Parallelized QC-LDPC Encoder on an NVIDIA GPU

An Evaluation of LDPC Codes Compliant with the 5G Standard

Parallell implementering av en QC-LDPC-kodare på en NVIDIA-GPU
En utvärdering av LDPC-koder i enlighet med 5G-standarden

Mustafah Rahimi
Jonos Mirzazada

Faculty of Health, Science and Technology

Computer Science

Bachelor's thesis 15hp

Supervisor: Hans Hedbom <hans.hedbom@kau.se>

Examiner: Tobias Pulls <tobias.pulls@kau.se>

Date: Karlstad, June 2th, 2025

Acknowledgements

We would like to thank our academic supervisor at Karlstad University, Hans Hedbom, for his continued support and guidance throughout the project. His input helped us stay focused and improve the quality of our work.

Our gratitude also goes to our supervisor at Tietoevry, Bengt Hallinger, for his technical support and practical advice, which was valuable during the different stages of the project.

Additionally, we acknowledge the use of AI-assisted tools during the writing and formatting of this report. AI was primarily employed for generating LaTeX tables, reviewing grammar and spelling, and creating a template for code snippet listings.

Abstract

With the exponential growth of data transmission in 5G networks, efficient channel coding has become essential [1]. QC-LDPC codes provide strong error correction and a hardware-friendly structure, but their high computational demands make real-time encoding challenging on standard CPUs.

This thesis investigates the use of Graphics Processing Units (GPUs) and CUDA-based parallel programming to accelerate QC-LDPC encoding. The core contribution lies in demonstrating how GPU parallelism can dramatically reduce encoding latency and scale more efficiently than CPU-based solutions. Experimental results show that GPU acceleration achieved speedups exceeding 80× for large block sizes—reducing encoding times from over 14 seconds on a CPU to less than 200 milliseconds on a GPU. Although performance improvements were limited for smaller messages, the GPU demonstrated a clear and growing advantage as message sizes grew linearly.

The findings highlight the strengths of GPUs in accelerating QC-LDPC encoding and point toward more efficient, scalable, and energy-aware solutions for 5G systems, meeting the growing demands of next-generation wireless networks.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

The rapid evolution of mobile communication technologies has significantly transformed the global landscape, driving unprecedented growth in connectivity and data consumption. As we approach 2030, the adoption and integration of 5G networks are expected to reach new heights, with global 5G subscriptions projected to surpass 6.3 billion—accounting for nearly 67% of all mobile subscriptions [2]. This remarkable growth underscores the critical role of 5G in shaping the future of mobile networks.

By 2030, 5G networks are anticipated to carry 80% of mobile data traffic, highlighting their importance in meeting the increasing demand for high-speed, reliable connectivity. The deployment of 5G standalone (SA) networks is essential for enabling service differentiation and exploring new performance-based business models. These networks will support a wide range of applications, from enhanced mobile broadband to mission-critical communications and IoT advancements.

Channel coding is a fundamental aspect of modern digital communication systems, particularly in the realm of 5G mobile networks. Low-Density Parity-Check (LDPC) codes are widely recognized for their ability to achieve performance close to the Shannon limit, offering efficient error correction capabilities [3]. LDPC encoding

is essential for 5G applications as it not only enhances data reliability but also supports high data rates and low latency communications. The adoption of LDPC codes in 5G represents a significant advancement in channel coding technology, facilitating the seamless delivery of high-quality services in an increasingly connected world.

With the increasing interest in LDPC codes, extensive research has been conducted to enhance the efficiency of encoding and decoding processes. This thesis focuses on optimizing the QC-LDPC encoding process by leveraging parallelized CUDA GPU architecture and comparing its performance with traditional CPU-based implementations. The objective is to improve the efficiency and performance of LDPC code execution, making it more suitable for next-generation communication systems.

1.1 Background

As introduced, Low-Density Parity-Check (LDPC) codes are essential for ensuring reliable and efficient data transmission in 5G communications. To meet the performance and implementation demands of 5G systems, a structured subclass known as Quasi-Cyclic LDPC (QC-LDPC) codes has been adopted [4]. Due to their cyclic structure, QC-LDPC codes support efficient encoding and decoding with lower computational complexity. However, despite these advantages, QC-LDPC encoding remains computationally demanding due to large matrix operations, making parallelization an attractive optimization strategy. Traditional CPU-based implementations often struggle to meet high-throughput requirements, leading to growing interest in parallel computing architectures such as Graphics Processing Units (GPUs).

GPUs offer massive parallelism, making them ideal for large-scale matrix operations.

NVIDIA's Compute Unified Device Architecture (CUDA) provides a framework to efficiently utilize GPU resources for LDPC encoding [5]. Exploiting CUDA's parallel processing capabilities can optimize QC-LDPC encoding, improving execution speed and efficiency compared to CPU-based approaches.

1.2 Problem Description

Efficient channel coding is essential for high-speed data transmission in 5G networks. While LDPC codes provide strong error correction, QC-LDPC encoding is computationally expensive due to large matrix operations like matrix-vector multiplications and cyclic shifts. Unlike decoding, which has been extensively optimized, encoding remains a bottleneck due to high memory overhead and computational complexity. Traditional methods, such as Gaussian elimination and the Richardson-Urbanke (RU) algorithm, either require excessive storage or suffer from long critical paths, making them unsuitable for high-throughput applications. [4].

To address this, GPU-based parallel QC-LDPC encoding offers a promising solution by leveraging parallelism to accelerate matrix operations and reduce computational overhead. Compared to sequential CPU-based encoding, GPU acceleration significantly improves performance, enabling higher throughput and efficiency for 5G communication systems [4].

In this thesis, we aim to answer the question: *How well does parallel QC-LDPC encoding on a GPU perform compared to sequential QC-LDPC encoding on a CPU?* We will investigate this question with a specific focus on QC-LDPC codes used in a 5G channel.

1.3 Thesis Objective

The objective of this thesis is to develop an efficient LDPC encoding method using GPU acceleration with CUDA. The focus is on implementing and optimizing the encoding process on the GPU and comparing its performance to a traditional CPU-based encoder. The implementation will be validated against MATLAB simulations, with a detailed analysis of its compliance with 5G Layer 1 specifications.

1.4 Ethics and Society

The ethical and sustainability considerations for this thesis primarily focus on fairness and environmental impact. Since this project does not involve any private or sensitive data, there are no concerns regarding privacy or security that need to be addressed.

A primary sustainability concern in this project is energy efficiency. While GPU acceleration enhances computational performance, it also leads to increased power consumption, which raises concerns regarding environmental sustainability [6]. However, energy efficiency depends significantly on the workload and implementation. Although GPUs can be more energy-efficient for highly parallel tasks, not all algorithms or implementations achieve this ideal. If the GPU is not fully utilized or the code is not optimized, power consumption may remain high relative to the speed gains. As the deployment of 5G networks expands, it is crucial to balance computational efficiency with responsible energy use to minimize the environmental footprint of high-performance computing.

Another significant societal aspect is accessibility. Advanced computing resources, such as NVIDIA GPUs, are not universally available, particularly in developing regions, which may exacerbate the digital divide [7]. Ensuring that advancements

in LDPC encoding contribute to broader accessibility, rather than reinforcing existing technological disparities, is essential for equitable progress in wireless communication.

1.5 Methodology

To address the research question, this thesis follows a structured methodology [8], consisting of four iterative phases: **Research, Implementation, Experimentation, and Evaluation**.

The study began with a **literature review** to develop an understanding of *QC-LDPC encoding, CUDA programming, and GPU acceleration* techniques. Based on this, a GPU-based QC-LDPC encoder was implemented and compared to a CPU-based implementation.

In the **experimentation** phase, encoding performance was measured using simulation data, focusing on execution time and computational efficiency. The results were analyzed, and adjustments were made to improve performance where necessary. The experiments were repeated, and the resulting data was analyzed to evaluate the performance and practicality of GPU-accelerated QC-LDPC encoding.

1.6 Stakeholders

This thesis project was carried out in collaboration with **Tietoevry** in Karlstad, a leading technology company specializing in software solutions for sectors including education, healthcare, digital commerce, and telecommunications. Tietoevry is actively involved in the development of the 5G network, with a focus on advancing data transmission and processing speeds.

The goal of this thesis is to contribute to Tietoevry's efforts by identifying ways to improve the efficiency of LDPC encoding within the 5G network. By optimizing LDPC encoding, we aim to enhance processing capabilities, ultimately supporting the overall performance and scalability of the 5G infrastructure.

1.7 Distribution of Work

The work was equally distributed across three main areas: implementation, information gathering, and report writing. Implementation was conducted using pair programming to ensure collaboration and efficiency. Information gathering was carried out individually, with both authors reviewing each other's findings for accuracy and completeness. The report was written jointly, with each author reviewing and refining the other's contributions to ensure clarity and consistency.

1.8 Delimitations

This thesis is confined to several key parameters that define the extent and direction of our research. The primary focus of our evaluation is the performance of QC-LDPC encoding, specifically regarding its execution time. While other important factors, such as memory utilization and power efficiency, could provide valuable insights, they are not addressed in this study.

Our research is restricted to QC-LDPC codes conforming to the 5G NR standard. The encoders implemented in this work are specifically designed to support the coding schemes defined by the 5G specifications. This narrow focus ensures that our findings are relevant to current industry practices and can be directly compared to existing 5G encoding solutions.

In addition, this thesis focuses solely on the development and evaluation of a

parallel QC-LDPC encoding algorithm implemented on a GPU. The performance results presented are dependent on the specific hardware and parallelization approach employed. While we concentrate on this particular encoding method, variations in algorithmic approach or hardware configuration could lead to different performance outcomes.

1.9 Outline

The thesis is structured as follows: Chapter 2 provides background information on LDPC codes, GPUs, and CUDA. Chapter 3 outlines the project design, and Chapter 4 details the implementation process. Chapter 5 presents the experimental results, followed by a discussion of potential future work in Chapter 6.

Chapter 2

Background and Related Work

This chapter provides essential background information for this thesis. We begin with an overview of the fundamental structure of communication systems, establishing the context for error correction techniques. Next, we examine channel coding with a particular emphasis on LDPC codes, highlighting their significance in modern communication standards. A key focus of this thesis is Quasi-Cyclic LDPC (QC-LDPC) encoding, which we describe in detail, including its algorithmic implementation. Additionally, we introduce fundamental concepts related to GPUs and the CUDA programming model, as they are central to the acceleration of LDPC encoding. Finally, we review related research and discuss how this thesis contributes to the current state of the field.

2.1 Communication Systems

Communication systems consist of a transmitter that sends information, a receiver that receives the information, and a channel (such as a wireless link, optical fiber, or coaxial cable) that carries the signal [9]. The transmitter contains several key blocks responsible for processing the message before transmission through

the channel. These blocks include the source, source encoder, channel encoder, modulator, and transmission antenna, as shown in Figure 2.1 [10]. Upon receiving the input from the source, the source encoder removes redundant data, making the message more bandwidth-efficient. Next, the channel encoder adds redundancy to the message by introducing error correction codes, enhancing the system's robustness against channel noise. The encoded message is then passed to the modulator, which alters the signal to make it suitable for transmission and distinguishable from other signals in the communication medium. The modulated signal is then transmitted through the channel. The receiver consists of blocks that perform

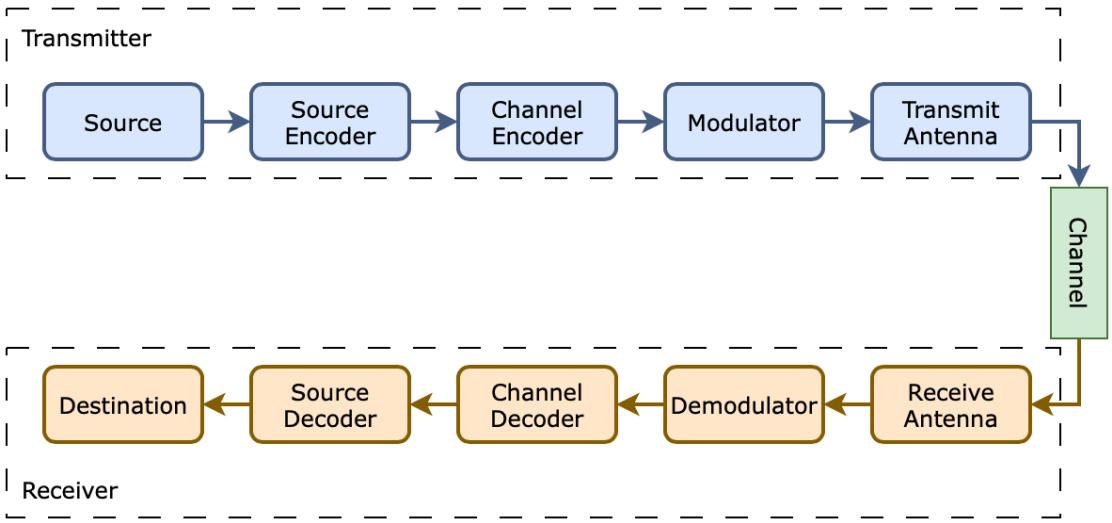

Figure 2.1: A schematic representation of a communication system.

the inverse of the transmitter's operations, as illustrated in Figure 2.1. Upon arrival at the receiver, the signal is first demodulated to extract the transmitted data. The demodulated signal is then passed to the channel decoder, which uses the redundant information to detect and correct any errors introduced during transmission. Finally, the source decoder reconstructs the original message, which is delivered to the destination block of the receiver.

