Utilizing massive parallelism in decoding of modern error-correcting codes for accelerating communication systems simulations

doc. RNDr. Eugen Ružický CSc.,
Faculty of Informatics
Paneuropean University
Bratislava, Slovakia

prof. Ing. Peter Farkaš, DrSc.,
Ing. Tomáš Páleník, PhD,
Institute of Telecommunications
FEI STU,
Bratislava, Slovakia
tomas.palenik@ieee.org

Univ.-Prof. -Dr.-Ing. Markus Rupp.
Institute of telecommunications
Vienna University of Technology
Vienna, Austria

Atílio Gameiro, PhD.
Instituto de Telecomunicações
Campus UniversitÃ¡rio Santiago
Aveiro, Portugal

***Abstract*— In this paper a novel approximate algorithm for massively-parallel decoding of trellis based error correcting codes (ECC) is presented. The potential effect of using such optimized decoder on acceleration of simulations of modern communication systems implementing the most recent communication standards, such as LTE-A (Long Term Evolution – Advanced) is evaluated quantitatively by presenting an original open source implementation in C running on a graphical processor (GPU). The focus of this design is to provide a seamless acceleration to Matlab simulations without breaking compatibility with existing CPU-based simulation frameworks. A quantitative throughput comparison with available open source and proprietary solutions is also presented.**

***Index Terms*—GPU, trellis-decoding, simulation acceleration, optimization.**

I. Introduction

Recent LTE-Advanced specifications, published by 3GPP [1] represent the most advanced communication standards available. They utilize many of the most advanced technological concepts, such as cognitive radio, advanced Multiple-Input-Multiple-Output (MIMO) techniques, coordinated multipoint and powerful Error-Correcting Codes (ECC). Convolutional Codes (CC) and turbo codes chosen by 3GPP for channel coding [2] are the focus of this paper. Several decoders for these trellis-based ECCs exist to date. Some use dedicated DSP hardware [3], others take advantage of an FPGA implementation [4]. Many software implementations also exist. A good overview of existing decoders, along with summarization of throughput claims is given in [5].

The applicability of these commercial decoders in academia is somewhat limited. Academic work can often be described in terms of iterations of proposing a novel idea, testing the concept using simulations, and even modifying the original concept based on simulation results. Therefore for the study and potential improvements of the most recent algorithms, employed in LTE-A and beyond, the presence of two key ingredients is necessary:

First, a good simulation ecosystem is essential. One of the most widely used and successful simulation platforms is Matlab. It combines the powerful statistical tools with the easy extensibility and interoperability with custom production-level code. The drawback of using convenient Matlab simulations is usually a prohibitively long simulation run-time in scenarios that include a complex experimental design that uses computationally intensive system blocks, such as decoders of modern ECCs. A second drawback is the closed nature of Matlab toolboxes – the source code of most of the interesting system components is for commercial reasons unavailable and therefore impossible to be used by outside researchers. This leads to the problem of closed-source reimplementation of even common communication system simulation blocks by many research labs.

The second key component, an open-source communication system simulation library, addresses this problem. Several more or less successful implementations exist to date. A prominent one, written in C++ is the Coded Modulation Library freely available online [6]. This library is then often utilized in other academic libraries that focus on simulations of modern communication systems, such as the LTE simulator maintained by Vienna University of Technology [7].

By experimenting with these tools, important observation can be made: Acceleration of selected extraordinarily-complex system components by manual optimization using a C++ implementation in the form of a MEX file is often not sufficient. The resulting Matlab simulation time is still prohibitively high. The problem of the limitations by the CPU

can be addressed by utilizing the power of a massively parallel Graphical Processing Unit (GPU), able to run thousands of parallel threads. This acceleration potential did not remain unnoticed by industry and recent versions of Matlab Communications System Toolbox already implement many important blocks also supporting GPUs [9,10]. The development of 3rd party CUDA accelerated code, to be seamlessly integrated to Matlab is also supported, while the problem of closed source code and consequent re-implementations remains.

