AN EFFICIENT GPU-BASED LDPC DECODER FOR LONG CODEWORDS

Stefan Grönroos (Turku Centre for Computer Science TUCS/Åbo Akademi University, Turku, Finland; stefan.gronroos@abo.fi); Kristian Nybom (Åbo Akademi University, Turku, Finland; kristian.nybom@abo.fi); Jerker Björkqvist (Åbo Akademi University, Turku, Finland; jerker.bjorkqvist@abo.fi)

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 interesting 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, given carefully designed algorithms, outperform general purpose CPUs 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 paper 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.

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 [1, 2] has been developed to address this need. This standard offers significantly increased capacity (bitrate) when compared to DVB-T. The increased capacity comes at the cost of more complex components for, among others, forward error correction (FEC).

The DVB-T2 standard makes use of two coding schemes, featuring LDPC (Low-Density Parity-Check) codes [3] with exceptionally long codeword lengths of 16200 or 64800 bits as the inner coding scheme. Furthermore, the standard specifies the use of an outer BCH (Bose-Chaudhuri-Hocquenghem) code in order to reduce the error floor caused by LDPC decoding. The second generation digital TV standards for satellite and cable transmissions, DVB-S2 [4] and DVB-C2 [5], respectively, also use very similar LDPC codes to DVB-T2. Due to the long codewords involved, LDPC decoding is one of the most computationally complex operations in a DVB-T2 receiver [6].

In this paper, we propose a method for highly parallel decoding of the long LDPC codes using GPUs (Graphics Processing Units). While a GPU implementation is likely less energy efficient than implementations based on for example ASICs (Application-Specific Integrated Circuits) and FPGAs (Field-Programmable Gate Arrays), the GPU has other advantages. Even high-end GPUs are quite affordable compared to capable FPGAs, and GPUs can be found in most personal home computers. GPUs are also highly reconfigurable similarly to a general purpose CPU (Central Processing Unit). These advantages make a GPU implementation interesting for software defined radio (SDR) systems built using commodity hardware, as well as for testing and simulation purposes.

In the paper, we describe the design of the algorithms and data structures, which, when implemented on a modern GPU, allowed us to reach the LDPC decoding throughput required by DVB-T2, DVB-S2, and DVB-C2. While the design decisions are generally applicable to GPU architectures overall, this particular implementation is built on the NVIDIA CUDA (Compute Unified Device Architecture) architecture [7], and tested on an NVIDIA GPU. 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 done by comparing the results of simulating DVB-T2 transmissions within a DVB-T2 physical layer simulator using both the GPU implementation and more accurate CPU-based implementations.

Prior related work can be found in [8–13]. We used similar data structures to those presented in [8], though with different implementations of the algorithms and levels of parallelism. The implementation described in [13] is quite similar to the implementation presented here in that it describes a realtime GPU-based LDPC 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 [13].

Differences in results, and their possible causes are discussed in section 5. The implementations described in [9–11] were written for different types of LDPC codes and very different code lengths from the implementation described here, and are thus hard to compare directly to our implementation.

An SDR implementation of a DVB-C2 receiver implemented on a normal PC is discussed in [14], where the authors use heavily simplified algorithms for FEC decoding in order to reach realtime performance. In this case, a GPU LDPC decoder could most likely provide significantly better error correction performance while also reducing the load on the main CPU.

The paper 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 CUDA architecture used in current NVIDIA GPUs. Section 4 describes the proposed approach to LDPC decoding on a GPU. In section 5, we present performance measurements, both in terms of throughput and error correction capability. Finally, section 6 concludes the paper.

2. LDPC CODES

A binary LDPC code [3] 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 Log-Likelihood Ratios (LLR), which are produced by the QAM (Quadrature Amplitude Modulation) constellation demapper [15] 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 $\mathbf{H}\hat{\mathbf{x}}^T = 0$, 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 [16], which is also, however, quite complex in that it uses costly operations such as hyperbolic tangent functions. The *min-sum* [17, 18] 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 $L_v(x_v) \geq 0$, 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.