2.2 Channel Coding

In the era of high-speed digital communication, ensuring reliable data transmission over noisy channels is a critical challenge [11]. Errors introduced during transmission, often caused by noise, interference, or fading, can degrade the quality of the received data. Channel coding, also known as error-correcting coding, addresses this issue by adding redundancy to the transmitted data, enabling the receiver to detect and correct errors [12]. Channel coding techniques are broadly categorized into two approaches: *Backward Error Correction* (BEC) and *Forward Error Correction* (FEC) [13].

BEC methods rely on error detection at the receiver, followed by a request for retransmission from the sender. While straightforward, this approach is inefficient for modern communication systems, as it increases latency and reduces throughput. In contrast, *FEC* techniques proactively embed redundancy into the transmitted data, allowing the receiver to correct errors without retransmission. This makes FEC particularly suitable for real-time applications, such as video streaming and wireless communication, where delays are unacceptable. FEC techniques are extensively used in telecommunication systems, making them relevant to this project.

A fundamental Forward Error Correction (FEC) technique is the repetition code, where each bit of the original message is transmitted multiple times to improve reliability [14]. For example, given an input sequence $s = 0\ 1\ 1\ 0$ and using a repetition factor of three, the encoded output becomes $t = 000\ 111\ 111\ 000$. Although this approach is simple to implement, it is highly inefficient in terms of bandwidth, as it significantly increases the volume of transmitted data.

To enhance efficiency, more sophisticated Forward Error Correction (FEC) methods, such as block codes, are utilized. Block codes segment the data into fixed-size

blocks and introduce redundancy to each block. A well-known example is the (7,4) Hamming code, which transforms a four-bit data sequence into a seven-bit encoded sequence by appending three parity-check bits [14]. These parity bits are determined using linear equations, allowing the receiver to detect and correct single-bit errors. For example, given the input $s = 0\ 1\ 0\ 0$, the (7,4) Hamming code produces the encoded output $t = 0100110$.

The encoding process for the (7,4) Hamming code can be represented mathematically using a generator matrix *G*. The transmitted message t is obtained by multiplying the source message vector *s* by G:

$$H = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 \end{bmatrix} \text{ and } t = s \cdot G \tag{2.1}$$

At the receiver, decoding involves calculating the *syndrome*, which indicates whether errors are present and, if so, their locations [14]. The syndrome is computed by comparing the received bits with the expected parity-check bits. If the syndrome is non-zero, it corresponds to a specific error pattern, which can be used to correct the errors. For example, if the syndrome indicates a single-bit error, the erroneous bit can be flipped to restore the original message.

While the (7,4) Hamming code is effective for correcting single-bit errors, it cannot handle multiple-bit errors. The probability of error for this code is proportional to the square of the bit error rate, $O(f^2)$, similar to a repetition code with three repetitions. However, the Hamming code achieves a higher transmission rate of $R = \frac{4}{7}$, making it more efficient in terms of bandwidth usage [14].

In recent years, Low-Density Parity-Check (LDPC) codes have become a leading

solution for near-Shannon-limit performance [15]. With sparse parity-check matrices, LDPC codes enable efficient encoding and decoding with low computational complexity, making them widely used in modern communication standards like 5G and Wi-Fi 6 [16].

2.3 LDPC

Low-Density Parity-Check (LDPC) codes are a class of error-correcting codes that have gained significant attention due to their near-capacity performance and efficient decoding algorithms. First introduced by Robert G. Gallagher in his seminal 1962 doctoral thesis [17]. LDPC codes remained largely unutilized for several decades due to the computational limitations of the time. However, advancements in computing hardware and coding theory have led to their widespread adoption in modern communication systems, including wireless networks, satellite communications, and data storage [18].

LDPC codes are defined by a sparse parity-check matrix H, where the term *low-density* refers to the small number of ones in the matrix relative to its size. This sparsity is a key feature that enables efficient encoding and decoding. The iterative message-passing algorithms used for decoding, such as the *Belief Propagation* (BP) algorithm, exploit the sparse structure of H to achieve near-optimal performance with relatively low computational complexity [19].

The parity-check matrix H can be represented graphically as a bipartite graph known as a Tanner graph, which consists of two types of nodes: *variable nodes* (representing codeword bits) and *check nodes* (representing parity-check equations), as illustrated in *Figure* 2.2 [20]. The structure of the Tanner graph directly reflects the parity-check matrix H, where each column in H corresponds to a variable node, and each row corresponds to a check node.

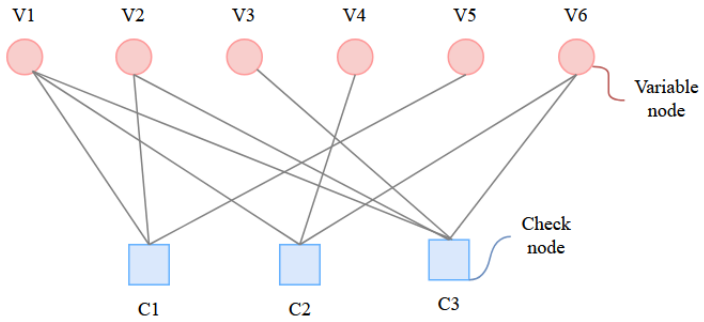

Figure 2.2: A Tanner graph for an LDPC code is a graphical representation where circles represent the variable nodes, and squares represent the check nodes.

An entry of 1 in the matrix H indicates an edge connecting a variable node to a check node in the Tanner graph. This relationship is illustrated in *Figure* 2.2, which shows the Tanner graph corresponding to the parity-check matrix H given in *Equation* 2.2.

$$H = \begin{bmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{2.2}$$

Since H has three rows and six columns, the corresponding Tanner graph contains six variable nodes and three check nodes. An edge exists between a check node c_i and a variable node v_j if the entry $H(i, j) = 1$. Observing the matrix, each check node connects to a subset of the variable nodes, ensuring that each parity-check equation constrains certain codeword bits.

2.4 QC-LDPC

Quasi-Cyclic Low-Density Parity-Check (QC-LDPC) codes are a specialized subclass of LDPC codes that have gained significant attention due to their structural properties

and practical advantages. These codes are prominently used in modern communication systems, including the 5G New Radio (NR) standard [21]. The defining feature of QC-LDPC codes is their parity-check matrix H, which is constructed using an array of submatrices. These submatrices are either zero matrices or circulant permutation matrices, enabling efficient encoding and decoding processes [22].

The parity-check matrix H of a QC-LDPC code is composed of circulant permutation matrices and zero matrices [22]. Let I denote the identity matrix of size $Z \times Z$, where Z is referred to as the *lifting size*. A circulant permutation matrix C_1 is derived by cyclically shifting the identity matrix I to the right by one position. Mathematically, C_1 is represented in Equation 2.3:

$$C_1 = \begin{bmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \end{bmatrix} \tag{2.3}$$

Similarly, C_i represents the identity matrix cyclically shifted to the right by i positions, where i is an integer satisfying $0 \leq i < Z$. If $i = -1$, C_{-1} corresponds to a **zero matrix** (or null matrix) of size $Z \times Z$.

The parity-check matrix H of a QC-LDPC code is then defined as an $mZ \times nZ$ block matrix:

$$H = \begin{bmatrix} C^{\alpha_{11}} & C^{\alpha_{12}} & \cdots & C^{\alpha_{1n}} \\ C^{\alpha_{21}} & C^{\alpha_{22}} & \cdots & C^{\alpha_{2n}} \\ \vdots & \vdots & \ddots & \vdots \\ C^{\alpha_{m1}} & C^{\alpha_{m2}} & \cdots & C^{\alpha_{mn}} \end{bmatrix} \tag{2.4}$$

where $\alpha_{ij} \in \{-1, 0, 1, \ldots, Z-1\}$ determines the cyclic shift for each submatrix.

The structure of H is shown in *Equation* 2.4, which depicts the block matrix composed of circulant permutation matrices and zero matrices.

The parameter Z (*lifting size*) plays a critical role in defining the size and performance of the code. Larger values of Z result in longer block lengths, which generally improve error-correcting performance but also increase computational complexity [23].

In the 5G NR standard, QC-LDPC codes are constructed using one of two *base graphs* [21]. These base graphs serve as prototypes that can be expanded by replacing their entries with circulant permutation matrices or zero matrices of size $Z \times Z$. The two base graphs are defined as follows:

- **Base Graph 1** (BG1): Consists of 46 rows and 68 columns. It is designed for higher code rates and longer block lengths.
- **Base Graph 2** (BG2): Consists of 42 rows and 52 columns. It is optimized for lower code rates and shorter block lengths.

The lifting size Z in the 5G standard ranges from *2 to 384*, allowing for significant flexibility in block length [21]. Consequently, the size of the parity-check matrix H can vary from 84×104 (for $Z = 2$) to $17,664 \times 26,112$ (for $Z = 384$). The choice of lifting size depends on the specific application and performance requirements.

The structure of the base graph directly influences the sparsity and connectivity of the parity-check matrix. For example:

- In *BG1*, the maximum number of non-zero entries (ones) in any row is 19.
- In *BG2*, the maximum number of non-zero entries in any row is 10.

A key advantage of using a circularly-shifted identity matrix as a permutation matrix is that each permutation can be represented by a single integer. This

significantly reduces memory usage during implementation and enables the use of a straightforward switch network for both encoding and decoding [24].

2.4.1 LDPC Encoding Algorithms

Low-Density Parity-Check (LDPC) encoding ensures that the codeword $\mathbf{C}$ satisfies the parity-check condition [4]:

$$H\mathbf{C}^T = 0 \tag{2.5}$$

This section presents two widely used LDPC encoding methods: *Gaussian Elimination (GE)* and the *Richardson-Urbanke (RU)* method, highlighting their computational complexity, storage requirements, and feasibility for 5G applications.

2.4.1.1 LDPC Encoding with Gaussian Elimination

The *Gaussian Elimination (GE)* method is a conventional approach that encodes LDPC codes by multiplying the information bits with the generator matrix G [4]. The generator matrix is derived from the parity-check matrix H by performing Gauss-Jordan elimination, transforming H into the form:

$$H = \begin{bmatrix} A & I_{N-K} \end{bmatrix} \tag{2.6}$$

where A is an $(N-K) \times K$ binary matrix and I_{N-K} is the identity matrix of order $(N-K)$ [4]. The generator matrix G is then obtained as:

$$G = \begin{bmatrix} I_K & A^T \end{bmatrix} \tag{2.7}$$

The codeword C is computed as:

$$C = SG \tag{2.8}$$

where S represents the systematic information bits.

While the GE method ensures correct encoding, it has a major drawback: the generator matrix G is typically dense, requiring significant memory storage and leading to high computational complexity $O(N^2)$. This makes it impractical for large block sizes, such as those used in 5G New Radio (NR) LDPC codes.

2.4.1.2 LDPC Encoding with the Richardson-Urbanke (RU) Method

The *Richardson-Urbanke (RU)* algorithm is a linear-time encoding approach that directly encodes an LDPC code using the parity-check matrix H [4]. Instead of computing G, the method transforms H into an Approximate Lower Triangular (ALT) form using only row and column permutations. The transformed matrix has the form:

$$H^T = \begin{bmatrix} A & B & T \\ C & D & E \end{bmatrix} \tag{2.9}$$

where T is a lower triangular matrix with 1s along the diagonal [4]. Through additional matrix operations, the encoding is performed efficiently without needing the explicit generator matrix. The parity-check equations are solved in two steps [4]:

1. Compute the first set of parity bits p_1 using:

$$p_1^T = D^{-1}(CS^T) \tag{2.10}$$

2. Compute the remaining parity bits p_2 using:

$$p_2^T = -T^{-1}(AS^T + Bp_1^T) \tag{2.11}$$

Since T is lower triangular, back substitution is used to determine p_2, reducing

encoding complexity to $O(N+G^2)$, where G is the gap between the ALT form and an ideal triangular form [4].
Although the RU method significantly reduces storage requirements and computational complexity, it introduces a long critical path due to multiple matrix operations. Additionally, it lacks a clear, programmable step-by-step algorithm, which complicates hardware implementation [4].

