Analog Integrated Circuits and Signal Processing manuscript No.
(will be inserted by the editor)

Complexity Analysis of Software Defined DVB-T2 Physical Layer

Stefan Grönroos · Kristian Nybom · Jerker Björkqvist

Received: date / Accepted: date

Abstract The second generation terrestrial TV broadcasting standard from the Digital Video Broadcasting (DVB) project, DVB-T2, has recently been standardized. In this article we perform a complexity analysis of our software defined implementation of the modulator/demodulator parts of a DVB-T2 transmitter and receiver. First we describe the various stages of a DVB-T2 modulator and demodulator, as well as how they have been implemented in our system. We then perform an analysis of the computational complexity of each signal processing block. The complexity analysis is performed in order to identify the blocks that are not feasible to run in realtime on a general purpose processor. Furthermore, we discuss implementing these computationally heavy blocks on other architectures, such as GPUs (Graphics Processing Units) and FPGAs (Field-Programmable Gate Arrays), that would still allow them to be implemented in software and thus be easily reconfigurable.

Keywords DVB-T2 · SDR · CUDA · x86

1 Introduction

The DVB-T (Digital Video Broadcast Terrestrial) system for digital television broadcasting is widely used for broadcasting around the world. As high bitrate High-Definition Television (HDTV) broadcasts become more prevalent, however, the need for a more spectrum efficient standard increases. The DVB-T2 standard [7, 21] has been developed to address this need.

Joukahaisenkatu 3-5, 20520, Turku, Finland
E-mail: stefan.gronroos@abo.fi
E-mail: kristian.nybom@abo.fi
E-mail: jerker.bjorkqvist@abo.fi

Compared to its predecessor, DVB-T2 has a more efficient physical layer using state-of-the-art technologies to achieve close to optimal performance in terms of true bitrate in quasi error free conditions: concatenated LDPC (Low-Density Parity-Check) and BCH (Bose-Chaudhuri-Hocquenghem) coding, rotated high-order QAM (Quadrature Amplitude Modulation) constellations, MISO (Multiple Input Single Output) antenna reception, efficient time and frequency interleaving, large FFT (Fast Fourier Transform) sizes, etc. All in all, DVB-T2 is expected to give an increase in capacity (bit rate) of at least 30% as compared to DVB-T, and for some configurations up to 70% [21]. The upcoming next generation mobile TV broadcasting system, DVB-NGH (Next Generation Handheld), is also expected to be based on DVB-T2.

In this article, we present a work-in-progress software defined DVB-T2 modulator and demodulator using the GNU Radio framework [2]. In addition to other benefits of a fully software defined implementation of DVB-T2, the reconfigurability of such an implementation can be very beneficial in developing future standards such as DVB-NGH. The GNU Radio based project is incomplete, and a number of parts have not yet been finished. However, the implementation is directly based on a DVB-T2 simulator, which is more complete.

We examine the computational complexity of the various parts of a DVB-T2 modulator and demodulator through benchmarks performed on the various signal processing blocks of our simulator. This gives us an indication of the complexity of each block in the system, and shows where the most effort needs to be placed when aiming for a realtime system. We are not comparing the results to ASIC (Application-Specific Integrated Circuit) implementations, as our main motivation for this work is to analyze the applicability of

a DVB-T2 system on a generic SDR (Software Defined Radio) platform.

Furthermore, we also discuss alternative implementations of the most computationally complex blocks on platforms such as GPUs (Graphics Processing Units) and FPGAs (Field-Programmable Gate Arrays), which may allow us to reach realtime performance, while still retaining the reconfigurability of a software defined implementation. Some signal processing blocks have been implemented on a GPU architecture by the authors, and these are analyzed alongside the general purpose CPU (Central Processing Unit) implementations.

Related work can be found in [19], where a GNU Radio implementation of a DVB-T modulator is described. A software defined DVB-C2 [6] (cable transmission) implementation, created within the GNU Radio framework, is discussed in [15]. As the DVB-C2 standard is quite similar in many ways to the DVB-T2 standard, we refer to the solutions presented in [15] in later sections. An SDR implementation of a DVB-T2 receiver is described in [16], where most of the system has been realized using FPGAs and DSPs (Digital Signal Processors). In contrast, our implementation aims at keeping most functionality, if possible, on general purpose, commodity hardware.

The article is laid out as follows. In section 2 we describe the various parts of a DVB-T2 system. In section 3 we introduce the experimental setup on which measurements were performed, as well as describe the implementation choices made for some signal processing blocks. Section 4 contains the results of the measurements. A discussion on results, as well as alternative efficient algorithms for implementing the most problematic parts of a DVB-T2 system is contained in section 5. Finally, we conclude the article in section 6.

2 DVB-T2 System Architecture

In this section, we describe the main building blocks of a DVB-T2 modulator, as defined in [7]. These are the blocks that were benchmarked for this article.

The input data streams, which are in the form of MPEG-2 Transport Streams or GSE (Generic Stream Encapsulation) encapsulated data are first split into one or more Physical Layer Pipes (PLPs), where each PLP may use different coding and modulation. In this article, we only consider a single-PLP system. The first module of such a system is the Input Processing module. This module converts the input data streams into DVB-T2 baseband frames. This module is not discussed further in this article, however. After passing through the Input Processing module, each baseband frame is processed by the Bit Interleaved Coding & Modulation (BICM) module, which contains the following stages (in order):

- FEC (Forward Error Correction) coding. DVB-T2 uses an outer BCH code, as well as an inner LDPC code. The resulting FEC blocks can be either 16200 (short code) or 64800 bits (long code) long. 6 different LDPC code rates are available.
- Bit Interleaver (not used for QPSK modulation). Consists of parity bit interleaving, followed by column twist interleaving.
- Mapper, which maps bits onto constellations. Produces cell words.
- Constellation Rotation, if rotated constellations are used. The cell values produced by the mapper are rotated in the complex plane (the angle depends on the modulation used), and the imaginary part is cyclically delayed by one cell.
- Cell Interleaver. Used to uniformly spread the cell words of a FEC block.
- Time Interleaver. In this block, cells of groups of FEC blocks, making up TI-blocks – which in turn make up Interleaving Frames – are interleaved.

