IMPLEMENTATION OF LOG-DOMAIN FFT BASED LDPC DECODER ON A GPU

by

Hanan Alqarni
A Thesis
Submitted to the
Graduate Faculty
of
George Mason University
In Partial fulfillment of
The Requirements for the Degree
of
Master of Science
Computer Engineering

Committee:

_____	Dr. Brian L. Mark, Thesis Director
_____	Dr. Bernd-Peter Paris, Committee Member
_____	Dr. Xiang Chen, Committee Member
_____	Dr. Monson H. Hayes, Department Chairperson
_____	Dr. Kenneth S. Ball, Dean, Volgenau School of Engineering

Date: _____ Spring Semester 2019
George Mason University
Fairfax, VA

Implementation of Log-Domain FFT Based LDPC Decoder on a GPU

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science at George Mason University

By

Hanan Alqarni
Bachelor of Science
Princess Nora Bint AdbulRahman University. Riyadh, KSA, 2014

Director: Dr. Brian L. Mark, Professor
Department of Electrical and Computer Engineering

Spring Semester 2019
George Mason University
Fairfax, VA

Acknowledgments

I would like to express my sincere gratitude to my advisor Prof. Brian Mark for the continuous support and guidance. As an advisor he has made sure that my graduate studies at GMU would be significantly vital and valuable. I am grateful to him for his significant guidance not only in my studies but also in this thesis. I am grateful for his support and the opportunity he offered for joining the DARPA challenge project last summer.

Beside my advisor, I would like to thank Dr. Bernd-Peter Paris who allowed me to join the DARPA challenge project in the summer 2018 and provide me the access to the project. This opportunity played significant role in my learning as I learn not only about the project but also how to tackle an issue and find the best solution for it.

I owe a special thanks to the graduate coordinator Ms. Patricia Sahs who has always helped me in the department. Ms. Sahs has assisted me in the smallest details, which has eased the process of my graduate studies, in things such as, course enrollment, and the thesis preparation process.

Table of Contents

List of Tables

List of Figures

Abstract

IMPLEMENTATION OF LOG-DOMAIN FFT BASED LDPC DECODER ON A GPU

Hanan Alqarni

George Mason University, 2019

Thesis Director: Dr. Brian L. Mark

Forward error correction enables reliable one-way communication over noisy channels by transmitting redundant data along with the message in order to detect and resolve errors at the receiver. Low-density parity-check (LDPC) codes achieve superior error-correction performance using belief propagation (BP) decoding. However, the computational complexity of a BP decoder is $O(q^2)$ operations for each checksum calculation, where q represents number of symbols in the underlying Galois field. The complexity is reduced by transforming the operations into the log and frequency domains. This thesis explores how a GPU implementation of a Log-domain FFT based LDPC decoder performs in comparison to a CPU implementation for regular MacKay construction. Numerical results show that the GPU implementation is about twice as fast as the CPU implementation. The thesis also studies the performance of GPU implementations of a Quasi-cyclic LDPC decoder for WIFI (IEEE 802.11n) and WIMAX (IEEE 802.16e) LDPC codes.

Chapter 1: Introduction

Low-density parity-check (LDPC) codes were first introduced by R. G. Gallager in 1960 [5], are a class of linear block codes. Theses codes are characterized by their parity-check matrix, a low density of non-zero elements. The benefit of LDPC codes is that closely approaches the capacity of many channels. In addition, LDPC codes are more convenient for heavy implementations. However, these codes were ignored until the 90's due to the computational complexity of the implementation and the limitation of the hardware.

A convenient way to represent LDPC codes is via a Tanner graph representation, which is an effective graphical representation that efficiently helps to describe the decoding algorithm. Tanner graphs are bipartite graphs, which have nodes separated into two different sets (variable nodes and check nodes), and edges connecting nodes of the two different sets, representing parity-check matrix.

The decoding algorithms decode the codes by performing local calculations and passing those local results via messages. This message passing algorithm. Belief-propagation algorithm is one of the most common message passing algorithm for decoding binary LDPC codes. The log-domain Fourier transform belief-propagation algorithm is message passing algorithm used for the case of non-binary LDPC codes which achieve good decoding performance.

A parallel computing architecture is effective method used to raise the performance of iterative decoding of highly computational codes such as LDPC. Compute Unified Device Architecture (CUDA) is a parallel computing architecture developed by NVIDIA Corporation, remains in the current NVIDIA graphics cards. It is accessible to software developers through variants of industry standard programming languages. Using regular programing languages such as C/C++ it is possible to program CUDA devices with NVIDIA extensions and certain restrictions. CUDA chip 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 Graphical User Interface (GPU) are single instruction, multiple data (SIMD) cores.

Motivated by both the parallel computation power of the CUDA architecture implementation of LDPC codes, the decoding performance of LDPC codes are studied in this thesis. Parallel implementation using CUDA is implemented using an existing C software based log-domain Fourier transform belief propagation algorithm, as well as Quasi-Cyclic LDPC algorithm. The main objective is achieving an efficient performance of CUDA implementation of decoding system and exposing to different implementation of LDPC decoding algorithm.

Chapter 2 introduces the necessary theoretical background for understanding LDPC codes both Log-domain FFT and Quasi-Cyclic algorithms, and CUDA architecture. The LDPC codes are presented together with their main characteristic, the parity-check matrix, and their graphical representation (Tanner graph), which is the base for performing the iterative LDPC decoding. Algorithms belief propagation and the log-domain Fourier transform, and QC-LDPC are explained step by step. Afterwords, the CUDA architecture is introduced and its parallel processing capabilities of these graphical cards, as well as performance considerations to achieve better performance is explained.

Chapter 3 discusses the CUDA implementation of the LDPC decoders in CPU in detail. Then, the belief propagation and the log-domain Fourier transform algorithms are parallelized and assembled in kernels using CUDA implementation. Each kernel uses the hardware resources in a different way, in order to achieve the best performance. The implantation of Log-FFT LDPC on Tesla k40m GPU is detailed in this chapter. Also, QC-LDPC implementation on GPU is discussed along with its optimization solution for memory latency.

Chapter 4 Discuss the performance of the implementation of Log-domain FFT based

LDPC on both GPU and CPU, and discuss the performance of QC-LDPC on GPU. Future work addressed for FPGA hardware implementation of LDPC for both regular code using regular parity-check matrix, WiFi and WiMax matrices.

Chapter 2: Theoretical Background

In channel coding, error correcting codes are essential mechanism due to the increase of data transmission and storage systems, as the speed of communications, distance of transmission, or amount of data to be transmitted are also increasing and introducing errors. Indeed, data transmissions are not limited to wireless communications, thus, cable or terrestrial transmission systems also adopt these mechanisms. The capability of detecting errors - even correcting them in some cases - by the recipient of a message, has made systems with error correcting mechanism widespread and available in many of recent standards. Among the different classes of error correcting codes, the focus of this chapter is on the study of binary linear block codes, especially Low-Density Parity-Check (LDPC) whose powerful characteristics have been widely addressed, investigated and exploited by the communication and information theory community [6].

The term Forward Error Correction (FEC) is used when the receiving equipment does most of the decoding work. In the case of block codes, once the errors are detected by the decoder, it corrects them. FEC technique can reduce bit error rate (BER) at a fixed power level or allow a specified error rate at a reduced power level at the cost of increased

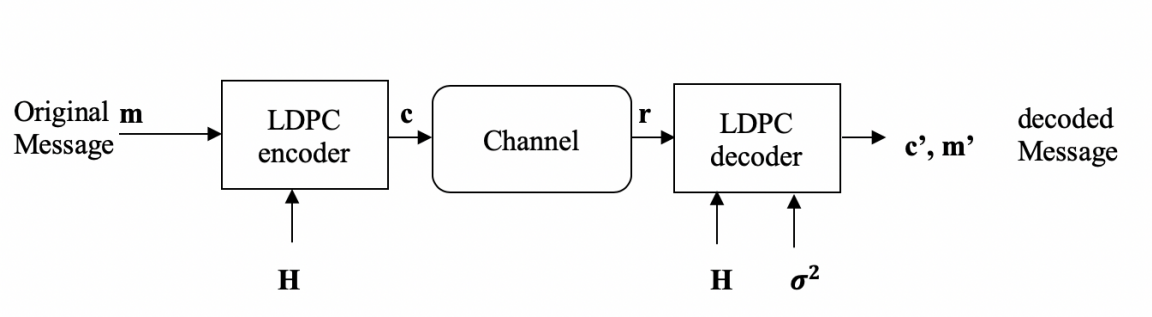

Figure 2.1: Simplified model of LDPC encoding and decoding.

bandwidth (or transmission delay) and processing burden [6].

Shannon's channel coding theorem has stimulated the improvement of error control codes. All the data rates r_b should be less than the channel capacity C so that it can satisfy Shannon theorem, and, therefore, it can have an arbitrarily small probability of error P_e [7]. The channel capacity C is given by the Shannon formula [1],

$$C = B\log_2[S/N] \quad (bits \quad per \quad second) \tag{2.1}$$

where B is the bandwith in Hz, and S/N is signal to noise ration (SNR) which is equal to $\frac{r_b E_b}{B N_0}$. Substituting S/N into (Eq.2.1) [1] we get,

$$\begin{aligned} \frac{C}{B} &= \log_2(1+\eta_{max}\frac{E_b}{N_0}) \quad (or) \\ \frac{E_b}{N_0} &= \frac{2^{\eta_{max}-1}}{\eta_{max}} \end{aligned} \tag{2.2}$$

where $\frac{E_b}{N_0}$ is called Shannon's Limit or specifically Shannon's power efficiency limit, which is used to to evaluate coding-modulation scheme. Hence, it gives the required limit to transmit data at a rate close to the channel capacity C [1].

2.1 Low-Density Parity-Check Codes

Low-Density Parity-Check (LDPC) Codes were initially introduced by Robert Gallager in 1962 [5] and they are sometimes called 'Gallager codes' [8]. LDPC codes were nor noticed or studied for decades due the limitations of computing capability - Hardware - at that time. LDPC codes were recognized - in 1965 - by their remarkable performance in which allowing the rate of data transmission more closely to Shannon limit by Mackay and Radford [9]. Lately, they have become popular due to proven performance of meeting channel capacity and achieving excellent Bit Error Rate (BER) due to computationally intensive algorithms

on the decoder side of the system of historical and recent LDPC codes. The powerful advantages LDPC codes offer have led to recent inclusion of LDPC in several standards, such as IEEE 802.16, IEEE 802.20, IEEE 802.3, and DBV-RS2 [10].

LDPC codes are a long linear block of codes that are defined by generator matrix **G**, N x K matrix, or parity check matrix **H** in non-systematic form, (N - K) x N matrix, which have a low density of non-zero elements. LDPC generator is called *Regular* when all the columns' weight w_c are constant and all the rows' weight w_r are constant. Otherwise, it is called *Irregular*.

A parity-check matrix **H** defines an (w_c, w_r, N) on LDPC, N denotes the length of the code, K is used to denote its dimension, and the redundancy is $M = N - K$. This **H** said to be *sparse* if less than half of its elements are non-zero. The linear block encode k information bits into N coded bits. The corresponding regular (w_c, w_r) code has a design rate of $R = 1 - w_c/w_r$. By using the row vector notation, the coded vector **c** is obtained from the information message **m** by vector-matrix multiplication,

$$\begin{aligned} \mathbf{c} &= (c_1, c_2, ..., c_n), \\ \mathbf{m} &= (m_1, m_2, ..., m_k), \\ \mathbf{c} &= \mathbf{m} \cdot \mathbf{G}. \end{aligned} \tag{2.3}$$

Each row of **H** provides a parity-check equation that any codeword **c** must satisfy,

$$\mathbf{c} \cdot \mathbf{H}^\top = \mathbf{0} \tag{2.4}$$

The direct decoding process of the received codeword $\mathbf{c}'$ can have a high complexity due to the huge dimensions that **H** can reach. Hence, LDPC decode the received codeword using *message passing algorithms* over a Tanner graph discussed in the next subsections.

2.2 Minimum Distance

The Hamming distance $D(x, y)$ between two codewords x and y is defined as the number of bits with different values between x and y. The Minimum Distance of a certain code c is the lowest Hamming distance between two codewords over the entire range of codewords of that code c [11]. The larger the minimum distance, the code and correct a larger number of bit errors. Hence, the transmitted codeword will be decoded to the closest valid codeword [11]. If the minimum distance is small, the received codeword could decode to a different codeword from the one transmitted. A weight w is defined for a codeword x and it represents the number of $1s$ in x. A weight of code is called, minimal weight, if the weight of the codeword has the lowest non-zero weight. Thus, the minimum distance of a linear code is the minimal weight of this code.

2.3 Tanner graph

A linear binary block code (N, K) can be described by a binary **H** matrix with dimen- sions (N - K) x N. Also, it can be elegantly represented by a Tanner graph[118] defined by edges connecting two distinct types of nodes:

- BitNodes(v_i) that is also called variable nodes with a v_i for each one of the N codeword of the linear system of equations, and;
- Check Nodes (c_i), also called restriction or test nodes with a c_i for each one of the (N - K) homogeneous independent linear system of equations represented by **H**.

Each c_i connects to all the v_i which have a contribution in that restriction or test equation. For the other type of nodes, each Bit Nodes corresponding to a v_i bit of the codeword, connects to all the Check Nodes equations where bit v_i participates in. The graph edges connect Bits Nodes with Check Nodes, but never nodes of the same type, which defines a bipartite graph. Every element in **H** where $\mathbf{H}_{ij} = 1$, represents a connection between v_j and c_i.

Tanner graph is factor graph that can be used to describe the LDPC iterative decoding algorithm. The terms below pertain to evaluate the algorithm in Tanner graph [1]:

Cycle:

A cycle in a Tanner graph is a sequence of connected codeword nodes and parity check nodes that begin and conclude at the same node and no other nodes can appear in the sequence more than once. This is useful to determine the convergence of the algorithm. The Sum-Product algorithm shows that it can be converged at the optimal solution if the Tanner graph has no cycle [1]. The presence of shorter cycles will degrade the algorithm.

Length:

The length of a cycle is typically the number of edges (the connecting line between the check node and the variable node) in the cycle.

