Parallelized QC-LDPC Encoder on an NVIDIA GPU

An Evaluation of LDPC Codes Compliant with the 5G Standard

Mustafah Rahimi
Jonos Mirzazada

Under the supervision of **Bengt Hallinger**

Faculty of Health, Science and Technology, Karlstad University

2025-06-26

tietoevry

Outline

Background and motivation

Method and implementation

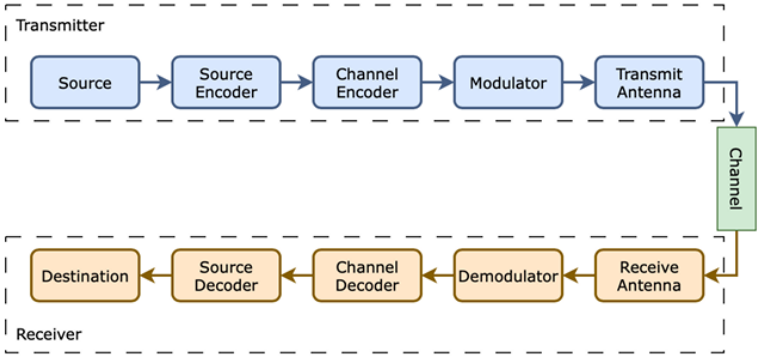
Important results

Future work

Background: The emergence of 5G and channel coding

- **5G growth:** By 2030, 5G subscriptions are expected to reach 6.3 billion and handle 80% of mobile data traffic (according to Ericsson's annual report Nov 2024).

- Peak data rates up to 20 Gbps
- Latency (RTT) as low as 1 ms
- 5G can connect up to 1 million devices per km^2

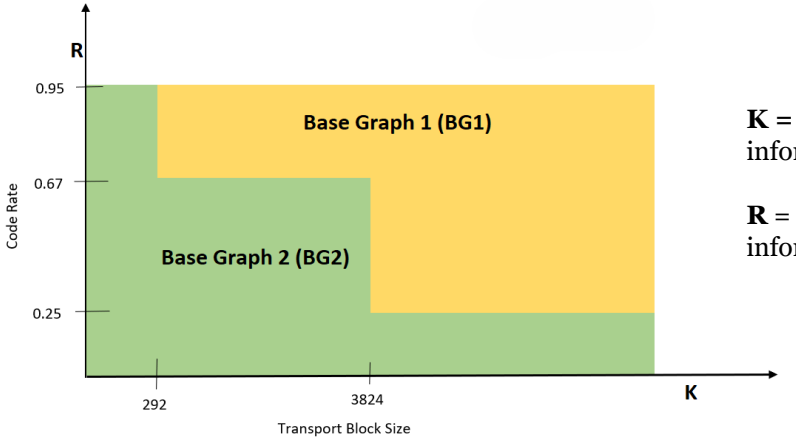


- **Channel coding:** Data sent over communication channels is often subject to noise and errors caused by ex. {physical obstacles, Electromagnetic inference, Multipath propagation (signal reflection) etc..}, Channel coding detects and corrects these errors and is essential for reliable transmission.

Error correction with LDPC codes in 5G NR

- **LDPC codes:** Low-Density Parity-Check codes offer performance close to the **Shannon limit**, used in 5G NR.

- LDPC codes (Data channel)
- Polar codes (control channel)
- Reed-Muller Codes
- Turbo codes



K = Transport block size (number of information bits before encoding)

R = Code rate i.e. the ratio of information bits to total coded bits:

$$R = \frac{k}{n}$$

- **QC-LDPC:** Quasi-cyclic LDPC codes provide structured, hardware-friendly coding for 5G and are based on two different base graphs, Base Graph 1 (46x68) and Base Graph 2 (42x52).

