OpenCL/CUDA Algorithms for Parallel Decoding of any Irregular LDPC Code using GPU

Jan Broulim, Alexander Ayriyan, Hovik Grigorian, and Vjaceslav Georgiev

***Abstract* — This article provides a scalable parallel approach of an iterative LDPC decoder. The proposed approach can be implemented in applications supporting massive parallel computing. The proposed mapping is suitable for decoding any irregular LDPC code without the limitation of the maximum node degree. The implementation of the LDPC decoder with the use the OpenCL and CUDA frameworks is discussed and a performance evaluation is given at the end of this contribution.**

***Keywords* — Error correction, GPU, LDPC, parallel decoder.**

I. Introduction

SINCE Shannon's work, the topic of error detection and error correction codes, related to channel coding, has seen significant growth. The first serious discussion of error correction codes emerged in Hamming's work in 1950 [1], where Hamming provided a method for the correction of single and the detection of double bit errors with minimum redundancy being added to the transmitted data. Since the second half of the 20th century, error correction codes have attracted much attention in research work and have been utilized in many applications, including deep space photography transmission, television broadcasting services, Ethernet, wireless communication networks, and other signal processing applications.

Paper received April 4, 2019; revised August 7, 2019; accepted August 26, 2019. Date of publication December 30, 2019. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Zorica Nikolić.

This work was supported by the project SGS-2018-001 'Research and development of modern methods for electronic and communication systems in scientific and engineering applications', and the project 'Engineering applications of microworld physics', no. CZ.02.1.01/0.0/0.0/16_019/0000766.

Jan Broulim with the University of West Bohemia, Univerzitni 22, 306 14 Pilsen and the Institute of Experimental and Applied Physics, Czech Technical University in Prague, Husova 240/5, 110 00, Praha 1, Czech Republic. (e-mail: broulim@kae.zcu.cz).

Alexander Ayriyan is with the Laboratory of Information Technologies, Joint Institute for Nuclear Research, Joliot-Curie 6, 141980 Dubna, Russia, and Alikhanyan National Laboratory, 2 Alikhanyan Brothers Street, Yerevan, 0036, Armenia (e-mail: ayriyan@jinr.ru).

Hovik Grigorian is with the Laboratory of Information Technologies, Joint Institute for Nuclear Research, Joliot-Curie 6, 141980 Dubna, Russia and Jerevan State University, 1 Alek Manukyan St, Yerevan 0025, Armenia, and Alikhanyan National Laboratory, 2 Alikhanyan Brothers Street, Yerevan, 0036, Armenia. (email: hovik.grigorian@gmail.com).

Vjaceslav Georgiev is with the University of West Bohemia, Univerzitni 22, 306 14 Pilsen, Czech Republic (e-mail: georg@kae.zcu.cz).

This paper provides a parallel approach of an iterative Low Density Parity Check (LDPC) [2], [5] decoder. The presented parallel approach can be implemented in platforms allowing massive parallel computing, such as Graphics Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and computer data storages. The proposed approach is not limited to certain families of LDPC codes, but it supports decoding of any irregular LDPC code, and the maximum node degree is not limited. Benchmarks of the LDPC decoder implemented using Open Computing Language (OpenCL) [7] and Compute Unified Device Architecture (CUDA) [8] frameworks are discussed and a performance comparison is given at the end of this contribution.

Inspired by various comparisons between the OpenCL and CUDA applications from different fields of research, e. g. [10]–[12], we developed parallel algorithms for LDPC decoding using OpenCL, CUDA and other parallel processing systems. Several contributions published so far deal with a general comparison of OpenCL and CUDA [9] and with fitting the LDPC decoder on GPU platform [13]–[20]. However, the decoders are mostly limited to applications with some families of LDPC codes or bounded with the maximum node degree in the associated Tanner graph [6]. The proposed parallel approach is suitable for decoding any irregular LDPC code without the bound in terms of the maximum node degree.

This contribution can be easily used as a tutorial for implementing an irregular LDPC decoder as well as a general parallel approach for additional optimizations in order to make further accelerations. The parallel decoding approach is suitable for fast decoders implemented in GPUs. It is also highly applicable for accelerating bit error rate simulations used in designing new LDPC codes.

