UNIVERSIDAD POLITECNICA DE VALENCIA

ESCUELA POLITECNICA SUPERIOR DE GANDIA

Master en Ingeniería de Sistemas Electrónicos

"Massive Parallel Decoding of Low-Density Parity-Check Codes Using Graphic Cards."

TESIS DE MASTER

Autor:
Enrique Monzó Solves

Director/es:
Dipl. Ing. Laurent Schmalen
Dra. Asunción Pérez Pascual

GANDIA, 2010

Contents

Chapter 1

Introduction

Low-density parity-check (LDPC) codes, first introduced by R. G. Gallager in 1960 [16], are a class of linear block codes characterized by their parity-check matrix, which contains a low density of non-zero elements. The main advantage of LDPC codes is that they provide a performance which is very close to the capacity for many channels. They are suited for implementations that make heavy use of parallelism. However, due to the computational effort in the implementation of these codes, they were mostly ignored until the 90's.

A way to represent LDPC codes is via a Tanner graph representation, which is an effective graphical representation for LDPC codes, because it efficiently helps to describe the decoding algorithm. Tanner graphs are bipartite graphs with nodes that are separated into two different sets and edges connecting the nodes of the two different sets. Each edge represents a non-zero value in the parity-check matrix. The two types of nodes in a Tanner graph are called variable nodes and check nodes.

Iterative suboptimal decoding algorithms are used to decode LDPC codes. Those algorithms perform local calculations and pass those local results via messages. This step is usually repeated several times until convergence is observed.

The most common algorithm for decoding binary LDPC codes is the belief propagation algorithm. For the case of non-binary LDPC codes, the log-domain Fourier transform belief propagation algorithm achieves a good decoding performance. Both algorithms are message passing algorithms.

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA Corporation, being present in the current NVIDIA graphics cards. It is accessible to software developers through variants of industry standard programming languages. Thus, it is possible to program CUDA devices using C/C++ with NVIDIA extensions and certain restrictions.

Each chip with CUDA technology is based on a multiprocessor with many cores and hundreds of ALU's, several thousand registers and some shared memory. Besides, a graphics card contains global memory, which can be accessed by all multiprocessors, local memory in each multiprocessor, and special memory for constants. The several multiprocessor cores in a GPU are SIMD (single instruction, multiple data) cores. This programming style is common for graphics algorithms and many scientific tasks, but it requires specific programming.

Motivated by the parallel computation power of the CUDA architecture, the CUDA implementation of an existing C++ software based LDPC decoding system is studied in this thesis. The main objective is the efficient CUDA implementation of the belief propagation and the log-domain Fourier transform algorithms for the binary and non-binary LDPC decoding processes respectively. A considerable GPU/CPU speedup is expected with the GPU implementation of these algorithms.

Chapter 2 introduces the necessary theoretical background for understanding LDPC codes and the CUDA architecture. The LDPC codes are presented together with their main characteristic, the parity-check matrix, and their graphical representation, the Tanner graph, which is the base for performing the iterative LDPC decoding. The belief propagation and the log-domain Fourier transform algorithms are explained step by step.

Afterwords, the CUDA architecture is introduced as a technology present in the modern graphic cards from NVIDIA. The parallel processing capabilities of these devices are introduced, and a review of the hardware resources and constraints is performed. Some performance considerations are introduced for understanding how to achieve better performance, taking advantage of the device architecture.

The CUDA implementation of the LDPC decoders is detailed in Chapter 3. The belief propagation and the log-domain Fourier transform algorithms are parallelized and assembled in kernels. Each kernel uses the hardware resources in a different way, in order to achieve the best performance.

Chapter 4 shows the results obtained by the CUDA LDPC decoder implementations compared with the respective C++ reference code. The GPU/CPU speedup is measured for different LDCP codes with different code sizes, number of iterations, etc. The results indicate how the massive parallel process capacity of the GPU's can be used for accelerating the decoding algorithms in a very impressive way.

Chapter 2

Theoretical and Technological Background

2.1 Low-Density Parity-Check Codes

Low-density parity-check (LDPC) codes were introduced by Robert G. Gallager in 1963 [16]. These codes were forgotten for a few decades because of the limited ability of computation. LDPC codes were recovered in 1995 by David J.C. MacKay and Radford M. Neal [3]. They realized that these codes offer remarkable performance to allow data transmission rates close to the Shannon limit.

LDPC codes are long linear block codes defined by a generator matrix $\underline{G}$, or a parity check matrix $\underline{H}$ with a low density of non-zero elements. If $\underline{H}$ has a constant number ω_c of non-zero elements in the colums and a constant number ω_r of non-zero elements in the rows, then the code is called *Regular* LDPC code. Otherwise, it is called *irregular.*

For a block length m, it is said that $\underline{H}$ defines an $(m,\ \omega_c,\ \omega_r)$ LDPC code. The parity check matrix is said to be *sparse* if less than half of the elements are non-zero. The linear block code encodes n information bits into m coded bits, with a rate $r = n/m < 1$. Using row vector notation, the coded vector $\underline{Y}$ is obtained from the information vector $\underline{X}$ by the vector multiplication $\underline{Y} = \underline{X} \cdot \underline{G}$. $\underline{G}$ is a matrix with dimension $n \times m$.

$$\underline{Y} = (y_1, y_2, \dots y_m)$$
$$\underline{X} = (x_1, x_2, \dots x_n)$$
$$\underline{Y} = \underline{X} \cdot \underline{G}$$

Each row of $\underline{H}$ provides a parity check equation that any code vector $\underline{Y}$ must satisfy.

$$\underline{Y} \cdot \underline{H}^T = 0$$

Since $\underline{H}$ can reach huge dimensions, the direct decoding process can have a high complexity. Thus, LDPC codes can be decoded iteratively using *message passing algorithms* over a *Tanner graph*. These codes been used in recent digital communications systems standards, such as DVB-S2, DVB-T2, WiMAX (IEEE 802.16), Wireles LAN (IEEE 802.11n).

2.1.1 Tanner Graph

A binary linear code can be represented by a Tanner graph using the parity check matrix $\underline{H}$. A Tanner graph is a bipartite graph with variable nodes on the left, corresponding to the different code symbols, and check nodes on the right, one for each check equation, or row in the parity check matrix. Each variable node ($\underline{H}$ column) is connected in the Tanner graph to the check nodes corresponding to non-zero values in the parity check matrix.

For example, the Tanner graph of the parity check matrix for the (7,4) Hamming code is depicted in Figure 2.1.

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

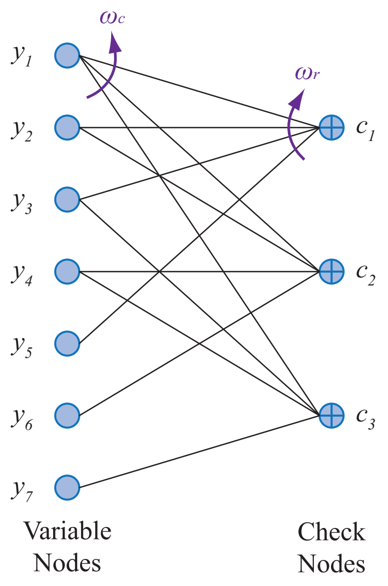

Figure 2.1: Tanner graph for the (7,4) Hamming code.

In Figure 2.1, the first node c_1 corresponds to the first row of the parity check matrix $y_1 + y_2 + y_3 + y_5 = 0$, according to the connections of the variable nodes y_1, y_2, y_3 and y_5 to the check node c_1.

2.1.2 Decoding Process on Binary LDPC Codes: Belief Propagation

Belief propagation is an extension of the message passing algorithm, where the information in the messages being passed is expressed in terms of logarithmic likelihood values (L-values) [19].

In the variable nodes, the incoming and the updated bits are represented as L_i and $\tilde{L}_{ij}$ respectively, and in the check nodes the estimated bits are represented as $\hat{L}_{ij}$.

The channel messages (L_i) are received in the variable nodes and the check equations are applied to them. If the hard decision of these input L-values satisfies the parity check equations, the decoding process is complete since the received bits are a valid codeword. If the check equations are not satisfied, then the iterative decoding process is initialized until the updated word is a valid word or the number of iterations reaches the maximum value. The description of the iterating process is as follows.

Variable Node Update: On each outgoing edge from the variable nodes, an updated L-value $\tilde{L}_{i,j}$ is sent out with a value corresponding to the sum of the corresponding node input L-value plus the extrinsic sum of the values coming in on the other edges (Figure 2.2).

$$\hat{L}_{i,j_1} = L_i + \sum_{j \neq j_1} \tilde{L}_{i,j}$$

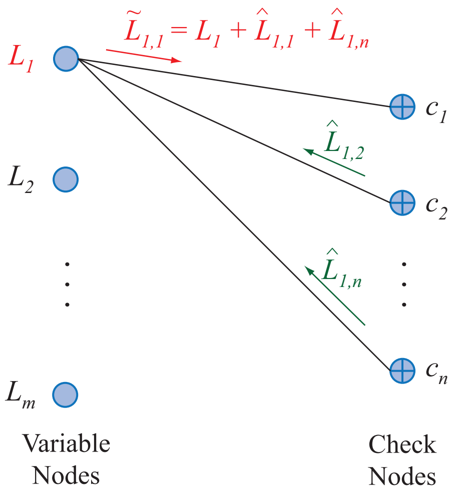

Figure 2.2: Updating $\tilde{L}_{i,j}$ values in the variable node connections.

Check Node Update: At the check nodes, the outgoing extrinsic L-values $\hat{L}_{i,j}$ are calculated according to the *box-plus* operation [15] whose expression is

$$\hat{L}_{i_1,j} = 2 \cdot \operatorname{atanh}\left(\prod_{i \neq i_1} \tanh\left(\frac{\tilde{L}_{i,j}}{2}\right)\right)$$

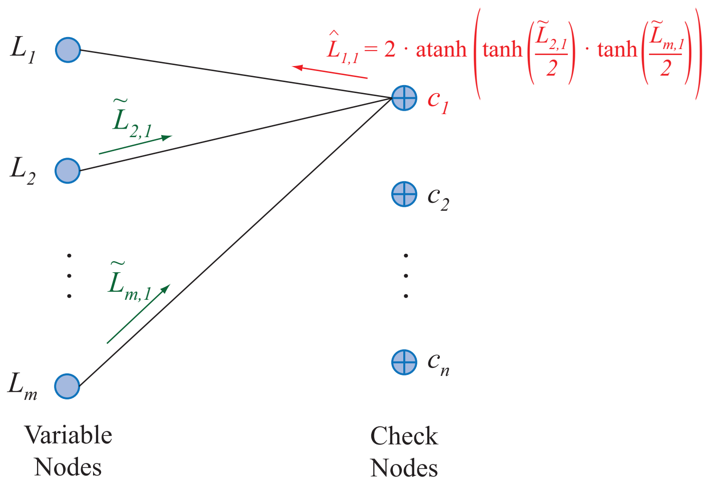
L_1
L_2
L_m
$\hat{L}_{1,1} = 2 \cdot \operatorname{atanh}\left(\tanh\left(\frac{\tilde{L}_{2,1}}{2}\right) \cdot \tanh\left(\frac{\tilde{L}_{m,1}}{2}\right)\right)$
$\tilde{L}_{2,1}$
$\tilde{L}_{m,1}$
c_1
c_2
c_n
Variable Nodes
Check Nodes

Figure 2.3: Updating $\hat{L}_{i,j}$ values in the check node connections.

After the check node update the iteration is finished. If all the check equations are fulfilled the iterative process is stopped. The decoded bits are obtained from the finally updated variable node L-values.

$$\hat{y}_i = \operatorname{sign}\left\{ L_i + \sum_j \hat{L}_{i,j} \right\}$$

where the sum is performed over all incident connections on the corresponding variable node.

2.1.3 Decoding Process on Non-Binary LDPC Codes: Log-Domain Fourier Transform Decoding

Non-binary LDPC codes are defined over the finite Galois field $GF(q = 2^p)$. By encoding over GF(q), each parity check becomes more complex. The decoding of non-binary LDPC codes is not equivalent to the binary case as the non-binary decoder operates on the symbol level, and not on the bit level. The decoding algorithm is based on the probability and log domain versions of the Fourier transform over GF(q), since it reduces the algorithm complexity [8]. The information in the messages being passed is expressed in terms of L-values.

The channel messages L_i^a are loaded into the variable nodes. These messages represent the prior L-values for the symbol a, where $a \in GF(q)$. If the channel messages satisfy the check equations, the decoding process is successful and is finished. If not, the iterative process continues as follows.

Variable Node Update: Each variable node generates outgoing messages for each connection and symbol, and they are calculated as the extrinsic sum of the previously calculated (or initialized) L-values $\hat{L}_{k,i}$.

$$\tilde{L}^a_{i,j} = L^a_i + \sum_{j \neq j_1} \hat{L}_{i,j} - \alpha_{i,j}$$

$$\alpha_{i,j} = \max \ \tilde{L}^a_{i,j}$$

The normalisation value $\alpha_{i,j}$ is introduced for avoiding numerical overflow problems.

Check Node Update: The check node messages are updated in the Fourier Domain. The fast Fourier transform (FFT) is used for transforming the messages into the spectral domain. The FFT can produce negative numbers. Thus, for handling these negative numbers in the log domain, two dimensions (magnitude and sign) are used.

The messages $\tilde{L}^a_{i,j}$ are converted to signed L-values, where $\tilde{Q}^a_{i,j}(mag) = \tilde{L}^a_{i,j}$ and $\tilde{Q}^a_{i,j}(sig) = 1$. These are permuted and transformed to the Fourier domain [8] according to

$$\left[\tilde{Q}^a_{i,j}(mag), \tilde{Q}^a_{i,j}(sig)\right] = \mathcal{F}\left[\Pi\left(\tilde{L}^a_{i,j}\right)\right].$$

The outgoing Fourier domain check node messages are updated as follows:

$$\hat{R}^a_{i,j}(sig) = \prod_{i \neq i_1} \tilde{Q}^a_{i,j}(sig)$$

$$\hat{R}^a_{i,j}(mag) = \sum_{i \neq i_1} \tilde{Q}^a_{i,j}(mag)$$

The outgoing check node signed L-values are converted to the log-domain applying an inverse FFT and an inverse permutation [8]

$$\left[\hat{L}^a_{i,j}(mag), \hat{L}^a_{i,j}(sig)\right] = \mathcal{F}^{-1}\left[\Pi^{-1}\left(\left[\hat{R}^a_{i,j}(mag), \hat{R}^a_{i,j}(sig)\right]\right)\right].$$

The signed L-values $\hat{R}^a_{i,j}(mag)$ are converted back to check node messages, where $\hat{L}^a_{i,j} = \hat{L}^a_{i,j}(mag)$

The check equations are applied to the estimated L-value messages (input L-value plus the sum of all the incoming L-value node connections) such that

$$\hat{y}_i = \text{argmax } L^a_i + \sum \hat{L}^a_{i,j}$$

If the check equations are satisfied the decoding process is finished. If not, the process continues with a new iteration until a valid message word is estimated or the maximum number of iterations is reached.

2.2 CUDA Architecture

The high parallel processing performance of graphics processing units (GPUs) has been used traditionally to transform, light and rasterize triangles in three-dimensional computer graphics applications. In recent architectures, however, the vectorized pipeline for processing triangles has been replaced by a unified scalar processing model based on a large set of streaming processors. This change has initiated a consideration of GPUs for solving general purpose computing problems, and triggered the field of general-purpose computing on graphics-processing units (GPGPU). GPGPU appears to be a natural target for scientific and engineering applications, many of which admit highly parallel algorithms [2].

The hardware abstraction based on the GPU unified scalar processing model is denoted CUDA (Compute Unified Device Architecture) in all the current NVIDIA graphic cards. This technology allows to execute thousands of concurrent threads for the same process, which makes this kind of devices ideal for massive parallel computation. The CUDA devices have special hardware resources (with their respective constraints) and, with a deep knowledge about them, it is possible to achieve high speedup values for a CUDA implementation compared to a non-parallelized CPU version.

The CUDA device and the computer which it is attached to are referred as *device* and *host* respectively. Also, all the CUDA or hardware mentioned specifications in this section are referred to the NVIDIA GeForce GTX 275 model, the one used in this thesis (for more details about the GTX 275 hardware specifications, please refer to Appendix C).

CUDA devices are programmed using a extended C/C++ code and compiled using the NVIDIA CUDA Compiler driver (NVCC). After the compilation process, the serial code is executed on the host and the parallel code is executed on the device using kernel functions.

2.2.1 Threads Arranging and Execution

A CUDA device has a limited number of streaming multiprocessors (SM). Each SM is able to execute concurrently up to 1024 threads, which have to be arranged in blocks. A block contains up to 512 threads, arranged with an 1D, 2D or 3D distribution (Figure 2.4).

A SM is able to process simultaneously between 2 and 8 blocks of threads and it is important to have a SM occupancy of 100% for achieving the maximum performance. A power-of-two number of threads per block helps to reach this goal. Finally, the blocks are partitioned in *grids*, where a grid is a 1D or 2D arrangement of the total number of blocks containing all the data to be processed. The threads/block and blocks/grid settings are configured for the execution of each kernel. Thus, each kernel launches the execution of one grid of blocks, which contains all the data to be processed arranged in blocks of a fixed number of threads.

A kernel is launched as a C/C++ function (but with some extensions) from the host and executed on the device. A call to a kernel can be written as *___global___ kernelName <<<numGrids, numBlocks>>> (parameters_list)*, where *___global___* indicates to the compiler that the kernel has to be executed on the device, *<<<numGrids, numBlocks>>>*

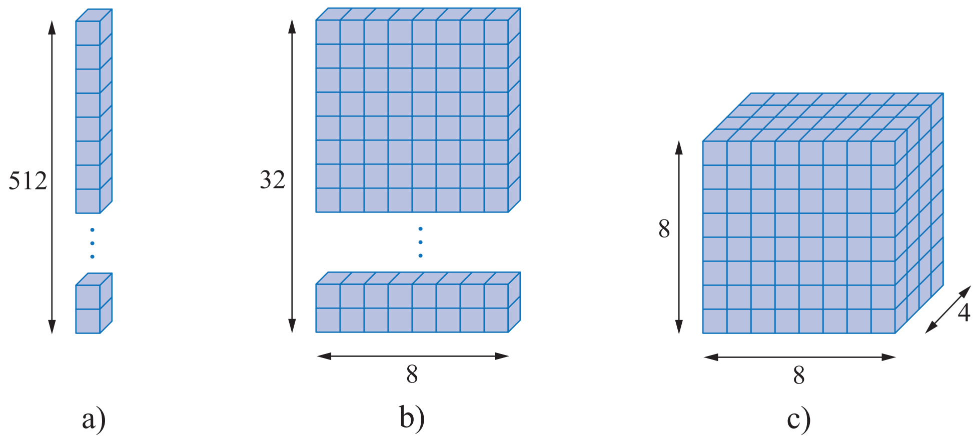

Figure 2.4: Different block arrangements for a number of 256 threads.

details the blocks/grid and threads/block settings and *parameters_list* is the typical parameters list passed to a standard C/C++ function. For more information, please refer to the CUDA programming manual [12].

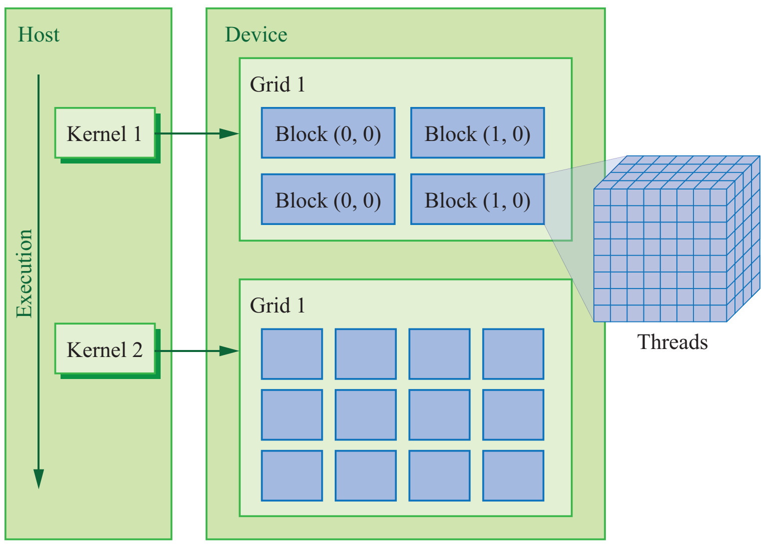

Figure 2.5: Grid organization and execution.

