4.7-Gb/s LDPC Decoder on GPU

Jinyang Yuan and Jin Sha, *Senior Member, IEEE*

***Abstract*— Graphics processing units (GPUs) are well-suited for decoding low-density parity-check (LDPC) codes because of their massive parallelisms and high flexibility. Implementations of high-throughput GPU-based LDPC decoders are described in this letter. By applying a novel message updating scheme and reducing shared memory consumption, the flooding-based and layered-based decoders outperform the corresponding state-of-the-art designs, with 55% and 34% improvements of normalized throughputs. On a single GeForce GTX 1080 Ti GPU, the two decoders achieve 4.77- and 3.67-Gb/s throughputs by incorporating early termination criterion. Throughputs of the GPU-based decoders are comparable with an implementation on the largest density field-programmable gate array.**

***Index Terms*— LDPC codes, high throughput, decoding, GPU.**

I. Introduction

LOW-DENSITY parity-check (LDPC) code was proposed by Gallager in 1962 [1], and rediscovered by Mackay and Neal in the late 1990s [2]. Field-programmable gate array (FPGA) and application-specific integrated circuit (ASIC) are usually applied to implement high-throughput LDPC decoders. Although decoders implemented on FPGA and ASIC achieve tremendous performance, they are less flexible, more costly, and demand much more efforts to design than software implementations on processors like central processing unit (CPU) and graphics processing unit (GPU). GPUs have massive parallelisms and are widely utilized in high-performance computing (HPC). They are optimized for throughput, and the latency is hidden with concurrent computations. As a result, GPUs are well-suited for computationally intensive tasks like decoding LDPC codes.

In this letter, we present two designs of GPU-based LDPC decoders. By applying a novel message updating scheme, the flooding-based decoder achieves 4.7 Gb/s throughput. Its normalized throughput is 55% higher than the state-of-the-art design. By reducing shared memory consumption, the layered-based decoder outperforms the existing implementations with 34% improvement of normalized throughput.

II. LDPC Codes and Decoding Algorithms

LDPC code is characterized by its parity-check matrix (PCM) **H**. In Tanner graph, each column of **H** corresponds to one variable node (VN), and each row of **H** is represented by one check node (CN). Sum-product algorithm (SPA) and min-sum algorithm (MSA) are the two most well-known iterative decoding algorithms. SPA leads to better error correcting performance but is more complex. Normalized and offset MSA [3] improve the error correcting performance of MSA while keeping the low complexity property.

A. Quasi-Cyclic LDPC Codes

Quasi-cyclic (QC) LDPC code is one type of structured LDPC codes, and various code construction methods based on progressive edge-growth (PEG) [4], protographs [5], finite geometries and fields [6], etc. have been proposed to obtain high performance QC-LDPC codes. PCM of a QC-LDPC code can be represented by a much smaller matrix **B** called base matrix. Each entry in **B** represents a submatrix that is either a circulant permutation matrix (CPM) or a zero matrix (ZM).

B. Message Passing Scheduling

Flooding, layered [7] and shuffled [8] schedulings are three commonly used message passing schedulings. Flooding scheduling updates variable-to-check (V2C) and check-to-variable (C2V) messages in two different phases. Layered and shuffled schedulings divide PCM into several groups in terms of rows and columns, respectively, and messages are updated in units of groups. Flooding scheduling has the highest degree of parallelism, and layered scheduling has the fastest convergence speed. Shuffled scheduling reduces the hardware complexity when LDPC codes have large row weights.

C. Normalized Min-Sum Algorithm

Normalized Min-Sum Algorithm (NMSA) is chosen as the decoding algorithm in this work. Let $M(v_j)$ be the set of CNs connected to the v_jth VN and $N(c_i)$ be the set of VNs connected to the c_ith CN. $L_{v_j}^{init}$ denotes the log-likelihood ratio (LLR) of the v_jth bit in the received codeword. $L_{v_j c_i}^{(n)}$ and $L_{c_i v_j}^{(n)}$ are the V2C message from the v_jth VN to the c_ith CN and C2V message from the c_ith CN to the v_jth VN at the nth iteration, respectively. $L_{v_j}^{app}$ stands for the a posteriori probability (APP) value of the v_jth bit in the decoded codeword. The following shows the procedures of NMSA.