II. LDPC

Introduction

LDPC codes are becoming increasingly difficult to ignore in novel signal processing systems due to their excellent performance at long codeword lengths [21]. Compared to other codes, they provide better generalization and scalability to various lengths and redundancies. Although the number of applications with LDPC codes has grown significantly with the increasing speed of computing resources, decoding is still a computationally intensive task, which limits the deployability of non-approximated decoding algorithms for medium and long block length codes. However, the decoding can be accelerated significantly with the use of parallel multicore computing architectures.

Basic definitions

In this section, we provide basic mathematical definitions related to channel coding and their associations to LDPC codes and the presented parallel decoder.

Let $C = (n, k)$ be a linear block code, where the number of code bits is denoted as n and the number of information bits is denoted as k. The information vector of k bits is denoted as **m** and the kn generator matrix is denoted as **G**. The codeword **c** is given by **c**=**mG**, which is encoding. The parity-check matrix associated with the code C is denoted as **H**. Any vector **v** is a codeword if and only if $\mathbf{vH}^T=0$. The product of the multiplication $\mathbf{vH}^T$ is called the syndrome **s**. If the parity-check matrix **H** of code C is sparse, the code C is said to be the Low-Density Parity-Check (LDPC) code. The Tanner graph is a bipartite graph of sets of variable nodes and check nodes defined by the parity-check matrix **H**. If the element $H_{i,j} = 1$ (i corresponds to the row, while j corresponds to the column of the matrix **H**), an edge occurs between the check node c_i and the variable node v_j. The Tanner graph is used for LDPC decoding, which is briefly described in the following section.

The vector of check nodes connected with the j-th variable node is denoted as M_j be and the vector of variable nodes connected with the i-th check node is denoted as N_i.

$$M_j = \{i\} \Leftrightarrow H_{i,j} = 1 \quad (1)$$

$$N_i = \{j\} \Leftrightarrow H_{i,j} = 1 \quad (2)$$

Decoding

Decoding is a method for correcting errors in a corrupted codeword and the device performing decoding is called the decoder. The output of the decoder is usually called the estimation $\hat{\mathbf{c}}$.

Soft-decision decoding, including the Sum-Product (SP) algorithm [5] and its derivations, is supposed for the implementation of the LDPC decoder and related benchmarks in this article.

Soft decision LDPC decoding is an iterative process of passing values as messages in the Tanner graph through its edges. An estimation of the codeword is calculated after finishing each iteration and if the estimation is a codeword of the LDPC code, decoding is stopped. If a codeword is not found after a certain number of iterations (typically 5-100), decoding is terminated as unsuccessful.

Although the number of operations needed to be performed grows with the number of edges in the graph, the algorithm can be accelerated when deployed on massive parallel architectures. Moreover, the potential acceleration achieved by the parallelization of calculations grows with the number of edges in the graph, because more values can be calculated simultaneously. This can lead to interesting applications for long block length codes providing excellent error correcting capabilities.

Messages outgoing from one set of nodes are calculated with the use of the incoming values from the opposite set of nodes. Edges are used as interfaces for passing messages between the set of variable nodes and the set of check nodes, while each message outgoing from a node is passed through an edge. Each message outgoing from a node in the Tanner graph depends on the incoming messages from the connected nodes excluding the value received from the node which is the destination node. The process is illustrated in the following example. As can be seen in Fig. 1, the variable node v_0 is connected with check nodes c_0, c_2, c_3, c_5. Considering the calculation of the value being passed from v_0 to c_0, the value depends on the incoming values from the nodes c_2, c_3 and c_5. In the second half of an iteration, the value being passed from c_3 to v_0 depends on the incoming values from v_4, v_{11}, v_{12}.

Some our related work includes [24]-[26].

