HIGH SPEED DECODING OF NON-BINARY IRREGULAR LDPC CODES USING GPUS

Moritz Beermann, Enrique Monzó[⋆]*, Laurent Schmalen*[†]*, Peter Vary*

Institute of Communication Systems and Data Processing (ind), RWTH Aachen University, Germany

[⋆] *now with* Leineweber GmbH, Aachen, Germany

[†] *now with* Bell Laboratories, Alcatel-Lucent, Stuttgart, Germany

{beermann|vary}@ind.rwth-aachen.de

ABSTRACT

Low-Density Parity-Check (LDPC) codes are very powerful channel coding schemes with a broad range of applications. The existence of low complexity (i.e., linear time) iterative message passing decoders with close to optimum error correction performance is one of the main strengths of LDPC codes. It has been shown that the performance of these decoders can be further enhanced if the LDPC codes are extended to higher order Galois fields, yielding so called non-binary LDPC codes. However, this performance gain comes at the cost of rapidly increasing decoding complexity. To deal with this increased complexity, we present an efficient implementation of a signed-log domain FFT decoder for non-binary irregular LDPC codes that exploits the inherent massive parallelization capabilities of message passing decoders. We employ Nvidia's *Compute Unified Device Architecture* (CUDA) to incorporate the available processing power of state-of-the-art *Graphics Processing Units* (GPUs).

Index Terms— non-binary LDPC codes, iterative decoding, GPU implementation

1. INTRODUCTION

Low-Density Parity-Check (LDPC) codes were originally presented in [1] and later rediscovered in [2]. The most prominent decoding algorithm for LDPC codes is the *belief propagation* (BP) algorithm [3], which has linear time complexity. Even though BP decoding is only an approximation of maximum likelihood for practical codes, a performance close to the Shannon limit is observed for long LDPC codes. It has been shown that the performance of short- to medium-length LDPC codes can be improved by employing so called non-binary LDPC codes over higher order Galois fields $\mathbb{F}_q$ with $q = 2^p$ ($p \in \mathbb{N}$) elements [4].

While the decoding complexity of binary LDPC codes of dimension N^{bin} is in the order of $O(N^{\text{bin}})$, this is not the case for non-binary LDPC codes of length $N = N^{\text{bin}}/p$ (this relation of N and N^{bin} serves for a fair comparison, as will be explained later). With a straightforward implementation [4] , the decoding complexity increases to $O(Nq^2) = O(N2^{2p})$. With an intelligent, *signed-log domain Fast Fourier transform* (FFT) based approach [5, 6, 7], this complexity can be reduced to $O(Nq \log_2 q)$, which will still be considerably higher than the complexity of binary belief propagation, if large Galois field dimensions are used.

Especially for applications where decoding complexity is not a main concern (e.g., deep space applications), employing non-binary LDPC codes is a good choice. They exhibit excellent error correction performance while the complexity stays linearithmic (i.e., better than any polynomial time algorithm with exponent greater than 1).

Approaches to further decrease the decoding complexity exist (e.g., [8]), but they suffer from an additional performance penalty compared to BP. However, for comparison, it is often desirable to assess the performance of the BP decoder without any approximations. Therefore, we propose to build a fast reference decoder using the parallel computing capabilities of today's graphic cards.

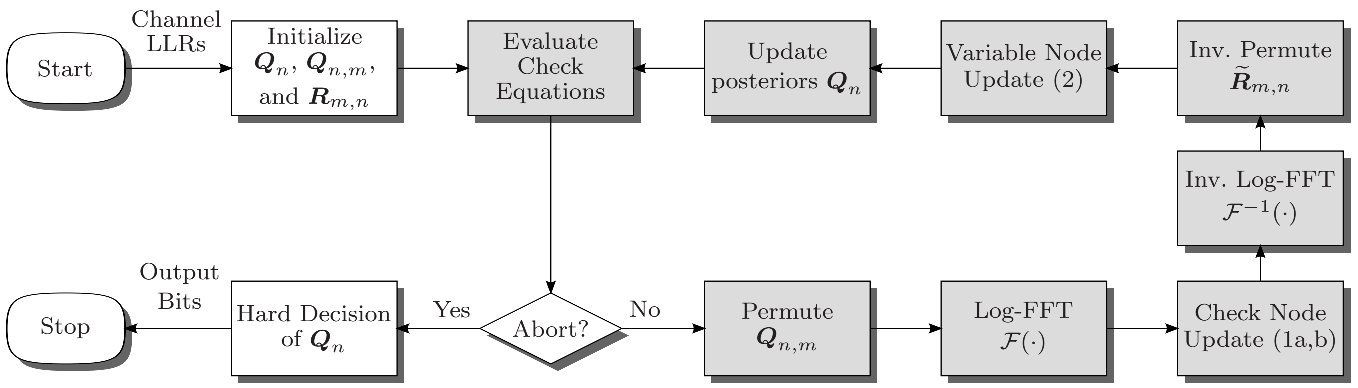
The massive computational power of state-of-the-art graphic cards has led to the utilization of *Graphics Processing Units* (GPUs) for signal processing applications, e.g., [9], [10]. Graphic cards with GPUs by Nvidia Corporation can be elegantly programmed in a language similar to C using Nvidia's *Compute Unified Device Architecture* (CUDA) [11]. The main advantage of GPUs is the large amount of processing cores available on the processor, even in low-end, low-cost products. These can be used for a parallel execution of many problems.

