OpenCL/CUDA algorithms for parallel decoding of any irregular LDPC code using GPU

Jan Broulim, Alexander Ayriyan, Vjaceslav Georgiev

***Abstract*—This article provides a scalable parallel approach of an iterative LDPC decoder, presented in a tutorial-based style. 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 the performance evaluation is given at the end of this contribution.**

***Index Terms*—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.

This paper provides a parallel approach of an iterative Low Density Parity Check (LDPC) decoder, presented in a tutorial style. 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 for 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) and Compute Unified Device Architecture (CUDA) frameworks are discussed and the performance comparison is given at the end of this contribution.

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 simmulations used in designing new LDPC codes.

Several contributions published so far deal with fitting the LDPC decoder on GPU platform [6]- [14]. However, the decoders are mostly limited for applications with some families of LDPC codes or bounded with the maximum node degree in the associated Tanner graph [5]. The proposed parallel approach is suitable for decoding any irregular LDPC code without the bound in terms of the maximum node degree.

September 7, 2016

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, Horska 3a/22, 128 00, Praha 2, 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 (e-mail: ayriyan@jinr.ru).

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

II. LDPC

A. Introduction

LDPC codes [2] represent the coding technique with the best known error correcting capabilities. LDPC codes surpassed other codes, including turbo codes and Reed Solomon codes, at the correcting performance, and they are becoming increasingly difficult to ignore in novel signal processing systems. Although the number of applications with LDPC codes has grown significantly with the increasing speed of computing resources, decoding is still a computionally 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. Our work related to LDPC codes include [16]–[19].

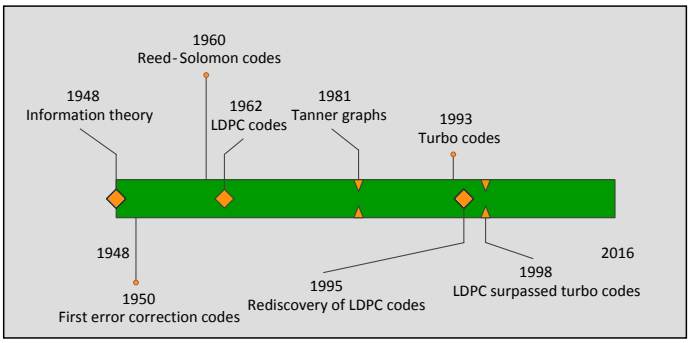


Fig. 1: Historical milestones in error correction coding theory.

B. Basic defitions

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

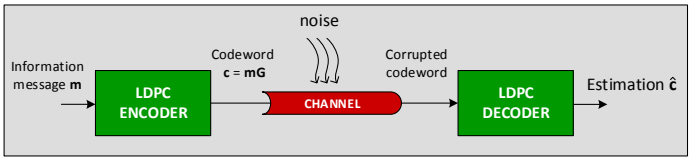


Fig. 2: Communication process - transfering information through the noisy channel.

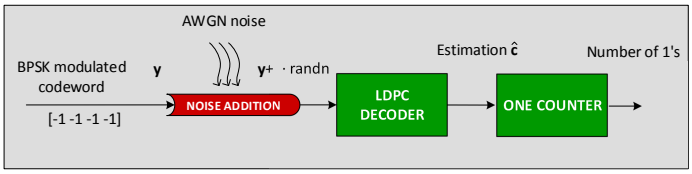


Fig. 3: Utilization of the noisy channel for the Bit Error Rate simulation.

Let $\mathcal{C} = (n, k)$ 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 $\mathbf{m}$ and the (kn) generator matrix is denoted as $\mathbf{G}$. The codeword $\mathbf{c}$ is given by $\mathbf{c} = \mathbf{mG}$, which is encoding. The parity-check matrix associated with the code $\mathcal{C}$ is denoted as $\mathbf{H}$. Any vector $\mathbf{v}$ is a codeword if and only if $\mathbf{vH}^T = \mathbf{0}$. The product of the multiplication $\mathbf{vH}^T$ is called the syndrome $\mathbf{s}$. If the parity-check matrix $\mathbf{H}$ of code $\mathcal{C}$ is sparse, the code $\mathcal{C}$ 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 $\mathbf{H}$. If the element $H_{i,j} = 1$ (i corresponds to the row, while j corresponds to the column of the matrix $\mathbf{H}$), 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 j-th variable node is denoted as $\mathcal{M}_j$ be and the vector of variable nodes connected with the i-th check node is denoted as $\mathcal{N}_i$. Then

$$\mathcal{M}_j = \{i\} \Leftrightarrow \mathbf{H}_{i,j} = 1 \tag{1}$$

$$\mathcal{N}_i = \{j\} \Leftrightarrow \mathbf{H}_{i,j} = 1 \tag{2}$$

C. 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 $\hat{\mathrm{c}}$, as illustrated in Fig. 2. Two main principles, listed below, can be considered for LDPC decoding. The principles are:

- Hard-decision, e. g. Bit-Flipping
- Soft-decision, working with probabilities during decoding process

Soft-decision decoding, including the Sum-Product (SP) algorithm [4] and its derivations, is supposed for the implementation of the LDPC decoder and related benchmarks in this article.