III. Parallelization of LDPC Decoding using GPU

Introduction

In recent years, there has been an increasing interest in implementing LDPC decoders in a wide variety of hardware architectures, including GPU. Several contributions deal with fitting the decoder on parallel architectures with the use of OpenCL or CUDA frameworks and discuss the benchmarks [13]-[20]. However, work reviewed so far deals mostly with some families of LDPC codes and the application of parallel decoders is limited. In this article, we propose a general parallel approach for the decoder of any irregular LDPC code. The proposed approach divides calculations into a scalable number of threads. Each thread performs the calculation of the value outgoing through the edge, which is associated with the thread itself (edge-level parallelization). The approach was chosen because of its suitability for any irregular LDPC matrices, scalability for any code block lengths and deployablity on many hardware architectures. It is also convenient for derived algorithms for LDPC decoding, such as Min-Sum (MS) or adaptive MS. In the previous work dealing with the parallel LDPC decoding, the calculations are mostly divided on the level of rows and columns of the parity-check matrices.

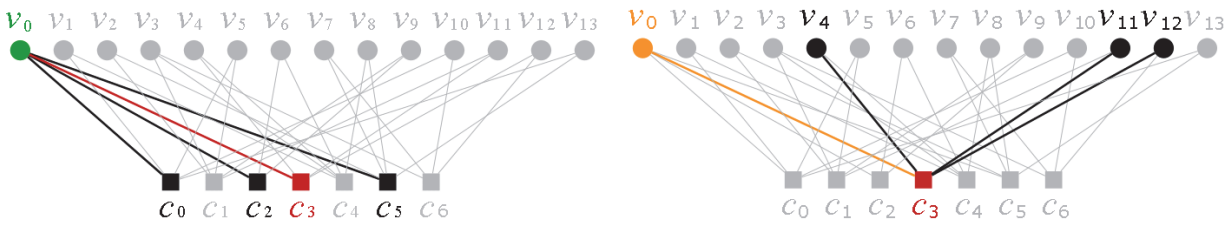
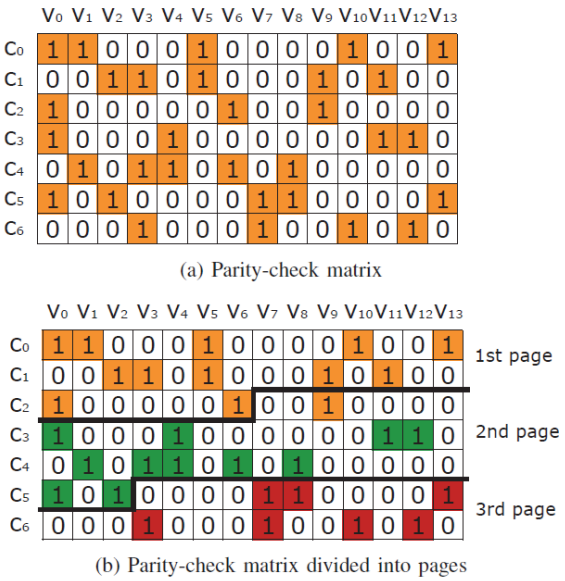


Fig. 1. The Tanner graph of the LDPC (14,7) code. The first half of the iteration - from variable nodes to check nodes. Values used for the calculation of the message between v_0 and c_3 are highlighted (left picture). The second half of the iteration - from check nodes to variable nodes. Values used for the calculation of the message between c_3 and v_0 are highlighted.



(a) Parity-check matrix

(b) Parity-check matrix divided into pages

Fig. 2. Parity-check matrix and the principle of the parallelization.

Our approach

In this section, we describe the approach of the edge-level parallelization used for the LDPC decoder. The principle is also shown in the illustrated example supported by consistent figures associated with the same LDPC (14,7) code.

Considering the code given by the parity-check matrix (Fig. 2) and the associated Tanner graph (Fig. 1), we define the following arrays used as address iterators for the parallel message passing algorithm (described in Algorithm 1):