In this project, we employ an alternative encoding method inspired by the approach described in [25], which computes parity bits by solving a system of linear equations derived from the sparse structure of the parity-check matrix H. This approach constructs the encoded codeword by first partitioning H into systematic and parity components and then solving for the parity bits. The following section provides the algorithmic formulation of this encoding process:

Step 1: Define the Parity-Check Matrix
Before proceeding, let us define the parameters used in this formulation:

- m is the number of rows in the base matrix B.
- n is the number of columns in the base matrix B, corresponding to the total number of bits in the codeword.
- k is the number of message bits in the codeword.
- z is the lifting factor, which is used to expand the base matrix B.

The parity-check matrix H for a QC-LDPC code is constructed by expanding the base matrix B using the lifting factor z. The resulting matrix H has dimensions:

$$H = \begin{bmatrix} H_1 & H_2 \end{bmatrix}, \quad H \in \mathbb{F}_2^{mZ \times nZ} \tag{2.12}$$

where:

- H_1 corresponds to the first kZ columns, related to the systematic message bits.
- H_2 corresponds to the last $(mZ - kZ)$ columns, used to compute parity bits.

The LDPC codeword $\mathbf{c}$ consists of systematic bits $\mathbf{s}$ and parity bits $\mathbf{p}$:

$$\mathbf{c} = \begin{bmatrix} \mathbf{s} \\ \mathbf{p} \end{bmatrix} \tag{2.13}$$

and must satisfy the parity-check equation:

$$H \cdot \mathbf{c}^T = 0. \tag{2.14}$$

Expanding this equation:

$$H_1 \cdot \mathbf{s}^T + H_2 \cdot \mathbf{p}^T = 0. \tag{2.15}$$

Step 2: Compute Parity Bits

The first step in computing the parity bits is to calculate the parity sum, which is the result of multiplying the matrix H_1 with the systematic message vector $\mathbf{s}$. This operation computes the contribution of the systematic message bits to the parity-check equation:

$$\mathbf{r} = H_1 \cdot \mathbf{s}$$

Here, H_1 is the submatrix of the parity-check matrix corresponding to the systematic bits, and $\mathbf{r}$ represents the intermediate vector that will be used to determine the parity bits.

The next step is to solve the system of linear equations that is formed by the parity-check matrix H_2 and the vector $\mathbf{r}$. This system is expressed as:

$$H_2 \cdot \mathbf{p} = \mathbf{r}$$

where H_2 is the submatrix of the parity-check matrix corresponding to the parity bits, and $\mathbf{p}$ is the vector of parity bits that we wish to solve for. To find $\mathbf{p}$, we apply the Sparse LU decomposition to H_2. This decomposition allows for an efficient solution of the system by computing the inverse of H_2, provided that H_2 is invertible:

$$\mathbf{p} = H_2^{-1} \cdot \mathbf{r}$$

This step results in the computed values for the parity bits, ensuring that the generated codeword satisfies the parity-check matrix constraints.

Step 3: Construct the Codeword

Once the parity bits are computed, the final LDPC codeword is constructed as:

$$\mathbf{c} = \begin{bmatrix} \mathbf{s} \\ \mathbf{p} \end{bmatrix} \tag{2.16}$$

2.4.2 GPU

The evolution of GPUs (Graphics Processing Units) was primarily driven by the increasing demand for high-performance graphics in PC and video games [26]. NVIDIA pioneered the first GPU in 1999, marking a significant milestone in computational hardware. Over time, GPUs gained recognition not only for graphical rendering but also for their exceptional floating-point performance and programmability, which opened doors to applications beyond graphics, such as scientific computing and machine learning. This shift led to the emergence of general-purpose GPU (GPGPU) programming, enabling developers to leverage GPUs for a wide range of computational tasks.

In contrast to CPUs (Central Processing Units), which are specifically designed for sequential processing and complex control tasks [27], GPUs are engineered

with a highly parallel architecture that enables the concurrent execution of thousands of operations. This architectural difference allows GPUs to efficiently handle computations that require massive parallelism. As shown in *Figure* 2.3[10], GPUs allocate a significantly larger proportion of their transistors to data processing, as opposed to caching or flow control mechanisms. This design choice is crucial for optimizing performance in tasks such as graphics rendering, scientific simulations, and other applications that demand high levels of parallel computation.

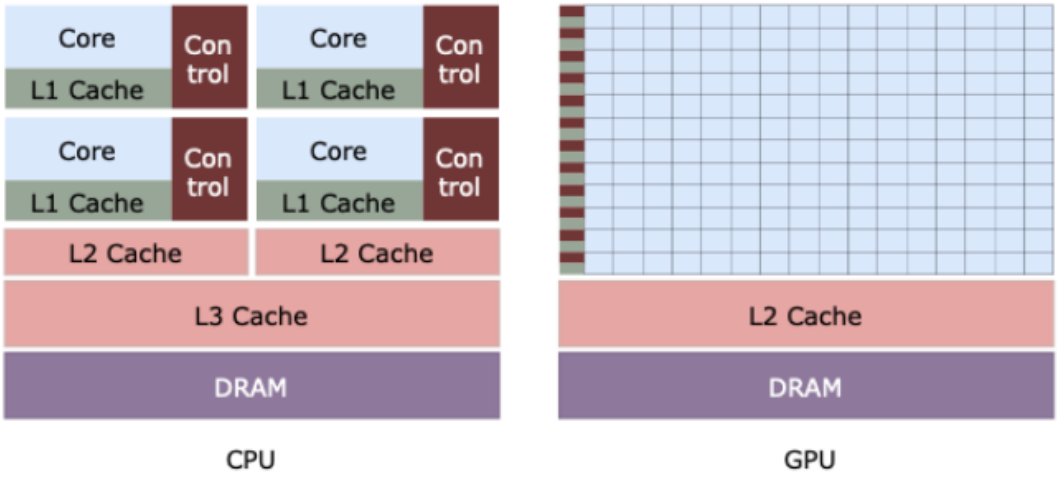

Figure 2.3: A structural representation of a CPU and a GPU.

2.4.3 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing platform and programming model developed by NVIDIA for general-purpose computing on GPUs [5]. By leveraging the parallel processing capabilities of GPUs, CUDA enables the efficient execution of computationally intensive tasks that would otherwise be slower on traditional CPUs. This makes CUDA particularly effective for accelerating algorithms that involve large-scale matrix operations and parallel computations, offering substantial performance gains in a variety of complex

computational tasks. To make GPU programming accessible, CUDA supports several widely used programming languages, including *C, C++, Java,* and *Python*. This is achieved through a set of minimal language extensions that allow programmers to manage the core functionalities required for GPU programming.

2.4.3.1 Kernels

A fundamental feature of CUDA is its capability to define `kernel-functions` executed concurrently across multiple threads on one or more GPUs, facilitating efficient parallel processing [5]. In CUDA *C++*, a kernel is declared using the `__global__` keyword before the function definition. The number of threads assigned to the kernel is specified during its invocation using `<<<...>>>` syntax. Each thread is assigned a unique thread ID, which can be accessed using built-in variables such as `threadIdx.x` and `threadIdx.y`, allowing fine-grained control over parallel execution and data processing.

For QC-LDPC encoding, kernels can be designed to parallelize matrix operations, such as matrix addition or matrix-vector multiplication, which are essential for generating codewords. Below is an example of a CUDA kernel that performs matrix addition, which can be adapted for QC-LDPC encoding tasks:

```
// Kernel definition for matrix addition
__global__ void MatAdd (float A[N][N], float B[N][N], float C[N][N]) {
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main() {
    ...
    // Kernel invocation with one block of N * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
```

```
    ...
}
```

Listing 2.1: CUDA example code of matrix addition

In the above example, each thread computes the sum of corresponding elements from matrices A and B, storing the result in matrix C. This approach allows for efficient parallelization of matrix addition, which can be adapted for QC-LDPC encoding tasks such as combining intermediate results or updating parity-check matrices.

2.4.3.2 Shared Memory and Synchronization

Threads within a block can collaborate by utilizing **shared memory**, a high-speed memory space that allows data exchange between threads [5]. Shared memory is allocated using the `__shared__` keyword and provides faster access compared to global memory. For dynamic allocation of shared memory, an additional parameter is passed during kernel invocation to specify the memory size. The dynamically allocated memory is accessed through an externally declared array. To ensure proper synchronization between threads, the `__syncthreads()` function is used, which forces all threads in a block to wait until every thread has reached the synchronization point.

In QC-LDPC encoding, shared memory can be used to store sub-matrices of the generator matrix or intermediate results during matrix operations. For example, when performing matrix addition or matrix-vector multiplication, shared memory can be used to cache portions of the input matrices or vectors to reduce global memory access overhead. Below is an example demonstrating the use of shared memory and synchronization:

```
__global__ void MatAddShared(float A[N][N], float B[N][N], float C[N][N]) {
    __shared__ float sharedA[16][16];
```

```
    __shared__ float sharedB[16][16];

    int i = threadIdx.x;
    int j = threadIdx.y;

    // Load data into shared memory
    sharedA[i][j] = A[i][j];
    sharedB[i][j] = B[i][j];
    __syncthreads();

    // Perform addition using shared memory
    C[i][j] = sharedA[i][j] + sharedB[i][j];
}

...
MatAddShared<<<numBlocks, threadsPerBlock>>>(A, B, C);
...
```

Listing 2.2: CUDA kernel for matrix addition with shared memory

The example above demonstrates how shared memory and synchronization can be utilized to optimize matrix addition, a fundamental operation in QC-LDPC encoding.

2.4.3.3 Memory Management in CUDA

The CUDA programming model assumes that the host (CPU) and the device (GPU) operate independently, each with its own memory space [5]. As a result, data must be explicitly transferred between the host and device to facilitate input and output operations for kernels.
This is particularly important for QC-LDPC encoding, where large matrices (e.g., the generator matrix) and input vectors need to be transferred to the GPU. The following example extends the matrix addition kernel to include the necessary memory management operations:

```
int main() {
    size_t size = N * N * sizeof(float);
    // Allocate matrices in host memory
    float h_A[N][N], h_B[N][N], h_C[N][N];

    // Initialize input matrices with values
    ...
    // Allocate matrices in device memory
    float *d_A, *d_B, *d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    // Copy input matrices from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Kernel invocation
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // Free host memory
    ...
}
```

Listing 2.3: CUDA host code for matrix addition kernel invocation

The code in Listing 2.3 demonstrates the typical workflow of a CUDA program, which involves memory allocation, data transfer between host and device, kernel execution, and memory de-allocation. For QC-LDPC encoding, similar memory management techniques are required to handle the large datasets involved in

the encoding process.

Although GPUs can significantly accelerate computational tasks, memory transfers between the host and device can become a performance bottleneck. The PCIe bus connecting CPU and GPU has considerably lower bandwidth than the GPU's internal memory system — for example, 8 GB/s between host and device versus 144 GB/s within device memory in older Tesla C2050 GPUs [28]. This disparity can make memory transfer the dominant factor in runtime if not carefully managed. Strategies such as minimizing data movement, using pinned memory, batching transfers, or overlapping transfers with kernel execution are essential to fully realize GPU acceleration benefits [28].

2.4.4 Related Work

Numerous research studies have explored parallelized LDPC encoding, which is essential for modern communication standards like 5G NR due to its structure that supports efficient parallel computation. This section reviews prior work on QC-LDPC encoding, particularly focusing on implementations on specialized hardware platforms and GPU-based solutions using CUDA. Several approaches have been suggested to enhance throughput and reduce memory usage.

2.4.4.1 LDPC Encoding Using Specialized Hardware

Several studies have explored the implementation of QC-LDPC encoders on programmable hardware platforms such as FPGAs and ASICs. In [4], a high-throughput, low-complexity encoder is introduced for QC-LDPC codes in the 5G NR standard. The study optimizes memory usage by storing only quantized permutation data rather than the entire parity-check matrix. The proposed ASIC design, implemented in 65-nm CMOS technology, achieves a throughput between 22.1 and 202.4 Gbps,

showcasing an effective balance between computational efficiency and resource constraints.