In this paper, we present a massively parallelized implementation of the signed-log domain FFT based decoder that was presented in [7]. The implementation is generic, i.e., it works for arbitrary irregular codes and does not require any special code structure. The CUDA framework is employed to adapt a single-core reference CPU implementation to be executed on one of Nvidia's general purpose GPUs. Decoder throughput results are presented for the CPU as well as the GPU implementations for different Galois field dimensions.

2. NON-BINARY LDPC CODES

2.1. Description

An (N, K) non-binary LDPC code over $\mathbb{F}_q := \{0, 1, \alpha, \ldots \alpha^{q-2}\}$ (α denoting a root of the primitive polynomial defining $\mathbb{F}_q$) with $q = 2^p$ (N coded symbols and K information symbols of $p \in \mathbb{N}$ bit each) is defined as the null space of a sparse parity check matrix $\mathbf{H}$ with entries from $\mathbb{F}_q$ and of dimension $M \times N$ (with $M = N - K$ being the number of parity check symbols). $H_{m,n} \in \mathbb{F}_q$ denotes the entry of $\mathbf{H}$ at row m and column n. The set $\mathcal{N}(m) = \{n : H_{m,n} \neq 0\}$ denotes the symbols that participate in parity check equation m. Similarly, the set $\mathcal{M}(n) = \{m : H_{m,n} \neq 0\}$ contains the check equations in which symbol n participates. Exclusion is denoted by the operator "\",e.g., $\mathcal{M}(n)\backslash\{m\}$ describes the set $\mathcal{M}(n)$ with check equation m excluded. An *irregular* LDPC code has the property that $|\mathcal{M}(n)| \neq$ const. and $|\mathcal{N}(m)| \neq$ const. Similarly to the binary case, non-binary LDPC codes are commonly described by their (weighted) Tanner graph [12], which is a bipartite graph with N variable nodes and M check nodes connected by edges of weight $H_{m,n}$ according to the entries of $\mathbf{H}$. Most decoding algorithms are so called *message passing* algorithms that iteratively pass messages between the variable and check nodes.



Channel LLRs
Start
Initialize $\boldsymbol{Q}_n$, $\boldsymbol{Q}_{n,m}$, and $\boldsymbol{R}_{m,n}$
Evaluate Check Equations
Update posteriors $\boldsymbol{Q}_n$
Variable Node Update (2)
Inv. Permute $\widetilde{\boldsymbol{R}}_{m,n}$
Inv. Log-FFT $\mathcal{F}^{-1}(\cdot)$
Output Bits
Stop
Hard Decision of $\boldsymbol{Q}_n$
Yes
Abort?
No
Permute $\boldsymbol{Q}_{n,m}$
Log-FFT $\mathcal{F}(\cdot)$
Check Node Update (1a,b)

Fig. 1. Flowchart of the non-binary signed-log domain FFT LDPC decoder. Each gray shaded block represents a *CUDA kernel* (see Sec. 3.2).

2.2. Decoding Algorithm: Non-Binary Belief Propagation

A decoding algorithm for non-binary LDPC codes over $\mathbb{F}_q$ has already been described in [4]. This algorithm uses the *Fast Fourier Transform* (FFT) over $\mathbb{F}_q$ for efficiently computing the check node operations (see also [5]). Its drawback is, however, that the implementation needs to be carried out in the probability domain and is, thus, not well suited for an implementation due to eventual numerical instabilities. An implementation in the log domain has been given in [13], which, however, does not make use of the Fourier transform for a low-complexity computation of the check node operation (which considerably reduces the complexity for large field sizes q). In this work, we follow the approach of [6, 7], which uses an FFT and a representation of messages in the *signed-log domain* to carry out the check node update. Other approaches to reduce the complexity aim at computing only the most relevant terms of the update equation [8], but the resulting decoder shows performance losses. In the following, we briefly describe the signed-log domain algorithm according to [7]. A flowchart is given in Fig. 1.

The *binary input additive white Gaussian noise* (BIAWGN) channel output vector of (binary) *log likelihood ratios* (LLRs) is denoted $L(\mathbf{z}^{\text{bin}}) = \left(L(z_1^{\text{bin}}), \dots L(z_{N\cdot p}^{\text{bin}})\right)$ with z_i^{bin} denoting the received, noisy *binary phase shift keying* (BPSK) symbol. To describe the q probabilities of a symbol v being equal to one of the q elements of $\mathbb{F}_q$ in the logarithmic domain, the notion of a q-dimensional LLR vector is introduced

$$\begin{aligned}\boldsymbol{L}(v) &= \left(L^{[0]}(v), L^{[1]}(v), L^{[2]}(v), \dots L^{[q-1]}(v)\right)\\ &= \left(\ln\frac{P(v=0)}{P(v=0)}, \ln\frac{P(v=1)}{P(v=0)}, \dots \ln\frac{P(v=\alpha^{q-2})}{P(v=0)}\right).\end{aligned}$$

