Received June 12, 2016, accepted July 12, 2016, date of publication July 27, 2016, date of current version November 8, 2016.

Digital Object Identifier 10.1109/ACCESS.2016.2594265

A Survey on Programmable LDPC Decoders

JOAO ANDRADE[1], (Student Member, IEEE), GABRIEL FALCAO[1], (Senior Member, IEEE), VITOR SILVA[1], AND LEONEL SOUSA[2], (Senior Member, IEEE)

[1]Department of Electrical and Computer Engineering, Instituto de Telecomunicações, University of Coimbra, Pólo II, 3030-290 Coimbra, Portugal
[2]Instituto de Engenharia de Sistemas e Computadores–Investigação e Desenvolvimento, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisboa, Portugal

Corresponding author: G. Falcao (gff@co.it.pt)

This work was supported in part by the National Funds through Fundação para a Ciência e a Tecnologia (FCT), via the doctoral scholarship under Grant SFRH/BD/78238/2011, in part by the Instituto de Telecomunicações under Grant UID/EEA/50008/2013, and in part by INESC-ID under Grant UID/CEC/50021/2013.

ABSTRACT Low-density parity-check (LDPC) block codes are popular forward error correction schemes due to their capacity-approaching characteristics. However, the realization of LDPC decoders that meet both low latency and high throughput is not a trivial challenge. Usually, this has been solved with the ASIC and FPGA technology that enables meeting the decoder design constraints. But the rise of parallel architectures, such as graphics processing units, and the scaling of CPU streaming extensions has shown that multicore and many-core technology can provide a flexible alternative to the development of dedicated LDPC decoders for the compute-intensive prototyping phase of the design of new codes. Under this light, this paper surveys the most relevant publications made in the past decade to programmable LDPC decoders. It looks at the advantages and disadvantages of parallel architectures and data-parallel programming models, and assesses how the design space exploration is pursued regarding key characteristics of the underlying code and decoding algorithm features. This paper concludes with a set of open problems in the field of communication systems on parallel programmable and reconfigurable architectures.

INDEX TERMS LDPC codes, LDPC decoders, parallel computing, CPU, GPU, reconfigurable computing, high-level synthesis.

I. INTRODUCTION

Known for more than fifty years [1], only recently *low-density parity-check* (LDPC) codes have been exploited under real-life *error-correcting code* (ECC) scenarios. They were left unused for more than thirty years, mainly due to the lack of computational power that was required to practically demonstrate their capacity-approaching characteristics [2]. Since the 1990s, they have become one of the most widely adopted coding schemes, along with Turbo codes, for operation in *forward error correcting* (FEC) systems in multiple communication standards: IEEE 802.3an, 802.11n, 802.15, 802.16, ETSI 2nd Gen. DVB, 3GPP LTE (4G) and ITU-T G.9960 and G.709 to name a few.

The vast majority of LDPC decoders found in the literature target dedicated *very large scale integration* (VLSI) decoders [3], either using *application-specific integrated circuit* (ASIC) technology or designing for reconfigurable computing devices (*field-programmable gate arrays* (FPGAs)) [4], [5]. The development of dedicated decoders for targeting an ASIC at certain technology node or an (FPGA) device incurs in high *non-recurring engineering* (NRE) costs and is usually an error-prone and protracted endeavor performed at the *register-transfer level* (RTL). In its turn, this entails that simulation and prototyping of the implemented solution be executed and verified over an alternative computing platform. Often, the simulation platform is not an obvious choice, since it performs only a simulation role in the whole development project. However, with the computing paradigm shift towards multi-core technology, and generally speaking, with the necessity to exploit parallelism in the computation to make the most out of the computational resources at hand, most processors include wide registers for vectorized operations, through streaming instructions, multithreading capabilities, and in the *graphics processing unit* (GPU) case, a massively multithreading environment with the vast majority of logic devoted to arithmetic units that is capable of executing at the TFLOPs range [6].

In the meantime, the ability to use GPUs as general-purpose processors has lead to an active field of research [7]–[35], [35]–[75], designated as *general-purpose*

GPU (GPGPU) [76]–[78], which has seen several works concerning the use of GPU devices programmed to operate as LDPC decoders. While we can only assume that *central processing unit*s (CPUs) are the first choice for code study and *bit error rate* (BER) performance analysis through Monte Carlo simulation, the majority of LDPC decoders found in the literature are not based on CPU architectures. With the odd exception, CPUs are mostly the underlying platform conveying proof of LDPC theory and concepts, but the decoder implementation is not the study focus, notwithstanding the fact that the advent of cross-platform parallel programming models and the growth in the register width of *single instruction multiple data* (SIMD)-vector units, has given CPUs a high level of computational power [75].

Several references found in the literature deal with LDPC decoders based on streaming architectures, namely, GPUs and other accelerators such as ARM mobile *system on a chip* (SoCs) [79], Intel *single-chip cloud computer* (SCC), the Cell B.E. [35] or experimental stream processors—the latter examples are less prevalent than GPU-based LDPC decoders. One of the reasons for GPUs popularity is the compromise between effort put into the development of a parallel algorithm that conveniently exploits the GPU *single instruction, multiple thread* (SIMT)-architecture, and the corresponding attained performance. First, the development time between testing and prototyping, and the final optimized decoder ready for deployment is not as high and does not incur in too high NRE costs, as hardware dedicated solutions do. Secondly, this flexibility usually means diminished returns in the performance of the decoding solution, due to the fixed instruction set, memory hierarchy and underlying architecture that are not custom-tailored to the developed decoder. Furthermore, the introduction of data parallel programming models such as *Compute Unified Device Architecture* (CUDA) [6] and *open computing language* (OpenCL) [80], together with the unification of the graphics pipeline into a single programmable processor, meant that high-level productivity scientific languages such as C/C++, Fortran, Python and Ruby could be utilized, instead of protracted graphics languages. The drawback is that only with sufficient knowledge of the underlying GPU architecture will the developed LDPC decoders perform with high decoding throughputs.

But while GPUs, due to their raw computational power (peaking in the TFLOPs range) and performance-to-watt-ratios orders of magnitude above CPUs, began to dip into the *high-performance computing* (HPC) market [81], and to some extent on the datacenter market too, *field-programmable gate array*s (FPGAs) were evolving too. Starting from a primitive "glue logic" status [82], they have given rise to the very active field of reconfigurable computing [83]–[86]. They bring more throughput per silicon area [85] and less energy is consumed in the process than using conventional processors [84]. Furthermore, due to the chip area of FPGAs, they usually accompany Moore's law technology nodes, while improvements on dedicated solutions, more often than not, fail to upgrade to faster, more efficient and smaller nodes in the same time-frame. Thus, the utilization of FPGAs as custom accelerators, usually designated as reconfigurable computing, addresses some of the issues surrounding the development of ASIC technology, but also set opened a whole new level of challenges purported by the availability of gate-level optimizations. Of particular interest to the work developed in this field, is the use of *high-level synthesis* (HLS) models, which extend C/C++ and other programming languages [87]–[89], in a somewhat similar way that *compute unified device architecture* (CUDA) and OpenCL did for GPUs, thereby avoiding the specific knowledge for developing VHDL and Verilog RTL-descriptions to generate circuits.

II. THE PROBLEM

ECC in FEC systems were deprived of the capacity-approaching capabilities of LDPC codes for over thirty years before sufficient computational power was available to enable their utilization. However, typical design approaches involve using VLSI technology, such as ASIC and FPGA development, which, thus far, require RTL-based development to reach within high decoding throughputs and low latencies.