2.4.4.2 GPU-Based LDPC Encoding Solutions

The potential of GPUs for implementing QC-LDPC encoding using CUDA has been explored by numerous researchers. In [29], a high-throughput and flexible-rate LDPC encoder is developed on an NVIDIA GPU. The implementation demonstrates a throughput range of 38 - 62 Gbps across various code rates, surpassing FPGA-based encoders in terms of flexibility. The study attributes this performance to efficient GPU parallelization, where multiple parity calculations are executed simultaneously. Additionally, the study highlights memory access patterns as a critical factor affecting execution speed.

In [30], a low-latency QC-LDPC encoder optimized for 5G NR achieves a peak throughput of 257.9 Gbps at an 800 MHz clock rate. This study introduces a highly parallelized encoding structure, using configurable circuit designs to support all 5G NR code lengths and rates. Unlike traditional serial-based encoding methods, this approach focuses on optimizing execution pipelines and leveraging multi-channel processing to minimize latency.

Our work contributes to this field by optimizing the encoding process through the use of CUDA, providing an efficient alternative to CPU-based solutions. We focus on improving performance by leveraging CUDA's parallel computing architecture to accelerate operations, which can handle a large number of simultaneous computations. Additionally, CUDA's memory management capabilities allow for more efficient data transfer between the CPU and GPU, reducing bottlenecks and improving throughput. By parallelizing matrix calculations, we aim to improve encoding speed and reduce latency, inspired by related works aligned with our setup.

Chapter 3

Design

This chapter provides a comprehensive overview of the encoder's project design, detailing methods to obtain input data, validate output, and the encoder's structure and function. The encoder is organized into four primary components: initializing variables, loading the base matrix and message vector, performing the encoding process, and verifying the output against the MATLAB implementation. Each component is discussed in detail in the following sections.

The 5G New Radio (5G NR) standard is responsible for managing radio communication between base stations and user equipment (UE)[31]. As illustrated in *Figure* 3.1, this communication is bidirectional, consisting of an uplink, where data is sent from the UE to the base station, and a downlink, where data is sent from the base station to the UE.

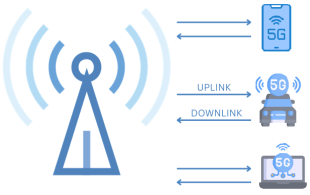

Figure 3.1: Communication between the base station and user equipment in 5G NR.

LDPC encoding is used at the transmitter side in both the uplink and downlink chains. Specifically, it is applied at the UE before transmitting uplink data, and at the base station before transmitting downlink data.
To ensure a consistent and structured development process, the encoder system was organized around a common input and output interface, as illustrated in *Figure* 3.2. This architecture allows for parallel development of three encoder implementations:

- A **MATLAB-based LDPC encoder**, used as a reference for correctness and functional validation.
- A **C++ encoder running on the CPU**, implemented primarily for comparison purposes with the CUDA-based implementation. This version serves as a baseline to evaluate the performance gains and efficiency improvements introduced by GPU acceleration.
- A **CUDA encoder running on the GPU**, optimized for performance by exploiting parallel processing capabilities.

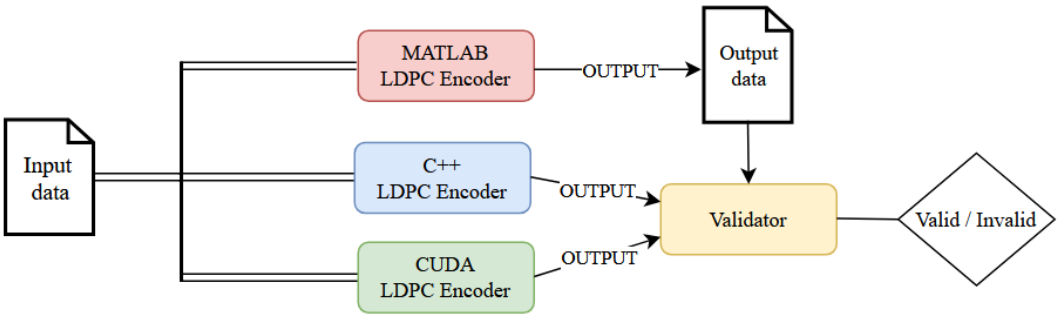

Figure 3.2: Implementation setup of the Encoders.

All implementations process identical input data to produce a codeword, which is then validated against a MATLAB-generated reference. The validator ensures

each codeword satisfies LDPC parity-check constraints and matches the reference, confirming correctness across CPU and GPU implementations.

3.1 Input Data

Before delving into the CPU and GPU-specific design implementations, it is essential to outline the structure and preparation of the input data used throughout the project. The LDPC encoder relies on two key components as input: the base graph matrices and the message vector.

The LDPC base graphs defined by the 5G NR standard—Base Graph 1 (BG1) and Base Graph 2 (BG2)—were used as the foundation for generating the expanded parity-check matrices. These base matrices were obtained from a public *GitHub* repository [32], which provides the standardized graph definitions used in 5G NR. For compatibility across implementations, the base matrices were included in the project directory as text files.

In the *C++* implementation, the matrices were loaded and stored using *Eigen's* matrix format, allowing for direct manipulation and expansion. For the *CUDA* implementation, the base graphs were preprocessed into flattened 1D arrays suitable for GPU memory and parallel processing.

In addition to the base graphs, message vectors were required to simulate actual data to be encoded. These vectors were generated randomly in *MATLAB*, ensuring that each vector matched the expected input length for the chosen base graph and lifting factor. Specifically, a range of lifting sizes $\{2, 8, 32, 128, 256\}$ was selected to create a series of message sizes reflecting the scaling behavior of the encoder. The message length is computed as $(n-k) \times Z$, where n and k are the number of columns and rows of the base matrix, respectively, and Z is the lifting size.

For BG1 (46×68), this resulted in message sizes of 44, 176, 704, 2816, and 5632

bits, while BG2 (42×52) produced message sizes of 20, 80, 320, 1280, and 2560 bits. This set of sizes was chosen based on standard-compliant lifting factors defined by 3GPP TS 38.212 [21], covering both small and large block lengths to enable meaningful performance comparisons across platforms.

This unified input structure ensured a fair and accurate comparison between implementations and allowed reliable validation of correctness across platforms.

3.2 C++ Design

The CPU-based LDPC encoder was implemented in *C++* using the Eigen library, which offers efficient and well-optimized tools for linear algebra operations [33]. This implementation was developed with two main objectives. First, it served as a modular and easily debuggable reference during the initial stages of development. Second, and more importantly, it was designed to provide a meaningful and fair baseline for performance comparison with the GPU-based implementation. To avoid introducing any bias in the evaluation, the CPU version utilizes Eigen's optimized data structures and sparse solver routines, ensuring that matrix operations and linear system solutions are performed efficiently. Additionally, the same core algorithm was used across both CPU and GPU versions to maintain consistency in functionality and output. The overall structure of the CPU encoder is illustrated in *Figure* 3.3.

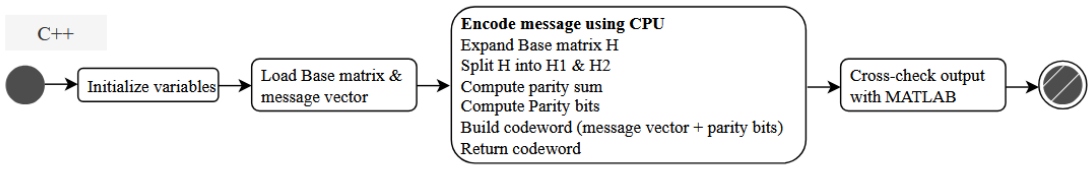

Figure 3.3: Activity diagram for the C++ implementation of LDPC Encoder.

The encoder workflow begins with the initialization of variables and the loading of the base matrix and message vector from external files. The base matrix, one of the 5G NR base graphs (BG1 or BG2), is expanded using a lifting factor Z, producing a large sparse parity-check matrix H. The message vector is a binary vector, generated in MATLAB and shared across all implementations for consistency.

Following expansion, the matrix H is divided into two submatrices: H_1, associated with the systematic part of the code, and H_2, associated with the parity bit generation. These matrices form the foundation for solving the LDPC parity-check equations. To compute the parity bits, a linear system defined by $H_1 \cdot x + H_2 \cdot p = 0$ is solved, where x is the known message vector and p is the unknown parity vector. Matrix-vector multiplication is performed using Eigen's sparse matrix routines, and the resulting equation is solved using Eigen's built-in sparse solvers. The final codeword is assembled by concatenating the original message vector x with the computed parity vector p, resulting in a complete encoded block.

To evaluate its performance, the encoder routine (illustrated in the third block of *Figure* 3.3) is executed ten times before the resulting codeword is passed to the validation module. Only the time spent within this block is measured; the first, second, and fourth blocks are excluded from the timing process to ensure that the results reflect the encoding routine alone. The third block contains the main LDPC encoder function, which internally calls several subroutines responsible for tasks such as base matrix expansion, partitioning, and parity generation. In practice, a function pointer to the main encoder routine is passed to a timing module, which repeatedly invokes the routine and returns a structured data object containing the minimum, maximum, and average execution times for later analysis.

The codeword is then passed to a validation module that verifies its structural correctness against the reference *MATLAB* implementation. The validator checks whether the codeword satisfies the parity-check condition and confirms functional equivalence between *CPU* and *GPU* implementations.

3.3 CUDA Design

The GPU-based LDPC encoder is implemented using *NVIDIA's CUDA* platform to take advantage of parallel processing capabilities. While the CPU implementation focuses on simplicity and serves as a baseline for comparison, the GPU implementation is designed to improve performance by utilizing the computational power of modern graphics hardware. An overview of the GPU encoder architecture is shown in *Figure* 3.4.

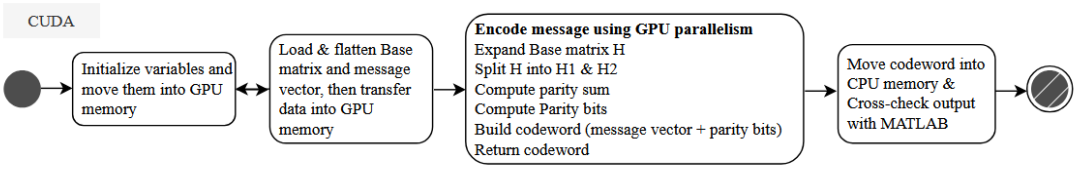

Figure 3.4: Activity diagram for the CUDA implementation of LDPC Encoder.

The GPU encoder begins by transferring the input data—comprising the base matrix and message vector—into GPU global memory. The base matrices (BG1 or BG2) are pre-loaded into flattened vectors for memory efficiency and transferred to the GPU via host-to-device memory copy operations. The message vector is read from a shared input file to ensure consistency across platforms.

Once on the GPU, the base matrix undergoes expansion based on the lifting factor Z, where each non-negative entry in the base graph indicates a circulant right-shifted identity submatrix. This expansion is performed in parallel using

custom CUDA kernels. These kernels are designed to distribute computation across multiple threads and blocks, enabling concurrent processing of matrix rows and submatrices.

Following the expansion, the matrix is split into H_1 and H_2, with H_2 stored in the Compressed Sparse Row (CSR) format to support efficient sparse operations. This format minimizes memory usage and allows cuSPARSE and cuSOLVER routines to access and process non-zero elements directly.

The multiplication of the H_1 matrix with the message vector is handled using `cuBLAS`, NVIDIA's high-performance linear algebra library. The result is used in conjunction with `cuSPARSE` and `cuSOLVER` to solve the sparse linear system and obtain the parity vector. The use of these libraries offloads complex numerical computations to highly optimized GPU routines.

To further enhance performance, the implementation makes use of *CUDA streams* and *shared memory*. *CUDA* streams allow for overlapping data transfers and computation, reducing idle time and increasing throughput. Shared memory provides fast, user-managed cache within a thread block, which is particularly useful for reusing frequently accessed data during matrix operations.

Finally, the message vector and computed parity vector are concatenated to form the final codeword. This codeword is then transferred back to the host (CPU) memory and passed to the same validation module used in the CPU workflow.

As in the CPU design, the GPU encoder routine (illustrated in the third block of *Figure* 3.4) is executed ten times prior to validation to evaluate performance. Only this block is timed, and the minimum, maximum, and average execution times are recorded using a shared timing data structure defined in the utilities module.

In the GPU version, the first and second blocks include additional steps to transfer

data between host and device memory, such as variable initialization and base matrix preparation. The core encoder logic in the third block, along with the validation steps in the fourth, remains consistent with the CPU implementation. As before, timing is strictly limited to the third block, ensuring that setup and memory transfer operations do not affect the results and enabling a fair comparison of encoding performance between the two implementations.

