Åbo Akademi

OPEN ACCESS

This is an electronic reprint of the original article. This reprint may differ from the original in pagination and typographic detail.

Efficient GPU and CPU-based LDPC decoders for long codewords

Grönroos, Stefan; Nybom, Kristian; Björkqvist, Jerker

Published in:
Analog Integrated Circuits and Signal Processing

DOI:
10.1007/s10470-012-9895-7

Publicerad: 01/01/2012

Link to publication

Please cite the original version:
Grönroos, S., Nybom, K., & Björkqvist, J. (2012). Efficient GPU and CPU-based LDPC decoders for long codewords. *Analog Integrated Circuits and Signal Processing*, *73*(2), 583–595. https://doi.org/10.1007/s10470-012-9895-7

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

This document is downloaded from the Research Information Portal of ÅAU: 20. Apr. 2021

Copyright Notice

The document is provided by the contributing author(s) as a means to ensure timely dissemination of scholarly and technical work on a non-commercial basis. This is the author's version of the work. The final version can be found on the publisher's webpage.

This document is made available only for personal use and must abide to copyrights of the publisher. Permission to make digital or hard copies of part or all of these works for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage. This works may not be reposted without the explicit permission of the copyright holder.

Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the corresponding copyright holders. It is understood that all persons copying this information will adhere to the terms and constraints invoked by each copyright holder.

The final version of this paper can be found at:
http://link.springer.com/article/10.1007/s10470-012-9895-7

© Springer. Pre-prints are provided only for personal use. The final publication is available at link.springer.com

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

Efficient GPU and CPU-based LDPC Decoders for Long Codewords

Stefan Grönroos · Kristian Nybom · Jerker Björkqvist

Received: date / Accepted: date

Abstract The next generation DVB-T2, DVB-S2, and DVB-C2 standards for digital television broadcasting specify the use of Low-Density Parity-Check (LDPC) codes with codeword lengths of up to 64800 bits. The real-time decoding of these codes on general purpose computing hardware is useful for completely software defined receivers, as well as for testing and simulation purposes. Modern graphics processing units (GPUs) are capable of massively parallel computation, and can in some cases, given carefully designed algorithms, outperform general purpose CPUs (central processing units) by an order of magnitude or more. The main problem in decoding LDPC codes on GPU hardware is that LDPC decoding generates irregular memory accesses, which tend to carry heavy performance penalties (in terms of efficiency) on GPUs. Memory accesses can be efficiently parallelized by decoding several codewords in parallel, as well as by using appropriate data structures. In this article we present the algorithms and data structures used to make log-domain decoding of the long LDPC codes specified by the DVB-T2 standard — at the high data rates required for television broadcasting — possible on a modern GPU. Furthermore, we also describe a similar decoder implemented on a general purpose CPU, and show that high performance LDPC decoders are also possible on modern multi-core CPUs.

Keywords DVB-T2 · LDPC · SDR · CUDA · SSE · SIMD

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

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, 26] has been developed to address this need. As compared to DVB-T, this standard offers significantly increased capacity (bitrate). The increased capacity is achieved at the cost of higher complexity components for, among others, forward error correction (FEC).

The DVB-T2 standard makes use of two FEC codes, featuring LDPC (low-density parity-check) codes [11] with exceptionally long codeword lengths of 16200 or 64800 bits as the inner code. As outer code, a BCH (Bose-Chaudhuri-Hocquenghem) code is employed to reduce the error floor caused by LDPC decoding. The second generation digital TV standards for satellite and cable transmissions, DVB-S2 [5] and DVB-C2 [6], respectively, also employ very similar LDPC codes to DVB-T2. Because of the long LDPC codewords, the decoding of these codes is one of the most computationally complex operations in a DVB-T2 receiver [12].

In this work, we propose a method for highly parallel decoding of the long LDPC codes using GPUs (graphics processing units) and general purpose CPUs (central processing units). While a GPU or CPU implementation is likely less energy efficient than implementations based on for example ASICs (application-specific integrated circuits) and FPGAs (field-programmable gate arrays), GPUs and CPUs have other advantages. Even high-end GPUs and CPUs are often quite affordable compared to capable FPGAs, and this hardware can be found in most personal home computers. Although

originally developed for graphics processing, modern GPUs are also highly reconfigurable similarly to general purpose CPUs. These advantages make a GPU or CPU implementation interesting for software defined radio (SDR) systems built using commodity hardware, as well as for testing and simulation purposes.

Algorithms and data structures that allow for reaching the LDPC decoding throughput bitrates required by DVB-T2, DVB-S2, and DVB-C2 when implemented on a modern GPU, are described in the article. While the design decisions are generally applicable to GPU architectures overall, this particular implementation is built on the NVIDIA CUDA (Compute Unified Device Architecture) [23], and tested on an NVIDIA GPU. We also compare the performance of the GPU implementation to a highly efficient multithreaded CPU implementation written for a consumer-grade Intel CPU. Furthermore, we examine the impact of limited numerical precision as well as applied algorithmic simplifications on the error correction performance of the decoder. This is accomplished through comparing the error correction performance of the proposed optimized implementations to more accurate CPU-based LDPC decoders, by simulating transmissions within a DVB-T2 physical layer simulator.

Prior related work can be found in [1, 8–10, 14, 25]. We employ similar data structures to those presented in [9], although with different implementations of the algorithms and levels of parallelism. The implementation described in [8] is quite similar to the implementation presented here in that it describes a realtime GPU-based decoder for DVB-S2 LDPC codes. As DVB-S2 and DVB-T2 codes are mostly identical, we compare performance results against the results obtained in [8]. Differences in results, and their possible causes are discussed in section 5. The implementations described in [1, 14, 25] were written for different types of LDPC codes and very different code lengths from the implementation described here, and are thus difficult to compare directly to the proposed implementation.

An SDR implementation of a DVB-C2 receiver implemented on a normal PC (personal computer) is discussed in [13], where the authors use heavily simplified algorithms for FEC decoding in order to reach realtime performance. In this case, a GPU-based LDPC decoder could most likely provide significantly better error correction performance while also reducing the load on the main CPU.

The article is laid out as follows. In section 2, we describe the basics behind LDPC codes, and the decoding of such codes. In section 3, we describe the architecture used in current NVIDIA GPUs, as well as the experimental setup. Section 4 describes the proposed approaches to LDPC decoding on a GPU and CPU. In section 5, we present performance measurements, both in terms of throughput and error correction capability. Finally, section 6 concludes the article.