The BICM module is followed by the Frame Builder, the task of which is to assemble so-called T2 frames from the Interleaving Frames of each PLP, as well as various signaling data. Cells that are going to be included in one OFDM (Orthogonal Frequency-Division Multiplexing) symbol are grouped together in this module. The Frame Builder module also includes frequency interleaving, where the cells belonging to an OFDM symbol are interleaved, providing interleaving in the frequency domain.

The frames produced by the Frame Builder are sent to the OFDM Generation module for further processing. The OFDM Generation module includes the following parts:

- MISO processing. This is optional, and allows for the generation of two slightly different output signals for transmission from two groups of transmitters.
- Pilot Insertion. Cells containing reference information are inserted at to the receiver known points in the transmitted signal. Pilots can be, among other uses, used to aid in synchronization and channel estimation at the receiver.
- IFFT (Inverse Fast Fourier Transform). The OFDM symbols are modulated here. FFT sizes of 1K, 2K, 4K, 8K, 16K, and 32K are supported by the standard.
- PAPR reduction. This optional part allows us to reduce the Peak-to-Average Power Ratio (PAPR) of the transmitted signal.

- Guard interval insertion. This is where we insert guard intervals, which are a cyclic continuation of the useful part of an OFDM symbol.
- P1 symbol insertion. The P1 symbols are special 1K OFDM symbols that are used mainly to aid the receiver in recognizing and tuning in to the DVB-T2 signal.

The output of the OFDM Generation module is a signal ready for transmission. In the following section, we discuss how the DVB-T2 simulator was used to benchmark the various functional blocks of a DVB-T2 system.

3 Experimental Setup

In this section follows an overview of both the software and hardware setup used for benchmarking in order to produce the results presented in section 4. We also briefly describe our implementations of the constellation demapper, LDPC decoder and demodulator-side FFT blocks on a GPU, as well as how they differ from the CPU implementations.

3.1 Software implementation

As mentioned in the introduction, we have implemented some of the functional blocks discussed in section 2 within the GNU Radio environment. The actual functionality is written in C and C++ and is directly based on the building blocks of a DVB-T2 simulator developed at Åbo Akademi University. Since the GNU Radio implementation lacked some of the functionality implemented in the simulator, we have benchmarked the blocks within the simulator instead of within the GNU Radio implementation. This should not affect the results significantly, as the code used in our GNU Radio implementation was directly based on the simulator code.

The simulator was not 100% complete, and lacked support for some configurations such as multiple PLPs, some pilot patterns etc. Neither the simulator nor the GNU Radio implementation included the necessary support for tuning in to a DVB-T2 channel, nor for various forms of receiver synchronization at the time when this article was written. As a consequence, some blocks, primarily on the receiver side, are significantly less complex in our implementation than they would be in an actual implementation.

Aside from synchronization functionality, the pilot removal block is perhaps the one that differs most from an actual implementation. This block would need to calculate channel estimates in a real implementation, while our implementation receives perfect channel estimates precalculated from a channel simulator. Also, the BCH decoder part of FEC decoding had not been implemented at the time of writing.

Synchronization as well as channel estimation functionality was implemented in the software DVB-C2 receiver implementation described in [15]. Due to the typically very few and short echoes and overall high signal quality in cable transmissions, several simplifications could be made to these components of the receiver. These simplifications are likely not applicable to the same degree to DVB-T2 receivers, much due to the longer and more plentiful echoes, as well as the lower signal quality in typical terrestrial broadcasting environments.

Also worth noting is that the constellation rotation (modulator) and derotation (demodulator) blocks are only used at runtime to insert and remove the Q-delay specified for rotated constellations, and are thus very fast. Actual rotation or derotation of the constellations does not happen at runtime, but is precalculated during initialization, as there is no need to do this at runtime.

3.1.1 CUDA

With modern GPUs being very powerful in data parallel computing tasks, we also implemented the constellation demapper, LDPC decoder, and FFT blocks on such hardware. These implementations were programmed for the NVIDIA CUDA (Compute Unified Device Architecture) [18] architecture.

In the CUDA C programming model, we define *kernels*, which are functions that are run by many threads in parallel. The threads executing one kernel are split up into thread blocks, where each thread block may execute independently, making it possible to execute different thread blocks on different GPU cores. The developer can define the number of threads that will run a kernel, as well as how many thread blocks the threads will be split into, within certain limits. While global memory accesses are often quite costly on GPUs, each thread block is also assigned a certain amount of fast shared memory and an L1 cache (in fairly recent GPUs). 32 threads of one thread block, making up a thread *warp*, are executed in a SIMD-like (Single Instruction, Multiple Data) fashion on one GPU core.

3.1.2 Demapper

The task of the demapper, given a 2^m-QAM configuration (conveying m bits per cell), is to convert each

received complex cell value into m Log-Likelihood Ratios (LLRs), where a positive LLR indicates that the corresponding bit was most likely transmitted with a value of 1, and vice versa. The perfect LLR values can be calculated [5,17] as:

$$LLR(b_i) = \ln\left(\frac{\sum_{x\in C_i^1} e^{-\frac{(I-\rho_I I_x)^2+(Q-\rho_Q Q_x)^2}{2\sigma^2}}}{\sum_{x\in C_i^0} e^{-\frac{(I-\rho_I I_x)^2+(Q-\rho_Q Q_x)^2}{2\sigma^2}}}\right) \quad (1)$$

where b_i corresponds to the ith bit (out of m). C_i^0 is the set of original constellation points (in the diagram), where bit i is 0, and C_i^1 is the set of points where bit i is equal to 1. I and Q signify the received I and Q components, respectively, while I_x and Q_x represent the I and Q components of point x in the constellation diagram. ρ_I and ρ_Q represent the amplitude-fading factors of the channel, while σ^2 represents noise variance. The amplitude-fading factors as well as noise are estimated at the receiver based on, for example, the known transmitted values of pilot patterns.

The max-log approximation:

$$\ln(e^{a_1} + \cdots + e^{a_k}) \approx \max_{i\in[1,k]} a^i \quad (2)$$

can be used to form an approximation of the LLR calculations, giving us the formula:

$$LLR(b_i) \approx \frac{1}{2\sigma^2}\Big\lfloor \min_{x\in C_i^0}((I-\rho_I I_x)^2 + (Q-\rho_Q Q_x)^2) - \min_{x\in C_i^1}((I-\rho_I I_x)^2 + (Q-\rho_Q Q_x)^2)\Big\rfloor \quad (3)$$

For the CPU version of the demapper, we used the above max-log approximation, in addition to dividing the constellation diagram into 4 overlapping quadrants as described in [17]. Figure 1 shows the division into quadrants for a rotated 16-QAM constellation diagram. The correct quadrant can be chosen on the basis of the signs of the incoming constellation points' I and Q components. One quadrant contains both minimum distance points needed to calculate the LLR using the max-log approximation, and thus we only calculate distances to $(\frac{\sqrt{2^m}}{2}+1)^2$ points instead of to all 2^m points.

