A Highly Parallel Design for Irregular LDPC Decoding on GPGPUs

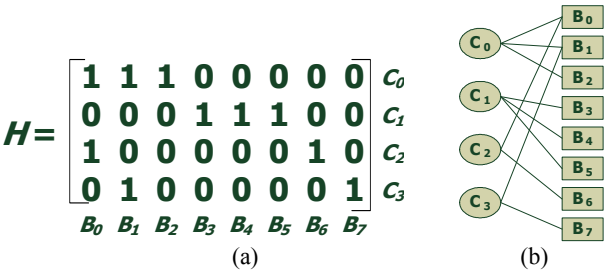
Tsou-Han Chiu, Hsien-Kai Kuo, and Bo-Cheng Charles Lai
National Chiao Tung Univeristy, Hsin-Chu, Taiwan
E-mail: chiupeter.ee95@nctu.edu.tw, hsienkai.kuo@gmail.com, bclai@mail.nctu.edu.tw
Tel: +886-3-5712121

***Abstract*— Low-Density Parity-Check (LDPC) code is a powerful error correcting code. It has been widely adopted by many communication systems. Finding a fast and efficient design of LDPC has been an active research area. This paper proposes a high performance design for irregular LDPC decoding on a general purpose graphic processing unit (GPGPU). A GPGPU is a many-core architecture which enables massively parallel computing. In this paper, a high degree of computation parallelism has been exposed by decoding multiple LDPC code-words concurrently. An innovative data structure is proposed to more efficiently leverage memory coalescing for the irregular data accesses of LDPC decoding. Data spatial locality is maximized by keeping more reusable data within the on-chip cache of a GPGPU. The data communication overhead between a host and a GPGPU is minimized through a single word copy for the convergence check. The experiment results show that the proposed design can achieve up to 55.68X runtime improvement, when compared with a sequential LDPC program on a CPU.**

I. INTRODUCTION

Low-density parity-check (LDPC) is a linear error correcting code which is widely used in communication systems, such as DVB-S2, Wi-Fi (802.11n), WiMAX (802.16e) and 10Gbit Ethernet (802.3an). It can achieve fast decoding and return excellent error correcting performance. LDPC was originally developed by Gallager in 1962 [1], and was rediscovered by Mackay and Neal in 1996 [2] with an attempt to advance coding performance. LDPC codes are capacity-approaching codes, which can achieve the performance that is very close to the Shannon limit [2].

An LDPC code is a linear block code with a sparse parity-check matrix which is usually composed of only a few ones in the matrix. Fig. 1(a) illustrates an example of a *H(n, m)* matrix, where *m* is the row number (also known as check nodes (CNs)), and *n* is the column number (also known as bit nodes (BNs)). The *H* matrix can be transformed into a Tanner graph as shown in Fig. 1(b). A value one in the H matrix represents a connection edge between a BN and a CN. If the number of 1's in each column and each row are the same, it is called a regular matrix. Otherwise, it is called an irregular matrix when the number of 1's in columns or rows are different. Mapping to a Tanner graph, a regular matrix equals the number of edges among all CNs as well as among all BNs. This regular structure has greatly simplified the data access patterns when compared with the irregular behavior. The



(a) (b)

Fig. 1 (a) Sparse parity-check matrix (b) A corresponding Tanner graph

previous research has shown that LDPC decoding with an irregular matrix returns better error correcting ability than a regular one [3]. However, the irregular matrix introduces a complex Tanner graph which requires more sophisticated implementation concerns. This is especially true when being executed on a parallel platform.

This paper focuses on the design and optimization of an irregular matrix on a massively parallel GPGPU system. The widely used min-sum algorithm [3] is adopted to implement a LDPC decoding mechanism. A min-sum algorithm is based on the belief propagation scheme which performs a highly concurrent message passing and updating among nodes BNs and CNs. To fully exploit the high parallelism of the min-sum algorithm, the underlying computation platform needs to support two critical features. First, a large number of processing cores needs to be deployed to enable highly parallel message evaluations. Second, the platform should enable an efficient inter-core communication to pass around the information of variables. The recently emerging Single Instruction Multiple Thread (SIMT) parallel architectures, such as RapidMind [4] and GPGPUs [5, 6], are good target platforms to execute such computation paradigm. However, SIMT architectures usually have three crucial design concerns in order to effectively utilize the potential performance benefit. (1) Alleviate the overhead of data transfers between the host (CPU) and SIMT machine. Ignoring such overhead could impose bottlenecks on the system performance. (2) Enable efficient usage of the limited on-chip data storage resources for concurrent threads. Allocating too much cache space for

particular threads would cause serious L1 cache conflict in a stream multiprocessor. (3) Reduce the number of reads and writes of concurrent threads. A significant amount of data accesses require high memory bandwidth and could become a limit to the performance. Without careful concerns on these three issues, the overall performance on a massively parallel computing platform could be considerably degraded.