2.2.2 Device Memory Types

CUDA supports several kinds of memory than can be used for achieving high speed performance in the kernels execution. The description of each memory is as follows:

Global memory: This is the largest capacity memory in the device (768 MB in the GTX 275). It is a read and write memory and can be accessed by all the threads during the application execution (the stored data remains accessible between different kernel executions). It is off chip, i.e. slow interfacing has to be used, but it can achieve good performance if each continuous thread index accesses continuous memory data positions. The global memory is the communication channel between host and device, since the data to be processed have to be sent from the host to the device, and sent back to the host after the application execution.

Constant memory: This memory is available to all the threads during the application execution. It is a read-only memory by the device, and only the host is able to write data to it. It is a short-latency, high-bandwidth memory if all threads simultaneously access the same location.

Shared memory: This is an on-chip memory available to all the threads in one block. It is a read and write memory, with a size of 16 KBytes per SM. The data is available during the kernel execution and the access can be very fast if the available shared memory banks are accessed in a parallel way.

Registers: These are on-chip memory registers available to each independent thread. The local variables in a kernel are usually stored in the registers, performing a full speed access. If a variable can not be allocated in the on-chip hardware registers, then it is stored in the global memory with a slower access.

The variables created in CUDA can be allocated in each of the available memory types, depending on which keyword precedes the variable declaration or where are they declared. Table 2.1 has a summary of the different variables and memory associations.

Variable declaration	*Memory*	*Scope*	*Lifetime*
Automatic variables other than arrays	Register	Thread	Kernel
Automatic array variables	Local	Thread	Kernel
___shared___, int SharedVar;	Shared	Block	Kernel
int GlobalVar;	Global	Grid	Application
___constant___	Constant	Grid	Application

Table 2.1: Memory types in a CUDA device.

2.2.3 Textures

The graphic cards are designed for a high performance in the 3D and 2D image processing. There is a special memory where bitmaps or textures are stored for a special processing. This memory is denoted texture memory. The texture memory can only be read by the threads, but it is accessed in a very fast way if continuous thread indexes access to spatially close *texels* (or texture elements). This memory also is optimized for certain functions such as linear interpolation or anti-aliasing.

The texture memory can be used as look-up table, with the extra function of linear interpolation between points. For a high performance access it is required that the stored data has some spatial relation. If not, the penalty is enough for not achieving better performance than the global memory.

2.2.4 Performance Considerations

The execution speed of a kernel depends on the resource constraints of the device. Some considerations have to be taken in mind while programming CUDA devices, in order to achieve the the maximum execution performance [11]. These considerations can be summarized as follows:

Block sizes: For a fast execution speed of a kernel, all the threads in a warp (group of 32 continuous threads) should execute the same instruction. Thus, an important performance factor is that the number of threads per block is proportional to 32.

Divergent threads: When using conditional sentences with the thread index involved, the threads are executed in two different branches. In this way, not all the threads are executing the same instruction and the performance could be slowed down.

Global memory access: The global memory is the largest memory in the CUDA device, but also it is the slowest one. For achieving a good global memory performance it is extremely important to access the data in a coalesced way. It means, each continuous thread index in a warp has to access continuous memory positions. This can be managed by taking care about the data size and the way the memory is indexed and accessed in the kernels.

Shared memory: Each SM has 16 KB of available shared memory. If the memory access is paralellized, then the access can be as fast as one on-chip register. Shared memory banks are organized such that successive 32-bit words are assigned to successive banks. Thus, if successive 32-bit threads access different shared memory banks, these access are paralellized achieving a high bandwidth. If not, the accesses are serialized (bank conflicts).

Occupancy: It is clear to understand that if the SM are 100% occupied, the performance is maximized. Thus, all the variables which can limit this occupancy: maximum number of threads per SM and the number of threads per block, the maximum available shared memory per SM and the amount of shared memory used by each kernel.

Hardware functions: CUDA has hardware implemented fast mathematical functions, however with a minor accuracy. If this precision lost is affordable, these functions can be used and the speed improvement is considerable.

For more information about CUDA programming, please refer to [11] [12] [13].

Chapter 3

Implementation

The LDPC decoder algorithms here implemented, are based on the C++ reference code in [10]. The objective of the implementation is to speed up the decoding process, taking advantage of the massive parallel processing offered by the CUDA technology. The reference code can be improved in several ways, depending on its nature. Sometimes the code only needs to be adapted to the CUDA architecture. Other times, the parallelization of the code simplifies the execution and enhances the performance. In occasions, it is better to start the code from zero. And in all these cases, the best option is to use all the specific hardware resources that seriously improve the processing time.

A good knowledge of the source implementation and its theoretical background, and a deep knowledge of the CUDA architecture leads to find the best way for achieving a good performance over the original implementation. This is the goal in this chapter.

The data used in the implementation has single precision, so all the variables involved in the data processing are *floats*. Single precision gives a good compromise between accuracy and performance in CUDA devices of architecture generation 1.3 or below.

The hardware platform used for the implementation is based on an Intel Core 2 Quad Q6600 @ 2.4 GHz / 2 GB RAM system, and a NVIDIA GeForce GTX 275 graphic card.

3.1 LDPC GF(2) Decoder

The (binary) LDPC GF(2) decoding is based on a message passing algorithm over the Tanner graph. This graph completely defines the parity check matrix and contains all the necessary information for creating all the variables (information and auxiliary variables) in the decoding algorithm.

The CUDA implementation is focused on a generic decoding algorithm which means that it has to be able to process LDPC codes of whatever parity check matrix size. This reduces the range of possibilities about using different memories in the CUDA device, since the matrices for storing the message passing L-values $\tilde{L}$ and $\hat{L}$ have to be stored in the global memory.

It is necessary to keep in mind some implicit characteristics of the LDPC GF(2) decoding algorithm. First, there are continuous random memory accesses during the decoding. This is not optimal for the CUDA technology, since the hardware could not be able to do coalesced memory data reading or writing. Second, the *tanh*() and *atanh*() functions have a high computational cost. The first handicap is not very tractable, because the aleatory memory accesses are in the LDPC codes nature. The second one can be treated in different ways, as it is explained in the next sections.

During this section, some recurrent variable names are used. The meaning of these variables is as follows:

BN/CN: Number of Tanner graph bit/check nodes.

cudaBNConn/cudaCNConn: Number of connections for a particular Tanner graph bit/check node.

***maxBNConn/maxCNConn*:** Maximum number of connections of the bit/check nodes.

3.1.1 Algorithm Flowchart

The algorithm flowchart is shown in Figure 3.1. This scheme follows the C++ reference implementation [10].

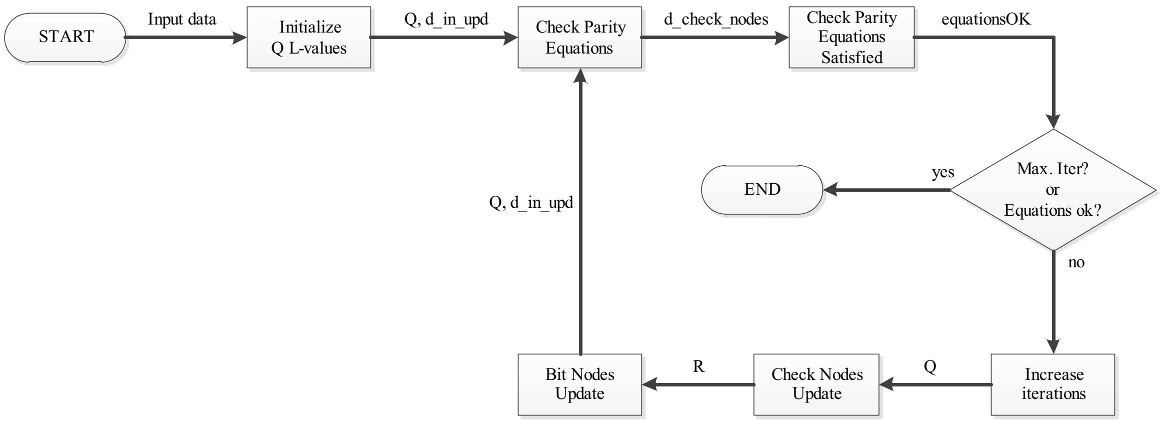

Figure 3.1: Algorithm flowchart.

The description of the variables in Figure 3.1 is as follows:

***d_in_upd*:** Updated input values with the L-values Q converted to binary information.

***d_check_nodes*:** Vector containing the result of the parity check equations in each Tanner graph check node.

***equationsOK*:** This variable indicates if all the parity equations are successful.

***Q*:** Outgoing L-values ($\hat{L}$) from the Tanner graph check nodes.

***R*:** Outgoing L-values ($\tilde{L}$) from the Tanner graph bit nodes (variable nodes).

Initially, the input data is loaded into the GPU device, and the iterations counter set to zero. Before it starts decoding, the L-values Q (matrix Q) are initialized with the input L-values, and the updated input data vector (*d_in_upd*) is initialized to the corresponding input L-value converted to binary. Here the loop starts, and the updated input data is immediately checked. If the checking succeeds, the decoding process is interrupted because the data has not bit errors and is valid. If this is not the case, the number of iterations is increased and the check node update performed. The check node update generates the extrinsic L-values R from the L-values Q initialized before. These values are used by *bit node update* for generating the new L-values Q and updating the input data again. Here, the loop is closed and performed once again by checking the parity check nodes with the updated input data.

Each process in the flowchart is implemented as a kernel in the CUDA source code. All the kernels are ___*global*___, since they are executed in the device. The loop itself is controlled on the host device.

3.1.2 Data Structures

The data structures used here are created according to the requirements of the decoding algorithm. The Tanner graph provides all the necessary information.

The Tanner graph bit nodes and check nodes share the same node connections, but connected in a random way. The algorithm needs two different matrices for managing the outgoing L-values from the different nodes (Figure 3.2).

The R matrix has as many rows as bit nodes exist in the Tanner graph. The number of columns is equal to the maximum number of bit node connections. The same holds for the Q matrix, but considering the Tanner graph check nodes. As Figure 3.2 shows, an irregular LDPC code has not the same number of connections in each node, so there are some no entries which are not valid. These cells can be avoided for saving processing time. Because of this, two auxiliary variables (*cudaBNConn* and *cudaCNConn*) contain the number of connections for each corresponding bit and check node (Figure 3.3)

Since R and Q are linked by the same node connections, one position in the R matrix can require to read or write the matching position in the Q matrix, and vice-versa. Thus, it is necessary to know the Q index position for all the R elements, and vice-versa. This is the reason for the creation of the *cudaBN2CN* and *cudaCN2BN* matrices (Figure 3.4).

A last auxiliary variable is needed for knowing to which bit node each check node is connected in the Tanner graph. This variable is *cudaCheckNodeConnections*, a dimension $CN \times maxCNConn$ matrix. It contains the bit node index for each check node connection.

All these variables are allocated in the global memory. The reason is that the algorithm has to have a general character, and has to be able to decode whatever LDPC code size is used. As the matrices have to be completely loaded in the memory (they can be accessed randomly), the only possible memory is the global memory. The data then has

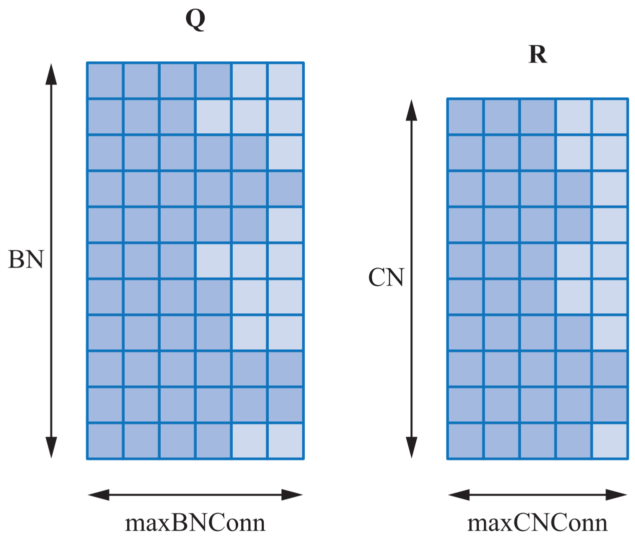

Figure 3.2: R and Q matrices.

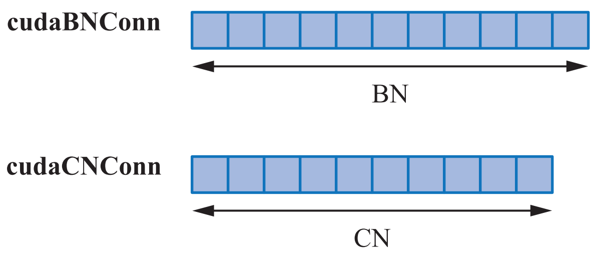

Figure 3.3: Vectors *cudaBNConn* and *cudaCNConn*.

to be arranged for being accessed in a coalesced way. Table 3.1 shows a summary of the variables, with the corresponding size and indexing expression. It has to be kept in mind that most of the CUDA variables have a 2D interpretation, but in fact they are 1D arrays.

Name	Dim. i	Dim. j	Indexing
Q	*BN*	maxBNConn	$j \cdot \mathrm{BN} + i$
R	*CN*	maxCNConn	$j \cdot \mathrm{BN} + i$
cudaBNConn	*BN*	-	i
cudaBNConn	*CN*	-	i
cudaBN2CN	*BN*	*maxBNConn*	$j \cdot \mathrm{BN} + i$
cudaCN2BN	*CN*	*maxCNConn*	$j \cdot \mathrm{CN} + i$
cudaCheckNodeConnections	*CN*	*maxCNConn*	$j \cdot \mathrm{CN} + i$

Table 3.1: Dimensions and indexing of the variables

Each kernel processes one Tanner graph node. Inside each kernel, a loop processes all the node connections, counting up to the number of connections of the node. When the variables are indexed as $j \cdot NODES + i$, they access adjacent memory positions with

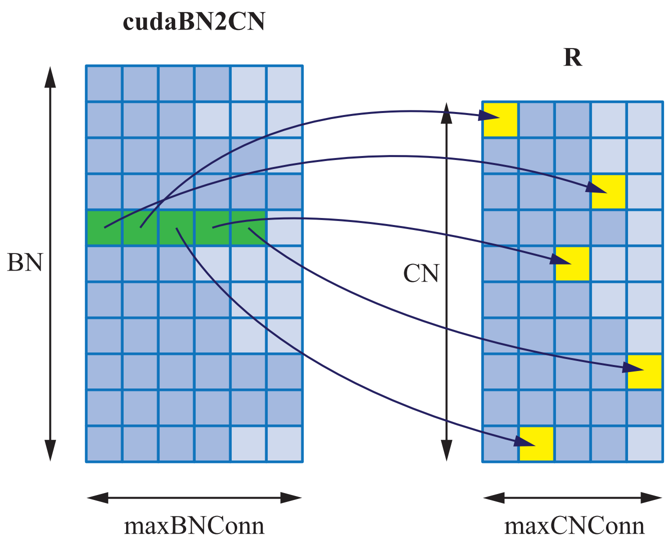

Figure 3.4: Matrix *cudaBN2CN* pointing the matching indices in Q.

adjacent threads: as each thread processes a node, during the loop execution the adjacent threads (nodes) access to coalesced memory positions in each iteration.

3.1.3 Kernel Implementations

The kernels execute one thread per node, with a loop responsible of processing the different node connections. The loop is a good option since the loop count is small and the nodes don't have the same number of connections. Therefore, all the kernels have 1D block and grid configuration.

3.1.3.1 Initialization of Q (*cudaInitializeQ*)

The Q matrix (estimated L-values) has to be initialized before the decoding loop starts. Each thread (Tanner graph bit node) reads the corresponding *d_in* input L-value and stores it in a temporal variable. This value is converted to binary and stored in *d_in_upd*. An internal loop process all the connections of the current node and assigns to them the input L-value stored in the temporal variable (Figure 3.5).

Each continuous thread index accesses continuous memory positions, optimizing in this way the memory access performance. Table 3.2 shows the block and grid configuration for this kernel.

	X Dimension
Threads/Block	256
Blocks/Grid	$\lceil BN/256 \rceil$

Table 3.2: Block and grid settings for the **cudaInitializeQ** kernel

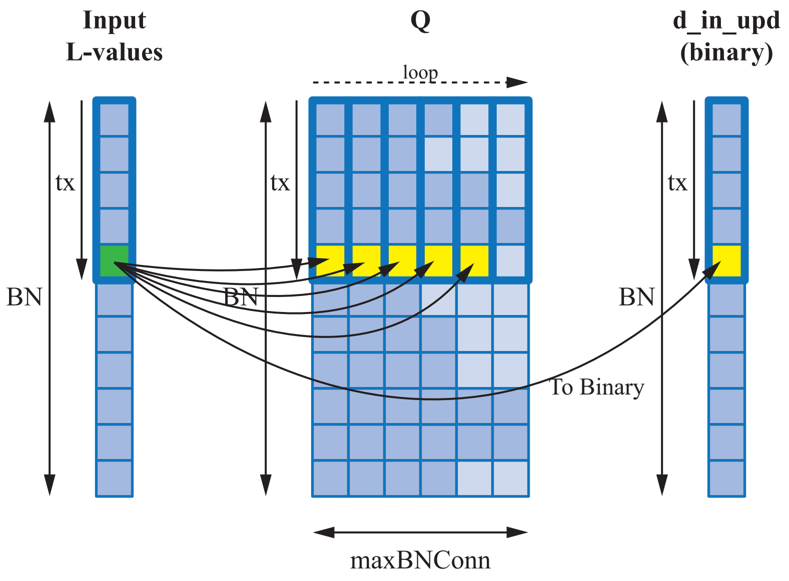

Figure 3.5: Description of the **cudaInitializeQ** kernel.

3.1.3.2 Check Parity Equations (*cudaCheckEquations*)

This kernel checks the parity equations in each Tanner graph check node. The input of the kernel is *d_in_upd*, where the estimated decoded word is stored in binary format.

For each thread (Tanner graph check node) a loop processes the *xor* of all the *d_in_upd* values corresponding to all the current check node connections. The *xor* result of each node is stored in the corresponding element of a temporal vector. If all the elements of the temporal vector are zero, then the estimated word is a valid code word.

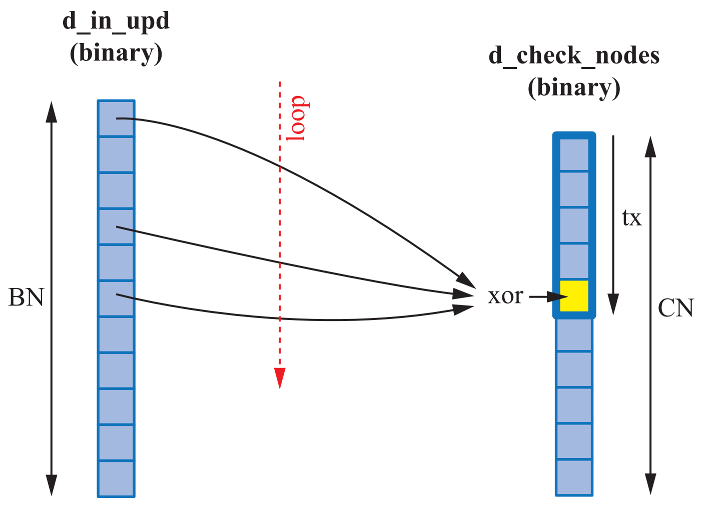

Figure 3.6: Description of the **cudaCheckEquations** kernel.

With the 1D block configuration all the read and write memory accesses are coalesced, since each continuous thread index accesses to continuous memory positions. If a 2D block configuration is performed (one thread per node connection), then a shared memory block is required as temporal memory. The shared memory can not be accessed without bank conflicts in this case, slowing down the kernel execution performance.

