Enabling High-Level Design Strategies for High-Throughput and Low-Power NB-LDPC Decoders

Srinivasan Subramaniyan
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Bengaluru 560035, India

Oscar Ferraz
Department of Electrical and Computer Engineering
Instituto de Telecomunicacoes
University of Coimbra
3030-290 Coimbra, Portugal

M. R. Ashuthosh and Santosh Krishna
Department of Electronics and Communication Engineering
PES Institute of Technology
Bangalore South Campus
Bengaluru 560100, India

Guohui Wang and Joseph R. Cavallaro
Department of Electrical and Computer Engineering
Rice University
Houston, TX 77005 USA

Vitor Silva and Gabriel Falcao
Department of Electrical and Computer Engineering
Instituto de Telecomunicacoes
University of Coimbra
3030-290 Coimbra, Portugal

Madhura Purnaprajna
Department of Computer Science and Engineering
Amrita Vishwa Vidyapeetham
Bengaluru 560035, India

Editor's notes:
Nonbinary low-density parity-check (NB-LDPC) codes enable reliable data transmission over noisy channels. This article presents efficient hardware design for LDPC codes.
—Partha Pratim Pande, Washington State University

Error-correcting codes (ECCs) play a significant role in modern-day communication systems for reducing the impact of errors over a transmission channel. Gallager [1] invented low-density parity-check (LDPC) codes in 1962; however, these codes were not implemented due to their complexity until the mid-1990s. Modern-day communication systems demand a bit error rate (BER) in the range of 10–15 as stated in [2]. Coding theory experts propose new codes to exploit lower BER and reduce the design complexity. Nonbinary LDPC (NB-LDPC) codes reach lower BER for short-to-moderate code word lengths when compared to the binary counterparts [3]. The main challenges in designing NB-LDPC codes are: 1) the higher computational complexity and the limited hardware resources; 2) increased irregular memory accesses; and 3)

Digital Object Identifier 10.1109/MDAT.2022.3202852
Date of publication: 29 August 2022; date of current version: 20 January 2023.

design space exploration has to be done efficiently to achieve high-throughput and low-power performance. This article analyzes the design advantages and their drawbacks. The analysis of theoretical peak performance for graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) are compared using a roofline model. Additionally, this work shows how high-level synthesis (HLS) and reconfigurable FPGA technology can be used for achieving high-throughput and low-power performance [4], which suits developers with limited hardware design knowledge.

The main contributions of this article are:

- Presenting the first results in the literature for GPU- and FPGA-based implementations of the min–max algorithm (MMA) for Galois field (GF) (2^4).
- Providing a detailed description of the GPU, HLS, and register transfer level (RTL) implementations.
- Sharing the open-source code for the GPU, HLS, and RTL implementations for each GF, thus providing three distinct parallel solutions available for the community to validate results and continue the research development.

Background

An LDPC code is a linear block code defined by a sparse parity check matrix (PCM) **H** or the equivalent Tanner graph representation. Figure 1 represents the overall block design for the LDPC decoder. For the system to transmit data, a message is encoded by a generator matrix and sent through a channel. During transmission, the encoded codeword (c) is prone to errors due to ambient noise. To correct these errors, LDPC decoders can be employed. When the message is received at the end of the channel, the variable nodes (VNs) are initialized with a probability of the corresponding received bit being "1" or "0" for the binary case. For the NB case, symbols are transmitted instead of bits. The decoding process works by exchanging those probabilities, in the form of messages, between check nodes (CNs) and VNs to calculate the most probable symbol. This exchange is executed until the check equation is valid ($\mathbf{H}^T\mathbf{c} = 0$) or until a fixed number of iterations between the CNs and VNs is reached.

The **H** matrix determines the connections between CNs and VNs. Equation (1) illustrates a binary representation of a **H** matrix with $M = 3$ rows (3 CNs) and $N = 6$ columns (6 VNs). If the nonzero elements per column and row are constant, the code is declared as regular; otherwise, it is irregular and has better coding performance but is computationally more complex. The row weights are represented using d_c and the column weights are represented using d_v and these are the number of nonzero entries of PCM for regular matrices. The following equation has a column weight of 2 ($d_v = 2$) and a row weight of 4 ($d_c = 4$):

$$\mathbf{H} = \begin{bmatrix} 1 & 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 1 \end{bmatrix}. \quad (1)$$

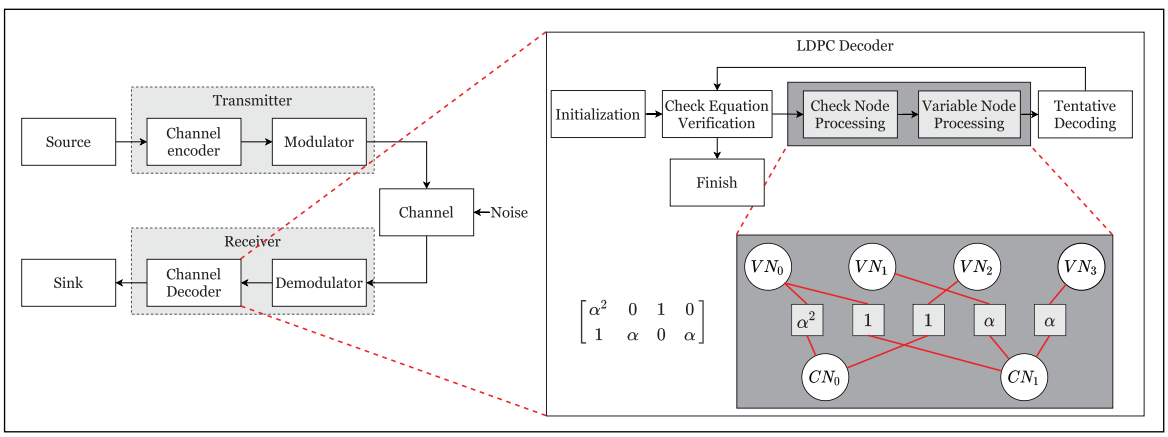

Figure 1. Overall transmission and receiving systems under noisy communication channels. The expansion box in the channel decoder represents the diagram blocks and corresponding Tanner graph of an NB-LDPC decoder.