- a sorted *m*-tuple of variable nodes $\mathbf{v}=(v_0,\ldots,v_m)$ starting with the lowest index and associated tuple of check nodes $\mathbf{c}=(c_0,\ldots,c_m)$, such i,j : $\mathbf{H}_{i,j}=1$ and $i\in[0,n-k)$, $j\in[0,n)$; then, (c_i,v_j) unequivocally defines an edge in the Tanner graph; n is the number of variable nodes and $(n–k)$ is the number of check nodes
- *m*-tuple of edges $\mathbf{e}=(e_0,\ldots,e_m)=(0,1,2,\ldots|\mathbf{c}|)$
- *m*-tuple of connected $\mathbf{t}=(t_0,\ldots,t_m)$ with a variable node v_k; then, $t_k=|\{v_k\}|, v_k\in\mathbf{v}$, and $k=\{0,1,\ldots,m-1\}$
- *m*-tuple of starting positions $\mathbf{s}=(s_0,\ldots,s_m)$ for iterating in order to calculate the value passed through the edge e_k; $s_k=\arg\min_k(v_k,v_k\in\mathbf{v})$
- *m*-tuple $\mathbf{u}=(u_0,\ldots,u_m)$ of relative positions of the e_k associated with the connected node v_k; $u_k=k-|(v_q):q<k,v_q\neq v_k|$

The arrays defined above are used as address iterators for calculations of messages outgoing from variable nodes to check nodes (the first half of the iteration). We also show the arrays in the illustrative example. Supposing the code (14,7) given by the parity-check matrix in Fig. 2, the arrays derived by the principle described above are shown in Table I. The first half of the iteration of the LDPC decoding process calculates the values passed from the variable nodes to the check nodes. With the use of the array iterators we can perform such calculations without any complicated operations with array indices. The pseudo code is shown in Algorithm 1. The local index of the thread (according to the OpenCL terminology) is denoted as *lid* and the number of synchronized threads working in parallel is denoted as *lgsize*.

Algorithm 1: *Message passing*

procedure ITERATE TO CHECK NODES
Input: r – incoming values e, s, t, u, v, p
Output: q
  for $(pg=0;\ pg<\text{totaledges};\ pg+=lgsize)$ **do**
    $value=p_{v_{lid+pg}}$
    for $i=s_{lid+pg}$ to $s_{lid+pg}+t_{lid+pg}-1$ **do**
      if $i=u_{lid+pg}+s_{lid+pg}$ **then** continue
      end if
      $value*=r_{i+pg}$ ▷ Example for SP algorithm
    end for
    $index=e_{lid+pg}$
    $q_{index}=value$
    ▷ Write in the memory and synchronize threads
  end for
end procedure

procedure ITERATE TO VARIABLE NODES
Input: q – incoming values $\overline{\mathbf{e}}$, $\overline{\mathbf{s}}$, $\overline{\mathbf{t}}$, $\overline{\mathbf{u}}$
Output: r
  for $(pg=0;\ pg<\text{totaledges};\ pg+=lgsize)$ **do**
    for $i=\overline{s_{lid+pg}}$ to $\overline{s_{lid+pg}}+\overline{t_{lid+pg}}-1$ **do**
      if $i=\overline{u_{lid+pg}}+\overline{s_{lid+pg}}$ **then** continue
      end if
      $value=$ ▷ Perform calculations here
    end for
    $index=\overline{e_{lid+pg}}$
    $r_{index}=value$
  end for
end procedure

Because all threads performing the calculations have to be synchronized after they finish writing in the memory and the number of synchronizable threads is strictly limited (e. g. 1024), the calculations are divided into several steps (pages) if necessary. This is when the number of edges is greater than the *lgsize* variable. An illustrative example for 12 synchronizable threads is shown in Fig. 2.

The arrays used for messages outgoing from the check nodes to the variable nodes are derived similarly. Keeping of the unique edge identifier $(c_i,\ v_j)$ and associated edge index e_k, the arrays **c**, **v, e** are sorted starting with the lowest check node index and other arrays are derived considering the messages outgoing from the check nodes. Such arrays are then denoted as $\overline{\mathbf{e}},\overline{\mathbf{c}},\overline{\mathbf{v}},\overline{\mathbf{t}},\overline{\mathbf{s}},\overline{\mathbf{u}}$ in the following descriptions. As a demonstrative example, the arrays for the second half of the iteration are shown in Table II.

The messages being passed from the variable nodes to the check nodes are denoted as **q** and the messages being passed from check nodes to variable nodes are denoted as **r**. The initial messages outgoing from variable to check nodes are given as $q_j=p_{vj}$, where $j\in[0;|\mathbf{e}|)$, v_j is the index of the associated variable node and p_j is calculated based on the received data and channel parameters [5].