The GPU algorithm did not use the quadrant-based approach used for the CPU implementation, nor did it use the max-log approximation. We implemented this algorithm as two CUDA kernels, where the first kernel calculates:

$$\forall x \in C : DIST(x) = e^{-\frac{(I-\rho_I I_x)^2+(Q-\rho_Q Q_x)^2}{2\sigma^2}} \quad (4)$$

where C denotes the entire set of 2^m constellation points in the constellation diagram. One thread is created on the GPU for each of the 2^m distance calculations. The second kernel calculates the LLRs for each of the m bits in parallel. The kernel thus completes the computation of $LLR(b_i)$ in equation 1 by calculating:

$$LLR(b_i) = \ln\left(\frac{\sum_{x\in C_i^1} DIST(x)}{\sum_{x\in C_i^0} DIST(x)}\right) \quad (5)$$

for each $0 \le i < m$. Here one thread is thus created for each of the m bits. While m threads is not enough to fully utilize the GPU, we grouped together calculations for all received cells belonging to the same FEC block in order to enable better parallelism.

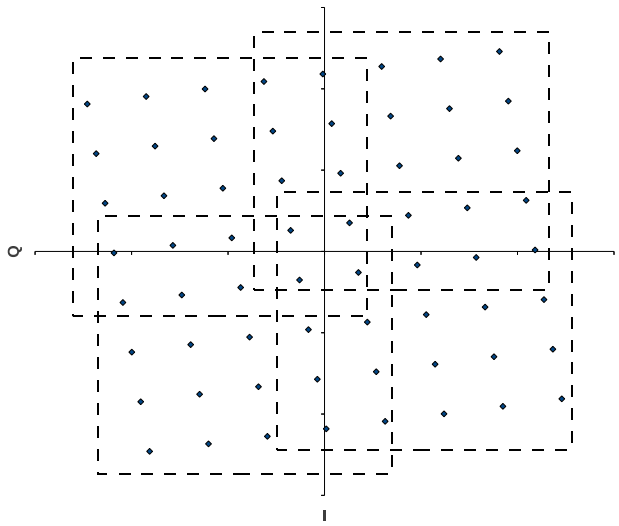

Fig. 1 The division of a 16-QAM rotated constellation diagram into 4 quadrants for use in the demapper.

3.1.3 LDPC decoder

A binary LDPC code [13] with code rate $r = k/n$ is defined by a sparse binary $(n-k)\times n$ parity-check matrix, $\mathbf{H}$. A valid codeword $\mathbf{x}$ of length n bits of an LDPC code satisfies the constraint $\mathbf{H}\mathbf{x}^T = 0$. As such, the parity-check matrix $\mathbf{H}$ describes the dependencies between the k information bits and the $n-k$ parity bits. The code can also be described using bipartite graphs, i.e., with n variable nodes and $n-k$ check nodes. If $\mathbf{H}_{i,j} = 1$, then there is an edge between variable node j and check node i.

LDPC codes are typically decoded using iterative belief propagation (BP) decoders. The procedure for BP decoding is the following. Each variable node v sends a message $L_{v\to c}$ of its belief on the bit value to each of its neighboring check nodes c, i.e. those connected to the variable node with edges. The initial belief corresponds to the received LLR. Then each check node c sends a unique LLR $L_{c\to v}$ to each of its neighboring

variable nodes v, such that the LLR sent to v' satisfies the parity-check constraint of c when disregarding the message $L_{v'\to c}$ that was received from the variable node v'. After receiving the messages from the check nodes, the variable nodes again send messages to the check nodes, where each message is the sum of the received LLR and all incoming messages $L_{c\to v}$ except for the message $L_{c'\to v}$ that came from the check node c' to where this message is being sent. In this step, a hard decision is also made. Each variable node translates the sum of the received LLR and all incoming messages to the most probable bit value and an estimate on the decoded codeword $\hat{\mathbf{x}}$ is obtained. If $\mathbf{H}\hat{\mathbf{x}}^T = 0$, a valid codeword has been found and a decoding success is declared. Otherwise, the iterations continue until either a maximum number of iterations has been performed or a valid codeword has been found.

The LDPC decoder is one of the most computationally complex blocks in a DVB-T2 receiver, especially given the long codeword lengths (n is 16200 or 64800, while k varies with the code rate used) used in the standard. The best iterative BP decoder algorithm is the *sum-product* decoder, which is also, however, quite complex in that it uses costly operations such as hyperbolic tangent functions. The *min-sum* [4, 22] decoder trades some error correction performance for speed by approximating the complex computations of outgoing messages from the check nodes. The resulting computations that are performed in the decoder are the following. Let $C(v)$ denote the set of check nodes which are connected to variable node v. Similarly let $V(c)$ denote the set of variable nodes which are connected to check node c. Furthermore, let $C(v)\backslash c$ represent the exclusion of c from $C(v)$, and $V(c)\backslash v$ represent the exclusion of v from $V(c)$. With this notation, the computations performed in the min-sum decoder are the following:

1. *initialization*: Each variable node v sends the message $L_{v\to c}(x_v) = LLR(v)$.
2. *check node update*: Each check node c sends the message

$$L_{c\to v}(x_v) = \left(\prod_{v'\in V(c)\backslash v} \operatorname{sign}(L_{v'\to c}(x_{v'}))\right) \times \min_{v'\in V(c)\backslash v} |L_{v'\to c}(x_{v'})| \tag{6}$$

   where $\operatorname{sign}(x) = 1$, if $x \geq 0$ and -1 otherwise.
3. *variable node update*: Each variable node v sends the message

$$L_{v\to c}(x_v) = LLR(v) + \sum_{c'\in C(v)\backslash c} L_{c'\to v}(x_v) \tag{7}$$