Note that even though the first entry is redundant, we still include it for implementation purpose since q is a power of two. The channel output vector $\boldsymbol{L}(\mathbf{z}) = (\boldsymbol{L}(z_1), \dots \boldsymbol{L}(z_N))$ on symbol level is a vector of LLR vectors $\boldsymbol{L}(z_n) = \left(L^{[0]}(z_n), \dots L^{[q-1]}(z_n)\right)$ whose entries are computed from the binary LLRs according to

$$L^{[b]}(z_n) = -\sum_{1\le i\le p:\, b_i=1} L\left(z^{\text{bin}}_{(n-1)\cdot p+i}\right) \quad 0 \le b \le q-1$$

where b_i is the i-th bit of the binary representation of the b-th element of $\mathbb{F}_q$. Representing all messages as LLR vectors and performing all summations and multiplications of LLR vectors element-wise, the signed-log domain FFT decoding algorithm can be described as follows.

The message vectors $\boldsymbol{Q}_{n,m}$ (message sent from variable node n to check node m) and the symbol posteriors $\boldsymbol{Q}_n$ are initialized with the channel output $\boldsymbol{L}(z_n)$. The messages $\boldsymbol{R}_{m,n}$ which are passed from check node m to variable node n are initialized with zeros. The check node update consists of multiple steps. First, the elements of each variable to check message vector are permuted according to the matrix entry $a = H_{m,n} \in \mathbb{F}_q \backslash \{0\}$

$$\widetilde{\boldsymbol{Q}}_{n,m} = \mathcal{P}_a\left(\boldsymbol{Q}_{n,m}\right), \quad \forall\, n, m.$$

The $q-1$ permute functions $\mathcal{P}_a(\cdot)$, with $a \in \mathbb{F}_q\backslash\{0\}$, move the ith element of $\boldsymbol{Q}_{n,m}$ to position $i\cdot a$ in $\widetilde{\boldsymbol{Q}}_{n,m}$. The permuted messages are then transformed into the signed-log domain according to

$$\begin{aligned}&\widetilde{\boldsymbol{\varphi}}_{n,m} = \left(\widetilde{\boldsymbol{\varphi}}^{\text{s}}_{n,m}, \widetilde{\boldsymbol{\varphi}}^{\text{m}}_{n,m}\right)\\ \text{with}\quad &\widetilde{\boldsymbol{\varphi}}^{\text{s}}_{n,m} = \mathbf{1}, \quad \widetilde{\boldsymbol{\varphi}}^{\text{m}}_{n,m} = \widetilde{\boldsymbol{Q}}_{n,m}, \quad \forall\, n, m\end{aligned}$$

where the superscript $(\cdot)^{\text{s}}$ denotes the sign and the superscript $(\cdot)^{\text{m}}$ denotes the magnitude of a signed-log domain value. To transform these values into the Fourier domain, the *Fast Walsh-Hadamard Transform* $\mathcal{F}(\cdot)$ is applied [7]:

$$\widetilde{\boldsymbol{\Phi}}_{n,m} = \mathcal{F}\left(\widetilde{\boldsymbol{\varphi}}_{n,m}\right), \quad \forall\, n, m.$$

Finally, the check node update equations can be written as

$$\widetilde{\boldsymbol{\Theta}}^{\text{s}}_{m,n} = \prod_{n'\in\mathcal{N}(m)\backslash\{n\}} \widetilde{\boldsymbol{\Phi}}^{\text{s}}_{n',m}, \quad \forall\, n, m \tag{1a}$$

$$\text{and}\quad \widetilde{\boldsymbol{\Theta}}^{\text{m}}_{m,n} = \sum_{n'\in\mathcal{N}(m)\backslash\{n\}} \widetilde{\boldsymbol{\Phi}}^{\text{m}}_{n',m}, \quad \forall\, n, m. \tag{1b}$$

Before the resulting message is sent back to a variable node, the inverse Fourier transform

$$\widetilde{\boldsymbol{\theta}}_{m,n} = \mathcal{F}^{-1}\left(\widetilde{\boldsymbol{\Theta}}_{m,n}\right), \quad \forall\, n, m$$

is computed, the magnitude is extracted from the signed-log domain

$$\widetilde{\boldsymbol{R}}_{m,n} = \widetilde{\boldsymbol{\theta}}^{\text{m}}_{m,n}, \quad \forall\, n, m,$$

and the inverse permutation (with $a = H_{m,n} \in \mathbb{F}_q\backslash\{0\}$)

$$\boldsymbol{R}_{m,n} = \mathcal{P}_a^{-1}\left(\widetilde{\boldsymbol{R}}_{m,n}\right), \quad \forall\, n, m$$

is applied to the message vector. The variable node update is executed according to

$$\boldsymbol{Q}_{n,m} = \boldsymbol{L}(z_n) + \left(\sum_{m' \in \mathcal{M}(n) \setminus \{m\}} \boldsymbol{R}_{m',n} \right) - \delta_{n,m}, \quad \forall\, n, m. \quad (2)$$

The normalization constant $\delta_{n,m} = \max_{b \in \mathbb{F}_q} Q_{n,m}^{[b]}$ helps avoiding numerical problems. Furthermore, the posterior LLR vector

$$\boldsymbol{Q}_n = \boldsymbol{L}(z_n) + \sum_{m \in \mathcal{M}(n)} \boldsymbol{R}_{m,n}, \quad \forall\, n$$

is computed for each variable node and a hard decision is performed according to $\hat{x}_n = \arg\max_{b \in \mathbb{F}_q} Q_n^{[b]}$. Based on this hard decision, the parity check equations are evaluated at the end of each iteration. The check and variable node updates are executed until all check equations are satisfied or a maximum iteration counter is reached.