The purpose of this paper is first to introduce a novel massively-parallel approximate logarithmic Maximum Aposteriori Probability (max-log-MAP) Soft Input Soft Output (SISO) decoder design. This decoder can be used in decoding of trellis-based codes, such as convolutional codes. The practical aspects of an open-source implementation are also presented, along with a quantitative evaluation of potential simulation speedup.

The rest of the paper is organized as follows: Section II briefly recaps the well-known concept of MAP decoding. Section III identifies possible parallelisms, that can be utilized in a parallel decoder implementation. Based on observations in this section, Section IV then presents the details of a novel approximate parallel decoder design. Section V then deals with practical issues that arose during our research work while Section VI presents quantitative comparison with other existing CPU and GPU enabled implementations. Section VII then concludes the paper.

II. SISO DECODING ON TRELLIS

Practical turbo codes present in LTE & LTE-A consist of constituent codes connected together using an interleaver. Since both the constituent codes are convolutional codes and, as specified in [2], the alternative to the turbo-design is also a CC, it is reasonable to focus on this particular class of codes. One of the encoder designs a Recursive Systematic CC (RSC) is shown in Fig. 1:

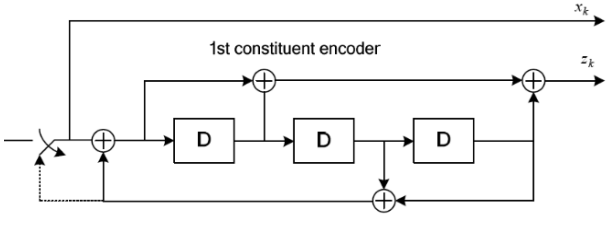

Fig. 1. Convolutional encoder in LTE-A.

All well-known decoder algorithms utilize a trellis representation to describe their operation [8]. As shown in Fig. 2, the trellis for even this simple encoder contains many edges. Decoding algorithm operate by sending messages along these edges in a structured way. The logarithmic version of MAP decoding algorithm estimates the probabilities of transmitted symbols one-by-one, based on the observation of the whole received sequence. To achieve this, three different logarithmic metrics A,B,Γ must be defined. These metrics are logarithmic equivalent of the well-known α, β γ metrics present in the BCJR (or MAP) decoding algorithm. For an RSC code of rate R=1/2, AWGN channel and BPSK modulation, the branch metric Γ may be defined as follows:

$$\Gamma_{(p,q)} = x^{(p,q)} \cdot \frac{\lambda_p}{2} + \frac{Lc}{2} \cdot x^{(p,q)} \cdot r^0 + \frac{Lc}{2} \cdot v^{(1,p,q)} \cdot r^1 , \quad (1.)$$

where p and q are the states of a trellis segment incident with a common edge, x is the value of a data bit that causes the state transition represented by that edge, $\boldsymbol{v}$ is the encoder output (In case of a systematic code x is the first element of $\boldsymbol{v}$.) and $\boldsymbol{r}$ is the vector or received samples (in Eq. (1.) both $\boldsymbol{r}$ and $\boldsymbol{v}$ are vectors of two components). Here, Lc is a channel reliability factor defined as

$$Lc = \frac{2\sqrt{R \cdot E_b}}{\sigma^2} \quad , \quad (2.)$$

and λ_p is the prior LLR (usually initialized to zero).

During the decoding process, Γ must be calculated for all possible state transitions in one trellis segment (section) and also for all segments of the whole codeword trellis. The node metrics - forward metric A and backward metric B may be then defined using branch metrics as follows (symbol Ψ denotes the set of all states (nodes)):

$$A(q) \approx \max_{p \in \Psi}(A(p) + \Gamma(p,q)) \quad (3.)$$

$$B(p) \approx \max_{q \in \Psi}(\Gamma(p,q) + B(q)) \quad (4.)$$

The posterior LLR, based on the observation of the whole received sequence $\bar{r}$ can then be calculated for each trellis segment:

$$\lambda(x/\bar{r}) = \max_{(p,q) \in S_1}(A_{(p)} + \Gamma_{(p,q)} + B_{(q)}) - \max_{(p,q) \in S_0}(A_{(p)} + \Gamma_{(p,q)} + B_{(q)}) , \quad (5.)$$

where S_1 is the set of edges that represent the state transition caused by the value "1" of the input data bit, while S_0 corresponds to the edge set caused by incoming zero symbol.