Chapter 4

Implementation

In this chapter, we describe the implementation details of the LDPC encoder introduced in the previous chapter. Both the *C++* and *CUDA* implementations follow a similar high-level structure, which is illustrated in the activity diagrams in *Figure* 3.3 and *Figure* 3.4 in Chapter 3. As in the design chapter, the main blocks of the activity diagram form the basis for the structure of this chapter, guiding the explanation of each step in the encoding process.

4.1 Variables

In the implementation of the LDPC encoder across both CPU and GPU components, several core variables are consistently utilized to manage input/output operations, matrix transformations, and encoding computations. These variables handle essential tasks such as file loading, matrix expansion, message encoding, and performance measurement. They provide a consistent interface for managing input data, encoding operations, and output results.

Table 4.1 provides an overview of the key variables, their types, and descriptions, highlighting their roles in the overall encoding workflow.

Variable	Type	Description
filePath	std::string	Path to the input base matrix file.
files	std::vector<std::string>	List of file paths for base matrices.
B	Eigen::MatrixXi or int*	Base matrix loaded from the file.
m, n	int	Number of rows (m) and columns (n) in the base matrix.
z	int	Lifting factor used to expand the base matrix.
H	Eigen::MatrixXf or double*	Expanded parity-check matrix.
msg	Eigen::VectorXf or double*	Input message vector loaded from a file or generated randomly.
msg_size	int	Size of the message vector, calculated as $(n-m)\cdot z$.
c	Eigen::VectorXf or double*	Encoded codeword (message bits + parity bits).
H1, H2	Eigen::MatrixXf or double*	Split parts of the expanded parity-check matrix.
p	Eigen::VectorXf or double*	Parity bits computed from the parity-check matrix and the message vector.
execTime	float or double	Time taken to encode the message.
isValid	bool	Indicates whether the generated codeword is valid.

Table 4.1: Common variables used in LDPC encoder implementation

In CUDA, flat arrays are used for efficient memory management and compatibility with GPU operations. They are simple, contiguous blocks of memory that allow for grouped memory access, which is critical for parallel computation. Flat arrays are allocated on the GPU using `cudaMalloc` and transferred between the host and device using `cudaMemcpy`.

Variables that need to be transferred between *CPU* and *GPU* memory are commonly named using the `h_` and `d_` prefixes, where `h_` indicates a host (CPU) variable and `d_` indicates a device (GPU) variable. For example, `h_msg` refers to the message vector on the host, while `d_msg` refers to its corresponding copy on the device. Elements in these arrays are accessed using index-based arithmetic, such as `array[i * cols + j]` for representing 2D data in a 1D array. Flat arrays are commonly used to store matrices and vectors, which CUDA kernels then process for tasks like matrix expansion, sparse matrix operations, and concatenation.

In C++, `Eigen` matrices and vectors are utilized for their high-level API and the ease they offer in linear algebra operations. `Eigen` provides intuitive syntax for matrix and vector manipulation, for example, accessing an element of a matrix

with `matrix`(i, j) and an element of a vector with `vector`(i). These structures are particularly useful for CPU-based operations such as loading data and performing matrix operations.

4.2 Loading and Preparing Input Data

In both the *C++* and *CUDA* implementations, the encoding process begins with preparing and loading the base matrix and the input message vector. These two components are critical for expanding the LDPC matrix and generating the encoded codeword.

4.2.1 Base Matrix Preparation

The base matrix is stored as a text file containing integer entries. To load the base matrix:

- In the *CPU* implementation, the matrix dimensions, denoted by `rows` (m) and `cols` (n), are first determined using the `getMatrixDimensions` function.
- The matrix is then loaded into an Eigen matrix (`Eigen::MatrixXi`) using the `loadMatrixFromFile` function.

An example from the CPU code is shown in *Listing* 4.1.

```
1 std::string filePath = "input_files/base_matrices/NR_1_2_20.txt";
2 int rows, cols;
3 getMatrixDimensions(filePath, rows, cols);
4 Eigen::MatrixXi B = loadMatrixFromFile(filePath, rows, cols);
```

Listing 4.1: Loading the base matrix in CPU implementation

In the GPU implementation, the base matrix is initially loaded in the same way into an Eigen matrix. However, for CUDA processing, it is subsequently flattened

into a one-dimensional array using the `eigenMatrixToFlatArray` function, as shown in *Listing* 4.2.

```
Eigen::MatrixXi B = loadMatrixFromFile(filePath, rows, cols);
int* h_baseMatrix = eigenMatrixToFlatArray(B);
```

Listing 4.2: Flattening the base matrix for CUDA operations

This two-step approach was chosen primarily for convenience, as the necessary functionality for parsing text files into Eigen matrices was already in place. While it might seem more efficient to load the data directly into a flat array for GPU processing, doing so would require additional parsing logic and is not guaranteed to result in a meaningful performance gain. The main overhead lies in reading and interpreting the text-based matrix data, rather than in converting the in-memory structure. Furthermore, since the base matrix preparation occurs outside the actual LDPC encoding routine, it does not impact the measured encoding times, and optimization efforts were therefore focused on the encoding process itself.

4.2.2 Message Vector Preparation

The message vector contains random or predefined binary values that are to be encoded.

In the *CPU* version, the message vector is directly loaded as an `Eigen::VectorXf` using the `loadVectorFromFile` function. This procedure is demonstrated in *Listing* 4.3.

```
Eigen::VectorXf msg = loadVectorFromFile("input_files/msgt.txt", (n - m) * z);
```

Listing 4.3: Loading the message vector in CPU implementation

In the *GPU* version, after loading the message vector, it is flattened into a `double*` array using the `eigenVectorToFlatArray` function to enable efficient device memory operations.

This is illustrated in *Listing* 4.4.

```
Eigen::VectorXf msg = loadVectorFromFile("input_files/msgt.txt", msg_bit_size);
double* h_msg = eigenVectorToFlatArray(msg);
```

Listing 4.4: Flattening the message vector for CUDA operations

4.3 Message Encoding Workflow

The message encoding process involves generating a complete codeword by combining the original message bits with parity bits derived from the base matrix structure. Both *CPU* and *GPU* implementations follow the same general sequence of steps for encoding; however, the methods, libraries, and tools used differ between the two to optimize performance for each setup.

In this section, we will first provide an overview of the common encoding steps and then describe the specific implementations for the *CPU* and *GPU* separately. The encoding process includes expanding the base matrix H, splitting H into submatrices H_1 and H_2, computing intermediate parity sums, calculating the final parity bits, and constructing the complete codeword by appending the parity bits to the original message vector.

4.3.1 Expand Base Matrix H

The base matrix B is a compact representation used to define the structure of the full LDPC parity-check matrix H. As discussed in Chapter 2, Base matrix expansion refers to the process of converting B into a larger matrix H by replacing each entry with a corresponding $z \times z$ block.

Entries with a value of -1 are expanded into zero matrices, while non-negative entries are expanded into circulantly shifted identity matrices based on the shift

value.
The following subsections describe how this expansion is implemented on the *CPU* and *GPU*.

4.3.1.1 CPU Implementation

On the *CPU*, the base matrix expansion is handled by several helper functions together with the main function `expandBaseMatrix` shown in *Listing* 4.5. The process uses the Eigen library for matrix operations.
The following helper functions are used:

- `fillBlockWithZeros`: Fills a $z \times z$ block in matrix H with zeros starting from a specified row and column.
- `generateIdentityMatrix`: Creates a $z \times z$ identity matrix.
- `circShift`: Applies a circular shift to a vector by a specified number of positions.
- `applyShiftedIdentityToH`: Applies a circulantly shifted identity matrix into a block of H based on the shift value from the base matrix B.

The function `expandBaseMatrix` iterates through each element of the base matrix B. If an element is -1, a block of zeros is inserted into H. Otherwise, a shifted identity matrix is generated and inserted at the appropriate location.

The code for the CPU implementation is shown in *Listing* 4.5.

```
void fillBlockWithZeros(Eigen::MatrixXf& H, int startRow, int startCol, int z) {
    H.block(startRow, startCol, z, z).setZero();
}

Eigen::MatrixXf generateIdentityMatrix(int z) {
    Eigen::MatrixXf idmatrix = Eigen::MatrixXf::Zero(z, z);
    idmatrix.diagonal().setOnes();
```

```
8     return idmatrix;
9 }
10
11 void circShift(Eigen::VectorXf& vec, int shiftSize) {
12     shiftSize = shiftSize % vec.size();
13     Eigen::VectorXf temp = vec;
14     vec.head(vec.size() - shiftSize) = temp.tail(vec.size() - shiftSize);
15     vec.tail(shiftSize) = temp.head(shiftSize);
16 }
17
18 void applyShiftedIdentityToH(Eigen::MatrixXf& H, int i, int j, const Eigen::MatrixXi& B
       , int z) {
19     Eigen::MatrixXf idmatrix = generateIdentityMatrix(z);
20     for (int row = 0; row < z; ++row) {
21         int shiftSize = B(i, j);
22         Eigen::VectorXf rowCopy = idmatrix.row(row);
23         circShift(rowCopy, shiftSize);
24         idmatrix.row(row) = rowCopy;
25     }
26
27     idmatrix.transposeInPlace();
28     H.block(i * z, j * z, z, z) = idmatrix;
29 }
30
31 Eigen::MatrixXf expandBaseMatrix(const Eigen::MatrixXi& B, int z) {
32     int m = B.rows();
33     int n = B.cols();
34     Eigen::MatrixXf H = Eigen::MatrixXf::Zero(m * z, n * z);
35     for (int i = 0; i < m; ++i) {
36         for (int j = 0; j < n; ++j) {
37             if (B(i, j) == -1) {
38                 fillBlockWithZeros(H, i * z, j * z, z);
39             } else {
40                 applyShiftedIdentityToH(H, i, j, B, z);
41             }
42         }
43     }
44     return H;
45 }
```

Listing 4.5: Base matrix expansion on CPU using Eigen

4.3.1.2 GPU Implementation

In the *GPU* implementation, the base matrix expansion is performed inside a CUDA kernel (*Listing* 4.6). The kernel `expandBaseMatrix` calculates the corresponding entry in H based on thread and block indices. Each GPU thread processes one element of the expanded matrix H. If the corresponding base matrix entry is -1, the output is set to zero. Otherwise, a circulant shift condition is used to determine whether to place a 1 or 0 at a given position. This parallel approach significantly accelerates the expansion step by utilizing the massive number of *GPU* cores.

```
__global__ void expandBaseMatrix(int* d_baseMatrix, double* d_H, int z, int m, int n) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m * z && col < n * z) {
        int baseMatrixRow = row / z;
        int baseMatrixCol = col / z;
        int shift = d_baseMatrix[baseMatrixRow * n + baseMatrixCol];

        if (shift == -1) {
            d_H[row * (n * z) + col] = 0.0;
        } else {
            int localRow = row % z;
            int localCol = col % z;
            d_H[row * (n * z) + col] = (localCol == (localRow + shift) % z) ? 1.0 :
                0.0;
        }
    }
}
```

Listing 4.6: Base matrix expansion on GPU using CUDA

4.4 Partitioning the Expanded Matrix H

After expanding the base matrix, the next step in the encoding process is to partition the expanded matrix H into two separate matrices, H_1 and H_2. This partitioning organizes the matrix into a form that simplifies later operations such as computing the parity bits. The matrix H_1 contains the part related to the original message bits, while H_2 relates to the parity bits. This procedure is performed similarly on both the *CPU* and *GPU*, although the implementations differ in terms of tools and approaches.

4.4.1 CPU Implementation

On the CPU, the partitioning is handled by the `splitHMatrix` function, as shown in *Listing* 4.7. The function receives the expanded matrix H along with dimensions m, n, and z.

- It first calculates the size of the message bits region.
- It then assigns the left part of H to a dense matrix H_1.
- Finally, it iterates over the remaining columns and selectively inserts nonzero elements into a sparse matrix H_2.

```
1  std::pair<Eigen::MatrixXf, Eigen::SparseMatrix<float>> splitHMatrix(const Eigen::
       MatrixXf& H, int m, int n, int z) {
2      int message_bit_size = (n - m) * z;
3      Eigen::MatrixXf H1(m * z, message_bit_size);
4      Eigen::SparseMatrix<float> H2(m * z, n * z - message_bit_size);
5
6      H1 = H.leftCols(message_bit_size);
7
8      for (int i = 0; i < m * z; ++i) {
9          for (int j = message_bit_size; j < n * z; ++j) {
10             if (H(i, j) != 0) {
```

```
                H2.insert(i, j - message_bit_size) = H(i, j);
            }
        }
    }

    return {H1, H2};
}
```

Listing 4.7: Partitioning the expanded matrix on the CPU

4.4.2 GPU Implementation

On the GPU, the partitioning is performed using two separate kernels: `copyToH1` and `copyToH2`, as shown in *Listings* 4.8 and 4.9. These kernels copy elements to H_1 and H_2 respectively.