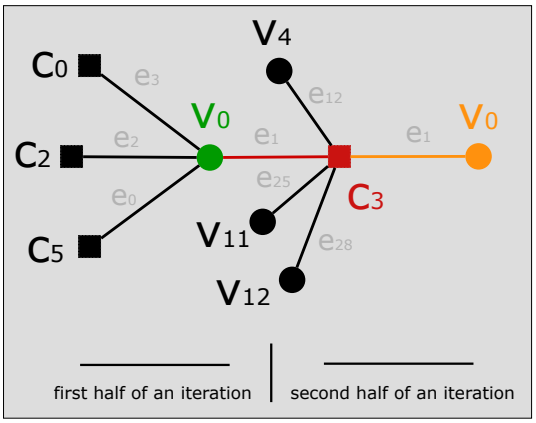


Fig. 5: Data flow in the Tanner graph when passing values from v_0 to c_3 and back.

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.

All messages passed in the Tanner graph represent probabilities, which are used for calculating the estimation after finishing every iteration. Because the algorithm convergence is affected significantly by the parameters of the Tanner graph (especially the number of short cycles), there is no reason for performing relatively high number of iterations. Therefore, the maximum number of iterations is limited.

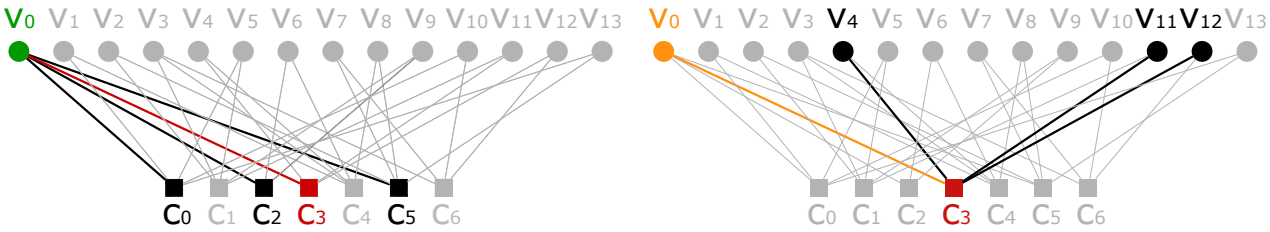
Messages outgoing from one set of nodes are calculated with the use of the incoming values from the opposite set of nodes. Edges are used as interfaces for passing messages between the set of variable nodes and the set of check nodes, while each message outgoing from a node is passed through an edge. 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, as Algorithm 1 describes in more detail. The process is ilustrated in the following example. As can be seen in Fig. 4, the variable node v_0 is connected with check nodes c_0, c_2, c_3, c_5. Considering the calculation of the value beeing 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 beeing passed from c_3 to v_0 depends on the incoming values from v_4, v_{11}, v_{12}. The data flow is shown in Fig. 5. The passed values are used for calculating estimations after each iteration.

Soft-decision decoding, described in terms of the pseudocode, is listed in Algorithm 2 and in referenced Algorithm 1. Formulas used in the pseudocode represent the SP algorithm without any simplifications and modifications.

III. Parallelization of LDPC decoding using GPU

A. Introduction

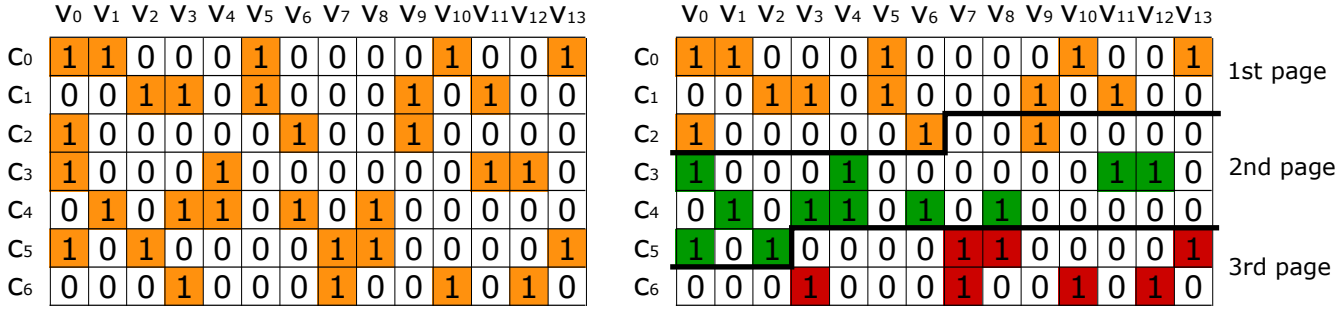
The SP algorithm works as an iterative process of message passing between the two sets of nodes (variable and check) in



(a) 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.

(b) 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.

Fig. 4: Tanner graph of the LDPC (14,7) code.



(a) Parity-check matrix

(b) Parity-check matrix divided into pages

Fig. 6: Parity-check matrix and the principle of the parallelization

the Tanner graph. 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.

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 OpenCL or CUDA frameworks and discuss the benchmarks [6]- [14]. However, work reviewed so far deal 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 derivated 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.