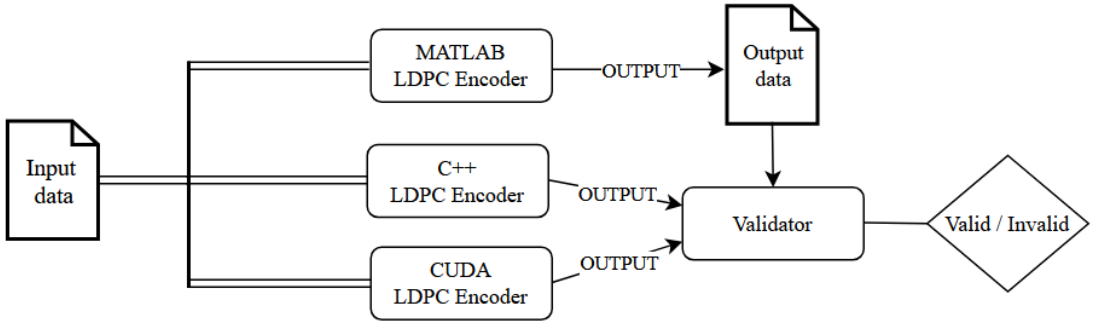
Example: QC-LDPC concept

- The basis matrix B is expanded into a parity check matrix H with lift size z.
- **Example:** Basis matrix $B = \begin{bmatrix} 0 & 1 & -1 \\ 1 & -1 & 0 \end{bmatrix}$, lifting size $z = 2$.
- **Expansion:**
- Value -1: Replaced with zero matrix $\begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$
- Value 0: Replaced with identity matrix I: $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$
- Value i > 0: Replaced with cyclically shifted identity matrix (shift in positions)
- For I = 1: $\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$ $(cyclic\ shift\ of\ I\ by\ one\ step\ to\ the\ right)$
- **Result:** $H = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$ $(4x6\ matris)$.

Motivation: Challenges with QC-LDPC coding

- **Computational requirements:** QC-LDPC encoding involves large matrix operations, challenging for CPUs.
- **Real-time requirements:** High-speed 5G systems require fast encoding to minimize latency.
- **GPU potential:** GPUs offer massive parallelism, ideal for computationally intensive and matrix-intensive tasks often found in signal processing and encoding.
- **Question:** How does parallel QC-LDPC encoding on GPU perform compared to sequential CPU encoding?

Project overview



- **Matlab Implementation:** Used as a reference to verify the results from the implementations in C++ and CUDA.
- **C++ Implementation:** Eigen was used for matrix and vector operations and STL for data structures and simpler vector operations.
- **CUDA Implementation:** cuBLAS for matrix multiplications, cuSPARSE for sparse operations, and cuSOLVER for QR factorization.

Performance Comparison: Base Graph 1 (BG1) & Base Graph 2 (BG2)

Table 1: Coding Times for BG1 (ms)

Message Size (bits)	CPU Average Time	CPU Improvement	GPU Average Time	GPU Improvement
44	3.44	-	2.60	1.3x
176	11.84	-	3.21	3.7x
704	110.52	-	7.14	15.5x
2816	1967.98	-	48.42	40.6x
5632	14167.7	-	170.67	83.0x

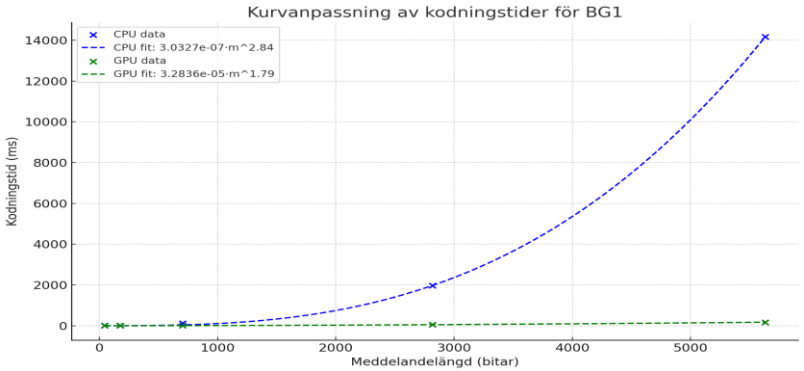
Table 2: Coding Times for BG2 (ms)

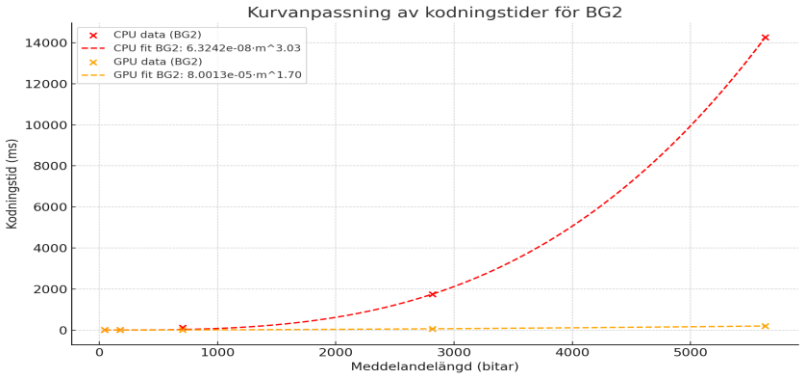
Message Size (bits)	CPU Average Time	CPU Improvement	GPU Average Time	GPU Improvement
20	2.84	-	2.68	1.1x
80	10.17	-	3.48	2.9x
320	90.87	-	8.91	10.2x
1280	1560.61	-	61.25	25.5x
2560	11134.9	-	231.30	48.2x

Each block size was run ten times with different randomly generated messages.

