Low-Latency CUDA LDPC Decoder for SDR Solutions

Igor Volkov, Aleksei Kharin, Evgeny Likhobabin
Aleksei Ovinnikov and Vladimir Vityazev, *Member*, *IEEE*
Department of Telecommunications and Foundations of Radio Engineering
Ryazan State Radio Engineering University, RSREU
Ryazan, Russia
volkov.i.y@tor.rsreu.ru, kharin.a.v@tor.rsreu.ru, likhobabin.e.a@tor.rsreu.ru
ovinnikov.a.a@tor.rsreu.ru, vityazev.v.v@rsreu.ru

***Abstract*—This paper outlines a method for low-latency CUDA LDPC decoder implementation applicable for software-defined radio (SDR) solutions. The traditional approach to CUDA LDPC decoder implementation allows to get very high decoder throughput but has high decoding latency.**

It is shown that the proposed implementation of the CUDA LDPC decoder allows to significantly reduce the decoding latency. NVIDIA RTX 2080 was used as a simulation platform. Obtained latency and throughput values satisfy technical requirements for ultra low latency 5G services in NGMN specifications.

***Index Terms*—LDPC codes, CUDA, min-sum algorithm, software defined radio, low latency.**

I. Introduction

NOWADAYS LDPC codes [1] are widely used in many modern communication standards like DVB-T2/S2 [2], WiFi [3], WiMax [4] and etc. LDPC decoding algorithms are very complex, therefore, in recent years, parallel implementations of such algorithms on GPUs have been widely used [5]-[12]. One of the most popular approaches for implementing such decoders is a massively-codeword decoder with numerous codewords processing in parallel [5], [7], [9], [12]. This approach allows to get a good ratio between an amount of computing and an overhead of interaction with a GPU and also allows to hide GPU memory latency during decoding process. Because of these features, such decoders show a large throughput that can reach several Gbps [11]. However, to achieve this throughput, it is necessary to accumulate a packet of input codewords, which leads to decoding latency increasing up to hundreds of milliseconds [9], [12]. This approach is well established for simulation of telecommunication systems, but in such cases as software defined radio (SDR) processing with high latency is unacceptable. For example, in modern communication standards latency of even 1 ms is critical [13], [14]. For short LDPC codes, e.g. (1944, 972) code from [3], effective approaches to developing low-latency decoders already exist [11]. In such approaches, all calculations of a decoder take place in the high-speed local memory of the GPU. However, the limited size of this memory does not allow the use of such design with codes longer than several thousand bits. For these reasons, in this paper, we propose CUDA LDPC decoder design with low latency (less than 0.3 ms) for SDR solutions which can process with quasi-cyclic (QC) LDPC codes of different length.

This work was supported by Russian Science Foundation under Grant 17-79-20302.

The paper is organized as follows. In the next section, we give a brief review of used LDPC code and CUDA programming model, then describe proposed decoder implementation in detail. In Section III results of decoder efficiency estimation are presented. Finally, Section IV concludes the paper.

II. Decoder implementation

A. LDPC code

Any LDPC code can be represented as a parity check matrix $\mathbf{H}$ which contains mostly zeros and only a small number of ones. A (N, K) LDPC code is represented as the matrix $\mathbf{H}$ which includes $M \geq N-K$ rows and N columns. We refer to the M rows of $\mathbf{H}$ as check nodes (CN), and to all the elements of a LDPC codeword as variable nodes (VN).

One of the well-known types of LDPC codes are QC-LDPC codes used in various communication standards [3], [4]. QC-LDPC codes have been considered as a promising candidate for forward error correction in 5G systems [15]. The matrix $\mathbf{H}$ for such codes can be represented as $\mathbf{H} = P^{H_b}$, where P is cyclic-permutation matrix, and H_b is the base matrix. H_b consist of two parts $H_b = [H_{b1}\ H_{b2}]$.

Size of H_{b1} is equal to $m_b \times k_b$, where $k_b = n_b - m_b$. In turn, $k_b = K/z$ and $n_b = N/z$ where z is the circulant size. This part corresponds to the systematic bits. The matrix H_{b2} of size $m_b \times m_b$, in turn, corresponds to the parity-check bits.

In this paper, we use the custom QC-LDPC (16200, 9000) code. The circulant size for the presented code is $z \times z$, where $z = 90$. Maximum row and column weights of the proposed code are greater than similar weights of the (16200, 9000) code from the DVB-S2 standard [2] and equal to 12 and 7 respectively. As shown in Figure 1, the proposed code demonstrates better bit error rate performance in comparison with the DVB-S2 (16200, 9000) code. Estimation of bit error rate was performed for the following parameters: decoding algorithm - normalized MSA, maximum decoding iteration number $I_{max} = 20$, modulation type - BPSK, channel model - AWGN.