B. 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. 6) and associated Tanner graph (Fig. 4), we define the following arrays used as address iterators for the parallel message passing algorithm (described in Algorithms 3 and 4):

- a sorted tuple of variable nodes $\mathbf{v} = (v_j)$ starting with the lowest index and associated tuple of check nodes $\mathbf{c} = (c_i)$, 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
- a tuple of edges $\mathbf{e} = (e_k) = (0, 1, 2, ..., |\mathbf{c}|)$
- a tuple of connected edges $\mathbf{t} = (t_k)$ with a variable node v_k; then, $t_k = |(v_k)|, v_k \in \mathbf{v}$
- a tuple of starting positions $\mathbf{s} = (s_k)$ for iterating in order to calculate the value passed through the edge e_k; $s_k = \arg\min_k(v_k : v_k \in \mathbf{v})$
- a tuple $\mathbf{u} = (u_k)$ 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 .6, 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

Algorithm 1 Message passing

1: **procedure** VALUES TO CHECK NODES
▷ First half on an iteration
Input: $\mathbf{p}$, $\mathbf{r}$
Output: $\mathbf{q}$
2: **for all** $j \in [\mathbf{0}, |\mathcal{M}|)$ **do**
3: **for all** $i \in [\mathbf{0}, |\mathcal{N}|)$ **do**
4: $q^0_{i,j} = p^0_j$
5: $q^1_{i,j} = p^1_j$
6: **for all** $i' \in \mathcal{M}_j \setminus i$ **do**
7: $q^0_{i,j} = q^0_{i,j} r^0_{i',j}$
8: $q^1_{i,j} = q^1_{i,j} r^1_{i',j}$
9: **end for**
10: **end for**
11: **end for**
12: **end procedure**

13: **procedure** VALUES TO VARIABLE NODES
▷ Second half on an iteration
Input: $\mathbf{q}$
Output: $\mathbf{r}$
14: **for all** $j \in [\mathbf{0}, |\mathcal{M}|)$ **do**
15: **for all** $i \in [\mathbf{0}, |\mathcal{N}|)$ **do**
16: $r^0_{i,j} = 1$
17: $r^1_{i,j} = 1$
18: **for all** $j' \in \mathcal{N}_i \setminus j$ **do**
19: $r^0_{i,j} = r^0_{i,j}(1 - 2q^1_{i,j'})$
20: **end for**
21: $r^0_{i,j} = 1/2 + 1/2 r^0_{i,j}$
22: $r^1_{i,j} = 1 - r^0_{i,j}$
23: **end for**
24: **end for**
25: **end procedure**

Algorithm 2 Soft-decision decoding

1: **procedure** DECODEAWGN ▷ SP algorithm
Input: $\mathbf{y}$ – output from a demodulator
ITERATIONS – maximum number of iterations
σ – variance of the channel
Output: $\widehat{\mathbf{c}}$
2: $\mathbf{q}$ =Initialize$(\mathbf{p}, \sigma)$ ▷ See lines 15–24
3: $\mathbf{r}$ =Values to Variable Nodes$(\mathbf{q})$
▷ See Algorithm 1 (serial) or 3 (parallel approach)
4: $\widehat{\mathbf{c}}$ =Calculate Estimation$(\mathbf{r})$ ▷ See lines 25–35
5: **if** $\widehat{\mathbf{c}}\mathbf{H}^T = \mathbf{0}$ **then return** $\widehat{\mathbf{c}}$
6: **end if**
7: **for** $it \in (0, \text{ITERATIONS})$ **do**
8: $\mathbf{q}$ =Values to Check Nodes$(\mathbf{r})$
▷ See Algorithm 1 (serial) or 3 (parallel approach)
9: $\mathbf{r}$ =Values to Variable Nodes$(\mathbf{q})$
▷ See Algorithm 1 (serial) or 3 (parallel approach)
10: $\widehat{\mathbf{c}}$ =Calculate Estimation$(\mathbf{r})$ ▷ See lines 25–35
11: **if** $\widehat{\mathbf{c}}\mathbf{H}^T = \mathbf{0}$ **then return** $\widehat{\mathbf{c}}$
12: **end if**
13: **end for**
14: **end procedure**

15: **procedure** INITIALIZE ▷ Probabilities for AWGN
Input: $\mathbf{y}$, σ
Output: $\mathbf{q}$
16: **for all** $y_j \in \mathbf{y}$ **do**
17: $p_j = 1.0/(1 + \exp(-2y_j/\sigma^2))$
18: **end for**
19: **for all** $j \in [\mathbf{0}, |\mathcal{M}|)$ **do**
20: **for all** $i \in [\mathbf{0}, |\mathcal{N}|)$ **do**
21: $q_{i,j} = p_j$
22: **end for**
23: **end for**
24: **end procedure**

25: **procedure** CALCULATE ESTIMATION
▷ See Algorithm 3 for parralel approach
Input: $\mathbf{p}$, $\mathbf{r}$
Output: $\widehat{\mathbf{c}}$
26: **for all** $j \in [\mathbf{0}, |\mathcal{M}|)$ **do**
27: $Q^0_{i,j} = p^0_j$
28: $Q^1_{i,j} = p^1_j$
29: **for all** $i \in \mathcal{M}_j$ **do**
30: $Q^0_{i,j} = Q^0_{i,j} r^0_{i,j}$
31: $Q^1_{i,j} = Q^1_{i,j} r^1_{i,j}$
32: **end for**
33: **if** $Q^0_{i,j} > Q^1_{i,j}$ **then** $\widehat{\mathrm{c}}_j = 0$
34: **else** $\widehat{\mathrm{c}}_j = 1$
35: **end if**
36: **end for**
37: **end procedure**

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 3. 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 $lgsize$. 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 in 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. 6.

The arrays for the second half on the iteration can be derived similarly. Keeping of the unique edge identifier (c_i, v_j) and associated edge index e_k, the arrays $\mathbf{c}, \mathbf{v}, \mathbf{e}$ 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 algorithm performing the second half of the iteration processes the arrays described above. Its pseudo code is shown in Algorithm 3. After finishing the second half of the iteration we can continue with the next iteration. The whole decoding principle remains the same, as described in Algorithm 1.