The `splitParityCheckMatrix` function (*Listing* 4.10) launches these two kernels after setting the appropriate grid and block dimensions.

- `copyToH1` copies the first columns corresponding to the message bits directly.
- `copyToH2` selectively copies nonzero elements from the remaining columns to H_2.
- Both operations are synchronized after launching to ensure completion.

```
__global__ void copyToH1(const double* H, double* H1, int m, int n, int z, int
    msg_bit_size) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m * z && col < msg_bit_size) {
        H1[row * msg_bit_size + col] = H[row * n * z + col];
    }
}
```

Listing 4.8: Copying to H_1 on the GPU

```
__global__ void copyToH2(const double* H, double* H2, int m, int n, int z, int
    msg_bit_size) {
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    if (row < m * z && (col >= msg_bit_size && col < n * z)) {
        double value = H[row * n * z + col];
        if (value != 0) {
            H2[row * (n * z - msg_bit_size) + (col - msg_bit_size)] = value;
        }
    }
}
```

Listing 4.9: Copying to H_2 on the GPU

```
void splitParityCheckMatrix(const double* d_H, double* d_H1, double* d_H2, int m, int n
    , int z) {
    int msg_bit_size = (n - m) * z;

    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks((n * z + 15) / 16, (m * z + 15) / 16);

    copyToH1<<<numBlocks, threadsPerBlock>>>(d_H, d_H1, m, n, z, msg_bit_size);
    copyToH2<<<numBlocks, threadsPerBlock>>>(d_H, d_H2, m, n, z, msg_bit_size);

    cudaDeviceSynchronize();
}
```

Listing 4.10: Partitioning function for the GPU

4.5 Computing the Parity Sum

After partitioning the expanded matrix, the next step is to compute the intermediate parity sum. The parity sum is obtained by multiplying the matrix H_1 with the message vector. Each resulting element is then floored and reduced modulo 2 to prepare it for the next stage of the encoding process. This step is performed

both on the *CPU* and the *GPU*, using slightly different methods but achieving the same output.

4.5.1 CPU Implementation

On the *CPU*, the computation of the parity sum is done in a compact form as shown in *Listing* 4.11. The matrix H_1 is multiplied with the message vector. The resulting values are floored and then reduced modulo 2 using an element-wise operation.

```
Eigen::VectorXf parity_sum = (H1 * msg).array().floor().unaryExpr([](float x)
{ return fmodf(x, 2); });
```

Listing 4.11: Computing the parity sum on the CPU

4.5.2 GPU Implementation

On the *GPU*, a kernel function called `computeParitySum` is responsible for this operation, as shown in *Listing* 4.12. Each thread processes one row of the matrix H_1, calculating the sum of products between the matrix row and the message vector elements. Afterward, the sum is floored and reduced modulo 2.

```
__global__ void computeParitySum(double* H1, double* msg, double* result, int rows, int
     cols) {
    int row = blockIdx.x * blockDim.x + threadIdx.x;
    if (row < rows) {
        double sum = 0.0;
        for (int col = 0; col < cols; ++col) {
            sum += H1[row * cols + col] * msg[col];
        }
        double floored = floor(sum);
        result[row] = fmod(floored, 2.0);
    }
}
```

Listing 4.12: Computing the parity sum on the GPU

4.6 Computing the Parity Bits

After computing the parity sum, the next step in the encoding process is to solve for the parity bits. The parity bits are determined by solving a linear system where the sparse matrix H_2 and the previously computed parity sum are involved. This operation is performed differently on the *CPU* and *GPU*, as shown below.

4.6.1 CPU Implementation

On the *CPU*, the function `computeParityBits` (*Listing* 4.13) first computes the parity sum by multiplying H_1 and the message vector. Then, it uses Eigen's `SparseLU` solver to solve the sparse linear system $H_2 \times p = \text{parity\_sum}$. After solving, the solution is rounded and reduced modulo 2.

```
Eigen::VectorXf computeParityBits(
    const Eigen::MatrixXf& H1, const Eigen::SparseMatrix<float>& H2,
    Eigen::VectorXf& msg)
{
    Eigen::VectorXf parity_sum = (H1 * msg).array().floor().unaryExpr([](float x) {
        return fmodf(x, 2); });

    Eigen::SparseLU<Eigen::SparseMatrix<float>> solver;
    solver.compute(H2);

    if (solver.info() != Eigen::Success) {
        std::cerr << "Sparse LU decomposition failed!" << std::endl;
        return {};
    }

    Eigen::VectorXf p = solver.solve(parity_sum);
    Eigen::VectorXf p_mod2 = p.unaryExpr([](float x) { return fmodf(round(x), 2); });
    return p_mod2;
}
```

Listing 4.13: Solving for parity bits on the CPU

4.6.2 GPU Implementation

On the *GPU*, the system $H_2 \times p =$ parity_sum is solved using the cuSOLVER library's QR factorization method. The function `qr_solver` (*Listing* 4.14) sets up and solves the system by calling `cusolverSpDcsrlsvqr`. It uses CSR (Compressed Sparse Row) formatted data for efficiency on sparse matrices. The function `computeParityBits` (*Listing* 4.15) acts as a wrapper to call `qr_solver`.

```
void qr_solver(
    int n, int nnz,
    const int* d_csrRowPtrA,
    const int* d_csrColIndA,
    const double* d_csrValA,
    const double* d_b,
    double* d_x
) {
    cusolverSpHandle_t cusolver_handle;
    CUSOLVER_CHECK(cusolverSpCreate(&cusolver_handle));

    cusparseMatDescr_t descr;
    CUSPARSE_CHECK(cusparseCreateMatDescr(&descr));
    CUSPARSE_CHECK(cusparseSetMatType(descr, CUSPARSE_MATRIX_TYPE_GENERAL));
    CUSPARSE_CHECK(cusparseSetMatIndexBase(descr, CUSPARSE_INDEX_BASE_ZERO));

    double tol = 1e-10;
    int reorder = 1;
    int singularity;
    CUSOLVER_CHECK(cusolverSpDcsrlsvqr(
        cusolver_handle, n, nnz, descr,
        d_csrValA, d_csrRowPtrA, d_csrColIndA,
        d_b, tol, reorder, d_x, &singularity));

    if (singularity != -1) {
        printf("Warning: Matrix is singular at row %d\n", singularity);
    }
    cusparseDestroyMatDescr(descr);
    cusolverSpDestroy(cusolver_handle);
}
```

Listing 4.14: QR solver for sparse systems on the GPU

```
void computeParityBits(
    cusparseHandle_t handle,
    double *d_values,
    int *d_rowPtr,
    int *d_colInd,
    double *d_parity,
    double *d_P,
    int n,
    int nnz
) {
    qr_solver(n, nnz, d_rowPtr, d_colInd, d_values, d_parity, d_P);
}
```

Listing 4.15: Wrapper for solving parity bits on the GPU

4.7 Building the Codeword

After calculating both the message part and the parity part, the final step is to combine them to form the complete encoded codeword. This process involves concatenating the message and parity vectors and applying modulo 2 operation to each element to ensure the codeword is binary.

4.7.1 CPU Implementation

On the CPU, the codeword is built by creating a new vector large enough to hold both the message and parity bits, as shown in *Listing* 4.16. The values from the message and parity vectors are copied and stored sequentially.

```
Eigen::VectorXf codeword(msg.size() + p.size());
```

Listing 4.16: Building the codeword on the CPU

Here, `msg` refers to the message vector, and `p` refers to the computed parity vector.

4.7.2 GPU Implementation

On the GPU, the function `concatenateVectors` (*Listing* 4.17) performs the concatenation of the message and parity vectors while simultaneously applying the rounding and modulo 2 operations to each element. This ensures that all values in the resulting codeword are either 0 or 1.

```
__global__ void concatenateVectors(
    const double* vec1, const double* vec2, double* result,
    int size1, int size2)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    int totalSize = size1 + size2;

    if (idx < totalSize) {
        if (idx < size1) {
            double val = fabs(vec1[idx]);
            double rounded = round(val);
            result[idx] = fmod(rounded, 2.0);
        } else {
            double val = fabs(vec2[idx - size1]);
            double rounded = round(val);
            result[idx] = fmod(rounded, 2.0);
        }
    }
}
```

Listing 4.17: Concatenating vectors and applying modulo 2 on the GPU

The kernel treats the two input vectors separately but stores their results in a single output vector. This ensures that the codeword is correctly assembled on the GPU without requiring extra memory copies between the device and host.

4.8 Cross-Validation with MATLAB Output

To ensure the correctness of the *C++* and *CUDA* implementations, the generated codewords are compared with the reference codewords produced by a *MATLAB*

encoder. MATLAB's `isequal` function is used to check if the two vectors are identical element-wise.

4.8.1 Validating C++ Output

In the C++ validation, the codeword generated by the Eigen-based encoder is exported to a file in a shared workspace, from which it can be loaded and validated using MATLAB's `isequal` function. An example of this validation using MATLAB's `isequal` function is shown in *Listing* 4.18.

```
is_equal = isequal(codeword_cpp, codeword_matlab);
if is_equal
    disp('C++ codeword matches MATLAB codeword.');
else
    disp('Mismatch between C++ and MATLAB codewords.');
end
```

Listing 4.18: Validating C++ codeword against MATLAB codeword

4.8.2 Validating CUDA Output

Similarly, the CUDA-accelerated codeword is transferred back from the GPU to the host memory, exported to a file in a shared workspace, and then compared against the MATLAB result. *Listing* 4.19 shows the corresponding MATLAB validation.

```
is_equal = isequal(codeword_cuda, codeword_matlab);

if is_equal
    disp('CUDA codeword matches MATLAB codeword.');
else
    disp('Mismatch between CUDA and MATLAB codewords.');
end
```

Listing 4.19: Validating CUDA codeword against MATLAB codeword

Chapter 5

Evaluation and Result

In this chapter, we present and analyze the results from evaluating the performance of our LDPC encoder for 5G NR. Although LDPC encoding is computationally less intensive than decoding, the increasing block lengths specified by the 5G standard—reaching up to 8448 bits for certain configurations—introduce additional computational complexity, particularly in systems that require repeated or real-time encoding [21].

While performance can be assessed through factors like memory usage or energy efficiency, this project focuses on execution time due to its practical relevance in 5G systems and time constraints. In this context, execution time refers specifically to the time taken by the main LDPC encoding routine. As discussed in Chapter 3, the measurement excludes initial setup stages such as processing input data and initializing data structures. This ensures that the reported times isolate the performance of the encoding computation itself, without being affected by surrounding preparatory steps.

To ensure fairness, both implementations used similar algorithmic structures and were tested under the same conditions: the CPU version was written in *C++*

using the *Eigen* library, and the GPU version in *CUDA* using *NVIDIA* hardware. The evaluation spans various block lengths by adjusting base graphs and lifting sizes as defined in the 5G NR specification. Execution times were compared to understand how performance scales with message size and to assess the practical benefits of GPU acceleration.

5.1 Experiment Setup

The experiments were conducted on a shared machine, with tests scheduled during periods of low system activity to minimize the impact of other users. For each configuration, multiple execution times were collected, and statistical measures including the minimum, maximum, average, and variance were calculated. While running experiments on shared hardware introduces potential uncertainties such as CPU and GPU resource contention, GPU scheduling delays, and background processes, which can cause non-deterministic fluctuations in execution time—especially for GPU-based encoding—the consistently low variance observed across repeated runs suggests that these effects were limited. Therefore, the reported averages are considered reliable and representative for comparative analysis.

The system used for the experiments featured an Intel Core i9-11900 processor with 8 cores, 16 threads, and a clock speed ranging from 2.5 to 5.2 GHz. It was equipped with 64 GB of DDR4 RAM, consisting of two 32 GB CL16 XMP 2.0 modules. Graphics processing was handled by an NVIDIA GeForce RTX 3090 OC GPU, which includes 24 GB of GDDR6X memory and 10,496 CUDA cores, based on the Ampere architecture. The software environment included GCC 11.4.0, CUDA Toolkit 12.8, and MATLAB 2024b.

5.2 Result and Analysis

The execution time results for LDPC encoding on both CPU and GPU platforms are summarized below, based on a range of message block sizes derived from the 5G NR base graphs. Experiments were performed using both Base Graph 1 (BG1) and Base Graph 2 (BG2), across multiple lifting sizes z. The lifting size directly affects the final codeword length and, consequently, the computational complexity of the encoding process.

