB will satisfy this requirement.

To show that our method is satisfactory for multiple standards, we implemented an LDPC decoder for DVB-S2 with a code rate of 2/5; Table 4 summaries the results. DVB-S2 is a standard for European satellite broadcasting. The H-matrix has 16,200 bits of code words and 9,720 bits of an information word. Table 4 shows that for an SNR of 1, we achieved a speedup of 24.5.

Table 5 compares our performance with those of recently reported [11,13]. The H-matrix in our decoding is about twice the size of those previously reported [11,13], which suggests that the performance of our proposed decoder is excellent.

Table 4 LDPC decoding time (second) versus SNR value (DVB-S2 short 2/5, frame 1,000,000)

SNR	CUDA & OpenMP	Normal	Only OpenMP	Avg. iter. num
1	85.6	2097.5	616.9	26.5
1.5	59.3	1370.2	403.0	17.9
2	47.1	1013.5	307.1	13.5
2.5	40.6	788.0	246.3	107
3	35.3	646.6	208.6	8.8
3.5	30.0	533.9	172.2	7.3
4	27.5	447.9	149.3	6.2
4.5	25.7	382.2	127.4	5.3
5	24.7	325.9	108.6	4.6
5.5	22.3	283.6	97.8	4.0
6	23.6	245.4	87.6	3.5
6.5	21.5	217.6	80.6	3.1

Table 2 LDPC decoding time (second) versus SNR value (CMMB 1/2, frame 10,000)

SNR	CUDA & OpenMP	Normal	Only OpenMP	Avg. iter. num
1.5	29.6	656.4	198.7	18.0
2	18.0	354.9	112.5	10.3
2.5	14.2	248.6	81.0	7.5
3	12.0	189.3	64.3	5.9
3.5	10.6	150.6	60.1	4.9
4	9.6	122.6	42.6	4.1
4.5	8.7	100.2	35.9	3.4
5	8.2	85.5	31.3	3.0
5.5	7.5	71.0	26.9	2.6
6	6.9	59.8	23.4	2.2

Table 5 LDPC decoding time speed up (edges/row = 6)

	Proposed algorithm	[11]	[13]
Speed up	22	1.5	6

5. Conclusion

Owing to the multiple standards and diverse device function needs of current digital communications, hardware-only implementations may not be cost-effective. Instead, software implementations of communication protocols using CPUs or GPUs are rapidly being adopted in digital communication system designs. In this article, we have described a software design that implements parallel processing of LDPC decoding algorithms. In our proposal, we use a combination of a multi-core processor and a GPU to achieve both flexibility and high performance. Specifically, we use OpenMP for parallelizing software on a multi-core processor and CUDA for parallel software running on a GPU. Test results show that our parallel software implementation of LDPC algorithms satisfies the CMMB performance and bandwidth requirements.

Competing interests
The authors declare that they have no competing interests.

References
1. Standardization Roadmap for IT839 StrategyVer. (2007)
2. R Gallager, *Low-Density Parity Check Codes* (MIT Press, Cambridge, MA, 1963)
3. D MacKay, R Neal, Near Shannon limit performance of low density parity check codes. IEE Electron Lett. **32**(18), 1645–1646 (1996). doi:10.1049/el:19961141
4. D MacKay, Good error-correcting codes based on very sparse matrices. IEEE Trans Inf Theory **45**(2), 399–431 (1999). doi:10.1109/18.748992
5. SY Chung, GD Forney Jr, TJ Richardson, R Urbanke, On the design of low-density parity-check codes within 0.0045dB of the Shannon limit. IEEE Commun Lett. **5**, 58–60 (2001). doi:10.1109/4234.905935
6. SDR Forum, http://www.wirelessinnovation.org/
7. L Sousa, S Momcilovic, V Silva, G Falcão, Multi-core platforms for signal processing: source and channel coding. *IEEE Press Multimedia Expo*, 1805–1808 (2009)
8. P Kollig, C Osborne, T Henriksson, Heterogeneous multi-core platform for consumer multimedia applications. *Date Conference*, 1254–1259 (2009)
9. CH(K) van Berkel, Multi-core for mobile phones. *Date Conference*, 1260–1265 (2009)
10. G Falcão, L Sousa, V Silva, Massive parallel LDPC decoding on GPU. *13th ACM SIGPLAN*, 83–90 (2008)
11. G Falcão, S Yamagiwa, L Sousa, V Silva, Parallel LDPC decoding on GPUs using a stream-based computing approach. J Comput Sci Technol. **24**(5), 913–924 (2009). doi:10.1007/s11390-009-9266-8
12. G Falcão, L Sousa, V Silva, How GPUs can outperform ASICs for fast LDPC decoding. *Proceedings of the 23rd international conference on Supercomputing*, 390–399 (2009)
13. S Wang, C S, Q Wu, A parallel decoding algorithm of LDPC codes using CUDA. *Signals, Systems and Computers, 2008 42nd Asilomar Conference*, 171–175 (2008)
14. H Ji, J Cho, W Sung, Massively parallel implementation of cyclic LDPC codes on a general purpose graphics processing unit. *Signal Processing Systems, SiPS 2009*, 285–290 (2009)
15. S Lin, DJ Costello Jr, in *Error Control Coding*, 2nd edn. (Prentice Hall, 2004)
16. ETSI EN 302 307, Digital Video Broadcasting (DVB), Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications.
17. GY/T 220.1-2006, Mobile Multimedia Broadcasting Part1.
18. NVIDIA CUDA Development Tools 2.3, Getting Started (July 2009)
19. NVIDIA CUDA C Programming Best Practices Guide, Optimization (July 2009)
20. B Chapman, G Jost, R van der Pas, in *Using OpenMP Portable Shared Memory Parallel Programming* (The MIT Press, 2007)