Girth:

The girth of a Tanner graph is the length of its shortest cycle.

Degree:

The degree of a node in the Tanner graph is the number of edges connected to it.

2.4 Parity-Check Matrix Construction

Parity-check matrix can be constructed in term of random sparse matrix, which makes it easy to create LDPC code with any code rate. Many codes is constructed by specifying a fixed weight for each column and row, then constructing the matrix at random subject to these constraints. LDPC code have different design criteria for its matrix that is reviewed in this thesis.

2.4.1 Gallager's Parity-Check Matrix Construction

Gallager [5] imposed in his work a fixed column weight w_c and a fixed row weight w_r. The parity check matrix was horizontally divided into w_c equal size sub-matrices; where each sub-matrix contain in each of its column a single $'1'$. The first sub-matrix was constructed

in predetermined manner. The subsequent sub-matrices were random column permutations of the first [12]. Gallager's construction technique [13] is outline as follows:

1. The code parameters:

   N , K

   w_c (the column weight),

   w_r (the row weight) are given,

2. The number of the rows is constructed using $\frac{N-K}{w_c}$ rows and the number of the columns is N,

3. The H_1 sub-matrix is constructed with $\frac{N-K}{w_c}$ x N dimension,

4. Let $\Pi(H_1)$ be a pseudo-random column permutation of H_1,

5. The parity-check matrix $\mathbf{H}$ is constructed by stacking w_c submatrices:

$$\mathbf{H} = \begin{bmatrix} \frac{H_1}{\Pi(H_1)} \\ \frac{}{\Pi(H_1)} \\ \end{bmatrix} \tag{2.5}$$

The feature of LDPC codes to perform near the Shannon's limit of a channel exist mostly for large block lengths. The large block length results in large parity-check and generator matrices. The complexity of multiplying a codeword with a matrix depends on the amount of 1's in the matrix [12]. Thus, the sparse matrix $\mathbf{H}$ should be in the systematic form such that $\mathbf{H} = [P^T \qquad I]$, where P is sub-matrix that is generally not sparse and I is identity matrix. Then the generator matrix $\mathbf{G}$ can be constructed as $\mathbf{G} = [I \qquad P]$. The sub-matrix P makes the encoding complexity quite high since it not sparse matrix. Therefore the complexity grows $O(N^2)$ even though sparse matrices do not result in a good performance if the block length gets very high [12], [13].

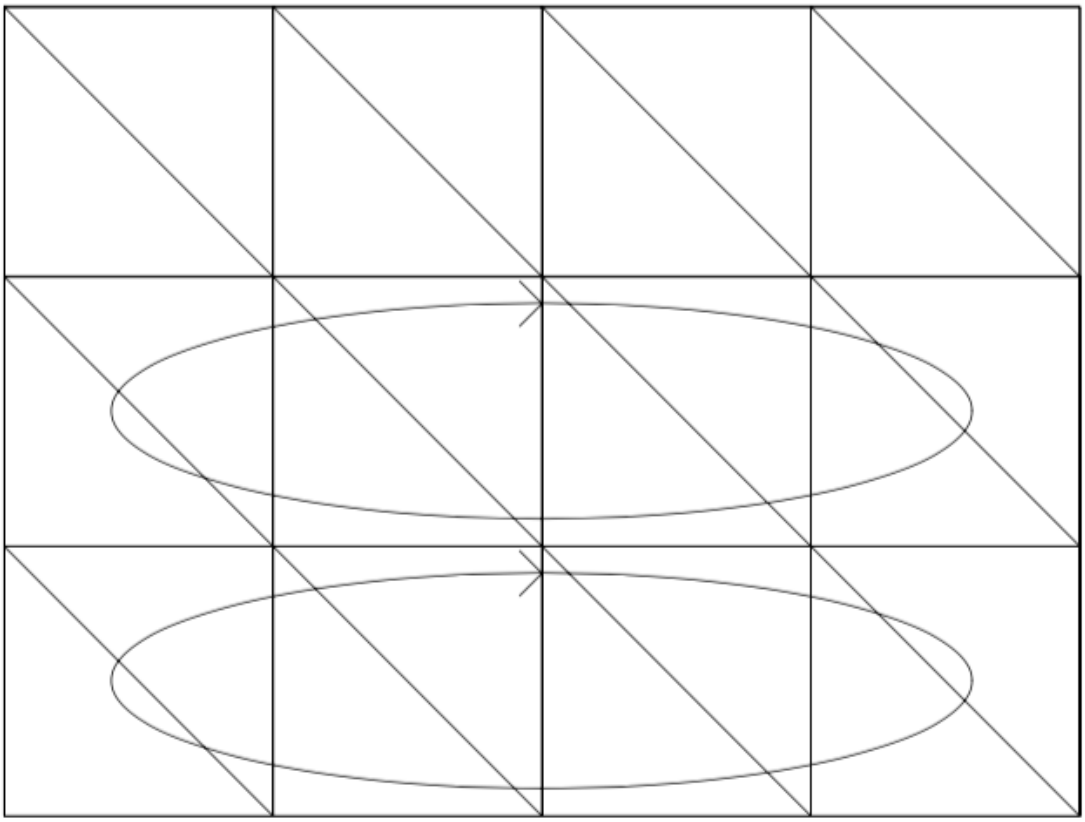

Figure 2.2: Gallager's **H** constructed matrix [1]. Note that the circles represent the permutation matrix and the diagonal lines represent the identity matrix

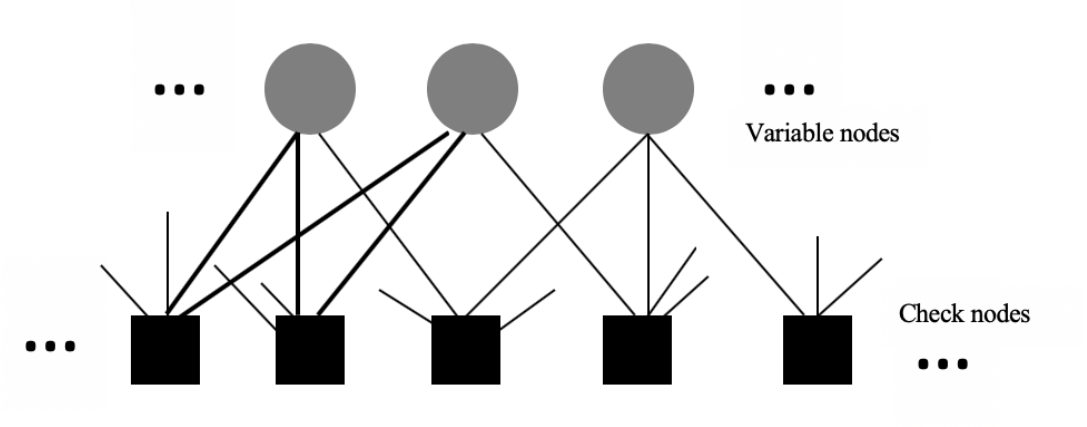

Figure 2.3: An example of **H** has a cycle of length 4 represented in Tanner graph [1]

2.4.2 MacKay's Parity Check Matrix Construction

Tanner graph is the representation of the parity-check matrix **H**, and short cycles in the Tanner graph cause degradation of the algorithm convergence. Thus, Mackay's main goal was to keep the short cycles in Tanner graph as minimum number as possible. Belif-Propagation decoder algorithm faces difficulties when short cycles are present in Tanner graph as illustrated in the following example [14]:

Figure 2.3 represents a cycle of length 4 that is indicated by the bold edge between v_i and c_i nodes. If the state of variable node v_i changes to $(v_i + \mathbf{c})$ for arbitrary $\mathbf{c} \in GF(2^p)$, then only one check c_j is affected. Therefore, the decoder will stuck and it is difficult to find the evidence of the other satisfied checks c_i. To ensure that any pair of columns in the **H** matrix has an overlap at most one, a cycle of length 4 is avoided [14]. The construction of **H** with no cycles of length 4 is described in three ways [14]:

1A construction: is the basic construction that has fixed weight per column w_c (i.e. $w_c = 3$), and keeping the weight per row w_r as uniform as possible. As well as the overlap between any two columns no greater than 1. See Figure 2.4.

2A construction: similar to 1A construction except that the up to $m/2$ columns have

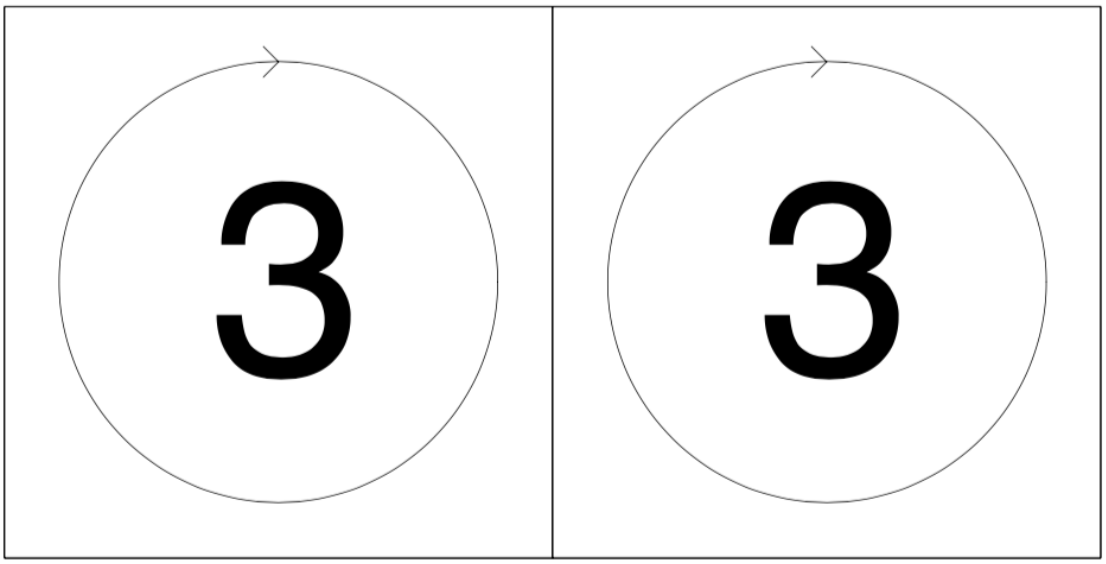

Figure 2.4: 1A construction of **H** with $w_c = 3$ and $w_r = 6$ and the rate is 1/2. Note that the integer in the circle represents the number of permutations matrix.

weight 2. These weight 2 columns are in form of two identity matrices of size $m/2$ x $m/2$, in which one is the above than the other one. See Figure 2.5. This irregular construction using weight columns was introduced by MacKay [14] that had better performance in practice.

1B, 2B construction: in those construction, a small number of columns are omitted from a matrix produced by 1A and 2A, respectively. So that the Tanner graph corresponding to the matrix has no short cycles of length less than some length l.

2.4.3 WiMax's Parity Check Matrix Standards

WiMAX standard (IEEE 802.16e) uses LDPC codes, whose decoders can be very demanding from a computational perspective [15]. For this reason, they are still implemented using dedicated hardware based on ASIC solutions. WiMAX standard is based on a special class of LDPC codes [15] that is characterized by a sparse binary block parity-check matrix H of the form:

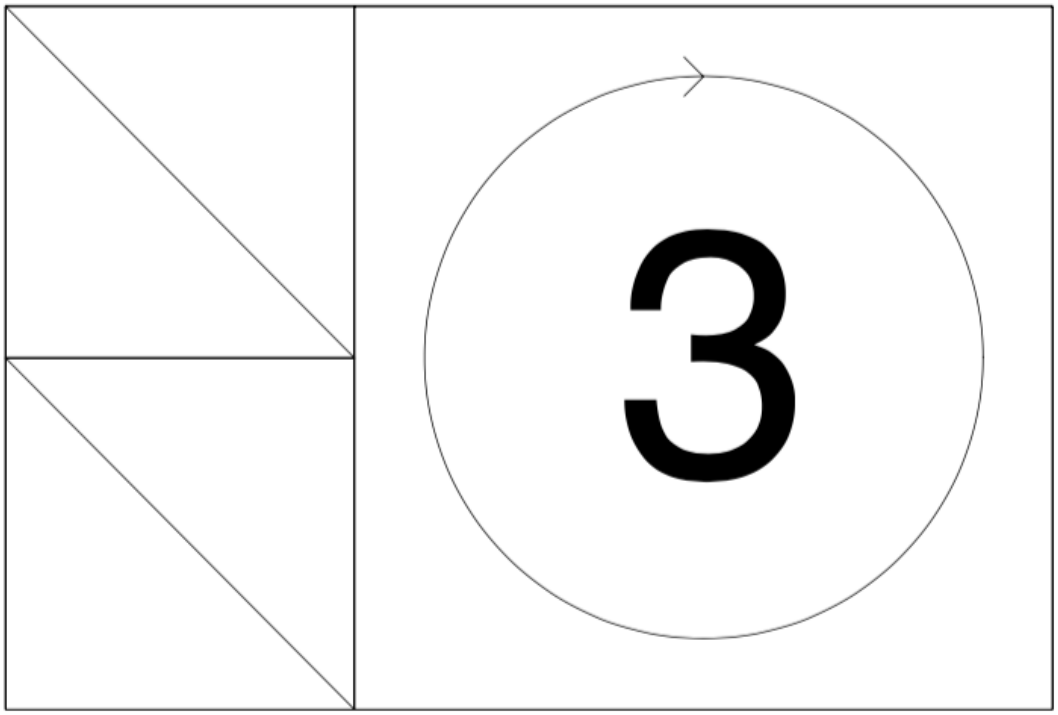

Figure 2.5: 2A construction of **H** with the rate is 1/3. Note that the diagonal line represents the identity matrix

$$\mathbf{H} = [\mathbf{H}_1 \quad \mathbf{H}_2] \tag{2.6}$$

$$= \begin{bmatrix} P_{0,0} & P_{0,1} & . & . & . & P_{0,k/z-1} & Pb_0 & I & 0 & ... & 0 \\ P_{1,0} & P_{1,1} & . & . & . & P_{1,k/z-1} & Pb_1 & I & I & ... & ... \\ ... & ... & . & . & . & ... & ... & ... & ... & ... & \\ P_{n-k/z-2,0} & P_{n-k/z-2,1} & . & . & . & P_{n-k/z-2,k/z-1} & ... & ... & ... & I & I \\ P_{n-k/z-1,0} & P_{n-k/z-1,1} & . & . & . & P_{n-k/z-1,k/z-1} & Pb_{n-k/z-1} & 0 & ... & 0 & I \end{bmatrix} \tag{2.7}$$

where $\mathbf{H}_1$ is sparse and has special periodicity constraints introduced in the pseudo random design of the matrix [IEEE P802.16e/D12, 2005; see paper [16]. $\mathbf{H}_2$ is a sparse lower triangular block matrix with a staircase profile. The periodic nature of these codes defines $\mathbf{H}_1$ based on permutation sub-matrices $\mathbf{P}_{i,j}$, which are:

- quasi-random circularly shifted right identity sub-matrices I , as depicted in Figure 2.6.The dimensions is $Z \times Z$ ranging from 24x24 to 96x96 and incremental granularity of 4.
- $Z \times Z$ null sub-matrices.