A. MOTIVATION

The continuing trend in semiconductor manufacturing, dictated by Moore's Law, has placed tremendous challenges to processor manufacturers as the end of Dennard's scaling [90] meant performance scaling could no longer be guaranteed with increasing of the clock frequency of operation. Hence, as technology progressed into the multicore and manycore realm, with massively multithreading processing enabling computational powers in the range of the GFLOPs and TFLOPs, LDPC decoders have been shown to achieve dozens to hundreds of Mbit/s. However, due to the fixed instruction set of the underlying CPU and GPU architectures, the considerations taken by the LDPC designers can be substantially different in nature than those considered when developing custom-made hardware for LDPC decoders. In particular, designers are faced with the hard challenge of mapping the LDPC decoding algorithms onto a limited set of arithmetic operations, offered by a fixed *instruction set architecture* (ISA), also constrained by the nature of the memory hierarchy of the processor and the computing system as a whole. Moreover, native support for certain type of arithmetic types may not be supported, as well as thoughtful considerations are due to how data is moved in the system as a whole, since distributed to shared memory addressing regions may exist, lying on- or off-chip. Only the correct manipulation and definition of suitable arithmetic-to-memory ratios and patterns allows for the maximization of the delivered bandwidth—in other words, the resources offered by the programmable processors should be explored wisely so as to fully exploit their computational capabilities.

B. NOTATION

The notation utilized throughout the paper is listed below, and is employed to systematize the description of the decoding solutions presented in the surveyed works in the following Sections of this paper. The basic understanding of the decoding problem can be perceived from the illustration of a binary LDPC code in matrix and graph representation in Fig. 1.

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

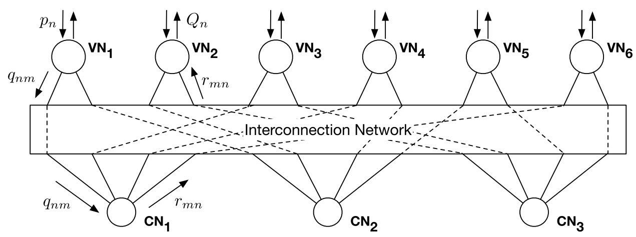


FIGURE 1. Parity-check matrix and Tanner graph example for the binary case. The parity-check matrix H defines the LDPC code and is the adjacency matrix to the associated bipartite graph designated as Tanner graph. The majority of the decoding algorithms are message-passing: an *a-priori* likelihood is given to each VN; VNs send q_{nm} messages to their adjacent CNs; CN update r_{mn} messages and send them back to their adjacent VNs; in their turn VNs produce new estimates on the q_{nm} messages and also a new *a-posteriori* estimage on the VN state Q_n.

- $\mathbf{H}$ is the parity-check matrix of the LDPC code;
- *check node* (CN) corresponds to a type of node in the Tanner graph and to a parity-check equation;
- *variable node* (VN) corresponds to the other type of node in the Tanner graph and to a symbol in the codeword;
- an edge exists in the Tanner graph connecting CN_m to VN_n whenever there is a non-null element h_{mn} in the parity-check matrix $\mathbf{H}$;
- p_n *a-priori* likelihood estimate for VN_n;
- q_{nm} message sent from VN_n to CN_m;
- r_{mn} message sent from CN_m to VN_n;
- Q_n *a-posteriori* likelihood estimate for VN_n.

C. EVALUATION

The characteristics and complexity of the LDPC decoding algorithms are surveyed for codes defined over binary and non-binary fields. The $O(\cdot)$ numerical complexity presented does not capture the totality of transposing the LDPC decoding algorithm into an efficient LDPC decoder operating on either programmable or reconfigurable architectures. We can enumerate a list of the most important challenges to overcome in the design of efficient and high-performance LDPC decoding solutions as follows.

i) the need to transform node connections imposed by the Tanner graph into a suitable memory layout and efficient addressing problem, considering that in most cases, irregular access patterns will be imposed by the Tanner graph structure;

ii) the weighing of suitable ratios of arithmetic-to-memory-instructions which maximize the efficiency of the LDPC decoding for the underlying computer architectures;

iii) the selected scheduling variants of the algorithm—*two-phased message passing* (TPMP) or *Turbo decoding message passing* (TDMP)—influence on the aforementioned items;

iv) parallelism has to be explored at different levels of complexity depending on the architecture being programmed;

v) the complex exploitation of the memory hierarchy of multi-core systems, or the complex problem of defining a suitable memory hierarchy for reconfigurable LDPC decoders.

D. FIGURES OF MERIT

The different solutions must be evaluated according to certain key figures of merit. Due to limited information that can be collected from the surveying of the literature, namely lack of profiling results, we limit our assessment mostly to decoding throughput and latency at a fixed number of decoding iterations—usually under the canonical level of 10, when applying the TPMP and 5 when applying TDMP decoding schedules. Another metric which aims at normalizing the results across LDPC decoders programmed in different GPU architectures is the *throughput of decoding normalized per core per frequency* (TDNC).

III. DECODING ON PROGRAMMABLE ARCHITECTURES

In this Section, the survey is focused on programmable architectures for LDPC decoder solutions. The most prevalent LDPC decoders found are GPU-based, then CPU-based, and finally, those based on streaming accelerators. We focus on key characteristics of the LDPC decoders, namely 1) task- and data-parallelism, 2) data representation, 3) LDPC code type and dimensions, 4) the indexing method of the messages circulating in the Tanner graph; and figures of merit of the LDPC decoder performance with respect to computational power, i.e., 5) decoding latency, 6) decoding throughput. Finally, the programmable platform is also identified. The following subsections are devoted to discussing the methodologies developed for defining efficient programmable LDPC decoders in light of their characteristics and design space constraints.

In addition, the reader is referred to Table 1 in Appendix A, which contains a tabulation of the surveyed LDPC decoders, and that can be found as a supplementary file to this manuscript.

A. PROGRAMMABLE LDPC DECODER MAPPING

Due to the programmable nature of the underlying computer architectures, a prototype isomorphic architecture [91] that is a direct mapping of the Tanner graph to CN units, VN units and the Tanner graph interconnection network is not truly possible [92]. Instead a programming description is required considering that the fixed underlying architecture and instruction set will demand the sharing of computational resources.

Only clever usage of the instruction set functionality and exploitation of the different regions within the memory hierarchy is guaranteed to optimize the LDPC decoders for performance and efficiency of computation [34]. While the term occupancy usually refers to GPU computing, the concept also extends to CPU architectures. Considering the limited but fixed number of logic arithmetic and memory resources available in programmable architectures, only if occupancy of the resources is high, will the performance of LDPC decoders peak. However, occupancy is a double-edged sword in the sense that it does not take into account over-utilization of resources leading to bottlenecks or deadlocks, that nevertheless keep occupancy high. Moreover, it is difficult to assess how efficiently a hardware resource is being utilized, based solely on the information provided by what authors have made available in their LDPC decoders on programmable architectures' publications. Thus, we are left with figures of merit contextualized for the LDPC decoding problem, throughput and latency.

B. TANNER GRAPH INDEXING SCHEMES

The LDPC decoder structure plays an important role on how efficiently the Tanner graph connections between nodes can be mapped. While regular codes might seem at first simpler than irregular ones, in practice they are not. Since the majority of LDPC codes also keeps simplicity of encoding and simplicity of Tanner graph indexing in mind, standardized codes make use of systematic coding schemes, exploring *repeat-accumulate* (RA) parity sub-matrices, mainly for encoding purposes, and structured irregularity in the remaining portion of the parity-check matrix concerning the connectivity of *information nodes* (INs) [93]. As discussed in the literature [1], each edge defines two messages, traversing in opposite directions, as seen in Fig. 1. When mapping the Tanner graph connections to a programmable processor, we must take into account that the messages have to be laid out in memory and, thus, their location index must be known with respect to both the CN and VN that define the edge. The memory index is usually not the same for messages traversing the same edge in different directions.