This paper proposes a high performance parallel design of the min-sum algorithm on a GPGPU architecture. The proposed design focuses on not only exposing the parallelism, but also novel techniques to avoid the performance pitfalls mentioned in the previous paragraph. The proposed design has three main contributions. First, a significant level of computation parallelism is exposed by decoding multiple LDPC code-words simultaneously. Second, an innovative data structure is proposed to more efficiently leverage memory coalescing for the irregular data accesses of LDPC decoding. This data structure can maximize data spatial locality by keeping more reusable data within the on-chip cache of a GPGPU. Third, the data communication overhead between a host and a GPGPU is minimized by copying only one single word for the convergence check.

The rest of paper is organized as follows. Section II discusses the LDPC decoding algorithm (min-sum algorithm) and related work. Section III introduces the architectures of Nvidia GPGPUs, which will be used as the parallel platform in this paper. Section IV introduces the design of a high performance parallel LDPC decoding scheme. Section V shows the experiment results and Section VI concludes this paper.

II. Related Work

In the last few years, many application-specific integrated circuit (ASIC) solutions of LDPC have been proposed [7, 8]. The min-sum algorithm is one of the widely used scheme due to its low computation complexity [9]. The ASIC solutions leverage the custom design circuits to achieve high throughput. However, the soaring design cost and effort have become the main concern to adopt the ASIC approach.

Parallel execution on multi-core systems has emerged in the recent years as a solution to enhance the sophisticated computation of LDPC decoding. The work in [10] present a straightforward concept to parallelize a Two-Phase Message-Passing (TPMP) and optimized the memory coalescing by interleaving data locations. Abburi [11] proposed a layered algorithm for LDPC decoding which used data packing to expose parallelism and synchronous memory copy to hide the data communication latency. The work in [12] implemented and compared the LDPC decoding on three kinds of multi-core platforms, including CUDA, openMP, and Cell/B.E.. Most of the previous works have addressed parallel design issues for regular LDPC decoding. However, the better error correcting ability makes irregular LDPC decoding an important scheme for the future communication systems. The design techniques proposed in the previous work cannot attain the same benefit when being applied directly to irregular LDPC decoding. This paper focuses on the parallel design of irregular LDPC decoding, and provides effective design techniques to enhance the overall performance on a GPGPU.

III. Architectures of Nvidia GPGPUs

This paper adopts the recently proposed Fermi architecture [13] of Nvidia GPGPUs. The main features of the Fermi architecture are the shared cache memory organization, and fine-grained multi-threading execution scheme. Designers need to take these features into their design concerns in order to fully exploit the benefits provided by the parallel platform.

A. Shared Cache Architecture

The memory system of the Fermi compute architecture is shown in Fig. 2. There are several per Stream Multiprocessor (SM) L1 caches which are of up to 48KB and a unified 768KB L2 cache. Each L1 cache is shared among all CUDA cores in each SM. The unified L2 cache is shared among all SMs. The line size of a L1 cache is 128Bytes for a memory request. The L1 Cache can be optionally disabled. Once it is disabled, every memory request would be sent directly to L2 cache and each memory transaction is 32Bytes.

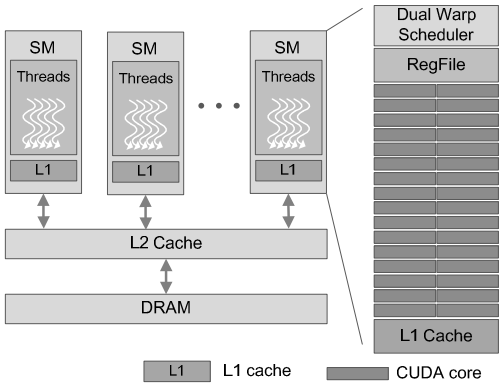


Fig. 2 Nvidia's Fermi compute architecture. The SM is consists of 32 CUDA cores which share a L1 cache. All SMs share the L2 cache and lower memory hierarchy.