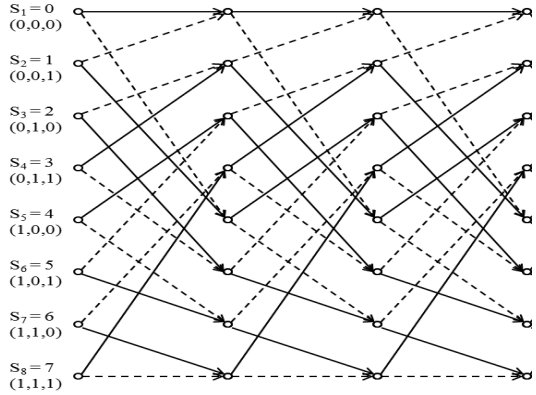

Fig. 2. Three segments of a trellis of RSC defined in Fig.1. Dashed lines indicate value of input data bit 1, continuous lines zero value.

III. PARALLELISMS IN TRELLIS-DECODING

Several observations regarding the potential for parallel processing can be made by careful review of the trellis description in previous section:

First, the branch metric Γ is position independent, that means, the values of Γ for edges of one particular trellis segment do not depend on values of any metric in the preceding or later segments. Furthermore the value of $\Gamma(p,q)$ for a given edge within a trellis segment does not depend on the branch metric of any other edge. If the problem is maximizing the utilization of parallelism, the computation of branch metrics represents the best possible situation. In case of an LTE-A defined R=1/3 CC with 64 states and maximum input bit sequence length of 6144 symbols [2] this leads to theoretical maximum of almost 400 thousand of branch metrics that can be computed in parallel. In real implementation, this satisfying number will be substantially limited by many practical factors.

The situation with forward and backward metrics A,B is not so fortunate. From Equations (3.) and (4.) it is obvious, that both are position dependent, which means that the values of these metrics in one trellis segment depend on values of preceding segments (one way or another). Thus the potential for parallelism is quite low: the maximum number of metrics computed in parallel is equal to the number of trellis states, which in LTE-A is limited to 64 (or 8 for the constituent CC of the turbo-decoder).

Finally, the computation of posterior LLRs can be in theory performed in parallel for all the trellis segments, provided that all the node metrics A and B are already available. This condition is easily fulfilled in theory. However, once practical hardware limitations such as memory size and latency come to play, this condition will likely not hold. The estimate of realizable parallelism strongly depends on the overall structure of a concrete implementation of the decoding algorithm.

IV. NOVEL PARALLEL ALGORITHM DESIGN

When designing a parallel decoder to run on a GPU, it is most important to focus on the structure of the parallel hardware itself and tailor the algorithm so that it copies this structure. That is the only way to achieve full utilization of the computational power provided by the GPU. The problem is of course that the structure of the MAP decoding algorithm and the general purpose parallel hardware do not really fit much. The design provided here tries to fit the decoding to existing nVidia hardware used during our research, with specific numbers easily replaced by specifications of newer hardware.

The basic idea of partitioning the decoding problem could be described a *horizontal partitioning*: Each of the parallel threads performs all the decoding steps described by the previous section, but for only one trellis segment.

This is a very straightforward design that allows for intensive reuse of code from a single thread design where Equations (1. – 5.) must be computed for all segments in a serial fashion. The parallel lightweight-thread operation can be then best described by the following pseudo code:

```
1. Compute branch metrics
     Loop X times {
           2. Compute Alphas estimates
           3. Compute Betas estimates
           4. Forward Alphas to the right
           5. Forward Betas to the left
     }
  6. Compute final posterior metrics
```

Fig. 3. Algorithm of one thread is performed on one trellis section.

As shown in Fig. 4, this design is based on the assumption that each thread can access only the received samples that correspond to its trellis segment, and that the A and B metrics are propagated to its neighboring threads.