2 LDPC CODES

A binary LDPC code [11] 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 $\mathbf{H}\mathbf{x}^T = \mathbf{0}$. 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 Log-Likelihood Ratios (LLR), which are produced by the QAM (Quadrature Amplitude Modulation) constellation demapper [3] in a DVB-T2 receiver. 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 $\mathbf{H}\hat{\mathbf{x}}^T = \mathbf{0}$, 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) specified in the standard. The best iterative BP decoder algorithm is the *sum-product* decoder [17], which is also, however, quite complex in that it uses costly operations such as hyperbolic tangent functions. The *min-sum* [2, 27]

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

$$L_{c\to v}(x_v) = \left(\prod_{v'\in V(c)\backslash v} \text{sign}(L_{v'\to c}(x_{v'})) \right) \times \min_{v'\in V(c)\backslash v} |L_{v'\to c}(x_{v'})| \quad (1)$$

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

$$L_{v\to c}(x_v) = LLR(v) + \sum_{c'\in C(v)\backslash c} L_{c'\to v}(x_v) \quad (2)$$

   and computes

$$L_v(x_v) = LLR(v) + \sum_{c\in C(v)} L_{c\to v}(x_v) \quad (3)$$

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

2.1 DVB-T2 code properties

The DVB-T2 standard [7] specifies LDPC codes with the codeword lengths 16200 bits (short code) and 64800 bits (long code). The code rate $r = k/n$ can be 1/2, 3/5, 2/3, 3/4, 4/5, or 5/6. Table 1 lists n, k, the average row and column degrees, as well as the total number of edges for a subset of these code rates. The average row and column degrees refer to the average number of ones in the rows and columns of $\mathbf{H}$, respectively. Please note that although the short codes are identified as 1/2, 3/4 and 5/6 in table 1 (also identified as such in [7]), the effective code rates for these codes are 4/9, 11/15, and 37/45 respectively.

Table 1 Properties of a subset of the LDPC codes supported in DVB-T2. The columns for average column degree (ACD) and average row degree (ARD) show the average number of ones in the columns and rows of $\mathbf{H}$, respectively. The "edges" column shows the total number of ones in $\mathbf{H}$.

Rate	n	k	ACD	ARD	Edges
1/2		7200	3.0	5.4	48599
3/4	16200	11880	2.9	11.0	47519
5/6		13320	3.0	17.1	49319
1/2		32400	3.5	7.0	226799
3/4	64800	48600	3.5	14.0	226799
5/6		54000	3.7	22.0	237599

3 HARDWARE ARCHITECTURES

In this section we describe the NVIDIA CUDA, and the specific GPU for which the GPU-based implementation was developed. Other relevant components of the system used for benchmarking the decoder implementations are also described, including the Intel CPU which was also the target for the CPU-optimized LDPC decoder.

3.1 CUDA

The NVIDIA CUDA [23] is used on modern NVIDIA GPUs. The architecture is well suited for data-parallel problems, i.e problems where the same operation can be executed on many data elements at once. At the time of writing this article, the latest variation of the CUDA used in GPUs was the Fermi architecture [20], which offers some improvements over earlier CUDA hardware architectures, such as an L1 cache, larger on-chip shared memory, faster context switching etc.

In the CUDA C programming model, we define kernels, which are functions that are run on the GPU 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 processors on a GPU. The GPU used for running the LDPC decoder implementation described in this paper was an NVIDIA GeForce GTX 570 [19, 21], featuring 15 streaming multiprocessors (SMs) containing 32 cores each. The scheduler schedules threads in groups of 32 threads, called thread *warps*. The Fermi hardware architecture features two warp schedulers per SM, meaning the cores of a group of 16 cores on one SM execute the same instruction from the same warp.

Each SM features 64 kB of fast on-chip memory that can be divided into 16 kB of L1 cache and 48 kB of shared memory ("scratchpad" memory) to be shared among all the threads of a thread block, or as 48 kB of

L1 cache and 16 kB of shared memory. There is also a per-SM register file containing 32,768 32-bit registers. All SMs of the GPU share a common large amount of global RAM memory (1280 MB for the GTX 570), to which access is typically quite costly in terms of latency, as opposed to the on-chip shared memories.

The long latencies involved when accessing global GPU memory can limit performance in memory intensive applications. Memory accesses can be optimized by allowing the GPU to *coalesce* the accesses. When the 32 threads of one warp access a continuous portion of memory (with certain alignment limitations), only one memory fetch/store request might be needed in the best case, instead of 32 separate requests if the memory locations accessed by the threads are scattered [23]. In fact, if the L1 cache is activated (can be disabled at compile time by the programmer), all global memory accesses fetch a minimum of 128 bytes (aligned to 128 bytes in global memory) in order to fill an L1 cache line. Memory access latencies can also be effectively hidden if some warps on an SM can run arithmetic operations while other warps are blocked by memory accesses. As the registers as well as shared memories are split between all warps that are scheduled to run on an SM, the number of active warps can be maximized by minimizing the register and shared memory requirements of each thread.

3.2 Measurement setup and CPU

The desktop computer system, of which the GeForce GPU was one component, also contained an Intel Core i7-950 main CPU running at a 3.06 GHz clock frequency. This CPU has 4 physical cores, utilizing Intel Hyper-Threading technology to present 8 logical cores to the system [15]. 6 GB of DDR3 RAM (Double Data Rate 3 random access memory) with a clock frequency of 1666 MHz was also present in the system. The operating system was the Ubuntu Linux distribution for 64-bit architectures.

The CPU supports the SSE (Streaming SIMD Extensions) SIMD (single instruction, multiple data) instruction sets [15] up to version 4.2. These vector instructions, operating on 128-bit registers, allow a single instruction to perform an operation on up to 16 packed 8-bit integer values (or 8 16-bit values, or 4 32-bit values) at once. There are also instructions operating on up to 4 32-bit floating point values. The optimized CPU-based LDPC decoder described in this article exploits these SIMD instructions in combination with multithreading to achieve high decoding speeds. For multithreading, the POSIX (Portable Operating System Interface) thread libraries are utilized.

Another possible approach to building a CPU decoder is to compile the CUDA code directly for the Intel CPU architecture using an appropriate compiler [24]. It is also possible to write the GPU kernels within the OpenCL (Open Computing Language) framework [16] instead of CUDA, as OpenCL compilers are available for both the GPU and CPU. Both of these approaches would still most likely require tuning the implementation separately for the two target architectures in order to achieve high performance, however. As our focus in this article lies on performance rather than portability, we chose to implement the CPU decoder using more well established CPU programming methods.