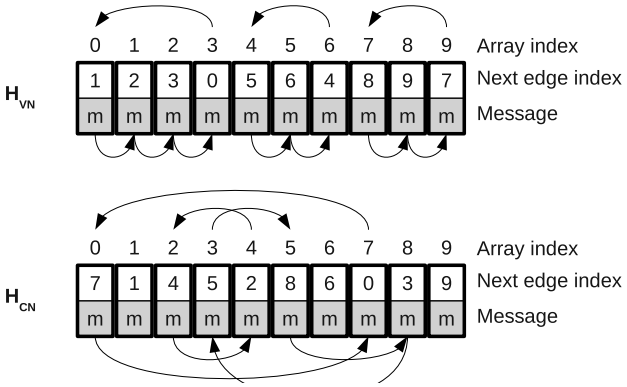

Fig. 2 The arrays $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$ corresponding to example $\mathbf{H}$ matrix.

   and computes

$$L_v(x_v) = LLR(v) + \sum_{c\in C(v)} L_{c\to v}(x_v) \tag{8}$$

4. *Decision*: Quantize $\hat{x}_v$ such that $\hat{x}_v = 1$ if $L_v(x_v) \geq 0$, and $\hat{x}_v = 0$ if $L_v(x_v) < 0$. If $\mathbf{H}\hat{\mathbf{x}}^T = 0$, $\hat{\mathbf{x}}$ is a valid codeword and the decoder outputs $\hat{\mathbf{x}}$. Otherwise, go to step 2.

This algorithm was implemented both for the x86 CPU, and the GPU. The GPU implementation was in quite an early stage at the time of writing this article, and as such the software algorithm used is still subject to optimizations.

The GPU implementation was implemented using two primary kernels for the variable node updates and check node updates, respectively. On the GPU, we used two compact representations, $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$, of the parity check matrix, $\mathbf{H}$. The data structures were inspired by and very similar to those described in [10]. To illustrate these structures, we use the following simple example $\mathbf{H}$ matrix:

$$\mathbf{H} = \begin{pmatrix} 1\,1\,1\,1\,0\,0 \\ 0\,0\,1\,1\,0\,1 \\ 1\,0\,0\,1\,1\,0 \end{pmatrix}$$

$\mathbf{H}_{VN}$ would then be an array of entries consisting of a message (floating point) and a cyclic index to the entry corresponding to the next 1 in the same row of the $\mathbf{H}$ matrix, while entries in $\mathbf{H}_{CN}$ would contain a message and an index to the entry corresponding to the next 1 in the same column. Each entry in $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$ thus represent an edge between a variable node and a check node in the bipartite graph corresponding to $\mathbf{H}$. The $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$ structures corresponding to our example $\mathbf{H}$ matrix are illustrated in figure 2.

The arrows in the figure help visualize where the next edge index value points. It is important that the

entries corresponding to the same edge are positioned at the same index in both $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$, so that messages written by the variable node update kernel can be read by the check node update kernel, and vice versa [10].

Using these structures, the variable node update kernel will follow the index pointers in $\mathbf{H}_{CN}$ in order to update each entry in $\mathbf{H}_{VN}$, while the check node update kernel will follow the index pointers in $\mathbf{H}_{VN}$ in order to update the entries in $\mathbf{H}_{CN}$. One thread will be created on the GPU for each entry in the compact structure (10 threads for the example $\mathbf{H}$ matrix), allowing us to update messages in parallel. In the example, the sixth thread (thread id 5) created by the variable node update will perform the following operation:

$$\mathbf{H}_{VN}(5) = LLR(3) + \mathbf{H}_{CN}(8) + \mathbf{H}_{CN}(3)$$

according to equation 7. $\mathbf{H}_{CN}(8)$ and $\mathbf{H}_{CN}(3)$ are found by following the next edge pointers starting from $\mathbf{H}_{CN}(5)$. The sixth thread of the check node update would perform the following:

$$\begin{aligned}\mathbf{H}_{CN}(5) =& \text{sign}(\mathbf{H}_{VN}(6)) \times \text{sign}(\mathbf{H}_{VN}(4)) \\ & \times \min\{|\mathbf{H}_{VN}(6)|, |\mathbf{H}_{VN}(4)|\}\end{aligned}$$

3.1.4 FFT

The FFT block was implemented both as CPU-only and GPU-accelerated versions. The CPU implementation used the highly optimized FFTW (Fastest Fourier Transform in the West) library [12]. The GPU implementation of the block instead used the NVIDIA CUFFT library, based on the CUDA architecture.

3.2 Hardware setup

Benchmarking was performed on an Ubuntu Linux operating system for x86-64 (64-bit) architectures using the Linux 2.6.35 kernel. The computer in question was equipped with an Intel Core i7 950 quad core CPU running at 3.07 GHz. During benchmarking, multithreading was not used within the measured functionality (and thus only one CPU core was exploited). The Intel Turbo Boost technology was not disabled on the CPU during benchmarking, allowing it to run at a maximum of 3.33 GHz on a single core. 6 GB of DDR3 RAM at 1666 MHz was available in the system. A 480-core NVIDIA GeForce GTX 570 GPU was also part of this hardware setup, and was used for running and benchmarking the CUDA-based blocks.

The measurements were performed by using the *clock_gettime* function (found within the GNU C library) to return the time before and after execution of what was considered to be the core functionality of a functional block, such as a main loop. The impact of additional memory transfers between blocks and similar setup operations were mostly ignored.

While the efficiency of the code has been considered when the blocks were written, the code does however not contain low level optimizations, such as optimized inline assembly code. The possibility for further optimizations means that the benchmarks presented in the following section should be seen mainly as indications of the relative complexity of the involved functional blocks, as well as indications of the feasibility of running the blocks on general purpose CPUs.

In the following section, we present the results of benchmarking the main functionality of the implemented parts of a DVB-T2 system.

4 Benchmark Results

In this section the results from benchmarking the algorithms are presented. As DVB-T2 offers quite many customization possibilities, we fixed most parameters in the configurations used for the benchmarks. We used pilot pattern 1, as defined in the standard [7], as well as a guard interval of 1/4 throughout the benchmark tests. Also, only the 8K FFT mode was considered.

Table 1 shows the measured throughputs of the various blocks in the modulator and demodulator for 16-QAM and 256-QAM configurations using both short and long LDPC codes. The block throughput was measured by timing the core functionality of the block, and dividing the time used for processing one FEC block by the size of the FEC block (16200 bits for short code length, and 64800 bits for long code). Thus, the throughput measure does not give the actual useful bitrate, but rather the bitrate including parity data. To gain an approximate useful bitrate, the throughput figure must be multiplied by the code rate.