The proposed parallel calculation of the estimation and the parallel calculation of the syndrome are listed in Algorithms 2 and 3.

IV. OpenCL and CUDA Implementation

The OpenCL is an open standard for parallel programming using the different computational devices, such as CPU, GPU, or FPGA. It provides a programming language based on the C99 standard. Unlike the OpenCL, CUDA is only for NVIDIA devices starting from G80 series (so called CUDA-enabled GPUs). When implementing an algorithm on GPU platform using the OpenCL or CUDA frameworks, two main issues have to be considered:

- size of the local memory (OpenCL) or shared memory (CUDA),
- size of the working group (OpenCL) or block size (CUDA).

Generally, the largest allocable size, typically in gigabytes for current devices, is located in the global memory. However, a higher speed is provided by the local memory.

Algorithm 2: *Calculating the estimation*

1: **procedure** CALCULATE ESTIMATION
▷ Parallel approach
Input: $\mathbf{r}$ – incoming values $\mathbf{s}$, $\mathbf{t}$, $\mathbf{v}$
Output: $\widehat{\mathbf{c}}$
2: **for** ($pg = 0$; $pg <$ totaledges; $pg+ = lgsize$) **do**
3: $Q_1 = r_{lid+pg}$
4: $Q_0 = 1 - r_{lid+pg}$
5: **for** $i = s_{lid+pg}$ to $s_{lid+pg} + t_{lid+pg} - 1$ **do**
6: $Q_1 = Q_1 r_{i+pg}$
7: $Q_0 = Q_0(1 - r_{i+pg})$
8: **end for**
9: $index = v_{lid+pg}$
10: **if** $Q_1 > Q_0$ **then** $\widehat{c}_{index} = 1$
11: **else** $\widehat{c}_{index} = 0$
12: **end if**
13: $index = v_{lid+pg}$
14: $q_{index} = value$
15: **end for**
16: **end procedure**

Algorithm 3: *Message passing*

1: **procedure** CALCULATE SYNDROME
▷ Parallel approach
Input: $\widehat{\mathbf{c}}$ – codeword estimation, $\overline{\mathbf{s}}$, $\overline{\mathbf{t}}$, $\overline{\mathbf{c}}$, $\overline{\mathbf{v}}$
Output: $\mathbf{z}$ – syndrome $\widehat{\mathbf{c}}\mathbf{H}^T$
2: **for** ($pg = 0$; $pg <$ totaledges; $pg+ = lgsize$) **do**
3: $value = 0$
4: **for** $i = \overline{s_{lid+pg}}$ to $\overline{s_{lid+pg}} + \overline{t_{lid+pg}} - 1$ **do**
5: $index = \overline{v_{lid+pg}}$
6: $value \hat{} = \widehat{c}_{index}$
7: **end for**
8: $index = \overline{c_{lid+pg}}$
9: $z_{index} = value$
10: **end for**
11: **end procedure**

The threads are split into working groups and they can be synchronized only among other threads at the same working group. The size of the working groups is strictly limited (typically 1024).

Both frameworks process two types of the code - host (runtime), running serially on CPU, and kernel (device), running parallely on GPU. The kernel is executed by the host. Because the kernel function has to be considered as a function running in parallel, each thread has its own unique identifier - the combination of global ID and local ID (denoted as *lid* in the algorithm listings) in OpenCL or the combination of thread ID and block ID in CUDA. The IDs can be recalculated vice versa.

V. Results

Developed algorithms for LDPC decoding were run on NVIDIA Tesla K40 (Atlas) and Intel Xeon E5-2695v2 platforms [24]. The NVIDIA device contains 2880 CUDA cores and runs at 745 MHz. The peak performance for double precision computations with floating point is 1.43 Tflops. The clock frequency of the Intel Xeon CPU is 2.4 GHz. All measurements include the time required for random generation, realized by the Xorshift+ algorithm and the Box-Muller transform.