4 DECODER IMPLEMENTATION

The GPU-based LDPC decoder implementation presented here consists mainly of two different CUDA kernels, where one kernel performs the variable node update (2), and the other performs the check node update (1). These two kernels are run in an alternating fashion for a specified maximum number of iterations. There is also a kernel for initialization of the decoder, and one special variable node update kernel, which is run last, and which includes the hard decision (quantization) step mentioned in section 2.

The architecture of the optimized CPU implementation is very similar to the GPU version. On the CPU, the kernels described above are implemented as C functions which are designed to run as threads on the CPU. Each single thread on the CPU, however, does significantly more work than a single thread running on a CUDA core.

4.1 General decoder architecture

For storage of messages passed between check nodes and variable nodes, we use 8-bit precision. As the initial LLR values were stored in floating point format on the host, we converted the LLRs to 8-bit signed integers by multiplying the floating point value by 2, and keeping the integer part (clamped to the range $[-127, +127]$). This effectively gave us a fixed point representation with 6 bits for the integer part and 1 bits for the decimal part. The best representation in terms of bit allocation is likely dependent on how the LLR values have been calculated and the range of those values. The mentioned bit allocation was found to give good results in our simulations, however this article does not focus on finding an optimal bit allocation for the integer and decimal parts. After this initial conversion

(which is performed on the CPU), the LDPC decoder algorithms use exclusively integer arithmetic.

GPU memory accesses can be fully coalesced if 32 consecutive threads access 32 consecutive 32-bit words in global memory, thus filling one cache line of 128 bytes. In order to gain good parallelism with regard to memory access patterns, we designed the decoder to decode 128 LDPC codewords in parallel. When reading messages from global memory, each of the 32 threads in a warp reads four consecutive messages packed into one 32-bit word. The messages are stored in such a way that the 32 32-bit words read by the threads of a warp are arranged consecutively in memory, and correspond to 128 8-bit messages belonging to 128 different codewords. This arrangement leads to coalescing of memory accesses. Computed messages are written back to global memory in the same fashion, also achieving full coalescence. While the Core i7 CPU only has 64 byte cache lines, we also designed the CPU decoder to decode 128 codewords at once, in order to keep the data structures of the GPU and CPU implementations equal (this decision should not decrease performance).

We use two compact representations, $\mathbf{H}_{VN}$ and $\mathbf{H}_{CN}$, of the parity check matrix $\mathbf{H}$. The data structures were inspired by those described in [9]. To illustrate these structures, we use the following simple example $\mathbf{H}$ matrix:

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

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

We use a separate array structure, $\mathbf{M}$, to store the actual messages passed between the variable and check node update phases. The $\mathbf{M}$ structure contains 128 messages for each one (edge) in $\mathbf{H}$, corresponding to the 128 codewords being processed in parallel. Each entry in $\mathbf{M}$ is one byte in size. The structure is stored in memory so that messages corresponding to the same edge (belonging to different codewords) are arranged consecutively. The entry $\mathbf{M}(i \times 128 + w)$ thus contains the message corresponding to edge i for the w:th codeword.

Furthermore, we use two structures (arrays) $\mathbf{R_f}$ and $\mathbf{C_f}$ to point to the first element of rows and columns,

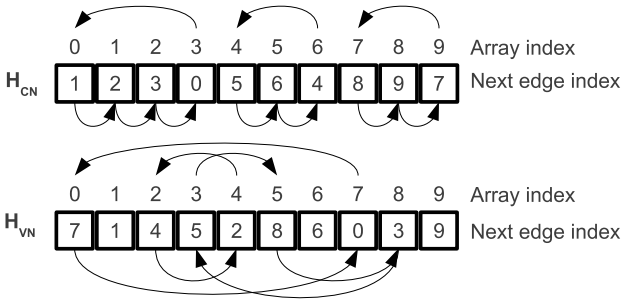


Fig. 1 The arrays $\mathbf{H}_{CN}$ and $\mathbf{H}_{VN}$ corresponding to example $\mathbf{H}$ matrix.

respectively, of the $\mathbf{H}$ matrix. For the example $\mathbf{H}$ matrix, we have $\mathbf{R_f} = (0\,4\,7)$, and $\mathbf{C_f} = (0\,1\,2\,3\,9\,6)$. The structure $\mathbf{LLR}$ contains the received initial beliefs for all codewords, and will have $n \times 128$ elements for an LDPC code of length n. $\mathbf{LLR}(x \times 128 + w)$ contains the initial belief for bit x of codeword w.

4.2 GPU Algorithms

In this subsection follows a more detailed description of the functionality in the GPU kernels. For the variable node update, we let each thread process four consecutive codewords for one column of $\mathbf{H}$, and similarly each thread of the check node update kernel will process one row of $\mathbf{H}$. Thus, 32 consecutive threads will process one column or row for all 128 codewords.

The procedure for the variable node update is roughly as follows, given an LDPC code defined by an $(n-k) \times n$ parity check matrix. We launch $n \times 32$ threads in total.

1. Given global thread id t, we process column $c = \lfloor \frac{t}{32} \rfloor$ of $\mathbf{H}$, and codewords $w = (t \mod 32) \times 4$ to $(t \mod 32) \times 4 + 3$.
2. Read four consecutive LLR values starting from $\mathbf{LLR}(c \times 128 + w)$ into 4-element vector $\mathbf{m}$. We expand these values to 16-bit precision to avoid wrap around problems in later additions.
3. Let $i = \mathbf{C_f}(c)$
4. For all edges in column c:
   4.1. Copy the four consecutive messages (8-bit) starting from $\mathbf{M}(i \times 128 + w)$ into 4-element vector $\mathbf{msg}$. This is achieved by reading one 32-bit word from memory.
   4.2. Add, element-wise, the elements of $\mathbf{msg}$ to the elements of $\mathbf{m}$ and store the results in $\mathbf{m}$.
   4.3. Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.
5. For all edges in column c:
   5.1. Again, copy four messages (8-bit) from $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 3)$ into 4-element vector $\mathbf{msg}$.

5.2. Perform $\mathbf{m} - \mathbf{msg}$ (element-wise subtraction of four elements), clamp the resulting values to the range $[-127, +127]$ (since $\mathbf{m}$ contains 16-bit integers, and $\mathbf{msg}$ contains 8-bit integers) and store the result in $\mathbf{msg}$.
5.3. Copy $\mathbf{msg}$ back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 3)$.
5.4. Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.
6. Variable node update completed.