B. Fine-Grained Multi-Threading for Numerous Threads

As illustrated in Fig. 2, each SM instantly handles a considerable number of threads. For Fermi, there are 1536 active threads (48 warps) residing in each SM. To fully utilize hardware resources, a shared cache GPGPU also supports multi-threading to context switch to another warp when the current warp encounters a stall such as divergence (branch) or cache misses [13]. In a SM, the dual warp scheduler determines which warp to be scheduled and issue one instruction from it as in Fig. 3(a).

The coalesced memory access technique can reduce number of memory accesses by combining several adjacent memory references into one memory transaction. Even though, it requires highly restrictive limitations on the ordering of memory addresses. In the Fermi, the coalescing technique also works on the cache hierarchy but without any ordering limitations. As illustrated in Fig. 3(b), when the cache line is brought in, threads in a warp can respectively gather the required data without any memory address ordering constraint

if the data reside in the same cache line. Each warp consists of 32 threads and the instructions are executed warp by warp. After, if any thread of the warp has not obtained the required data, the corresponding memory requests would be continually generated until all threads in a warp are served.

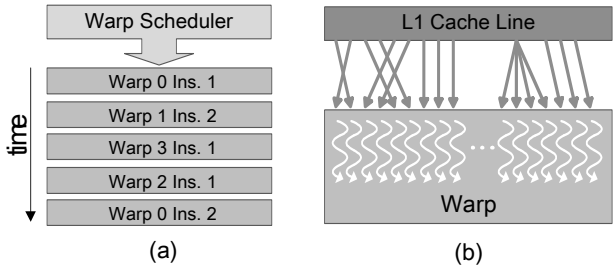


Fig. 3 Coalesced memory access ability and warp scheduling of Nvidia's Fermi compute architecture. (a) The scheduling unit is the instruction from a warp and the scheduler can perform fast a context switch to the next warp if the current warp involves the long stall such as divergence and cache miss. (b) Fermi forms a coalesced memory access without any memory ordering constraint.

IV. Design A High Performance Parallel LDPC Decoding on A GPGPU

This section will introduce a highly parallelized LDPC decoding scheme on a GPGPU. Section A introduces the background of the message passing scheme used in the LDPC decoding. Section B describes the data structure to enhance memory coalescing for irregular LDPC decoding. Section C describes how we divide the workload to GPU. Section D shows how we increase parallelism for decoding code-words. Section E describes our approach to reduce data communication between a host and a device.

A. Introduction to the Message Passing in LDPC Decoding

The LDPC decoding mechanism performs iterative computation and exchanges messages between nodes to achieve a convergence of the algorithm. In a LPDC decoding, the message $m=(m_1,m_2,\cdots,m_m)$ will be encoded by H matrix and produce a code-word $c=(c_1,c_2,\cdots,c_n)$. As described in equation (1), this code-word is mapped onto a mapping sequence $x=(x_1,x_2,\cdots,x_n)$ with a binary phase shift keying (BPSK) modulation. The mapping sequence will be transmitted through an additive white Gaussian noise channel (AWGN). The receiver then receives a sequence $r=(r_1,r_2,\cdots,r_n)$. As illustrated in equation (2), n_i represents the channel noise and is a random variable with zero mean and variance σ^2.

$$x_i=\begin{cases}+1, if & c_i=0\\ -1, if & c_i=1\end{cases} \quad (1)$$

$$r_i=x_i+n_i \quad (2)$$

Fig. 4 shows the pseudo code of the min-sum algorithm for LDPC decoding. Line 1 performs initialization and line 2 specifies the terminal conditions for message passing routines. A message passing will be terminated when it reaches the iteration bound or there is no error bit in a code-word. Line 3 updates the parameter $L_{c_j\to x_i}$, which is also known as the horizontal procedure. Line 4 updates the parameter $L_{x_i\to c_j}$, which is also referred as the vertical procedure. Line 5 calculates the posteriori message. The decoded code-word is determined by a hard decision in line 6. In the end, a syndrome is calculated to check if there exist any error bits in the decoded code-word. The message passing routine will keep running if the terminal conditions are not met.