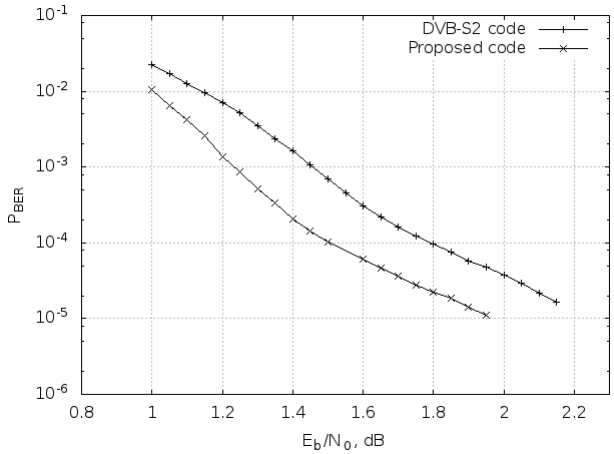

Fig. 1. Bit error rates for proposed QC-LDPC (16200, 9000) and for DVB-S2 (16200, 9000) codes

B. Decoding algorithm

As a decoding algorithm the Min-Sum Algorithm (MSA) was chosen in this work. The MSA is a simplified version of the near-optimal log-likelihood ratio based belief propagation (LLR BP) algorithm and is based on the criterion of a symbol-wise maximum of a posteriori probability (APP) [16], [17]. It is the widespread algorithm described in a detail in many other papers [8]-[12]. The algorithm is based on the transmission of messages with bit reliability between CNs and VNs with subsequent calculation of APP for all N bits of a codeword.

Denote messages between ith variable and jth check nodes (V2C) as $L_{i\to j}$ and messages between j check and i variable nodes (C2V) as $L_{j\to i}$. Also define L_i^{APP} as APP of ith bit of a codeword.

The proposed implementation of the MSA differs from the state-of-art decoder one main feature: for better memory utilization $L_{i\to j}$ and $L_{j\to i}$ messages are stored as constituent parts (first and second minimal C2V messages, minimal bit position as index of minimal V2C message, signs of C2V messages) and computed as necessary as:

$$L_{j\to i} = \alpha_{i\to j} \cdot \begin{cases} submin(\beta_{i\to j}), & \text{if } L_{i\to j} \text{ is min V2C} \\ min(\beta_{i\to j}), & \text{otherwise} \end{cases}, \quad (1)$$

$$L_{i\to j} = L_i^{APP} - L_{j\to i}, \quad (2)$$

where $\alpha_{i\to j}$, $\beta_{i\to j}$ – a sign and an absolute value of the message $L_{j\to i}$ respectively. This approach allowed us to hide memory latency during decoding process.

C. CUDA programming model

The main component of any CUDA device is a set of streaming multiprocessors (SMs) that perform actual computations. Each SM has a small amount of high-speed memory and consists of several streaming processors (SPs). Besides this memory, each GPU has a large on-board DRAM [18].

Programming model of CUDA device consists of executable threads combined in a thread blocks grid. CUDA memory hierarchy consists of three main levels: per-thread local memory, per-block shared memory and device memory which is divided into global and constant. Constant memory, as well as global memory, is a global scope memory, but it is read-only and has faster access than global memory [18].

As GPU development experience shows [5]-[12], the organization of proceeding with memory largely determines the overall efficiency and throughput of the final software. Therefore, there are two main requirements for GPU software developed that can be distinguished: high-speed local, shared and constant memory should be used as much as possible; memory collisions should be minimized.

D. Memory organization

As follows from the MSA description [16], [17], there are four main components in the decoding algorithm: check matrix **H**, input estimations of received symbols (LLRs), a posteriori probability for each element of a codeword (APPs), C2V and V2C messages. The distribution of these components in the CUDA device memory is shown in Figure 3.

To reduce the required amount of memory the check matrix **H** is represented in the form of two smaller matrixes: one for positions and one for shifts of non-zero elements of **H**.

Both, LLR and APP values are stored in global memory. For better memory utilization, at the beginning of each decoding iteration latest APP values are copied to shared memory. Even though LLR values are constant during decoding, they cannot be stored in constant memory due to their size.