1) Initializations:

$$L_{v_j}^{app} = L_{v_j}^{init} \qquad L_{c_i v_j}^{(0)} = 0 \tag{1}$$

2) Updates of V2C Messages:

$$L_{v_j c_i}^{(n)} = L_{v_j}^{app} - L_{c_i v_j}^{(n)} \tag{2}$$

3) Updates of C2V Messages:

$$L_{c_i v_j}^{(n+1)} = \alpha \prod_{v' \in N(c_i) \backslash v_j} \operatorname{sgn}(L_{v' c_i}^{(n)}) \min_{v' \in N(c_i) \backslash v_j} \left| L_{v' c_i}^{(n)} \right| \tag{3}$$

Manuscript received October 16, 2017; accepted November 25, 2017. Date of publication December 1, 2017; date of current version March 8, 2018. This work was supported by the National Key R&D Program of China (No. 2016YFA0202102), the National Natural Science Foundation of China (No. 61370040), the Project on the Industry Key Technologies of Jiangsu Province (BE2017153) and PAPD. The associate editor coordinating the review of this paper and approving it for publication was Z. Ma. *(Corresponding author: Jin Sha.)*

The authors are with the School of Electronic Science and Engineering, Nanjing University, Nanjing 210093, China (e-mail: shajin@nju.edu.cn).

Digital Object Identifier 10.1109/LCOMM.2017.2778727

4) Updates of APP Values:

$$L_{v_j}^{app} = L_{v_j}^{llr} + \sum_{c_i \in M(v_j)} L_{c_i v_j}^{(n+1)} \tag{4}$$

The α appeared in updates of C2V messages is the normalized factor. If set to 1, NMSA will turn into MSA.

III. Implementations of LDPC Decoders on GPU

LDPC decoders are developed using CUDA (compute unified device architecture) parallel computing platform. To illustrate the proposed designs more concretely, the described decoders are for (1944, 972) LDPC code in 802.11n. Without loss of generality, the design considerations mentioned are applicable to all types of LDPC codes and CUDA-enabled GPUs. The flooding-based and layered-based architectures are presented. Decoders with shuffled scheduling can be designed with little modifications to either of the two architectures.

A. CUDA-Enabled GPU

Streaming multiprocessors (SMs) are the building blocks of CUDA devices. Each SM consists of several streaming processors (SPs) and a small amount of high-speed memory. Besides the on-chip memory residing on SMs, a CUDA device also possesses gigabytes of on-board memory. The 4 types of memory utilized in the implementations of LDPC decoders are registers, shared memory, global memory and constant memory. Registers and shared memory are on-chip, and the rest two types are on-board. Constant memory has faster read speed than global memory because it is read-only.

Appropriate memory access patterns are crucial to high performance. Shared memory suffers from the bank conflict problem, and coalescing should be taken into consideration when using global memory. A good practice is to let successive threads access consecutive memory addresses.

B. Memory Organization

1) Base Matrix: (1944, 972) LDPC code in 802.11n standard is quasi-cyclic. Each row of the base matrix contains more than 2/3 ZM entries. Because ZMs do not participate in the updates of V2C and C2V messages, skipping unnecessary accesses of these entries improves the throughputs of the decoders. As shown in Fig. 1a, allocating two arrays to store locations and shift values of CPMs separately is a viable approach. A row is padded with -1 if it contains less than 8 CPMs. These two arrays are stored in constant memory because they are read-only and fit in the cache.

2) LLR and APP Values: LLR values are stored in global memory instead of constant memory although they are read-only, because LLR values of all blocks do not fit in the 64 KB constant memory. APP values are kept in shared memory during the iterative procedure and copied to global memory after the final iteration. In designs that allocate threads in terms of VNs, APP values may also be stored in registers.