To compensate for the different initialization time, the first measurement was excluded, and the average was calculated from the nine subsequent runs.

Graphical analysis of the results





The graphical analysis shows clear differences in how CPU and GPU handle LDPC encoding as message size increases.

For small blocks, the execution times are roughly the same, but the CPU becomes significantly slower at larger sizes, while the GPU scales more evenly and efficiently for both BG1 and BG2.

CPU Optimizations and Bottlenecks

CPU Optimizations

- Used Eigen library for optimized dense and sparse matrix operations. For instance *Eigen::SparseMatrix<float>* Was used to store the H2 matrix for parity computation.
- Used *SparseLU* solver for efficient parity bit computation.
- The parity sum computation used Eigen's element-wise operations to floor and reduce modulo 2, which optimized the process by avoiding explicit loops and leveraging Eigen's optimized array operations.

CPU Bottlenecks

- CPUs sequential processing of large matrices leads to quadratic to almost cubic scaling (e.g., 14s for 5632 bits)
- Matrix expansion (*B* to *H)* overhead for large lifting sizes.
- Dense matrix operations increase memory and computational demands, Specifically, storing the expanded *H* matrix in dense format and using it directly when splitting *H* into *H1* and *H2*.
- Experiment ran on a shared system; low timing variance indicates minimal but noteworthy interference.

GPU Optimizations and Bottlenecks

GPU Optimizations

- The base matrix expansion was parallelized using a CUDA kernel – *expandBaseMatrix(B, msg)*, where each thread computed one element of the expanded matrix H.
- The *cuSOLVER* library was used to solve sparse linear systems and compute parity bits using an optimized *QR-factorization* routine for CSR-formatted matrices.
- The *cuSPARSE* library was implicitly utilized for sparse matrix operations.
- CUDA streams enabled overlapping memory transfers with computation, reducing idle time and improving throughput.

GPU Bottlenecks

- For small message sizes (e.g., 20–44 bits), GPU gains were modest due to CUDA launch and memory transfer overhead outweighing parallelism benefits.
- Using multiple small kernels introduced initialization overhead; combining them into larger kernels could improve performance in the future.
- Expanding the base matrix *B* into *H* within the main encoder routine caused a slowdown in the encoding process.
- Converting the *H2* submatrix to CSR format during encoding introduced runtime overhead, (necessary for *cuSOLVER's* sparse solver).

Key insights

- **GPU superiority:** GPU achieves up to 83× speedup for large block sizes (e.g. 5632 bits for BG1).
- **Scalability**: CPU encoding time grows exponentially, while GPU scales near linearly.
- **Impact of matrix structure:** BG2, despite its smaller size, has higher GPU encoding times due to the matrix structure.
- **Limitations:** Focus on execution time; energy consumption and memory usage not evaluated.

Main Challenges and Approaches to Solve Them

Challenges

- Initial use of *STL* for matrices/vectors required translation to linear algebra library formats, slowing execution.
- Different floating-point precisions in solver routines complicated output validation against *MATLAB*.
- Sequential CPU algorithm structure was challenging to parallelize for GPU's architecture.

Solutions

- Rewrote *CPU* code to use Eigen's data structures from the start, eliminating translation routines and improving performance.
- Experimented with multiple solver routines to align floating-point precision with MATLAB, ensuring complete validation.
- Started with developing small, separate CUDA kernels for each algorithm step, tested them, and combined into larger kernels where possible to not create a major performance bottleneck with several mini kernels.

Future work

- **Optimize Base Graph Extension:** Use shift-based operations or precomputed tables to avoid explicit matrix expansion.
- **Precompute CSR for H2:** Convert H2 to CSR format offline to reduce runtime overhead.
- **Additional code optimizations:** There is potential for further optimization of the code structure and algorithms.
- **Energy Analysis:** Profile energy consumption with tools like NVIDIA Nsight Systems and compute for sustainable 5G solutions.

Thank you for your attention!

We are happy to answer your questions.