	X Dimension
Threads/Block	256
Blocks/Grid	⌈CN/256⌉

Table 3.3: Block and grid settings for the **cudaCheckEquations** kernel

3.1.3.3 Check Parity Equations Satisfied (*cudaCheckEquationsSatisfied*)

A sum reduction algorithm is applied to *d_check_upd* in order to verify if all the parity check equations have been satisfied.

A sum reduction algorithm is used taking advantage of the parallel processing. In this way, the sum is processed in $\log_2 x$ steps, where x is the number of elements of the input vector. The reduction algorithm is efficient for a power-of-two number of elements. So, if the input vector has not power-of-two elements, the necessary zeros are appended to it. After the necessary iterations, the sum reduction algorithm returns 0 in its first index if all the elements of the input vector are zero (Figure 3.7).

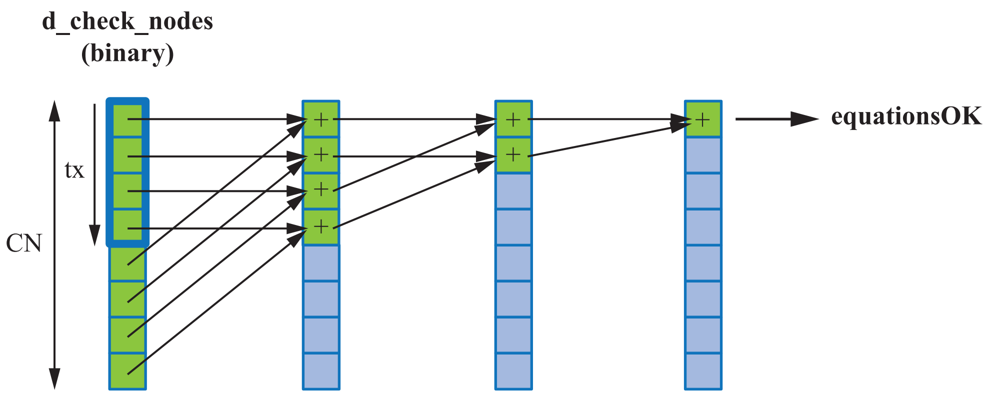

Figure 3.7: Description of the **cudaCheckEquationsSatisfied** kernel.

This is the only one kernel using 512 threads per block as it is faster in most cases. The block and grid configuration is shown in Table 3.4.

	X Dimension
Threads/Block	512
Blocks/Grid	⌈CN/512⌉

Table 3.4: Block and grid settings for the **cudaCheckEquationsSatisfied** kernel

3.1.3.4 Check Node Update (*cudaCheckNodeUpdate*)

Each thread processes one Tanner graph check node. The objective is to calculate the estimated L-values from the extrinsic sum of the updated L-values coming from the corresponding connections of the bit nodes.

A first loop processes all the current node connections. For each node connection, the matching L-value from Q (random access through *cudaCN2BN*) is loaded. The *tanh* operation is applied to the loaded value divided by 2. At the same time, this value is multiplied with a total product temporal variable. Once the loop is finished, the shared memory has stored each of the $tanh(x)/2$ values, and the product of them is saved in a thread temporary variable.

A second loop processes again all the node connections, but now each value in the shared memory divides the total product (that is, the extrinsic product) and then the *arctanh* is applied and multiplied by 2. The final L-value is written in R (coalesced writing). The process is shown in Figure 3.8.

$$\mathrm{R}_{i_1,j} = 2 \cdot \mathrm{atanh}\left(\prod_{i \neq i_1} \tanh\left(\frac{\mathrm{Q}_{i,j}}{2}\right)\right)$$

The shared memory indexing is $i \cdot \mathrm{maxCNConn} + j$. This indexing eliminates all the bank conflicts, since each adjacent thread operates on a different shared memory bank. A parallel and full speed access is performed to the shared memory.

The standard $tanh()$ and $atanh()$ functions need a lot of processing time. The first option was to use these standard functions, but the kernel execution time was very slow. The second option was to use look-up tables for these functions and to store them in the constant memory or the texture memory. This improved performance, however the random accesses and the non-existent data locality made this option not as fast as expected. The final decision is to use optimized hardware functions in the GPU. Thus, the *tanh* and *atanh* functions can be expressed as follows:

$$\mathrm{atanh}(x) = \frac{1}{2}\mathrm{ln}\frac{1+x}{1-x}, \quad |x < |1$$

$$\tanh(x) = \frac{\mathrm{e}^{2x}-1}{\mathrm{e}^{2x}+1}$$

These expressions are implemented using the *___expf()*, *___logf()* and *___fdividef()* hardware CUDA functions, leading to a further performance boost at the expenses of less precision.

The block and grid configuration is shown in Table 3.5. The number of threads per block has a maximum value of 256. The used shared memory is, as maximum, 4 KBytes per block.

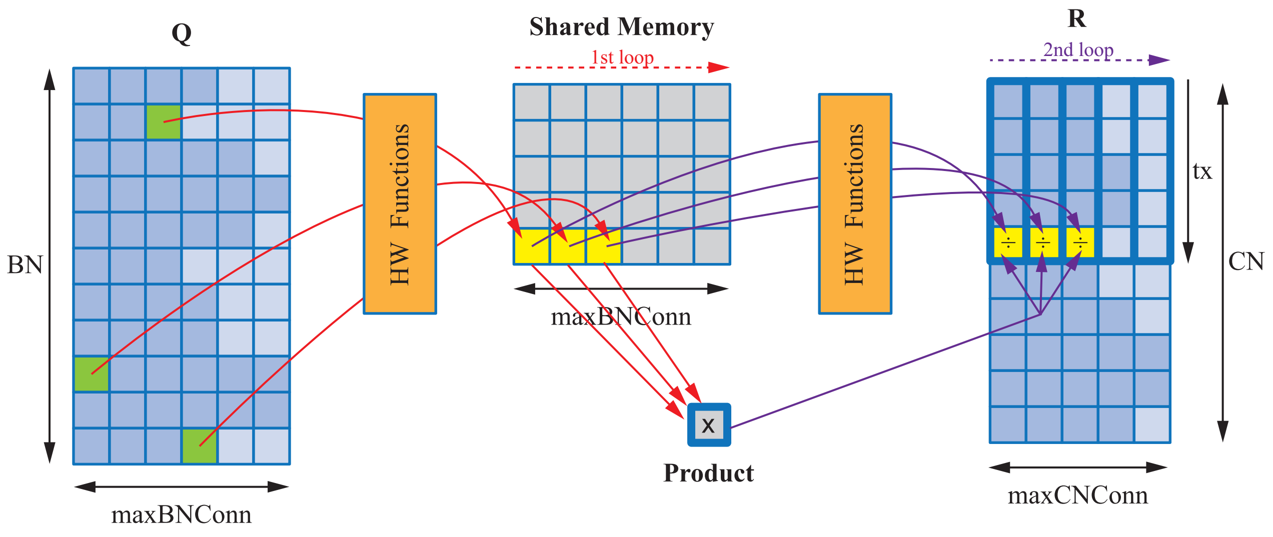

Figure 3.8: Description of the **cudaCheckNodeUpdate** kernel.

	X Dimension
Threads/Block	$\lfloor 256/\text{maxCNConn} \rfloor$
Blocks/Grid	$\text{CN}/\lfloor 256/\text{maxCNConn} \rfloor$
Shared Mem.	$\leq$ 4096 KBytes

Table 3.5: Block and grid settings for the **cudaCheckNodeUpdate** kernel.

With the current 1D block configuration the occupancy (calculated with the *CUDA Occupancy Calculator* tool) is not 100%, but it runs faster than with a 2D block arrangement. With a 2D block configuration, each thread corresponds to a different node connection. Thus, each thread needs to access simultaneously several shared memory positions for performing the extrinsic product, resulting in several bank conflicts. In such a case, the result is a low performance in the kernel execution speed.

3.1.3.5 Bit Node Updates (*cudaBitNodeUpdate*)

This kernel updates Q and *d_in_upd* from the recently calculated L-values R. Each Tanner graph bit node connection in Q is updated with the extrinsic sum of the matching L-values R. For each bit node, the updated input value is calculated as the sum of the original input L-value plus the sum of all the matching connections in R.

Each thread processes a Tanner graph bit node and contains a temporal variable for storing the sum of the associated L-values R. A first loop (up to *cudaBNConn*) processes all the bit node connections for storing them in the shared memory and simultaneously calculating the sum of the L-values. The entries of R are accessed in a random way, using *cudaBN2CN*. Once the loop is finished, the sum is calculated and the updated input value is converted to a binary value and written in a the output vector (coalesced writing). A second loop subtracts each imported L-value to the complete sum (extrinsic sum) and writes in a coalesced way this value to Q. The process is shown in Figure 3.9.

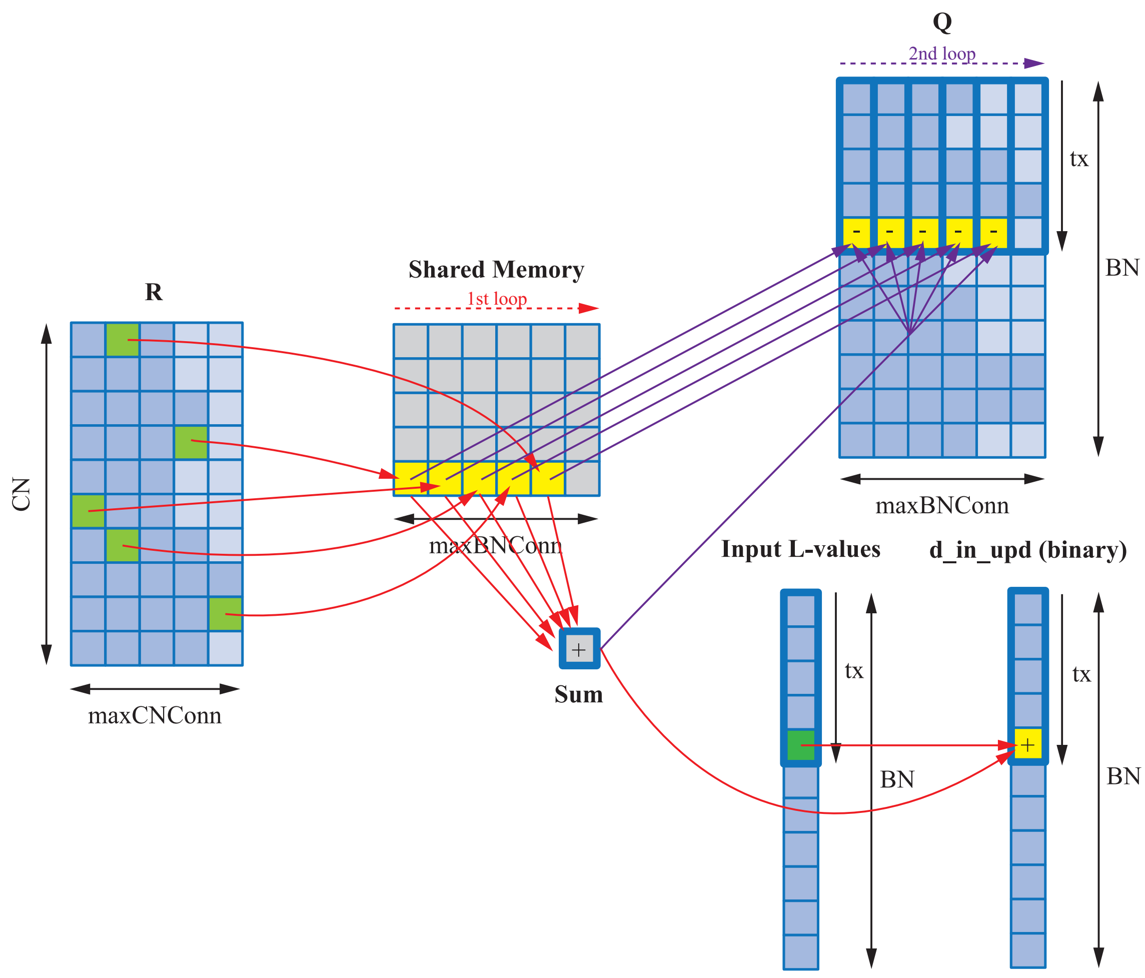

Figure 3.9: Description of the **cudaBitNodeUpdate** kernel.

Using a 2D block arrangement with shared memory, the kernel execution has not a good performance. The cause is that each thread needs to access simultaneously different shared memory positions, performing memory bank conflicts. These bank conflicts cause the serialization of the shared memory accesses, slowing down the kernel execution performance.

	X Dimension
Threads/Block	$\lfloor 256/\text{maxBNConn} \rfloor$
Blocks/Grid	$\text{BN}/\lfloor 256/\text{maxBNConn} \rfloor$
Shared Mem.	$\leq$ 4096 KBytes

Table 3.6: Block and grid settings for the **cudaBitNodeUpdate** kernel.

The occupancy of the streaming multiprocessor is not 100%, because the use of the shared memory limits the number of blocks per SM. Since the shared memory has to contain (number of threads · maxBNConn) floats ($\leq$ 256), it is only possible to have as many shared memory blocks as can fit in the available memory. This limits the number of threads per block, reducing the occupation to 50%, but it has been found that in this case the performance is superior than in other cases.

3.2 LDPC GF(q) Decoder

For the implementation of the LDPC GF(q) it is again necessary to focus the attention to two main different points: the arrangement of the data in the device memory and the use of the capabilities of the hardware for accelerating the data processing.

From the point of view of the data arrangement, it is interesting to interpret the variables as three-dimensional matrices, but bearing in mind that CUDA stores and process the data in 1D arrays. In the implemented algorithm, 2D and 3D data structures are needed, but the 3D ones are the most important since they are present in almost all the kernels in the main loop and are used all the time. The arrangement of the data in these 3D matrices is of huge importance since the memory needs to be accessed in a coalesced way for achieving high performance, such that various *float* values are handled in only one read/write memory access.

Some names of variables from the CUDA source code appear throughout this section. Their meaning is explained as follows:

BN: Number of bit nodes.

CN: Number or check nodes.

maxBNConn: Maximum number of bit node connections.

maxCNConn: Maximum number of check node connections.

3.2.1 Algorithm Flowchart

The flowchart of the LDPC GF(q) decoding algorithm is shown in Figure 3.10. Each process block represents a basic part of the algorithm and is implemented as a CUDA kernel. The incoming and outgoing variables of each process are also shown in the figure and their name is the same as in the CUDA code for simplicity and for a better understanding of the code.

The diagram of Figure 3.10 follows exactly the algorithm present in the C++ source code implementation in [8]. Initially, the number of iterations is zero and the input data is transferred to the *device* for calculating the initial L-values (*in_L*). The loop is executed at least once for checking the parity equations. The iterative decoding stops if there are no errors in the decoded data or if the number of iterations reaches the maximum value. When the iterative decoding process finishes, the data is transformed into a binary output and sent back to the *host*. The meaning of the variables and their respective properties are exposed in Section 3.2.2.

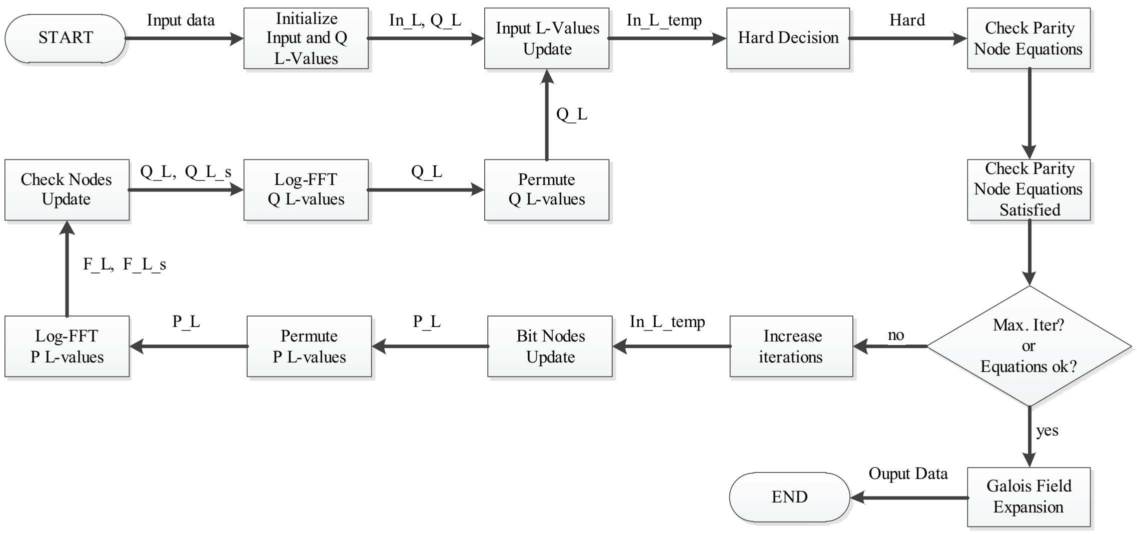

Figure 3.10: LDPC GF(q) Decoder flowchart.

Almost all the *kernels* follow approximately the C++ source code structure and have been adapted for CUDA parallel processing. The *log-FFT* algorithm has been completely designed from zero, since the C++ source code is based on the serial implementation [8].

3.2.2 Data Structures

Although the data is stored in the device memory as 1D array, they are interpreted as 1D, 2D or 3D structures, depending on the indexing of each one. This indexing points to the exact position of one element in the structure, and helps to set the grids and blocks configuration of the kernels. The order of the indices in the structures determine the final position of the data and, therefore, the performance of the memory access.

There are three kind of structures for the variables that are shown in Figure 3.11. CUDA needs fixed sizes for data structures, and since irregular LDPC codes have not a constant number of Tanner graph connections, it is obligatory to set the node connections dimension to the maximum value, taking care of not accessing the invalid positions during the processing. This is achieved using some auxiliary variables.

The 1D vectors (Figure 3.11-d) are used for the *hard* and output variables. The 2D matrices (Figure 3.11-b-c) are used for the *in_L* and *in_L_temp* variables. Finally, the 3D matrix (Figure 3.11-a) is applied to *P_L*, *FP_L*, *FP_L_s*, *Q_L*, *QP_L* and *QP_L_s*, and it becomes the most important data structure in the implementation, since the corresponding variables take part in the iterative loop, the most important part of the decoding algorithm. The first index is pertinent to the node, the second to the node connection and the third to the symbol element. By arranging the data in this order, it is possible to access the memory in a coalesced way. If each q adjacent Galois field elements (indexes defined by the nodes and the node connections) are accessed by q adjacent threads, this

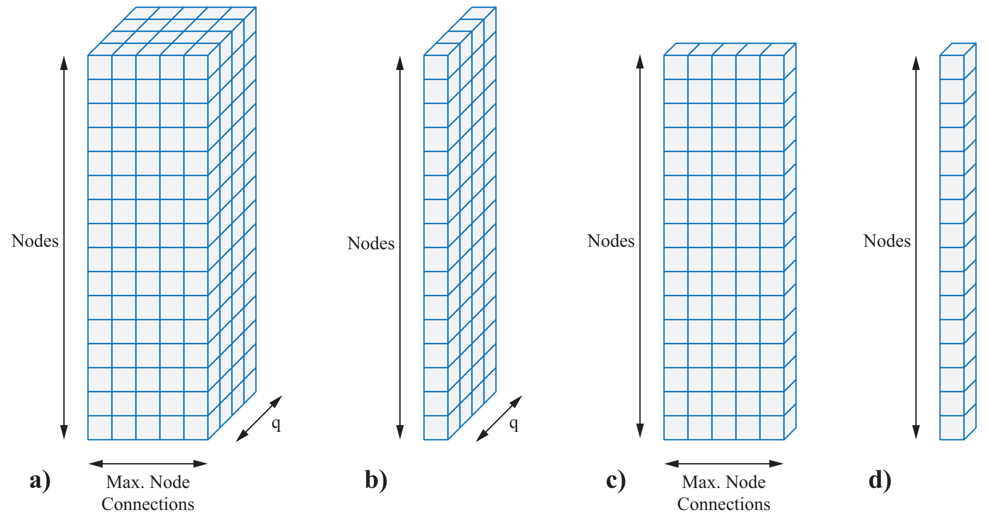

Figure 3.11: Data structures used in the LDPC GF(q) CUDA algorithm.

access is coalesced, as it is shown in Figure 3.12. This arrangement boosts the performance with increasing q.