NB-LDPC decoders are capable of achieving higher BER performance under a binary Gaussian channel when compared to binary LDPC codes described in [3]. Furthermore, NB-LDPC decoders show more computational complexity and consume more memory. However, these decoders achieve a better decoding capability at moderate code lengths. Contrary to binary LDPC codes, NB-LDPC calculates the probability of symbols in their respective GF instead of calculating "0" or "1." Using symbols increases the decoder's complexity exponentially as the operating field increases, requiring more resources when compared to binary LDPC codes.

The matrix in the following equation represents a PCM for an NB-LDPC code:

$$\mathbf{H} = \begin{bmatrix} \alpha^2 & 0 & 1 & 0 \\ 1 & \alpha & 0 & \alpha \end{bmatrix}. \quad (2)$$

Figure 1 shows a tanner graph representation of the **H** matrix (2), where the VNs represent the columns and CNs represent the rows of **H**. Instead of propagating probabilities of two digits (0 or 1), in NB-LDPC codes, the probabilities are propagated as 2^m symbols, for a GF of order 2^m. Similarly, the probabilities assigned to each symbol are initialized on the VNs and the symbols are propagated between VNs and CNs. Moreover, these probabilities are multiplied and divided by the corresponding symbol defined in the **H** matrix. Thus, the complexity of the hardware implementation of NB-LDPC decoders becomes higher as we increase the order of the field

Min–Max

V. Savin described the MMA [5] that proposed an approach for decoding NB-LDPC codes achieving even more low-complexity decoders by sacrificing error-correcting performance compared to the extended min-sum algorithm (EMSA). The algorithm is initialized by the probabilities given by the transmission channel in the log-likelihood ratio (LLR) domain. For each symbol transmitted, the decoding algorithm will calculate the probability for each symbol in GF*(q)* with $q = 2^m$, ensuring that the lowest calculated probability will be the most likely symbol to belong to that codeword [5]. From the given probabilities, the *a priori* information (γ_n(a)) and the VN messages ($\alpha_{m,n}(a)$) are calculated and initialized as shown in lines 2 and 3 of Algorithm 1. The row-wise CN processing is composed of the forward and backward matrices (*F*, *B*) calculation. *F* is initialized by the first element on each row (line 6) and calculates the next elements using the previously calculated ones (line 7). The *B* is calculated in the same way as the *F* but instead, the calculation starts from the last element to the first (lines 8 and 9). The first and last elements of the *B* and *F*, respectively, are attributed to the first and last elements of the CN message ($\beta_{m,n}(a)$) as shown in lines 10 and 11. The remaining values (line 12) are attributed by calculating the minimum of the maximum of different symbols of both *F* and *B*, following $a' + h_{m,n_i}\, a = a''$.

Algorithm 1. Min-max decoding algorithm 1.

1: {**Initialization:**}
2: $\gamma_n(a) = \frac{ln(Pr(c_n=s_n|channel))}{ln(Pr(c_n=a|channel))}$
3: $\alpha_{m,n}(a) = \gamma_n(a)$
4: **while** ($\mathbf{c_n}\,\mathbf{H}^T \neq \mathbf{0} \wedge$ it $<$ I) {c_n-decoded word; I-Max. no. of iterations.}
do
5: {**Check node processing:**}
6: $F_0(a) = \alpha_{m,n_0}(h^{-1}_{m,n_0} a)$
7: $F_i(a) = \min_{a'+h_{m,n_i}a''=a}(max(F_{i-1}(a'), \alpha_{m,n_i}(a'')))$
8: $B_{dc-1}(a) = \alpha_{m,n_{(dc-1)}}(h^{-1}_{m,n_{(dc-1)}} a)$
9: $B_i(a) = \min_{a'+h_{m,n_i}a''=a}(max(B_{i+1}(a'), \alpha_{m,n_i}(a')))$
10: $\beta_{m,n_0}(a) = B_1(h_{m,n_0} a)$
11: $\beta_{m,n_{(dc-1)}}(a) = F_{dc-2}(h_{m,n_{(dc-1)}} a)$
12: $\beta_{m,n_i}(a) = \min_{a'+h_{m,n_i}a=a''}(max(F_{i-1}(a''), B_{i+1}(a')))$
13: {**Variable node processing:**}
14: $\alpha'_{m,n}(a) = \gamma_n(a) + \sum_{m'\in M(n)\setminus\{m\}} \beta_{m',n}(a)$
15: $\alpha'_{m,n} = \min_{a\in GF(q)} \alpha'_{m,n}(a)$
16: $\alpha_{m,n}(a) = \alpha'_{m,n}(a) - \alpha'_{m,n}$
17: {**Tentative decoding:**}
18: $\widetilde{\gamma}_n(a) = \gamma_n(a) + \sum_{m\in M(n)} \beta_{m,n}(a)$
19: $c_n = arg \min_{a\in GF(q)}(\widetilde{\gamma}_n(a))$
20: **end while**

The VN processing is calculated columnwise and uses the CN messages ($\beta_{m,n}(a)$) calculated in the CN processing and the *a priori* information (γ_n(a)) to calculate a temporary VN message ($\alpha'_{m,n}(a)$) as shown in line 14. From ($\alpha'_{m,n}(a)$), the minimum value for each element is calculated in line 15 and it is subtracted from every symbol in GF*(q)* for each element, producing the VN message ($\alpha_{m,n}(a)$) (line 16) to be used in the next iteration of the CN processing.