Since the LDPC code parity-check matrix is also the adjacency matrix of the Tanner graph, any given LDPC code can be stored through matrix storage methods. Due to its sparsity, sparse matrix storage methods have lower memory footprints and can be employed for any type of code, and in fact they are. Regular codes, typically those constructed through *progressive edge growth* (PG) methods and made available in the Encyclopedia of Sparse Graph Codes [94] (Mackay codes), are stored most of the times in *compressed row storage* (CRS)- and *compressed column storage* (CCS)-like formats. This method of indexing the Tanner graph is shown in Fig. 2. It is readily seen that out of the four memory accesses to q_{nm} and r_{mn} messages, two can effectively be contiguous, a feature most of the surveyed decoders implement—this exposes high bandwidth due to coalesced data accesses on GPUs and high cache hit rates on CPUs. The scenario

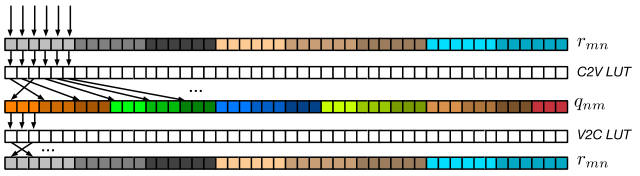


FIGURE 2. Tanner graph indexing based on sparse matrix storage.

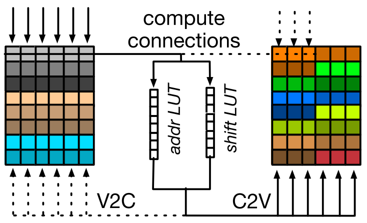


FIGURE 3. LDPC-IRA Tanner graph indexing based on sparse matrix storage.

depicted in Fig. 2 defines reading accesses to be contiguous and writing accesses as non-contiguous [32]. Thus CNs require indexes relative to their adjacent VNs, and read a memory location offset from a *lookup-table* (LUT) (CN$_{idx}$), and VNs, likewise, read a memory offset from the other LUT (VN$_{idx}$). Essentially, VN$_{idx}$ corresponds to the messages positions in memory for a reshaped column-wise parity-check matrix, and the CN$_{idx}$ to its row-wise reshaped. As a consequence, the number of elements required to store the connections of a binary LDPC code Tanner graph is

$$TG_{\text{sparse}} = 2\times \sum_{m=0}^{M}\sum_{n=0}^{N} h_{m,n}. \tag{1}$$

This indexing method can also be employed for standardized codes [51], though the memory footprint of the Tanner graph mapping can be reduced by orders of magnitude [40], [95]. In the particular case of *LDPC irregular Repeat-Accumulate* (LDPC-IRA) codes, such as those employed in the 2^{nd} *generation DVB* (DVB 2) standards, shown in Fig. 3, the LDPC code Tanner graph is systematically constructed in a way that permits indexing using a barrel shifter approach [95]. For instance, the number of elements required to index the DVB 2 Tanner graph codes is

$$TG_{\text{DVB 2}} = 2 \times d_c \times r_f, \quad r_f \ll N, \tag{2}$$

with r_f a code construction regularity parameter [95], [96]. This allows an *on-the-fly* computation of the memory locations to where each message reads and writes. In particular, q_{nm} messages will read and write to a contiguously increasing base offset, but writes will be shifted, and r_{mn} messages are read from an indexed base offset and written shifted while maintaining the offset. Hence, an address and a shift LUT, with a much lower size than the CCS and CRS sparse matrix storage (2) can be employed.

Likewise, *quasi-cyclic LDPC* (QC-LDPC) codes can also be indexed by small-sized *lookup tables* (LUTs), as shown

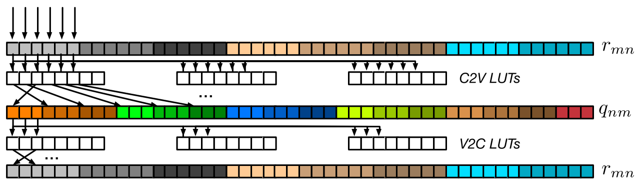


FIGURE 4. QC-LDPC Tanner graph indexing based on sparse matrix storage.

in Fig. 4, by performing an *on-the-fly* computation of the memory location to where a message is sent after computation. This method in particular [40], defines contiguous reading of messages and indexed writing to memory, with a footprint that is independent of the LDPC code dimensions. In a way, the code protograph is sparsely indexed using the first method, though extra computation of indexes is required. The memory footprint of this method is

$$TG_{\text{QC}} = 2\times3\times\left(\sum_{m=0}^{M}\sum_{n=0}^{N} f_{m,n} \neq \infty\right) + M_f + N_f, \quad 0 \leq m < M_f, \quad 0 \leq n < N_f, \qquad (3)$$

with M_f and N_f being the dimensions of the protograph matrix that generates the QC-LDPC code. The great advantage to this indexing scheme is that, regardless of the final codeword length which depends on the expansion factor z_f, the indexing LUTs size remain the same [40].

The memory footprint is not the most pressing issue in programmable architectures, as the memory addressing space size of modern CPU and GPU systems can be larger than the Tanner graph indexing memory footprint (1) (2) (3). However, the indexing method becomes a source for memory contention if for every computed message a memory index location requires loading, reducing the overall bandwidth to memory—it contributes to poorer cache hit ratios on CPUs [75] and adds further pressure to GPU memory engines [30]. The best performing LDPC decoders are those exploring structure sparse storage that exploit the Tanner graph structure, as opposed to a generic sparse matrix storage method. For instance, LDPC decoders implementing the former methodology achieve much higher throughputs than those exploring the latter [30].

Several parameters, other than the Tanner graph indexing, influence the decoding throughput attained, however, a clear trend is observed in this case. For all thread-parallelism techniques employed, only the *thread-per-codeword* (TpC) strategy overcomes the overhead in the Tanner graph indexing scheme [50], since the imposed overhead is negligible with regards to the amount of data moved in the GPU architecture. The remaining decoders see throughputs of a few to a dozen Mbit/s. Only structured indexing schemes consistently see decoding throughputs in the hundreds of Mbit/s.

C. PROGRAMMING MODELS

A prevalence of C/C++-based families can be observed throughout the surveyed LDPC decoders, adding to the popularity language under an HPC challenge such as the one concerning the development of LDPC decoders churning out very high decoding throughputs.

In particular, parallelism in CPU-based decoders has been exploited using the *open multi-processing* (OPenMP) programming model in a number of decoders [15], [16], [47], [54], [55]. Therein, the strategy to extract parallelism is based on the automatic parallelization of computation loops wherein the CN and the VN processing are defined. This is achieved with appropriate OPenMP directives. A minority of LDPC decoders replace the functionality provided by the acOPenMP loop parallelization directives with explicit thread-partitioning using POSIX threads [37]. Despite the usage of a lower-level *application programming interface* (API) to perform multi-threading, the opportunity to improve on the decoding throughput is not fully captured by this approach. However, POSIX threads are the basis of the Cell B.E. *software development kit* (SDK), which is a C-extended programming model [8], [35]. Other authors choose to put upon the OpenCL cross-platform capabilities to use on CPU technology [33], [34], [36] making minor adjustments from the GPU-optimized decoder onto a computing substrate with lower parallel capabilities. Similar to the aforementioned OPenMP and POSIX threads strategies, the delivered throughput is limited by a number of factors, the most important of which is the OpenCL compiler ability to pack data elements within wide registers over which *single instruction, multiple data* (SIMD) computation is performed [34].