In the decoding process, C2V and V2C values are used quite often, and it would be better to place them in fast shared memory. However, the amount of shared memory is insufficient to store all messages even in transformed representation. Therefore, C2V and V2C messages are stored in global memory and, like APPs, are copied to the shared memory at the beginning of each decoding iteration.

E. Decoder architecture

Referring to Figure 2a the decoder can be divided into four main parts, each part is represented by its own CUDA program kernel: codeword initialization, early termination check (syndrome calculation), APP updating, hard decision making.

The decoder initialization kernel is designed to set the initial values of all the arrays involved in the decoding process.

The APP update core is the main core of the LDPC decoder. It is intended for node messages and APPs updating and is launched by M check threads. The main steps of this core are presented in Figure 2b. At the beginning of each iteration the kernel copies all values required for calculation from global to local memory: first and second minimal check values, the position of the bit with minimal absolute value of message for the current check, signs of bit messages and APPs for connected bits from the previous iteration. At the next step search for new first and second minimum values, their positions, as well as recalculation of message signs are performed. Afterward, each check calculates messages transmitted to connected bits and sums them into the corresponding APP value. To avoid collisions, an atomic addition is used when summing messages from multiple checks to a single APP. At the end the kernel

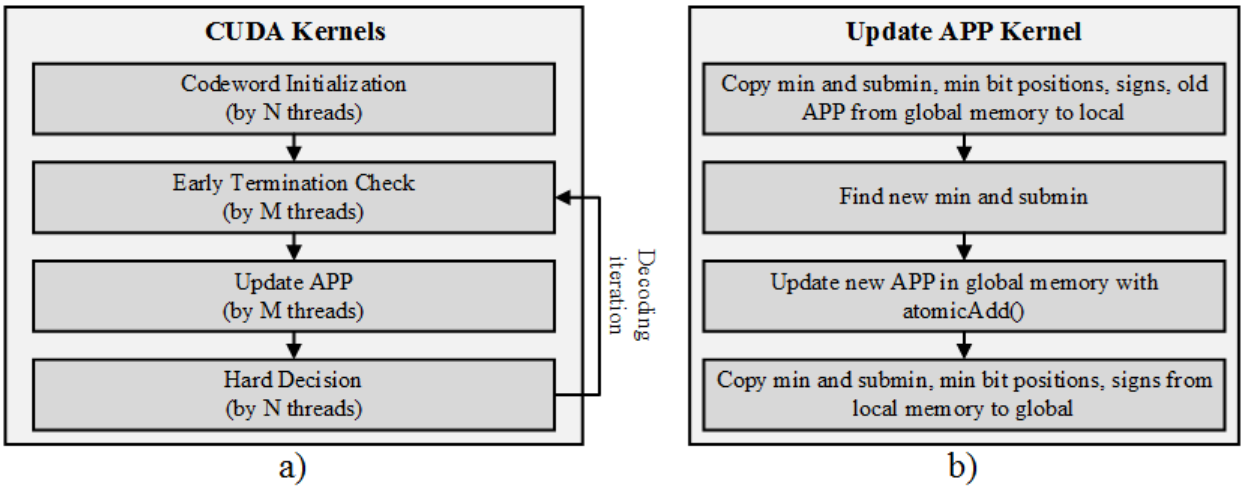

Fig. 2. Decoder architecture

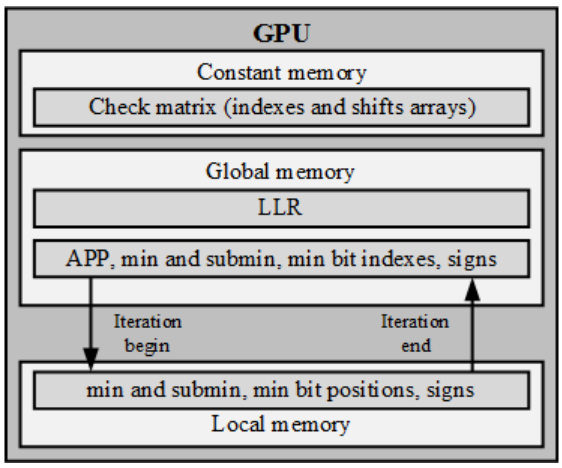

Fig. 3. Memory organization of CUDA device

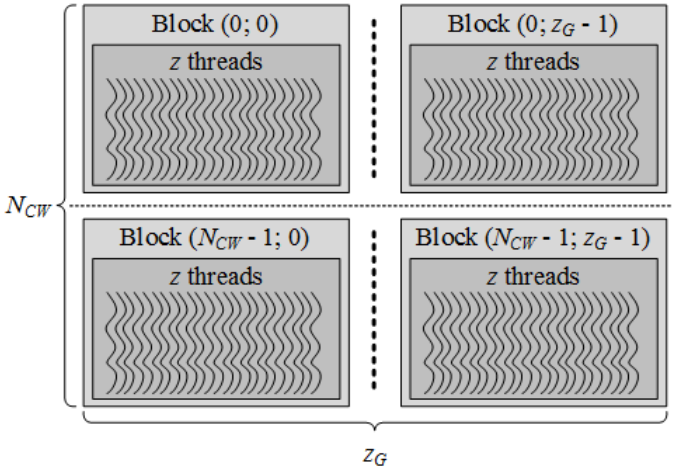

Fig. 4. Execution grid of thread blocks

copies new first and second minimum values, minimal bit position, and recalculated signs of bits to check messages backward to global memory.

The hard decision making kernel provides an estimation of transmitted codeword based on calculated APPs.

The early termination (ET) check kernel is developed to calculate a decoding syndrome **S**. If **S** = 0, the kernel sets the corresponding flag in global memory and subsequent decoding iterations are skipped. After decoding, this flag is copied to the host as a mark of successful decoding.

The main parameter that determines how the algorithm will be executed on a CUDA device is a grid of thread blocks that can be one-, two-, or three-dimensional. This grid describes how many threads and blocks will be used by the kernel. In the proposed implementation (see Figure 4), a two-dimensional $N_{CW} \times z_G$ grid is used. Rows and columns of the grids are representing processed codewords and check matrix circulants