5.2.1 Execution Time Measurements

As presented in *Table* 5.1, encoding times for BG1 exhibit a clear upward trend as the Message block size increases. On the CPU, the smallest tested sizes (44 and 176 bits) required an average of 3.44 ms and 11.84 ms, respectively—demonstrating relatively low overhead for lightweight encoding tasks. However, performance degraded rapidly with increasing block size. At 704 bits, the average encoding time increased to approximately 110.5 ms, while 2816 bits took nearly 1968 ms. For the largest case, 5632 bits, the CPU execution time exceeded 14 seconds, reflecting the high computational burden of large matrix operations in serial processing.

Message size (bits)	Avg runtime (ms)	Min runtime (ms)	Max runtime (ms)
44	3.44	2.43	4.19
176	11.84	11.05	14.97
704	110.52	108.76	113.97
2816	1967.98	1902.59	2045.36
5632	14167.7	14091.4	14223.2

Table 5.1: LDPC Encoding Execution Times for Base Graph 1 (BG1) - CPU

On the GPU, as shown in *Table* 5.2, encoding performance scaled much more efficiently. For the smallest case (44 bits), the GPU achieved a modest improvement

over the CPU, with an average execution time of 2.60 ms. But as the block size increased, the GPU's advantage became more pronounced. At 2816 bits, the encoding time dropped to just 48.4 ms—roughly 40× faster than the CPU. For 5632 bits, the GPU completed the task in 170.7 ms, delivering a speedup of over 80× compared to the CPU. This trend highlights the GPU's ability to exploit parallelism more effectively for larger workloads, where matrix operations benefit significantly from hardware acceleration.

Message size (bits)	Avg runtime (ms)	Min runtime (ms)	Max runtime (ms)
44	2.60	2.54	2.81
176	3.21	3.12	3.41
704	7.14	7.07	7.31
2816	48.42	47.04	53.29
5632	170.67	169.84	171.69

Table 5.2: LDPC Encoding Execution Times for Base Graph 1 (BG1) - GPU

The results for Base Graph 2 (BG2) are shown in *Table* 5.3 for the CPU and *Table* 5.4 for the GPU. As with BG1, execution times increased with message size across both platforms. On the CPU, encoding times ranged from approximately 2.84 ms for the smallest block size (20 bits) to over 11 seconds for the largest tested size (2560 bits), highlighting the steep performance cost of handling larger matrices in serial.

Message size (bits)	Avg runtime (ms)	Min runtime (ms)	Max runtime (ms)
20	2.84	1.98	3.50
80	10.17	9.65	11.12
320	90.87	89.71	92.97
1280	1560.61	1495.38	1689.41
2560	11134.9	11256.7	11393.8

Table 5.3: LDPC Encoding Execution Times for Base Graph 2 (BG2) - CPU

While the GPU advantage was relatively modest for very small block sizes (e.g., 20 or 80 bits), it became increasingly significant as the block length grew. For instance, at 1280 bits, the CPU required an average of 1.56 seconds, while the

GPU completed the same task in only 61.25 ms—yielding a speedup of over 25×. Even at the maximum tested size of 2560 bits, the GPU maintained strong efficiency, completing encoding in under a quarter of a second compared to more than 11 seconds on the CPU. These results further confirm that GPU acceleration offers substantial performance benefits for LDPC encoding, particularly in scenarios involving medium to large block sizes.

Message size (bits)	Avg runtime (ms)	Min runtime (ms)	Max runtime (ms)
20	2.68	2.58	2.89
80	3.48	3.45	3.58
320	8.91	8.84	9.20
1280	61.25	59.86	66.94
2560	231.30	230.39	232.36

Table 5.4: LDPC Encoding Execution Times for Base Graph 2 (BG2) - GPU

An interesting observation from the results is that the GPU encoding times for base graph 2 (BG2) are consistently higher than those for base graph 1 (BG1), despite BG2 having a smaller base matrix dimension (42×52 compared to BG1's 46×68) and being associated with shorter information block lengths. This highlights that the encoding complexity is not solely determined by the base matrix size.

One likely explanation lies in the structure and sparsity pattern of the parity-check matrices after expansion. BG2 may result in denser or less regular matrices, particularly in the H2 submatrix used during parity calculation, making operations like matrix multiplication and solving systems of equations more computationally demanding. Additionally, BG2's structure might lead to less efficient utilization of GPU resources, with more thread divergence or memory-bound operations. These factors combined contribute to higher average execution times for BG2, even though the input sizes are smaller. This insight underscores the importance of considering matrix structure and algorithmic efficiency—not just matrix dimensions—when evaluating LDPC encoder performance.

5.2.2 Graphical Analysis

The graphical analysis of LDPC encoding times across Base Graph 1 (BG1) and Base Graph 2 (BG2) reveals clear differences in how execution time scales with message size on both CPU and GPU platforms. For BG1, the CPU exhibits a sharply accelerating trend, as seen in *Figure* 5.1, where execution time increases gradually for smaller message sizes but begins to rise significantly after 704 bits. This steep upward curve signals a clear scalability bottleneck, as the CPU becomes increasingly inefficient for handling larger block lengths.

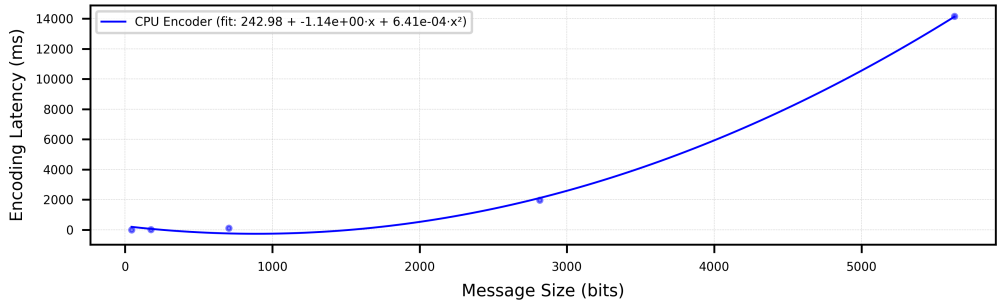

Figure 5.1: LDPC encoding times for BG1 on CPU.

Conversely, the GPU encoding times for BG1, shown in *Figure* 5.2, exhibit a stable, near-linear trend, with only a modest quadratic component.

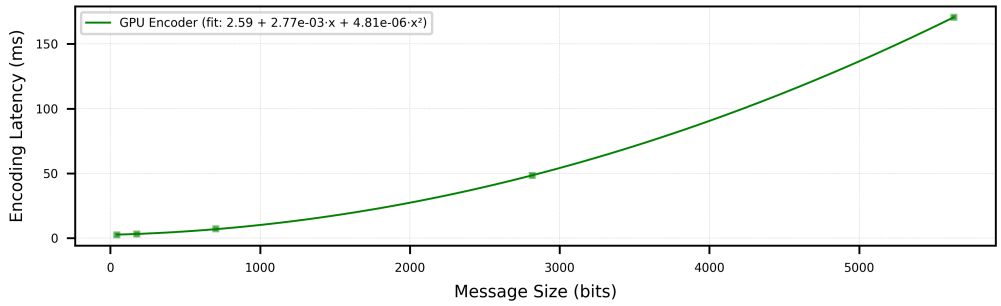

Figure 5.2: LDPC encoding times for BG1 on GPU.

This illustrates the GPU's ability to maintain predictable performance even as

the workload scales, helpful in scenarios where timing consistency is preferred.

Figure 5.3 highlights how CPU and GPU encoding times for BG1 grow with message size. While both increase following a quadratic pattern, the CPU's time climbs sharply, showing slower performance as messages get larger. The GPU's curve stays much flatter, meaning it handles bigger workloads much more smoothly—for instance, at 5632 bits, the GPU takes about 170 ms to encode, whereas the CPU needs over 14 seconds.

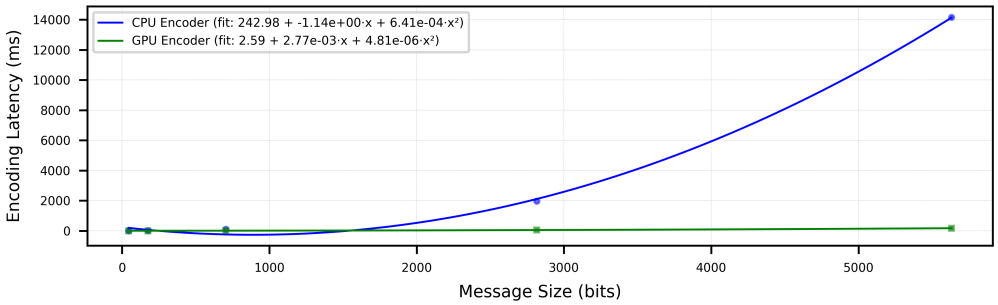

Figure 5.3: CPU vs. GPU encoding times for BG1.

Building on the analysis of Base Graph 1 (BG1), the results for Base Graph 2 (BG2) further highlight the differences between CPU and GPU performance in LDPC encoding. For smaller message sizes, the execution times on both CPU

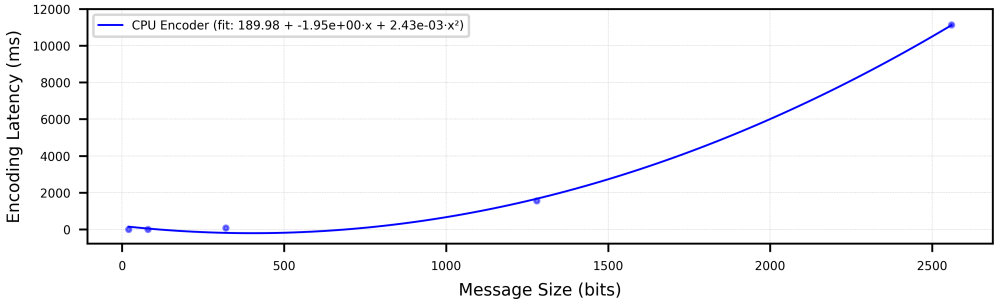

Figure 5.4: LDPC encoding times for BG2 on CPU.

and GPU are quite similar—any differences are minimal and can be considered

negligible. However, as the message size increases, these differences become much more noticeable. In *Figure* 5.4, the CPU's execution time starts off gradually but increases sharply after 704 bits, indicating a clear scalability issue.

In contrast, the GPU's encoding time for BG2, shown in *Figure* 5.5, grows steadily in a near-linear fashion, demonstrating its ability to handle larger workloads efficiently. This contrast is clear in *Figure* 5.6, where the CPU's curve rises steeply while the GPU remains relatively flat.

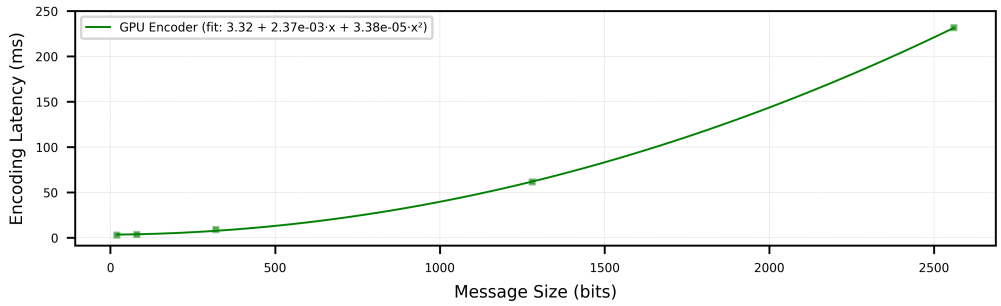

Figure 5.5: LDPC encoding times for BG2 on GPU.

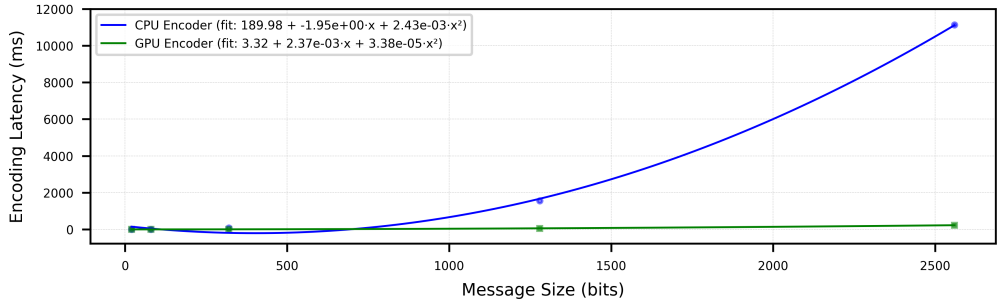

Figure 5.6: CPU vs. GPU encoding times for BG2.