The tentative decoding calculates the sums of every value by the column of CN messages ($\beta_{m,n}(a)$) and adds the result to the corresponding *a priori* information (γ_n(a)), producing *a posteriori* information ($\widetilde{\gamma}_n$(a)) as shown in line 18. The decoded codeword is composed of the argument of the lowest value in each element, that is, the lowest probability symbol as stated in line 19. This process repeats until either the check equation ($c_n^T H = 0$) is satisfied or the maximum number of iterations is reached.

Min–max design and implementation

The NB-LDPC code used in the MMA is defined by the sparse **H** matrix. To reduce memory usage, efficient data structures must be employed to represent sparse matrices. We propose the use of compressed sparse row (CSR) and compressed sparse column (CSC) formats. These compressed formats allow for the elimination of zeros on the **H** matrix, reducing its memory usage by up to 95%.

Graphics processing unit

The GPU implementation uses three kernels with different thread configurations as shown in Figure 2a. The first kernel calculates F and B (lines 6–9) and isolates the data dependencies between the metrics and β, the next kernel determines the CN messages ($\beta_{m,n}(a)$) (lines 10–12) and the third kernel computes the VN processing (lines 14–16). The F and B metrics kernel is executed in a configuration of X by 1 by 2^m threads per block, with X being an integer submultiple of M. The second dimension of the kernel configuration runs one thread per block because of the presence of data dependencies on the F and B metrics. The β kernel does not have data dependencies and can be executed with a configuration of X by d_c by 2^m. On the VN processor, the processing is columnwise, instead of row-wise as in the forward and backward metrics kernel. The VN kernel executes with d_v by Y by 2^m threads per block, with Y being an integer submultiple of N. These kernel configurations ensure better kernel occupancy rates and isolate data dependencies. Shared memory has a faster memory access time and can be used to store intermediate values, namely, in the "min" and "max" calculations in the forward and backward metrics kernel (lines 7 and 9 of Algorithm 1) and in the β kernel (line 12 of Algorithm 1). On the VN processor, a reduction tree is employed with shared memory to execute the operation in line 15 of Algorithm 1.

Field-programmable gate array

For the FPGA implementation, the HLS- and the RTL-based approaches were used. For the HLS-based design, separate function blocks were used for the CN and VN processing, as depicted in Figure 2b. The F and B matrices were completely partitioned, while the α and β matrices were block partitioned with a factor of m. Design-space exploration was executed on the algorithm as suggested in [6] and pipelining was applied on the CN and the VN as stated in [2]. The following design primitives were used to optimize the design.

Array partitioning: Employing array partitioning techniques allows the use of several parallel accesses to memory in place of a large memory unit with a single R/W port. In this design, complete array partitioning was used for the input matrices, the backward and forward metrics. For higher GF using complete array partitioning results in a long compilation time and also negatively affects the operating frequency. Arrays α and β are block partitioned where each element is stored in individual registers.

Loop unrolling: This technique is beneficial in reducing latency within loops that do not exhibit any dependency between iterations. However, if the loop body involves array accesses, loop unrolling needs to accompany array partitioning to facilitate parallel array accesses for the unrolled version. As a result, for larger loop iteration, this directly results in increased area consumption for accommodating the logic in the unrolled loop and the multidimensional array access. We have selectively used loop unrolling in higher GFs to avoid loss in frequency. Lines 6 and 8 in Algorithm 1, which computes the forward and backward metrics initial values, are unrolled for each symbol in the corresponding field, as well as the first and last values of β (lines 10 and 11). In the VN processing, the sum operation of line 14 is unrolled for each symbol and each connected CN (excluding the one CN chosen). Lines 15 and 16 are also unrolled for each symbol.

Loop pipelining: This technique reduces the latency of execution by partially overlapping multiple loop iterations. This technique is especially useful in cases where loop unrolling cannot be applied on account of dependency between statements in a loop. By ensuring a valid initiation interval (II), pipelining can be applied to such loops to reduce the latency and the II remains relatively lower than the depth of the loop. In the CN processing, pipelines are used for each CN, and inside that pipeline, lines 7, 9, and 12 also use pipelines for each symbol, as depicted in Figure 2b. In the VN processing, each VN has a pipeline but with a II of three clock cycles due to concurrent data accesses in line 14.