respectively. Thus, each block contains $z = 90$ threads that process rows or columns of circulant, depending on the current running CUDA kernel. For the codeword initialization and the hard decision kernels $z_G = N/z$, one thread in these kernels represents the processing of one bit of a codeword. In its turn, for the update APP and the ET check kernels $z_G = M/z$, in this case, one thread represents the processing of one check of a codeword. It is worth noting that in the CUDA a number of threads must be a multiple of 32, so 96 threads are started for each circulant and 6 of them are idle.

III. Simulation Results

During the simulations the test platform with the following parameters was used: CPU - Intel Xeon E5-2620 2.4 GHz (12 Cores), RAM - 16 GB DDR-4 PC4-17000, GPU - NVIDIA RTX 2080 with 2944 CUDA Cores.

As previously stated, there are two main criteria of decoder efficiency: throughput and latency, where latency in our case is time required to accumulate a packet of N_{CW} codewords. The throughput C_{LDPC} of the proposed decoder was measured on NVIDIA RTX 2080 for different values of N_{CW} and for 20 decoding iterations. Obtained results are presented in the table I. The measured throughput also includes codeword

TABLE I
THROUGHPUT AND LATENCY COMPARISON

Paper	Platform	Code	Codewords count, N_{CW}	Throughput, C_{LDPC}, Mbps	Latency, t_{LAT}, ms
[9]	GTX 660 Ti	(20000, 10000)	4480	250	358.4
[7]	GTX 570	(64800, 32400)	128	163.4	50.761
		(16200, 9000)	128	186.1	11.142
[6]	Fermi C2050	(64800, 32400)	16	107.8	9.618
[12]	Titan XP	(64800, 32400)	8000	390	1329.23
		(16200, 9000)	8000	450	288.01
This paper	RTX 2080	(16200, 9000)	1	60.2	0.269
		(16200, 9000)	6	130.5	0.745

initialization and ET check. All calculations were performed with single precision.

To calculate decoder latency, we assume that the LDPC decoder delay contributes most to overall receiver delay. In this case, decoder latency can be calculated as follows:

$$t_{LAT} = \frac{C_{LDPC}}{N \cdot N_{CW}}, \tag{3}$$

where C_{LDPC} — maximum throughput of the LDPC decoder.

Calculated latency of the proposed decoder for different values of N_{CW} also presented in the Table I. An analysis of the developed decoder showed that for N_{CW} greater than 6, the decoder throughput ceases to increase due to performance restrictions of the given GPU.

As shown in Table I, the proposed decoder implementation with NVIDIA RTX 2080 GPU allows getting decoding latency of less than 1 ms. The resulting latency is much less than the latency in decoders with only codewords parallelization [6], [7], [9], [12]. At the same time, the decoder throughput remains quite high and reaches 130 Mbps.

The loading of GPU SMs in the proposed design is up to 95-100%, which indicates the efficient organization of the algorithm. Such a small decoding latency and decoder throughput allow to use the proposed decoder for SDR solutions.