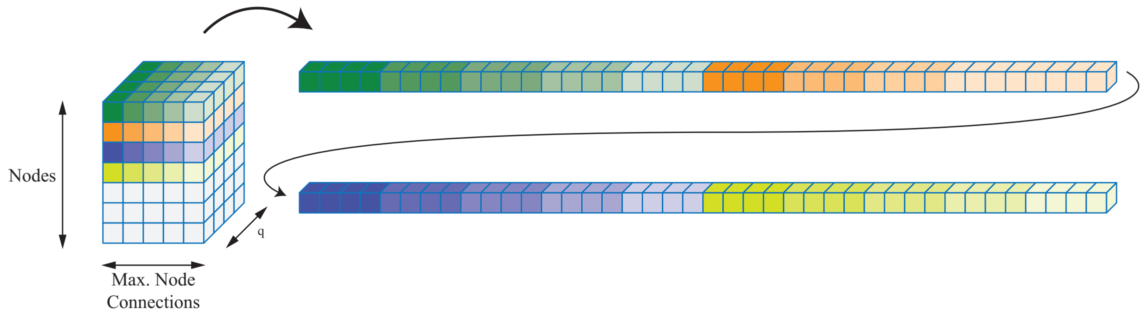

Figure 3.12: Continuous data allocation in the global memory.

The different variables present in the algorithm with their respective data structures and indexing expression are described in the following.

3.2.2.1 Non-constant variables

These variables are stored in the global memory because they need to be fully accessible (whichever position could be required in a random memory access) and they could be too huge for fitting in smaller memories (like, e.g., the constant memory or the texture memory). The description of each variable is as follows:

***in_L*:** Input L-values.

***in_L_temp*:** Input L-values updated with the Q_L values.

***P_L*:** Outgoing L-values from the bit nodes connections.

***FP_L*:** *log-FFT* module of *P_L* .

***FP_L_s*:** *log-FFT* sign of *P_L* .

***Q_L*:** Outgoing L-values from the check nodes connections.

***QP_L*:** *log-FFT* module of *Q_L* .

***QP_L_s*:** *log-FFT* sign of *Q_L* .

Their size and indexing properties are shown in Table 3.7.

Name	Dim. i	Dim. j	Dim. k	Indexing
in_L	BN	q	-	$i \cdot q + j$
in_L_temp	BN	q	-	$i \cdot q + j$
P_L	BN	maxBNConn	q	$i \cdot \text{maxBNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$
FP_L	BN	maxBNConn	q	$i \cdot \text{maxBNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$
FP_L_s	BN	maxBNConn	q	$i \cdot \text{maxBNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$
Q_L	CN	maxCNConn	q	$i \cdot \text{maxCNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$
QP_L	CN	maxCNConn	q	$i \cdot \text{maxCNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$
QP_L_s	CN	maxCNConn	q	$i \cdot \text{maxCNConn} \cdot \text{q} + j \cdot \text{maxBNConn} + k$

Table 3.7: Properties of the non-constant variables

3.2.2.2 Constant variables

As the size of these variables depends on the size of the parity check matrix and as they are accessed occasionally in an aleatory way, they are stored in the global memory. They are used for obtaining information from the parity check matrix. The following variables are constant:

***cudaNumBNConn*:** Number of connections for each bit node.

***cudaValBNConn*:** Value of the parity check matrix for each index.

***cudaIndBNConn*:** Index of the check node to which the actual bit node connection is attached.

***cudaBN2CN*:** Index of the actual bit node connection from the check node point of view.

***cudaNumCNConn*:** Number of connections for each check node.

***cudaValCNConn*:** Value of the parity check matrix for each index.

***cudaIndCNConn*:** Index of the bit node to which the actual check node connection is attached.

***cudaCN2BN*:** Index of the actual check node connection from the bit node point of view.

Table 3.8 provides information about sizes and indexing of these variables.

Name	Dim. i	Dim. j	Dim. k	Indexing
cudaNumBNConn	BN	-	-	i
cudaValBNConn	BN	maxBNConn	-	$i \cdot \text{maxBNConn} + j$
cudaIndBNConn	BN	maxBNConn	-	$i \cdot \text{maxBNConn} + j$
cudaBN2CN	BN	maxBNConn	q	$i \cdot \text{maxBNConn} \cdot \text{q} + j \cdot \text{q} + k$
cudaNumCNConn	CN	-	-	i
cudaValCNConn	CN	maxCNConn	-	$i \cdot \text{maxCNConn} + j$
cudaIndCNConn	CN	maxCNConn	-	$i \cdot \text{maxCNConn} + j$
cudaCN2BN	CN	maxCNConn	q	$i \cdot \text{maxCNConn} \cdot \text{q} + j \cdot \text{q} + k$

Table 3.8: Properties of the constant variables

3.2.2.3 Texture Tables

Some special functions are required to operate over the GF(q). These functions are addition, multiplication, inversion, decimal to exponential conversion and exponential to decimal conversion. These functions are implemented in tables of dimensions q or $q \cdot q$. The tables are allocated in the texture memory (1D or 2D arrangement) for taking advantage of the data locality and the memory speed (the texture memory has special fast access for data in near space). The table values are calculated following the C++ reference implementation [10].

***gf_inv*:** Inversion function.

***gf_add*:** Addition function.

***gf_mul*:** Multiplication function.

***gf_dec2exp*:** Decimal to exponential conversion.

***gf_exp2dec*:** Exponential to decimal conversion.

Table 3.9 shows the size and the arrangement of textures for these functions. The access to the textures is done with the **tex1Dfetch()** and **tex2D()** CUDA functions [12].

Name	Dim. i	Dim. j	Access
gf_inv	q −1	-	**tex1Dfetch**(*gf_inv* , i)
gf_add	q	q	**tex2D**(*gf_add* , i, j)
gf_mul	q	q	**tex2D**(*gf_mul* , i, j)
gf_dec2exp	q	-	**tex1Dfetch**(*gf_dec2exp* , i)
gf_exp2dec	q	-	**tex1Dfetch**(*gf_exp2dec* , i)

Table 3.9: Properties of the variables allocated in the texture memory

3.2.3 Kernel Implementations

The data structures have specific dimensions, and it is essential to configure the grid and block dimensions on each kernel for an optimal and fast execution. It is also necessary to keep in mind the constraints of the hardware for achieving high performance: maximum number of threads per SM, maximum number of blocks per SM, shared memory per SM, etc. The number of threads per block is fixed here to 256 because in this way the occupancy is 100% in most cases, using 4 blocks per SM and without exceeding the maximum available shared memory. The number of Galois field elements is a constant power-of-two value. This makes this value a good candidate for one of the block dimensions, because it is easy to achieve power-of-two block of threads in this way. The blocks with a power-of-two size are accessed faster by the hardware. The reason is that for a typical power-of-two value of threads per block (this is 64, 128, 256 or 512) each warp (32 continuous threads) can execute the same instruction.

All the kernels in this section are implemented as ___*global*___ kernels. The threads in the x and y dimensions are referred as tx and tx respectively.

3.2.3.1 Input L-Values Initialization (*cudaInLInitialize*)

The kernel initializes the input L-values matrix from the main input data of the decoder. It follows the C++ reference implementation [10], but adapted to CUDA.

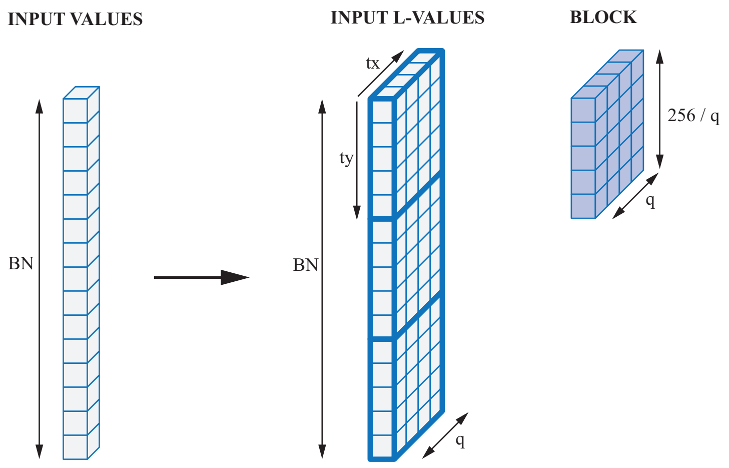

Figure 3.13: Description of the **cudaInLInitialize** kernel.

The output values have a 2D structure, as it can be seen in Figure 3.13. Each thread processes a different Galois field element in a particular Tanner graph bit node connection. The width of the block is the number of Galois field elements, which is a power-of-two number. The height of the block corresponds to $256/q$ threads or nodes. Thus, each ty identifies the Tanner graph variable node, and each tx identifies the Galois field element. Since q is a constant value, the grid is arranged in one dimension, and the number of blocks per grid is the total number of variable nodes divided by the block height (see Table 3.10).

The input values are indexed by $\mathrm{index} \cdot \mathrm{p} + k$, where k is the control variable of an internal loop. So, as the index has not a linear relation with the thread number, the access is not coalesced. Since this kernel is only executed once at the beginning, this is not significant.

	X Dimension	Y Dimension
Threads/Block	q	256/q
Blocks/Grid	BN/(256/q)	

Table 3.10: Block and grid settings for the **cudaInLInitialize** kernel.

The output values are indexed by $ty \cdot \mathrm{q} + tx$ as the thread number and the position have a linear relationship, the access to the memory is continuous.

3.2.3.2 Input L-values Update (*cudaInLUpdate*)

The input L-values are updated with the L-values Q_L in each decoding iteration. These updated L-values are used later for the hard decision. Each Galois field element in a variable node is the sum of the corresponding input L-value plus the corresponding Q_L values for all the variable node connections.

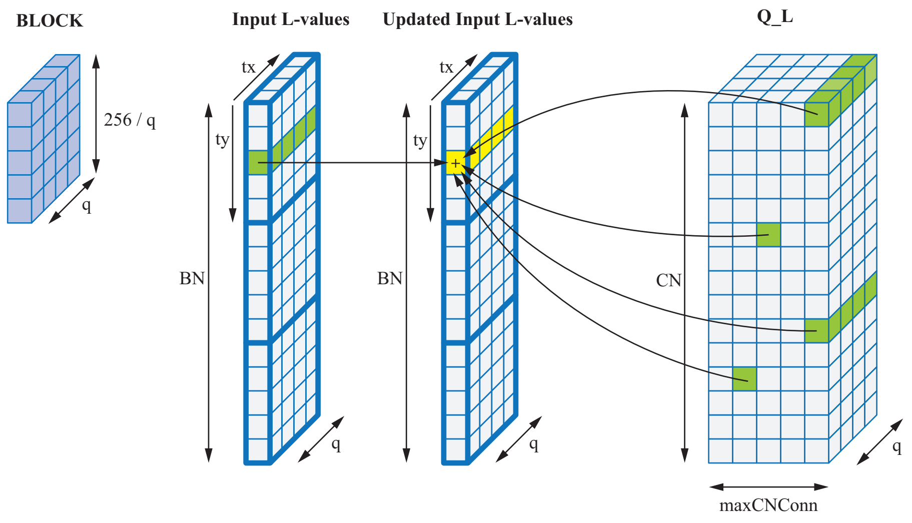

Figure 3.14: Description of the **cudaInLUpdate** kernel.

The block has a 2D structure. The width is set to q, because it is a constant power-of-two value, and the height is set to $256/q$. Each ty represents the Tanner graph variable node, and each tx represents the Galois field element. A loop is used in each thread for processing all the Tanner graph variable node connections and obtaining the correspondent symbol value.

	X Dimension	Y Dimension
Threads/Block	q	$256/q$
Blocks/Grid	BN/(256/q)	

Table 3.11: Block and grid settings for the **cudaInLUpdate** kernel

For each Tanner graph variable node, the loop counts up to the number of variable node connections, given by *maxBNConn*. For each iteration, the corresponding Galois field element value is read from Q_L using the corresponding *cudaBN2CN* index and its value is added to the final sum. Once the loop is completed, the final sum is written to *in_L_Temp*.

The input and output data are indexed by $\mathrm{ty} \cdot \mathrm{maxBNConn} \cdot \mathrm{q} + tx$. Since the index and the thread number have a linear relationship, the global memory is accessed in a continuous way. For the sum, the matrix Q_L is indexed $[\mathrm{BN2CN}[ty \cdot \mathrm{maxBNConn} \cdot \mathrm{q} + \mathrm{iteration} \cdot \mathrm{q} + tx]$, which is a random access. Since the random memory access is performed for a given node and node connection combination, there are q continuous memory accesses in every new aleatory indexing.

3.2.3.3 Hard Decision (*cudaHard*)

The updated L-values are required for the hard decision. Each thread processes a given Tanner graph node. In each thread, a loop processes all the Galois field elements searching for the symbol with maximum value. The position of this maximum value is converted to exponential using the *gf_exp2dec* texture table, and is written to the output structure.

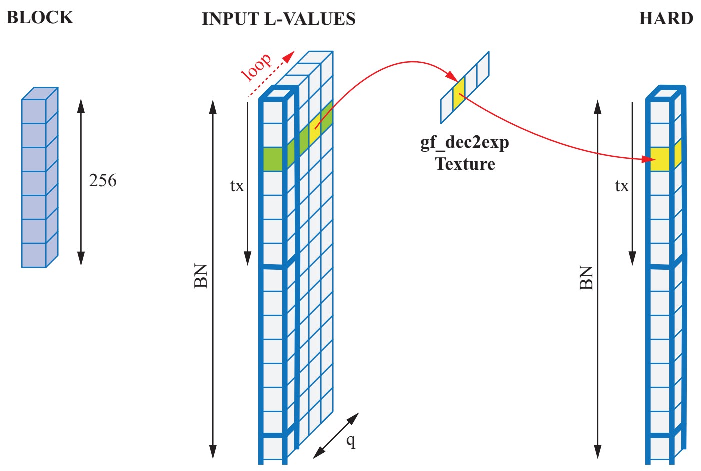

Figure 3.15: Description of the **cudaHard** kernel.

The block has only one dimension, and each thread represents a Tanner graph variable node. For each node, a *for* loop processes all the Galois field elements looking for the maximum value and its position.

	X Dimension
Threads/Block	256
Blocks/Grid	BN/256

Table 3.12: Block and grid settings for the **cudaHard** kernel

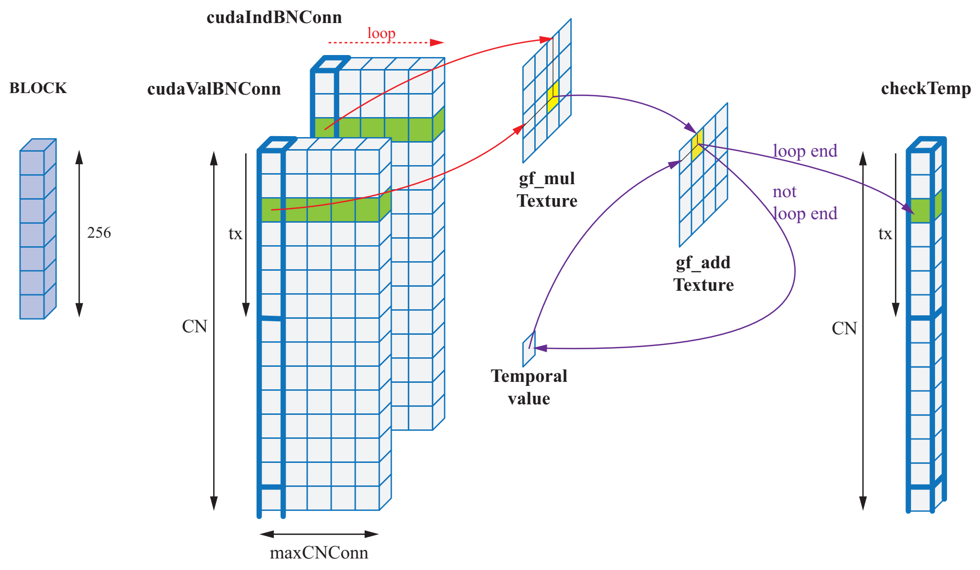

Figure 3.16: Description of the **cudaCheckNodeEquations** kernel.

This implementation is not optimal since the loop has to perform q iterations, but it is of high computational cost. An improvement can be to search for the maximum value with a reduction algorithm applied to each thread (node), using the shared memory (Figure 3.15).

3.2.3.4 Check Node Equations (*cudaCheckNodeEquations*)

The kernel checks the parity equations in all the check nodes, obtaining a binary output vector of size CN. After the kernel execution, each element of the vector is 0 if the correspondent parity check equation is satisfied, or 1 if not. The data structures involved in this kernel are *cudaValCNConn*, *cudaIndCNConn*, *gf_mul* and *gf_add*.

Each thread defines a Tanner graph check node, as it is shown in Figure 3.16. Following the C++ reference code, a temporal variable is set to -1 for each check node. A loop processes all the check node connections and increases the temporal variable with some operations in the GF(q) space (refer to the code for detailed information). The variables involved are *cudaValCNConn*, *cudaIndCNConn*, *gf_mul* and *gf_add*.

The indexing of all the variables in the global memory is performed with a linear relationship with the thread index for achieving a fast memory access.

	X Dimension
Threads/Block	256
Blocks/Grid	CN/256

Table 3.13: Block and grid settings for the **cudaCheckNodeEquations** kernel

3.2.3.5 Check Node Equations Satisfied (*cudaCheckNodeEquationsSatisfied*)

An addition reduction algorithm is applied to the temporal vector obtained in the *cudaCheckNodeEquations*. The partity equations are fully satisfied if the result of the sum is zero (all the node equations are satisfied), and in this case the decoding process can be interrupted.

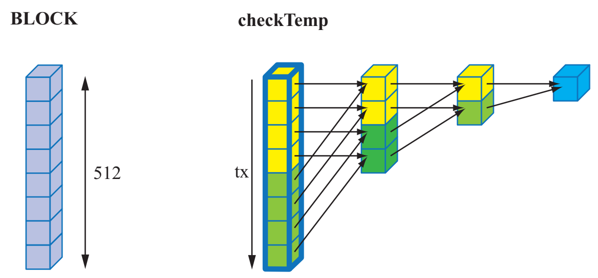

Figure 3.17: Description of the **cudaCheckNodeEquationsSatisfied** kernel.

The reduction algorithm takes advantage of the parallelization, performing the sum in $log_2(CN)$ iterations, where CN is the number of elements in the input vector. In each iteration, the upper half of the vector is added to the lower half in a simple parallel step. When the reduction is finished, the result is stored in the element with index zero.

The blocks have 1D structure and execute 512 threads for maximizing the performance. The shared memory is used to process the reduction algorithm in a small loop. The memory amount used is 4096KBytes per SM, since only two 512 threads blocks are processed in each SM.

	X Dimension
Threads/Block	512
Blocks/Grid	CN/512
Shared Memory	2048 KBytes

Table 3.14: Block and grid settings for the **cudaCheckNodeEquationsSatisfied** kernel

3.2.3.6 Bit Node Update (*cudaBitNodeUpdate*)

The main function of this kernel is to generate the new updated L-values P_L from the input L-values in_L and the estimated L-values Q_L. The kernel processes one thread per Tanner graph variable node. An internal loop processes all the node connections corresponding to each thread.

Each thread (Tanner graph variable node) reads the corresponding updated input L-values (q Galois field elements) from in_L. This initial value is updated with the extrinsic sum of the estimated L-values Q_L. For each node connection, the maximum value among the Galois field elements is subtracted from all of them, as indicated in [8].