Customizing bit width: Vivado HLS tool provides arbitrary precision (ap_int.h) data types, which allows to allocate custom bits to variables and arrays. For the MMA, the variable bit width is decided by

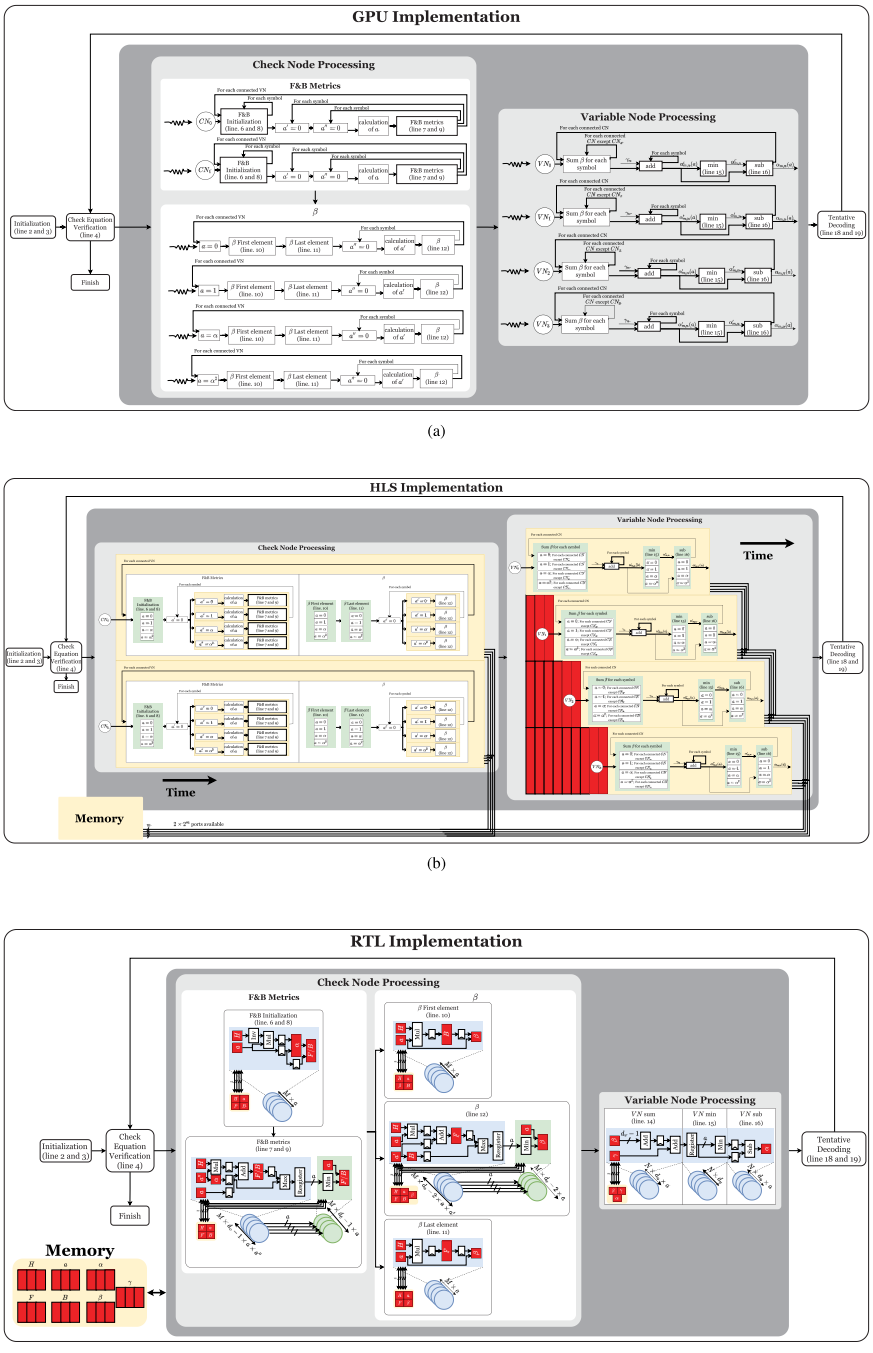

Figure 2. Illustration of the proposed GPU, HLS, and RTL parallel decoder architectures using distinct approaches. (a) GPU implementation is divided into three kernels where the *F* and *B* metrics kernel uses one thread per CN, the β kernel uses one thread per symbol, and the VN kernel uses one thread per VN. (b) HLS implementation using custom bit widths, array partitioning, pipelining (represented in yellow), and unrolling (represented in green). This implementation uses one pipeline per CN/VN with some parts unrolled. In VN processing, the red blocks represent the II of three cycles per pipeline. (c) In the RTL implementation, block diagrams in blue and green are replicated to fit the LDPC decoder's characteristics. The red blocks represent the BRAM used to write and read data.

the size of the GF. This helps save memory resources. The fixed bit representation leads to uncomplicated synthesized circuits with reduced latency compared to the floating-point or integer data types.

For the RTL-based approach, α and β are 3-D matrices stored in BRAMs, illustrated in red in Figure 2c. Each BRAM has a locations where a is the number of GF elements. The parallelization of the CN and VN blocks uses $M \times d_c$ BRAMs for both α and β so that each location of α (columnwise) and β (row-wise) could be simultaneously accessed. In the CN block, the operations are row-independent with M rows. The ($F(a)$) and ($B(a)$) metric registers are filled simultaneously from the GF arithmetic between α and **H**. The β BRAM is written with the data obtained from the $F(a)$ and $B(a)$ metric registers. The CN processing involves logic to calculate the corresponding addresses for every location to be read from α and written into β. In the VN processing block, the operations are column-independent with N columns. The β is read in this block along with γ in parallel from every column to compute the data to be written into α. The VN processing also involves logic that calculates the corresponding addresses for every location to be read from β and written into α. The control block orchestrates the whole process in the form of a finite state machine. Once the maximum iteration is reached, the output data (β) is given into the postprocessing block through a register, one value per clock cycle.