The check node update launches $(n-k)\times 32$ threads, and the procedure is the following:

1. Given global thread id t, we process row $r = \lfloor \frac{t}{32} \rfloor$ of $\mathbf{H}$, and codewords $w = (t \mod 32) \times 4$ to $(t \mod 32) \times 4 + 3$.
2. Define four 4-element vectors $\mathbf{sign}, \mathbf{min}, \mathbf{nmin}$ and $\mathbf{mi}$. Initialize elements of $\mathbf{sign}$ to 1, and elements of $\mathbf{min}$ and $\mathbf{nmin}$ to 127.
3. Let $i = \mathbf{R_f}(r)$.
4. Let $j = 0$ (iteration counter).
5. For all edges in row r:
   5.1. Copy four consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into 4-element vector $\mathbf{msg}$
   5.2. For all element indices $x \in [0, 3]$, if $|\mathbf{msg}(x)| < \mathbf{min}(x)$, let $\mathbf{min}(x) = |\mathbf{msg}(x)|$ and set $\mathbf{mi}(x) = j$. Otherwise, if $|\mathbf{msg}(x)| < \mathbf{nmin}(x)$, let $\mathbf{nmin}(x) = |\mathbf{msg}(x)|$.
   5.3. Also, for all $x \in [0, 3]$, let $\mathbf{sign}(x)$ be negative if $\mathbf{msg}(x) \times \mathbf{sign}(x)$ is negative, and positive otherwise.
   5.4. Set j equal to $j + 1$.
   5.5. Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.
6. Let $j = 0$.
7. For all edges in row r:
   7.1. Copy four consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into 4-element vector $\mathbf{msg}$.
   7.2. For all $x \in [0, 3]$, if $\mathbf{mi}(x) \neq j$, let $\mathbf{msg}(x) = \text{sign}(\mathbf{sign}(x) \times \mathbf{msg}(x)) \times \mathbf{min}(x)$. Otherwise, if $\mathbf{mi}(x) = j$, let $\mathbf{msg}(x) = \text{sign}(\mathbf{sign}(x) \times \mathbf{msg}(x)) \times \mathbf{nmin}(x)$.
   7.3. Copy $\mathbf{msg}$ back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 3)$.
   7.4. Set j equal to $j + 1$.
   7.5. Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.
8. Check node update completed.

The special variable node update kernel that includes hard decision, adds an additional step to the end of the variable node update kernel. Depending on if $\mathbf{m}(x)$, for $x \in [0, 3]$, is positive or negative, it writes a zero or one to index $c \times 128 + w + x$ of an array structure $\mathbf{B}$ as specified in the last step of the min-sum decoder procedure described in section 2. The $\mathbf{B}$ structure is copied back from the GPU to the host upon completed decoding.

4.3 CPU Algorithms

As mentioned, each single thread in the CPU version performs a larger amount of the total work than in the GPU case. As we use the integer SSE instructions operating on 128-bit (16-byte) registers, we generally operate on 16 8-bit messages belonging to 16 different codewords in each SSE instruction. In the variable node update, each thread computes a fraction (depending on the preferred number of CPU threads) of the columns of $\mathbf{H}$ for all 128 codewords. Likewise, a check node update thread computes a fraction of the rows for all codewords. As in the GPU implementation, the lifetime of one CPU thread is one iteration of either a variable node update or a check node update.

The procedure for the variable node update is as follows, given an LDPC code defined by an $(n - k) \times n$ parity check matrix. We launch T_V threads, where the optimal T_V depends on factors such as CPU core count. Let $t \in [0, T_V - 1]$ denote the current thread. Hexadecimal values are written using the $0x$ prefix.

1. Given thread id t, we process columns $c \in [\frac{t \times n}{T_V}, \frac{(t+1) \times n}{T_V} - 1]$, and for each column, we process 8 groups of 16 codewords, $cg \in [0, 7]$.
2. Let $w = (cg \times 16)$
3. Read sixteen consecutive LLR values starting from $\mathbf{LLR}(c \times 128 + w)$ into 16-element vector $\mathbf{m}$.
4. Let $i = \mathbf{C_f}(c)$
5. For all edges in column c:
   5.1. Copy the sixteen consecutive messages (8-bit) starting from $\mathbf{M}(i \times 128 + w)$ into 16-element vector $\mathbf{msg}$.
   5.2. Add, element-wise, the elements of $\mathbf{msg}$ to the elements of $\mathbf{m}$ and store the results in $\mathbf{m}$ (SSE PADDSB saturating addition instruction).
   5.3. Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.
6. For all edges in column c:
   6.1. Copy the sixteen consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into 16-element vector $\mathbf{msg}$.
   6.2. Perform $\mathbf{m} - \mathbf{msg}$ and store result in $\mathbf{msg}$. The SSE PSUBSB saturating subtraction instruction is used for this.
   6.3. If any element in $\mathbf{msg}$ is equal to -128, set it to -127. Performed by comparing $\mathbf{msg}$ to a vector containing only -128 using the PCMPEQB in-

struction, followed by the PBLENDVB instruction to replace values of -128 with -127 in **msg**.

6.4. Copy **msg** back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 15)$.

6.5. Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.

7. Variable node update completed.

In the CPU implementation there is also a special variable node update function including hard decision. This function calculates the hard decision using SSE instructions by right shifting the values of **m** by 7 bits, so that the sign bit becomes the least significant bit. All bits other than the least significant are set to zero, giving us the hard decision bit values as bytes. Elements equal to -128 are set to -127 in step 6.3 to make the range of positive and negative values equal. Failing to do so was found to result in disastrous error correction performance.

The check node update launches T_C threads, and $t \in [0, T_C - 1]$ denotes the current thread. The procedure is the following:

1. Given thread id t, we process rows $r \in [\frac{t \times (n-k)}{T_C}, \frac{(t+1) \times (n-k)}{T_C} - 1]$, and for each column, we process 8 groups of 16 codewords, $cg \in [0, 7]$.
2. Let $w = (cg \times 16)$
3. Define 16-element vectors **sign**, **min**, **nmin**, and **mi**. Initialize elements of **sign** to 1, and elements of **min** and **nmin** to 127.
4. Let $i = \mathbf{R_f}(r)$.
5. Let $j = 0$ (iteration counter).
6. For all edges in row r:

6.1. Copy sixteen consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into vector **msg**.

6.2. Compute **sign** $\oplus$ **msg**, and store result in **sign**. SSE PXOR instruction for bitwise XOR operation on two 128-bit registers is used.

6.3. Compute element-wise absolute values of **msg**, and store result in **msg**, using the SSE instruction PABSB for absolute value.