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

2.1. DVB-T2 code properties

The DVB-T2 standard [1] 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 [1]), the effective code rates for these codes are 4/9, 11/15, and 37/45 respectively.

3. THE CUDA ARCHITECTURE

The NVIDIA CUDA[7] architecture 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 paper, the latest variation of the CUDA architecture used in GPUs was the Fermi architecture [19], which offers some improvements over earlier CUDA architectures, such as an L1 cache, larger shared memory, faster context switching and so on.

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 [20, 21], featuring 15 so-called 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 [7]. 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.

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.

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 8, and keeping the integer part (clamped to the range $[-127, +127]$). This effectively gave us a fixed point representation with 4 bits for the integer part and 3 bits for the decimal part. After the initial conversion on the host, the GPU-side algorithms use exclusively integer arithmetic.

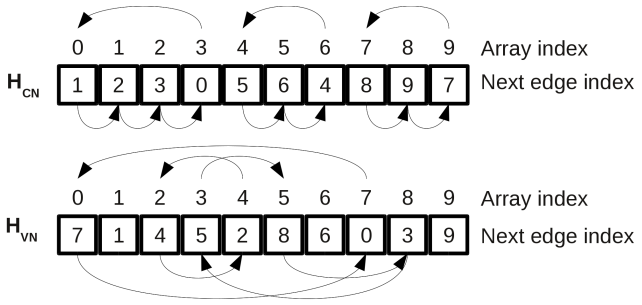


Figure 1: The arrays $\mathbf{H}_{CN}$ and $\mathbf{H}_{VN}$ corresponding to example $\mathbf{H}$ matrix.

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.

On the GPU, 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 [8]. 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 figure 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 the 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, 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.

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.

- 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 $(t \mod 32) \times 4 + 3$.
- 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.
- Let $i = \mathbf{C_f}(c)$
- For all edges in column c:
  - 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.
  - Add, element-wise, the elements of msg to the elements of m and store the results in m.
  - Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.
- For all edges in column c:
  - 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 $\mathbf{msg}$.
  - 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 $\mathbf{msg}$.
  - Copy $\mathbf{msg}$ back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 3)$.
  - Let $i = \mathbf{H_{VN}}(i)$. If $i = \mathbf{C_f}(c)$, we have processed all edges.

- Variable node update completed.

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

- 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 $(t \mod 32) \times 4 + 3$.
- 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.
- Let $i = \mathbf{R_f}(r)$.
- Let $j = 0$ (iteration counter).
- For all edges in row r:
  - Copy four consecutive messages starting from $M(i \times 128 + w)$ into 4-element vector $\mathbf{msg}$
  - 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 $\mathbf{nmin}(x) = |\mathbf{msg}(x)|$.
  - 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.
  - Set j equal to $j + 1$.
  - Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.
- Let $j = 0$.
- For all edges in row r:
  - Copy four consecutive messages starting from $\mathbf{M}(i \times 128 + w)$ into 4-element vector $\mathbf{msg}$.
  - 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 $\mathbf{msg}(x) = \text{sign}(\mathbf{sign}(x) \times \mathbf{msg}(x)) \times \mathbf{nmin}(x)$.
  - Copy $\mathbf{msg}$ back to the memory positions of $\mathbf{M}(i \times 128 + w)$ to $\mathbf{M}(i \times 128 + w + 3)$.
  - Set j equal to $j + 1$.
  - Let $i = \mathbf{H_{CN}}(i)$. If $i = \mathbf{R_f}(r)$, we have processed all edges.
- 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 one or zero value 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.

4.2. 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.

Notice that, in both main kernels, we copy the same four elements to $\mathbf{msg}$ from $\mathbf{M}$ 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 [22], further suggesting that the kernels are indeed instruction bound.

An initial approach at an LDPC decoder more closely resembled the implementation described in [8], 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, 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.

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)

5. PERFORMANCE

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

5.1. Throughput measurements

The GPU used for measuring performance was, as mentioned, a GeForce GTX 570 card [20, 21]. The host computer was equipped with an Intel Core i7 950 quad core CPU running at 3.07 GHz, as well as 6GB of DDR3 RAM. It ran the 64-bit version of Ubuntu Linux 10.10 with Linux kernel version 2.6.35.