Experiments and results

The GPU version of the MMA was written in compute unified device architecture (CUDA) and was implemented on the Nvidia Jetson TX2. This board is equipped with a low-power Nvidia GP10B GPU, providing 256 CUDA cores (1.3 GHz). Running at 2 GHz, the central processing unit (CPU) module is composed of a dual-core Nvidia Denver 2 CPU and a quad-core ARM Cortex-A57 CPU. Vivado 2019.1 was used for performing the synthesis runs on the FPGA. XCVU13p-FSGA2577-1-i from the Virtex Ultrascale+ family was used to conduct the experiments in HLS and RTL. The CPU version provided by [7] has been used to validate the results for the GPU and FPGA implementations. Each experiment was executed 20 times and the mean values for the 20 executions are presented in this article. Experiments were released for GF(2^2), GF(2^3), and GF(2^4) executing ten decoding iterations. The H matrix was provided by [8] with $M = 256$, $N = 384$, $d_c = 3$, and $d_v = 2$. The Xilinx tools were run on the Nimbix online platform (it provides 20 CPU cores and 192 GB of RAM), which helped to obtain the results within a reasonable time. For the HLS implementation, the C/RTL co-simulation was performed for the functional verification.

Graphics processing unit

Four techniques were proposed for the GPU implementation using shared memory, vectorized memory accesses, and streaming, albeit the latter two also use shared memory. The results for each of these techniques are listed below.

1. *Shared memory:* The throughput achieved for GF(2^2), GF(2^3), and GF(2^4) using shared memory is shown in Figure 3a. One important conclusion taken from this experiment is that shared memory improves throughput.
2. *Vectorized memory accesses—packing by row:* Vectorized memory accesses reduce memory transactions by packing data together and exploiting the data bus more efficiently. This implementation packs groups of rows within the same column and for the same symbol for the kernels in the CN processor. The obtained values are listed in Figure 3b. Using vectorized memory accesses reduces memory transactions but increases register pressure.
3. *Vectorized memory accesses—packing by field:* Using the kernel developed with shared memory, another way of implementing vectorized memory accesses is to pack samples by field. It was evident that thread divergence decreases throughput performance, as shown in Figure 3a. This approach does not improve the execution time because it increases the number of instructions per thread by at least two times due to the "min" and "max" operations as indicated in Algorithm 1.
4. *Streams:* Streams allow overlapping memory transactions with GPU execution. Because of data dependencies between CN and VN, the streams must be synchronized, meaning that the CN and VN execution finishes when the last streams finish, thus increasing overall execution time, as depicted in Figure 3c.

Field-programmable gate array

For the HL implementation, the throughput results for GF(2^2), GF(2^3), and GF(2^4) without

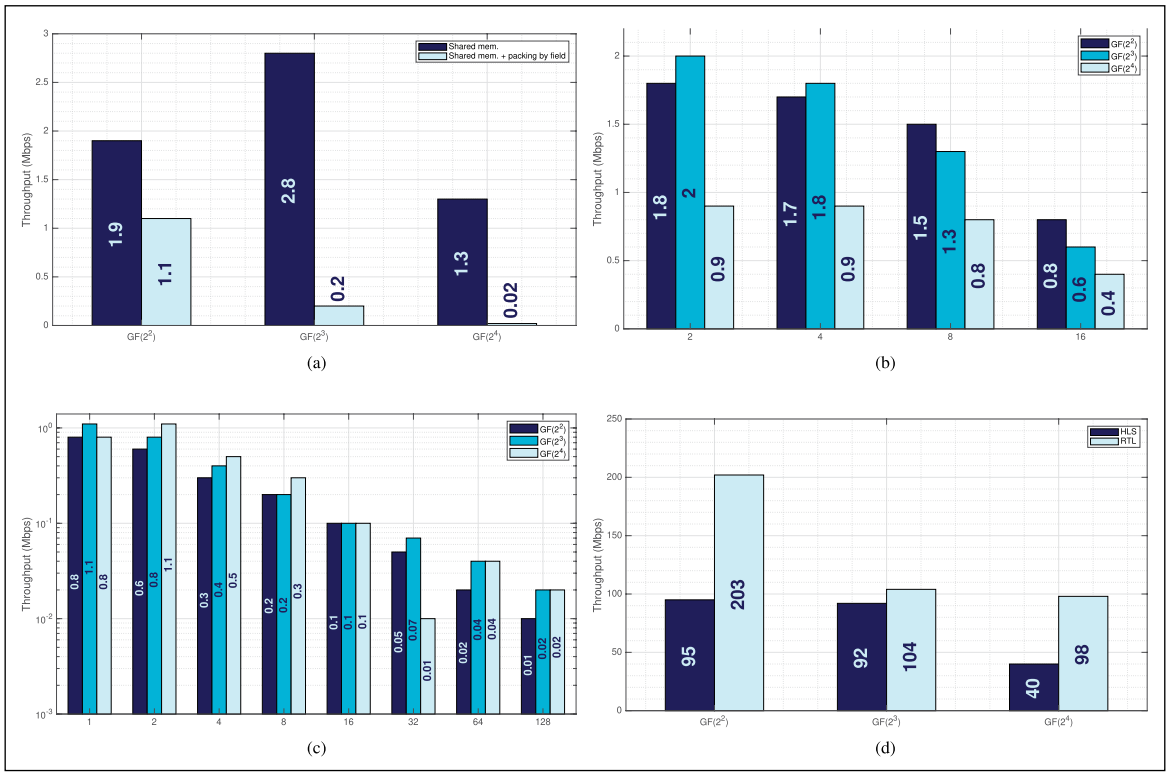

Figure 3. Performance results for the GPU, HLS, and RTL implementations. (a) GPU implementation using shared memory and vectorized accesses by field. (b) GPU implementation using shared memory and vectorized accesses packed by row, where the horizontal scale depicts the number of packed rows. (c) GPU implementation using shared memory and streams, where the horizontal scale depicts the number of used streams. (d) HLS and RTL implementations results.