6.4. $\forall e \in [0, 15]$, let the value of $\mathbf{mask_1}(e)$ be $0xFF$ if $\mathbf{msg}(e) < \mathbf{min}(e)$, and $0x00$ otherwise. The SSE instruction PCMPGTBr accomplishes this.

6.5. $\forall e \in [0, 15]$, let the value of $\mathbf{mask_2}(e)$ be $0xFF$ if $\mathbf{msg}(e) < \mathbf{nmin}(e)$, and $0x00$ otherwise (PCMPGTBr instruction).

6.6. $\forall e \in [0, 15]$, let $\mathbf{temp}(e) = \mathbf{min}(e)$ if $\mathbf{mask_1}(e)$ equals $0xFF$, and otherwise let $\mathbf{temp}(e) = \mathbf{msg}(e)$. The SSE instruction PBLENDVB is used.

6.7. $\forall e \in [0, 15]$, let $\mathbf{nmin}(e) = \mathbf{temp}(e)$ if $\mathbf{mask_2}(e)$ equals $0xFF$, and otherwise let $\mathbf{nmin}(e) = \mathbf{nmin}(e)$ (PBLENDVB).

6.8. $\forall e \in [0, 15]$, let $\mathbf{min}(e) = \mathbf{msg}(e)$ if $\mathbf{mask_1}(e)$ equals $0xFF$, and otherwise let $\mathbf{min}(e) = \mathbf{min}(e)$ (PBLENDVB).

6.9. $\forall e \in [0, 15]$, let $\mathbf{mi}(e) = j$ if $\mathbf{mask_1}(e)$ equals $0xFF$, and otherwise let $\mathbf{mi}(e) = \mathbf{mi}(e)$ (PBLENDVB).

6.10. Set j equal to $j + 1$.

6.11. Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.

7. Let $j = 0$.
8. $\forall e \in [0, 15]$, let $\mathbf{sign}(e)$ equal -1 ($0xFF$) if $\mathbf{sign}(e) < 0$, and 0 otherwise. This is accomplished by the SSE PCMPGTBr instruction (compare to zero vector).
9. For all edges in row r:

9.1. Copy sixteen consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into vector **msg**.

9.2. $\forall e \in [0, 15]$, let the value of $\mathbf{mask_1}(e)$ be $0xFF$ if $\mathbf{mi}(e) = j$, and $0x00$ otherwise. SSE instruction PCMPEQB accomplishes this.

9.3. $\forall e \in [0, 15]$, let the value of $\mathbf{mask_2}(e)$ be $0xFF$ if $\mathbf{msg}(e) < 0$, and $0x00$ otherwise (PCMPGTBr).

9.4. $\forall e \in [0, 15]$, let $\mathbf{mask_3}(e) = \mathbf{sign}(e) \oplus \mathbf{mask_2}(e)$ (PXOR).

9.5. $\forall e \in [0, 15]$, let $\mathbf{mask_3}(e) = \mathbf{mask_3}(e) \vee 1$ (SSE POR instruction).

9.6. $\forall e \in [0, 15]$, let $\mathbf{temp_1}(e)$ equal $-\mathbf{min}(e)$ if $\mathbf{mask_3}(e) < 0$, and $\mathbf{min}(e)$ otherwise. The SSE instruction PSIGNB is used for this.

9.7. $\forall e \in [0, 15]$, let $\mathbf{temp_2}(e)$ equal $-\mathbf{nmin}(e)$ if $\mathbf{mask_3}(e) < 0$, and $\mathbf{nmin}(e)$ otherwise (PSIGNB).

9.8. $\forall e \in [0, 15]$, let $\mathbf{msg}(e) = \mathbf{temp_2}(e)$ if $\mathbf{mask_1}(e)$ equals $0xFF$, and otherwise let $\mathbf{msg}(e) = \mathbf{temp_1}(e)$ (PBLENDVB).

9.9. Copy **msg** back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 15)$

9.10. Set j equal to $j + 1$.

9.11. Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.

10. Check node update completed.

4.4 Optimization strategies

In this subsection, we discuss various design choices made during implementation to improve decoding speed. The optimizations were verified by benchmarking, as well as profiling of the implementation.

4.4.1 GPU

Notice that, in both main CUDA kernels, we copy the same four elements to **msg** from **M** twice (once in each loop). The second read could have been avoided by storing the elements into fast on-chip shared memory the

first time. Through experiments, however, we noticed that we got significantly improved performance by not reserving the extra storage space in shared memory. This is mostly due to the fact that we can instead have a larger number of active threads at a time on an SM, when each thread requires fewer on-chip resources. A larger number of active threads can effectively "hide" the latency caused by global memory accesses.

Significant performance gains were also achieved by using bit twiddling operations to avoid branches and costly instructions such as multiplications in places where they were not necessary. The fact that this kind of optimizations had a significant impact on performance suggests that this implementation is instruction bound rather than memory access bound despite the many scattered memory accesses performed in the decoder. Through profiling of the two main kernels, we also found that the ratio of instructions issued per byte of memory traffic to or from global memory was significantly higher than the optimum values suggested in optimization guidelines [18], further suggesting that the kernels are indeed instruction bound.

An initial approach at an LDPC decoder more closely resembled the implementation described in [9], in that we used one thread to update one message, instead of having threads update all connected variable nodes or check nodes. This lead to a larger number of quite small and simple kernels. This first implementation was however significantly slower than the currently proposed implementation. One major benefit of the proposed approach is that fewer redundant memory accesses are generated, especially for codes where the average row and/or column degree is high.

As mentioned in section 3.1, the Fermi architecture allows the programmer to choose between 16 kB of shared memory and 48 kB of L1 cache, or vice versa. We used the 48 kB L1 cache setting in the final implementation, as we did not use any shared memory. This clearly improved performance compared to the alternative setting.

4.4.2 CPU

On the CPU we noticed that choosing a significantly higher value for the number of threads (T_V and T_C) per variable or check node update iteration than the number of logical cores in the test setup improved performance significantly. On the test system, we found $T_V = T_C = 32$ to be a good value, although only 8 logical cores were present. It was also found important to process the 8 groups of 16 codewords for a particular row or column of $\mathbf{H}$ before processing another row/column, in order to improve cache utilization. Bit twiddling operations played an even more important role on the CPU than on the GPU, due to the fact that, for example, there is no 8-bit integer multiplication instruction in SSE.