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.

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. In these benchmarks we did not check wether we had actually arrived at a valid codeword. This task was instead handled by the BCH decoder. If desired, the GPU could check the validity of a codeword at a performance 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 a 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.

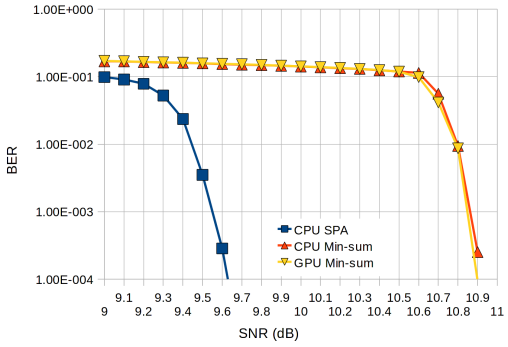
The measured throughputs are presented in table 2 for long code, and in table 3 for short code configurations. We decoded 10 batches of 128 codewords and recorded the average time as well as the maximum time for decoding a batch, giving us the average throughput as well as a minimum throughput (shown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

5.2. Results discussion

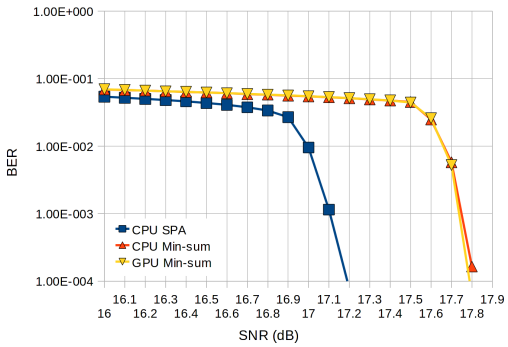
Annex C of the DVB-T2 standard assumes that received cells can be read from a deinterleaver buffer at 7.6×10^6 OFDM (Orthogonal Frequency-Division Multiplexing) cells per second [1, 15]. At the highest modulation mode supported by DVB-T2, 256-QAM, we can represent 8 bits per cell. This means that the LDPC decoder should be able to perform at a bitrate of at least 60.8 Mbps (Megabits per second). As we can see from the results, the proposed implementation is able to meet this realtime constraint even while performing 50 iterations.

DVB-S2 [4] and DVB-C2 [5, 23] 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 [13]. 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 [13] 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 [13]. The paper does, 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 [13] 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 [13], however we believe 128 parallel code-



(a) Rate 1/2



(b) Rate 5/6

Figure 2: Simulation results for 64-QAM long code configurations when using GPU and CPU implementations of LDPC decoder algorithms.

words 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 [19]. 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 [13] 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 [13] was an NVIDIA Tesla C2050 [24]. 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 performance comparison by comparing throughput values difficult. A rough estimate on the speed differences between the two cards 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 [13] 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 [13] 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.

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 implementation. Within the simulation framework used for testing the decoder, however, we had 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. The performance of this CPU version was measured in [6]. As it was not as highly optimized — in not using SIMD (single instruction, multiple data) instructions, and being single-threaded — as the GPU implementation, we do not compare its performance to the GPU implementation in this paper.

We simulated DVB-T2 transmissions using both CPU-based implementations as well as the proposed GPU-based implementation, 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.

Figure 2 shows simulation results for a 64-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. For each SNR level, simulations were allowed to run until 20 FEC blocks containing erroneous bits (after BCH decoding) had been encountered, or until at least 2048 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. The maximum number of LDPC decoder iterations allowed was set to 50.

As we can see in figure 2, the lower precision GPU implementation performs very close (within 0.1dB) to the CPU implementation on the AWGN channel. The impact of using the simplified min-sum algorithm as opposed to the superior SPA algorithm is much greater than the choice of precision. The error correction performance advantage of the SPA algorithm also remains small (please note the fine scale of the x-axes of figure 2), however, with approximately a 1.3 dB advantage for 1/2-rate and less than 1 dB for 5/6-rate at a BER level of 10^{-4}.