3. GPU IMPLEMENTATION

For an effective implementation, two main points have to be considered: the arrangement of the data in the device memory and the use of the capabilities of the hardware for accelerating the data processing. For an introduction to the CUDA terminology and specifics, we refer the reader to [14].

3.1. Description of the Memory Arrangement

The most important structures are the messages $\boldsymbol{Q}_{n,m}$ and $\boldsymbol{R}_{m,n}$ that are passed between variable nodes and check nodes and the posteriors $\boldsymbol{Q}_n$. All of these are accessed very frequently in most kernels of the main loop. LDPC codes of different sizes and degree profiles as well as different Galois field dimensions should be supported, such that the potentially required amount of memory for these message vectors can become too large for on-chip memory. In addition, the need for quasi-random access to the message structures only leaves the global device memory as suitable solution.

Although the data is stored as a one-dimensional array in the device, it is interpreted as 2D ($\boldsymbol{Q}_n$) or 3D ($\boldsymbol{Q}_{n,m}$ and $\boldsymbol{R}_{m,n}$) structures, depending on the indexing as shown exemplarily for the 3D case in Fig. 2. For the 3D arrays, the first index is pertinent to the node, the second to the node connection and the third to the Galois field element that corresponds to one entry of an LLR vector. The 2D arrays are indexed first by the node and second by the Galois field element. This indexing points to the exact position of one element in the structure, and helps to set the *CUDA grid* and *block* configuration [14] of the kernels.

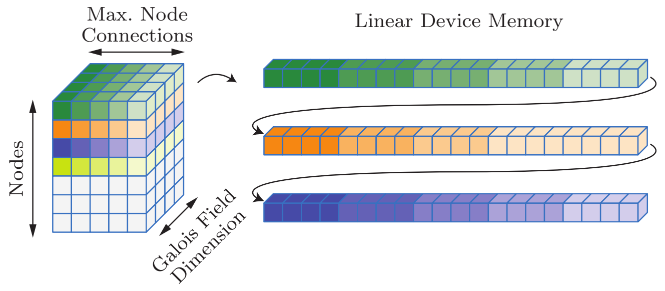


Fig. 2. Three-dimensional structures used for message vectors and corresponding linear data allocation in the global device memory.

By arranging the data in this order, it is possible to access the memory in a coalesced way, such that multiple float values are handled in only one read/write memory access. If each of the q adjacent LLR vector entries are accessed by q adjacent threads, this access will be coalesced, which is a critical requirement for achieving high throughput performance. This arrangement boosts the performance with increasing q.

Note that CUDA requires fixed sizes for data structures, and since irregular LDPC codes do not have a constant number of Tanner graph edges per node, it is obligatory to set the node connection dimension to the maximum value of all node degrees, taking care of not accessing the invalid positions during the processing.

Temporary data that is needed within the kernels is either stored in fast on-chip registers or in the on-chip shared memory if the data is required by multiple *threads* within one block. A more extensive description of the employed data structures can be found in [15].

Another GPU-specific implementation detail is the employment of the GPU's *texture memory* for the Galois field arithmetic. Two-dimensional lookup tables of size $q \times q$ for the addition and multiplication of Galois field elements as well as one-dimensional lookup tables for the inversion (size $q-1$) and conversion between exponential and decimal representation (size q) are initialized in the texture memory and are, thus, available to all kernels.

3.2. Description of the CUDA Kernels

The implementation closely follows the algorithm described in Sec. 2.2 and depicted in Fig. 1. Each of the gray shaded blocks corresponds to one *CUDA kernel* (in CUDA terminology, a kernel denotes an enclosed subroutine).

Initially, the number of iterations is zero. After initialization, the input data is transferred to the CUDA device for an initial check of the parity equations. If they are not fulfilled, the main decoding loop is executed until they are, or until a maximum number of iterations is reached. When the iterative decoding process finishes, a hard decision is conducted to obtain the final binary decoding result. In the following, the most important aspects of the main loop's CUDA kernel implementations are explained. An extensive description including source code for the main routines can be found in [15].