Benchmarks (Figs. 3 and 4, Table III) were performed through the calculation of the Bit Error Rate at E_b / N_0 = 2dB for a code given by the NASA CCSDS standard [23] and its protographically expanded derivations [4]. Based on the results obtained from NVIDIA Tesla K80, we got a slightly better performance with the use of the CUDA framework. Compared to the CPU implementation run on Intel Xeon, the acceleration grows with the size of working groups and the number of decoders running in parallel to the limit of the device.

GPUs become very effective for longer block length codes, as also shown in Table III. The ratio between CPU (C++ compiler with O3 optimization) and GPU was 25 for a code of 262144 bits (Fig. 4).

VI. Conclusions

In this paper, we have provided a general parallel approach for decoding any irregular LDPC code, without the limitation for specific types of LDPC codes (including the common limit of the maximum node degree). Benchmarks for the OpenCL and CUDA approaches were performed on the CCSDS (256,128) standard and its protographically expanded derivations [3], [4]. The results were compared against the C++ implementation.

The acceleration was shown to be up to 22 times compared against the C++ implementation compiled with the O3 optimization, and up to 58 times compared against the C++ implementation without the optimization. To keep the generality, no simplifications in the decoding algorithm were applied and the experimental evaluation was performed with the use of the global memory. For further acceleration, several tasks can be considered, i. e. usage of the local memory, variables with a lower precision, look-up tables, or modifications of the algorithm for certain families of LDPC codes. For example, by moving the part of variables in the local (shared) memory, the decoder works approximately 40% faster in our experience. However, it was not possible to decode longer codewords because of the size limitations (240 kB of the local memory per working group for the used devices).

Another possibility for greater optimization could be the parallelization of less computationally intensive functions. After applying parallel algorithms for passing messages, calculating the syndrome and the estimation, the most serial time-consuming operation is checking syndrome for all

TABLE III: COMPARISON FOR OPENCL AND CUDA FRAMEWORK (LOCAL GROUP OF 512 THREADS AND 100 DECODERS WORKING IN PARALLEL) AGAINST THE CPU IMPLEMENTATION USING C++ COMPILER WITH O3 OPTIMIZATION. TIME WAS MEASURED FOR 10000 DECODED CODEWORDS EB/N0 = 2DB.

code	edges	OpenCL	CUDA	C++	C++ with O3 optimization
(256,128)	1024	0.32 s	0.32 s	24.24 s	3.11 s
(512,256)	2048	0.64 s	0.61 s	26.98 s	6.24 s
(1024,512)	4096	1.26 s	1.24 s	99.59 s	12.52 s
(2048,1024)	8192	2.56 s	2.51 s	105.56 s	25.27 s
(4096,2048)	16384	5.54 s	5.46 s	415.35 s	69.17 s
(8192,4096)	32768	12.08 s	12.08 s	545.74 s	172.67 s
(16384,8192)	65536	26.27 s	26.08 s	1717.25 s	367.75 s
(32768,16384)	131072	57.40 s	56.02 s	2893.91 s	1025.9 s
(65536,32768)	242144	117.31 s	116.86 s	8572.08 s	1989.26 s
(131072,65536)	524288	244.36 s	242.43 s	14082.71 s	5215.11 s
(262144,131072)	1048576	510.06 s	498.16 s	35104.28 s	12287.61 s