Under this light, explicit utilization of SIMD instructions is pursued on both x86 and ARM processors. The first have since evolved from their assembly-accessible 128-bit MMX registers [25], [32], [61] to the richer extensions provided by *streaming SIMD extensions* (SSE) and *advanced vector extensions* (AVX) at 128-, 256- and 512-bit register widths, though LDPC decoders in the literature exploit only 256-bit AVX registers [75]. As the instruction-set functionality of the SIMD extensions became richer, so did the abstraction concerning its use. While MMX required explicit assembly coding, *streaming simd extension* (SSE) and AVX support high-level C/C++ intrinsics. On the ARM processors, the NEON extensions provide 64- and 128-bit registers, exploited through a set of C/C++ intrinsics [37]. SIMD computation also faces another challenge. The indexing of the Tanner graph connections renders data element packing and unpacking unavoidable [75]. This means that under MMX register, a non-negligible overhead of data management housekeeping tasks offsets the performance gains enjoyed from SIMD computation. On the other hand, only the increased functionality and width of the SIMD units can guarantee higher performances, on a par with GPUs, due to the increase levels of data-parallelism purported by the data packing for SIMD execution [75].

On the GPU LDPC decoders side, apart from the seminal approaches utilizing streaming models [97] based on graphics programming languages [23]–[25] and early streaming computing models [97], the majority of LDPC decoders makes use of CUDA and a minority of OpenCL. While in certain aspects both programming models are alike, with many language traits and constructs referring to the same hardware features of the GPU processor, only with different designations and handles [80], [98], CUDA popularity overwhelms that of OpenCL. On Nvidia platforms the reason is clear, as OpenCL is mapped on top of CUDA, with the performance of the former only reaching that of the latter at best [99]. Also, despite the similar ways to explore arithmetic instructions, data types and memory addressing spaces of both models, OpenCL is a cross-platform programming model, rendering superfluous management instructions and verbose coding requirements not really necessary when cross-platform is not truly intended. Nevertheless, the surveyed decoders see no clear performance gap between the use of CUDA or OpenCL LDPC. CUDA-based LDPC decoders range from the inception of CUDA in 2007 to the latest stable version. However, majority of the LDPC decoders based on it explore only a limited subset of its features. In particular, more advanced features such as kernel self-calling and reconfiguring the GPU execution grid without the host CPU intervention (*dynamic parallelism*) [98] and advanced memory synchronization and fencing operations available to the whole thread execution grid are not explored in the surveyed works, though they address limitations identified in some of the works [17]–[19], [59], [74]. With this regard, the OpenCL specification suffers from a lower evolution pace, with features sensitive to GPU programming having to be ratified for inclusion in a cross-platform model also supporting CPUs and FPGAs. Thus, OpenCL-based decoders are somewhat more limited to some features that CUDA addresses [33], [37], [49], [64]. Notwithstanding, as such features are not explored, there is no evident loss in choosing one instead of the other.

With more or less control of the underlying hardware, all of utilized programming models allow the development of parallel LDPC decoding algorithms. Parallelism is naturally exposed in these [100], [101], but other parallel features pertain only to a certain algorithm expression, in its turn tightly coupled to an underlying architecture. This concerns parallelism at the task-level and at the data-level that are discussed next.

D. THREAD-LEVEL PARALLELISM

Several parallelism strategies have been proposed on multi-core CPU and many-core GPU architectures that divide LDPC decoding tasks between concurrent execution threads. These strategies entail constraints to other important design features of the LDPC decoders, especially with regards to data-parallelism and also to the decoding schedule of nodes, which are on the following Subsection.

1) TAXONOMY

Due to the rich set of parameters explored by researchers in the development of programmable LDPC decoders, an appropriate taxonomy for LDPC decoding on programmable architectures will be introduced herein. First regarding parallelism, whereupon the nodes functionality is translated onto task- and data-parallelism at certain granularity levels among physical or logical cores or between execution threads. To keep a low simplicity of taxonomy, we define it in terms of thread-parallelism, which is also in accord with the majority of the surveyed programmable LDPC decoders that take in thread-based programs, a convenient feature as it will not require a set of terms for multi-core architectures and another for many-core ones.

2) PpE DECODING

pixel-per-edge (PpE) is the oldest parallelism strategy dating from the time where GPGPU was at its inception, with graphics languages being the only way available to perform non-graphical computation on GPU processors. Back then, data elements had to be mapped onto ''graphical data elements'' in order to be processed by the pixel shaders, thereof stemming the designation of this thread-parallelism strategy.

While results seemed highly promising at a time when the only prospective way to achieve high-throughput would be to develop a dedicated hardware accelerator, they were still lacking the performance seen for the LDPC decoders under CUDA and OpenCL, once the graphics pipeline had been unified onto a single processor [6]. Approaches such as those proposed by Falcao *et al.* allowed decoding throughputs of dozens to hundreds of Mbit/s [23]–[25], combining a graphics language approach with Caravela streams [97], as illustrated in Fig. 5.

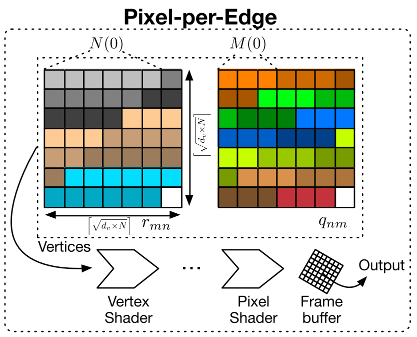


FIGURE 5. Pixel-per-Edge LDPC decoder thread-parallelism.

3) TpE DECODING

thread-per-edge (TpE) corresponds to the strategy with the finest granularity, which brings upon the LDPC decoder designer a granularity tradeoff. For the one, if consecutive threads deal with the update of messages belonging to consecutive nodes in the Tanner graph, then there is a high exposure of both spatial and temporal data locality, as Fig. 6 illustrates. For the other, most GPU-based LDPC decoders that exploit this granularity have been developed for pre-Fermi or Fermi architectures that do not have a caching mechanism available to threads for computation [6]—it exists only for off-chip

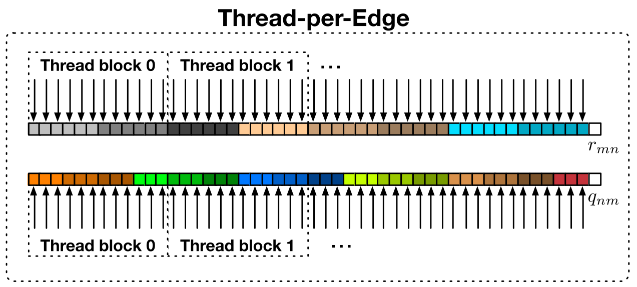


FIGURE 6. Thread-per-Edge LDPC decoder thread-parallelism approach.

memory transaction. For instance, locality is automatically explored by the x86 cache system in heavily SIMD-based LDPC decoders [75], leading to over 90% cache hit rates that maximize the LDPC decoder bandwidth. Alas, this is not the case in the many-core-based decoders utilizing this strategy. Under the methodology proposed by Chang *et al.* throughputs peak lower than ∼2 Mbit/s for moderate length codes (816, 4000 and 8000 bits) [17]–[19]. This approach, requires the spawning of the highest number of threads, compared to the remaining approaches. For a regular LDPC code, the VN processing sees $N \times d_v$ threads spawned, and likewise the CN processing spawns $M \times d_c$ threads, implying the thread-parallelism granularity level putting the most pressure through the generation of thousands of threads, even though computation within each thread is not as heavy as coarser thread-parallelism strategies, which are discussed next.