For example, the address iterators for LDPC (14,7) code are listed in Table I and Table II. Both tables are particularly useful for understanding the principle and checking the correctness of the implementation. To keep the consistency and for tutorial purposes, both tables are associated with the LDPC (14,7) code given by the parity-check matrix from Fig. 6.

Algorithm 3 Message passing

procedure ITERATE TO CHECK NODES
▷ Half on an iteration
Input: $\mathbf{r}$ – incoming values $\mathbf{e}$, $\mathbf{s}$, $\mathbf{t}$, $\mathbf{u}$
Output: $\mathbf{q}$
  for (p = 0; $p <$ totaledges; $p+ = lgsize$) **do**
    for i = s_{lid+p} to $s_{lid+p} + t_{lid+p} - 1$ **do**
      if $i = u_{lid+p} + s_{lid+p}$ **then** continue
      end if
      $value$ = perform calculations ▷ Algorithm 1
    end for
    $index = e_{lid+p}$
    $q_{index} = value$
  end for
end procedure

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

IV. BIT ERROR RATE SIMULATOR

Apart from the implementation of the LDPC decoder, we also considered a Bit Error Rate simulator based on the Additive White Gaussian Noise (AWGN). The simulator is a highly useful tool for benchmarks and code evaluation purposes. The code evaluation requires up to billions of operations to be performed and it is the most time-consuming part of algorithms designing new and innovative LDPC codes. Therefore, its parallelization leads to a significant acceleration of a code design process and more precise simulations become possible. Fast simulations are also needed for evaluating

Algorithm 4 Calculate Estimation

procedure CALCULATE ESTIMATION
▷ Parallel approach
Input: $\mathbf{r}$ – incoming values $\mathbf{s}$, $\mathbf{t}$, $\mathbf{v}$
Output: $\widehat{\mathbf{c}}$
  for (p = 0; $p <$ totaledges; $p+ = lgsize$) **do**
    $Q_1 = r_{lid+p}$
    $Q_0 = 1 - r_{lid+p}$
    for i = s_{lid+p} to $s_{lid+p} + t_{lid+p} - 1$ **do**
      $Q_1 = Q_1 r_{i+p}$
      $Q_0 = Q_0(1 - r_{i+p})$
    end for
    $index = v_{lid+p}$
    if $Q_1 > Q_0$ **then** $\widehat{c}_{index} = 1$
    else $\widehat{c}_{index} = 0$
    end if
    $index = v_{lid+p}$
    $q_{index} = value$
  end for
end procedure

candidate solutions when applying algorithms for performing LDPC code optimizations.

For BER calculation, codewords are modulated and transmitted through the AWGN channel given by the parameter σ (often recalculated to the E_b/N_0 ratio), as can be seen in Fig. 3. The decoder then receives noised vectors, which are decoded, and Hamming distances between decoded vectors and original codewords are calculated. Due to the linearity of LDPC codes, it is enough to transmit only zero codewords and count the number of 1's at the output of the decoder (Fig. 3).

$$\sigma^2 = \frac{1}{R}\frac{N_0}{2} \tag{3}$$

$$R = \frac{k}{n} \tag{4}$$

where k is the length of the information message, n is the length of the codeword, and E_b is the energy per bit.

V. OPENCL AND CUDA IMPLEMENTATION

In current signal and data processing systems, there is an unambiguous trend to use parallel architectures to increase the processing speed, which plays a crucial role in real time applications and determines a deployability of computationally complex algorithms in hardware. Hardware devices supporting massively parallel processing algorithms generally include Graphics Processing Units (GPUs), which are considered in this tutorial article.

In this work, the CUDA and the OpenCL frameworks are used for GPU computations. 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 OpenCL, CUDA is only for NVIDIA devices starting from G80 series (so called CUDA-enabled GPUs). CUDA gives a possibility

TABLE I: Addresses for message calculation outgoing from variable nodes.

array	values
e	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
v	0 0 0 0 1 1 2 2 3 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 11 12 12 13 13
c	5 3 2 0 4 0 5 1 6 4 1 4 3 1 0 4 2 6 5 5 4 2 1 6 0 3 1 6 3 5 0
t	4 4 4 4 2 2 2 2 3 3 3 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
s	0 0 0 0 4 4 6 6 8 8 8 11 11 13 13 15 15 17 17 19 19 21 21 23 23 25 25 27 27 29 29
u	0 1 2 3 0 1 0 1 0 1 2 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0 1

TABLE II: Addresses for message calculation outgoing from check nodes.

array	values
$\overline{\mathbf{e}}$	3 5 14 24 30 7 10 13 22 26 2 16 21 1 12 25 28 4 9 11 15 20 0 6 18 19 29 8 17 23 27
$\overline{\mathbf{v}}$	0 1 5 10 13 2 3 5 9 11 0 6 9 0 4 11 12 1 3 4 6 8 0 2 7 8 13 3 7 10 12
$\overline{\mathbf{c}}$	0 0 0 0 0 1 1 1 1 1 2 2 2 3 3 3 3 4 4 4 4 4 5 5 5 5 5 6 6 6 6
$\overline{\mathbf{t}}$	5 5 5 5 5 5 5 5 5 5 3 3 3 4 4 4 4 5 5 5 5 5 5 5 5 5 5 4 4 4 4
$\overline{\mathbf{s}}$	0 0 0 0 0 5 5 5 5 5 10 10 10 13 13 13 13 17 17 17 17 17 22 22 22 22 22 27 27 27 27
$\overline{\mathbf{u}}$	0 1 2 3 4 0 1 2 3 4 0 1 2 0 1 2 3 0 1 2 3 4 0 1 2 3 4 0 1 2 3

to write programs based on the C/C++ and Fortran languages. OpenCL and CUDA programming models are illustrated in Fig. 7.