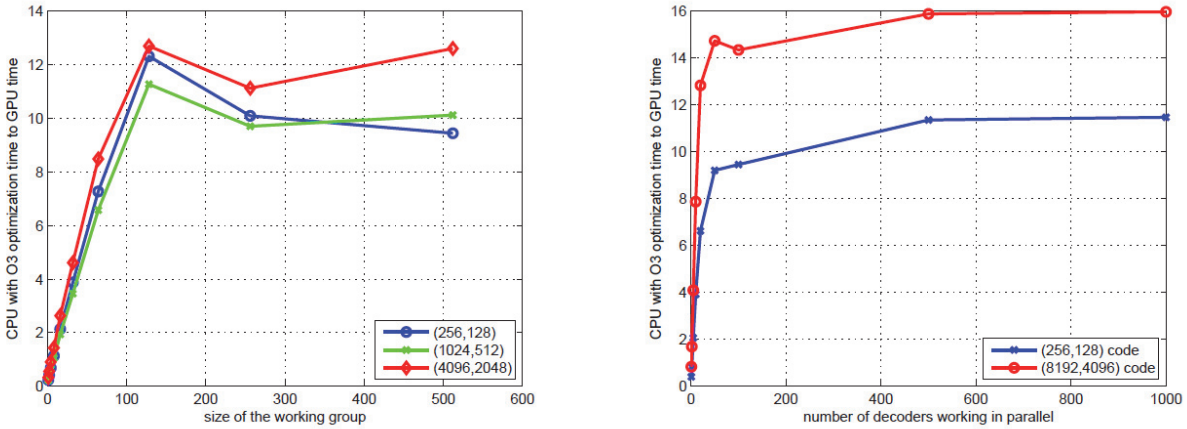


Fig. 3. Measured acceleration with the use of CUDA framework. Acceleration dependence on the block (working group) for 100 decoders running in parallel (left) and acceleration dependence on the number of decoders working in parallel when the size of the working group is 512 (right).

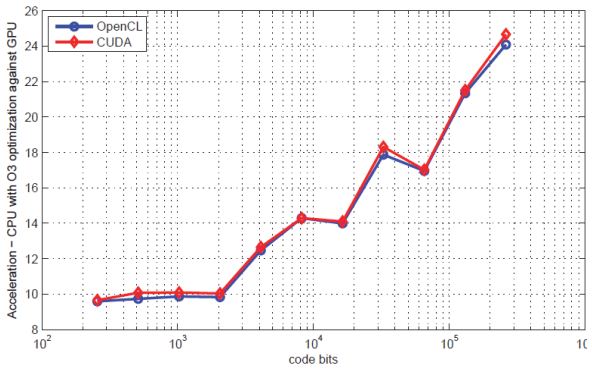


Fig. 4. Acceleration dependence on the length of the code. Comparison for OpenCL and CUDA frameworks (local group of 512 threads and 100 decoders working in parallel) against CPU implementation using C++ compiler with O3 optimization. Time was mesuared for 10000 decoded codewords at Eb/N0 =2dB.

zero equality (approximately 34% of the decoding function in our experience).

Because the OpenCL framework has found utilization in programming FPGA-based systems [22], the proposed algorithms and their potential modifications can be easily used in such devices.

ACKNOWLEDGMENT

The access to the heterogeneous cluster HybriLIT, provided by the Joint Institute for Nuclear Research, Dubna, Russia, is highly appreciated.

We would like to thank Vladimir Korenkov and Ivan Stekl for arranging the cooperation, Stefan Berezny for professional comments on this work, Jan Busa for technical support, and George Adam for interest and comments on this work.

REFERENCES

[1] R. Hamming, “Error detecting and error correcting codes,” *Bell Syst. Technical Journal*, vol. 29, pp. 41-56, 1950.

[2] R. G. Gallager, “Low Density Parity Check Codes,” *Transactions of the IRE Professional Group on Information Theory*, vol. IT-8, January 1962, pp. 2l-28.

[3] J. Thorpe, “Low-Density Parity-Check (LDPC) Codes Constructed from Protographs,” IPN Progress Report 42-154, 2003.

[4] Y. Fang, G. Bi, Y. L. Guan and F. C. M. Lau, “A Survey on Protograph LDPC Codes and Their Applications,” in *IEEE Communications Surveys & Tutorials*, vol. 17, no. 4, pp. 1989-2016, Fourthquarter 2015.

[5] N. Wiberg, Codes and Decoding on General Graphs. PhD thesis, Dept. of Electrical Engineering, Linköping, Sweden, 1996. Lionkoing studies in Science and Technology. Dissertation No. 440.

[6] R. M. Tanner, “A Recursive Approach to Low Complexity Codes. Information Theory,” *IEEE Transactions*, vol. 27, no. 5, pp.533,547, 1981.

[7] Khronos OpenCL Working Group, The OpenCL Specification, 2011 [Online]. Available:

https://www.khronos.org/registry/cl/specs/opencl-1.2.pdf (last visit 07/08/2019).