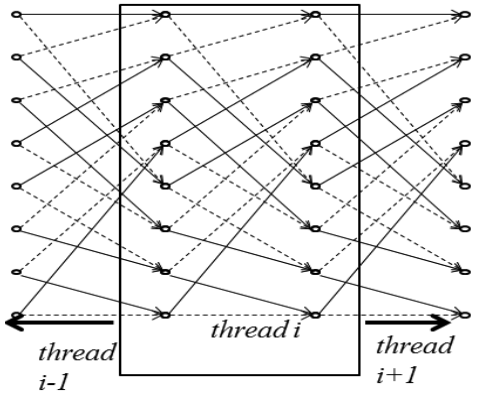

Fig. 4. Algorithm of one thread is performed on one trellis section.

It also implies the necessity of local memory used by each thread, along with a shared memory to be used for inter-thread communication (point 4. and 5. of the algorithm). The A and B metrics estimates can be initialized to values corresponding to log-uniform distribution and they slowly converge to their correct values as they propagate through the trellis in the forwarding steps.

Somewhat unclear is the value of X – the number of iterations of the loop. To clarify this , the structure of a serial single-threaded CPU designs is helpful:

```
Loop through all trellis segments {
  1. Compute branch metrics
  2. Compute and store Alphas
  3. Compute and store Betas (backwards)
}
Loop through all trellis segments {
  4. Compute final posterior metrics
}
```

Fig. 5. Algorithm of one thread is performed on the whole trellis.

The basic idea is here that in order to get the same (correct) results as with a serial algorithm, there must be as many threads run in parallel as there are trellis segments. Furthermore, the number of iterations X must be quite large in order to propagate the correct A metrics from the first trellis segment up to the last (and the B metrics in the opposite direction). In fact, in order to produce exactly the same results,

X must be as high as the number of trellis segments. If this number has to be used, all that is achieved by this parallel design would be a massive increase in parallel processor utilization, without any desired reduction of the processing time. Since the number of threads equals the number of trellis sections K, for instance as defined in [2] to range from 40 to 6144, this design would be terribly wasteful. The following observation comes to remedy the situation:

Observation 1: In a max-log MAP decoding the forward metrics A has a much stronger influence on nearby trellis segments than on more distant segments. The greater is the distance between two trellis segments, the weaker is the dependence of the A metrics between them.

This observation comes from a detailed study of Equation (3.) and can be also observed by experiment. The same hold true also for the B metrics. Based on this observation, an approximate decoder design can be derived by simply setting the value of X to a lower value Δ. For maximizing the speedup of this algorithm, it is desirable that Δ is as small as possible. This value can be determined experimentally. In our experiments value of Δ=32 performed quite well.

With this value in mind, it is possible to estimate a potential theoretical decoding speedup comparing to a single-threaded design by defining a *speedup factor sf* :

$$sf = \frac{K}{\Delta}. \qquad (6.)$$

The speedup heavily depends on the choice of K and for LTE-A can be expected between 1.25 and 192. For longer codewords it will be even better.

Unfortunately there are some less obvious problems with this approximate design: It can be expected that such a designdoes not perform well in an iterative multi-decoder design, such as the case of decoding of turbo codes that utilizes two constituent decoders that share extrinsic information by an interleaver. It is precisely the information that gets discarded by this approximation that is leveraged by the interleaver in a turbo design. This can have a very negative effect on the error performance of such a configuration.

V. Practical Considerations

The design presented in the previous section is simple and clear and allows for a potentially great decoding speedup when the circumstances are right – when the codeword is sufficiently long and when there are enough hardware resources on the parallel processor to natively handle all the threads with their memory demands. While studying the specification for even the most advanced CUDA enabled GPUs [10], it becomes clear that this is not going to be reality. There are many practical limitations constraining the local memory a thread can use, the overall number of threads, the number of threads that can be resident in a multiprocessor (MP) and most importantly the amount of fast on-chip memory to use for sharing the forward and backward metrics between the concurrent threads.