It is worth noting that the BCH and LDPC encoder and decoder functionality depends on code rate. The throughputs in table 1 are measured for code rate 1/2. It was found that the BCH encoder's throughput was roughly halved from the lowest code rate, 1/2, to the highest, 5/6. As mentioned, a BCH decoder had not been completely implemented at the time of writing, which is the reason for the missing measurements on the demodulator side.

The LDPC encoder was found to not vary significantly in performance between code rates. The LDPC decoder was found to vary between code rates, and performance was reduced to about 55% of the 1/2-rate performance for some configurations. This also applies for

Table 1 Modulator and demodulator block throughputs (Mbps)

	Modulator				Demodulator			
	Short Code		Long Code		Short Code		Long Code	
	16-QAM	256-QAM	16-QAM	256-QAM	16-QAM	256-QAM	16-QAM	256-QAM
BCH	9.0	9.0	6.9	6.9	n/a	n/a	n/a	n/a
LDPC	100	100	69	69	1.8	1.8	0.9	0.9
Bit Interleaver	55	55	53	55	81	83	78	81
Mapper	85	85	87	87	19.7	3.6	19.7	3.6
Q delay	3240	5400	3600	7200	2700	5400	2817	5400
Cell Interleaver	3240	8100	2400	5400	1157	2612	771	1661
Time Interleaver	130	257	129	265	135	305	140	270
Frame Builder	560	1122	557	1109	1514	2938	1548	3157
Frequency Interleaver	633	1268	629	1266	449	825	472	926
Pilot Insertion	378	758	376	757	538	1049	546	1088
IFFT	122	247	122	247	128	256	127	257

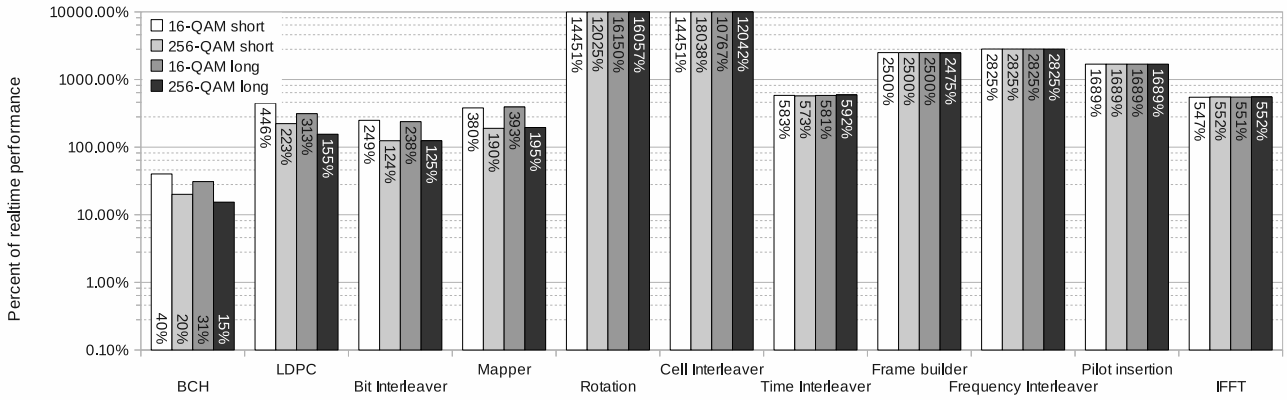

Fig. 3 Modulator block throughput relative to required throughput for realtime performance (100% is realtime)

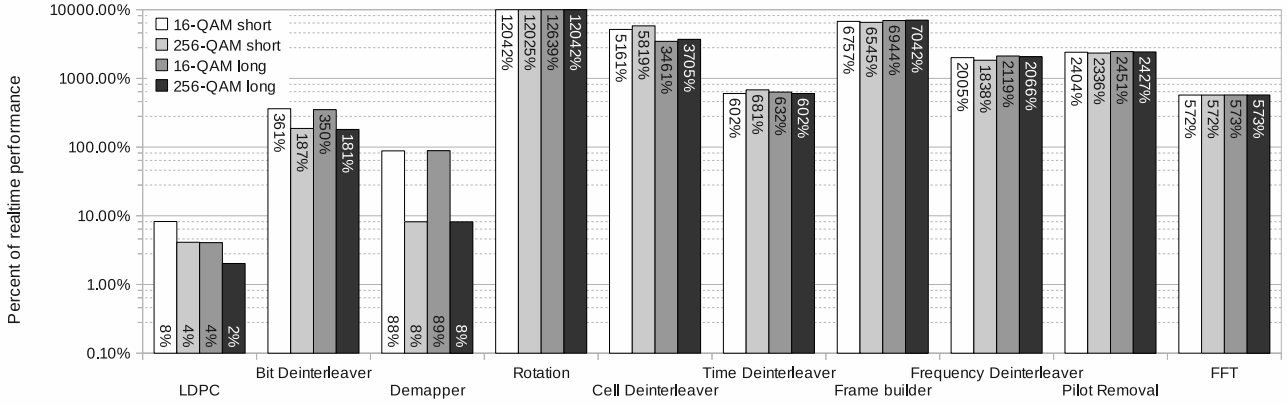

Fig. 4 Demodulator block throughput relative to required throughput for realtime performance.

Table 2 Demodulator GPU block throughputs (Mbps)

	Demodulator			
	Short Code		Long Code	
	16-QAM	256-QAM	16-QAM	256-QAM
LDPC	11.3	11.3	5.3	5.3
Mapper	98	49	113	56
FFT	444	882	444	889

the CUDA-based LDPC decoder. The throughput for the LDPC decoder is expressed as the throughput when running 10 iterations of the iterative BP decoder.

In order to gain a clearer view of which blocks are suitable for realtime operation, it can be mentioned that using the 8K FFT mode with the extended carrier mode enabled, each OFDM symbol contains 6296 data cells, with each cell representing 2, 4, 6, or 8 bits for QPSK, 16-QAM, 64-QAM, and 256-QAM modulation, respectively. The maximum number of OFDM symbols in a frame using the 8K mode, and a guard interval of 1/4 (as in the experiments), is 223. The duration of such a DVB-T2 frame is 250 ms. From the above information, we can calculate that in order to fully process such frames, we require throughputs of roughly 22.5 and 45 Mbps for 16-QAM and 256-QAM, respectively. The throughputs of the modulator and demodulator (as presented in table 1) divided by these "target" throughputs for realtime performance, expressed as percentage values on a logarithmic scale, are presented in figures 3 and 4. In these figures, a percentage value at or above 100% thus means that a block has a throughput at or above the required realtime throughput.