4) TpN DECODING

thread-per-node (TPN) is the most prevalent strategy, with a thread being spawned per node in the LDPC code Tanner graph. While this strategy quickly depletes the number of concurrent threads that can be active simultaneously on many-core GPUs for moderate to long block lengths, this pressure is not as high as in the TpE strategy case for short to moderate codes. One of the reason behind this strategy being by far the most popular strategy is related to its elegance. Each node in the Tanner graph can be assigned to an execution thread in the absence of a *de facto* isomorphic transformation [3] into a *functional unit* (FU).

The first LDPC decoder observed to utilize this thread-granularity (see Fig. 7) was proposed by Falcao *et al.* for short to moderate length Mackay codes [94] (rate 1/2 1024, 4000 and 4896 bits) reaching up to ∼1.63 Mbit/s. The decoder forcibly defined a 2-D texture mapping of the *log-likelihood ratios* (LLRs) messages that contributes to the poor performance yielded [31]. Under a more general-purpose computing memory mapping, the authors were able to elevate the decoding throughputs to ∼87 Mbit/s for the normal frame DVB-S2 codes [30], and to ∼40 Mbit/s for rate 1/2 Mackay codes (1024 to 20000 bits) [32]. The difference in the attained performance shows how data-parallelism design decisions and Tanner graph indexing methods are pivotal to elevating the decoding throughputs attained by GPU-based LDPC decoders. This is also verified by the work by Grönroos *et al.*, elevating their initial decoding throughput assessment (<1.80 Mbit/s) [38] to higher levels for higher data-parallelism levels (157∼192 Mbit/s). The limits to the scalability of this approach were tested by Zaldivar *et al.* for a TpN-variant executed by a 3-dimensional thread grid [51], for various d_c and d_v configurations. Chiu *et al.* report high latency times 0.2∼1.8 s for a short length (672 bits) 802.15.3c code [62]. Wang *et al.* propose a TpN decoder for 802.11n and 802.16e codes reaching 40∼52 Mbit/s [65]. However, other TpN approaches achieve only limited decoding throughputs [41], [54], [55], without a clear pattern to what lead to such low levels of throughput performance—especially in light of the employed highly efficient Tanner graph indexing methods for cyclic and quasi-cyclic codes [39], [40].

While LDPC codes working as the ECC basis of FEC in communication systems imply an application agnostic operation, i.e., all bits being equally protected, they also perform well under more applied applications such as video coding [59]. On their application to video coding benchmarks (Hall Monitor, Foreman, Coastguard and Soccer), excellent frame reconstruction is obtained, though for offline coding, i.e. it is not applied to real-time decoding [52], [53], [59]. Other applications of LDPC decoding worth mentioning is their use for *quantum-key distribution* (QKD) reconciliation [102]. Mink *et al.* were able to reach high decoding throughputs, though for this purpose a lower number of iterations is required [74].

Finally, non-binary LDPC decoders that implement this strategy have also been proposed [13], [64]. Beerman *et al.* define a 3-dimensional grid of computation to properly exploit parallelism exposed at the *binary extension field* (GF(2^m)) dimension and by the scheduling of operations within the processing of the Tanner graph nodes. Under the proposed strategy, equivalent decoding throughput (∼2 Mbit/s) is obtained for GF(2^m) spanning the binary field (GF(2)) to GF(2^8).

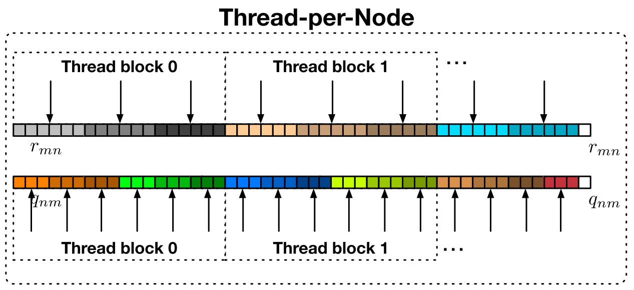


FIGURE 7. Thread-per-Node LDPC decoder thread-parallelism approach.

5) BpC DECODING

block-per-codeword (BpC) is a strategy available to GPU execution as it is based on the concept of a thread block [6], a CUDA concept that finds its equivalent as workgroup

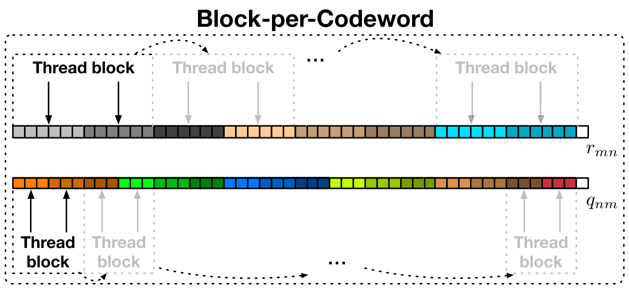


FIGURE 8. Block-per-Codeword LDPC decoder thread-parallelism.

in OpenCL [80]. The rationale stems from the execution grid composed of threads and divided onto blocks that permit a suitable exploitation of the memory hierarchy in many-core GPU architectures, as depicted in Fig. 8. Threads within the same block are allowed synchronization and fencing mechanisms for tighter cooperative computation as they can access the shared memory space, an addressing block physically unavailable to inter-block communication. A strategy based on this granularity fails short of utilizing all the GPU *stream multiprocessor*s (SMs), as the number of blocks required is independent of the code length. Hence, this strategy is usually accompanied by data-parallelism levels that spawn more blocks to decode more codewords in parallel[1] throughout the remaining SMs of the architecture [7], [42], [69]. Notwithstanding, there is a constraint on how many threads can compose a block, that is influenced not only by the capabilities of the underlying hardware—high-end GPUs can execute blocks with a higher thread count than low-end ones—but also by how the developed decoding algorithm consumes registers and shared memory [6]. For QC-LDPC codes, it might make sense to define z_f threads per block as it matches the protograph expansion factor. However, this might be too low of a value, leading to poor resource utilization, or too high, preventing this strategy to be accommodated onto lower-end GPUs [69]. Equivalent tradeoffs can be seen for *LDPC irregular repeat-accumulate* (LDPC-IRA), random LDPC codes and non-binary regular ones [103]. This thread-granularity level is also the *de facto* strategy able to efficiently implement *turbo decoding message passing* (TDMP) and TPMP decoding schedules, as opposed to the remaining strategies which are usually limited to the TPMP, as explained next.

First proposed by Abburi [7], this strategy has been applied to *worldwide interoperability for microwave access* (WiMAX) (IEEE 802.16e) codes, and also to *Wi-Fi* (Wi-Fi) codes (IEEE 802.11n) [69], for moderate to high decoding throughputs achieved (24.50∼160 Mbit/s), at relatively low latencies under 12 ms. This method has also been explored for the quick evaluation of a QC-LDPC construction method [42].

[1]Note that we adopt the designation codeword in a broader sense. Codeword can be the set of codewords that can fit onto the same data type, thus, it can be a single element or several ones packed into a vector data type [34].

6) BpN DECODING

block-per-node (BpN) is, in a sense, a particular case of the BpC approach. Non-binary LDPC codes define another dimension exposing parallelism, the GF(2^m) dimension. This strategy is adopted for the particular case of non-binary LDPC decoding, but instead of defining a block of threads depending on the LDPC Tanner graph regular features, it depends on the *binary extension field* (GF(2^m)) dimension [57]. This approach is tested for a GF(2^8) code, yielding throughputs in the Mbit/s range (<6) under different levels of data-parallelism for a pure sequential decoding schedule. Wang *et al.* also use this approach for a non-binary LDPC decoder for both OpenCL-based execution on CPU and GPU architectures. Though their approach has considerably low latencies (<5 ms) it achieves only 1.26 Mbit/s at best for GF(2^m) dimensions of 2^2, 2^3 and 2^4.