Pseudo Code of Two-Phase Message-Passing

1: $L_{\text{int}}(x_i)=\frac{2}{\sigma^2}r_i,\forall x_i;\quad L_{c_l\to x_i}(x_i)=0,\forall c_l,x_i;$
$L_{x_i\to c_l}(x_i)=L_{\text{int}}(x_i),\forall c_l,x_i;$

2: **while** (iter < MAX || syndrome $\neq 0$)

3: $L_{c_j\to x_i}(x_i)=(\prod_{l=1,l\neq i}^{d}\text{sgn}(L_{x_l\to c_j}(x_l)))*\min_{l\in BN(j)\setminus\{i\}}(\left|L_{x_l\to c_j}(x_l)\right|)$

4: $L_{x_i\to c_j}(x_i)=L_{\text{int}}(x_i)+\sum_{l=1,l\neq i}^{d}L_{c_l\to x_i}(x_i)$

5: $L_{post}(x_i)=L_{\text{int}}(x_i)+\sum_{l=1}^{d}L_{c_l\to x_i}(x_i)$

6: $\hat{x}_i\begin{cases}1 \Leftarrow L_{post}(x_i)<0\\ 0 \Leftarrow L_{post}(x_i)\geq 0\end{cases}$

7: $H\hat{x}=syndrome;\ \ iter++;$

8: **endwhile**

Fig. 4 Pseudo code of min-sum algorithm

B. Data Structure

The interleaving data structure, shown in Fig. 5(a), is an effective scheme to enhance the data coalescing in a parallel platform [10, 12]. However, it cannot be applied directly to an irregular LDPC decoding. The main reason is that each bit nodes (BNs) or check nodes (CNs) are connected with different number of edges. If a regular data interleaving is applied, each thread would access data with the same array distance. As illustrated in Fig. 5(b), some redundant elements must be padded into the array to keep a regular array distance. This would increase the total data footprint and processors may bring non-used elements into the local cache. In this case, cache space is not well utilized and contention between threads could occur more frequently.

The data structure used in this paper trades off the memory coalescing and data reusing. A fully memory coalescing is implemented for the regular data access, for example the control signals. However, the memory coalescing has been degraded for the irregular data access, for example the computation of $L_{c_j\to x_i}$ and $L_{x_i\to c_j}$. To compromise this disadvantage, for the irregular part, each SM will read several cache lines into L1, and utilize as much data as it can before the cache line is evicted. This scheme can benefit from the memory hierarchy of the Fermi architecture. Fermi has a L2 cache and each SM can access data from it. By allocating

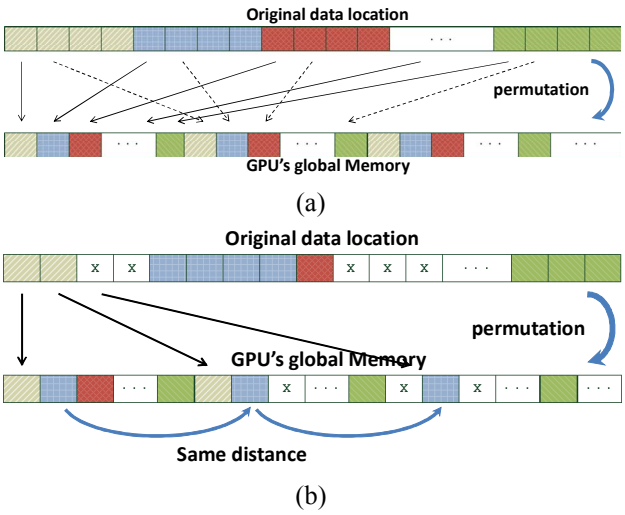


Fig. 5 (a) Interleaving data layout for regular data access. (b) Interleaving data layout for irregular data access.

more useful data in L1, the data locality at L1 is enhanced and the redundant data stored in L2 is reduced at the same time. In this case, L2 can store more useful data for the future usage. Therefore L1 cache has a higher data hit rate at L2 and can bring the required data from L2 with a shorter latency than from the global memory.

Fig. 6 illustrates an example of the proposed data structure for updating $L_{c_j \to x_i}$. *Lxc* array records all $L_{x_i \to c_j}$ messages for each CN. Two arrays, *LxcBegin* and *LxcSize*, are implemented to help threads get correct elements from the *Lxc* array. *LxcBegin* array records the start point of *Lxc* array for each CN. *LxcSize* array records the number of connected BNs for each CN. *LcxPosition* array records the writing index for *Lcx*.