3.2.1. Permute Message Vectors

Before and after the check node update, the elements of the message LLR-vectors have to be permuted according to the Galois field element at the corresponding check matrix position. A shared memory block is used as temporary memory, allowing a parallel implementation of the permute operation. For permuting $\boldsymbol{R}_{m,n}$, the shared memory stores the input data using the permuted indices according to the matrix entry $H_{m,n}$ (using the texture tables for multiplication), and writes the output data directly. For the permutation of $\boldsymbol{Q}_{n,m}$ the shared memory reads the input values directly, but the output is written using the permuted indices as shown in Fig. 3. This way, one *thread* (i.e., one of the many processing units of the GPU) per node and per Galois field element can employed, without accessing the same *shared memory banks*. Thus, no *bank conflicts* are generated, resulting in full speed memory access.

3.2.2. Signed-Log Domain FFT

The implementations of the signed-log-FFT and the inverse signed-log-FFT are identical and follow a butterfly diagram as shown in Fig. 4. As explained in [7], additions in the log domain consist of two

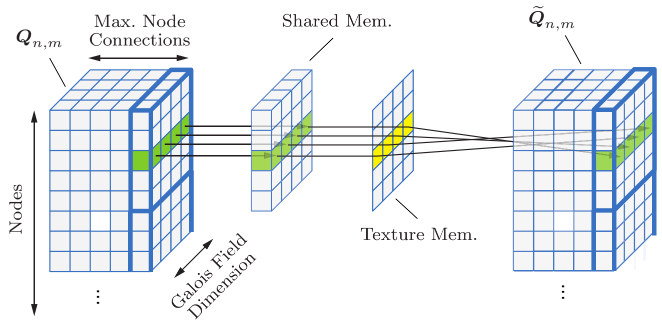


Fig. 3. Permutation of message vectors $\boldsymbol{Q}_{n,m}$.

separate operations for the magnitude and the sign, which is why two shared memory blocks are required for this purpose. Both operations are executed simultaneously in the CUDA kernel implementation. The signed-log-FFT is applied independently to each LLR-vector of size q. Thus, the CUDA block width is fixed to q. For each node, a loop processes all node connections to compute the signed-log-FFT of the q LLR-vector elements. The execution of the butterfly is supervised by a small loop of $\log_2 q$ iterations. Since each continuous thread accesses a different shared memory bank during the butterfly, no bank conflicts are produced.

The exponential and logarithm functions that are needed for the signed-log domain operations [7] are realized by the fast hardware implemented functions which are provided by the GPU. The transfer of data between the global memory and the shared memory is done in a coalesced way, which is ensured by proper indexing.

3.2.3. *Check Node Update*

The implementation is visualized in Fig. 5 and uses one thread per node and Galois field element to compute the outgoing message LLR-vectors for each edge of the corresponding check node. Since the computations are performed in the signed-log domain, magnitude and sign calculus are performed separately. In each thread, a first loop over the node's edges calculates the total sum of the magnitude values and the total product of the sign values. In a second loop, the current magnitude value is subtracted from the total magnitude sum to acquire the extrinsic sum, and the total sign is multiplied by the current sign value for each edge. Shared memory is not required since all memory accesses are executed in a coalesced fashion and, thus, provide high throughput performance.

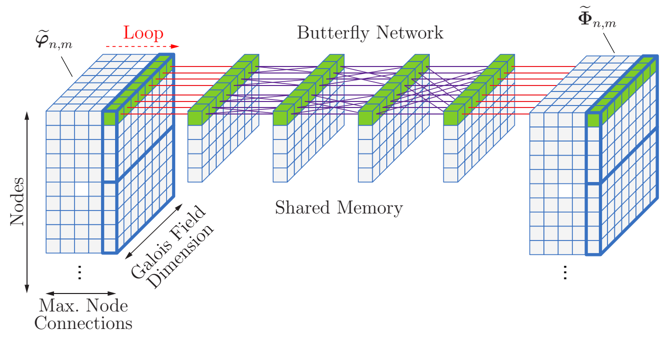


Fig. 4. Butterfly network for FFT implementation.

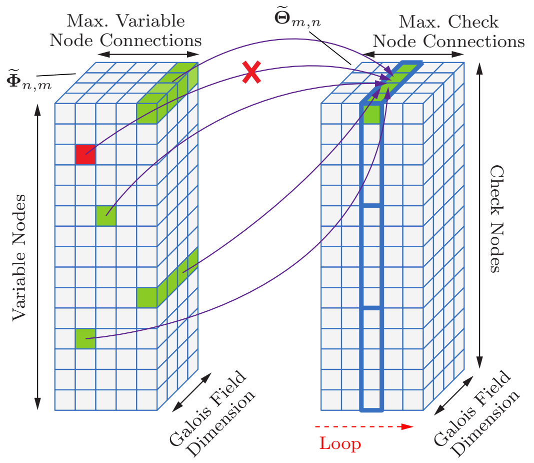


Fig. 5. Check node update of message vectors $\widetilde{\boldsymbol{\Theta}}_{m,n}$.

3.2.4. *Variable Node Update*

The kernel processes one thread per variable node and Galois field dimension (see Fig. 6). An internal loop processes all the node connections within one thread. Each thread first reads the input LLR $\boldsymbol{z}_n$ into a shared memory block and then adds all extrinsic information from $\boldsymbol{R}_{m,n}$ for the current edge. In a second shared memory block, the maximum resulting element $\delta_{n,m}$ of the LLR-vector is found and subtracted from each entry from the first shared memory block before the result is written into $\boldsymbol{Q}_{n,m}$. Since the shared memory operations are inside the loop over the node connections, continuous thread numbers access different shared memory banks in each iteration. This avoids bank conflicts and provides maximum performance for the shared memory.