A. Necessary considerations

When implementing an algorithm on GPU platform using 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).

GPU devices offer several types of the allocable memory, which differ in their speed and their size. The memory type used to store variables is specified in the source code by the prefix according to the OpenCL or CUDA syntax rules. Generally, the largest allocable size, typically in gigabytes for current devices, is located in the global memory. However, the global memory is also the slowest one. A higher speed is provided by the local memory, but the size is typically only in kilobytes. Exceeding the limited size of the local memory usually leads to incorrect results without any warnings in the compilation report.

Another crucial issue related to an algorithm implementation in GPU devices is the working group size. Although the GPU can run thousands of threads in parallel, these threads are not synchronized among each other in terms of writing in the memory. 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).

B. Coding

Both frameworks processes two types of code

- host (runtime), running serially on CPU
- kernel (device), running parallely on GPU

Listing 1: Types

```
typedef struct Edge{
   int index; // e array
   int vn;    // v array
   int cn;    // c array
   int edgesConnectedToNode;  // t array
   int absoluteStartIndex;    // s array
   int relativeIndexFromNode; // u array
} Edge;

typedef struct EdgeData{
   double passedValue;
} EdgeData;

typedef struct CodeInfo{
   int totalEdges;  // number of edges
   int varNodes;    // number of variable nodes
   int checkNodes;  // number of check nodes
} CodeInfo;
```

The kernel is executed by the host. In CUDA, the kernel execution is more straightforward compared to OpenCL, as can be seen in the consistent examples in Listing 2. Both codes execute the kernel $berSimulate$ in 100 working groups (blocks) with 512 threads per one working group. After finishing the kernel, the results are copied in the $berOut$ array and processed 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 in OpenCL or the combination of thread ID and block ID in CUDA, which can be recalculated vice versa. The parallel implementation of the function $decodeAWGN$, defined in Algorithm 2, is shown in Listing 3. Types used for code definition and passing messages are pointed in Listing 1.

Some main differencies between the OpenCL and CUDA syntax rules are shown in Table III, which can be used when moving the source code from one framework to another one.

VI. Results

A. Experimental evaluation

Developed algorithms for LDPC decoding were run on NVIDIA Tesla K40 (Atlas) and Intel Xeon E5-2695v2 platforms [21], [22]. 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, realised by the Xorshift+ algorithm and the Box-Muller transform.

Benchmarks were performed through the calculation of the Bit Error Rate at $E_b/N_0 = 2\mathrm{dB}$ for a code given by the NASA CCSDS standard [20] and its protographically expanded derivations [3]. Based on the results obtained from NVIDIA Tesla K80, we got slightly better performance with the use of the CUDA framework, as shown in Fig. 9. 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, as illustrated in Fig. 8. GPU become very effective for longer block length codes, as also shown in Table IV. The ratio between CPU (C++ compiler with O3 optimization) and GPU was 25 for code of 262144 bits.

B. Further acceleration

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 is not possible to decode longer codewords because of the size limitations (240 kB of the local memory per working group). Another possibility for greater optimization could be the parallelization of less computationally intensive functions. After applying parrallel algorithms for passing messages, calculating the syndrome and the estimation, the most serial time-consuming operation is checking syndrome for all zero equality (approximately 34% of the decoding function in our experience).

VII. Conclusions

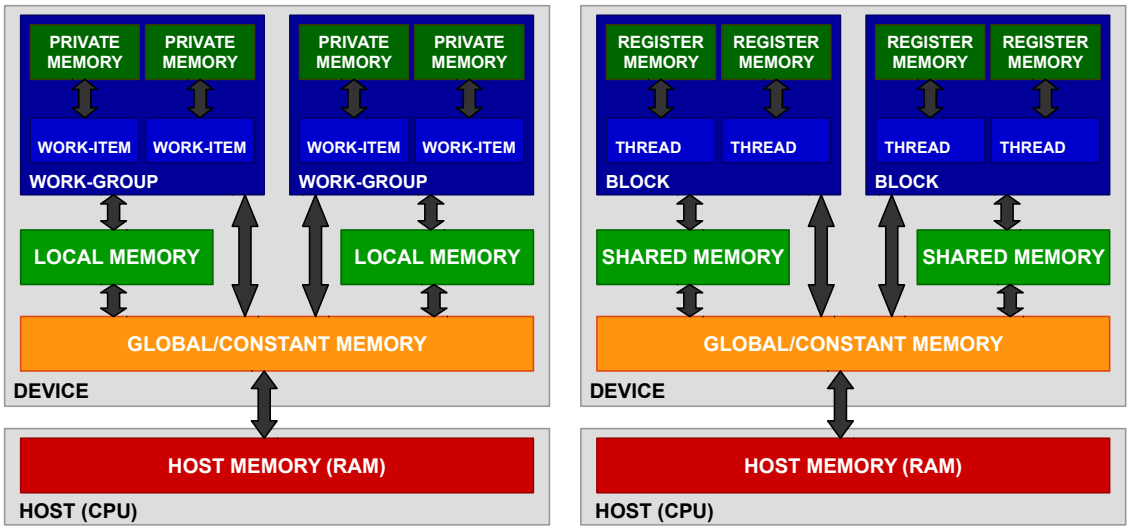
The development of multicore architectures supporting parallel data processing has led to a paradigm shift. Signal processing algorithms has to be considered working asynchronously in separated threads, while the threads are synchronized only when writing in registers (memory). Therefore, there is a need for novel approaches and frameworks allowing an algorithm deployabality in modern signal and data processing systems.