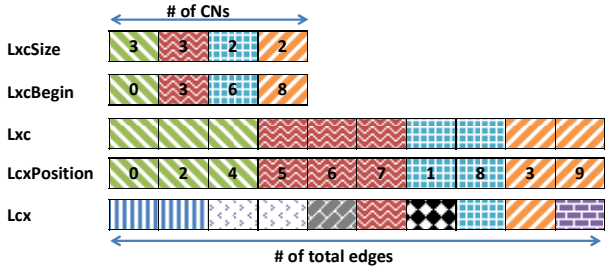


Fig. 6 Data structure for updating Lcx on GPGPU

C. *Create One Thread Per Node*

According to the pseudo code in Fig.4, there are two phases of message updating. The implementations for these two phases are similar. One thread is created for one node, and updates all the messages for this node. Take the first phase (horizontal procedure) as an example. All the threads will first concurrently read one element from *LxcBegin* and *LxcSize*. Then a thread can know where to start reading *Lxc* array and also know how many *Lxc* elements need to be read. Finally, the *LcxPosition* is used to get the updating index and updates new L_{cx} values into the *Lcx* array. Fig. 7 shows an example when threads are updating the L_{cx} of the first connected BN.

From Fig. 7, reading the arrays of *LxcBegin* and *LxcSize* is fully coalesced. Reading arrays from *Lxc* and *LcxPosition* is partial coalesced. A cache line contains multiple words. Therefore when a thread reads the first element from *Lxc*, it will also take the consecutive array elements into L1 cache. This particular thread will also need these consecutive elements soon to collect all the $L_{x_i \to c_j}$ in the same CN. With this scheme, cache contention between threads can be mitigated by keeping a proper number of threads in one block.

D. *Decode Multiple Code-words Concurrently*

The data-parallelism is limited by the total code length of a LDPC decoding. A short code length could provide only limited parallel computing tasks, and result in a low utilization of the GPGPU computation resource. To enhance the utilization rate, this paper develops a kernel to decode multiple code-words concurrently. A suitable number of concurrent code-words can be decided based on the actual computing resources of a GPGPU and data usage of the LDPC algorithm.

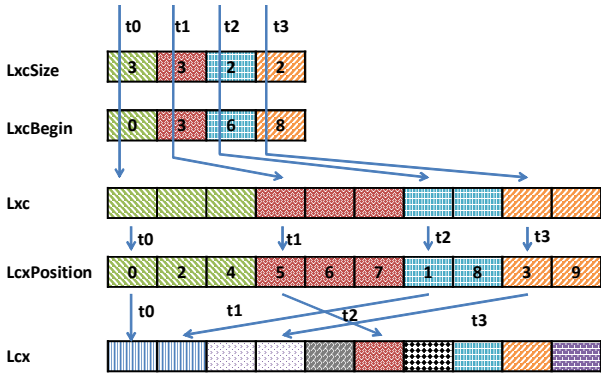


Fig. 7 An example for threads update their first connected edge.

E. *Reduce Memory Copy between Host and Device*

A large number of memory copies between a GPGPU and a host can significantly hurt the overall execution time. Reduce communication between a host and a device is a critical issue for programmers. For the implementation of LDPC decoding in this paper, only the Lint array is copied from the host to the GPGPU. After the initialization, each thread will read Lint and write the results into *Lxc* array. During the message passing phase, the decoding performed by the two kernels is executed within the GPGPU. The functions for calculating the posterior message and hard decision are also executed on the GPGPU. Each iteration only copies an integer syndrome back to the host to check the terminal condition. Fig. 8 illustrates the data communication and algorithm flow. We copy one array before the message passing routine and no more data copies would be performed from a host to a device in the rest of the routine. Only one integer is copied from the GPGPU to the host for each iteration. In this way, the data communication between the host and the device is reduced significantly.

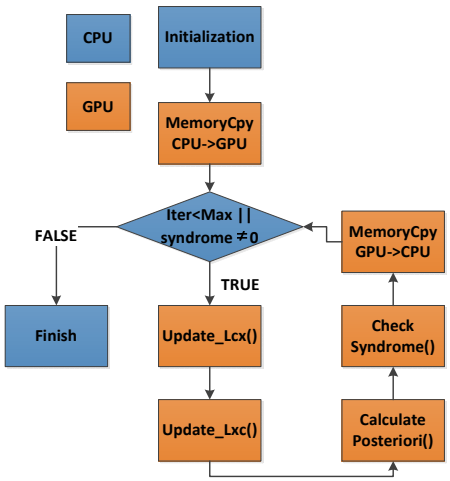


Fig. 8 A flow for illustrating algorithm flow and data communication.