It is worth noting that while we expanded the intermediate result m to a 16-bit integer in the variable node update on the GPU, we kept precision at 8-bit throughout the operation on the CPU. Expanding the intermediate values in an SSE-based implementation would have required many extra operations, sacrificing performance. This solution leads to a somewhat less precise CPU decoder. In section 5.3, we compare the error correction performance of the GPU and CPU implementations.

5 PERFORMANCE

In this section, we present performance figures for both the CUDA-based and SSE SIMD-based LDPC decoders presented in section 4, both in terms of throughput and error correction performance. We show that the GPU implementation achieved throughputs required by the DVB-T2 standard with acceptable error correction performance.

5.1 Throughput measurements

The system described in section 3 was used for benchmarking the two min-sum LDPC decoders. Decoder throughput was measured by timing the decoding procedure for 128 codewords processed in parallel, and dividing the codeword length used (16200 bits for short code length, and 64800 bits for long code) times 128 by the time consumed. 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. We benchmarked the decoder for both the short and long codeword lengths supported by the DVB-T2 standard. Moreover, we measured three different code rates: 1/2, 3/4, and 5/6.

For the GPU implementation, the time measured included copying LLR values to the GPU, running a message initialization kernel, running the variable node and check node update kernels for as many iterations as desired before running the variable node update kernel including hard decision, and finally copying the hard decisions back to host memory. Timing the CPU version included the same steps, except transferring data to and from the GPU, which is not necessary in that case. In these benchmarks we did not check whether we had actually arrived at a valid codeword. This task

Table 2 GPU decoder average throughput in Mbps (Megabits per second), long code ($n = 64800$). Minimum throughput in parentheses.

Rate	20 iterations	30 iter.	50 iter.
1/2	163.4 (160.1)	112.5 (110.9)	69.3 (68.7)
3/4	164.1 (160.6)	112.9 (111.4)	69.5 (68.9)
5/6	157.2 (153.9)	107.9 (106.3)	66.4 (65.9)

Table 3 GPU decoder average throughput in Mbps, short code ($n = 16200$). Minimum throughput in parentheses.

Rate	20 iterations	30 iter.	50 iter.
1/2	186.1 (179.4)	128.6 (125.1)	79.5 (78.2)
3/4	192.4 (185.2)	133.1 (129.6)	82.4 (81.0)
5/6	189.6 (181.8)	131.2 (127.3)	81.2 (79.7)

Table 4 CPU decoder average throughput in Mbps, long code ($n = 64800$). Minimum throughput in parentheses.

Rate	20 iterations	30 iter.	50 iter.
1/2	44.5 (43.4)	30.6 (30.0)	18.7 (18.4)
3/4	42.1 (40.8)	28.8 (28.4)	17.5 (17.4)
5/6	40.1 (38.5)	27.6 (27.4)	17.0 (16.8)

Table 5 CPU decoder average throughput in Mbps, short code ($n = 16200$). Minimum throughput in parentheses.

Rate	20 iterations	30 iter.	50 iter.
1/2	47.4 (44.3)	30.4 (28.5)	18.2 (16.5)
3/4	47.5 (45.7)	30.6 (28.5)	19.1 (17.3)
5/6	45.8 (43.4)	29.7 (27.4)	17.5 (15.5)

was instead handled by the BCH decoder. If desired, we can check the validity of a codeword at a throughput penalty (penalty depending on how often we check for validity). This may for example be done together with hard decision in order to be able to terminate the decoder early upon successful recovery of all 128 codewords. In this case, however, we specify a set number of iterations to run before one final hard decision. Note that the $\mathbf{H}_{CN}$ and $\mathbf{H}_{VN}$ structures only need to be transferred to the GPU at decoder initialization (i.e. when LDPC code parameters change), and that this time is thus not included in the measured time.

The measured throughputs of the GPU implementation are presented in table 2 for long code, and in table 3 for short code configurations. The corresponding throughput figures for the CPU implementation are presented in tables 4 and 5. We decoded 10 batches of 128 codewords and recorded the average time as well as the maximum time for decoding a batch, giving us the average throughput as well as a minimum throughput (shown within parentheses in the tables) for each configuration.

5.2 Results discussion

Annex C of the DVB-T2 standard assumes that received cells can be read from a deinterleaver buffer at 7.6×10^6 OFDM (orthogonal frequency-division multiplexing) cells per second [3, 7]. At the highest modulation mode supported by DVB-T2, 256-QAM, we can represent 8 bits per cell. This means that the LDPC decoder should be able to perform at a bitrate of at least 60.8 Mbps (Megabits per second). As seen from the results, the proposed GPU implementation is able to meet this realtime constraint even while performing 50 iterations.

DVB-S2 [5] and DVB-C2 [4, 6] use the same codeword lengths as DVB-T2, though they specify partly different sets of code rates to suite their application domains. DVB-C2 may require processing up to 7.5×10^6 cells per second, which, coupled with a maximum modulation mode of 4096-QAM, gives us 90 Mbps maximum required throughput. DVB-S2 also may require about 90 Mbps maximum throughput [8]. By interpolation of the values in table 2, we observe that we should be able to meet the throughput requirements of these standards at up to roughly 35 iterations.

As mentioned, the GPU-based LDPC decoder described in [8] decodes DVB-S2 codes, and as such should be comparable to the implementation presented in this paper. The implementation details of this decoder are not explained in-depth in [8]. The authors of [8] do, however, state that the min-sum algorithm is used for decoding, and that 8-bit data representation is used, which both also apply to the implementation discussed in section 4 of this paper. The level of parallelism differs, however, where the implementation in [8] decodes 16 codewords in parallel, while the implementation described in section 4 of this paper decodes 128 codewords in parallel. We do not know in detail how the parallelism is realized in [8], however we believe 128 parallel codewords will allow for improved memory coalescing, due to the fact that Fermi GPUs can read up to 32 successive 32-bit words very efficiently when accessed by the 32 threads in a warp [20]. The higher level of parallelism does introduce some additional latency in a receiver chain, which is however only on the order of fractions of a second considering the high throughputs involved. The authors of [8] do however use lookup tables stored in fast constant memory to calculate message addresses, whereas we fetch the address offsets from global memory.