7) TpC DECODING

TpC is in all similar to the former approach, except that in the LDPC program description there is not the concept of a thread executing a codeword. For instance *core-per-codeword* (CpC) in an x86 CPU implies that one thread, corresponding to a logical core will execute the LDPC decoder in some of the physical cores. However, the program can be explicitly defined in terms of an executing thread, which decodes a codeword (see Fig. 9), hence, the distinction between two approaches would otherwise be blurred. Also, typically multi-threading is explored to elevate the parallelism levels and improve the decoding throughput performances by exploiting a higher occupancy of the hardware. Namely, Abburi *et al.* propose a Cell-based LDPC decoder where a thread per *synergistic processing element* (SPE) is assigned with the execution of the longest length rate 1/2 802.16e codewords, peaking at 270 Mbit/s [8].

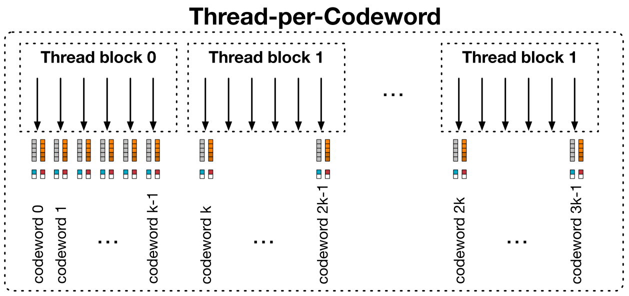


FIGURE 9. Thread-per-Codeword LDPC decoder thread-parallelism approach.

Furthermore, other authors propose this approach for the many-core GPU architecture [46] and compare the performance of their approach to their previously presented LDPC decoder [70], reducing by one order of magnitude the time required to perform BER Monte Carlo simulation for a Mackay code [94]. Also, Lin *et al.* were able to achieve decoding throughputs in the range 212∼550 Mbit/s, though for high latencies (53∼421 ms) using short to long length codes (204∼20000 bits) [50]. Finally, Wang *et al.*, in order to assess the performance of the construction of QC-LDPC convolutional code developed a TpC decoder peaking at 15 Mbit/s (using 768 to 1536 rate 1/2 and 2/3 codes) [71].

8) CpC DECODING

CpC is a thread-parallelism granularity that has no equivalent method in GPU computing, it is only available to CPU architectures. Herein, we consider the logical core definition of "hyperthreaded" processors, which defines a core as equivalent to an execution thread. Thus, a logical core will be responsible for executing a codeword or batch of codewords. However, the scenario under consideration is somewhat vaster here, as several approaches can be taken to implement this task-parallelism strategy.

For the upcoming exascale computing platforms [104], Diavastos *et al.* studied the scalability of LDPC decoders under 1) distributed and 2) shared memory cooperative execution model, and 3) shared memory non-cooperative model [22]. Regarding scalability, 1) saw a reduction of the throughput to less than 1% of the single core baseline reference when all cores were committed to the computation, mainly due to high communication overheads caused by absence of caching mechanisms, 2) reported a sub-linear scalability of up to 11× when 48 cores were committed to the computation, while 3), presented a 41× speedup when compared to the baseline [22]. Other approaches with regards to distributed computing involve the use of streaming accelerators applied to Mackay codes [94] and achieve moderate throughputs (<79 Mbit/s) for low latencies between 0.69∼1.53 ms [28], [32]. 802.16e standard codes can be decoded at throughputs of 72∼80 Mbit/s under this methodology on the Cell B.E. processor [35]. Furthermore, this approach is also explored on a mobile SOC platform (a Quad ARM *system on a chip* (SOC) Exynos 4412), whereupon short and normal frame DVB-T2 codes have been tested, reaching high latencies peaking in the 500∼2592 ms range at throughputs of ∼3 Mbit/s [36].

While some of the aforementioned LDPC decoders do not make use of vector processing [22], SIMD processing is a widely employed technique to improve high performance and efficiency in CPU architectures. Namely, the LDPC decoders based on the Cell B.E. make use of extensive SIMD-instructions by increasing the data parallelism within each core [28], [32]. The work proposed by Falcao *et al.* for their x86-based decoder is a particular type of CpC strategy. The OPenMP model was used to parallelize the computation inside the CN and the VN processing that were encapsulated by loops. Therefore, their true approach was Processor-per-Codeword, which in a sense is a special case of the CpC strategy [32]. Also, Intel CPU-based LDPC decoders are able to explore SSE and AVX SIMD-extensions to improve the data throughput while keeping latency at low levels. Le Gal *et al.* proposed a CpC approach where several multiple codewords are decoded simultaneously by all logical cores in the processor, using the 128- SSE and the 256-bit AVX registers of the CPU to set the decoding throughput within 250∼560 Mbit/s, for CMMB, 802.11n, 802.16e and *digital video broadcasting - satellite 2* (DVB-S2) codes [75]. Furthermore, their approach is able to minimize the decoding latency, which is kept at under 10 ms in the majority of the cases, with 802.11n and 802.16e codes in the hundreds of μs range.

E. DATA-LEVEL PARALLELISM

Data-parallelism expresses how the same operations can be applied to different data elements at the same time. Generally speaking, we can define it, with regards to LDPC decoding, as the number of codewords that are decoded simultaneously. The motivation for exploring data-parallelism is clear since short to moderate length codes cannot utilize all the resources that multi-core and many-core processor architectures possess. Thus, to avoid wasting logic resources that would otherwise be sitting idle, several codewords are loaded and decoded simultaneously to elevate the decoding throughput. However, herein lies a tradeoff. Not only does the decoding throughput sees diminishing returns as the hardware occupancy is elevated, but decoding latency, a figure of merit of the computational performance that should be kept low, also increases. Therefore, only a handful of data-parallelism strategies elevate the decoding throughput to the desired high levels without sacrificing latency beyond admissible levels for real-time operation [64], [75].

1) TAXONOMY

Similar to the thread-parallelism case, a proper taxonomy is due for data-parallelism within LDPC decoders on programmable hardware. Moreover, regarding data-parallelism, the differences between methods that solely concern one type of processor but not the other do not exist. Each of the presented methods is exploited on both CPUs and GPUs alike. Furthermore, because data representation is tightly coupled to the design decisions regarding data-parallelism, it is herein discussed as well.

2) CODEWORD BATCH

CPU and GPU memory engines are optimized for certain alignments. Memory transactions bandwidth can be increased by moving increasingly larger data types until the memory engine saturates at the maximum permitted alignment. Instead of storing an LLR using a `float` data type, a `float4` type can be utilized to store 4 LLR contiguously. In fact, data-parallelism strategies go even further and apply bit slicing operations, usually not natively supported by C/C++ languages, unless by SIMD intrinsics, to pack more data elements into a vector type. Considering the negligible BER performance loss when data is no longer represented using floating-point, but instead low bitwidths fixed-point types are used (typically between 4 and 8 bits [105]), an `int` data type can be employed to store 4 data elements and an `int4` 128 bit vector type [99] can store 16 codewords [34]. Single codeword and codeword batch storage is depicted in Fig. 10.