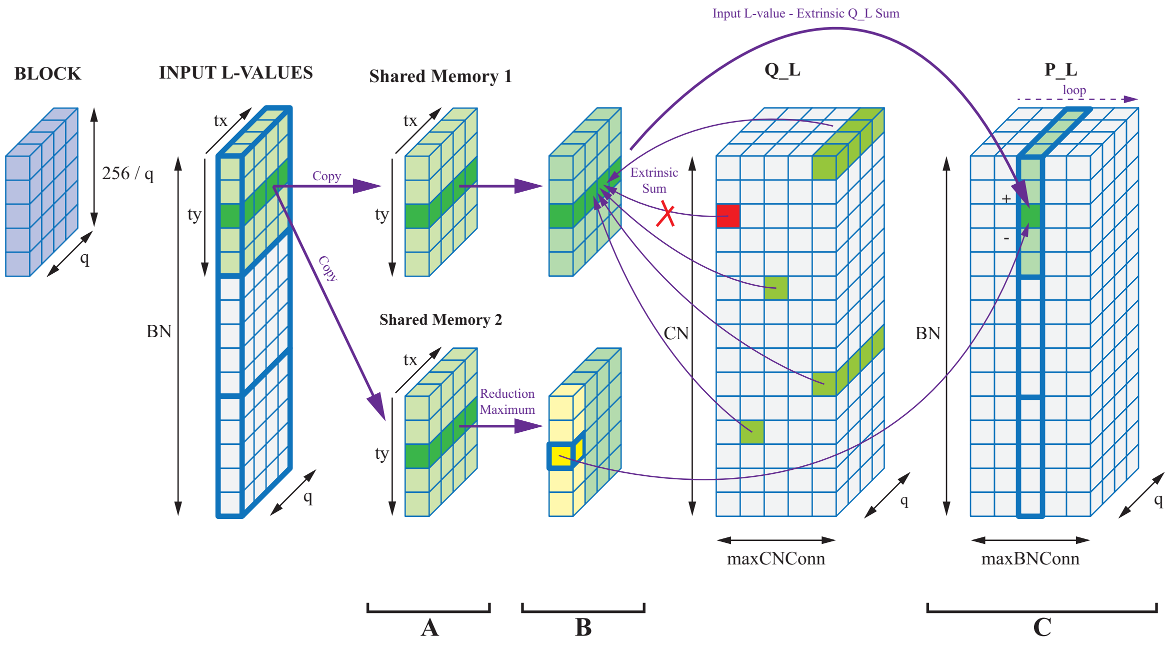

Figure 3.18: Description of the **cudaBitNodeUpdate** kernel.

The block width is fixed to q, and the block height is fixed to $256/q$. Thus, ty defines the Tanner graph variable node, and tx defines the Galois field element. The internal loop processes all the node connections (Figure 3.18-C). For each loop iteration (each variable node connection), the bit input L-values corresponding to the actual node are copied twice into the shared memory (Figure 3.18-A) in two different memory blocks. The extrinsic sum of the matching values Q_L (accessed through *cudaBN2CN*) is added to the first shared memory block (input L-values plus the extrinsic sum of Q_L values). At the same time, the second shared memory block performs the search of the maximum Galois field element of the input L-values for each node (Figure 3.18-B) using a reduction algorithm. This reduction algorithm performs a parallel search of the maximum value using a loop with $\log_2(q)$ iterations, where q is the Galois field dimension. After the reduction, the maximum symbol value is present in each zero index position. Finally, each Galois field element of the actual node and iteration (variable node connection) is updated with the values of the first shared memory block minus the maximum value, calculated in the second shared memory block (Figure 3.18-C). The use of the loop is important since it has a few number of looping iterations (not the same number for every node), and achieves a very good performance with this 2D block configuration.

Each block requires 2048 KBytes of shared memory (two shared memory blocks of 256(cells)$\cdot$ 4(bytes/cell)). Using 256 threads per block, each SM can manage up to 4 blocks. So, the maximum amount of shared memory required is 8 KBytes.

Each shared memory block is indexed by (block number)$\cdot 256 + ty \cdot \mathrm{q} + tx$. Since the shared memory operations are inside the node connections loop, continuous thread numbers access to different shared bank memory in each iteration. This avoids bank conflicts and provides maximum performance for the shared memory.

	X Dimension	**Y Dimension**
Threads/Block	q	256/q
Blocks/Grid	1	BN/(256/q)
Shared Mem.	2 · 4096 KBytes	

Table 3.15: Block and grid settings for the **cudaBitNodeUpdate** kernel

The input and output variable indexing has a linear relationship with the thread index. This means that the memory is accessed in a continuous way, achieving high performance. Each random access to Q_L performs q continuous readings. Thus, the random memory accesses performs better for increasing q values.

3.2.3.7 Permutation of *P_L* and *Q_L* (*cudaPermuteP* and *cudaPermuteQ*)

The Galois field elements of the Tanner graph variable node connections in P_L and the check node connections in Q_L are permuted by moving the $a-th$ element of the vector to position $a \cdot H_{i,j}$, where the multiplication is performed over GF(q) [8]. This process applies to P_L and Q_L almost in the same way (the Q_L permutation is the inverse permutation and uses a slightly different indexing). A shared memory block is used as temporary memory, allowing a parallel implementation of the permute operation. In *cudaPermuteP*, the shared memory stores the input data using the permuted indices (calculated using *gf_dec2exp*, *gf_mul* and *gf_exp2dec*), and writes the output data directly. The *cudaPermuteQ* only changes the way the permutation is done. In this case, the shared memory reads the input values using the same indices, but the output is written using the permuted indices. Since each thread accesses different shared memory banks, no bank conflicts are generated, performing a full speed memory access.

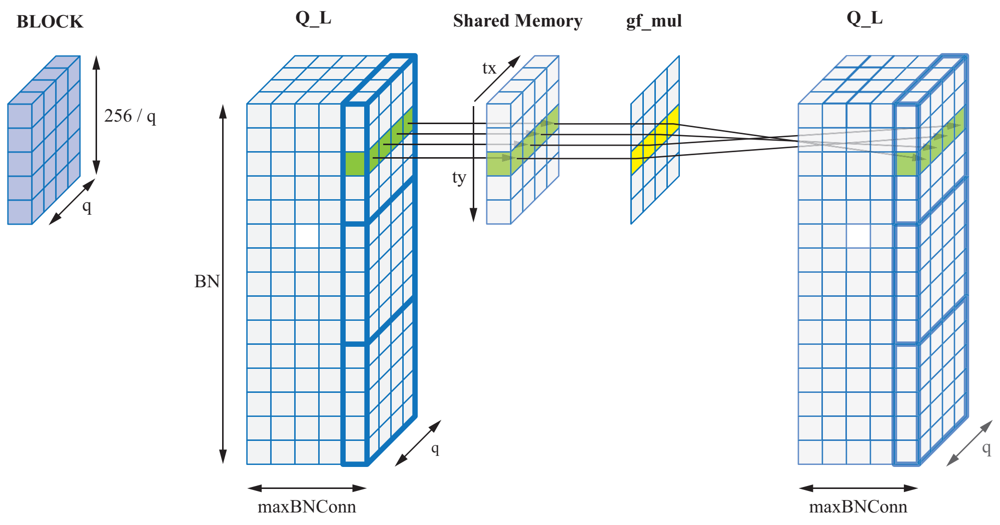

Figure 3.19: Description of the **cudaPermuteP** and **cudaPermuteQ** kernels.

Each *ty* corresponds to a different Tanner graph node connection, and each *tx* corresponds to the Galois field element. The *P_L* and *Q_L* data is interpreted as groups of q Galois field elements. The block width is fixed to q and the block height is fixed to $256/q$. Each thread is independent if it uses a shared memory block as temporal memory. The required amount of shared memory per block is 1 KByte. As the number of blocks per SM is 4, the shared memory is not a restriction.

cudaPermuteP		
	X Dimension	**Y Dimension**
Threads/Block	q	256/q
Blocks/Grid	1	BN · maxBNConn/(256/q)
Shared Mem.	4096 KBytes	

cudaPermuteQ		
	X Dimension	**Y Dimension**
Threads/Block	q	256/q
Blocks/Grid	1	CN · maxCNConn/(256/q)
Shared Mem.	4096 KBytes	

Table 3.16: Block and grid settings for the **cudaPermuteP** and **cudaPermuteQ** kernels

The complete Galois field vector is transferred to the shared memory. The new indexes are calculated using the *gf_exp2dec*, *gf_dec2exp* and *gf_mul* texture tables. The permutation is performed reading the new index from the shared memory and writing the old one in the *P_L* structure.

The input and output data indexing has a linear relationship with the thread index, so the memory access is continuous. The shared memory is mapped for avoiding bank conflicts, at least in the permutation of Q, since the access is not exactly the same in both kernels [10].

3.2.3.8 Log-FFT (*cudaLogFFTP* and *cudaLogFFTQ*)

The algorithm for performing the log-FFT is the same for P and Q structures and it follows a butterfly diagram. As explained in [8], two log-FFT executions (one for the module and one for the sign) have to be performed. Both of them are executed simultaneously in the CUDA kernel implementation. Two shared memory blocks are required for this purpose.

The log-FFT is applied to each q Galois field elements group. Thus, each Galois field elements group is independent. The block width is fixed to q and the block height is fixed to 256/q. Each *ty* represents a Tanner graph node connection, and each *tx* represents a Galois field element, as Figure 3.20 shows. The 1D grid contains Nodes/(256/q) blocks. The required shared memory is 2 KB per block of threads (two shared memory blocks of 256 floats per block).

For each Tanner graph node, a loop processes all node connections for processing the log-FFT of the q Galois field elements.

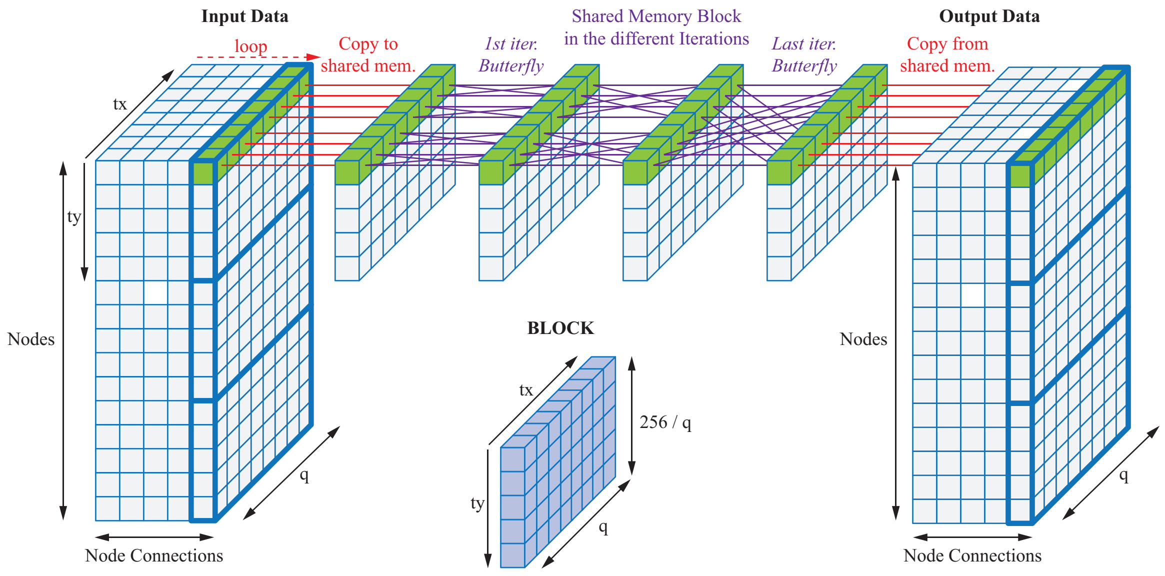

Figure 3.20: Description of the log-FFT process.

cudaLogFFTP		
	X Dimension	**Y Dimension**
Threads/Block	q	256/q
Blocks/Grid	1	BN/(256/q)
Shared Mem.	2 · 4096 KBytes	

cudaLogFFTQ		
	X Dimension	**Y Dimension**
Threads/Block	q	256/q
Blocks/Grid	1	CN/(256/q)
Shared Mem.	2 · 4096 KBytes	

Table 3.17: Block and grid settings for the **cudaLogFFTP** and **cudaLogFFTQQ** kernels

The execution of the butterfly diagram is supervised by a small *loop* of $\log_2(q)$ iterations and the control loop variable. The crossed additions and subtractions in the log-domain are performed over the shared memory. Since each continuous thread accesses a different shared memory bank during the butterfly crosses, no bank conflicts are produced. The result is transferred to the original data structure once the loop is finished.

Both kernels, **cudaLogFFTP** and **cudaLogFFTQ**, process log-magnitude and log-sign float values. In the case of **cudaLogFFTP**, only the magnitude is available as input (*P_L*), so the sign variable is initialized to one. The output variables are *FP_L* and *FP_L_s*. In the case of **cudaLogFFTQ**, *QP_L* and *QP_L_s* are the input, and only *QP_L* is generated.

The additions and subtractions in the different iterations of the loop are performed over the log-domain, as explained in [8]. Two special *__device__* kernels (**cudaBoxPlusM** and **cudaBoxPlusS**) have been implemented for calculating the signed log-domain addition and subtraction in the module and sign case respectively. Both functions are called from inside the kernel. The code of those device kernels follows exactly the C++ reference implementation [10]. The exponential and logarithm functions are used in the signed log-domain function. The fast hardware implemented *___expf()* and *___logf()* functions are used for this purpose, achieving the fastest performance in this way.

The transfer of data between the global memory and the shared memory is done in a coalesced way, since the indexing has a linear relationship with the thread number. The shared memory is accessed from within the loop. Thus, the indexing *(memory block)* $\cdot 256 + ty \cdot \mathrm{q} + tx$ performs accesses without bank conflicts.

3.2.3.9 Check Node Update (*cudaCheckNodeUpdate*)

Each thread processes the extrinsic sum of the *FP_L* and *FP_L_s* symbol values for all the node connections in each Tanner graph check node.

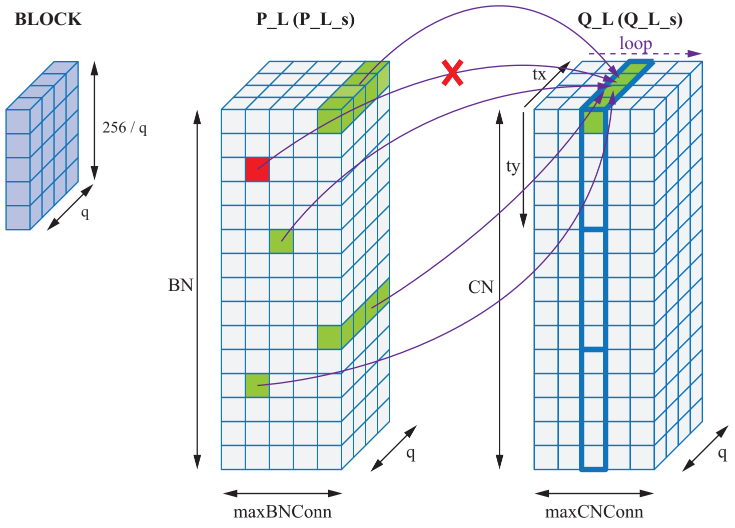

Figure 3.21: Description of the **cudaCheckNodeUpdate** kernel.

Since this kernel is processed in the frequency log-domain, magnitude and sign calculus are performed separately. For each thread, a first *loop* processes all the node connections for calculating the total sum of the magnitude values and the total product of the sign values. After this, a second *loop* processes the node connections again, for individually subtracting the current magnitude value from the total magnitude sum (extrinsic sum), and individually dividing the current sign value from the total sign product (extrinsic product). Shared memory is not required since all the memory accesses (*QP_L* writing and *FP_L* & *FP_L_s* reading through *cudaCN2BN*) are coalesced.

The block width is fixed to q and the block height is fixed to $256/q$. For a given Tanner graph check node, each thread processes a Galois field element and the extrinsic sum is achieved using the loops processing the symbols of the node connections.

	X Dimension	Y Dimension
Threads/Block	q	256/q
Blocks/Grid	1	CN/(256/q)

Table 3.18: Block and grid settings for the **cudaCheckNodeUpdate** kernel.

3.2.3.10 Galois Field Expansion (*cudaExpandGalois*)

Once the execution of the decoding process finishes, the binary output is generated from the hard decision vector. The CUDA code is based on the C++ reference code [10].

	X Dimension	Y Dimension
Threads/Block	p	256/q
Blocks/Grid	1	(BN-CN)/(256/q)

Table 3.19: Block and grid settings for the **cudaCheckNodeUpdate** kernel.

The block and grid settings are described in Table 3.19, where p is the number of bits of the Galois field elements (q = 2^p). Each ty represents a Tanner graph node, and each tx represents one bit of a Galois field element. The memory accesses are continuous since the indexing has a linear relationship with the thread index.

Chapter 4

Results

The goal of this thesis is to accelerate the LDPC decoding process of the C++ reference implementation [10] using the CUDA technology. So, the study of the GPU/CPU speedup in different scenarios is the main task in this chapter.

The CPU implementation runs on a software simulation system where a group of input frames is decoded using a determined number of decoding iterations. Each simulation is executed for a given E_b/N_0 value.

The CUDA algorithm implementation is executed under the same settings as the CPU reference implementation, and the speedup is measured while increasing code size and the number of iterations. From the obtained values, it is possible to analyse the evolution of the performance of the CUDA implementation. The results are presented in tables and graphs for a better understanding of the obtained numbers.

Additional kernel post-implementation analysis are performed using the *NVIDIA Profiler* tool in this chapter. This tool provides information about how the kernels take advantage of the hardware resources and how they are behaving during the execution.

Before starting to explore the results, it is important to keep in mind that the obtained results are related to the hardware platforms used in the simulations. Thus, the results change if using different CPU's and/or GPU's. The considered hardware in this thesis is listed as follows (for more information, please refer to annexes):

CPU: Intel Core 2 Duo Q6600 @ 2.40 GHz / 2 GB RAM

GPU: NVIDIA GeForce GTX 275 / 786 MB RAM / CUDA Architecture 1.3

4.1 Binary LDPC Decoder

Table 4.1 shows the binary LDPC codes used in the simulations. All the simulations are performed for $E_b/N_0 = -1$ dB.

Id. Name	*BN*	*CN*	*Regularity*	*Sort*	*maxBNConn*	*maxCNConn*
WiMAX_2304	2304	1152	1/2	Irregular	6	7
Regular_8000	8000	4000	1/2	Regular	3	6
Regular_16000	16000	8000	1/2	Regular	3	6
DVB-S2_64800	64800	32400	1/2	Irregular	7	8

Table 4.1: Binary LDPC Codes used for the simulations.

4.1.1 Kernel Execution Time

The *NVIDIA Visual Profiler* tool provides interesting visual information about the execution time of each kernel.

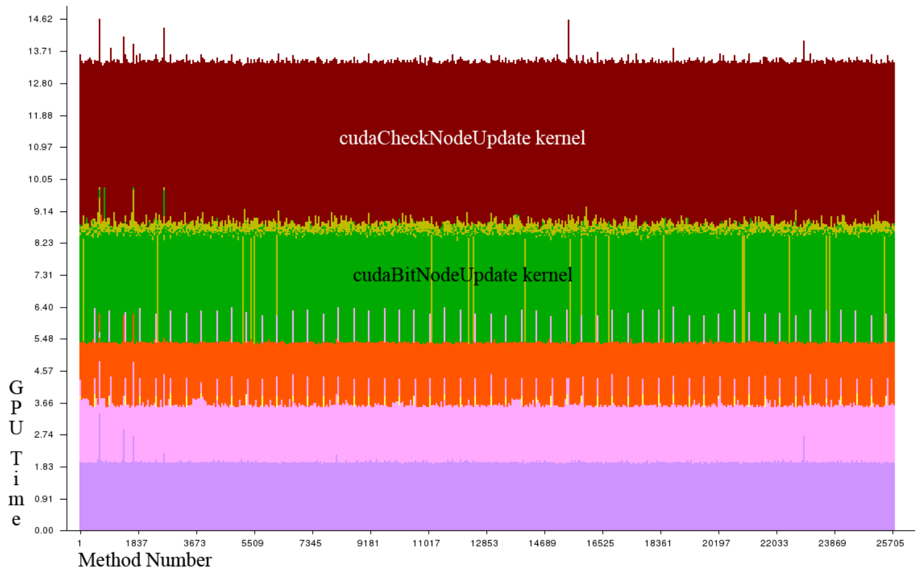

Figure 4.1: Kernel execution times for a binary DVB-S2 LDPC code.