pipelining the CNs and VNs were 18, 26, and 40 Mb/s, respectively. Using combinatorial circuits for representing GF multiplication was better than using lookup tables (LUTs), since the predetermined values need not be stored in memory. The kernel was further optimized by pipelining the CNs and VNs and unrolling the inner loops. This was possible only for $GF(2^2)$ and $GF(2^3)$. For $GF(2^4)$, this was not possible since the area resources in the FPGA were insufficient. The final throughput for $GF(2^2)$, $GF(2^3)$, and $GF(2^4)$ were 95, 92, and 40 Mb/s as shown in Figure 3d.

For the RTL implementation, the designs for $GF(2^2)$ and $GF(2^3)$ were implemented as a single finite state machine within a single module, whereas for the $GF(2^4)$ implementation, the code was broken into CN and VN modules to reduce the synthesis time. The throughput for $GF(2^2)$, $GF(2^3)$, and $GF(2^4)$ were 203, 104, and 98 Mb/s, respectively. The variations in throughput for the RTL implementation are shown in Figure 3d. The critical path limited the frequency of operation to 156 MHz. Each synthesis run took about 48–120 hours. For $GF(2^4)$, separate modules were synthesized to avoid the overutilization of resources. The maximum frequency of operation is limited by the critical path that was present in the VN module.

Roofline and energy efficiency analysis

The variation in energy and efficiency for CPU, GPU, and FPGA implementations are listed in Table 1. Due to the sampling rate of the power measurement device being slower than the execution of the code, a mean power of 5W was assumed for the CPU and GPU versions. For FPGAs, the power was measured after the synthesis and implementation of the Vivado HLx tool. The RTL implementation has the best energy efficiency compared to the HLS implementation for $GF(2^2)$ and $GF(2^3)$. For $GF(2^4)$, the energy efficiency diminishes for the RTL implementation

due to higher resource utilization compared to the HLS implementation. Figure 4 shows a roofline model that provides a comprehensive overview of the performance bottlenecks in the system. The design space exploration for the MMA algorithm was performed across all three platforms (CPU, GPU, and FPGA) for GF(2^2), GF(2^3), and GF(2^4). All the implementations were within the compute-bound region. Design techniques such as array partitioning, loop unrolling, and pipelining increase the performance for the same number of operations. The memory requirement for HLS and RTL implementations was decreased by using reduced precision of 6 bits instead of double or single-precision floating-point numbers.

Discussions and tradeoffs

NB-LDPC codes are computationally intensive and present many data dependencies. For higher fields, to achieve maximum throughput, higher-end GPUs can be used but, it does come with an additional cost of higher power requirements and energy consumption. The code design effort is much lower for HLS than for custom RTL codes and requires less time to synthesize. Loop optimizations and code reordering can be implemented to ensure that the data dependencies are met, and the designs can be pipelined and unrolled. However, the HLS tool is a black box. There are few details about the scheduling algorithm and how optimizations are applied in certain portions of the program. The code generation has limitations

Table 1. Energy consumption and efficiency for MMA on CPU, GPU, and FPGA for HLS and RTL.

	Energy consumption (uJ)				Efficiency (Mbps/W)			
	CPU[1]	GPU[1]	HLS	RTL	CPU[1]	GPU[1]	HLS	RTL
GF(2^2)	5180	2068	73	**34**	0.15	0.37	10.22	**22.16**
GF(2^3)	16390	2465	136	**126**	0.07	0.56	8.42	**9.14**
GF(2^4)	588218	5950	279	**254**	0.03	0.26	**6.15**	6.05

[1]Assuming 5 W running on the Jetson TX2.