V. Experimental Results

Our work is implemented with CUDA environment and executed on NVidia's Tesla C2050. All the experiments are conducted on a 2.83GHz Intel®Core(TM)2 CPU sprinting CentOS 5.5 with 8GB of memory. Table I shows the results for the benchmark of IEEE802.15.3c (672,336) [14]. IEEE802.15.3c (672,336) uses an irregular matrix, which has 672 CNs and 336 BNs with a total of 2184 edges. The second column lists the results of the reference implementation on a CPU. The third column shows the parallel design which decodes one code-word on a GPGPU (GPU_SC). The fourth column lists the parallel design which decodes sixteen code-words concurrently on a GPGPU (GPU_MC).

In our experiments, the computation parallelism is limited by the number of nodes in a LDPC decoding scheme. There are total 640 threads and 10752 threads for GPU_SC and GPU_MC respectively. GPU_MC has higher parallelism, because it decodes sixteen code-words at the same time. Since the NVidia's Tesla C2050 has 448 cores, decoding sixteen code-words concurrently (10752 concurrent threads) has already approached the maximum resource utilization. Increasing more threads did not reveal a better performance. However, the authors certainly believe that decoding more code-words can provide better performance scalability when the number of cores in a GPGPU increases in the future generations.

VI. Conclusion

This paper proposed a high performance design of irregular LDPC decoding on a general purpose graphic processing unit (GPGPU). A high degree of computation parallelism has been exposed by decoding multiple LDPC code-words concurrently. An innovative data structure is proposed to take advantages of GPGPU architectures and data coalescing. The associated data management approach has further minimized the data communication overhead between a host and a GPGPU. The experiment results show that the proposed design can achieve up to 55.68X runtime improvement, when compared with a sequential LDPC program on a CPU.

TABLE I
RUN TIME FOR BENCHMARK *IEEE802.15.3C* (672,336)

	CPU	GPU_SC	GPU_MC
Iter5	89.12 (sec)	18.05 (sec)	2.39 (sec)
Iter10	178.39 (sec)	23.87 (sec)	3.75 (sec)
Iter20	355.82 (sec)	43.65 (sec)	6.39 (sec)
Max Speedup		**8.15**	**55.68**
Average Speedup		**6.85**	**46.85**

Acknowledgment

This work is sponsored by Nation Science Council under grant NSC 101-2220-E-009 -037.

References

[1] R. Gallager, "Low-density parity-check codes," *Information Theory, IRE Transactions on,* vol. 8, pp. 21-28, 1962.
[2] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electronics Letters,* vol. 32, p. 1645, 1996.
[3] W. E. Ryan, "An introduction to LDPC codes," in *CRC Handbook for Coding and Signal Processing for Recording Systems*, B. Vasic, Ed., ed. Boca Raton: FL: CRC, 2004.
[4] D. R. Johnson, *et al.*, "Rigel: A 1,024-Core Single-Chip Accelerator Architecture," *Micro, IEEE,* vol. 31, pp. 30-41, 2011.
[5] GPGPU, "http://gpgpu.org/."
[6] NVIDIA_CUDA_Homepage, "http://www.nvidia.com/object/cuda_home_new.html."
[7] Z. Tong and K. K. Parhi, "Joint (3,k)-regular LDPC code and decoder/encoder design," *Signal Processing, IEEE Transactions on,* vol. 52, pp. 1065-1079, 2004.
[8] J. Dielissen, *et al.*, "Low cost LDPC decoder for DVB-S2," in *Design, Automation and Test in Europe, 2006. DATE '06. Proceedings*, 2006, pp. 1-6.
[9] P. Li and W. K. Leung, "Decoding low density parity check codes with finite quantization bits," *Communications Letters, IEEE,* vol. 4, pp. 62-64, 2000.
[10] G. Falcao., *et al.*, "How GPUs can outperform ASICs for fast LDPC decoding," presented at the Proceedings of the 23rd international conference on Supercomputing, Yorktown Heights, NY, USA, 2009.
[11] K. K. Abburi, "A Scalable LDPC Decoder on GPU," in *VLSI Design (VLSI Design), 2011 24th International Conference on*, 2011, pp. 183-188.
[12] G. Falcao, *et al.*, "Massively LDPC Decoding on Multicore Architectures," *Parallel and Distributed Systems, IEEE Transactions on,* vol. 22, pp. 309-322, 2011.
[13] Nvidia. *NVIDIA's Next Generation CUDA Compute Architecture: Fermi*. Available: http://www.nvidia.com.tw/object/LO_gtx400_whitepaper_architecture_tw.html
[14] T. Baykas, *et al.*, "IEEE 802.15.3c: the first IEEE wireless standard for data rates over 1 Gb/s," *Communications Magazine, IEEE,* vol. 49, pp. 114-121, 2011.