The GPU used in [8] was an NVIDIA Tesla C2050 [22]. While based on the Fermi architecture, the C2050 differs from the GTX 570 in several ways, such as clock frequencies and memory bus width, making direct per-

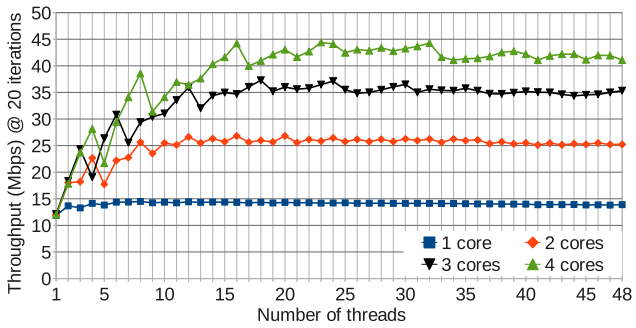


Fig. 2 CPU decoder throughput with 1/2-rate long code at 20 iterations as a function of the number of threads (T_V and T_C). Different curves for 1 to 4 cores available to the decoder.

formance comparison by comparing throughput values difficult. A rough estimate on the speed differences between the two GPUs could be based on the differences in SM clock frequency and the number of SMs. The C2050 has 14 SMs running at 1.15 GHz, while the GTX 570 has 15 SMs running at 1.46 GHz. The authors of [8] reported a throughput of 75.8 Mbps per 30 iterations of the 1/2-rate long DVB-S2 code. From table 2, we can see that we obtained a throughput of 112.5 Mbps for the same code and the same number of iterations. Dividing the throughputs by number of SMs times clock frequency, we get 4.7 and 5.1 kbps per cycle per SM for the implementations in [8] and in this paper, respectively. This comparison is not valid in case an implementation is memory bound rather than arithmetic bound, however, as global memory bandwidth would then be the likely bottleneck.

From tables 4 and 5 we see the throughputs of the CPU decoder at 20, 30, and 50 iterations. We can see that the CPU implementation generally performs at slightly higher than 25% of the throughput of the GPU implementation. As the throughput increases quite linearly with a decreasing maximum number of iterations, we can derive that about 12 iterations should give us the required maximum bitrate of the DVB-T2 standard (60.8 Mbps). Indeed simulations at the slowest setting, 5/6-rate long code, revealed that at 12 iterations, we achieve 63.7 Mbps throughput with the CPU. This low amount of iterations would have a significant negative impact on error correction performance, which is demonstrated in section 5.3.

It should be noted that the throughput of the CPU implementation is the throughput when the CPU is completely dedicated to the task of decoding LDPC codewords. In a single processor system running a software defined receiver, this would not be the case. The CPU capacity would in that case need to be shared among all the signal processing blocks in the receiver chain (in addition to tasks such as video and audio decoding). In this respect, the GPU implementation yields an advantage in addition to higher throughput. If the GPU is assigned the task of LDPC decoding, the CPU is free to perform other tasks.

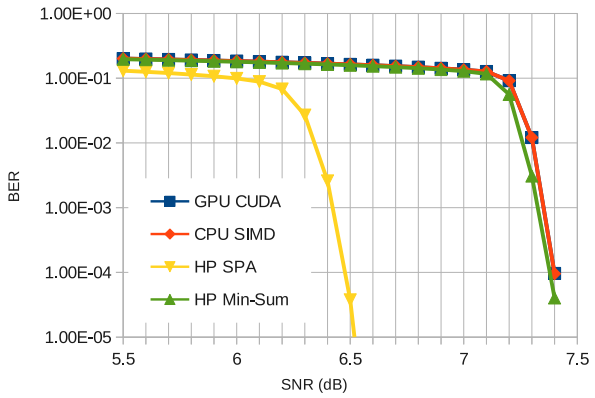
Fig. 2 shows throughput of the CPU implementation (1/2-rate long code, 20 iterations) as a function of varying the amount of threads (T_V and T_C) when different numbers of cores are available to the decoder. It should be noted that a core in Fig. 2 refers to a physical core, which consists of two logical cores, due to the presence of Intel Hyper-Threading technology. The Intel Turbo Boost feature, which allows a core to run at a higher than default clock frequency when other cores are idle, was disabled during this measurement. The speedup factors when utilizing two, three, and four physical cores with the optimal amount of threads are 1.9, 2.6, and 3.1, respectively. Varying the amount of cores used on the GPU is, to the authors' knowledge, not possible, and a similar scalability study was thus not performed on the GPU.

5.3 Error correction performance

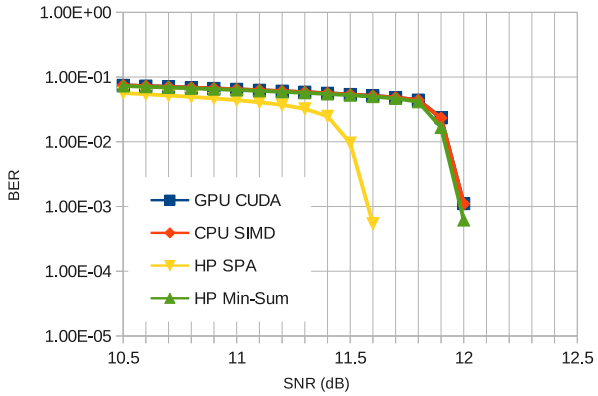
Many dedicated hardware LDPC decoders use a precision of 8 bits or less for messages, and should thus have similar or worse error correction performance compared to the proposed implementations. Within the simulation framework used for testing the decoder, however, we had high-precision implementations of LDPC decoders using both the sum-product algorithm (SPA), as well as the min-sum algorithm. These implementations were written for a standard x86-based CPU, and used 32-bit floating point message representation.

Simulations of DVB-T2 transmissions using both high-precision CPU-based implementations as well as the proposed GPU-based and CPU-based implementations, were performed in order to determine the cost of the lower precision of message representations as well as the use of min-sum over SPA in terms of decoder error correction capability.

Fig. 3 shows simulation results for a 16-QAM configuration at the code rates 1/2 and 5/6 of the long code. The simulations were performed on signal-to-noise ratio (SNR) levels 0.1 dB apart. When simulating using the high-precision CPU implementations, 2000 codewords were simulated for each SNR level. As the proposed implementations were orders of magnitude faster, we simulated 16000 codewords per SNR level for these implementations, in order to be able to detect possible low error floors. The average bit error rate (BER) was calculated by comparing the sent and decoded data. A channel model simulating an AWGN (additive white



(a) Rate 1/2



(b) Rate 5/6

Fig. 3 Simulation results for 16-QAM long code configurations when using the proposed CUDA GPU and SSE SIMD CPU implementations, as well as high precision (HP) CPU implementations of SPA and min-sum algorithms. (Note that BER values of 0 are not included in the graph, which means the curve ends when BER goes down to 0.)

Gaussian noise) channel was used. The maximum number of LDPC decoder iterations allowed was set to 50.