In this article, we touched with recent frameworks for Graphics Processing Units and probably the best known error correction coding technique, LDPC. In a tutorial-based style, we have provided a general parallel approach for decoding any irregular LDPC code and presented a demonstrative application in consistent examples associated with LDPC (14,7) code. The presented approaches are based on the edge-level parallelization, where each thread performs the calculation of a particular value passed through the associated edge (one thread for one edge). The potential acceleration achieved by the parallelization of the calculations grows with the number of edges in the graph. This can lead to interesting applications for long block length codes providing excellent error correcting capabilities.

Hardware devices supporting massively parallel processing algorithms generally include GPUs, which were considered in this tutorial article. Differencies and similarities, in terms of the terminology and source codes, between the OpenCL and CUDA frameworks used for GPU programming were shown in the paper. Benchmarks for the OpenCL and CUDA approaches were performed on the NASA CCSDS (256,128) standard and its protographically expanded derivations [3], and the results were compared against the C++ implementation.

Results shown the acceleration which is up to 22 times compared against C++ with O3 optimization, and up to 58 times compared against C++ compilation without optimization.

Because the OpenCL framework has found utilization in programming FPGA-based systems [15], the proposed algorithms and their potential modifications can be easily used in a wide variety of fast signal processing systems.



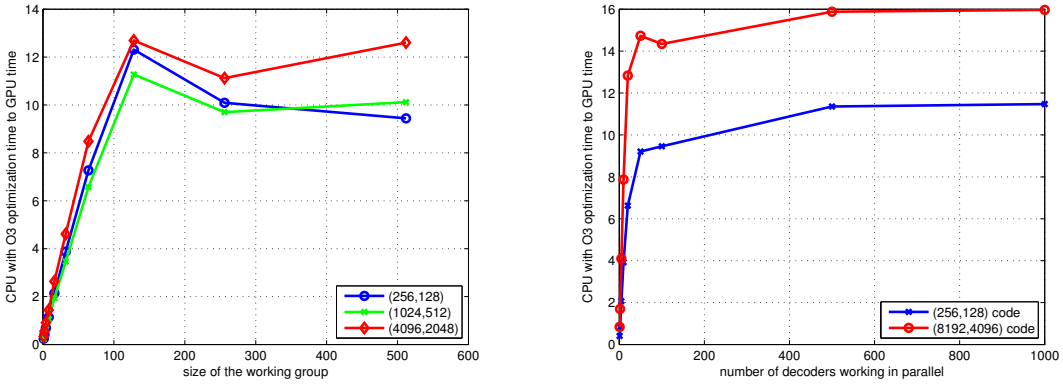
(a) OpenCL programing model.

(b) CUDA programming model.

Fig. 7: OpenCL and CUDA programming models.

TABLE III: Comparison of OpenCL and CUDA syntax rules

command	OpenCL	CUDA
thread synchronization	`barrier(CLK_GLOBAL_MEM_FENCE);`	`__syncthreads();`
kernel prefix	`__kernel`	`__global__`
local memory prefix	`__local`	`__shared__`
get local ID	`int lid = get_local_id(0);`	`int lid = threadIdx.x;`
get global ID	`int gid = get_global_id(0);`	`int gid = blockIdx.x* blockDim.x+ threadIdx.x;`



(a) Acceleration dependence on the block (working group) for 100 decoders running in parallel.

(b) Acceleration dependence on the number of decoders working in parallel when the size of the working group is 512.

Fig. 8: Measured acceleration with the use of CUDA framework.

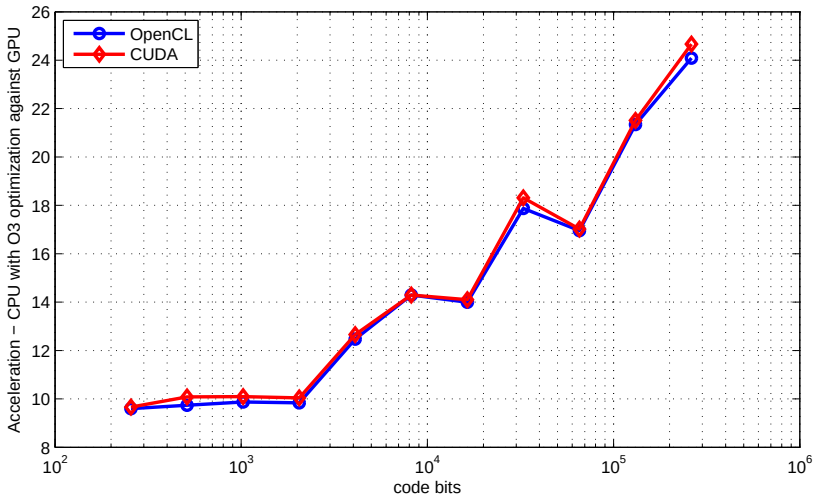


Fig. 9: Acceleration dependence on the length of the code. Comparison for OpenCL and CUDA frameworks (local group of 512 threads and 100 decoders working in parallel) against CPU implementation using C++ compiler with O3 optimization. Time was mesuared for 10000 decoded codewords at E_b/N_0 =2dB.