	0	1	2	3	4	5	6	7
0	0	4	6	8	10	12	13	-1
1	0	2	4	8	9	13	14	-1
2	0	4	5	8	9	14	15	-1
3	0	1	4	7	8	15	16	-1
4	0	3	4	7	8	16	17	-1
5	0	4	6	8	11	17	18	-1
6	0	1	2	6	8	12	18	19
7	0	4	5	8	10	19	20	-1
8	0	4	5	8	11	20	21	-1
9	1	3	4	8	9	21	22	-1
10	0	1	3	4	10	22	23	-1
11	0	2	4	7	8	11	12	23

	0	1	2	3	4	5	6	7
0	57	50	11	50	79	1	0	-1
1	3	28	0	55	7	0	0	-1
2	30	24	37	56	14	0	0	-1
3	62	53	53	3	35	0	0	-1
4	40	20	66	22	28	0	0	-1
5	0	8	42	50	8	0	0	-1
6	69	79	79	56	52	0	0	0
7	65	38	57	72	27	0	0	-1
8	64	14	52	30	32	0	0	-1
9	45	70	0	77	9	0	0	-1
10	2	56	57	35	12	0	0	-1
11	24	61	60	27	51	16	1	0

(a) Original locations and shift values

	0	1	2	3	4	5	6	7
0	0	4	8	6	10	12	13	-1
1	2	0	4	8	9	13	14	-1
2	14	15	0	4	8	5	9	-1
3	7	16	15	0	4	8	1	-1
4	16	3	17	7	0	4	8	-1
5	8	6	11	17	18	0	4	-1
6	1	8	2	12	6	18	19	0
7	4	5	8	10	20	19	0	-1
8	0	4	5	8	11	20	21	-1
9	3	1	4	21	8	9	22	-1
10	22	0	23	4	1	3	10	-1
11	11	2	7	23	12	8	0	4

(b) Shifted locations

Fig. 1. Arrays of compressed base matrix. The more the same values appear in each column of locations arrays, the darker the entries are.

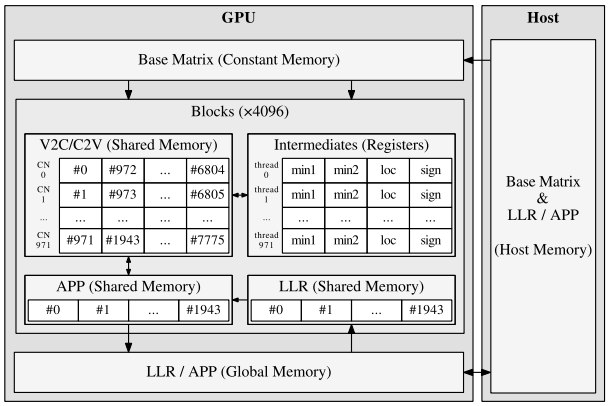


Fig. 2. Architecture of the flooding-based LDPC decoder.

3) V2C and C2V Messages: Considering the frequent updates and dynamically indexed accesses of V2C and C2V messages, shared memory is chosen to store them. Computations of C2V messages in NMSA have another representation. For each c_ith CN and V2C messages $L_{v_j c_i}^{(n)}$ connected to it, let $min1_{c_i}^{(n)}$, $min2_{c_i}^{(n)}$ and $loc_{c_i}^{(n)}$ be the first minimum, second minimum, and location of the first minimum of $\left|L_{v_j c_i}^{(n)}\right|$. $S_{c_i}^{(n)}$ denotes the product of signs of all $L_{v_j c_i}^{(n)}$, and $s_{v_j c_i}^{(n)}$ represents the sign of $L_{v_j c_i}^{(n)}$. Equation (3) becomes

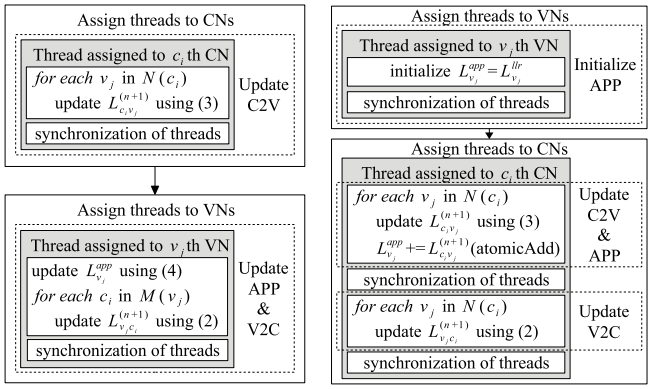
$$L_{c_i v_j}^{(n+1)} = \begin{cases} \alpha \cdot S_{c_i}^{(n)} \cdot s_{v_j c_i}^{(n)} \cdot min2_{c_i}^{(n)} & v_j = loc_{c_i}^{(n)} \\ \alpha \cdot S_{c_i}^{(n)} \cdot s_{v_j c_i}^{(n)} \cdot min1_{c_i}^{(n)} & \text{otherwise} \end{cases} \tag{5}$$

Storing messages in the transformed representation reduces memory consumption at the cost of redundant computations of C2V messages.