3.2.5. *Update Posteriors*

The implementation of this kernel follows the one for the variable node update, except that the search and subtraction of the maximum element $\delta_{n,m}$ is omitted and the information of all edges rather than all but one is added to the input LLR to yield the posterior LLR-vectors $\boldsymbol{Q}_n$.

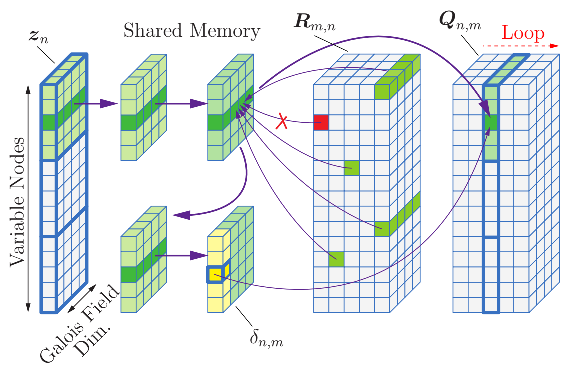


Fig. 6. Variable node update of message vectors $\boldsymbol{Q}_{n,m}$.

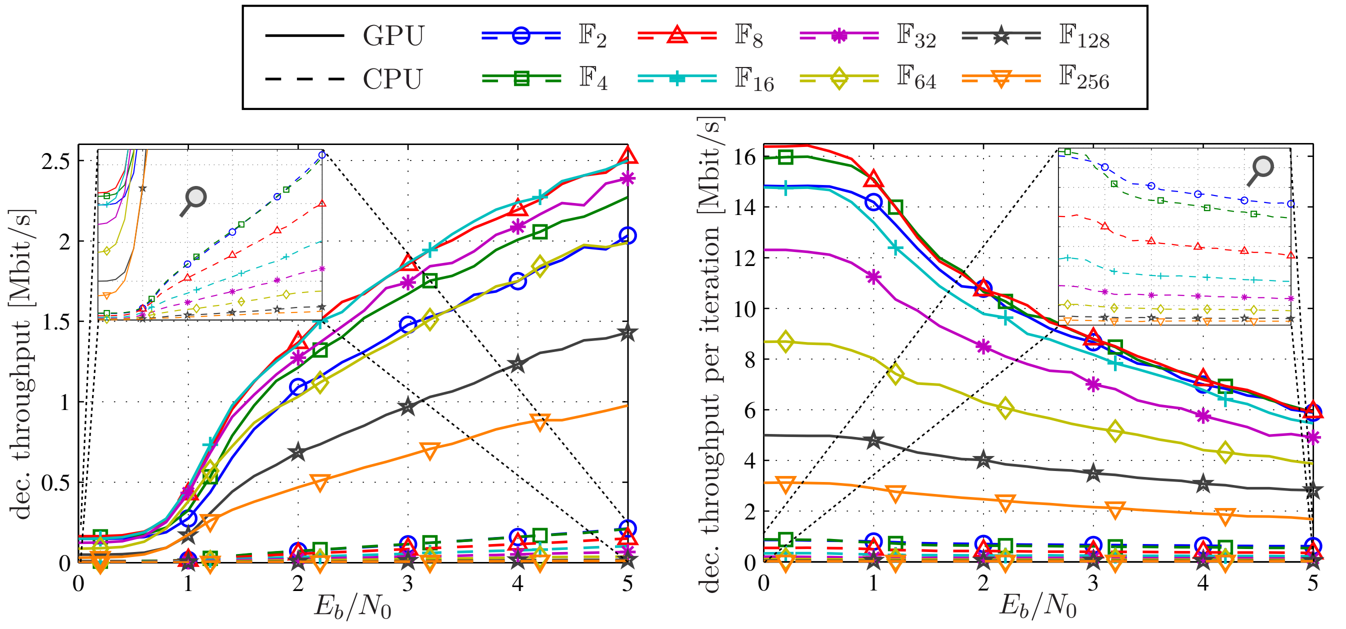


Fig. 7. Average total decoder throughput (left) and average throughput per iteration (right) of coded bits over channel quality in E_b/N_0 at a maximum of 100 iterations. The zoomed region shows the CPU implementation.

3.2.6. *Evaluate Check Equations*

First, one thread per variable node is employed to determine a hard decision of the node's posterior LLR-vector by looping over the Galois field dimension to find the maximum element. After this, one thread per check node loops over its edges and utilizes the texture tables for Galois field addition and multiplication to add up the hard decision values. The result is a binary vector of length M. To decide if decoding can be stopped, a parallelized reduction algorithm is applied to sum up the elements of this vector in $\log_2 M$ steps. The parity equations are fully satisfied if the result of the sum is zero.