TABLE IV: 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 mesuared for 10000 decoded codewords at $E_b/N0$ =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

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 to Vladimir Vasilievich Korenkov and Ivan Stekl for arranging the cooperation, to Jan Busa for technical support and especially for professional LaTeX consultations, and to Gheorge Adam for his professional comments and interest in 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] N. Wiberg, *Codes and Decoding on General Graphs*. PhD thesis, Dept. of Electrical Engineering, Lionkoing, Sweden, 1996. Lionkoing studies in Science and Technologz. Dissertation No. 440.

[5] R. M. Tanner, *A Recursive Approach to Low Complexity Codes*. Information Theory, IEEE Transactions, vol.27, no.5, pp.533,547, 1981, doi: 10.1109/TIT.1981.1056404.

[6] 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. 2011

[7] 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.

[8] 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.

[9] 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.

[10] 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. doi: 10.1109/SiPS.2013.6674477

[11] Yue Zhao, Xu Chen, Chiu-Wing Sham, Wai M. Tam, and Francis C.M. Lau, *Efficient Decoding of QC-LDPC Codes Using GPUs*. 11th International Conference, ICA3PP, Melbourne, Australia, October 24-26, 2011, Proceedings, Part I

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

[13] 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. doi: 10.1109/ICASSP.2014.6855061

[14] 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.

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

[16] J. Broulim, P. Broulim, J. Moldaschl, V. Georgiev and R. Salom, *Fully parallel FPGA decoder for irregular LDPC codes*, Telecommunications Forum Telfor (TELFOR), 2015 23rd, Belgrade, 2015, pp. 309-312. doi: 10.1109/TELFOR.2015.7377471

[17] R. Salom and J. Broulim, *LDPC (512,480) genetic design as alternative to CRC in implementation of AODV routing protocol stack*, Telecommunications Forum Telfor (TELFOR), 2015 23rd, Belgrade, 2015, pp. 643-645. doi: 10.1109/TELFOR.2015.7377549

[18] J. Broulim and V. Georgiev, *LDPC error correction code utilization*, Telecommunications Forum (TELFOR), 2012 20th, Belgrade, 2012, pp. 1048-1051. doi: 10.1109/TELFOR.2012.6419390

[19] J. Broulim, V. Georgiev, J. Moldaschl and L. Palocko, *LDPC code optimization based on Tanner graph mutations*, Telecommunications Forum (TELFOR), 2013 21st, Belgrade, 2013, pp. 389-392. doi: 10.1109/TELFOR.2013.6716251

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

[21] Ryan Smith, *NVIDIA Launches Tesla K80, GK210 GPU*. AnandTech (November 17, 2014), http://www.anandtech.com/tag/gpus (last visit 02/06/2016).

[22] *Whitepaper of NVIDIAs Next Generation CUDA Compute Architecture: Kepler GK110/210*. http://www.nvidia.com/object/gpu-architecture.html (last visit 02/06/2016).

APPENDIX

Listing 2: Execution of the kernel function from runtime

(a) OpenCL source code

```
// Create an OpenCL context and command queue
// Build the program from source and create kernel
cl_context context=clCreateContext(NULL,
    CL_DEVICE_TYPE_GPU,&device_id,NULL,NULL,NULL);
cl_cmd_queue cmd_queue = clCreateCommandQueue(
    context,device_id,0,NULL);
cl_program program=clCreateProgramWithSource(context
    ,1,(const char**)&source,NULL,NULL);
clBuildProgram(program,1,&device_id,NULL,NULL,NULL);
cl_kernel kernelSim = clCreateKernel(program, "
    berSimulate", NULL);
// Create buffers
cl_mem codeInfo_obj = clCreateBuffer(context,
    CL_MEM_READ_ONLY,
sizeof(CodeInfo), NULL, NULL);
cl_mem edgesFromVariable_obj =
    clCreateBuffer(context, CL_MEM_READ_ONLY,
    sizeof(Edge) *totalEdges, NULL, NULL);
...
// Set kernel parameters
clSetKernelArg(kernelSim, 0, sizeof(cl_mem),
    (void*) &codeInfo_obj);
clSetKernelArg(kernelSim, 1, sizeof(cl_mem),
    (void*) &edgesFromVariable_obj);
...
// Copy in the buffers
clEnqueueWriteBuffer(command_queue, codeInfo_obj,
    CL_TRUE, 0, sizeof(CodeInfo), codeInfo, 0, NULL,
    NULL);
clEnqueueWriteBuffer(command_queue,
    edgesFromVariable_obj, CL_TRUE, 0, sizeof(Edge)
    *totalEdges, edgesFromVariable, 0, NULL, NULL);
int decoders = 100;
local_item_size = 512;
global_item_size = local_item_size * decoders;
// Execute the OpenCL kernel
clEnqueueNDRangeKernel(command_queue, kernelSim, 1,
    NULL, &global_item_size, &local_item_size, 0,
    NULL, NULL);
// Copy the results back
clEnqueueReadBuffer(command_queue, berOut_obj,
    CL_TRUE, 0, sizeof(double) *maxPoints, berOut,
    0, NULL, NULL);
```