The GPU implementations of the demapper, LDPC decoder, and FFT blocks were also benchmarked, and the throughputs are presented in table 2. The graph comparing these throughputs to our realtime requirements is presented in figure 5.

On a multicore CPU, it would be possible to run the various blocks in a threaded fashion in order to distribute tasks over several cores. Figures 6 and 7 present the measured performance figures in an alternative way, in order to aid in determining how the processing could be distributed on a multicore architecture. These figures present the maximum required throughput divided by the achieved throughput of groups of blocks combined[1]. This essentially gives the required amount of computing capacity relative to one CPU core — or one GPU for the LDPC and demapper blocks in figure 7 — in the test setup. The signal processing blocks have been grouped

[1] The "combined" throughput figure was achieved by adding the measured execution times of all blocks in the group, and calculating the throughput as described earlier for one block.

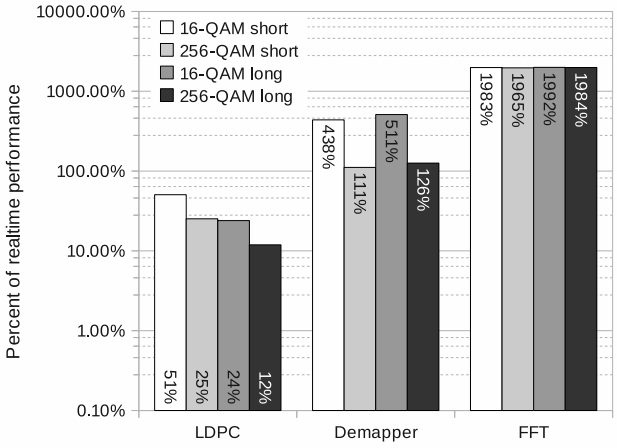

Fig. 5 Demodulator GPU block throughput relative to required throughput for realtime performance.

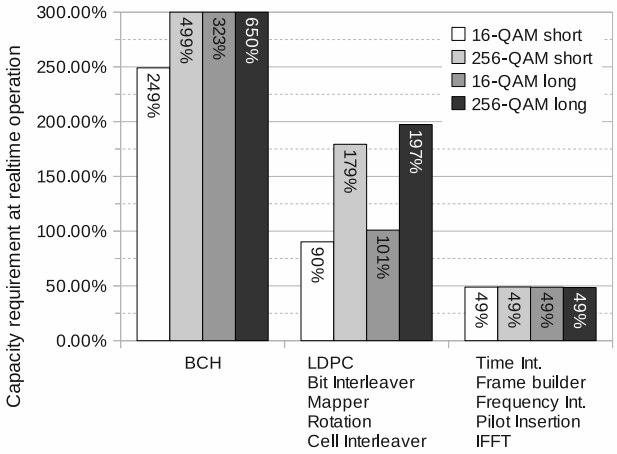

Fig. 6 Required processing capacity of groups of modulator signal processing blocks, presented as a percentage of the capacity of one CPU core on the test setup.

into two main groups in figures 6 and 7. The first group includes the BCH, LDPC, bit interleaver, mapper, rotation, and cell interleaver blocks, while the second group includes the time interleaver, frame builder, frequency interleaver, pilot insertion or removal, and FFT blocks. Furthermore the most complex blocks have been separated from these groupings. The reason for these main groups is that the blocks of the first group of signal processing blocks operate quite independently on units of data corresponding to one FEC block or less, making parallel processing relatively straightforward even inside the individual signal processing blocks. The blocks of the second group operate on larger blocks of data, and thus yield fewer opportunities for parallelism.

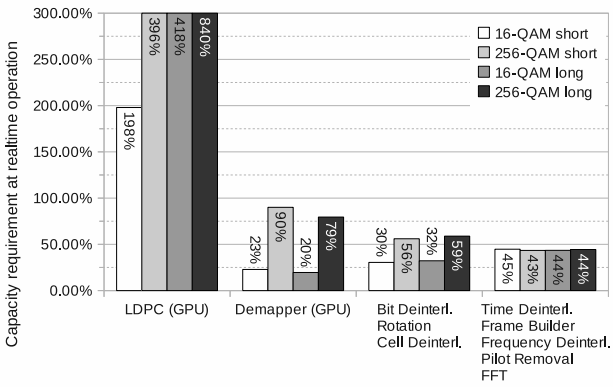

Fig. 7 Required processing capacity of groups of demodulator signal processing blocks, presented as a percentage of the capacity of one CPU core, or the GPU (GPU values for LDPC and Demapper), on the test setup.

The following section is dedicated to the discussion of these results, as well as optimization strategies and alternative implementation platforms for the most computationally costly blocks.

5 Discussion

In this section we discuss the results presented in the last section. It is worth noting that the benchmarked throughputs indicate the throughput that the main functionality of the block is capable of if it is run repeatedly with no other blocks competing for CPU time. In an actual system, these blocks will need to share the computing resources available. While the benchmarks only used one CPU core, opportunities for distributing computation over the multiple cores of modern CPUs are discussed further in section 5.4.

From figure 3, we can see that the BCH block was the slowest block of the modulator with less than 20% of the target bitrate in the slowest case, i.e. 256-QAM with long code length. With 16-QAM and short code length, this block performed at about 40% of the target. While these figures are quite low, further code optimizations might still be able to make the BCH encoder capable of realtime performance. The other blocks exceeded realtime performance with the bit interleaver block being the only block to perform at less than 150% of our realtime goal in certain configurations.

It is on the demodulator side that larger problems emerged. As seen in figure 4, most blocks operated at above realtime performance, except for the constellation demapper and LDPC decoder.

5.1 Demapper

While the demapper was very close to the realtime requirement when 16-QAM modulation was used, the 256-QAM benchmarks showed throughputs less than 4 Mbps or less than 10% of our requirement. Due to the larger number of distance calculations required for 256-QAM, significantly worse performance compared to 16-QAM is indeed also to be expected.