[8] NVIDIA Corporation, Cuda Runtime API, Reference manual, 2015 [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA Runtime API.pdf (last visit 07/08/2019).

[9] J. Fang, A. L. Varbanescu and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," *2011 International Conference on Parallel Processing*, Taipei City, 2011, pp. 216-225.

[10] C. Heinemann, S. S. Chaduvu, A. Byerly and A. Uskov, "OpenCL and CUDA software implementations of encryption/decryption algorithms for IPsec VPNs," *2016 IEEE International Conference on Electro Information Technology (EIT)*, Grand Forks, ND, 2016, pp. 0765-0770.

[11] G. Bernab, G. D. Guerrero and J. Fernndez, "CUDA and OpenCL implementations of 3D Fast Wavelet Transform," *Circuits and Systems (LASCAS)*, *2012 IEEE Third Latin American Symposium on*, Playa del Carmen, 2012, pp. 1-4.

[12] J. P. Arun, M. Mishra and S. V. Subramaniam, "Parallel implementation of MOPSO on GPU using OpenCL and CUDA," *2011 18th International Conference on High Performance Computing*, Bangalore, 2011, pp. 1-10.

[13] Yue Zhao, Xu Chen, Chiu-Wing Sham, Wai M. Tam, and Francis C.M. Lau, "Efficient Decoding of QC-LDPC Codes Using GPUs," *Algorithms and Architectures for Parallel Processing*, ICA3PP, 2011.

[14] G. Falcao, V. Silva, L. Sousa and J. Andrade, "Portable LDPC Decoding on Multicores Using OpenCL [Applications Corner]," *IEEE Signal Processing Magazine*, vol. 29, no. 4, pp. 81-109, July 2012.

[15] Y. Zhao and F. C. M. Lau, "Implementation of Decoders for LDPC Block Codes and LDPC Convolutional Codes Based on GPUs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 663-672, March 2014.

[16] S. Wang, S. Cheng and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," *2008 42nd Asilomar Conference on Signals, Systems and Computers*, Pacific Grove, CA, 2008, pp. 171-175.

[17] M. Beermann, E. Monr, L. Schmalen and P. Vary, "High speed decoding of non-binary irregular LDPC codes using GPUs," *SiPS 2013 Proceedings*, Taipei City, 2013, pp. 36-41.

[18] Joo-Yul Park and Ki-Seok Chung, "Parallel LDPC decoding using CUDA and OpenMP," *EURASIP Journal on Wireless Communications and Networking*, 2011.

[19] X. Wen et al., "A high throughput LDPC decoder using a mid-range GPU," *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, Florence, 2014, pp. 7515-7519.

[20] G. Wang, M. Wu, B. Yin and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," *Global Conference on Signal and Information Processing (GlobalSIP)*, 2013 IEEE, Austin, TX, 2013, pp. 1258-1261.

[21] Sae-Young Chung, G. D. Forney, T. J. Richardson and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communications Letters*, vol. 5, no. 2, pp. 58-60, Feb 2001.

[22] Implementing FPGA Design with the OpenCL Standard, Altera, 2013.

[23] Short Block Length LDPC Codes for TC Synchronization and Channel Coding. CCSDS Experimental Specification. NASA, 2015.

[24] Whitepaper of NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110/210. http://www.nvidia.com/object/gpu-architecture.html (last visit 07/08/2019).

[25] J. Broulim and V. Georgiev, "LDPC error correction code utilization," *20th Telecommunications Forum (TELFOR)*, 2012, Belgrade, 2012, pp. 1048-1051.

[26] J. Broulim, P. Broulim, J. Moldaschl, V. Georgiev and R. Salom, "Fully parallel FPGA decoder for irregular LDPC codes," *23rd Telecommunications Forum Telfor (TELFOR)*, 2015, Belgrade, 2015, pp. 309-312.

[27] J. Broulim, S. Davarzani, V. Georgiev and J. Zich, "Genetic optimization of a short block length LDPC code accelerated by distributed algorithms," *24th Telecommunications Forum (TELFOR)*, 2016, Belgrade, 2016, pp. 250-253.