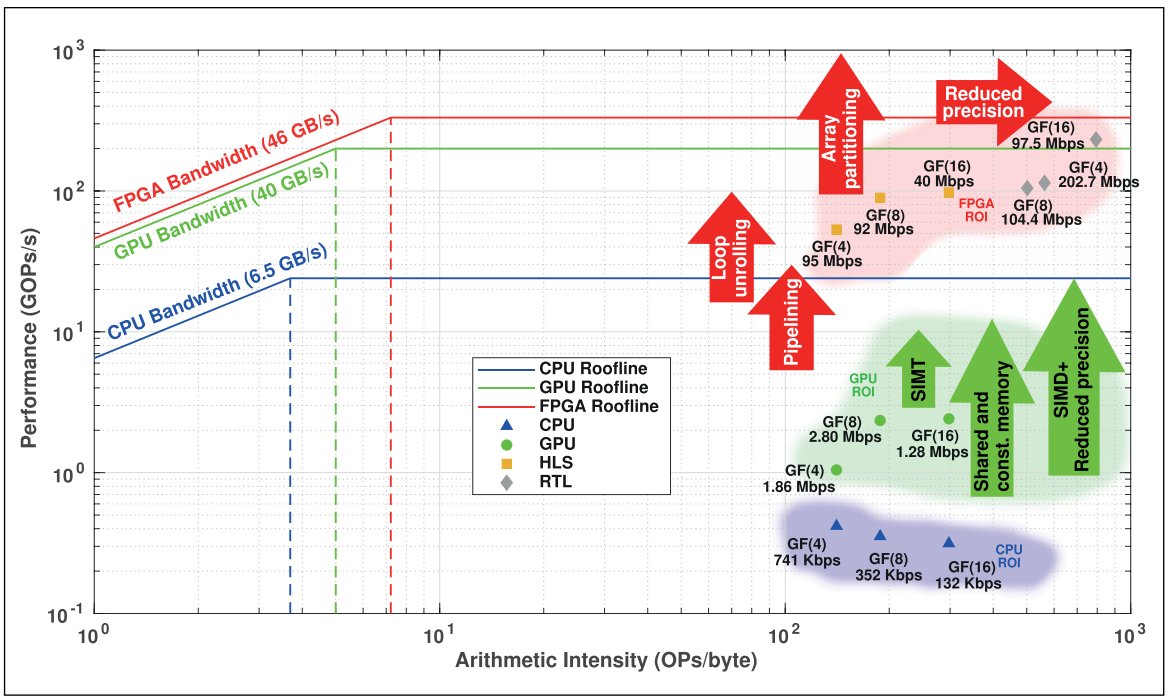

Figure 4. Roofline models for the CPU, GPU, and FPGA proposed architectures.

and may not produce the most efficient solution since the tool may ignore some basic hardware techniques that can be used, like substructure sharing and numerical strength reductions. RTL design takes a significant amount of effort and is time-consuming. A quick solution to identify the critical path and design constraints can be performed using HLS tools. For GF(2^2) and GF(2^3), we propose to use RTL implementations that require smaller energy consumption. For higher fields, it would be advisable to use a multi-FPGA setup properly orchestrated since resources are fully consumed in single-FPGA mode. GPUs, and, in particular, high-end GPUs, seem more appropriate for development and prototyping. From the application's perspective, for instance, in Internet of Things (IoT) applications that impose low energy constraints, FPGAs can achieve great throughput while demanding low energy consumption. However, if the requirements for throughput and power are relaxed, low-power GPUs can be a viable option, reducing cost and development effort/time.

Literature survey

An overall study was performed across the literature for the design of NB-LDPC decoders [9]. With respect to throughput, the custom hardware developed by the authors in [10] for the MMA uses a simplified Trellis MMA that requires less memory elements than the conventional implementation of this algorithm, at the expense of BER degradation, outperforming all the other implementations in terms of performance with 630 Mb/s. The weighted bit reliability-based (WBRB) NB-LDPC decoding algorithm and the full bit reliability-based (FBRB) developed by Huang and Yuan [11] achieved a throughput over 95 Mb/s, requiring less memory than the MMA. With the design of a semiparallel custom architecture that overlaps CN and VN computations, the EMSA achieved a throughput of 50 Mb/s [12], however, this approach results in a more complex decoder. For the MMA, Zhang and Cai [13] proposed a partial-parallel decoder that employed layered decoding and an overlapped scheme that simplifies the architecture which resulted in a throughput of 9.3 Mb/s. Pham et al. [14] proposed a parallel block-layered approach to the MMA decoder further improving performance and reaching 9.795 Mb/s using the Nvidia GTX Titan Black. The HLS-based fast Fourier transform (FFT)-SPA implementation [6] achieved a throughput of 14 Mb/s by instantiating multiple decoder blocks in parallel.

In this article, we performed a detailed study on how to implement MMA across different accelerators. For the GPU-based approach, using shared memory performs better than performing vectorized memory accesses. Our GPU implementation was able to achieve an efficiency of 0.56 Mb/s compared to 0.04 Mb/s of [14], while using less CUDA cores and power. For the FPGA implementation, we used both the HLS-based and the RTL-based approaches. HLS helps in providing quick design solutions. The RTL-based approach is more suitable for manually introducing parallelism in the hardware. Optimizing the GF operators for higher frequencies increases overall throughput. The area resources were completely consumed for GF(2^4). The shortage of FPGA resources restricts the design for higher GFs. The decoder IP core could be replicated to accommodate the entire FPGA area, with a small area reserved for synchronization, orchestration/control, and communication, which would increase data parallelism and throughput. Future work addresses the construction of designing efficient architectures for higher GFs and thus more efficient designs using HLS that can be developed by coding theorists/mathematicians without advanced hardware knowledge. The source code for designing these systems on GPUs and FPGAs are available at https://github.com/oscarferraz/NBLDPCTutorial. ■

Acknowledgments

ECHO is a joint work supported under the Indo-Portugal Bilateral Scientific and Technological Cooperation funded by Instituto de Telecomunicações and Fundação para a Ciência e a Tecnologia in Portugal, PhD Scholarship 2020.07124.BD, under Grant UIDB/EEA/50008/2020 and Grant PTDC/ EEI-HAC/30485/2017; and in part by the Department of Science and Technology, Government of India, under Grant INT/PORTUGAL/P-12/2017. This work was also supported under the Just-in-Time Customization: Research Platform for Accelerating Algorithms-to-Systems funded within the Early Career Research Award by Science and Engineering Research Board, India, under Grant ECR/2016/001765. Joseph R. Cavallaro was supported in part by U.S. NSF under Grants CNS-2016727 and Grant CNS-1827940, for the "PAWR Platform POWDER-RENEW: A Platform for Open Wireless Data-driven Experimental Research with Massive MIMO Capabilities."

References

[1] R. G. Gallager, "Low-density parity-check codes," *IRE Trans. Inf. Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[2] S. Subramaniyan et al., "Pushing the limits of energy efficiency for non-binary LDPC decoders on GPUs and FPGAs," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2020, pp. 1–6.