C. Flooding-Based Decoder

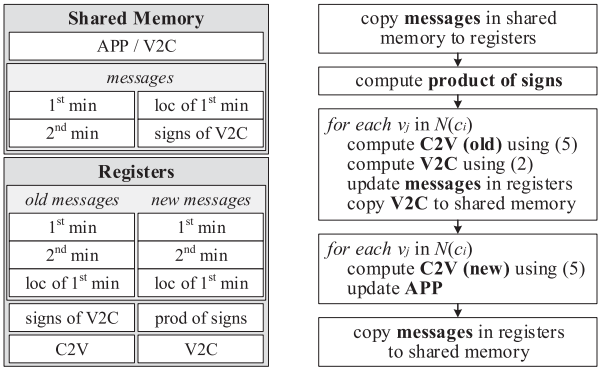
The architecture of the flooding-based decoder is illustrated in Fig. 2. Intermediates used to compute C2V messages are stored in registers. The main differences between the proposed and existing flooding-based decoders (e.g. [9]–[11]) are threads assignments and synchronization methods applied during the updates of messages. A novel method to store V2C/C2V messages in registers instead of shared or global memory is also presented. The details are as follows.

1) Threads Assignments: V2C and C2V messages are accessed more frequently than LLR and APP values. Hence, optimizing memory access patterns for them is more beneficial. Existing flooding-based decoders update C2V messages

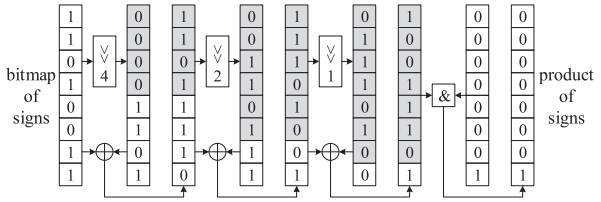


(a) Previous work (b) Proposed

Fig. 3. Message updating schemes of flooding scheduling.



(a) On-chip memory organization and message updating procedure



(b) Computation of product of signs

Fig. 4. Implementation details of the layered-based decoder.

in terms of CNs and V2C messages in terms of VNs (Fig. 3a). The differences of threads assignments make it hard, if not impossible to design a message storage scheme that is optimized for both phases. Assigning threads to CNs in both phases (Fig. 3b) results in better memory access pattern.

2) Atomic Operations: Modifying the assignments of threads requires additional synchronizations to preserve the degree of parallelism, because the problem of concurrent writes to the same memory address will occur if all CNs update corresponding APP values simultaneously. Existing GPU-based decoders use barrier synchronizations when communications between threads are required. Instead of utilizing this mechanism, the extra synchronizations are accomplished by atomic operations. Performance is not degraded noticeably by introducing additional atomic operations, because the number of threads that update the same APP value is constrained by the column weights of PCM. To lower the probability of concurrent updates of the same APP value and improve the performance of atomic operations, arrays of compressed base matrix are rearranged as shown in Fig. 1b.

3) Storage of Messages in Registers: Assigning threads to CNs in both phases of flooding scheduling enables the usage of registers to store the transformed representation of messages, because each thread only processes V2C and C2V messages in one row. It reduces shared memory accesses at the cost of redundant computations of C2V messages. Limited by the maximum number of threads per block, applying this method to LDPC codes with more than 1024 CNs requires manual expansions of *for* loops.

D. Layered-Based Decoder

Implementation of the layered-based decoder is illustrated in Fig. 4. It provides a new solution to breaking the performance bottleneck of layered-based decoders. The representation of V2C/C2V messages and resultant memory organization are different from existing GPU-based decoders (e.g. [9]–[12]). Several optimization techniques are presented for this memory organization as well. The design is described below.