When data-parallelism levels cannot be raised by increasing the number of elements in a data type, reducing each

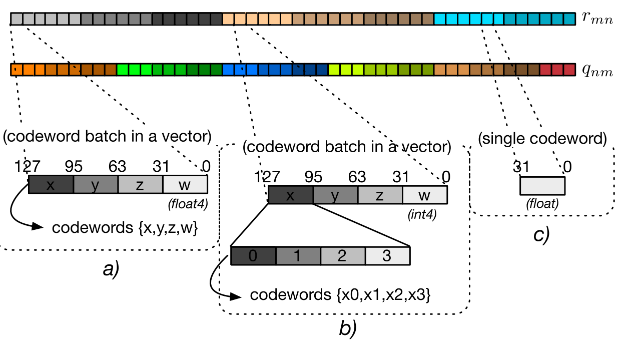


FIGURE 10. Single codeword and codeword batch on a vector. Data elements are stored a) several elements per vector type without resorting to bit slicing operations, b) several elements per vector type but bit slicing enables the packing of more data elements, or c) a single element is stored per data type.

element bitwidth would hurt the BER performance, and going as further as defining a custom data structure will fail short of improving the bandwidth once the maximum alignment permitted is surpassed [34]. At this point, increasing the number of codeword batches must see the replication of the memory layout of the methodology pursued for a single data type, in one of two approaches possible.

3) PADDED CODEWORD BATCHES

Under this approach the basic level of data-parallelism within a batch is replicated as whole with a memory stride equal to the number of data type elements in memory. Thus, codeword batches become padded in memory, as shown in Fig. 11, with the Tanner graph indexing method applied D times to D different base offsets to memory. This method does not impose any relevant constraint to the BpC, TpC or CpC approaches. In fact, throughput performance can only reach acceptable levels once data-parallelism levels are raised [36], [46]. Using a TpN approach this means the spawning of more threads to deal with the extra batches of codewords in the TpN approach [33], [47]–[49], [63].

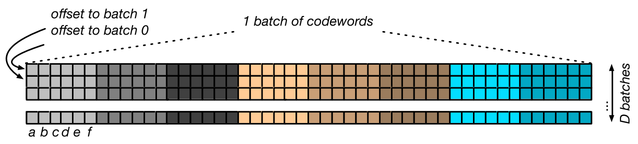


FIGURE 11. Padded data-parallelism approach. The Tanner graph indexing method is replicated for *D* codeword batches by padding copies of layout consecutively in memory.

One of the disadvantages behind this method is the inability to address unbalanced memory transactions that may occur when thread-parallelism does not account for threads having different memory access patterns. More data is accessed for higher CN and VN degrees. Thus, for irregular codes, there can be certain threads computing and moving a higher load of data than others. This problem is addressed by Kang *et al.* by evening out the accesses among threads in the same thread block [43].

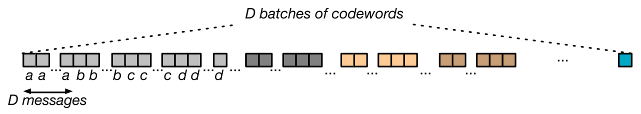


FIGURE 12. Interleaved data-parallelism approach. The Tanner graph indexing method is replicated for *D* codeword batches by interleaving data elements in memory.

4) INTERLEAVED CODEWORD BATCHES

This method, illustrated in Fig. 12, defines D codeword batches as the data-parallelism level and interleaves data elements from different batches at basic data type granularity in memory. The advantage drawn here is that accesses are evened out to large blocks of data moved to and from contiguous locations, regardless of the Tanner graph indexing method. This method is highly suited for SIMD computation in x86 CPUs, with cache hit rations for short to moderate length codes reaching 90% [75]. Furthermore, this method is also highly efficient for GPU architectures, enabling real-time decoding throughputs and simultaneously real-time decoding latencies [64].

F. DECODING ALGORITHMS

Among the countless decoding algorithms, by far, the most popular ones are soft-decoding message-passing ones. In particular, LLR-based algorithms are adopted in the majority of decoders on programmable hardware. Not only does the support of floating-point data types makes them an appealing choice due to easiness of implementation, but also because of their BER performance, even when a step further is taken and fixed-point data types are emulated—as they are not natively supported on CPU and GPU hardware. An exception to this observation is the decoder based on the *impl.-efficient reliability ratio-based weighted bit-flipping alg.* (IRRWBF) [61]. However, the required arithmetic operations in fixed-point and hard-decision operations are emulated using integer types. The most favored choice is for decoding algorithms whose underlying architecture instruction set architecture (ISA) can provide the required datatypes without emulation or with little overhead. In fact, the most popular choice is the MSA in its uncorrected version, offset-corrected OMSA or normalized-corrected NMSA variations. SPA decoders in the probability domain, in the pmf Fourier domain (FFT-SPA), in the LLR domain (LSPA), and in the signed-log Fourier domain (signed-log FFT-SPA) can be found, and also the odd Min-Max and *parity likelihood ratio algorithm* (PLRA) decoder.

The decoding algorithm choice can be tightly coupled to the data type representation chosen for a particular decoding design. Probability domain decoders use floating-point types, as they extensively rely on multiplication, with multiplication not supported natively on programmable hardware in fixed-point types. As a consequence, opportunities to improve the decoding throughput by increasing data-parallelism will be limited by this design decision. GPU hardware, usually aligned for 128 bit data types can pack only 4 floating-point words, while they can pack 16 fixed-point words with a bitwidth of 8 bits [32], and a similar trade-off is expected on CPUs, though they usually implement more sophisticated integer arithmetic than GPUs. As a consequence, all the *sum-product algorithm* (SPA) decoders explore single-precision floating-point (32-bit), and though some MSA-based decoders also do so, the majority of them rely on 6∼8 bit fixed-point data representations. This way, parallelism can be raised by increasing the number of words inside a data type defined by the programming model and language.

G. DECODING SCHEDULES

The decoding of LDPC codes can be scheduled mainly in two approaches. First approach is the so-called *flooding* or TPMP schedule. In this type of scheduling, the exposed parallelism lies at the complete dimension of the LDPC code, since all nodes can be schedule for processing one type of node at a time. Thus, all CNs can be updated at the same time, and all VNs can also be updated at the same time, provided that the CN and the VN processing is not concurrent. As a consequence, when developing a parallel programmable decoder, a memory fencing mechanism which prevents the scheduling for execution of nodes that are consuming messages from nodes which have not still updated their produced messages is required. Otherwise *write-after-read*s (WARs) hazards unfold. Notwithstanding, this is not particularly challenging to guarantee on either CPU, GPU or other accelerator devices, so as long as CN processing and VN process is defined by different functions or kernels. This way, the function or kernel call implicitly sets a synchronization routine preventing any WAR hazard. LDPC decoders using this decoding schedule are among those reaching the highest decoding throughputs, since the TPMP schedule is usually accompanied by a heavily multi-thread approach, usually TpN.

The TDMP schedule seen in the LDPC decoders that implement it are CN-based, i.e., CNs are scheduled for execution sequentially and after each CN is updated, their adjacent VNs ($N(m)$) are updated *on-the-fly* as well [106]. As this decoding schedule is applied to LDPC codes designed for the TDMP, such as QC-LDPC codes, this allows the execution of z_f CNs and their adjacent VNs simultaneously as it does not unfold any WAR hazard. The potential for high throughputs for this decoding schedule as been shown for both CUDA-enabled GPUs [7], [48], [49], the Cell B.E. accelerator [8] and a conventional Intel x86 CPU [75]—decoding throughputs range from 140 to 900 Mbit/s. Other approaches [42], [44], [45] fail short of such high throughputs, but are still in the same range as those obtained with the TPMP schedule. An interesting result is presented for a non-binary LDPC code case, defined over GF(2^4). The authors [57] study both a sequential and a TPMP schedule, based on the BpN approach. For equivalent BER levels achieved, the authors report lower throughputs (3∼8.5 Mbit/s) for the TPMP than for the sequential approach (5∼12.5) Mbit/s.