Figure 4.1 shows the time execution of the kernels during a simulation of the DVB-S2_64800 LDPC code. The x axis is related to different execution instances, and the y axis is related to the used GPU time. The graph indicates that the *cudaCheckNode-Update* kernel (dark red color) has the highest computational cost. The random memory accesses plus the hardware implemented functions *__exp(), __log()* and *__fdividef()* are the cause of this execution time. When the *tanh()* and *atanh()* functions were implemented using standard functions or look-up tables, the execution times of this function were even higher. For the look-up table implementation, discrete *tanh* and *atanh* values were stored in the constant memory or in the texture memory. But since the look-up tables (more or equal than 2048 elements per table) were accessed in a random way and their elements had no locality, the performance was not as good as expected.

4.1.2 Summary CUDA Optimizations

The *NVIDIA Visual Profiler* tool also provides information about how the hardware resources are being used. This tool is helpful during the debug phase for achieving better speed performance. It executes the implemented software and obtains information from internal hardware registers. Table 4.2 shows the results obtained in the binary implementation for the DVB-S2_32400 code case.

The analysis of the table contents is performed as follows.

GPU Time: The column shows that the *cudaCheckNodeUpdate* kernel is the slowest one, because of the random accesses plus the exponential, logarithm and division hardware functions.

Occupancy: The *cudaCheckNodeUpdate* and *cudaBitNodeUpdate* kernels only have an occupancy of 50% because of the amount of used shared memory. Although the 256 threads per block configuration leads to the execution of eight blocks per streaming multiprocessor, the block size of the shared memory limits this amount to only four. Thus, the occupancy is reduced by a factor of two.

Dynamic Shared Memory: Column provides information about the amount of shared memory used by each kernel. If the amount of used shared memory exceeds the maximum available, it causes not to have full occupancy in the SM.

Used Registers: The number of registers has to be less or equal to 16 for achieving the maximum occupancy (calculated with the *CUDA Occuancy Calculator*). The column shows that this is accomplished. Thus, the access to the internal kernel variables is done in an optimal way since all of them are implemented with the use of registers.

GLD and GST Coalesced: Global Load and Global Store Coalesced are the complete names of these table entries. They provide the number of coalesced accesses to the memory. All the non-coalesced accesses are produced by the random memory accesses.

Divergent branches: The number of divergent branches in function of the total number is low. E.g. the *cudaCheckNodeUpdate* has 2 divergent branches of a total of 131. The CUDA code performs better with a minimum number of divergent branches. Thus, this factor does not considerably degrade the performance.

Warp Serialize: A bank conflict is produced if the shared memory is not accessed at different memory banks by continuous thread indexes. If this is the case, the hardware serializes the access to the shared memory, slowing down the performance. The column shows that all the kernels using shared memory perform accesses without bank conflicts. Thus, the shared memory is accessed almost as fast as the hardware registers.

GLD and GST XXb: These columns indicate the number of 32, 64 or 128 bit accesses to the global memory. For a better performance, only one kind of these accesses should be performed. Here, it is not easy to see a good performance. A deeper

Kernel	*GPU Time*	*Occupancy*	*Dynamic Shared Mem.*	*Used Registers*	*GLD Coal.*	*GST Coal.*	*Divergent Branches*	*Warp Serialize*
cudaInitializeQ	3.84	1	0	4	16	672	0/64	0
cudaCheckNodeUpdate	13.34	0.5	1008	12	653	258	2/131	0
cudaBitNodeUpdate	8.64	0.5	1008	10	768	524	14/86	0
cudaCheckEquations	8.51	1	0	8	0	0	0	0/0
cudaCheckEquationsSatisfied	5.22	1	2048	4	0	0	0/0	0

Kernel	*GLD 32b*	*GLD 64b*	*GLD 128b*	*GST 32b*	*GST 64b*	*GST 128b*	*Global Mem. Throughput*
cudaInitializeQ	0	16	0	48	64	40	27.56
cudaCheckNodeUpdate	571	35	47	47	13	14	22.36
cudaBitNodeUpdate	600	86	82	40	45	33	44.11
cudaCheckEquations	0	0	0	0	0	0	26.92
cudaCheckEquationsSatisfied	0	0	0	0	0	0	0.80

Table 4.2: Binary LDPC Codes used for the simulations.

analysis of the code leads to know that all the non-random accesses are performed for only one size and in a coalesced way. The random accesses are present in three kernels and these are performed using different access sizes (*cudaBitNodeUpdate* and *cudaCheckNodeUpdate*).

4.1.3 GPU/CPU Speedup versus Number of Iterations

A manifold of simulations have been performed with a different number of iterations for each code. The results in terms of speedup are summarized in Table 4.1. All the simulations run with the same settings for both implementations, GPU and CPU. The speedup is calculated as (GPU time)/(CPU time). The results for each of four binary LDPC codes, can be shown as follows.

	GPU/CPU Speedup			
Iterations	WiMAX_2304	Regular_8000	Regular_16000	DVB-S2_64800
10	5.5	6.1	7.0	13.6
20	8.1	9.6	10.9	20.9
40	10.5	15.4	18.5	37.0
60	12.2	19.4	24.2	43.1
100	13.6	24.5	32.6	55.1
200	15.2	31.4	42.5	76.6
300	15.9	33.0	49.5	86.8

Table 4.3: Simulation results of binary LDPC codes for different number of iterations.

At first sight, it is clear that for a given number of iterations the performance improves with an increasing code size. Also, for a given LDPC code, the performance is better with an increasing number of iterations. From Table 4.3 some graphs are drawn for a better observation of the speedup in function of the iterations and the LDPC code size.

Figure 4.2 shows that the speedup has an asymptotic behaviour with on increasing number of iterations. The figure also shows that the larger is the code size, the better is the performance of the CUDA code. The four graphs tend to a different upper limit value. While the WiMAX code has reached this limit before the 300 iterations, the DVB-S2 seems to be able to improve the speed-up for a number of iterations greater than 300. The reason for this behaviour is that for a larger code size, the parallel execution performs better.

4.1.4 GPU/CPU Speedup versus Code Size

Figure 4.3 represents the Table 4.3 data in a different way. Now, the x axis represents the code size and the different plotted lines represent different number of iterations.

As expected, the greater the code size is (and the number of iterations), the higher the speedup value is. The graph shows that with an increasing number of iterations, the

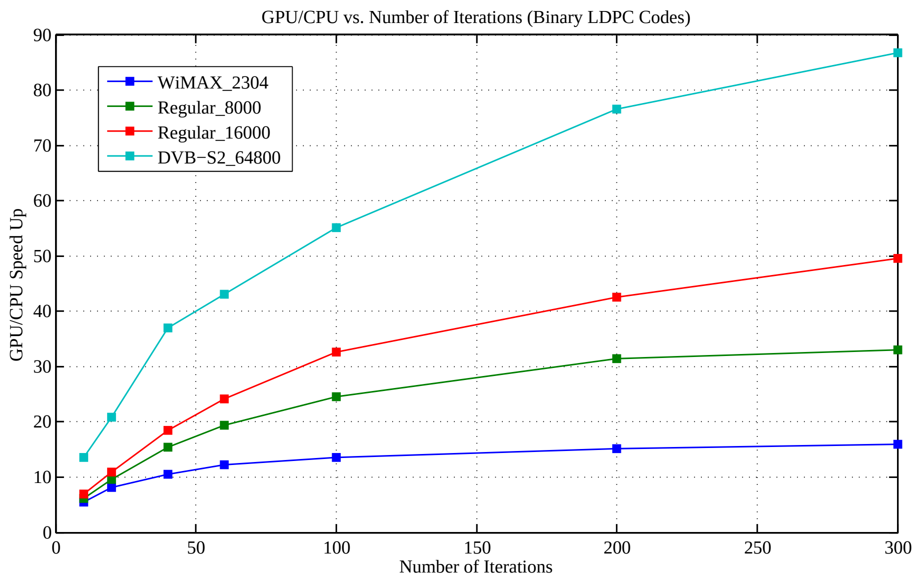

Figure 4.2: GPU/CPU speedup vs. number of iterations (binary LDPC codes)

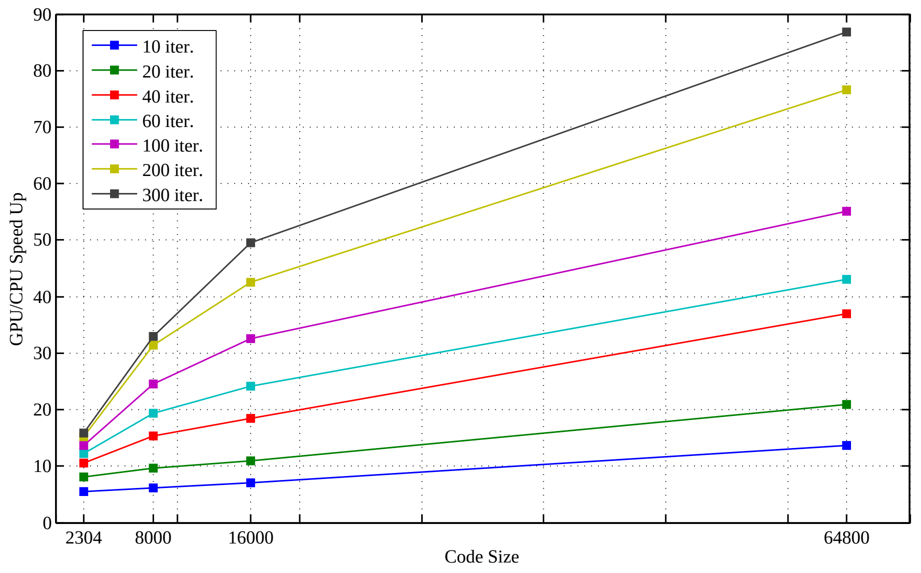

Figure 4.3: GPU/CPU speedup vs. code size for binary LDPC codes

speedup also increases its value. Although there is only a small number of measurements, the behaviour is expected to be asymptotic.

4.1.5 BER Comparison between CPU and GPU Implementations

Comparing the BER versus E_b/N_0 graphs for both implementation, it is possible to check if the CUDA decoding algorithm is performing as the C++ reference one. Figure 4.4 shows this graph for a range of E_b/N_0 from 0 dB to 0.8 dB.

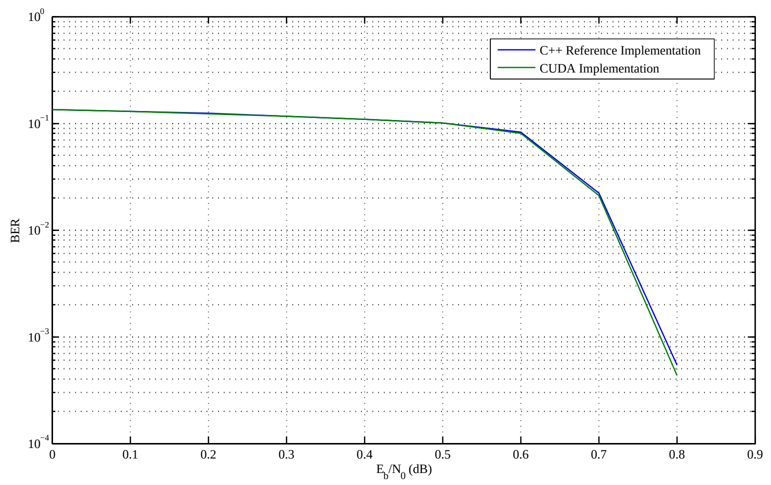

Figure 4.4: BER vs. E_b/N_0 (DVB-S2_64800 code and 60 iterations).

The figure shows that the CUDA implementation (green colour) performs a little bit better than the C++ implementation (blue colour), despite the fact that the CUDA version uses single precision and the C++ implementation uses double precision. The reason for this is the use of look-up tables in the C++ version for implementing the *tanh* and *atanh* functions.

4.2 Non-binary LDPC Decoder

For the case of binary LDPC codes, the code size is a good reference for comparing different simulations. But for the case of non-binary LDPC codes the situation is different, since the Galois field dimension changes the number of the Tanner graph bit nodes in function of the input bits number and the code rate (see Table 4.4). Thus, the number of input bits is going to be the reference for comparing the sizes of the non-binary LDPC codes.

Table 4.4 shows the different non-binary LDPC codes used for obtaining the results in this section. As summary, LDPC GF(2) (binary), GF(8), GF(16), GF(32), GF(64) and GF(256) codes are used for the simulations. For all of them, different input bit sizes (504, 1152, 5040 and 32400) are selected. The fact of having groups of q Galois field elements in continuous memory positions, leads to an interesting performance boost, as can be verified later in this section.

4.2.1 Kernel Execution Time

The *CUDA Visual Profiler* tool provides visual information about the execution time for each kernel during a simulation. Figure 4.5 shows the kernel execution times for the GF64_5040 code.

Id. Name	q	*Input Bits*	*BN*	*CN*	*Rate*	*Sort*	*maxBNConn*	*maxCNConn*
GF2_504	2	504	1008	504	1/2	Irregular	15	9
GF2_1152	2	1152	2304	1152	1/2	Irregular	15	9
GF2_5040	2	5040	10080	5040	1/2	Irregular	15	9
GF2_32400	2	32400	64800	32400	1/2	Irregular	15	9
GF8_504	8	504	336	168	1/2	Irregular	5	6
GF8_1152	8	1152	768	384	1/2	Irregular	5	6
GF8_5040	8	5040	3360	1680	1/2	Irregular	5	6
GF8_32400	8	32400	21600	10800	1/2	Irregular	5	6
GF16_504	16	504	252	126	1/2	Irregular	5	6
GF16_1152	16	1152	576	288	1/2	Irregular	5	6
GF16_5040	16	5040	2520	1260	1/2	Irregular	5	6
GF16_32400	16	32400	16200	8100	1/2	Irregular	5	6
GF32_505	32	505	202	101	1/2	Irregular	4	5
GF32_1155	32	1115	462	231	1/2	Irregular	4	5
GF32_5040	32	5040	2016	1008	1/2	Irregular	4	5
GF32_32400	32	32400	12960	6480	1/2	Irregular	4	5
GF64_504	64	504	168	84	1/2	Irregular	4	5
GF64_1152	64	1152	384	192	1/2	Irregular	4	5
GF64_5040	64	5040	1680	840	1/2	Irregular	4	5
GF64_32400	64	32400	10800	5400	1/2	Irregular	4	5
GF256_505	256	505	126	63	1/2	Irregular	3	5
GF256_1155	256	462	288	144	1/2	Irregular	3	5
GF256_5040	256	5040	1260	630	1/2	Irregular	3	5

Table 4.4: Non-binary LDPC Codes used for the simulations.

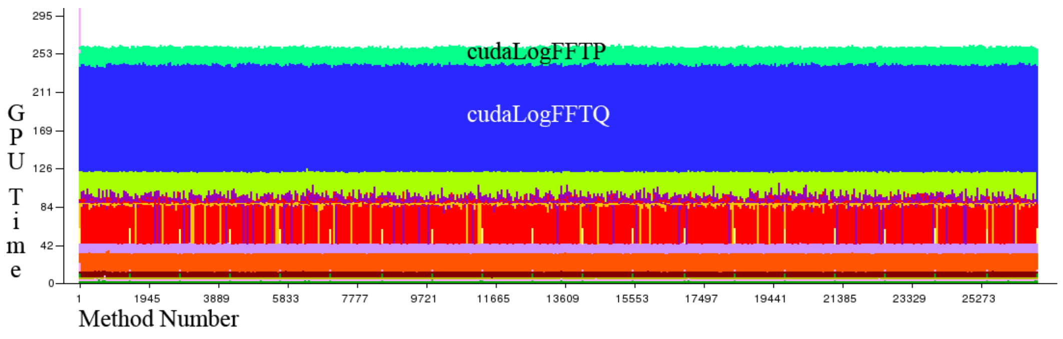

Figure 4.5: Kernel execution times for a GF64_5040 code (*CUDA Visual Profiler* capture).

As the *CUDA Visual Profiler* capture shows, the *cudaLogFFTP* (green colour) and *cudaLogFFTQ* (blue color) have the highest computational cost. The reason of these high values is that these kernels execute the __*expf()* and __*logf()* hardware functions during the log-FFT processing, and these functions have a high computational cost.

4.2.2 CUDA Optimization Summary

CUDA Visual Profiler provides more detailed information about the CUDA non-binary LDPC decoder implementation. Some decoding simulations are performed from the *Profiler* environment, and an important amount of information is obtained from the internal registers of the device. Table 4.5 gathers the most relevant obtained data, and it is analysed as follows.

GPU Time: The *cudaLogFFTP* and *cudaLogFFTQ* are the slowest kernels, as it has been shown before (Figure 4.5). The utilization of the *___expf()* and *___log()* hardware functions is the reason of these execution times.

Occupancy: The occupancy is 100% and, since the number of threads per block is 256, each SM processes 4 blocks and the shared memory requirements are not a constraint.

Dynamic Shared Memory: Keeping in mind that each SM executes 4 concurrent blocks, the required amount of shared memory for each SM is 4·2048 KBytes = 8192 KBytes, less than the maximum available shared memory per block.

Used Registers: The maximum number of registers usage is 16 per thread (value calculated using *CUDA Occupancy Calculator*). Since all the kernels use less or equal than 16 registers, all of them use the hardware on chip registers, being accessed at the maximum speed.

GLD and GST Coalesced: This column provides information about the number of coalesced accesses to the memory.

Divergent branches: Because of the algorithm implementation, it is not possible to avoid the branches in the thread execution. Even so, the number of divergent branches is small compared with the total branch number. This small percentage of divergent branches does not seriously affect the total performance.

Warp Serialize: Only one of the six kernels using shared memory has bank conflicts. This is caused by the way the algorithm indexes the shared memory. The performance is not seriously affected.

GLD and GST XXb: The indexing of the variables allocated in the global memory causes that the reading memory accesses avoid 128 bit accesses, and some of the kernels only perform 64 bit accesses. In the case of writing, almost 100% of the accesses are 64 bit, boosting the memory access performance.

Global Memory Throughput: The *cudaInLUpdate*, *cudaHard*, *cudaBitNodeUpdate*, *cudaPermuteP*, *cudaCheckNodeUpdate* and *cudaPermuteQ* are main kernels and all of them have a good memory throughput. Only the log-FFT kernels could have better memory performance, but a deeper study of the kernel implementation is required. The throughput in the rest of the kernels is not so relevant since they are executed only once or they only write to a few memory positions.

4.2.3 GPU/CPU Speedup versus Number of Iterations

Many simulations with different number of iterations have been performed for all the codes in Table 4.6 with input bit size 5040. The reason of this size is that this is the biggest size available for the non-binary LDPC codes. The results are obtained for an

Kernel	*GPU Time*	*Occupancy*	*Dynamic Shared Mem.*	*Used Registers*	*GLD Coal.*	*GST Coal.*	*Divergent Branches*	*Warp Serialize*
cudaInLInitialize	60.8	1	0	7	6720	12096	560/3472	0
cudaInLUpdate	32.13	1	0	8	4152	2688	0/688	0
cudaHardDecision	42.27	1	0	7	0	0	0/0	0
cudaCheckNodeEquations	12.19	1	0	9	1137	64	0/0	0
cudaCheckNodeEquationsSatisfied	5.41	1	2048	4	0	0	0/0	0
cudaBitNodeUpdate	122.53	1	2048	13	9708	5616	702/12544	0
cudaPermuteP	92.51	1	1024	16	6820	10752	346/4336	766
cudaLogFFTP	442.56	1	2048	10	2688	21504	5110/47731	0
cudaCheckNodeUpdate	87.71	1	0	11	11600	11264	0/740	0
cudaLogFFTQ	280.67	1	2048	10	3360	6720	2263/26454	0
cudaPermuteQ	82.4	1	1024	7	5848	6720	248/2964	0

Kernel	*GLD 32b*	*GLD 64b*	*GLD 128b*	*GST 32b*	*GST 64b*	*GST 128b*	*Global Memory Throughput*
cudaInLInitialize	4032	2688	0	672	2688	0	81.35
cudaInLUpdate	672	3480	0	0	672	0	88.84
cudaHardDecision	0	0	0	0	0	0	90.73
cudaCheckNodeEquations	579	183	375	0	16	0	16.84
cudaCheckNodeEquationsSatisfied	0	0	0	0	0	0	0.71
cudaBitNodeUpdate	5088	4620	0	0	1404	0	43.64
cudaPermuteP	2688	4132	0	0	2688	0	55.71
cudaLogFFTP	0	2688	0	0	5376	0	11.51
cudaCheckNodeUpdate	1744	9856	0	0	2816	0	98.05
cudaLogFFTQ	0	3360	0	0	1680	0	11.48
cudaPermuteQ	1720	1736	2392	0	1680	0	69.71

Table 4.5: Binary LDPC Codes used for the simulations.