The periodic nature of such codes allowed simplifying the architecture of the system and storage requirements without code performance loss [15]. Also, the right sub-matrix $\mathbf{H}_2$ is formed by identity: I sub-matrices of dimension $Z \times Z$; or null sub-matrices of dimension $Z \times Z$.

2.4.4 WiFi's Parity Check Matrix Standards

The special structure of the IEEE 802.11n LDPC parity check matrices makes the encoding process is done very efficient. The IEEE 802.11n LDPC codes are based on block-structured

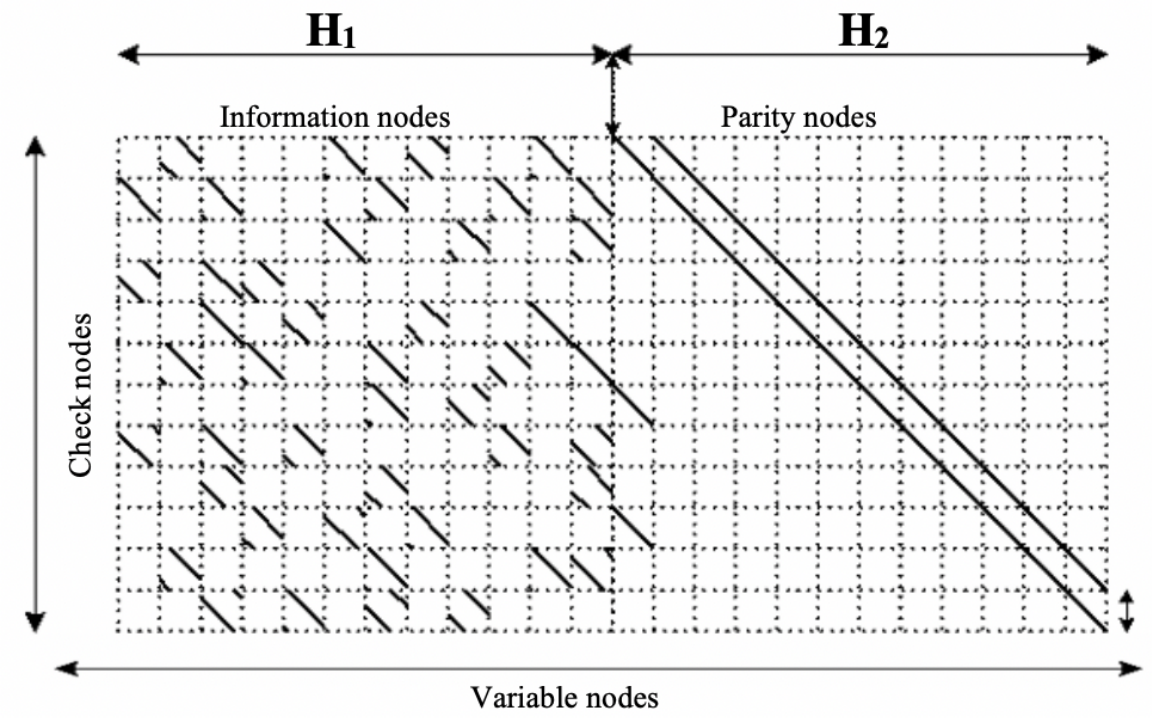

Figure 2.6: Periodicity $Z \times Z$ = 96x96 for a matrix with n=2304, rate=1/2

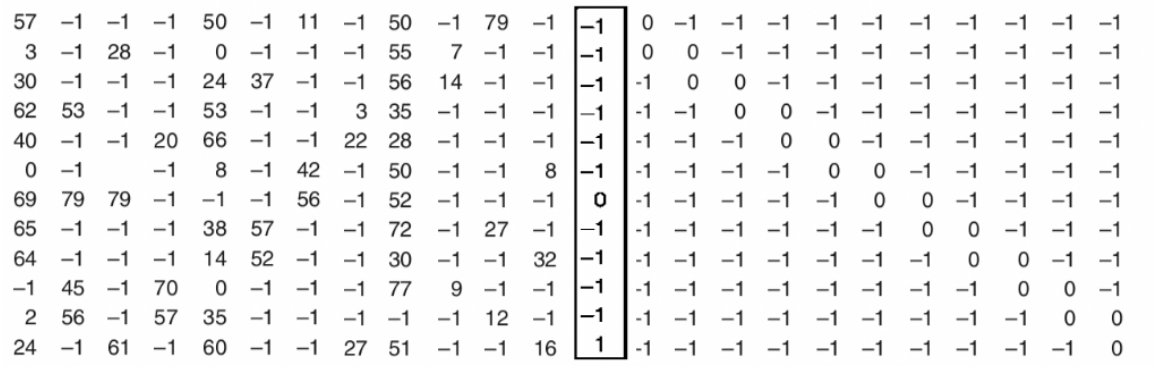

Figure 2.7: Example base matrix for N=1944 bits, rate = 1/2 LDPC code

LDPC codes with circular block, the entire parity-check matrix **H** can be partitioned into an array of block matrices [17], [18]; each block matrix is either a zero matrix (null) or a right cyclic shift of an identity matrix. The parity check matrix designed in this way can be conveniently represented by a base (block) matrix.

An example of WIFI (IEEE 802.11n) base matrix $\mathbf{H}_b$ for an LDPC code is illustrated [18] in Figure 2.7, with code length $N = 1944$ bits and $rate = 1/2$. The block size is $Z = 81$ bits with $m_b = 12$ and $n_b = 24$. In $\mathbf{H}_b$ matrix, each entry represents a circular right-shift of the identity matrix I_Z. i.e. if $Z = 3$ and the entry is 1, then the corresponding block is $[010; 001; 100]$. The -1 entry means a null block. Thus, this $\mathbf{H}_b$ matrix is a compact expression of a binary $2D$ such that $[M = 12-81, N = 24-81]$ matrix.

The parity-check matrix will be in the form of the following;

$$\mathbf{H}_b = \begin{bmatrix} h_{0,0} & h_{0,1} & . & . & . & h_{0,k_b-1} & 1 & 0 & ... & -1 & \\ h_{1,0} & h_{1,1} & . & . & . & h_{1,k_b-1} & -1 & 0 & ... & -1 & \\ ... & ... & . & . & . & ... & ... & ... & ... & ... & \\ h_{m_b-2,0} & h_{m_b-2,1} & . & . & . & h_{m_b-2,k_b-1} & 0 & 0 & ... & 0 & -1 \\ h_{m_b-1,0} & h_{m_b-1,1} & . & . & . & h_{m_b-1,k_b-1} & 1 & -1 & ... & 0 & 0 \end{bmatrix} \tag{2.8}$$

Table 2.1: Minimum distance of 802.11n LDPC codes with large packet

Code Rate r	**Weight of Matrix** (independent of n)	**Search using Parity Check Matrix** (n = 1944 bits)
1/2	27	33
2/3	17	21
3/4	12	17
5/6	10	14

Table 2.2: Minimum distance of 802.11n LDPC codes with medium-size packet

Code Rate r	**Weight of Matrix** (independent of n)	**Search using Parity Check Matrix** (n = 1296 bits)
1/2	31	23
2/3	17	15
3/4	22	10
5/6	17	9

There are four code rates for each of three block length that are specified in this standard. The following three tables [19]. The Table 2.1 highlights the largest block length codes when $n = 1944$ bits, Table 2.2 shows the medium block length codes where $n = 1296$ bits, and lastly Table 2.3 illustrates the smallest code length where $n = 648$ bits. The tables shown the following code rates 1/2, 2/3, 3/4, and 5/6, for the following $\mathbf{H}$ matrices: 12×24, 8×24, 6×24, and 4×24, respectively.

2.5 Binary LDPC decoding: Belief-Propagation and Iterative Decoding

Tanner graphs as in section 2.3, have often been proposed to perform approximate inference calculations [20]. They are based on iterative intensive message-passing algorithms

Table 2.3: Minimum distance of 802.11n LDPC codes with small packet

Code Rate r	**Weight of Matrix** (independent of n)	**Search using Parity Check Matrix** (n = 648 bits)
1/2	31	15
2/3	27	12
3/4	14	8
5/6	19	8

also known as belief propagation (BP) which, under certain circumstances, can become computationally prohibitive. BP, also known as the Sum-Product Algorithm (SPA)[1], is an iterative algorithm [21]. For the computation of joint probabilities on graphs, it is commonly used in information theory such as in channel coding, as well as artificial intelligence (AI) and computer vision (e.g., stereo vision). It has proved to be efficient and is used in numerous applications including LDPC codes [22] and other codes.

SPA applied to LDPC decoding operates with probabilities, to exchange information and update messages between neighbors over successive iterations. Considering a codeword $\mathbf{c}$ is to be transmitted over a noisy channel, the theory of graphs applied to error correcting codes has fostered codes are extremely close to the Shannon limit [23]. In bipartite graphs, particularly in those with large dimensions (e.g., N >1000 bit), the uncertainty on a bit can be spread over neighboring bits of a codeword allowing, in certain circumstances, in the presence of noise, to recover the correct bits on the decoder side of the system. In a graph representing a linear block error correcting code, reasoning algorithms exploit probabilistic relationships between nodes imposed by parity-check conditions that allow inferring the most likely transmitted codeword. The BP mentioned before allows to find the maximum a posteriori probability (APP) of vertices in a graph [21].

Propagating probabilities across nodes of the Tanner graph, rather than just flipping bits [10] which is considered hard decoding, is defined as soft decoding. This iterative procedure accumulates evidence imposed by parity-check equations that try to infer the true value for each bit of the received word [10]. Using the probabilistic decoding algorithm sum-product or BP [10], at each step the posterior probability of the value of each noise symbol —variable node v_i —given the received signal $\mathbf{r}$ and the channel properties.

$$P(v_n|\mathbf{r}, \mathbf{H}^\top \mathbf{c} = \mathbf{0}) \tag{2.9}$$

This process is represented Tanner graph as a message passing algorithm on bipartite

[1]In the literature, the terms BP and SPA are commonly undistinguished. Both are used in this text without differentiation.

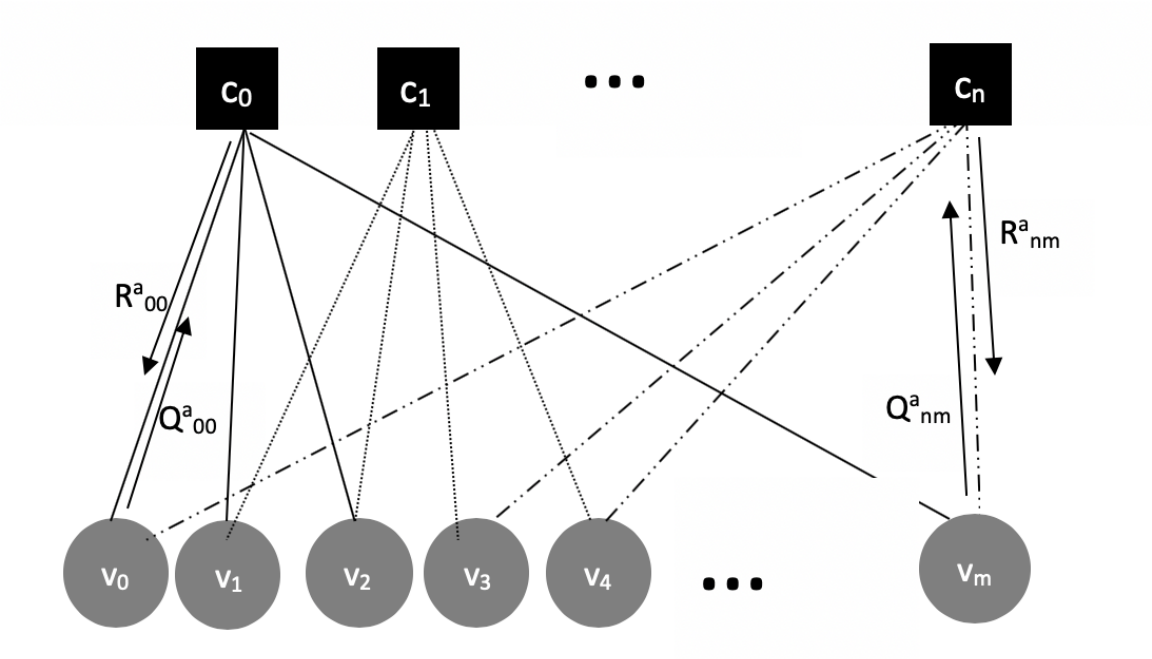

Figure 2.8: Massage Passing in Tanner graph or Bipartite graph.

graph defined by $\mathbf{H}$ in which the two sets of nodes: variable nodes v_i representing the noise symbols and and check nodes c_i. The nodes v_i and c_i are connected if the corresponding $\mathbf{H}_{ij}$ is non-zero. The directed edges show the relationships such that the state of a check node is determined by the state of variable nodes to which it is connected. This can be represented in the tree graph; where the neighbors of variable nodes v_i as the *parents* and the neighbor of check nodes c_i as the *childern*.