As can be seen in Fig. 3, the proposed lower precision GPU and CPU implementations perform very close (within 0.1 dB) to the high-precision min-sum CPU implementation on the AWGN channel. The simulations clearly indicate that the impact of using the simplified min-sum algorithm as opposed to the superior SPA algorithm is much greater than the choice of message precision. The error correction performance advantage of the SPA algorithm also remains relatively small (please note the fine scale of the x-axes of Fig. 3), however, with slightly less than a 1 dB advantage for 1/2-rate and roughly 0.5 dB for 5/6-rate at a BER level of 10^{-4}.

As mentioned in section 5.2, the CPU implementation could perform only 12 iterations in order to reach the maximum required throughput of DVB-T2, while

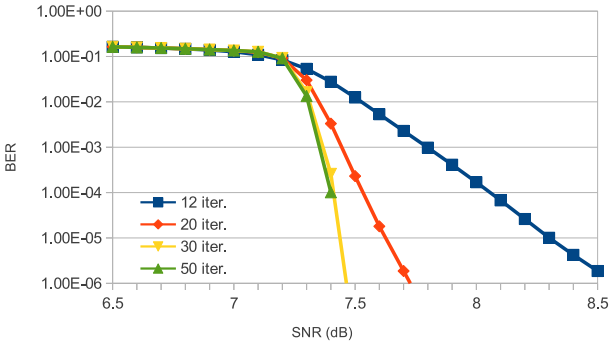


Fig. 4 Simulation results for 16-QAM 1/2-rate long code configuration when varying the maximum number of LDPC decoder iterations. Simulations were performed using the proposed SIMD CPU implementation.

the GPU implementation manages to perform in excess of 50 iterations under the same constraints. In Fig. 4, we demonstrate how varying the amount of maximum iterations performed by the proposed CPU min-sum decoder implementation impacts error correction performance. The figure shows simulation results for a 16-QAM configuration, with 1/2-rate long code over an AWGN channel. All SNR levels were simulated over 2048 codewords. Fig. 4 reveals that 12 iterations of the min-sum decoder does not yield very good error correction performance. The difference between 12 and 50 iterations is roughly 0.7 dB at a BER level of 10^{-4}, which is perhaps not a great amount. At 12 iterations, however, the steepness of the "waterfall" region of the SNR-BER curve is notably worse than at 50 iterations, which is undesirable. Fig. 4 also shows that 30 iterations does not give significantly worse results than 50 iterations.

6 CONCLUSION

In this paper, we have presented two implementations of LDPC decoders optimized for decoding the long codewords specified by the next generation digital television broadcasting standards DVB-T2, DVB-S2, and DVB-C2. The GPU implementation is a highly parallel decoder optimized for a modern GPU architecture. We have shown that we can achieve the throughputs required by these standards at high numbers of iterations, giving good error correction performance. Furthermore, we have shown that our implementation compares well to another similar implementation [8]. We have also shown that a modern multi-core SIMD-enabled CPU is capable of quite high throughputs, though perhaps not

quite enough for the most demanding configurations of the DVB standards.

In the future, we hope to integrate the decoder implementations with other software defined signal processing blocks to build a completely software defined, realtime, receiver chain. In [12], it was shown that besides the LDPC decoder, the QAM constellation demapper — converting received constellation points in the complex plane to LLR values — is one of the most computationally complex blocks in a DVB-T2 receiver chain. As the demapper produces the input to the LDPC decoder (a bit deinterleaver does however separate the two signal processing blocks), a good next step would be to perform both the demapping and LDPC decoding on the GPU, further reducing the main CPU load.

References

1. Abburi, 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. 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
3. DVB BlueBook A133: Implementation guidelines for a second generation digital terrestrial television broadcasting system (DVB-T2). DVB Technical Report (2009)
4. DVB Bluebook A147: DVB-C2 Implementation Guidelines. DVB Technical Specification (2010)
5. ETSI EN 302 307 v1.2.1: Digital Video Broadcasting (DVB); Second generation framing structure, channel coding and modulation systems for Broadcasting, Interactive Services, News Gathering and other broadband satellite applications (DVB-S2). ETSI Technical Report (2009)
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., 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
10. Falcão, 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
11. Gallager, R.: Low-Density Parity-Check Codes. Ph.D. thesis, M.I.T. (1963)
12. Grönroos, S., Nybom, K., Björkqvist, J.: Complexity analysis of software defined DVB-T2 physical layer. Analog Integrated Circuits and Signal Processing **69**, 131–142 (2011). 10.1007/s10470-011-9724-4
13. 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)
14. Hyunwoo, J., Junho, C., Wonyong, S.: Massively parallel implementation of cyclic LDPC codes on a general purpose graphics processing unit. In: Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on, pp. 285–290 (2009). DOI 10.1109/SIPS.2009.5336268
15. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual. Manual, http://www.intel.com (2011)
16. Khronos Group: OpenCL - The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl. Accessed May 2012.
17. MacKay, D.: Good error-correcting codes based on very sparse matrices. Information Theory, IEEE Transactions on **45**(2), 399–431 (1999). DOI 10.1109/18.748992
18. Micikevicius, P.: Analysis-Driven Optimization. Presented at the GPU Technology Conference 2010, San Jose, California, USA (2010)
19. NVIDIA: GeForce GTX 570. http://www.nvidia.com/object/product-geforce-gtx-570-us.html. Accessed June 2011.
20. NVIDIA: NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, http://www.nvidia.com (2009)
21. NVIDIA: NVIDIA GeForce GTX 570 GPU Datasheet. Datasheet, http://www.nvidia.com (2010)
22. NVIDIA: Tesla C2050 and Tesla C2070 Computing Processor Board. Board Specification, http://www.nvidia.com (2010)
23. NVIDIA: CUDA C Programming Guide v.4.0. http://www.nvidia.com (2011)
24. Portland Group, The: PGI CUDA-x86. http://www.pgroup.com/resources/cuda-x86.htm. Accessed May 2012.
25. Shuang, W., Cheng, S., Qiang, W.: A parallel decoding algorithm of LDPC codes using CUDA. In: Signals, Systems and Computers, 2008 42nd Asilomar Conference on, pp. 171–175 (2008). DOI 10.1109/ACSSC.2008.5074385
26. Vangelista, L., Benvenuto, N., Tomasin, S., Nokes, C., Stott, J., Filippi, A., Vlot, M., Mignone, V., Morello, A.: 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
27. Wiberg, N.: Codes and Decoding on General Graphs. Ph.D. thesis, Linköping University (1996)