Actually, the shared on-chip memory size turned out to be the greatest limiting factor. The hardware used – an nVidia GeForce GTX 680 has only 48kB of this shared memory per MP. This value isn't a technological anomaly – one cannot expect the next generation of GPU cores to suddenly substantially improve this number. If more memory is needed for inter-thread communication, a much slower off-chip memory has to be used now, and will continue to do so. With latencies two orders of magnitude higher, this could render the decoder designed in previous section completely useless.

Because of these limitations, the design presented must be altered to a slightly more complex form with less potential parallelism, but more fitting to existing hardware limits.

A. More practical decoder design

The trellis will be processed in chunks of fixed length determined by the sum of all HW limitation. For the given hardware a chunk size of 512 trellis segments was selected based on the sum of technological constraints. These chunks are then processed in a serial manner. This serial processing is the key speedup limiting factor. Fig. 6 shows the partitioning of a trellis 2048 segments long into four chunks. The thick arrow indicates the direction of serial processing of chunks. Each chunk is processed by 512 parallel threads as described in section IV.

The problem with this approach is that the A and B metrics that propagate through the trellis must also propagate between the chunks. Since the progress of metrics A is in natural alignment with the processing of chunks, this problem is easily solved by copying the A metric buffers from the rightmost thread to the A metric buffers of the leftmost thread on chunk switch (Indicated by dashed vertical line in Fig. 6).

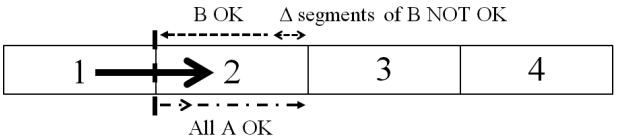

Fig. 6. Segmentation of the trellis to chunks of 512 segments illustrating a B metric propagation problem .

The problem is worse with the backward metric B, whose processing direction defined by the BCJR algorithm is opposite of the chunk processing. On the other hand, for an approximate decoder design based on observation 1, the calculation values of B only takes into account Δ trellis segments to the right. Therefore most of the B metrics in a chunk will be computed correctly while only in the Δ rightmost segments will not.

To correct also this problem, even this practical design had to be modified slightly – the chunks must be enlarged by Δ segments and they will have to *overlap* by exactly those extra Δ segments as illustrated in Figure 7. With this overlapping, after Δ iterations of algorithm shown in Fig. 3, there will be exactly Δ rightmost trellis sections with incorrect values of B metrics. Since the trellis chunks overlap, these results will not be used any further – their only purpose is to supply correct B

metrics to the rightmost trellis segment that counts (indicated in Fig. 7 by a vertical line).

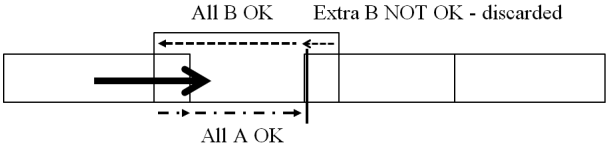

Fig. 7. Segmentation of the trellis to chunks of 512 +32 segments with 32 segments overlap, solving the B metric propagation problem.

The processing of these Δ extra segments presents an additional overhead.

VI. Quantitative evaluation

The quantitative evaluation of the new parallel decoder implementation consist of evaluation of two different aspects: first the error performance of the new algorithm must be evaluated to assess the possible negative impact of the approximation introduced as a result to observation 1. A right shift of the waterfall curve in the decoding of a convolutional code can be expected.

As shown in Fig. 8, simulations comparing a reference single threaded full max-log-MAP implementation (without the described approximation with an intentionally small choice of Δ = 16) reveal more than one dB difference in error-performance:

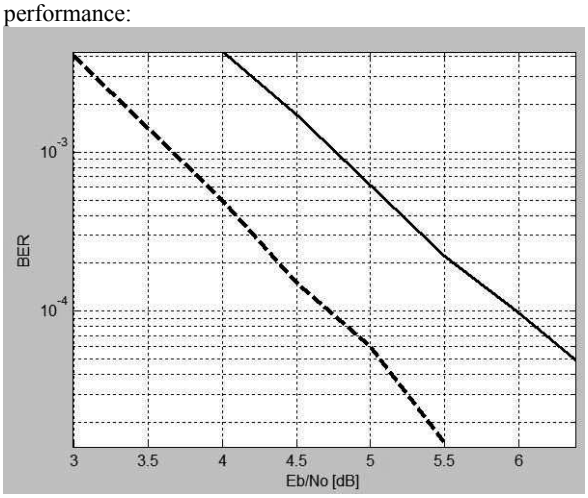

Fig. 8. Waterfall curve for max-log-MAP decoding of CC: single threaded full implementation (dashed) –vs- approximate parallel implementation.