At each step of decoding each variable node v_i sends message q_{ij}^a to each child /check nodes c_i connected to it, to approximate the node's belief in state a, given messages are received from all its other children. In addition, the check node c_i sends messages r_{ij}^a to each parent v_i that is involved to the check equation approximating the probability of check c_i being satisfied if the parent is assumed to be in state a, given that the messages are received from all its other parents [24]. After each step, the messages are updated iteratively until

the decoding satisfies the observed syndrome vector. Declaration of decoding algorithm can be (success) if the syndrome vector is **0**, and can be (failure) if the maximum number if iteration is reached.

Initialization:

The message q_{ij}^{a} is set to f_{j}^{a} which is the prior probability that the jth variable node is a [24].

Updating Check Nodes Messages r_{ij}^{a}:

Check nodes messages r_{ij}^{a} that the check i sends to variable j should be the probability of check i being satisfied if the variable was in stat a. That is, the check i is satisdied if it agrees with the corresponding syndrome symboled s_i.

$$P(c_i|v_j = a) = \sum_{\mathbf{v}:v_j=a} P(c_i|\mathbf{v})P(\mathbf{v}|v_j = a) \tag{2.10}$$

Hence, all **v** configurations is summed for which the check is satisfied and the variable is in state a and added up the probability of the configurations; that is the product of associated Q messages. Now the message r_{ij}^{a} for node c_i is updated to the node v_i for each state a as follow:

$$r_{ij}^{a} = \sum_{\mathbf{v}:v_j=a} P(c_i|\mathbf{v}) \prod_{k\in\mathcal{N}(i)\backslash j} q_{ik}^{v_k} \tag{2.11}$$

where $\mathcal{N}(i)$ is the set of indices of the variable nodes v_i and $\mathcal{N}(i)\backslash j$ is the the set of indices of the variable nodes v_i except node v_j. The probability $P(c_i|\mathbf{v})$ is either 0 or 1 for any given configuration **v**.

Updating Variable Nodes Messages q_{ij}^{a}:

Variable nodes messages q_{ij}^{a} that the noise node j sends to check i the belief it has in state a. The message is updated as follow:

$$q_{ij}^{a} = \alpha_{ij} f_{j}^{a} \prod_{k \in \mathcal{M}(j) \backslash i} r_{kj}^{a} \tag{2.12}$$

where $\mathcal{M}(j)$ is the set of indices of the check nodes c_i and the f_j^a is the prior probability that v_i in state a. The α_{ij} is the normalized constant ensure that $\sum_a q_{ij}^a = 1$.

Decoding Decision:

After updating the messages r and q for possible states a, at each index $j = 1, ..., n$, the quantity is calculated accordingly:

$$\hat{n_i} = \arg\max_{a} f_{j}^{a} \prod_{k \in \mathcal{M}(j)} q_{kj}^{a} \tag{2.13}$$

where $\hat{n_i}$ is the decoding decision. If this satisfied the syndrome equation $\mathbf{H}^T \hat{n} = \mathbf{s}$, then the decoding is terminated and declared a success. Otherwise the decoding keep decode iteratively until it reached the maximum number of iteration and declare either success or failure [24].

2.6 Non-Binary LDPC Codes: Log-Domain Fast Fourier Transform Decoding

A binary LDPC can be represented using a sparse binary matrices or corresponding Tanner graph. Binary LDPC codes can be generalized by using the same Tanner graph; however, some changes are made such that making the variable nodes v_i take the values from some finite alphabet and making the check nodes c_i impose some complex constraints than the binary LDPC codes [24]. Defining the variable nodes v_i over the finite filed GF(q) required the check c_i as follows:

$$\sum_{j \in \mathcal{N}(i)} f_{ij}(v_j) = 0 \tag{2.14}$$

where $\mathcal{N}(i)$ is the set of variable nodes connected to check i, and each f_{ij} is one of the (q-1)! permutations of the field elements [24].

Non-binary LDPC codes are defined over the 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 [25].

The channel messages are loaded into the variable nodes v_i. These messages represent the prior f^a for the state a, where $a \in GF(q)$.

$$f^a := \prod_{i=1}^{p} f^a_{v_i} \tag{2.15}$$

This means that the decoder interprets p bits (v_1, ..., v_p) from the channel as single q-array symbol and sets the prior distributions for that symbol. The prior $f^a_{v_i}$ is the likelihood the ith bit of variable (v_i) is equal to a_i. such that the set (a_1, ..., a_p) is the binary representation of the symbol a [24].

The complexity of decoding of non-binary LDPC is raised due to the large number of convolution operations and multiplication [26]. These complexity can be reduced by introducing Fast Fourier Transform (FFT) and logarithmic domain. FFT is used to convert the large number of convolution operations into multiplication operations in frequency domain, while the log-domain is used to convert the multiplication of the original FFT algorithm into addition and a lookup table [27].

Similarly with binary LDPC the decoding can be divided into four steps.

Initialization:

The variable node message is set from q^a_{ij} to f^a_j; where $q^a_{ij} = [q^0_{ij}, ..., q^{q-1}_{ij}]$. Then, the log-Domain initialized its information F^a_j such that:

$$F_j^a = \log\left(f_j^a\right)$$

$$hence; F_j^a = Q_{ij}^a \tag{2.16}$$

$$and; Q_{ij}^a = \log\left(q_{ij}^a\right)$$

Updating Check Nodes Messages R_{ij}^a:

In this process the probability information in the log-domain should be converted to exponential domain, as follow:

$$q_{ij}^a = \exp\left(Q_{ij}^a\right) \tag{2.17}$$

Then, this is combined with $q_{ij}^a = [q_{ij}^0, ..., q_{ij}^{q-1}]$, to obtain:

$$u_{ij}^a = \alpha_{ij}\mathcal{FFT}(\mathbf{P}_{h_{ij}} q_{ij}^a) \tag{2.18}$$

where $\mathbf{P}_{h_{ij}}$ is the permutation matrix of message propagation from variable nodes v_i. The term α_{ij} is normalized term to ensure $\sum_{a\in(0,q-1)} q_{ij}^a = 1$. Then, log-domain applied to convert the multiplication to additions, and inverse of FFT is used to update the check message in the state a as follows:

$$U_{ij}^a = \log\left(u_{ij}^a\right) \tag{2.19}$$

$$R_{ij}^a = \mathbf{P}_{h_{ij}}^{-1}\mathcal{IFFT}(\sum_{k\in\mathcal{N}(i)\setminus j} U_{ik}^a) \tag{2.20}$$

where $\mathbf{P}_{h_{ij}}$ is the permutation matrix of message propagation from check nodes c_i to variable nodes v_i.

Updating Variable Nodes Messages Q_{ij}^{a}:

This process of updating variable nodes only used log-domain such that,

$$Q_{ij}^{a} = F_{j}^{a} + \sum_{k \in \mathcal{M}(j) \backslash i} R_{kj}^{a} \tag{2.21}$$

Decoding Decision:

After updating the messages R and Q for possible states a or iteration, each variable node v_i needs to be hard decision after it updated again to declare whether the decoding is successful or failure.

$$\hat{y_i} = \arg\max_{a}(F_{j}^{a} + \sum_{k \in \mathcal{M}(j)} R_{kj}^{a}) \tag{2.22}$$

If the check equations are satisfied the decoding process is finished and it should declare the state of decoding algorithm. If not, the process continues with a new iteration until a valid message word is estimated or the maximum number of iterations is reached out [27], [28].

2.7 Quasi-Cyclic LDPC Codes: Log-Domain Fast Fourier Transform Decoding

Quasi-Cyclic LDPC (QC-LDPC) codes are a special class of LDPC codes with a structured H matrix, which can be generated by the expansion of a $Z \times Z$ base matrix. For WiMax $\mathbf{H}$ matrix, each square box with I matrix represents an $Z \times Z$ circularly right-shifted identity matrix with a shifted value of x, and each empty box represents an $Z \times Z$ zero matrix (null-sub-matrix). Similarly for WiFi matrix (IEEE 802.11n) the $\mathbf{H}$ matrix is constructed by the expansion of a $Z \times Z$ base matrix, an example is disused in [4]. QC-LDPC uses Log-Domain Sum-Product (Log-SPA) decoder algorithm algorithm which is massage passing (iterative) algorithm.

For a codeword $\mathbf{c}_n$, and a decoded codeword $\mathbf{c'}_n$, the a posteriori probability (APP) log-likelihood ratio (LLR) is soft information for $\mathbf{c}_n$ and can be defined as $L_j = \log((P_r(\mathbf{c}_n = 0)/P_r(\mathbf{c}_n = 1))$.

Initialization:

L_j is initialized to be the input channel LLR. The v_i-to-c_i (VTC) message Q_{ij} and the c_i-to-v_i (CTV) message R_{ij} are initialized to 0.

Iterative Decoding:

For each v_i, the Q_{ij} is calculated by;

$$Q_{ij} = L_j + \sum_{k \in \mathcal{M}(j) \backslash i} R^a_{kj} \tag{2.23}$$

where $\mathcal{M}(j)\backslash i$ denotes the set of all the checks connected with v_j except c_i. Then, for each c_i, compute the new CTV message R'_{ij} and Δ_{ij} by;

$$R'_{ij} = Q_{ij_1} \boxplus Q_{ij_2} \boxplus ... \boxplus Q_{ij_n}, \tag{2.24}$$

$$\Delta_{ij} = R'_{ij} - R_{ij} \tag{2.25}$$

where $j_1, j_2, ..., j_n \in \mathcal{M}(i)\backslash j$ and $\mathcal{M}(i)\backslash j$ denotes the set of all the checks $c's$ connected with v_i except c_j The $\boxplus$ operation is defined below:

$$\begin{aligned} x \boxplus y &= \text{sign(x)sign(y)} \min(|x|, |y|) + \mathbf{S}(x, y), \\ \mathbf{S}(x, y) &= \log(1 + e^{-|x+y|}) - \log(1 + e^{-|x-y|}) \end{aligned} \tag{2.26}$$

Updating APP and Decoding Decision:

$$L'_j = L_i + \sum \Delta_{ij_i} \tag{2.27}$$

Then, the decoder makes a hard decision to get the decoded bit $\mathbf{c'}_n$ by quantizing the APP value L'_j into 1 and 0, that is, if L'_j <0, then $\mathbf{c'}_n = 1$, otherwise $\mathbf{c'}_n = 0$. The decoding process terminates when the codeword $\mathbf{c'}_n$ satisfies $\mathbf{H}\ \mathbf{c'}_n^T = 0$, or the pre-set maximum number of iterations is reached. Otherwise, go back to iterative decoding step and start a new iteration of decoding.

2.8 CUDA Architecture

Graphics processing units (GPUs) are performing high parallel processing that have been used for solving general purpose computing problems, and triggered the field of general-purpose computing on graphics-processing units (GPGPU). In just a few years, GPUs have evolved into flexible platforms for general computing [29]. Initially, GPUs were programmed by low-level languages which restricted its application as computing workhorses. The release of Cg, a high-level programming language for GPU, facilitated the application of GPU for a general purpose computation [30], [3]. However, Cg is not user-friendly enough, because it requested programmers must have fundamental knowledge on computer graphics for using this high-level programming language. NVIDIA releases the Compute Unified Device Architecture (CUDA) [3], programmers can write codes for both CPU and GPU in a similar way by using the instruction set of CUDA [31]. CUDA is a GPU software development kit proposed by David Kirk and Mark Harris [32]; thus, advantage of CUDA is that it is an extension of the standard C programming language. That is, those who are familiar with the C/C++ programming language can learn how to program in CUDA relatively easily. This technology allows to execute thousands of concurrent threads for the same process, which makes this kind of devices ideal for massive parallel computation. 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.

In this thesis, the hardwere used for CUDA programming is referred to NVIDIA Tesla K40m model. The device specifications are mentioned in NVIDIA website [33].

2.8.1 NVIDIA's Tesla K40m Architecture

The architecture of NVIDIA Tesla K40m has (15) Multiprocessors, each multiprocessor has (192) single precision (SP) thread processors which make up to 2880 CUDA GPU cores. and 12GB GDDR5 memory. It supports Dynamic Parallelism and HyperQ features and include GPU Boost that increased clock speeds to 745 MHz. Each SP can process a block of data with a thread allocation in parallel. However, it is not possible for the CPU and the GPU to share memory space. Thus, the GPU must make a copy of the shared data to its own memory space in advance. If the CPU wants data stored in the memory of the GPU, a similar copy operation must take place. These copy operations incur significant overhead.

To achieve high speed performance in the kernels execution, the study of memories type in CUDA GPU is essential. Device memory can be sorted as read-write per-thread registers, read-write per-thread local memory, read-write per-block shared memory, read-write per-grid global memory, read-only per-grid constant memory and read-only per-grid texture memory. Since the shared memory is embedded on the multiprocessor, it provides a very fast read and write access for threads. Thus, here are some discussion of memory types in CUDA GPU.