From figure 5, we can see that the CUDA implementation of the demapper performed above 400% of the required rate in 16-QAM configurations, while it was also able to perform slightly above the requirement in the 256-QAM configurations.

The CPU-based demapper performance could perhaps be further improved by using lookup tables to perform parts of the LLR calculations in advance. A proposed methodology for exploiting the large amounts of memory typically present on general purpose computing systems in order to accelerate SDR systems, can be found in [20]. A lookup table based approach could possibly also improve the performance of other blocks in the system. An FPGA implementation of a DVB-T2 demapper is explained in [17].

5.2 LDPC decoder

As mentioned in section 4, the LDPC decoder results are the performance assuming 10 iterations of the iterative BP decoder. The number of iterations necessary might however vary with the quality of the received signal, making the LDPC decoder complexity highly variable. In order to give an indication of how limiting the maximum number of LDPC decoder iterations impacts error correction performance, figure 8 shows simulation results for a 256-QAM configuration at 1/2 code rate. The simulations were performed on signal-to-noise ratio (SNR) levels 0.1 dB apart. For each SNR level, simulations were allowed to run until 20 FEC blocks containing erroneous bits (after BCH decoding[2]) had been encountered, or until 8000 blocks had been simulated without finding 20 erroneous blocks. The average bit error rate (BER) was calculated by comparing the sent and decoded data. A channel model simulating an AWGN (Additive White Gaussian Noise) channel was used. We limited the maximum number of LDPC decoder iterations to 5, 10, 20, 30, 40, and 50 to produce the six curves in figure 8. From the figure, we can

[2] BCH decoding was not actually performed. We do, however, know exactly how many bits a BCH decoder would be able to correct, which is sufficient for simulating the performance of the BCH decoder.

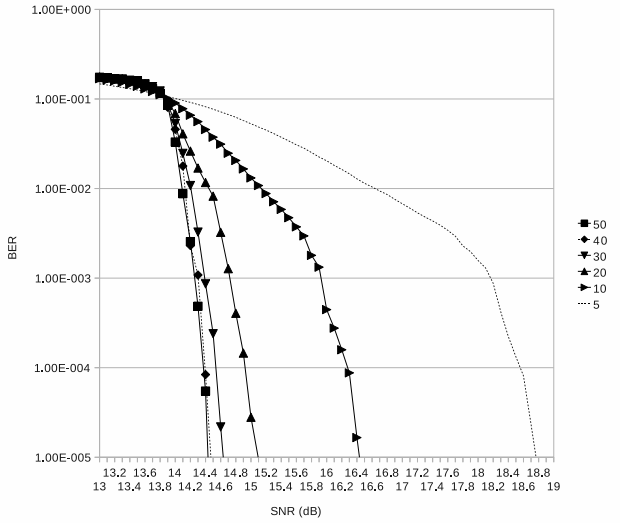

Fig. 8 Simulation results for 256-QAM 1/2-rate setting, when limiting the maximum number of LDPC decoder iterations.

see that letting the LDPC decoder perform only 10 iterations or less demands significantly higher SNRs to achieve a low BER than when allowing 50 iterations, for example.

Neither the CPU nor GPU implementations of the LDPC decoder could perform fast enough to be able to provide good performance in realtime. Using the approach for CUDA-based LDPC decoding described in section 3.1.3, the check node update kernel can access and modify mostly adjacent memory locations, which gives relatively good performance for that kernel. The variable node update kernel does, however, need to access memory locations that are quite random, which leads to slow performance, seemingly due to random global memory accesses. We could verify this by measuring the performance when either the variable node update or check node update kernel was disabled. While the computational instructions involved in the variable node update kernel are not complex, this kernel was found to perform roughly 10 times slower than the check node update kernel, despite the check node update being more computationally complex. As mentioned, the GPU implementation was in an early stage of development and several optimizations, such as decoding several codewords in parallel, as well as using lower precision for messages should still be attempted (as suggested for example in [1]. Further in-depth discussion on GPU acceleration of LDPC decoding can be found in [1, 9–11]. In [11], LDPC decoding was also implemented on the CELL Broadband Engine multicore architecture, which however was deemed unsuitable for codes with large $\mathbf{H}$ matrices, such as those used in DVB-T2. During editing of this article, a description of a real-time capable GPU-based LDPC decoder for DVB-S2 codes (almost identical to DVB-T2 codes) has been published in [8].

FPGA implementations of LDPC decoders for the kind of codes used in DVB-T2 have been demonstrated to be feasible [14, 16]. The FPGA implementation discussed in [14] exploits the periodicity of the LDPC codes used for DVB-S2. This periodicity could perhaps also be exploited in GPU implementations.

The software implementation of DVB-C2 described in [15] simplified the demapper by not calculating LLR values at all, but instead performing a hard decision on the bit values sent to the LDPC decoder. Also, DVB-C2 [6] does not support rotated constellations, making distance calculations less complex. The DVB-C2 standard does however support up to 4096-QAM constellations. The LDPC decoder in [15] was implemented using a simple bit-flipping decoder. This allowed the authors of [15] to reach realtime performance on a general purpose CPU, though at the cost of significantly higher demands on signal quality.

5.3 FFT

As mentioned, we benchmarked our system only using the 8K FFT size (because the other sizes were largely untested), while DVB-T2 supports up to 32K FFTs. Use of the larger FFT sizes will likely be quite detrimental to the throughputs of the FFT blocks in the modulator and demodulator. The FFTW library should however compute discrete Fourier transforms in $O(n \log n)$ time for FFT length n [12], which would seem to not make the use of larger FFT sizes on general purpose processors infeasible.

As seen in figure 5, the GPU CUFFT library is very fast in the 8K mode, and should be able to handle considerably larger FFT sizes than 8K in realtime. A reconfigurable hardware architecture, implemented on an FPGA, for processing multiple FFT sizes present in digital TV standards, is proposed in [3].

5.4 Parallelization

While we only measured the performance of various signal processing blocks on one CPU core (or one GPU), many opportunities for distributing the signal processing over several cores exist. In the modulator case, we can see from figure 6 that the blocks from the time interleaver to the IFFT block only require about half of the capacity of one core together. The part of the chain beginning with the LDPC encoder and ending with the

cell Interleaver requires up to twice the capacity of one core (256-QAM, long code). As the operations of these signal processing blocks on each FEC frame are quite independent, we could most likely distribute the signal processing of this group quite evenly over two cores (perhaps slightly more due to overheads). As already discussed, the BCH block would need to be optimized, or it would require many cores for itself.