The graphical analysis highlights how CPU and GPU execution times scale differently as message size increases. While their performance is almost similar for small blocks, the CPU's encoding time grows sharply with larger messages, whereas the GPU's time scales more smoothly and efficiently across both BG1 and BG2.

Chapter 6

Conclusions and Future Work

This study investigated the performance of LDPC encoding in 5G NR by implementing and evaluating both CPU-based and GPU-accelerated encoders. The results highlight the significant advantages of GPU acceleration, particularly for larger block sizes. While the CPU implementation using the Eigen library was adequate for small messages, it exhibited quadratic growth in execution time as the block size increased. In contrast, the CUDA-based GPU implementation delivered consistently low and predictable runtimes, achieving speedups of over 80× in some cases.

These findings emphasize the practicality of GPU acceleration in real-time and high-throughput 5G systems, where fast encoding is essential. Notably, the performance gains were influenced not only by message size but also by the structural characteristics of the parity-check matrices. Although Base Graph 2 (BG2) involves smaller input sizes than Base Graph 1 (BG1), it occasionally incurred higher runtimes due to its less favorable matrix structure. This suggests that matrix sparsity and regularity play a crucial role in determining encoding efficiency.

However, the study has several limitations. The evaluation focused exclusively

on execution time, omitting other critical metrics such as energy consumption, memory usage, and latency variability. Additionally, experiments were conducted on a shared system, where background processes may have influenced the results despite efforts to monitor variance.

6.1 Discussion

In the development of the LDPC encoder, both the CPU and GPU implementations presented unique challenges and learning opportunities. The transition from the CPU to the GPU encoder highlighted the importance of parallelization, as the CPU implementation was limited by its single-threaded nature. The CPU encoder performed well for smaller message sizes but scaled poorly with larger block sizes due to its sequential nature.

When implementing the GPU-based encoder, we initially focused on translating the CPU algorithm into a parallel version, which was not always straightforward. Early versions of the GPU encoder used simple memory access patterns that were inefficient. We learned that optimizing memory usage, especially by minimizing global memory accesses and better utilizing shared memory, was crucial for improving performance. As we refined the CUDA implementation, we began to see large performance gains, particularly for larger block sizes.
To further improve performance, we utilized CUDA streams to enable concurrent execution of memory transfers and kernel launches. CUDA streams allowed data transfer and computation to run simultaneously, reducing idle time and boosting overall performance.

Additionally, we made use of optimized routines from CUDA libraries like cuSolver and cuSPARSE to handle complex matrix operations. For example, cuSolver's QR factorization was used to compute parity bits efficiently, leveraging its reliable

and well-optimized design. This allowed us to focus more on fine-tuning the encoder logic rather than rebuilding existing solutions from scratch.

6.2 Relevance to Existing Solutions

While the studies reviewed in Chapter 2 introduce highly efficient LDPC encoder designs implemented on specialized hardware platforms such as ASICs, FPGAs, and performance-tuned GPUs, a direct performance comparison with our implementation is limited by several factors. Most of the referenced approaches are tailored to specific hardware constraints and benefit from custom circuit-level optimizations or vendor-specific enhancements that are not applicable to our general-purpose CUDA-based solution. Moreover, those implementations are often part of complete, production-grade encoding systems, whereas our work is primarily exploratory, aimed at evaluating the effectiveness of GPU parallelization using standardized 5G NR base matrices.

Nonetheless, our findings exhibit trends consistent with those reported in prior work — particularly with respect to the efficiency gains offered by parallel computation. The observed speedup over CPU-based encoding in our experiments reinforces the viability of GPU acceleration for LDPC encoding and highlights the potential for further optimization in future work.

6.3 Future Work

The evaluation of the GPU-accelerated QC-LDPC encoder demonstrated significant performance improvements over the CPU-based implementation, particularly for larger block sizes. However, several opportunities for further optimization and exploration remain. This section outlines potential directions for future

work to enhance the efficiency, scalability, and practicality of QC-LDPC encoding for 5G NR applications.

6.3.1 Optimizing Base Graph Expansion

In the current implementation, the parity-check matrix H is expanded from the base graph (BG1 or BG2) during the encoding process, which involves replacing each entry in the base matrix with a $z \times z$ circulant permutation matrix or zero matrix. This expansion step, performed on both CPU and GPU, is computationally expensive and contributes significantly to the overall execution time, especially for large lifting sizes z.

A promising direction for future work is to explore methods that avoid explicit expansion of H. For instance, encoding could be performed directly on the compact base graph representation by leveraging the cyclic structure of QC-LDPC codes. Techniques such as shift-based operations or precomputed lookup tables for circulant shifts could reduce the need for constructing the full H matrix, potentially lowering computational overhead and memory requirements. Investigating such approaches could lead to faster encoding times, making the encoder more suitable for real-time 5G applications.

6.3.2 Converting H2 to CSR Outside the Encoder

In the GPU implementation, the H2 submatrix is converted to Compressed Sparse Row (CSR) format during encoding to enable efficient sparse matrix operations, as required by the QR factorization routine in the cuSOLVER CUDA library. While this conversion enables optimized computation, it introduces additional runtime overhead within the encoder workflow.

A potential optimization is to perform the CSR conversion of H2 offline, prior to

the encoding process. By precomputing and storing the CSR representation of H2 for specific base graphs and lifting sizes, the encoder could bypass this step during runtime, reducing execution times. Future work could investigate the trade-offs of this approach, including the storage requirements for precomputed CSR data and its impact on encoding performance across various 5G NR configurations. This optimization could be particularly beneficial for systems requiring frequent encoding with fixed base graphs.

6.3.3 Improving Memory Usage

Memory usage plays a big role in the performance of QC-LDPC encoding, especially because 5G NR uses very large matrices. In the current GPU implementation, the expanded H, H1, and H2 matrices are stored in global memory, which can slow things down when the lifting size is large. To improve this, future work could focus on better memory handling. For example, using in-place matrix operations could help avoid extra memory allocations during matrix splitting and parity bit calculation. Reusing memory buffers or using memory pooling across encoding rounds could also make the process more efficient. Another idea is to take better advantage of the sparsity of QC-LDPC matrices by writing custom CUDA kernels that work directly on sparse data. This could lower memory usage and make better use of the GPU cache. These improvements would help the encoder scale better for very large data blocks and systems with limited resources.

6.3.4 Investigating Energy Consumption

As 5G networks grow, understanding the energy consumption of high-performance computing systems becomes increasingly important due to their environmental impact. Although the GPU implementation of the QC-LDPC encoder offers significant

speed improvements, GPUs typically consume more power than CPUs. Future work could focus on measuring and analyzing the energy usage of the encoder when running on GPUs. This could involve using profiling tools such as *NVIDIA Nsight* Systems, *NVIDIA Management Library* (NVML), or NVIDIA's *nvidia-smi* to monitor real-time power draw, energy consumption, and GPU utilization. By collecting and comparing energy usage across different lifting sizes, block lengths, and kernel settings, it becomes possible to better understand the trade-offs between performance and power consumption. This kind of analysis would support more balanced and sustainable use of GPU resources in 5G systems.

Bibliography

[1] Cedric Dehos, Jose Luis González, Antonio De Domenico, Dimitri Kténas, and Laurent Dussopt. Millimeter-wave access and backhauling: the solution to the exponential data traffic increase in 5g mobile communications systems? *IEEE Communications Magazine*, 52(9):88–95, 2014.

[2] Ericsson. Ericsson mobility report, November 2024. Available at: `https://www.ericsson.com/4adb7e/assets/local/reports-papers/mobility-report/documents/2024/ericsson-mobility-report-november-2024.pdf` (visited on 26/02/2025).

[3] Jun Xu, Changqing Yang, and Yifei Yuan. *Channel Coding in 5G New Radio*. CRC Press, 2023.

[4] Tram Thi Bao Nguyen, Tuy Nguyen Tan, and Hanho Lee. Efficient qc-ldpc encoder for 5g new radio. *Electronics*, 8(6):668, 2019.

[5] NVIDIA Corporation. CUDA C Programming Guide, Version 12.4, February 2025. Available at: `https://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`. Visited on 2025-02-27.

[6] Sparsh Mittal and Jeffrey S Vetter. A survey of methods for analyzing

and improving gpu energy efficiency. *ACM Computing Surveys (CSUR)*, 47(2):1–23, 2014.

[7] Jan van Dijk. *The Digital Divide*. John Wiley & Sons, 2020.

[8] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, pages 126–130. sn, 2002.

[9] John G. Proakis. *Digital Communication*. McGraw-Hill, 4th edition, 2001.

[10] Olivia Hedlund. Parallelized qc-ldpc decoder ona gpu: An evaluation targeting ldpc codes adhering to the 5g standard, 2024.

[11] Muhammad Daniel Alif Jumairi. Challenges in data communication and networking. *Authorea Preprints*, 2023.

[12] Mohammad Rowshan, Min Qiu, Yixuan Xie, Xinyi Gu, and Jinhong Yuan. Channel coding towards 6g: Technical overview and outlook. *IEEE Open Journal of the Communications Society*, 2024.

[13] Alberto Sendin, Javier Matanza, and Ramon Ferrús. *Smart Grid Telecommunications: Fundamentals and Technologies in the 5G Era*. John Wiley & Sons, 2021.

[14] Stephen B Wicker. *Error control systems for digital communication and storage*, volume 1. Prentice hall Englewood Cliffs, 1995.

[15] Mody Sy. Demystifying 5g polar and ldpc codes: A comprehensive review and foundations. *arXiv preprint arXiv:2502.11053*, 2025.

[16] Robert G Maunder. The 5g channel code contenders. 2016.

[17] Robert Gallager. Low-density parity-check codes. *IRE Transactions on information theory*, 8(1):21–28, 1962.

[18] David JC MacKay. Good error-correcting codes based on very sparse matrices. *IEEE transactions on Information Theory*, 45(2):399–431, 1999.

[19] Shu Lin. Error control coding: Fundamentals and applications. *Prentice Hall, Inc google schola*, 2:300–303, 1983.

[20] Cheng-Hung Lin, Hsin-Hao Su, Tang-Syun Chen, and Cheng-Kai Lu. Reconfigurable low-density parity-check (ldpc) decoder for multi-standard 60 ghz wireless local area networks. *Electronics*, 11(5), 2022.

[21] 3GPP. 5G NR; Multiplexing and Channel Coding. Technical Report TS 38.212, 3rd Generation Partnership Project (3GPP), June 2020. (Visisted on 16/03/2025).

[22] Marc PC Fossorier. Quasicyclic low-density parity-check codes from circulant permutation matrices. *IEEE transactions on information theory*, 50(8):1788–1793, 2004.

[23] Thomas J Richardson and Rüdiger L Urbanke. The capacity of low-density parity-check codes under message-passing decoding. *IEEE Transactions on information theory*, 47(2):599–618, 2002.

[24] Tsern-Huei Lee and SJ Liu. Banyan network nonblocking with respect to cyclic shifts. *Electronics Letters*, 27(16):1474–1476, 1991.

[25] Yuta Iketo and Takayuki Nozaki. Efficient encoding algorithm of binary and non-binary ldpc codes using block triangulation. *arXiv preprint arXiv:2103.01560*, 2021.

[26] John D Owens, Mike Houston, David Luebke, Simon Green, John E Stone, and James C Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[27] NVIDIA Corporation. *CUDA C++ Programming Guide*, Jan. 2024. Accessed: 2025-03-07.

[28] Mark Harris. How to optimize data transfers in cuda c/c++, December 2012. Available at: https://developer.nvidia.com/blog/how-optimize-data-transfers-cuda-cc/. Visited on 2025-06-10.

[29] Shixin Liao, Yueying Zhan, Ziyuan Shi, and Lei Yang. A high throughput and flexible rate 5g nr ldpc encoder on a single gpu. In *2021 23rd International Conference on Advanced Communication Technology (ICACT)*, pages 29–34. IEEE, 2021.

[30] Yunke Tian, Yong Bai, and Dake Liu. Low-latency qc-ldpc encoder design for 5g nr. *Sensors*, 21(18):6266, 2021.

[31] 3GPP. 5G System Overview. https://www.3gpp.org/technologies/5g-system-overview, 2023. Accessed on 15/04/2025.

[32] Manuts Vasquez. Nr-ldpc-bg: Base graphs for 5g nr ldpc. https://github.com/manuts/NR-LDPC-BG, 2020. Accessed: 2025-04-25.

[33] Eigen: A c++ template library for linear algebra. Available at: https://eigen.tuxfamily.org/index.php?title=Main_Page (visited on 10/06/2025).