Global memory: This is the largest capacity memory in the device which is a read and write memory and can be accessed by all the threads during the application execution. Tesla k40m has 11441 MBytes total amount of global memory. That is the stored data remains accessible between different kernel executions. It is used for the communication channel between host and device. In which, the data is sent from the host the device and sent back to the host after execution, then global memory is processing the data. Global memory can achieve good performance if each continuous thread index accesses continuous

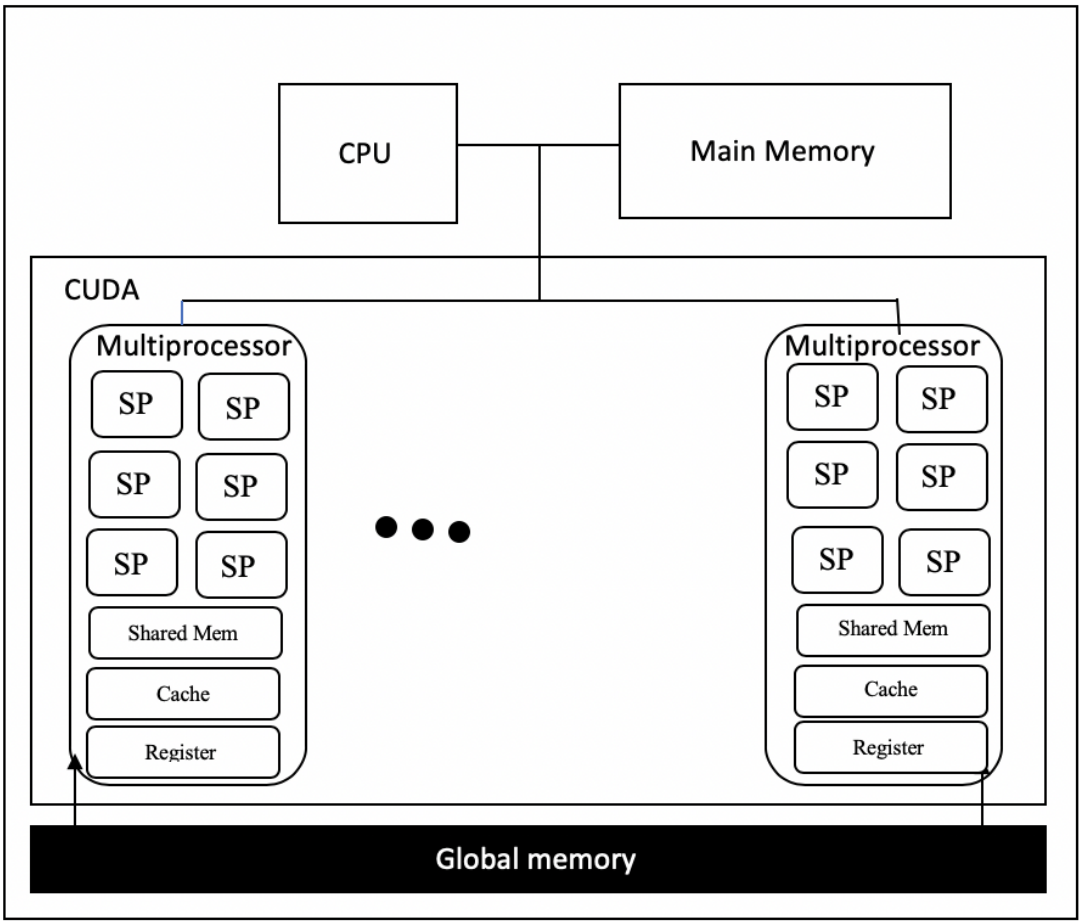

Figure 2.9: GPU Architecture showing multiple stream multiprocessors SM each with SP [2]

memory data positions.

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. Tesla k40m has 65536 bytes of constant memory that performs with short-latency and high-bandwidth when all threads simultaneously access the same location.

Shared memory: This is a read and write memory that is available to all the threads in one block. 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. Tesla k40m has 49152 bytes total amount of shared memory per block.

Registers: These are available to each independent thread; where the total number of threads per multiprocessor in Telsa k40m is 2048. 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 total number of registers per block in Telsa k40m is 65536 registers.

The variables created in CUDA programing can be allocated in each of the available memory types, depending on which keyword precedes the variable declaration or where are they declared.

2.8.2 Threads Organization and Execution

A Tesla k40m GPU has a 15 number of streaming multiprocessors (SM). Each SM is able to execute concurrently up to 2048 threads which arranged into blocks. A block contains up to 1024 threads, arranged with an 1D, 2D or 3D distribution [34].

The relation between a block and a thread in the GPU illustrated in Figure 2.11. A kernel function is executed for one thread at a time. The blocks are partitioned in grids, such that 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 [34]. Hence, the execution for one grid of is lunched by each kernel. When a function is invoked, the thread and the block index are identified by the

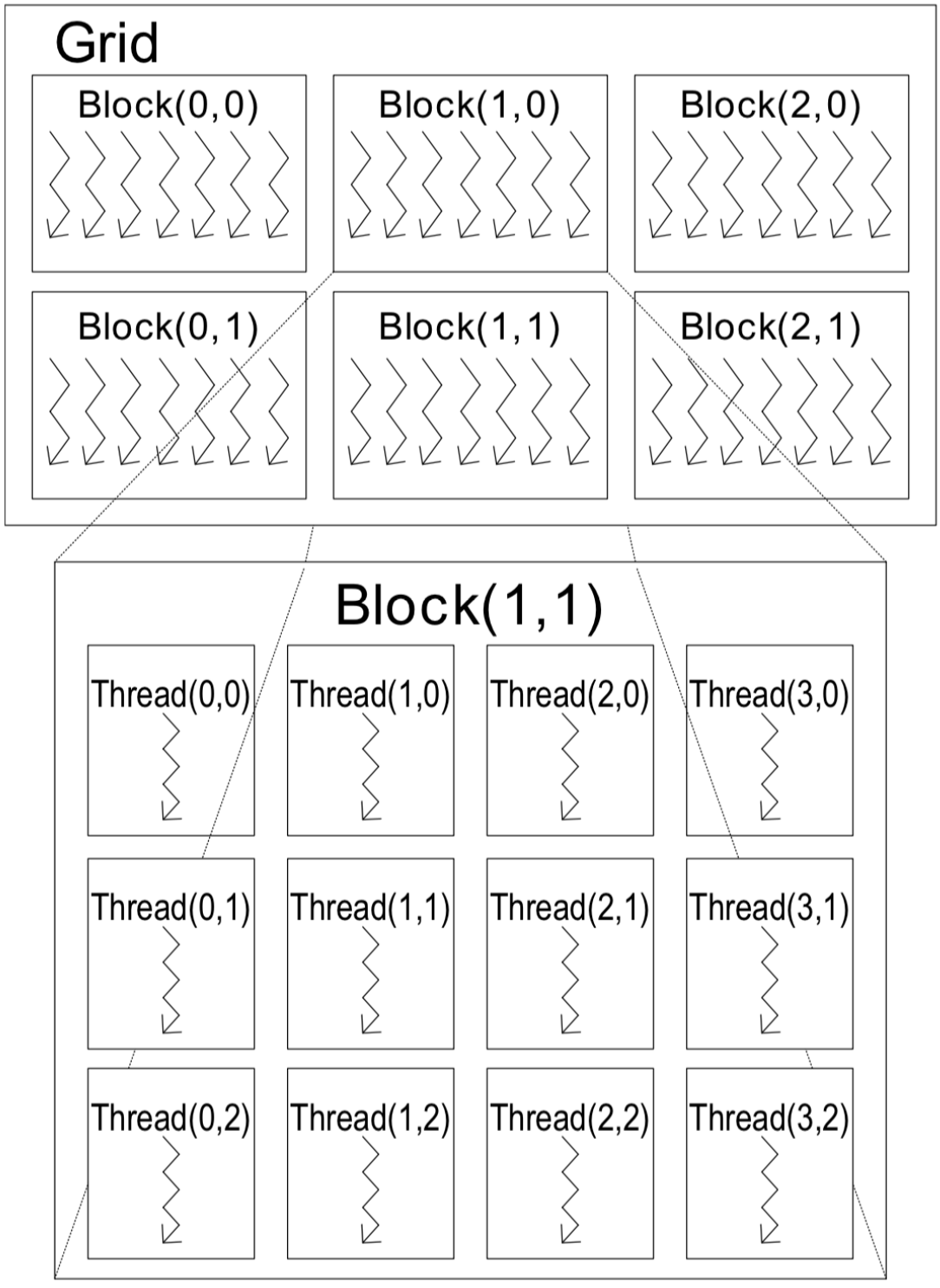

Figure 2.10: Blocks and threads arrangements for a grid [2]

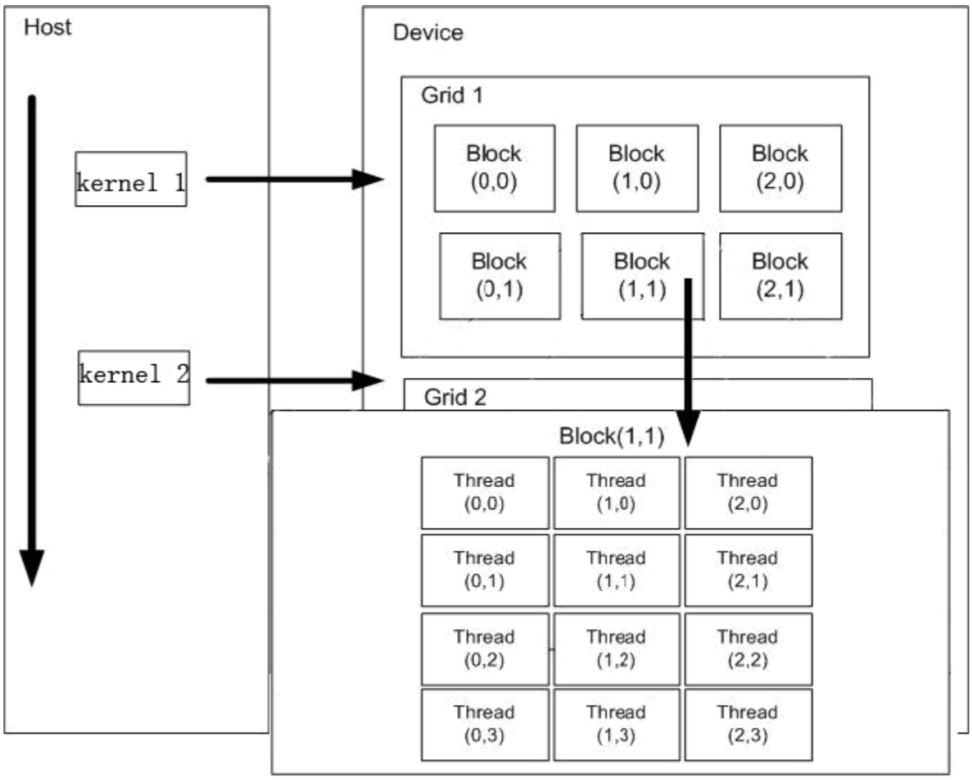

Figure 2.11: Thread batching: the host issues a succession of kernel invocations to the device (CPU host). Each kernel is executed as a batch of threads organized as a grid of thread blocks [3]

'thread_idx' and 'block_idx' variables, respectively [34].

A kernel is launched as a $C/C++$ function with some extensions from the host and executed on the device. The kernel is called by using *__global__kernelName <<<numGrids, numBlocks >>>(parameters_ list);* where, *__global__* which indicates to the compiler that the kernel has to be executed on the following device *<<<numGrids, numBlocks >>>*.

The blocks/grid, threads/block settings, and parameters_ list are the typical parameters that passes to a standard $C/C++$ function. CUDA programming manual is provides more detailed information for how CUDA is used in regular $C/C++$ program [34].

Chapter 3: Implementation

3.1 Log-Domain FFT decoding based LDPC Codes: CPU Implementation

The LDPC decoder algorithm has been implemented using C language by Takamura [35]. The implementation is based on Log-domain FFT for $GF(2^m)$ field. A lookup table is created for each $GF(q)$ where q is the range from $2^1, ..., 2^8$. The lookup table is used to reduce the complexity of the FFT multiplication operations. such that the addition and subtraction operations are defined in GF(q). The input to the simulation is **H** matrix that is constructed by MacKay [36].

3.1.1 Performance and Results

The obtained results are related to the hardware architecture used in the simulations and will differ if different CPUs are used. For the experiments of work by Takamura [35], the following hardware and softwere standard were used on Ubuntu 16.04 operating system:

- CPU: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz,
- GCC compiler version 5.4.0

The performance of the decoder shows that the LDPC decoder was successfully decoded the regular MacKay [36] code; while the decoder fails when the parity-check matrix is constructed in a different way.

Table 3.1 shows the latency (in millisecond) of the decoding algorithm that is implemented on CPU with 2.20 GHz speed. The table shows that as the q symbols for the GF field increase the decoding takes more time to decode the message. The block size of the

Table 3.1: GF(q) For Log-Domain FFT decoding based LDPC on CPU

GF(q)	Code rate	Dimension	Time in msec	number of iterations
$GF(2^1)$	0.5	20000x10000	313	27
$GF(2^2)$	0.67	9000x6000	66	7
$GF(2^3)$	0.66	6000x4000	448	6
$GF(2^4)$	0.66	9000x6000	348	9
$GF(2^5)$	0.66	9000x6000	728	9
$GF(2^6)$	0.66	9000x6000	1846	11
$GF(2^7)$	0.66	6000x4000	17872	71
$GF(2^8)$	0.66	6000x4000	18832	84

code is affecting the decoding time as the binary LDPC decoder over $GF(2)$ is shown in Table 3.1 has larger block size, thus it took longer time to decode than over $GF(2^2)$ with smaller block size of $9000x6000$. Belief-Propagation algorithm is iterative decoding algorithm, thus the decoder here illustrated for NB-LDPC that the iteration increased when we have more symbols in the code.

3.2 Log-Domain FFT decoding based LDPC Codes: GPU Implementation

LDPC is implemented using CUDA programming language supporting multiple standards such as GF(q) and data rates using multiple H-matrices that are stored as files. The host CPU *'__device__'* reads the H-matrix for a given standard. The host CPU then generates an address table of data processed in parallel by the GPU. Generation of the address table is parallelized by CUDA functions. Next, generated address information is transferred to the memory in the GPU *'__global__'* . This copy operation takes place only if there is a change in standard or code rate.

When signals from the channel are received, the host CPU delivers them to the GPU. The GPU executes the proposed LDPC decoding software in parallel. Upon completion of the decoding, decoded bits are transferred to the host CPU. A CUDA API called *'cudaMemcpy()'* is used to exchange data between the host and the GPU. The copy overhead

may be significant, so it is crucial to minimize it. In this log-Domain FFT based LDPC implementation the copy operation takes place only for generated address transfers, received signal transfers, and decoded bit transfers.

3.2.1 LDPC Memory Arrangement on GPU

The most important structures are Q_{ij}^a and R_{ij}^a messages which are passed between variable nodes and check nodes and the posteriors F_j^a. All of these are accessed very frequently in most kernels of the main functions. LDPC codes of different sizes and different Galois field dimensions should be supported, such that the potential required amount of memory should be considered for these message vectors.