IV. CONCLUSION

The developed implementation of the LDPC decoder allows getting an low level of decoding latency (less than 1 ms) while maintaining a high throughput (more than 130 Mbps). These latency and throughput values satisfy technical requirements for ultra low latency 5G services in NGMN [14], which allows using such decoders in modern SDR solutions. Moreover, the proposed decoder satisfies the technical requirements of most other services, except for four of them requiring greater throughput in downlink. However, achieved throughput can be significantly increased by using more powerful GPUs. In addition, this implementation can be also easily scaled for effective use with other GPUs by changing the number of codewords processed in parallel.

REFERENCES

[1] R. G. Gallager, "Low-density parity-check codes", Cambridge, MA: M.I.T. Press, 1963.

[2] European Telecommunications Standards Institute, "Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications; Part 1: DVB-S2", European Telecommunications Standards Institute, ETSI EN 302 307-1 V1.4.1, November 2014. [Online]. Available: http://www.etsi.org. [Accessed: Jan. 20, 2020].

[3] 802.11n-2009 - IEEE Standard for Information technology– Local and metropolitan area networks– Specific requirements– Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications Amendment 5: Enhancements for Higher Throughput, 29 Oct. 2009, DOI: 10.1109/IEEESTD.2009.5307322

[4] 802.16e-2005 - IEEE Standard for Local and Metropolitan Area Networks - Part 16: Air Interface for Fixed and Mobile Broadband Wireless Access Systems - Amendment for Physical and Medium Access Control Layers for Combined Fixed and Mobile Operation in Licensed Bands, 28 Feb. 2006, DOI: 10.1109/IEEESTD.2006.99107

[5] G. Falcao, L. Sousa, V. Silva, L. Sousa, "Massively LDPC Decoding on Multicore Architectures", IEEE Transactions on Parallel and Distributed Systems, vol. 22 (2), pp. 309-322, April 2010.

[6] G. Falcao, J. Andrade, V. Silva, L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection", Electronics Letters, vol. 47 (9), pp. 542-543, April 2011.

[7] Stefan Grönroos, Kristian Nybom, Jerker Björkqvist, "Efficient GPU and CPU-based LDPC decoders for long codewords", Analog Integrated Circuits and Signal Processing, vol. 73 (2), pp. 583–595, November 2012.

[8] G. Wang, M. Wu. B. Yin, J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems", 2013 IEEE Global Conference on Signal and Information Processing, Austin, TX, USA, 3-5 December 2013.

[9] Y. Lin, W. Niu, "High Throughput LDPC Decoder on GPU", IEEE Communications Letters, vol. 18, no. 2, pp. 344-347, February 2014.

[10] S. Keskin, T. Kocak, "GPU-Based Gigabit LDPC Decoder", IEEE Communications Letters, vol. 21 (8), pp. 1703-1706, August 2017.

[11] J. Yuan, J. Sha, "4.7-Gb/s LDPC Decoder on GPU", IEEE Communications Letters, vol. 22 (3), pp. 478-481, March 2018.

[12] D. Kun, "High throughput GPU LDPC encoder and decoder for DVB-S2", 2018 IEEE Aerospace Conference, Big Sky, MT, 3-10 March 2018.

[13] C.-P. Li, J. Jiang, W. Chen, T. Ji, J. Smee, "5G ultra-reliable and low-latency systems design", 2017 European Conference on Networks and Communications (EuCNC), Oulu, Finland, 12-15 June 2017.

[14] Yu H, Lee H, Jeon H (2017) What is 5G? Emerging 5G mobile services and network requirements. Sustainability, 9, October 2017.

[15] "Double QC-LDPC codes with degree-3 for NR", National Taiwan University, 3GPP TSG RAN WG1 86, Gothenburg, Sweden, Aug. 2016.

[16] J. Chen, A. Dholakia, E. Eleftheriou, M. Fossorier, X.Y. Hu, "Near-Optimal Reduced-Complexity Decoding Algorithms for LDPC Codes", Proceedings of IEEE International Symposium on Information Theory, Lausanne, Switzerland, July, 2002.

[17] D. Declercq, M. Fossorier and E. Biglieri, "Min-sum decoding" in Channel Coding: Theory, Algorithms, and Applications. Waltham, MA, USA: Academic Press Library in Mobile and Wireless Communications, 2014, ch. 4, sec. 3.4, pp. 230-232.

[18] NVIDIA Corp. (2019, Nov.). CUDA Toolkit Documentation v10.2.89. [Online]. Available: https://docs.nvidia.com/cuda/