4. PERFORMANCE EVALUATION

The CUDA algorithm implementation is executed under the same settings as the single-core CPU reference implementation, and the decoder throughput is measured at several channel SNRs (E_b/N_0 values) and for codes over different Galois field sizes q between 2^1 and 2^8 in a binary input AWGN channel environment with BPSK modulation. It should be noted that the obtained results are related to the hardware architecture used in the simulations and will differ if different CPUs and/or GPUs are used. For the experiments of this paper, the following standard hardware and graphic card were used:

- CPU: Intel Core i7-920, 2.67GHz, 6GB RAM
- GPU: GeForce GTX 580, 512 CUDA cores, 1536 MB RAM, CUDA Compute Capability 2.0.

Table 1 shows the different non-binary rate $\frac{1}{2}$ LDPC codes, used for obtaining the results in this section. Since one Galois field symbol is composed of p bits, the code size N is always chosen according to $N = N^{\text{bin}}/p$ (with $p = \log_2 q$), such that the equivalent number N^{bin} of bits per code word is constant. Concentrated check node degree distributions have been used for all codes. The non-binary check matrices have been constructed with the PEG algorithm [16], by first constructing a binary matrix of the given dimensions and subsequently replacing all ones by non-zero Galois field elements chosen uniformly at random. A maximum number of 100 decoding iterations have been conducted. For all employed codes, the error rate results of the GPU and the CPU implementation are identical within the boundaries given by the different number representations.

The left hand side of Fig. 7 shows the average total throughput of (coded) bits that is processed by one core of the CPU (dashed lines) as well as the GPU implementation (solid lines) over the channel quality in E_b/N_0. The right hand side shows the according average throughput per decoding iteration. In both plots, a zoom in shows the curves for the CPU implementation in more detail.

The very low total throughput at SNR values below 1 dB is due to the high maximum number of 100 decoding iterations that are almost always executed, before the waterfall region of the code's error rate curve is reached. For higher E_b/N_0 values, the total throughput steadily increases, since fewer decoding iterations have to be executed. The curves in the right hand side plot, however, steadily decrease with increasing SNR, which is due to the fact that a roughly constant amount of execution time for initialization and all processing outside the main decoding loop is needed, independent of the number of actually conducted iterations.

Table 1. Dimensions and degree distributions $\sum L_i x^i$ in node perspective of employed non-binary rate $\frac{1}{2}$ LDPC codes. Degree distributions for $2^p \in \{8, 16, 32, 64\}$ were taken from [16]

2^p	$N = \frac{N^{\text{bin}}}{p}$	M	L_2	L_3	L_4	L_5
2	2304	1152	0.549	0.144	0.001	0.306
4	1152	576	0.537	0.302	0.042	0.119
8	768	384	0.643	0.150	0.194	0.013
16	576	288	0.773	0.102	0.115	0.010
32	462	231	0.848	0.143	0.009	0
64	384	192	0.940	0.050	0.010	0
128	330	165	0.852	0	0.148	0
256	288	144	0.990	0.010	0	0

Table 2. Achieved Speedup $\frac{\text{throughput GPU}}{\text{throughput CPU}}$ for different $\mathbb{F}_q$ and E_b/N_0

E_b/N_0	$\mathbb{F}_2$	$\mathbb{F}_4$	$\mathbb{F}_8$	$\mathbb{F}_{16}$	$\mathbb{F}_{32}$	$\mathbb{F}_{64}$	$\mathbb{F}_{128}$	$\mathbb{F}_{256}$
0 dB	17	18	30	44	61	84	112	128
1 dB	18	20	30	47	66	92	124	148
2 dB	15	17	25	39	56	76	109	113
3 dB	13	15	22	34	48	65	100	100

The speedup achieved by the GPU implementation with respect to the CPU implementation is shown in Tab. 2 for several SNR values. While the speedup hugely increases from below 20 to almost 150 with increasing Galois field dimension, it is not affected too much by the varying channel quality. The increasing speedup is mainly due to the fact that groups of q Galois field elements are stored in continuous memory positions, which leads to increased efficiency of the coalesced memory access. This increasing efficiency of the GPU implementation leads to the interesting effect that the average throughput is not monotonically decreasing with the Galois field dimension, even though the decoding complexity increases as $O(N^{\text{bin}}2^p)$ where $p = \log_2 q$ is the number of bits per Galois field symbol.

In Fig. 8 the achieved average total throughput is plotted against the number of bits per symbol for different fixed E_b/N_0 values. Again, a zoomed region is shown for the CPU implementation. As expected, the throughput of the CPU implementation quickly decreases with the number of bits per symbol. The GPU implementation, on the other hand, shows an increasing throughput up to around 4 bits per symbol and then also starts to decrease. This behavior can be explained by the fact that for moderate Galois field dimensions, the increasing computational complexity is overcompensated by the more efficient use of the GPU's capabilities due to the coalesced memory access of q addresses at once. However, for larger Galois field dimensions $q \geq 2^5$ a saturation occurs and the throughput starts to decrease.