Although the data is stored as a one-dimensional array in the device, it is interpreted as 2D F_j^a or 3D Q_{ij}^a and R_{ij}^a structures. This depends on the indexing as illustrated in Figure 3.1; where for the 3D arrays case, the first index is pertinent to the node, the second to the node connection and the third to the Galois field element that corresponds to one entry of a prior vector [3]. While 2D arrays are indexed first by the node and second by the Galois field element. This indexing points to the exact position of one element in the structure, and helps to set the CUDA grid and block configuration of the kernels [37]. Therefore, multiple float values are handled in only one read/write memory access which allows accessing the memory in a coalesced way and helps in boosting the performance.

A GPU's texture memory is employed and available for all kernels for Galois field arithmetic operation; such as 2D lookup tables for for the addition and multiplication, and 1D lookup tables for the inversion (size q-1) and conversion between exponential and decimal representation (size q).

3.2.2 LDPC Codes On GPU

Initialization:

The number of iterations is initialized to zero. Then, the input data is transferred to the CUDA device to do the initial check of the parity equations. If they are not fulfilled, the

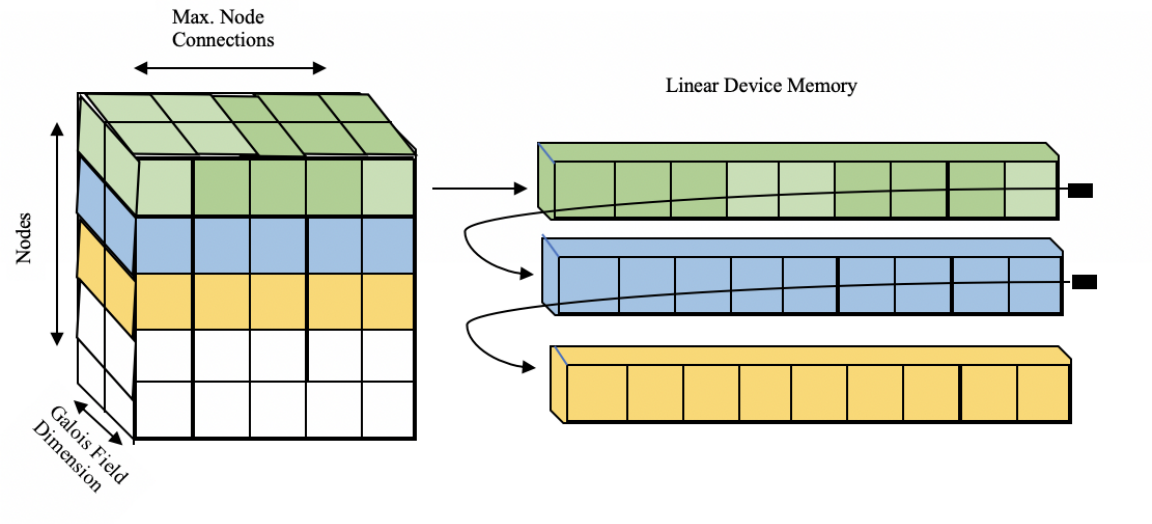

Figure 3.1: 3D structures used for message vectors and correspond to linear data allocation in the global device memory as shown in [3]

main decoding loop is executed until they are, or until a maximum number of iterations (1000) is reached. After the iterative decoding process finishes, a hard decision is conducted to obtain the final decoding result.

Permute Message Vectors:

Before and after the check node update R_{ij}^a, the elements of the message F_{ij}^a vectors have to be permuted according to the Galois field element at the corresponding check matrix position. A shared memory block is used as a temporary memory for parallel implementation of the permute operation. For R_{ij}^a to be permuted, the shared memory stores the input data using the permuted indices according to the matrix entry $\mathbf{H}_{i,j}$, and writes the output data directly. For Q_{ij}^a to be the permuted, the shared memory reads the input values directly, but the output is written using the permuted indices [as illustrated in [38] fig. 4]. In this way, one thread per node and per GF(q) element can employed without accessing the same shared memory banks. Therefore, no bank conflicts are generated, resulting in full speed memory access [25].

Log Domain FFT:

The implementations of the log-domain FFT and the inverse IFFT domain are identical and follow a butterfly diagram [25] [as shown in [38] fig. 5]. The addition operations carried out in the log-domain which consists of two separate operations for the magnitude and the sign. Thus, two shared memory blocks are required for this purpose. Both operations are executed simultaneously in the CUDA kernel implementation. The log-domain FFT is applied independently to each F_{ij}^{a} vector of size q. Thus, the CUDA block width is fixed to q; for example in case of $GF(2^8)$ the CUDA block size equals to 256. For each node (variable and check), a loop processes all node connections to compute the log-domain FFT of the F_{ij}^{a} vector elements. The execution of the butterfly is supervised by a small loop of $\log_2(q)$ iterations. Since each continuous thread accesses a different shared memory bank during the butterfly, no bank conflicts are produced. The exponential and logarithm functions that are needed for the log-domain operations [25] are realized by the fast hardware implemented functions which are provided by the GPU.

Check Node Update R_{ij}^{a}:

The implementation uses one thread per node and GF(q) element to compute the outgoing message F_{ij}^{a} vectors for each edge of the corresponding check node [as visualized in [38] fig. 6]. The computations that performed in the log domain are separated in two threads. In the first, loop over the node's edges calculates the total sum of the magnitude values and the total product of the sign values. In the second loop, the current magnitude value is subtracted from the total magnitude sum to acquire the extrinsic sum, and the total sign is multiplied by the current sign value for each edge.

Variable Node Update Q_{ij}^{a}:

The kernel processes one thread per variable node Q_{ij}^{a} and GF(q) dimension [as shown in [38] fig. 7]. An internal loop processes all the node connections within one thread. Each thread first reads the input F_{ij}^{a} into a shared memory block and then adds all extrinsic information from R_{ij}^{a} for the current edge and result is written into Q_{ij}^{a}. Since the shared memory operations are inside the loop over the node connections, continuous thread numbers access

different shared memory banks in each iteration. This avoids bank conflicts and provides maximum performance for the shared memory.

Update Posteriors:

The implementation of this kernel follows the one for the variable node update and the information of all edges rather than all but one is added to the input F_j^a to yield the posterior F_{ij}^a vectors Q_j^a.

Evaluate Check Equations:

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

3.2.3 Performance and Results

The results is obtained using specific hardware architecture and software tools, the hardware and software specifications are listed below:

- CPU: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz,
- GPU: Telsa km40, 2880 CUDA cores, 12GB RAM,
- CUDA tools kit v9.1

The performance of this implanted algorithm is of this using the same parity-check matrix as it used for the decoder implemented on CPU. The following Table 3.2 illustrates the latency of the decoder that increases as q increased. The performance is boosted by mainly the fact that the elements q of GF field are stored in continuous memory positions, which leads to boost the efficiency of the coalesced memory access.

Table 3.2: GF(q) For Log-Domain FFT decoding based LDPC on GPU

GF(q)	Code rate	Dimension	Time in msec	number of iterations
$GF(2^1)$	0.5	20000x10000	37.3	7
$GF(2^2)$	0.67	9000x6000	27	5
$GF(2^3)$	0.66	6000x4000	44.50	6
$GF(2^4)$	0.66	9000x6000	48	7
$GF(2^5)$	0.66	9000x6000	83.4	9
$GF(2^6)$	0.66	9000x6000	166.3	9
$GF(2^7)$	0.66	6000x4000	178	20
$GF(2^8)$	0.66	6000x4000	186	28

3.3 Quasi-Cyclic LDPC: GPU Implementation

QC-LDPC was implemented by [4] using standard parity-check matrix for WiFi (IEEE 802.11n) and WiMax (IEEE 802.16e).

3.3.1 Mapping QC-LDPC Decoding Algorithm to GPU Kernels

The following equations (2.24), (2.25), and (2.27) makes the decoding process to be split into two stages: the horizontal processing stage and the APP update stage; in which one computational kernel is created for each stage. This kernel runs in GPU, while the host code running in the CPU which is responsible of the CUDA initialization and memory copy between host and device [4].

Horizontal Processing: CUDA Kernel 1

In this stage, and since all the CTV messages are calculated independently, a kernel is created that uses many parallel threads to process these CTV messages. For an M x N H matrix, M threads are generated, and each thread processes a row. Since all non-zero entries in a sub-matrix of H have the same shift value, threads processing the same layer have almost exactly the same operations when calculating the CTV messages M_{sub} [4].

APP value update: CUDA Kernel 2

In this stage, there are N APP values to be updated. Similarly, the APP value update is independent among variable nodes v_i. Thus, N_{sub} thread blocks are used, with Z threads in each thread block. In this stage, Kernel 2 makes a hard decision for each bit.

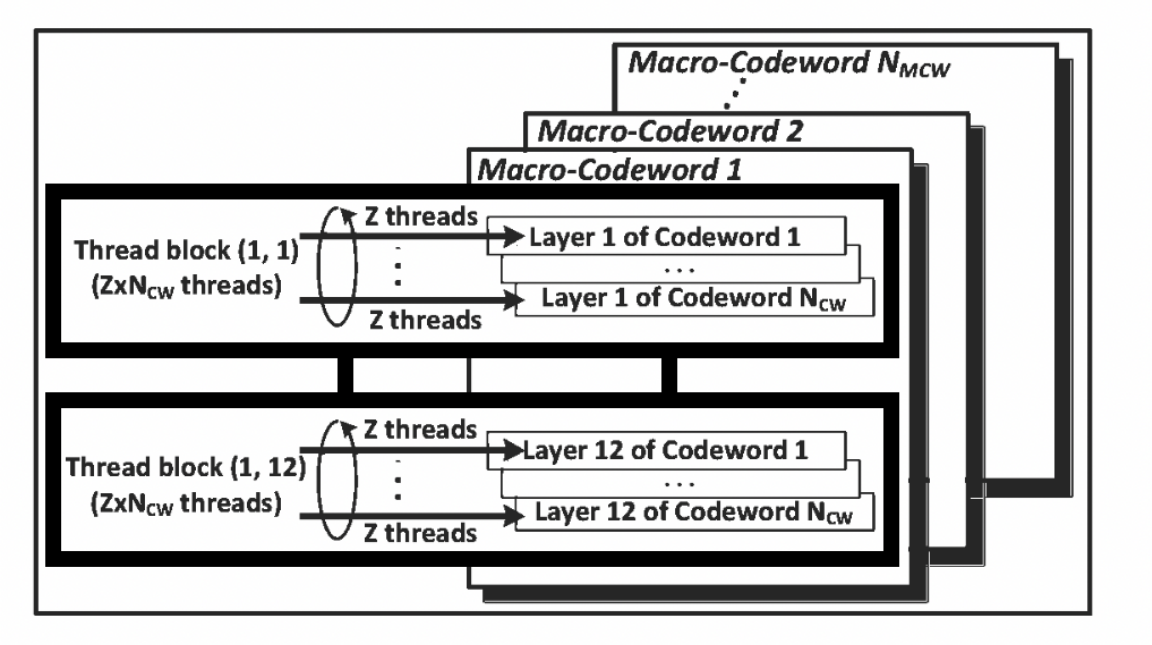

Figure 3.2: Multi-codeword parallel decoding algorithm by [4]. N_CW represents the number of codewords in one Macro-codeword (MCW). N_{MCW} represents the number of MCWs.

3.3.2 Multi-codeword Parallel Decoding

A two-level multi-codeword scheme is designed to further increase the parallelism of the workload. N of codewords are first packed into one macro-codeword (MCW). Each MCW is decoded by a thread block and N_{MCW} MCWs are decoded by a group of thread blocks. The multi-codeword parallel decoding algorithm is described in Figure 3.2. Since multiple codewords in one MCW are decoded by the threads within the same thread block, all the threads follow the same execution path. Also, the latency of read-after-write dependencies and memory bank conflicts can be completely hidden by a sufficient number of active threads [4].

3.3.3 Implementation of Early Termination Scheme

Wang and his colleagues [4] proposed an early termination (ET) algorithm that is used to avoid unnecessary computations when the decoder already converges to the correct codeword. A new CUDA kernel with M threads is launched and each thread calculates one parity check equation c_i independently. The parity check results are used by all the threads, and on-chip shared memory is used to speed up the memory access. After the concurrent threads finish computing the parity check equations, these threads are re-used to perform a reduction operation on all the parity check result to generate the final ET check result. This ET results indicates the correctness of the codeword.

Wang and his colleagues [4], also, proposed a tag-based ET for multi-codeword parallel decoding. In which, one tag is assigned per codeword and the tag is marked once the corresponding parity check equation is satisfied. Then, if the tags for all the codewords are marked, the iterative decoding process is terminated.

3.3.4 Optimizing Memory Access on GPU

The latency of memory access is one of the main bottlenecks which limits the performance of the LDPC decoder. Two memory access optimization techniques are employed to further increase the throughput [4].

Memory Optimization for H Matrix:

Reading from the constant memory is as fast as reading from a register as long as all the threads within a half-warp read the same address. Since all the Z threads in one thread block access the same entry of the H matrix simultaneously, the H matrix is stored in the constant memory and used in the broadcasting mode that is offered by constant memory.