As the value of Δ gets larger, the difference between the curves grows smaller, while the simulation time increases by the resulting increased processing.

The second important aspect to evaluate is the practical simulation speedup brought about by the decoder parallelization. This can be measured in more than one way. For the purpose of this paper the *throughput* was selected as a measure, to be able to compare results with other existing decoders. For the sake of being most useful to practical utilization of simulations, the following definition of throughput was employed:

$$Throughput = \frac{Number\ of\ simulated\ bit\ transfers - not\ data\ bits!}{Total\ simulation\ time\ spent\ in\ decoder} \quad .(7.)$$

Experiments were made with several available MAP-derived decoders, which were run using codeword of size K = 2048. The decoders were configured in a turbo decoding setup, using the LTE-A turbo code consisting of two constituent convolutional codes defined in [2]. Reason for this setup will be given later.

The simulation was always configured to obtain at least 100 erroneous bits, while the SNR range of the simulation was chosen in the waterfall area near a target BER of 10^{-4}. This value was chosen because it represents a reasonably small error ratio, while enabling to perform the throughput tests in reasonable time. Since the decoders implement various variants of decoding algorithm, and the statistical character of the simulation, there are small differences in error performance – the E_b N_0 value at the crossing of BER=10^{-4} is indicated as an extra column in Table I. All tests were run on a quad-core Intel Core i5 650 CPU system with an nVidia GTX680 GPU attached.

It can be argued that, because of the potential early stopping criterion implemented differently in various decoders, throughput depends on SNR value. Since the SNR range was set the same in all experiments, the value of throughput in the third column of Table I represents an average.

TABLE I. SISO decoders comparison In Turbo Setup

Various turbo-decoders throughput comparison			
	Implementation	***Throughput [kbps]***	***Eb/N0 [dB]***
1	Mathworks CPU	225	1.56
2	Mathworks GPU	107	1.54
3	Mathworks GPUopt	339	1.56
4	IS CML	196	N/A
5	Matlab CPU naive	1.17	1.38
6	Matlab CPU MEX	167	1.38
7	Matlab GPU exp	1424	N/A

The decoders in Table I deserve further explanation:

The first three decoders are implemented directly by Mathworks and are included in the Communications System Toolbox in the latest Matlab release (R2013a) and all are proprietary with no source code available. The GPU implementations require also the Parallel Computing toolbox to be present. All were tested by running the demo simulations provided by the toolbox.

The first one post fixed here as CPU is a MEX implementation that runs on the CPU. During this simulation, actually two cores of the CPU were utilized, which indicates that the simulation actually used two threads. However this was hidden from the user, and benefitted him with a high

throughput value. On the other hand, it makes the comparison with other decoders slightly less clear.

The second Mathworks decoder, denoted GPU, utilizes the parallel processing of a CUDA-enabled GPU. As shown in Table I, the throughput is even lower than of the CPU implementation. This is because the simulation is not structured to specifically facilitate GPU processing. It is exposing the latencies inherent in transferring the data between main system memory and on-device GPU memory through the PCI-Express bus, thus reducing the throughput significantly.

The third decoder – GPUopt shows the result for the same parallel CUDA decoder when the simulation is structured in a way where multiple codewords are processed together in a block, which limits the data transfers latency problem.

The fourth implementation is an open-source IS CML [6] C++ implementation run on CPU as a MEX file. This is a widely-known and highly optimized single-thread implementation that gives very good results and can serve as a good reference for throughput measurement.