1) Reduction of Memory Consumption: Shared memory consumption is the bottleneck of the layered-based decoder due to the partial parallel property of layered scheduling. Limited by the size of submatrix, the number of concurrent blocks residing on each SM is at most 25. To reach this upper bound, each block should allocate less than 3.84 KB shared memory. Such small amount of memory consumption, however, is unachievable. Hence, utilization of global memory results in higher throughput if storing V2C/C2V messages in the direct representation [12]. To take advantage of faster on-chip shared memory, messages are stored in the transformed representation in this work. It reduces memory consumption by 40%, 50% and 55% for 8-bit, 16-bit and 32-bit quantizations, respectively.

2) Duplicate Copy and Reuse of Memory: Messages stored in shared memory are copied to registers before the update of each layer in order to reduce shared memory accesses. Registers of signs, products of signs, and C2V messages are reused when computing old and new C2V messages. Intermediate V2C messages are copied to shared memory allocated for APP values to dispense with recomputations.

3) Computation of Product of Signs: To reduce memory consumption, signs of V2C messages are stored using bitmaps. Existing GPU-based decoders that apply this method compute products of signs sequentially. In [10], products are recovered with floating-point multiplications instead of XOR operations because the instruction throughput of the former ones is slightly higher on GPU. The parallel property of XOR operations is exploited in this work (shown in Fig. 4b), and less instructions are required to compute the results.

IV. Results and Discussions

The experiments are conducted on GeForce GTX 1080 Ti GPU, with programs compiled by CUDA Toolkit 8.0. Performances of the proposed decoders are simulated over AWGN (additive white Gaussian noise) channel, with BPSK (binary phase-shift keying) modulation.

The error correcting performance of the (1944, 972) code is shown in Fig. 5. Frame error rate (FER) and bit error rate (BER) are simulated down to 10^{-7} and 10^{-10},

TABLE I
COMPARISON OF THROUGHPUT PERFORMANCE WITH OTHER GPU-BASED LDPC DECODERS (SUPERSRICPT *: STORING MESSAGES IN REGISTERS)

		[9]		[10]	[11]	[12]	This Work	
GPU (TFLOPS)		GTX 470 (1.09)		GTX TITAN (4.50)	TITAN X (10.16)	GTX 480 (1.35)	GTX 1080 Ti (10.61)	
Algorithm		SPA	NMSA	NMSA	MSA	MSA	NMSA	
Scheduling		Flooding		Flooding	Flooding	Layered	Flooding	Layered
Throughput (Mb/s)	w/ ET (SNR)	22.5-100.3 (1.5-5dB)		N/A	N/A	85.3-364.9 (1.0-5.5dB)	1153.0-4771.8 (1.0-5.5dB)	708.4-3672.1 (1.0-5.5dB)
	w/o ET	39.98	39.82	236.70	913	91.8	1474.2 (1502.4*)	971.5
Normalized Throughput	w/ ET (SNR)	20.7-92.1 (1.5-5dB)		N/A	N/A	63.4-271.3 (1.0-5.5dB)	108.7-449.8 (1.0-5.5dB)	66.8-346.1 (1.0-5.5dB)
	w/o ET	36.7	36.6	52.6	89.9	68.3	139.0 (141.6*)	91.6

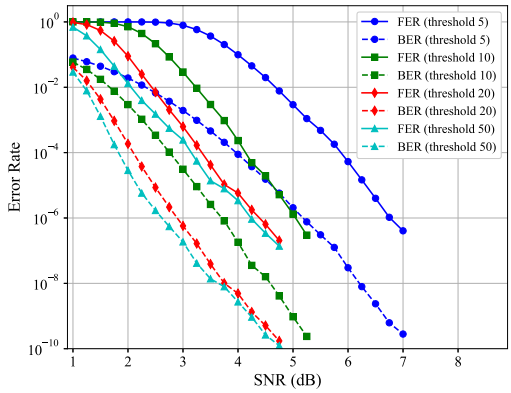


Fig. 5. Error correcting performance of (1944, 972) code in 802.11n standard.