The quasi-cyclic characteristic of the QC-LDPC code allows us to efficiently store the sparse H matrix. For WiFi parity-check matrix shown in Figure 3.3, the $\mathbf{H}_{kernel1}$ and $\mathbf{H}_{kernel2}$ are compacted by compressing $\mathbf{H}$ horizontally and vertically, respectively (see in Figure 3.4). This results in reducing the device memory usage, therefore, the time spent on reading the H matrix from device memory is reduced. Moreover, the number of branch

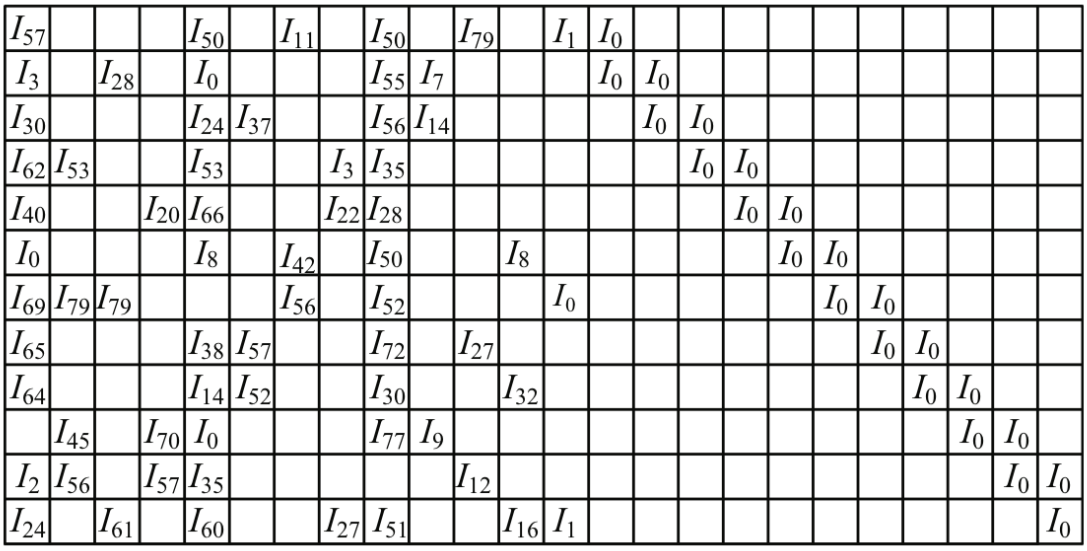

Figure 3.3: Example of Parity-check matrix for WiFi IEEE 802.11n (1944, 972) LDPC code. **H** consists of M_{sub} x N_{sub} = 12x24 [4]

instructions which may cause throughput degradation are also reduced since there is no need to check whether an entry of H is empty.

Coalescing Device Memory Access:

In CUDA kernel 1, R_{ij} and Δ_{ij} values are stored in the device memory. There are only one R_{ij} and one Δ_{ij} values per row in each sub-matrix of **H**. Thus, the compressed format can be used to store R_{ij} and Δ_{ij}. This results in total, the memory saving forR_{ij} and Δ_{ij} is more than halved. Therefore, the compressed R_{ij} and Δ_{ij} matrices are written column-wise, and all memory accesses to R_{ij} and Δ_{ij} are coalesced.

3.3.5 Performance and Results

The obtained results are using the same hardware architecture as for Log-Domain FFT algorithm. The simulations of QC-LDPC is tested on the following standard hardware and graphic card were used for Log-Domain FFT simulation:

- CPU: Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz,

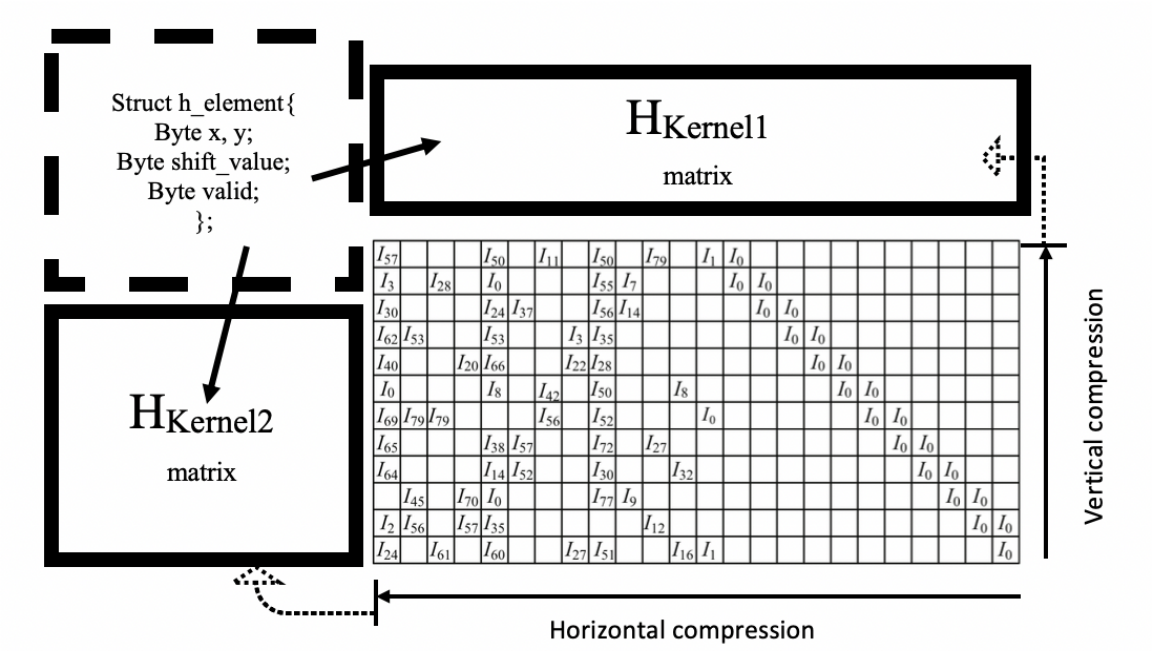

Figure 3.4: Using the **H** matrix example in Figure 3.3 by [4], after the horizontal and vertical are compressed, $\mathbf{H}_{kernel1}$ and $\mathbf{H}_{kernel2}$ are produced, respectively. Each entry of the compressed **H** matrix contains 48-bit data indicating the row and column index of the element in the original **H** matrix. The shift value and a valid flag which shows whether the current entry is empty or not.

Table 3.3: Decoding Latency and Throughput for QC-LDPC

Codes	**Code rate**	**Dimension**	GPU_{Time} **in msec**	CPU_{Time} **in msec**	**Throughput**
$WIFI(802.11n)$	0.5	1944x972	1059.7	7511.6	196.30
$WIMAX(802.16e)$	0.5	2304x1152	1057.6	7488.3	196.32

- GPU: Telsa km40, 2880 CUDA cores, 12GB RAM,
- CUDA tools kit v9.1

The implementation of this algorithm focuses on improving the throughput. The throughput is calculated as $Throughput = (N_{bits}xN_{Sim}xN_{codeword})/T_{total}$, where N_{bits} is codeword length, $N_{codeword}$ is the total number of the codewords, and N_{Sim} is the simulation number, these parameters are divided by T_{total} which is the running time. Simulation on the Tesla k40m shows that around 1280 codewords is parallelized.

According to the capacity of Tesla k40m GPU, around 1280 codewords are processed in parallel in the multi-codeword decoding scheme ($N_{codeword} = 1280$). Table 3.3 shows the throughput and the latency of [4] implementation for both the WIFI (802.11n) code and WiMAX (802.16e) code. The results also show that the decoder for the WiMAX (802.16e) code has almost the same throughput compared to the WIFI (802.11n) code. Similar for the latency of the decoding algorithm, the latency is almost the same for both application on GPU, while on the CPU the WIMAx application have slightly higher latency, this is due to longer codeword of the WIMAX compared to that of WIFI.

3.4 Results and Comparisons

In this section the performance of Log-Domain FFT based LDPC decoder on CPU is compared with the performance of the GPU implementation. The main measurement is the speed of the decoding algorithm for large codeword. Afterward, the performance QC-LDPC that used for application such as WIFI and WIMAX is compared with Log-Domain FFT based LDPC decoder.

3.4.1 Log-Domain FFT based LDPC Performance on GPU vs CPU

The GPU implementation allows to prallelized the serial implementation of Log-Domain FFT based LDPC decoder. Both Table 3.1 and Table 3.2 show that the GPU implementation is faster than the CPU. For the Binary LDPC decoder the GPU implementation is 8 times faster than CPU implementation. While for the NB-LDPC the GPU is almost up to 10 times faster than CPU.

The Figure 3.5 below illustrates the number of iterations for each simulation for Log-domain FFT based LDPC. The CPU have fluctuated iterations line for different GF(q). The block size $6000x4000$ for GF(8), GF(128), and GF(256) has higher iteration numbers compared with block size $9000x6000$. While GPU have smooth increasing of iteration number as q elements increased.

The illustration of the latency of each simulation for Log-domain FFT based LDPC decoder is in Figure 3.6 below. This figure shows that the GPU simulation of the decoder has latency less than one seconds for all elements of q. While the latency of CPU implementation increased dramatically from approximately one second for GF(64) to almost 19 seconds latency for GF(256). The reason for GPU to have maintained the speed of decoder is because of the cascaded memory that each group of q elements is stored in.

3.4.2 QC-LDPC on GPU vs Log-Domain FFT based LDPC Performance on GPU vs CPU

Both applications , WIFI and WIMAX, have similar QC-LDPC decoding latency on GPU which is approximately one second for block length $1944x972$ and $2304x1152$, respectively. While they have approximately 8 seconds decoding latency on CPU. Comparing with Log-Domain FFT on GPU for regular code and different field elements, the latency of LDPC decoder is less than one second. This main conclusion that Log-Domain is faster even though the block length is larger. Figure 3.7 below shows the latency of all algorithms that have been studied.

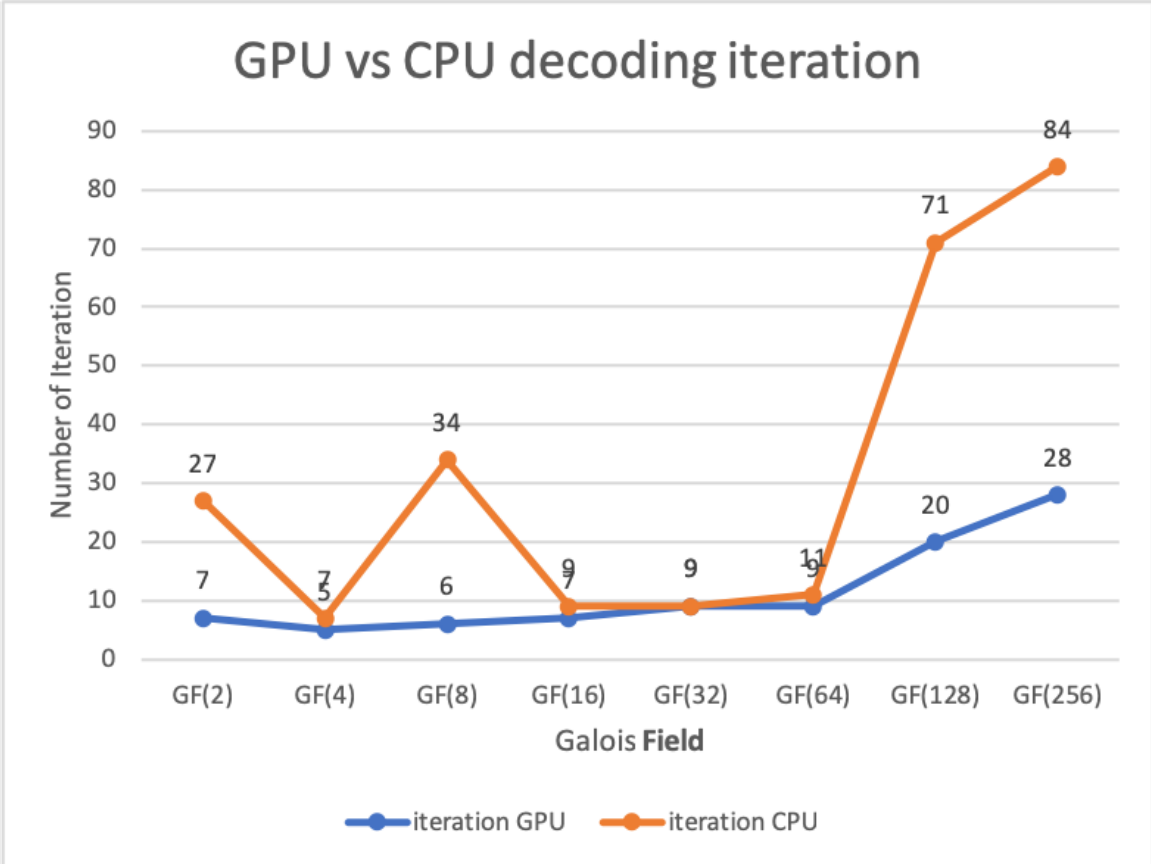

Figure 3.5: Number of iterations Log-domain FFT based LDPC decoder on GPU vs on CPU