(b) CUDA source code

```
CodeInfo* codeInfo_obj;
cudaMalloc((void**) &codeInfo_obj, sizeof(CodeInfo));
Edge* edgesFromVariable_obj;
cudaMalloc((void**) &edgesFromVariable_obj,
    sizeof(Edge) *totalEdges);
...
// Copy to the device
cudaMemcpy(codeInfo_obj, codeInfo, sizeof(CodeInfo),
    cudaMemcpyHostToDevice);
cudaMemcpy(edgesFromVariable_obj, edgesFromVariable,
    sizeof(Edge) *totalEdges,
    cudaMemcpyHostToDevice);
// Same meaning as local_item_size in OpenCL
int block_size = 512;
int decoders = 100;
int blocks = decoders;
// Kernel execution
berSimulate <<< blocks, block_size >>> (codeInfo_obj,
    edgesFromVariable_obj, edgesFromCheck_obj,
simParams_obj, simulatedPoints_obj,
    edgeDataInitToCheck_obj, edgeDataToVariable_obj,
    edgeDataToCheck_obj,
randomGenVariables_obj, estimation_obj, syndrome_obj,
    noisedVector_obj, berOut_obj);
// Copy the results to host
cudaMemcpy(berOut, berOut_obj,
    sizeof(double) *maxPoints,
    cudaMemcpyDeviceToHost);
```

Listing 3: Decoder function run from kernel

(a) OpenCL source code

```
void decodeAWGN(__global CodeInfo* codeInfo,
__global Edge* edgesFromVariable,
__global Edge* edgesFromCheck,
__global EdgeData* edgeDataInitToCheck,
__global EdgeData* edgeDataToCheck,
__global EdgeData* edgeDataToVariable,
__global int* estimation,
__global int* syndromeOut,
__global double* noisedVector,
int iterations, double sigma2, int lid,
int totalEdges, int lgsize, int glPageStartIndex)
{
   int index_e, p;
   // initial messages to check nodes
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
         initProbCalcAWGN(noisedVector,
          edgesFromVariable, edgeDataInitToCheck,
          sigma2, glPageStartIndex, index_e);
   }
   barrier(CLK_GLOBAL_MEM_FENCE);
   // iteration back to variable nodes
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
         iterateToVariables(...);
   }
   // calculate the estimation
   barrier(CLK_GLOBAL_MEM_FENCE);
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalEdges)
         estimationCalc(...);
   }
   // calculate the syndrome
   barrier(CLK_GLOBAL_MEM_FENCE);
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalEdges)
      syndromeCalc(edgesFromCheck, estimation,
     syndromeOut, glPageStartIndex, index_e);
   }
   barrier(CLK_GLOBAL_MEM_FENCE);
   int cnodes = codeInfo[0].checkNodes;
   int parity = isAllZero(syndromeOut, cnodes,
     glPageStartIndex);
   // if syndrome is ok, decoding is successful
   if (parity == 1) return;
   barrier(CLK_GLOBAL_MEM_FENCE);
   int i;
   // forward and back iterations
   for (i = 0; i < iterations; i++)
   {
      // iteration to check nodes
      for (p = 0; p < totalEdges; p += lgsize)
      {
         index_lid = p + lid;
         if (index_lid < totalEdges)
            iterateToCheck(...);
      }
      barrier(CLK_GLOBAL_MEM_FENCE);
      . . .
      // iteration back to variable nodes
      barrier(CLK_GLOBAL_MEM_FENCE);
     . . .
      // calculate the estimation
      barrier(CLK_GLOBAL_MEM_FENCE);
      . . .
      // calculate the syndrome
      barrier(CLK_GLOBAL_MEM_FENCE);
      // if syndrome is ok, decoding is successful
      parity = isAllZero(syndromeOut, cnodes,
     glPageStartIndex);
      if (parity == 1) return;
   }
}
```

(b) CUDA source code

```
__device__ void decodeAWGN(CodeInfo* codeInfo,
Edge* edgesFromVariable,
Edge* edgesFromCheck,
EdgeData* edgeDataInitToCheck,
EdgeData* edgeDataToCheck,
EdgeData* edgeDataToVariable,
int* estimation,
int* syndromeOut,
double* noisedVector,
int iterations, double sigma2, int lid,
int totalEdges, int lgsize, int glPageStartIndex)
{
   int index_e, p;
   // initial messages to check nodes
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
         initProbCalcAWGN(noisedVector,
          edgesFromVariable, edgeDataInitToCheck,
          sigma2, glPageStartIndex, index_e);
   }
   __syncthreads();
   // iteration back to variable nodes
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
         iterateToVariables(...);
   }
   __syncthreads();
   // calculate the estimation
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
         estimationCalc(...);
   }
   // calculate the syndrome
   __syncthreads();
   for (p = 0; p < totalEdges; p += lgsize)
   {
      index_lid = p + lid;
      if (index_lid < totalSize)
      syndromeCalc(edgesFromCheck, estimation,
     syndromeOut, glPageStartIndex, index_e);
   }
   __syncthreads();
   int cnodes = codeInfo[0].checkNodes;
   int parity = isAllZero(syndromeOut, cnodes,
     glPageStartIndex);
   // if syndrome is ok, decoding is successful
   if (parity == 1) return;
   __syncthreads();
   int i;
   // forward and back iterations
   for (i = 0; i < iterations; i++)
   {
   // iteration to check nodes
      for (p = 0; p < totalEdges; p += lgsize)
      {
         index_lid = p + lid;
         if (index_lid < totalSize)
            iterateToCheck(...);
      }
      __syncthreads();
      . . .
      // iteration back to variable nodes
      __syncthreads();
      . . .
      // calculate the estimation
      __syncthreads();
      . . .
      // calculate the syndrome
      __syncthreads();
      // if syndrome is ok, decoding is successful
      parity = isAllZero(syndromeOut, cnodes,
     glPageStartIndex);
      if (parity == 1) return;
   }
}
```