The fifth implementation is a naive Matlab-only decoder based on the description provided by textbooks [8] with no optimizations implemented in MEX files and no early stopping criterion present that would terminate the decoding process in case of faster convergence. The purpose of this implementation is to provide insight, how much optimized the other decoders really are. It can be seen that the other decoders outperform the pure Matlab code by two orders of magnitude.

The sixth decoder is a single-threaded C++ MEX decoder, run on CPU, implemented as a part of work on this paper. This implementation is fully open source published under a very permissive LGPL license to be used widely by industry and academia.

Finally the seventh implementation is the experimental massively parallel CUDA implementation, run on GPU directly from Matlab with parameter Δ set to 32 segments, it outperforms other decoders significantly. This decoder was again implemented as an open source LGPL code. The problem with this implementation is that while it brings significant speedup, it also performs worse in terms of BER. Furthermore, as confirmed by the experimental turbo-setup, the approximations are in direct opposition to the code structure introduced in turbo design by the interleaver, which results in a prohibitively high error-floor of slightly more that BER= 10^{-4}, which renders this design incompatible with turbo decoding setup.

VII. Conclusion

A novel massively-parallel decoder structure for a trellis-based posterior SISO decoding was presented along with experimental evaluation of resulting simulation acceleration potential and comparison with various existing decoders. The approximations to the max-log-MAP algorithm enabled faster parallel processing while slightly reducing error performance of the decoder and prohibiting its potential utilization in an iterative turbo-decoding setup. Two version of the decoder were implemented in C++ to be used together with Matlab environment as MEX files: a single threaded full max-log-MAP decoder and an experimental parallel CUDA implementation. Both are available to the academic community under a very permissive open-source LGPL license.

Acknowledgment

This work was supported by, Slovak Research and Development Agency under contracts SK-AT-0020-12 and SK-PT-0014-12, by Scientific Grant Agency of Ministry of Education of Slovak Republic and Slovak Academy of Sciences under contract VEGA 1/0518/13, by EU RTD Framework Programme under ICT COST Action IC 1104 and by Visegrad Fund and National Scientific Council of Taiwan under IVF–NSC, Taiwan Joint Research Projects Program application no. 21280013 "The Smoke in the Chimney - An Intelligent Sensor - based TeleCare Solution for Homes".

References

[1] 3GPP The Mobile standards, LTE-Advanced. Available online: <http://www.3gpp.org/LTE-Advanced>

[2] 3GPP, LTE: Evolved Universal Terrestrial Radio Access (E-UTRA); Multiplexing and channel coding (3GPP TS 36.212 version 10.3.0 Release 10) ETSI TS 136 212 V11.0.0

[3] Song, Y., Liu, G., & Yang, H. (2005). The implementation of turbo decoder on DSP in W-CDMA system. In International conference on wireless communications, networking and mobile computing (pp. 1281-1283).

[4] K. G. Nezami, S. D. Walker and P. W. Stephens, "An FPGA Implementation of a Memory Efficient, Low Complexity Turbo Decoder Architecture for TETRA Release 2 Application", Proc. of 12th European Wireless Conference - EW2006, Apr. 2006

[5] M. Wu, Y. Sun, G. Wang, J. Cavallaro. Implementation of a High Throughput 3GPP Turbo Decoder on GPU. Journal of Signal Processing Systems (JSPS), 2011. DOI 10.1007/s11265-011-0617-7.

[6] Iterative Solutions CML: Coded Modulation Library, Available: <http://www.iterativesolutions.com/Matlab.htm >

[7] Technische Universitaet Wien, LTE and Wimax simulators, Online:<www.nt.tuwien.ac.at/downloads/featured-downloads>

[8] MOON, T. K. Error Correction Coding - Mathematical methods and Algorithms. New Jersey : Wiley,2005. ISBN 978-0471648000.Halfhill, T., Parallel processing with cuda. In Microprocessor report, Arizona, USA, January 2008.

[9] MathWorks, Matlab 2010b Parallel Computing toolbox documentation. Massachusets, USA, September 2010. Available:<http://www.mathworks.com/help/toolbox/distcomp>

[10] NVIDIA CUDA: Programming Guide, available http://www.nvidia.com

.