From figure 7, we observe that all blocks, excluding the LDPC decoding and constellation demapper blocks, of the modulator chain together would seem to require little more than the full capacity of one CPU core. Again, at least the blocks from the FFT to the time deinterleaver are quite easily parallelizable on a per FEC block basis. While the mapper fits on one GPU, the LDPC decoder would, as already mentioned, have to be optimized to fit on one GPU and even more so to share one GPU with the demapper.

6 Summary

In this article, we have measured the performance of our software implementations of most of the various signal processing blocks of a DVB-T2 modulator and demodulator. The results were presented as throughputs based on timing the core functionality of each block.

The results indicate that the modulator should be possible to realize entirely in software on general purpose computing systems, given some further optimization of the algorithms involved. The demodulator, however, might not be suitable for running exclusively on general purpose processors. Results indicate that the most computationally heavy parts of the demodulator are the FEC decoding and constellation demapper functional blocks. We have discussed alternative architectures, specifically GPUs and FPGAs, and their suitability for executing these tasks. These architectures would retain the reconfigurability that is a major benefit of SDR systems.

As the GNU Radio implementation is not finished, the next steps would be to implement all of the remaining blocks in the GNU Radio environment, as well as optimize both the GPU and CPU implementations of the signal processing blocks further in order to achieve realtime performance. FPGA implementations might also be realized and integrated into the SDR system in the near future.

References

1. Abburi, K.K.: A Scalable LDPC Decoder on GPU. In: VLSI Design (VLSI Design), 2011 24th International Conference on, pp. 183–188 (2011). DOI 10.1109/VLSID.2011.44
2. Blossom, E.: GNU radio: tools for exploring the radio frequency spectrum. Linux Journal (June, 2004)
3. Camarda, F., Prevotet, J.C., Nouvel, F.: Implementation of a reconfigurable Fast Fourier Transform application to digital terrestrial television broadcasting. In: Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on, pp. 353–358 (2009). DOI 10.1109/FPL.2009.5272266
4. Chen, J., Dholakia, A., Eleftheriou, E., Fossorier, M., Hu, X.Y.: Reduced-Complexity Decoding of LDPC Codes. Communications, IEEE Transactions on **53**(8), 1288–1299 (2005). DOI 10.1109/TCOMM.2005.852852
5. Draft ETSI TR 102 831 V0.10.4: Implementation guidelines for a second generation digital terrestrial television broadcasting system (DVB-T2). ETSI Technical Report (2010)
6. ETSI EN 302 769 V1.2.1: Frame structure channel coding and modulation for a second generation digital transmission system for cable systems (DVB-C2). ETSI Technical Report (2011)
7. ETSI EN 302755 v1.1.1: Digital Video Broadcasting (DVB); Frame Structure Channel Coding and Modulation for a Second Generation Digital Terrestrial Television Broadcasting System (DVB-T2). ETSI Technical Report (2009)
8. Falcão, G., Andrade, J., Silva, V., Sousa, L.: GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection. Electronics Letters **47**(9), 542–543 (2011). DOI 10.1049/el.2011.0201
9. Falcão, G., Silva, V., Sousa, L.: How GPUs can outperform ASICs for fast LDPC decoding. In: Proceedings of the 23rd international conference on Supercomputing, ICS '09, pp. 390–399. ACM, New York, NY, USA (2009). DOI 10.1145/1542275.1542330
10. Falcão, G., Sousa, L., Silva, V.: Massive parallel LDPC decoding on GPU. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, PPoPP '08, pp. 83–90. ACM, New York, NY, USA (2008). DOI 10.1145/1345206.1345221
11. Falcao, G., Sousa, L., Silva, V.: Massively LDPC Decoding on Multicore Architectures. Parallel and Distributed Systems, IEEE Transactions on **22**(2), 309–322 (2011). DOI 10.1109/TPDS.2010.66
12. Frigo, M., Johnson, S.: The Design and Implementation of FFTW3. Proceedings of the IEEE **93**(2), 216–231 (2005). DOI 10.1109/JPROC.2004.840301
13. Gallager, R.: Low-Density Parity-Check Codes. Ph.D. thesis, M.I.T. (1963)
14. Gomes, M., Falcao, G., Silva, V., Ferreira, V., Sengo, A., Falcao, M.: Flexible Parallel Architecture for DVB-S2 LDPC Decoders. In: Global Telecommunications Conference, 2007. GLOBECOM '07. IEEE, pp. 3265–3269 (2007). DOI 10.1109/GLOCOM.2007.619
15. Hasse, P., Robert, J.: A Software-Based Real-Time DVB-C2 Receiver. In: Broadband Multimedia Systems and Broadcasting (BMSB), 2011. IEEE International Symposium on (2011)
16. Kocks, C., Viessmann, A., Waadt, A., Spiegel, C., et al.: A DVB-T2 receiver realization based on a software-defined radio concept. In: Communications, Control and Signal Processing (ISCCSP), 2010 4th International Symposium on, pp. 1–4 (2010). DOI 10.1109/ISCCSP.2010.5463488
17. Li, M., Nour, C., Jego, C., Douillard, C.: Design of rotated QAM mapper/demapper for the DVB-T2

standard. In: Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on, pp. 18–23 (2009). DOI 10.1109/SIPS.2009.5336265
18. NVIDIA: CUDA C Programming Guide v.3.2. http://www.nvidia.com (2010)
19. Pellegrini, V., Bacci, G., Luise, M.: Soft-DVB, a Fully Software, GNURadio Based ETSI DVB-T Modulator. 5th Karlsruhe Workshop on Software Radios (2008)
20. Pellegrini, V., Di Dio, M., Rose, L., Luise, M.: On the Computation/Memory Trade-Off in Software Defined Radios. In: GLOBECOM 2010, 2010 IEEE Global Telecommunications Conference, pp. 1–5 (2010). DOI 10.1109/GLOCOM.2010.5683494
21. Vangelista, L., Benvenuto, N., Tomasin, S., Nokes, C., et al.: Key technologies for next-generation terrestrial digital television standard DVB-T2. Communications Magazine, IEEE **47**(10), 146–153 (2009). DOI 10.1109/MCOM.2009.5273822
22. Wiberg, N.: Codes and Decoding on General Graphs. Ph.D. thesis, Linköping University (1996)