$E_b/N_0 = -1$ dB. Table 4.6 shows the speedup results of the simulations in terms of (GPU time)/(CPU time).

	GPU/CPU Speedup					
Iter.	*GF2_5040*	*GF8_5040*	*GF16_5040*	*GF32_5040*	*GF64_5040*	*GF256_5040*
10	35.5	122.7	192.9	280.5	403.0	620.4
20	43.0	166.4	243.4	320.5	483.5	670.2
40	49.4	195.2	278.4	369.8	537.0	698.3
60	51.8	220.4	307.4	396.6	557.6	707.5
100	54.1	234.4	329.6	407.7	578.8	714.2
200	55.4	249.8	342.2	425.8	592.2	724.8
300	56.3	253.1	347.3	428.1	599.3	728.0

Table 4.6: Simulation results for non-binary LDPC codes for different number of iterations.

The results for the GF(2) code are similar to the results obtained with the binary LDPC codes. Table 4.6 shows that as the Galois field dimension grows, the much better is the performance. For Galois field dimension greater than 2, the speedup values are over 100. The case GF(256) has the highest scores, starting with a minimum of 620.4 and reaching a speedup of 728.0 for 300 iterations. The behaviour of the speedup is plotted and analysed in Figure 4.6 and Figure 4.7.

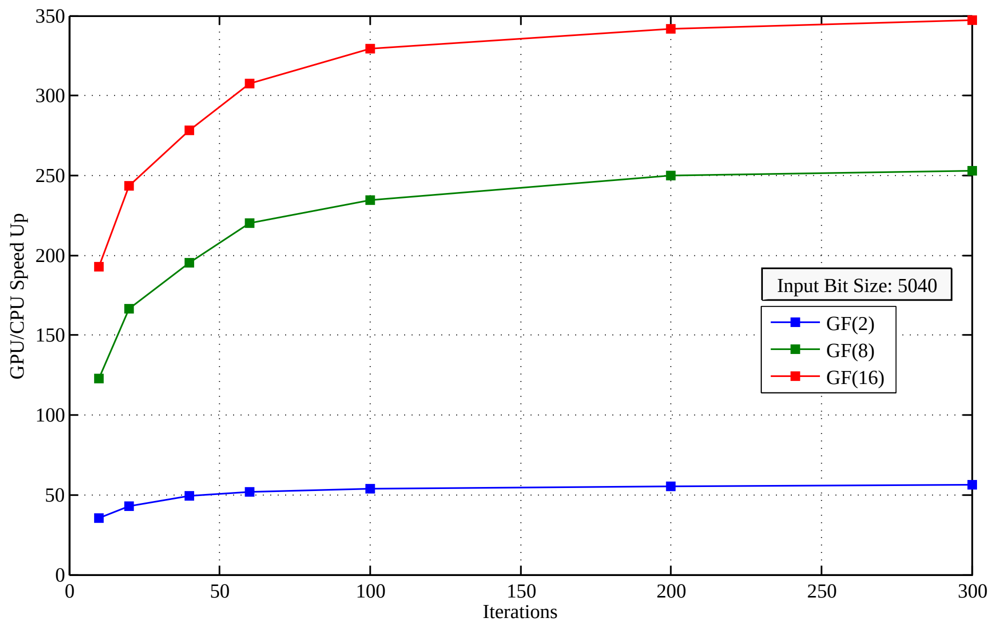

Figure 4.6: GPU/CPU speedup vs. number of iterations (GF2_5040, GF8_5040 and GF16_5040)

Figure 4.6 shows the speedup versus the number of iterations for the GF2_5040, GF8_5040 and GF16_5040 codes. The speedup has a clear asymptotic behaviour. The change of the speedup value for GF2_5040 is really small as the number of iterations grows. For greater Galois field dimension, the variation is higher.

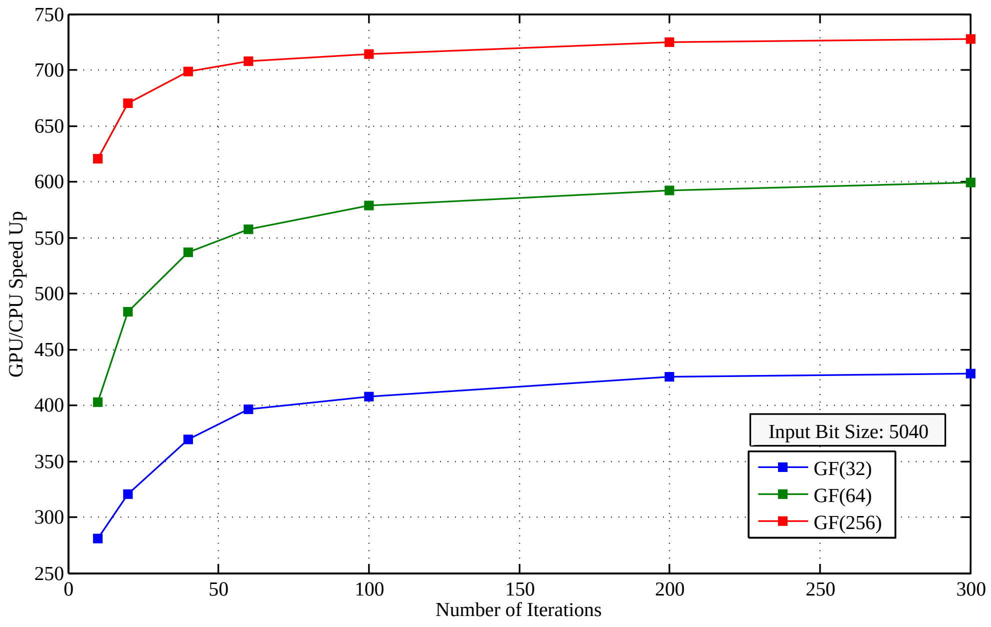

Figure 4.7: GPU/CPU speedup vs. number of iterations (non-binary LDPC codes

Results for GF32_5040, GF64_5040 and GF256_5040 codes are shown in Figure 4.7. The speedup is even larger there seems to be a the maximum limit value at 300 iterations. The distance between speedup values at 300 iterations is not being reduced with an increasing Galois field dimension value. Thus, it is possible to have better performance for larger Galois field dimensions.

4.2.4 GPU/CPU Speedup versus Galois Field Dimension

The simulation results of different Galois field dimension codes for the different input bit sizes (504, 1152, 5040 and 32400) and for 60 iterations, are summarized in Table 4.7.

	GPU/CPU Speedup			
GF Dimension	*Input Size 504*	*Input Size 1152*	*Input Size 5040*	*Input Size 32400*
2	9.8	20.5	51.8	55.6
8	45.0	90.9	220.4	278.9
16	68.8	139.1	307.6	423.3
32	133.2	253.6	396.6	534.2
64	215.7	360.0	557.6	654.3
256	292.5	548.6	707.5	-

Table 4.7: Simulation results for different LDPC GF(q) codes and input bit sizes (60 iterations).

Confronting speedup versus Galois Field dimension for the different input bit sizes, it is possible to see another asymptotic behaviour, as Figure 4.8 and Figure 4.9 show.

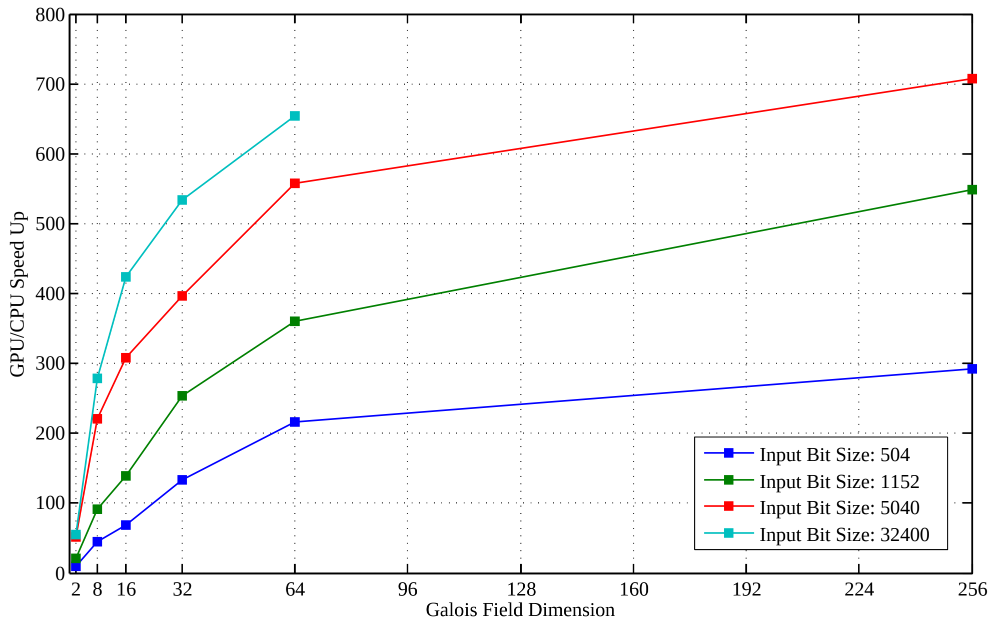

Figure 4.8: GPU/CPU speedup vs. Galois field dimension for different input bit sizes.

For a given input bit size, the larger the Galois Field dimension is, the higher the speedup achieved by the CUDA implementation is(Figure 4.8). At the same time, for a given Galois Field dimension, if the input bit size increases, then the performance also does. Figure 4.9 contains the same information, but showing the results in a different presentation.

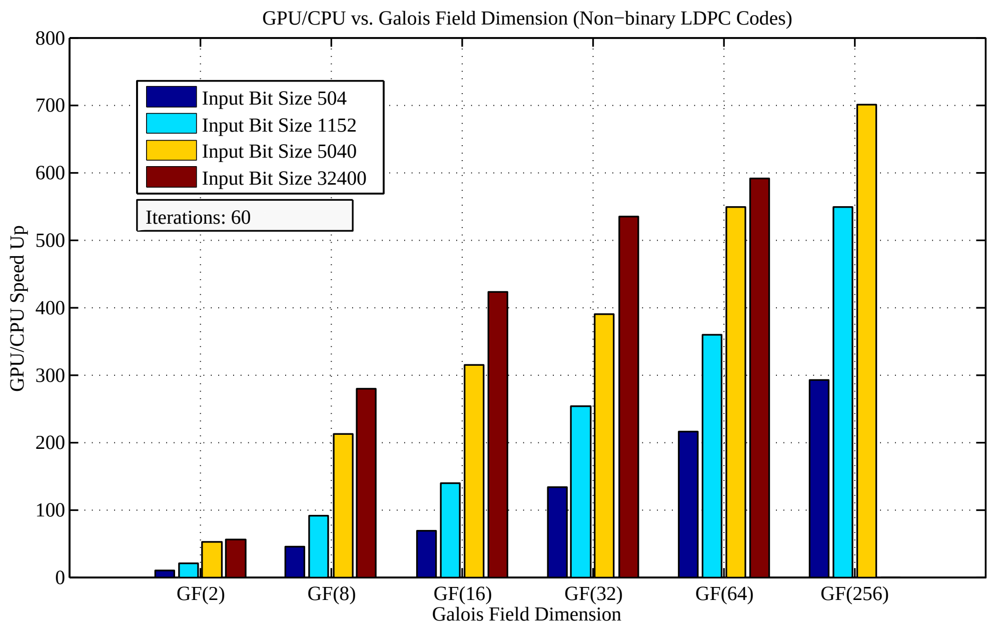

Figure 4.9: GPU/CPU speedup vs. Galois field dimension for different input bit sizes.

4.2.5 BER Comparison Between CPU and GPU version

A very interesting curve for the LPDC decoder is the BER versus E_b/N_0. The comparison between the BER curve of the C++ reference implementation and the one from the CUDA implementation, provides an idea about the algorithm behaviour and the accuracy. Since the CUDA implementation is based on the C++ reference code, it is expected to have the same curve shape. The most important difference between both implementations is that the C++ reference code uses double precision (*double* variables), and the CUDA implementation uses simple precision data (*float* variables). Figure 4.10 shows the curves for both implementations, for the GF64_5040 code and different number of iterations. The E_b/N_0 values are in the range from 0 to 2.1, with 0.1 steps.

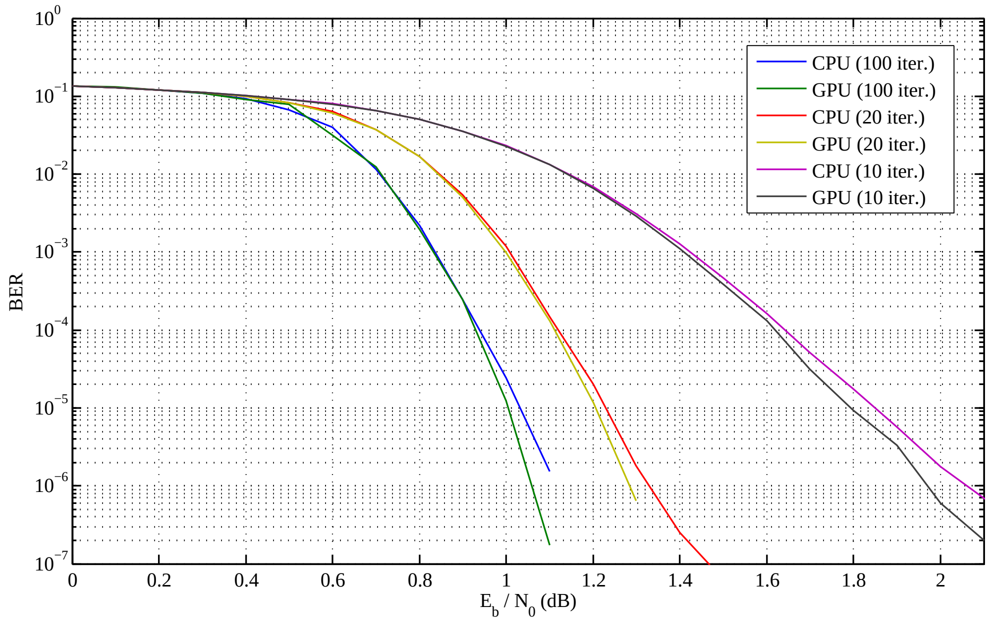

Figure 4.10: BER vs. E_b/N_0 (GF64_5040 code).

As expected, the curves have the same shape. The CUDA implementation has the same behaviour than the C++ reference implementation for low E_b/N_0 values and achieves better BER results with an increasing E_b/N_0 value.

Chapter 5

Summary and Conclusions

Low-density parity check codes are linear block codes which have a huge sparse parity-check matrix with a low number of non-zero entries. These codes provide a performance close to the Shannon Limits on a large collection of transmission channels. LDPC codes where introduced by Robert G. Gallager in 1960 [16], but it was impractical to implement them due to their high computational cost. LDPC codes were rediscovered in the 90's by David J. C. MacKay and Radford M. Neal [3].

LDPC codes can be represented using a Tanner graph. A Tanner Graph is a bipartite graph which represents the parity-check matrix of an error correcting code. This bipartite graph has two classes of nodes, variable nodes and check nodes, and no edge connects two nodes from the same class. A variable node connects to a check node if the corresponding parity-check matrix element is a non-zero value.

Iterative algorithms are used for decoding LDPC codes. These iterative algorithms are based on messages passing algorithms and perform a probabilistic decoding, where the messages passed are in terms of logarithmic likelihood (L-values). Belief propagation is an example of algorithm for decoding binary LPDC codes, and the log-domain Fourier transform is the case for non-binary LDPC codes.

CUDA (Computer Unified Device Architecture) is a hardware architecture abstraction of the modern GPUs present in the NVIDIA graphic cards. This architecture can be used for massive parallel processing of many scientific or high computational tasks. CUDA can be programmed using the standard programming C/C++ language with some specific CUDA extensions.

In this thesis, a CUDA parallel implementation of the belief propagation and the log-domain Fourier transform algorithms has been performed. Both implementations are based on a previous C++ reference implementation and perform LDPC decoding using an existing software simulation platform. The objective of the thesis is to speed up the LDPC decoding process over the CPU implementation, taking advantage of the parallel execution capabilities of the CUDA architecture. An Intel Core 2 Quad Q6600 @ 2.4 GHz has been used as CPU platform, and a NVIDIA GeForce GTX 275 has been used as GPU platform for the CUDA implementation.

A CUDA LDPC decoding implementations has been developed in this thesis. For both of the binary and non-binary LDPC decoder implementations, a first analysis of the algorithm flowchart has been performed and the data structures used during the implementations have been studied. Finally, each kernel has been described with the performance options applied to it. During the implementation, some difficulties have been encountered. In the binary case, the random accesses have forced the use of the global memory for allocating the message passing L-value matrices. Thus, a special attention has been paid to the indexing of these matrices for achieving the maximum memory performance. The use of shared memory in some kernel has allowed a faster kernel execution when acting as a fast temporal memory. Finally, the *tanh* and *atanh* functions have been implemented with fast hardware functions, considerably reducing the execution time of the check node update kernel.

In the non-binary case, the random accesses have been present again, but the Galois field dimension has allowed to access q continuous data for each element of the parity-check matrix, increasing speed of the global memory accesses. The shared memory has allowed to execute some kernels faster, since it has been acting as temporary memory. The log-domain Fourier transform has been completely parallelized using the butterfly structure, and the log-domain addition function has been implemented with the fast hardware available functions.

Finally, the results of the different simulations have been compared. In the binary case, four LDPC codes (the WiMAX code of size 2304, a regular code of size 8000, a regular code of size 16000 and the DVB-S2 code of size 64800) have been used in the simulations, and the speedup has been measured for different amounts of iterations and code sizes. In both cases, an asymptotic behaviour of the speedup has been observed, with a better performance for larger codes. Although the speedup values are conditioned by the utilized hardware, the results have shown a maximum speedup value of 15.9 for the WiMAX code of size 2304, 33.0 for the regular code of size 8000, 49.5 for the regular code of size 16000, and 86.8 for the DVB-S2 code of size 64800, all of them performed for a 300 decoding iterations simulation. As expected, the BER curve of the CUDA implementation follows exactly the shape of the serial CPU implementation.

In the non-binary case, several LDPC codes with different Galois field dimensions ($q = 2,\ 8,\ 16,\ 32,\ 64$ and 256) and different input bit sizes (504, 1152, 5040 and 32400) have been compared. The speedup has been measured for different Galois field dimensions, different input bit sizes and different number of iterations, observing an asymptotic behaviour in all of them. The performance is better for larger values of Galois field dimension, input bit sizes and number of iterations. For an input bit size of 5040 and 300 decoding iterations, the measured speedup has reached a maximum value of 56.3 for a GF(2) code, 253.1 for a GF(8) code, 347.3 for a GF(16) code, 428.1 for a GF(32) code, 599.3 for a GF(64) code and 728.0 for a GF(256) code. The BER curve of the CUDA implementation ($q = 256$, input bit size of 5040) is coincident with the BER curve of the serial CPU implementation.

Since the graphic card model used in this thesis does not belong to the latest CUDA generation, higher performance can be expected if a newest model (*Fermi* architecture with compute capability 2.0 [14]) is used. Also, using the more powerful branch of the CUDA devices (*Tesla*), can lead to further performance gains.