[3] L. Barnault and D. Declercq, "Fast decoding algorithm for LDPC over GF(2^q)," in *Proc. IEEE Inf. Theory Workshop*, Mar./Apr. 2003, pp. 70–73.

[4] D. Sepranos and M. Wolf, "Challenges and opportunities in VLSI IoT devices and systems," *IEEE Des. Test*, vol. 36, no. 4, pp. 24–30, Aug. 2019.

[5] V. Savin, "Min-max decoding for non binary LDPC codes," in *Proc. IEEE Int. Symp. Inf. Theory*, Jul. 2008, Art. no. 960964.

[6] J. Andrade et al., "From low-architectural expertise up to high-throughput non-binary LDPC decoders: Optimization guidelines using high-level synthesis," in *Proc. 25th Int. Conf. Field Program. Log. Appl. (FPL)*, vol. 1, Sep. 2015, pp. 1–8.

[7] G. Wang et al., "Parallel nonbinary LDPC decoding on GPU," in *Proc. Conf. Rec. 46th Asilomar Conf. Signals, Syst. Comput. (ASILOMAR)*, Nov. 2012, pp. 1277–1281.

[8] J. Andrade et al., "Flexible non-binary LDPC decoding on FPGAs," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 1936–1940.

[9] O. Ferraz et al., "A survey on high-throughput non-binary LDPC decoders: ASIC, FPGA, and GPU architectures," *IEEE Commun. Surveys Tuts.*, vol. 24, no. 1, pp. 524–556, 1st Quart., 2022.

[10] J. O. Lacruz et al., "A 630 Mbps non-binary LDPC decoder for FPGA," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2015, pp. 1989–1992.

[11] Q. Huang and S. Yuan, "Bit reliability-based decoders for non-binary LDPC codes," *IEEE Trans. Commun.*, vol. 64, no. 1, pp. 38–48, Jan. 2016.

[12] Y. Sun et al., "FPGA implementation of nonbinary quasi-cyclic LDPC decoder based on EMS algorithm," in *Proc. Int. Conf. Commun., Circuits Syst.*, Jul. 2009, pp. 1061–1065.

[13] X. Zhang and F. Cai, "Efficient partial-parallel decoder architecture for quasi-cyclic nonbinary LDPC codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 2, pp. 402–414, Feb. 2011.

[14] H. T. Pham, S. Ajaz, and H. Lee, "Parallel block-layered nonbinary QC-LDPC decoding on GPU," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2015, pp. 1–6.

Srinivasan Subramaniyan is pursuing a PhD with the Department of Electrical and Computer Engineering, Ohio State University, Columbus, OH 43210, USA. His research interests include graphics processing unit (GPU) scheduling and power optimization techniques.

Oscar Ferraz is pursuing a PhD with the University of Coimbra, 3030-290, Coimbra, Portugal. His research interests include parallel computer architectures, energy-efficient processing, and high-performance computing. Ferraz has an MSc from the University of Coimbra. He is a Student Member of IEEE, the IEEE Signal Processing Society, and the IEEE Computer Society.

M. R. Ashuthosh is a research assistant. His research interests include digital system design, computer architecture, and heterogeneous computing.

Santosh Krishna is a communications system engineer. His research interests include signal processing, field-programmable gate array (FPGA) architecture, and digital communications.

Guohui Wang is with ByteDance, Mountain View, CA 94041, USA. His research interests include mobile computing and computer vision. Wang has a PhD from the Department of Electrical and Computer Engineering, Rice University, Houston, TX, USA. He is a Student Member of IEEE.

Joseph R. Cavallaro is a professor of electrical and computer engineering and an associate chair at Rice University, Houston, TX 77005, USA. His research interests include computer arithmetic, and digital signal processing (DSP), graphics processing unit (GPU), field-programmable gate array (FPGA), and very large scale integration (VLSI) architectures for applications in wireless communications. Cavallaro has a PhD in electrical engineering from Cornell University, Ithaca, NY, USA. He is an advisory board member of the IEEE SPS TC on Design and Implementation of Signal Processing Systems and the past chair of the IEEE CAS TC on Circuits and Systems for Communications. He is a Fellow of IEEE.

Vitor Silva is an assistant professor in the Department of Electrical and Computer Engineering at the University of Coimbra, 3030-290, Coimbra, Portugal, where he is also currently the director of the Instituto de Telecomunicações. His research interests include signal processing, image and video compression,

and coding theory. Silva has a PhD from the University of Coimbra.

Gabriel Falcao is a tenured assistant professor at the University of Coimbra, 3030-290, Coimbra, Portugal, where he is also a researcher with Instituto de Telecomunicações. His research interests include parallel computer architectures, energy-efficient processing, graphics processing unit (GPU)- and field-programmable gate array (FPGA)-based accelerators, and compute-intensive signal processing applications. He is a Senior Member of IEEE.

Madhura Purnaprajna is an associate professor in the Department of Computer Science at Amrita Vishwa Vidyapeetham, Bengaluru, 560035, India. Her research interests include reconfigurable computing and processor architectures. Purnaprajna has a PhD in electrical engineering from the Heinz Nixdorf Institute, University of Paderborn, Paderborn, Germany.

■ Direct questions and comments about this article to Srinivasan Subramaniyan, Department of Computer Science and Engineering, Amrita Vishwa Vidyapeetham, Bengaluru 560035, India; srinivasansubramaniam74@gmail.com.