5. CONCLUSION

One way of improving the performance of LDPC codes is to use codes over higher order Galois Fields $\mathbb{F}_q$ and according non-binary decoders. While the signed-log domain FFT decoding algorithm provides increasing error correction performance with increasing Galois field dimension, the computational complexity increases significantly at the same time. In this paper, we have presented an efficient implementation of the signed-log domain FFT decoder exploiting the massive parallel compute capabilities of of today's GPUs using Nvidia's CUDA framework. The key strengths of our implementation are the increasing efficiency for increasing Galois field dimensions and the applicability to arbitrary non-binary LDPC codes. While higher speedup can be expected if the codes are constrained to be, e.g., regular and/or quasi-cyclic, the universality of the presented GPU implementation enables accelerated performance estimation without the need for any specific code properties.

6. REFERENCES

[1] R. G. Gallager, *Low-Density Parity-Check Codes*, M.I.T. Press, Cambridge, MA, USA, 1963.

[2] D. J. C. MacKay and R. M. Neal, "Good Codes based on Very Sparse Matrices," in *Cryptography and Coding, 5th IMA Conference*, Springer, Berlin, Germany, 1995, pp. 100–111.

[3] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, San Mateo, CA, USA, 1988.

[4] M. C. Davey and D. J. C. MacKay, "Low-Density Parity Check Codes over GF(q)," *IEEE Comm. Lett.*, vol. 2, no. 6, pp. 165–167, June 1998.

[5] L. Barnault and D. Declercq, "Fast Decoding Algorithm for LDPC over $GF(2^q)$," in *Proc. IEEE Inform. Theory Workshop (ITW)*, Apr. 2003, pp. 70–73.

[6] H. Song and J.R. Cruz, "Reduced-Complexity Decoding of Q-ary LDPC Codes for Magnetic Recording," *IEEE Trans. Magn.*, vol. 39, no. 2, pp. 1081–1087, Mar. 2003.

[7] G. J. Byers and F. Takawira, "Fourier Transform Decoding of Non-Binary LDPC Codes," in *Proc. Southern African Telecommunication Networks and Applications Conference (SATNAC)*, Sept. 2004.

[8] A. Voicila, D. Declercq, F. Verdier, M. Fossorier, and P. Urard, "Low-Complexity Decoding for Non-Binary LDPC Codes in High Order Fields," *IEEE Trans. Commun.*, vol. 58, no. 5, pp. 1365–1375, May 2010.

[9] T. P. Chen and Y.-K. Chen, "Challenges and Opportunities of Obtaining Performance from Multi-Core CPUs and Many-Core GPUs," in *Proc. IEEE ICASSP*, Taipei, Taiwan, Apr. 2009, pp. 609–613.

[10] C.-I. Colombo Nilsen and I. Hafizovic, "Digital Beamforming Using a GPU," in *Proc. IEEE ICASSP*, Taipei, Taiwan, Apr. 2009, pp. 609–613.

[11] T. R. Halfhill, "Parallel Processing with CUDA," *Microprocessor Report*, Jan. 2008.

[12] R. M. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Trans. Inform. Theory*, vol. 27, no. 5, pp. 533–547, Sept. 1981.

[13] H. Wymeersch, H. Steendam, and M. Moeneclaey, "Log-Domain Decoding of LDPC Codes over GF(q)," in *Proc. IEEE International Conference on Communications*, June 2004.

[14] Shane Cook, *CUDA Programming: A Developer's Guide to Parallel Computing With GPUs*, Morgan Kaufmann, 2012.

[15] E. Monzó, "Massive Parallel Decoding of Low-Density Parity-Check Codes Using Graphic Cards," M.S. thesis, RWTH Aachen University & Universidad Politecnica de Valencia, 2010.

[16] X. Y. Hu, E. Eleftheriou, and D. M. Arnold, "Regular and Irregular Progressive Edge-Growth Tanner Graphs," *IEEE Trans. Inform. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.

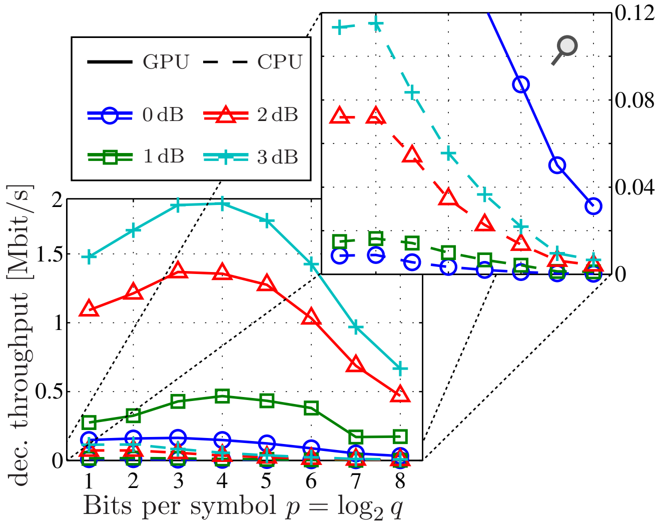


Fig. 8. Throughput over Galois field dimension for $E_b/N_0 \in \{0\,\text{dB}, 1\,\text{dB}, 2\,\text{dB}, 3\,\text{dB}\}$. Zoom shows CPU implementation.