Chapter 6

Appendix A
GF(2) CUDA Kernels Source Code

```
__global__ void cudaInitializeQ (float *vector_in, int *vector_in_upd,
float *q, int tileWidthBN, int BN)
{
  unsigned int tx, bx, index, k;
  float temp;
  tx=threadIdx.x; bx=blockIdx.x;
  index=bx*blockDim.x+tx;

  if (index<BN) {
    temp=vector_in[index];
    vector_in_upd[index]=(temp<0.0f)?(1):(0);
    for (k=0; k<tileWidthBN; k++)
      q[k*BN+index]=temp;
  }
}

__global__ void cudaCheckEquations (int *v_in, int *v_temp,
int *cudaCheckNodeConnections, int tileWidthCN, int CN, int *cudaCNConn)
{
  unsigned int tx, bx, k, kmax, index, temp, data;
  tx=threadIdx.x; bx=blockIdx.x;
  index=bx*blockDim.x+tx;

  if (index<CN) {
    kmax=cudaCNConn[index];
    temp=0;
    for (k=0; k<kmax; k++) {
      data=v_in[cudaCheckNodeConnections[k*CN+index]];
      temp=(bool)temp^(bool)data;
    }
    v_temp[index]=temp;
  }
}

__global__ void cudaCheckEquationsSatisfied (int *v_in, int *v_out,
int CN)
{
  extern __shared__ int sm[];
  unsigned int tx, index, stride;
  tx=threadIdx.x;
  index=blockIdx.x*blockDim.x+threadIdx.x;

  sm[tx]=(index<CN)?(v_in[index]):(0);
  __syncthreads();
  for (stride=blockDim.x>>1; stride>0; stride>>= 1) {
    if (tx<stride)
      sm[tx]+=sm[tx+stride];
    __syncthreads();
  }
  if (tx==0)
    v_out[blockIdx.x]=sm[0];
}

__global__ void cudaCheckNodeUpdate (float *r, float *q, int *cudaCN2BN,
int tileWidthCN, int CN, int *cudaCNConn)
{
  extern __shared__ float qs[];
  unsigned int tx, bx, index, k, kmax;
  float temp, temp2, temp3, sig, prod;
  tx=threadIdx.x; bx=blockIdx.x;
  index=bx*blockDim.x+tx;

  if (index<CN) {
    prod=1.0;
    kmax=cudaCNConn[index];
    for (k=0; k<kmax; k++) {
      temp=q[ cudaCN2BN[k*CN+index] ];
      sig=(temp<0.0f)?(-1.0f):(1.0f);
      if (sig*temp>40.0f) {
        temp=sig;
      } else {
        temp2=__expf(temp);
        temp=__fdividef(temp2-1.0f,temp2+1.0f);
      }
      prod*=temp;
      qs[k*blockDim.x+tx]=temp;
    }
    __syncthreads();
    for (k=0; k<kmax; k++) {
      temp=__fdividef(prod, qs[k*blockDim.x+tx]);
      sig=(temp<0.0f)?(-1.0f):(1.0f);
      temp2=sig*temp;
      temp3=(temp2<1.0f-BP_EPSILON)?
```

```
          __fdividef(1.0f+temp2,1.0f-temp2):BP_INFINITY;
        temp=sig*__logf(temp3);
        r[k*CN+index]=temp;
      }
    }
}

__global__ void cudaBitNodeUpdate (float *vector_in, int *vector_in_upd,
float *r, float *q, int *cudaBN2CN, int tileWidthBN, int BN,
int *cudaBNConn)
{
  extern __shared__ float rs[];
  unsigned int tx, bx, index, k, kmax;
  float sum, temp;
  tx=threadIdx.x; bx=blockIdx.x;
  index=bx*blockDim.x+tx;

  if (index<BN) {
    sum=vector_in[index];
    kmax=cudaBNConn[index];
    for (k=0; k<kmax; ++k) {
      temp=r[cudaBN2CN[k*BN+index]];
      sum+=temp;
      rs[k*blockDim.x+tx]=temp;
    }
    __syncthreads();
    vector_in_upd[index]=(sum<0.0f)?(1):(0);
    for (k=0; k<kmax; ++k)
      q[k*BN+index]=sum-rs[k*blockDim.x+tx];
  }
}
```

Chapter 7

Appendix B GF(q) CUDA Kernels Source Code

```
__global__ void cudaInLInitialize(float *v_in, float *in_L, int BN, int p,
int q)
{
  unsigned int tx, ty, bx, index;
  tx=threadIdx.x;
  ty=threadIdx.y;
  bx=blockIdx.x;

  index=bx*blockDim.y+ty;
  if (index<BN) {
    for (int k=0; k<p; k++) {
      if ((tx>>(p-k-1))&0x1==1) {
        in_L[index*q+tx]-=v_in[index*p+k];
      }
    }
  }
}

__global__ void cudaHardDecision (float *in_L, int *hard, int BN, int q)
{
  unsigned int tx, bx, index;
  int maxDec;
  float maxProb;
  tx=threadIdx.x;
  bx=blockIdx.x;
  index=bx*blockDim.x+tx;

  if (index<BN) {
    maxDec=0;
    maxProb=in_L[index*q];
    for (int i=1; i<q; i++) {
      if (in_L[index*q+i]>maxProb) {
        maxProb=in_L[index*q+i];
        maxDec=i;
      }
    }
    hard[index]=tex1Dfetch(gf_dec2exp_tex, maxDec);
  }
}

__global__ void cudaCheckNodeEquationsSatisfied (int *v_in, int *v_out,
int CN)
{
  extern __shared__ int shared[];
  unsigned int tx, index, stride;
  tx=threadIdx.x;
  index=blockIdx.x*blockDim.x+threadIdx.x;

  shared[tx]=(index<CN) ? (v_in[index]) : (0);
  __syncthreads();
  for (stride=blockDim.x>>1; stride>0; stride>>=1) {
    if (tx<stride)
      shared[tx]+=shared[tx+stride];
    __syncthreads();
  }
  if (tx == 0)
    v_out[blockIdx.x]=shared[0];
}

__global__ void cudaCheckNodeEquations (int *hard, int *output,
int *cudaNumCNConn, int *cudaValCNConn, int *cudaIndCNConn, int CN,
int maxCNConn)
{
  unsigned int index;
  int val, mul, a1, a2;
  index=blockIdx.x*blockDim.x+threadIdx.x;

  if (index<CN) {
    val=-1;
    for (int i=0; i<cudaNumCNConn[index]; i++) {
      a1=cudaValCNConn[index*maxCNConn+i];
      a2=cudaIndCNConn[index*maxCNConn+i];
      mul=tex2D(gf_mul_tex, a1+1, hard[a2]+1);
      val=tex2D(gf_add_tex, val+1, mul+1);
    }
    output[index]=(val!=-1)?1:0;
  }
}

__device__ float cudaBoxPlusM(float a_m, float a_s, float b_m,
float b_s)
{
  if (a_m<=-BP_INFINITY+0.1f || b_m<=-BP_INFINITY+0.1f) {
        if (a_m>-BP_INFINITY+0.1f) return a_m;
        if (b_m>-BP_INFINITY+0.1f) return b_m;
        return -BP_INFINITY;
    }
    if (fabs(a_m-b_m)<BP_EPSILON)
        if (a_s!=b_s) return -BP_INFINITY;
        else return a_m+0.693147180559945f;
    if (a_m>=b_m)
        return a_m+__logf(1.0f+(a_s*b_s)*__expf(b_m-a_m));
    else
        return b_m+__logf(1.0f+(a_s*b_s)*__expf(a_m-b_m));
}

__device__ float cudaBoxPlusS(float a_m, float a_s, float b_m,
float b_s) {
    if (a_s == b_s || a_m>=b_m) return a_s;
    return -a_s;
}

__global__ void cudaLogFFTP(float *P_L, float *FP_L, float *FP_L_s,
int BN, int maxBNConn, int p, int q, int *numBNConn)
{
    extern __shared__ float tempFFT[];
    unsigned int tx, txx, ty, by, indexy;
    unsigned int maskBit;
    float a_m, b_m, a_s, b_s;
    tx=threadIdx.x;
    ty=threadIdx.y;
    by=blockIdx.y;
    indexy=by*blockDim.y+ty;
    txx=ty*q+tx;

    if (indexy<BN) {
        for (int j=0; j<numBNConn[indexy]; j++) {
            tempFFT[0*256+txx]=P_L[indexy*maxBNConn*q+j*q+tx];
            tempFFT[1*256+txx]=1.0f;
            __syncthreads();
            for (int i=0; i<p; i++) {
                maskBit=1<<i;
                if (!(tx & maskBit)) {
                    a_m=tempFFT[0*256+txx];
                    a_s=tempFFT[1*256+txx];
                    b_m=tempFFT[0*256+txx+maskBit];
                    b_s=tempFFT[1*256+txx+maskBit];
                }
                else {
                    a_m=tempFFT[0*256+txx-maskBit];
                    a_s=tempFFT[1*256+txx-maskBit];
                    b_m=tempFFT[0*256+txx];
                    b_s=-tempFFT[1*256+txx];
                }
                __syncthreads();
        tempFFT[0*256+txx]=cudaBoxPlusM(a_m,a_s,b_m,b_s);
        tempFFT[1*256+txx]=cudaBoxPlusS(a_m,a_s,b_m,b_s);
        __syncthreads();
      }
      FP_L[indexy*maxBNConn*q+j*q+tx]=tempFFT[0*256+txx];
      FP_L_s[indexy*maxBNConn*q+j*q+tx]=tempFFT[1*256+txx];
    }
  }
}

__global__ void cudaLogFFTQ(float *FQ_L, float *FQ_L_s, float *Q_L,
int CN, int maxCNConn, int p, int q, int *numCNConn)
{
  extern __shared__ float tempFFT[];
  unsigned int tx, txx, ty, by, indexy;
  unsigned int maskBit;
  float a_m, b_m, a_s, b_s;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;
  txx=ty*q+tx;

  if (indexy<CN) {
    for (int j=0; j<numCNConn[indexy]; j++) {
      tempFFT[0*256+txx]=FQ_L[indexy*maxCNConn*q+j*q+tx];
      tempFFT[1*256+txx]=FQ_L_s[indexy*maxCNConn*q+j*q+tx];
      __syncthreads();
      for (int i=0; i<p; i++) {
        maskBit=1<<i;
        if (!(tx & maskBit)) {
          a_m=tempFFT[0*256+txx];
          a_s=tempFFT[1*256+txx];
          b_m=tempFFT[0*256+txx+maskBit];
          b_s=tempFFT[1*256+txx+maskBit];
        }
        else {
          a_m=tempFFT[0*256+txx-maskBit];
          a_s=tempFFT[1*256+txx-maskBit];
          b_m=tempFFT[0*256+txx];
          b_s=-tempFFT[1*256+txx];

        }
        __syncthreads();
        tempFFT[0*256+txx]=cudaBoxPlusM(a_m,a_s,b_m,b_s);
        tempFFT[1*256+txx]=cudaBoxPlusS(a_m,a_s,b_m,b_s);
        __syncthreads();
      }
      Q_L[indexy*maxCNConn*q+j*q+tx]=tempFFT[0*256+txx];
    }
  }
}
```

```
__global__ void cudaPermuteP(float *P_L, int *numBNConn, int *valBNConn,
int BN, int maxBNConn, int q)
{
  extern __shared__ float sm[];
  unsigned int tx, ty, by, indexy;
  int h, temp1, temp2;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;

  if (indexy<BN*maxBNConn) {
    sm[ty*q+tx]=P_L[indexy*q+tx];
    __syncthreads();
    if (tx>0) {
      h=valBNConn[indexy];
      if (h!=0) {
        if (tx == 1)
          temp1=tex1Dfetch(gf_exp2dec_tex,h+1);
        else {
          temp1=tex1Dfetch(gf_dec2exp_tex,tx);
          temp2=tex2D(gf_mul_tex, temp1+1,h+1);
          temp1=tex1Dfetch(gf_exp2dec_tex,temp2+1);
        }
        sm[ty*q+temp1]=P_L[indexy*q+tx];
      }
    }
    __syncthreads();
    P_L[indexy*q+tx]=sm[ty*q+tx];
  }
}

__global__ void cudaPermuteQ(float *Q_L, int *numCNConn, int *valCNConn,
int CN, int maxCNConn, int q)
{
  extern __shared__ float sm[];
  unsigned int tx, ty, by, indexy;
  int h, temp1, temp2;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;

  if (indexy<CN*maxCNConn) {
    sm[ty*q+tx]=Q_L[indexy*q+tx];
    __syncthreads();
    if (tx>0) {
      h=valCNConn[indexy];
      if (h!=0) {
        if (tx == 1)
          temp1=tex1Dfetch(gf_exp2dec_tex,h+1);
        else {
          temp1=tex1Dfetch(gf_dec2exp_tex,tx);
          temp2=tex2D(gf_mul_tex, temp1+1,h+1);
          temp1=tex1Dfetch(gf_exp2dec_tex,temp2+1);
        }
        sm[ty*q+tx]=Q_L[indexy*q+temp1];
      }
    }
    __syncthreads();
    Q_L[indexy*q+tx]=sm[ty*q+tx];
  }
}

__global__ void cudaBitNodeUpdate(float *in_L, float *Q_L, float *P_L,
int *numBNConn, int *BN2CN, int BN, int q, int maxBNConn)
{
  extern __shared__ float sm[];
  unsigned int tx, txx, ty, by, indexy;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;
  txx=ty*q+tx;

  if (indexy<BN) {
    for (int j=0; j<numBNConn[indexy]; j++) {
      sm[0*256+txx]=in_L[indexy*q+tx];
      __syncthreads();
      for (int k=0; k<numBNConn[indexy]; k++) {
        if (k!=j) {
          sm[0*256+txx]+=Q_L[BN2CN[indexy*maxBNConn*q+k*q+tx]];
        }
      }
      sm[1*256+txx]=sm[0*256+txx];
      __syncthreads();
      for (int stride=q>>1; stride>0; stride>>=1) {
        if (tx<stride) {
          if (sm[1*256+txx+stride]>sm[1*256+txx])
            sm[1*256+txx]=sm[1*256+txx+stride];
        }
        __syncthreads();
      }
      if (tx == 0)
        if (sm[1*256+(ty*q+0)]<-BP_INFINITY)
          sm[1*256+(ty*q+0)]=-BP_INFINITY;
      P_L[indexy*maxBNConn*q+j*q+tx]=sm[0*256+txx]-sm[1*256+(ty*q+0)];
    }
  }
}

__global__ void cudaCheckNodeUpdate(float *FP_L, float *FP_L_s, float *FQ_L,
float *FQ_L_s, int *numCNConn, int *CN2BN, int CN, int q, int maxCNConn)
```

```
{
  unsigned int tx, ty, by, indexy;
  float module_sum, sign_prod;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;

  if (indexy<CN) {
    module_sum=0.0f;
    sign_prod =1.0f;
    for (int j=0; j<numCNConn[indexy]; j++) {
      module_sum+= FP_L[CN2BN[indexy*maxCNConn*q+j*q+tx]];
      sign_prod*=FP_L_s[CN2BN[indexy*maxCNConn*q+j*q+tx]];
    }
    for (int j=0; j<numCNConn[indexy]; j++) {
      FQ_L[indexy*maxCNConn*q+j*q+tx]=
        module_sum-FP_L[CN2BN[indexy*maxCNConn*q+j*q+tx]];
      FQ_L_s[indexy*maxCNConn*q+j*q+tx]=
        sign_prod/FP_L_s[CN2BN[indexy*maxCNConn*q+j*q+tx]];
    }
  }
}

__global__ void cudaInLUpdate(float *in_L, float *in_L_temp, float *Q_L,
int *BN2CN, int *numBNConn, int BN, int maxBNConn, int q)
{
  unsigned int tx, ty, by, indexy;
  float sum;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;

  if (indexy<BN) {
    sum=in_L[indexy*q+tx];
    for (int j=0; j<numBNConn[indexy]; j++)
        sum+=Q_L[BN2CN[indexy*maxBNConn*q+j*q+tx]];
    in_L_temp[indexy*q+tx]=sum;
  }
}

__global__ void cudaExpandGalois(int *hard, int *d_out, int p, int K)
{
  unsigned int tx, ty, by, indexy;
  int temp;
  tx=threadIdx.x;
  ty=threadIdx.y;
  by=blockIdx.y;
  indexy=by*blockDim.y+ty;

  if (indexy<K) {
    temp=tex1Dfetch(gf_exp2dec_tex, hard[indexy]+1);
    d_out[indexy*p+tx]=(int) ((temp>>(p-tx-1)) & 0x1);
  }
}
```

Chapter 8

Appendix C NVIDIA GeForce GTX 275 Specifications

System Architecture	GT200
Compute Capability	1.3
Multiprocessor Count	30
CUDA Cores	240
Thread Occupancy on SM	1024
Block Occupancy on SM	8
Maximum threads per block	512
Maximum Block X-dimension	512
Maximum Block Y-dimension	512
Maximum Block Z-dimension	64
Maximum Grid X-Dimension	65535
Maximum Grid Y-Dimension	65535
Warp Size	32
Total Registers per Block	16 KBytes
Total Shared Memory per SM	16 KBytes
Total Constant Memory	64 KBytes
Total Global Memory	896 MB
Memory Interface Width	448 bit
Memory Clock	1134 MHz
Memory Bandwith	127 GB/s

Bibliography

[1] Andrew D. Copeland, Nicholas B. Chang, and Stephen Leung. GPU Accelerated Decoding of High Performance Error Correcting Codes. Technical report, Massachusetts Institute of Technology, 2009.

[2] Burkhard Zink. A general relativistic evolution code on CUDA. 2008.

[3] D. J. C. MacKay, R. M. Neal. Good Codes Base on Very Sparse Matrices. In *Cryptography and Coding, 5th IMA Conference*, 1995.

[4] David D. Kirk, Wen-mei W. Hwu. *Programming Massively Parallel Processors.* Morgan Kaufmann, 2009.

[5] Gabriel Falçao, Leonel Sousa, Vitor Silva. Massive Parallel LDPC Decoding on GPU. *IEEE Parallel and Distributed Systems*, 2008.

[6] Gabriel Falçao, Vitor Silva, Leonel Sousa. How GPUs Can Outperform ASICs for Fast LDPC Decoding. In *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[7] Gabriel Falçao, Vitor Silva, Leonel Sousa. Parallel LDPC Decoding on GPUs Using a Stream-Based Computing Approach. *Springer Boston Journal of Computer Science and Technology*, 2009.

[8] Geoffrey J. Byers, Fambirai Takawira. Fourier Transform Decoding of Non-Binary LDPC Codes.

[9] Jason Sanders, Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming.* Addison-Wesley, 2010.

[10] Moritz Beermann. Entwurf und Optimierung von LDPC- und Turbo-Codes für Hochgeschwindigkeits-Mobilfunknetze. Master's thesis, Institut für Nachrichtengeräte und Datenverarbeitung, RWTH Aachen University, 2010.

[11] NVIDIA Corporation. *NVIDIA CUDA C Programming Best Practices Guide version 2.3.* NVIDIA Corporation, 2009.

[12] NVIDIA Corporation. *NVIDIA CUDA Programming Guide version 2.3.* NVIDIA Corporation, 2009.

[13] NVIDIA Corporation. *NVIDIA CUDA Reference Manual version 2.3*. NVIDIA Corporation, 2009.

[14] NVIDIA Corporation. NVIDIA GF100. Whitepaper, 2010.

[15] P. Vary. Advanced Coding and Modulation. Overview. 2010.

[16] R. G. Gallager. Low-Density Parity-Check Codes. *Cambridge, MA: MIT Press*, 1963.

[17] Shuang Wang, Samuel Cheng, Qiang Wu. A Parallel Decoding Algorithm of LDPC Codes using CUDA. In *Conference on Signals, Systems and Computers, 2008 42nd Asilomar*, 2008.

[18] Todd K. Moon. *Error Correction Coding. Mathematical Methods and Algorithms.* Wiley-Interscience, 2005.

[19] Upamanyu Madhow. *Fundamentals of Digital Communication.* Cambridge University Press, 2008.

[20] Various. *GPU Gems 2: Programming Techniques for High-Performance Graphics and General-Purpose Computation.* Addison-Wesley, 2005.