,

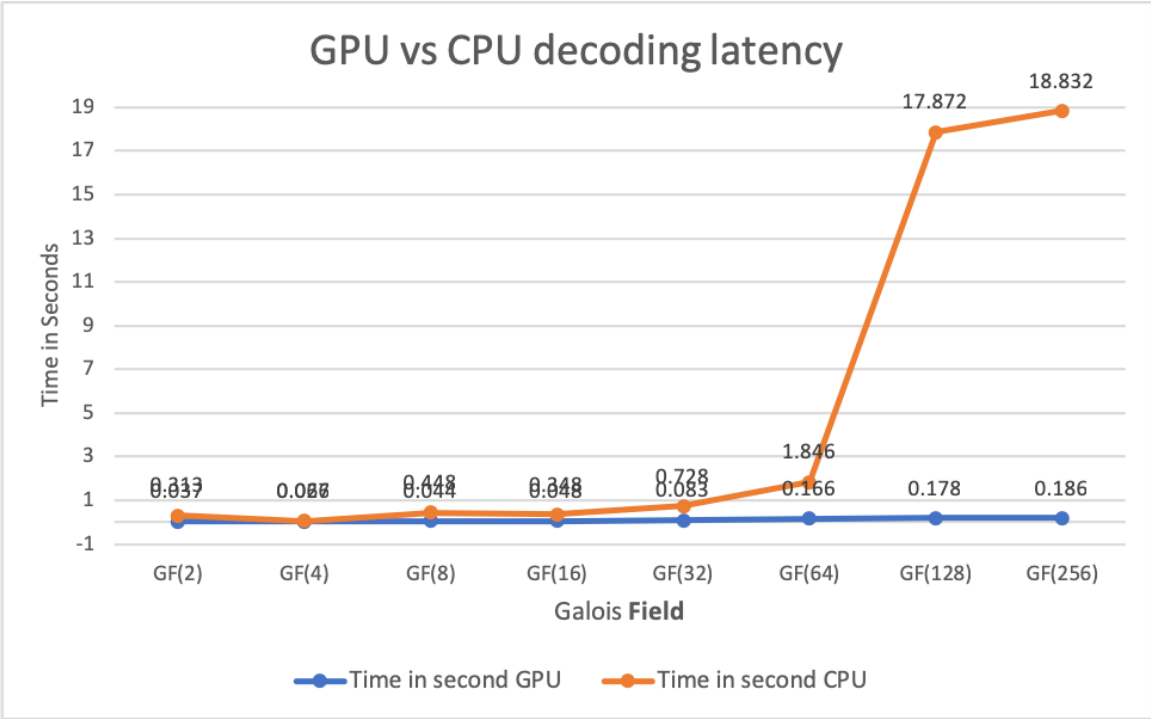

Figure 3.6: Latency of Log-domain FFT based LDPC decoder on GPU vs on CPU

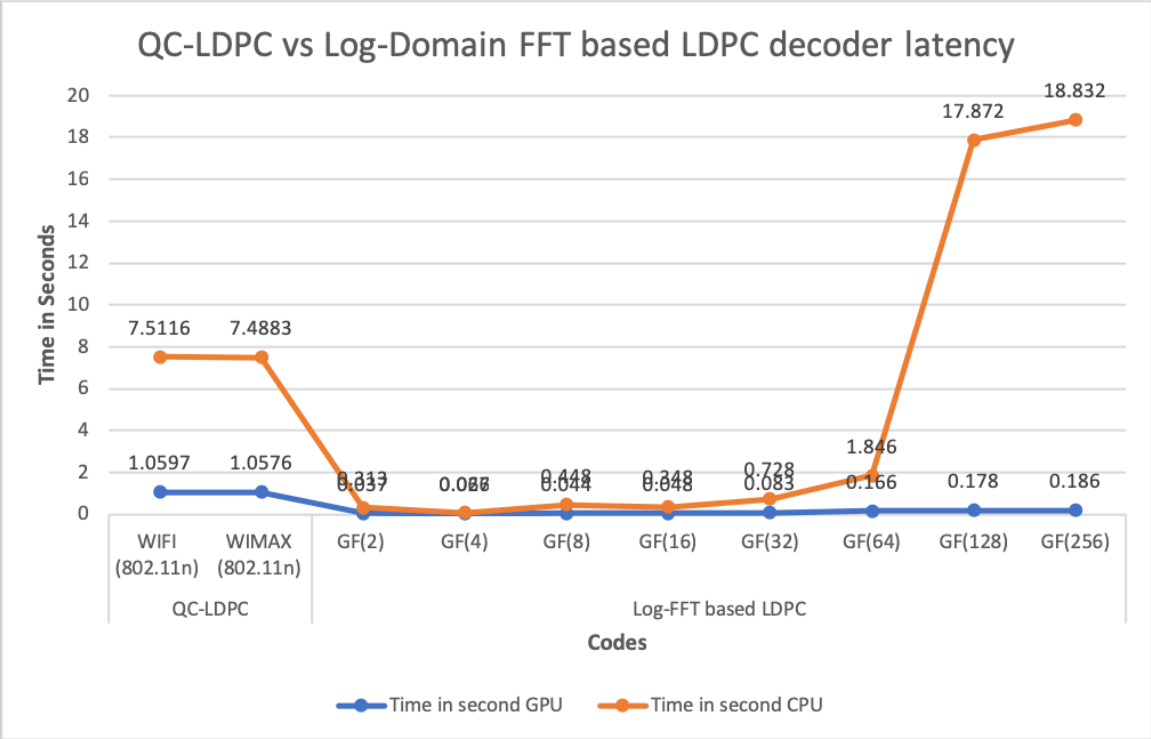

Figure 3.7: Latency of Log-domain FFT based LDPC decoder on GPU vs on CPU

Chapter 4: Summary and Conclusions

This thesis presents the techniques and design methodology to fully utilize a GPU textquotesingles computational resources to accelerate a computation-intensive of LDPC decoding algorithm. The implementation of Log-Domain FFT based LDPC decoder on GPU is presented and has resulted in faster decoding process compared with the case study, the implementation of Log-Domain FFT based LDPC decoder on CPU. Another case study, a massively parallel implementation of LDPC decoder on GPU, is presented. The LDPC decoder is demonstrated with different parity-check matrices constructions, WIFI (IEEE 802.11n), WiMAX (802.16e), and regular MacKay parity-check codes. The simulation results exhibit that our LDPC decoder can achieve high speed around up to 10x faster than CPU and up to 0.186 millisecond. A future implantation for achieving higher speed LDPC decoding algorithm using FPGA is detailed in the following section.

4.0.1 Future Work

LDPC for regular MacKay codes and for WIFI and WIMAX codes can be implemented in hardware such as FPGA (Felid Programmable Gateway). FPGAs have proven to be useful under high load; i.e. they improve the throughput of each server by a factor of 95% for a fixed latency distribution[39]. However, the fact that the computer architecture is largely unconstrained, enables an efficient tailoring of the architecture to the particular application. This cause difficulty to maintain flexibility of architecture design while offering a productive programming. High-level synthesis (HLS) is a tool that supports a high level application description that can be effectively synthesized in efficient hardware, finding that description is a complex design problem requiring solid hardware understanding. While HLS tools have been proven worthy as testing and validation tools, we find them, backed on the designs

herein proposed, sufficiently mature nowadays to synthesize efficient hardware accelerators [40].

HLS tools have greatly improved and are able to provide a convenient environment to speed up the development process and empower the developers to quickly explore a much wider design space. Xilinx Vivado HLS [39] is a state-of-the-art High-level synthesis (HLS) tool that accepts design inputs in C, C++ or SystemC and enables the designer to quickly perform algorithm verification before automatically translating the high-level design input into an Register-Transfer-Level (RTL) implementation. HLS also permits the designer to use compiler directives, when necessary, to guide the tool to explore some architectural options. Due to this high degree of control on the underlying architecture and the low effort of retargeting different devices, we used Vivado HLS to develop the non-binary LDPC decoder [39].

LDPC decoding is a complex signal processing application that offers plenty of parallelism for FPGA-based acceleration. Vivado HLS can be used to develop HLS IP cores that implement the LDPC decoding functionality. However, to realize the FPGA accelerator, one solution is to integrate these cores into a high-level architecture that contains essential system components, such as the interconnection bus and external memory controller, and the clock and control circuitry. This is needed because the storage space required for larger values of m in $GF(2^m)$ can exceed the local storage capacity of typical FPGAs [39].

To design the $GF(2^m)$ LDPC FFT-SPA decoder in two ways HLS is used to design the algorithm, either by proper code writing, such that in the case of an existing C application, the code should be refactored, or by annotating the code with optimization directives. These directives can be inlined with C code using the *' # pragma'* construct, or systematized in a *Tcl* script with equivalent syntax, both of which are mutually exclusive methods to instruct the HLS tool. One method is using **Mapping the FFT-SPA to HLS C** which synthsize the C code to HLS than they will be simulated using Verilog simulator. Then, **Multiplication Kernels** which consists of *'vn_proc' and 'cn_proc'* are the kernels that perform multiplication, each composed of a triple-nested loop structure, number of edges, GF(q) dimension,

and number of elements in variable and check nodes. Afterward, **Permute and De-permute Kernels** is needed to deal with the permutation and de-permutation of probabilities F^a and are described as a double nested loop structure. The *'GF_read'* loop loads data and shuffles it according to the permutation/de-permutation into a local copy. Then, *'GF_write'* loop writes data back contiguously to the correct BRAM location. Last kernel is **FFT Kernel** that implements the FWHT, a special case of the FFT where the twiddle factors are always -1 or 1, thus only additions and subtractions are executed by this kernel.

FPGA implementation using HLS can be further optimized to avoid the decoder to perform sequential steps and leverage the available parallelism process.

Bibliography

Bibliography

[1] I. Gautam, "Parity check matrix and construction methods of Low Density Parity Check code," *International Journal of Innovative Research in Information Security*, vol. 1, no. 3, pp. 29–35, 2014.

[2] J.-Y. Park and K.-S. Chung, "Parallel LDPC decoding using CUDA and OpenMP," *EURASIP Journal on Wireless Communications and Networking*, vol. 2011, no. 1, p. 172, 2011.

[3] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *2008 42nd Asilomar Conference on Signals, Systems and Computers*. IEEE, 2008, pp. 171–175.

[4] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *2011 IEEE 9th Symposium on Application Specific Processors (SASP)*. IEEE, 2011, pp. 82–85.

[5] R. Gallager, "Low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, 1962.

[6] B. Sklar and F. J. Harris, "The ABCs of linear block codes," *IEEE Signal Processing Magazine*, vol. 21, no. 4, pp. 14–35, 2004.

[7] B. Vucetic and J. Y. C. Principles, "Applications' Boston," 2000.

[8] L. Hardesty, "Explained: Gallager codes," Jan 2010. [Online]. Available: http://news.mit.edu/2010/gallager-codes-0121

[9] D. J. MacKay and R. M. Neal, "Good codes based on very sparse matrices," in *IMA International Conference on Cryptography and Coding*. Springer, 1995, pp. 100–111.

[10] T. K. Moon, *Error Correction Coding: Mathematical Methods and Algorithms*. New York, NY, USA: Wiley-Interscience, 2005.

[11] Y. Akhtman, R. G. Maunder, and L. Hanzo, "An Approximate Coding-Rate Versus Minimum Distance Formula for Binary Codes," *arXiv preprint arXiv:1206.6584*, 2012.

[12] R. Shedsale and N. Sarwade, "A review of construction methods for regular ldpc codes," *Indian Journal of Computer Science and Engineering (IJCSE) Vol*, vol. 3, pp. 380–385, 2012.

[13] O. O. Khalifa, S. Khan, M. Zaid, and M. Nawawi, "Performance evaluation of low density parity check codes," 2008.

[14] D. J. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE transactions on Information Theory*, vol. 45, no. 2, pp. 399–431, 1999.

[15] G. Falcao, V. Silva, J. Marinho, and L. Sousa, "LDPC decoders for the WiMAX (IEEE 802.16 e) based on multicore architectures," in *WIMAX New Developments*. IntechOpen, 2009.

[16] T. Brack, M. Alles, F. Kienle, and N. Wehn, "A synthesizable IP core for WIMAX 802.16 e LDPC code decoding," in *2006 IEEE 17th international symposium on personal, indoor and mobile radio communications*. IEEE, 2006, pp. 1–5.

[17] H. Zhong and T. Zhang, "Block-LDPC: A practical LDPC coding system design approach," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 52, no. 4, pp. 766–775, 2005.

[18] Z. Cai, J. Hao, P. Tan, S. Sun, and P. Chin, "Efficient encoding of IEEE 802.11 n LDPC codes," *Electronics Letters*, vol. 42, no. 25, pp. 1471–1472, 2006.

[19] B. K. Butler, "Minimum distances of the QC-LDPC Codes in IEEE 802 Communication Standards," *arXiv preprint arXiv:1602.02831*, 2016.

[20] E. B. Sudderth and W. T. Freeman, "Signal and image processing with belief propagation," *IEEE Signal Processing Magazine*, vol. 25, no. 2, p. 114, 2008.

[21] S. B. Wicker and S. Kim, *Fundamentals of codes, graphs, and iterative decoding*. Springer Science & Business Media, 2006, vol. 714.

[22] J. Zhang and M. Fossorier, "Shuffled belief propagation decoding," in *Signals, Systems and Computers, 2002. Conference Record of the Thirty-Sixth Asilomar Conference on*, vol. 1. IEEE, 2002, pp. 8–15.

[23] S.-Y. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Communications letters*, vol. 5, no. 2, pp. 58–60, 2001.

[24] M. C. Davey, "Error-correction using low-density parity-check codes," Ph.D. dissertation, University of Cambridge, 2000.

[25] G. J. Byers and F. Takawira, "Fourier transform decoding of non-binary LDPC codes," in *Proceedings Southern African Telecommunication Networks and Applications Conference (SATNAC). Spier Wine Estate, Western Cape, South Africa*, 2004.

[26] J. Patel, N. Chapatwala, and M. Patel, "FFT based sum product decoding algorithm of LDPC coder for GF (q)," in *2014 2nd International Conference on Emerging Technology Trends in Electronics, Communication and Networking*. IEEE, 2014, pp. 1–4.

[27] Z. Wang, J. Meng, Z. Deng, L. Zhang, and J. Gao, "FPGA Implementation Scheme of Mixed Logarithmic Domain FFT-BP Decoding Algorithm Based on Non-Binary LDPC Codes," in *2018 37th Chinese Control Conference (CCC)*. IEEE, 2018, pp. 8459–8464.

[28] S. Aruna and M. Anbuselvi, "FFT-SPA based non-binary LDPC decoder for IEEE 802.11 n standard," in *2013 International Conference on Communication and Signal Processing*. IEEE, 2013, pp. 566–569.

[29] C. Nvidia, "Compute unified device architecture programming guide," 2007.

[30] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Transactions on Graphics (TOG)*, vol. 22, no. 3, pp. 896–907, 2003.

[31] R. Shams and N. Barnes, "Speeding up mutual information computation using NVIDIA CUDA hardware," in *9th Biennial Conference of the Australian Pattern Recognition Society on Digital Image Computing Techniques and Applications (DICTA 2007)*. IEEE, 2007, pp. 555–560.

[32] D. Kirk *et al.*, "NVIDIA CUDA software and GPU parallel computing architecture," in *ISMM*, vol. 7, 2007, pp. 103–104.

[33] "NVIDIA Tesla Supercomputing Solutions." [Online]. Available: https://www.nvidia.com/en-us/data-center/tesla/

[34] C. Nvidia, "Cuda c programming guide, version 9.1," *NVIDIA Corp*, 2018.

[35] "David Varodayan." [Online]. Available: http://ivms.stanford.edu/~varodayan/software.html

[36] D. MacKay, "Alist format." [Online]. Available: http://www.inference.org.uk/mackay/codes/alist.html

[37] S. Cook, *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[38] M. Beermann, E. Monzo, L. Schmalen, and P. Vary, "GPU accelerated belief propagation decoding of non-binary LDPC codes with parallel and sequential scheduling," *Journal of Signal Processing Systems*, vol. 78, no. 1, pp. 21–34, 2015.

[39] J. Andrade, N. George, K. Karras, D. Novo, V. Silva, P. Ienne, and G. Falcao, "From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis," in *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE, 2015, pp. 1–8.

[40] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *IEEE Design & Test of Computers*, vol. 26, no. 4, pp. 18–25, 2009.