6. CONCLUSION

In this paper, we have presented an implementation of an LDPC decoder optimized for decoding the long codewords specified by the next generation digital television broadcasting standards DVB-T2, DVB-S2, and DVB-C2. This 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 with another similar implementation [13].

In the future, we hope to integrate this decoder with other software defined signal processing blocks to build a completely software defined, realtime, receiver chain. In [6], 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, reducing the need to send data back and forth between the host computer and GPU.

7. REFERENCES

[1] ETSI EN 302 755 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.

[2] L. Vangelista, N. Benvenuto, S. Tomasin, C. Nokes, J. Stott, A. Filippi, M. Vlot, V. Mignone, and A. Morello. Key technologies for next-generation terrestrial digital television standard DVB-T2. *Communications Magazine, IEEE*, 47(10):146–153, Oct. 2009.

[3] R. Gallager. *Low-Density Parity-Check Codes*. PhD thesis, M.I.T., 1963.

[4] 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.

[5] 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.

[6] S. Grönroos, K. Nybom, and J. Björkqvist. Complexity analysis of software defined DVB-T2 physical layer. *Analog Integrated Circuits and Signal Processing*, pages 1–12, 2011. DOI: 10.1007/s10470-011-9724-4.

[7] NVIDIA. CUDA C Programming Guide v.4.0. http://www.nvidia.com, 2011.

[8] G. Falcão, L. Sousa, and V. Silva. Massive parallel LDPC decoding on GPU. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 83–90, New York, NY, USA, 2008. ACM.

[9] K.K. Abburi. A Scalable LDPC Decoder on GPU. In *VLSI Design (VLSI Design), 2011 24th International Conference on*, pages 183–188, Jan. 2011.

[10] Ji Hyunwoo, C. Junho, and S. Wonyong. Massively parallel implementation of cyclic LDPC codes on a general purpose graphics processing unit. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 285–290, Oct. 2009.

[11] W. Shuang, S. Cheng, and W. Qiang. A parallel decoding algorithm of LDPC codes using CUDA. In *Signals, Systems and Computers, 2008 42nd Asilomar Conference on*, pages 171–175, Oct. 2008.

[12] G. Falcão, L. Sousa, and V. Silva. Massively LDPC Decoding on Multicore Architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 22(2):309–322, Feb. 2011.

[13] G. Falcão, J. Andrade, V. Silva, and L. Sousa. GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection. *Electronics Letters*, 47(9):542–543, April 2011.

[14] P. Hasse and J. Robert. A Software-Based Real-Time DVB-C2 Receiver. In *Broadband Multimedia Systems and Broadcasting (BMSB), 2011. IEEE International Symposium on*, June 2011.

[15] DVB BlueBook A133. Implementation guidelines for a second generation digital terrestrial television broadcasting system (DVB-T2). DVB Technical Report, 2009.

[16] D.J.C. MacKay. Good error-correcting codes based on very sparse matrices. *Information Theory, IEEE Transactions on*, 45(2):399–431, Mar 1999.

[17] N. Wiberg. *Codes and Decoding on General Graphs*. PhD thesis, Linköping University, 1996.

[18] J. Chen, A. Dholakia, E. Eleftheriou, M.P.C. Fossorier, and X.-Y. Hu. Reduced-Complexity Decoding of LDPC Codes. *Communications, IEEE Transactions on*, 53(8):1288–1299, Aug. 2005.

[19] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. Whitepaper, http://www.nvidia.com, 2009.

[20] NVIDIA. NVIDIA GeForce GTX 570 GPU Datasheet. Datasheet, http://www.nvidia.com, 2010.

[21] NVIDIA. GeForce GTX 570. http://www.nvidia.com/object/product-geforce-gtx-570-us.html (as of June 2011).

[22] P. Micikevicius. Analysis-Driven Optimization. Presented at the GPU Technology Conference 2010, San Jose, California, USA, 2010.

[23] DVB Bluebook A147. DVB-C2 Implementation Guidelines. DVB Technical Specification, 2010.

[24] NVIDIA. Tesla C2050 and Tesla C2070 Computing Processor Board. Board Specification, http://www.nvidia.com, 2010.