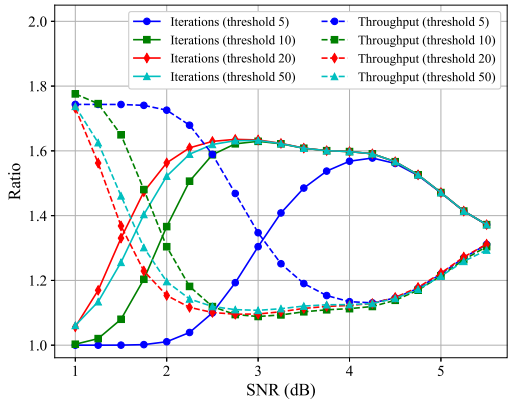


Fig. 6. Ratios of average iterations and throughputs between flooding-based and layered-based decoders for (1944, 972) code in 802.11n standard.

respectively. Empirical results suggest that a reasonable threshold of maximum iterations is about 20.

The flooding-based decoder achieves higher throughput than the layered-based decoder for the (1944, 972) LDPC code, even though it requires more iterations. Fig. 6 presents the ratios of average iterations and throughputs between the two decoders for different thresholds of maximum iterations. For LDPC codes with large submatrix size, the degree of parallelism is not the dominant factor because the threads available for each block is limited to 1024. The throughput of the proposed layered-based decoder is 36% higher than the flooding-based decoder when decoding a (18360, 16524) QC-LDPC code with submatrix size 306.

Comparison with other work is shown in TABLE I. The maximum iterations of all decoders are 10. Considering the differences in GPU performances, normalized throughputs (throughputs divided by TFLOPS) are computed to make a fair comparison. Both flooding-based and layered-based decoders outperform the corresponding state-of-the-art designs, with 55% and 34% improvements of normalized throughputs. The two decoders reach 4.77 and 3.67 Gb/s throughputs by incorporating early termination (ET) criterion. A flooding-based decoder is also implemented on Virtex UltraScale 440 FPGA (the largest density FPGA on market). The throughput achieved is up to 23.72 Gb/s with 76% resource utilization.

V. CONCLUSION

On a recently released desktop GPU, the proposed LDPC decoder achieves 4.7 Gb/s throughput. The high performance together with high flexibility make GPU an ideal platform for the study of LDPC codes, such as performances of decoding algorithms and properties of error floors. Tweaking the proposed architectures for other LDPC codes, and investigating their performances could be done in the future.

REFERENCES

[1] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.
[2] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electron. Lett.*, vol. 32, no. 18, pp. 1645–1646, Aug. 1996.
[3] J. Chen and M. P. C. Fossorier, "Density evolution for two improved BP-based decoding algorithms of LDPC codes," *IEEE Commun. Lett.*, vol. 6, no. 5, pp. 208–210, May 2002.
[4] X.-Y. Hu, E. Eleftheriou, and D. M. Arnold, "Progressive edge-growth Tanner graphs," in *Proc. IEEE Global Telecommun. Conf. (GlobeCom)*, San Antonio, TX, USA, Nov. 2001, pp. 995–1001.
[5] J. Thorpe, "Low-density parity-check (LDPC) codes constructed from protographs," Jet Propulsion Lab., Pasadena, CA, USA, IPN Progr. Rep. 42-154, Aug. 2003.
[6] Y. Kou, S. Lin, and M. P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 7, pp. 2711–2736, Nov. 2001.
[7] M. M. Mansour and N. R. Shanbhag, "High-throughput LDPC decoders," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 11, no. 6, pp. 976–996, Dec. 2003.
[8] J. Zhang and M. P. C. Fossorier, "Shuffled iterative decoding," *IEEE Trans. Commun.*, vol. 53, no. 2, pp. 209–213, Feb. 2005.
[9] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Proc. IEEE Symp. Appl. Specific Process. (SASP)*, Jun. 2011, pp. 82–85.
[10] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency ldpc decoding on gpu for sdr systems," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2013, pp. 1258–1261.
[11] S. Keskin and T. Kocak, "GPU-based gigabit LDPC decoder," *IEEE Commun. Lett.*, vol. 21, no. 8, pp. 1703–1706, Aug. 2017.
[12] X. Wen *et al.*, "A high throughput LDPC decoder using a mid-range GPU," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 7515–7519.