The TPMP, or flooding schedule, is the most widely implemented decoding schedule. However, a certain misconception may lie in the heart of this design preference. This type of decoding algorithm is the one permitting highest level of simultaneous scheduling of operations. All CNs can be scheduled for execution at the same time, and the same holds for the VNs, provided the execution of CNs and VNs does not overlap in time. On the contrary the TDMP, despited converging faster and reducing the number of required decoding iterations to reach the same BER by roughly half, can only schedule a limited number of operations. If the LDPC code design has not been constructed with this scheduling in mind, there can be as little as no opportunity to schedule more than a single node at a time, though in practice this does not happen as the widely standardized quasi-cyclic codes are designed with this in mind. However, the TPMP schedule implies a data consumption/production pattern for each individual node where each message is accessed once per decoding iteration and per processing phase—for instance, a q_{nm} message is produced by VN$_n$ and is consumed by CN$_m$. This type of access pattern benefits little from a cache system. On the other hand, under the TDMP schedule, where data locality can be exploited temporally for short to moderate code lengths [75]. For earlier GPU generations this advantage meant little, as there was no caching system available. On newer models, L1 caches can exploit this feature of the TDMP schedule. In fact, among the surveyed LDPC decoders, the highest decoding throughputs found for CPU and GPU architecture uses the TDMP [48], [75].

IV. FUTURE DIRECTIONS: RECONFIGURABLE ARCHITECTURES USING HIGH-LEVEL SYNTHESIS

Efforts to survey the LDPC decoders developed for reconfigurable computing [83] would span out of the intended scope for this manuscript. In particular, we refrain from dwelling into reconfigurable LDPC decoders that are not developed using HLS models, with, by far and large, the great majority of decoders found in the literature for reconfigurable computing developed using traditional RTL approaches, and as a consequence, a limited set of decoders fits in this requirement [107]–[110].

A. PROGRAMMING MODELS

OpenCL has recently become supported by the major FPGA manufacturers [87], [111], the OpenCL programming model used for the development of an LDPC decoder [33] is the Silicon-to-OpenCL academic tool [112]. The tool takes in OpenCL kernel C descriptions, though not fully compliant to the OpenCL specification [80], and generates a custom wide-pipeline accelerator.

Moreover, the Vivado HLS [89] defines a comprehensive support for the C/C++ programming languages that get mapped onto circuits on the FPGA board based on a number of HLS directives that instruct how the tool should perform optimizations to different traits of the language (see Fig. 13). It supports optimizations to 1) memory blocks, 2) arithmetic functions, 3) dataflow directives for loops and functions, through pipeline or unrolling and 4) instantiation of certain IP

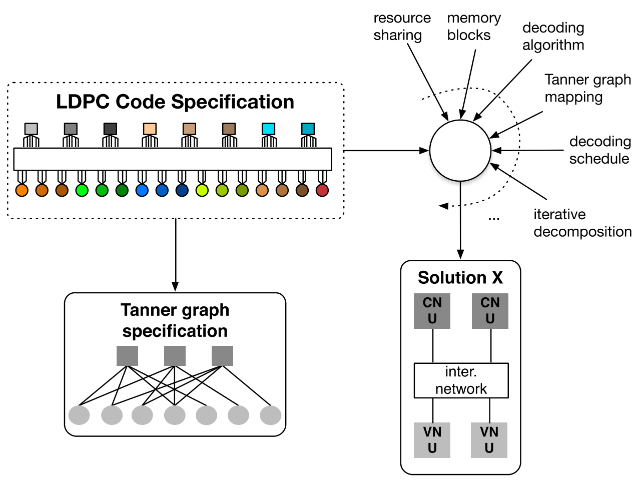


FIGURE 13. Tanner graph isomorphic mapping under a generalized reconfigurable computing approach.

cores in the C/C++ language for I/O interaction with other logic blocks [111].

B. PARALLELISM

Notwithstanding, the fact that the OpenCL programming model defines the concept of work-items, a similar concept to execution threads, in the reconfigurable fabric, the generated accelerator defines no such physical nor logical element that is an execution thread. In fact, computation will be defined by the circuits configuration, thus while data-parallelism concepts remain perfectly valid, there is not thread-parallelism equivalent taxonomy to the reconfigurable LDPC decoders case.

Nevertheless, we are able to define the OpenCL LDPC decoder on FPGA, in its inception a TPN decoder, as a wide-pipeline decoder [33], and the Vivado HLS decoder as a wid-epipeline accelerator as well, though, this approach defined the TPMP node processing phases in computation loops [58]. As a consequence, we prefer the designation of loop-annotated decoder since it is through the optimization directives written as annotations (directives) to loops where computation occurs that the hardware generation is guided. Both approaches see modest throughputs of dozens of Mbit/s achieved for short to moderate length codes. The greatest advantage with this approach is the low latency, ranging <3 ms in the OpenCL decoder case and <500 μs in the Vivado HLS case.

V. SUMMARY

LDPC decoders on programmable hardware can mostly be applied to simulation purposes, due to the methodology pursued in most of the literature be prone to increasing the decoding latency. Notable exceptions to this tradeoff, are the works of Le Gal and Jego [75] and Wang *et al.* [66], which effectively keep latencies at low and real-time compliant levels. Notwithstanding, surveying the decoders in the literature, we observe that the better suited strategies for LDPC decoding are based on LLR-based decoding algorithms, mostly defining fixed-point data representation. This allows for the packing of multiple messages with small bitwidths, usually in the 8 range, to be packed onto wider words. Furthermore, data-parallelism levels are usually pushed beyond the wide word, or vector datatype, granularity, often at the expense of spawning more threads in the decoding underlying architecture. Task-parallelism employed in the literature is explored at all conceived levels, from coarse (CpC) to fine-granularity (TpE), although the strategies attaining the highest performance are mostly fine-grained ones. In particular, the TPN task-parallelism granularity has scored the most prevalent method to expose parallelism for computation.

Regarding LDPC decoders in reconfigurable hardware, the surveyed LDPC decoders on HLS programming models show that this field provides interesting prospects, but remains a larger untapped field. In particular, it remains unclear how to best direct an HLS compiler to generate efficient hardware [113]. The incipient maturity of the tools used in the LDPC decoders [33], [58] already attain competitive decoding throughput and latency, as observed during the inception of LDPC decoding on programmable many-core and multi-core architectures. Furthermore, other programming models such as the Altera OpenCL, more recent versions of the Vivado infrastructure and the Maxeler dataflow decoders [88] promise much lower NRE efforts to target LDPC decoders with high throughputs and higher energy efficiency than programmable computer architectures [85].

The reader is referred to Appendix A, in particular to the exhaustive set of surveyed LDPC decoders (Table 1), which can be found as a supplementary file to this manuscript.

REFERENCES

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

[2] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *Electron. Lett.*, vol. 33, no. 6, pp. 457–458, 1997.

[3] C. Roth, A. Cevrero, C. Studer, Y. Leblebici, and A. Burg, "Area, throughput, and energy-efficiency trade-offs in the VLSI implementation of LDPC decoders," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2011, pp. 1772–1775.

[4] T. Brack *et al.*, "A survey on LDPC codes and decoders for OFDM-based UWB systems," in *Proc. IEEE 65th Veh. Technol. Conf. (VTC-Spring)*, Apr. 2007, pp. 1549–1553.

[5] C. Studer, N. Preyss, C. Roth, and A. Burg, "Configurable high-throughput decoder architecture for quasi-cyclic LDPC codes," in *Proc. 42nd Asilomar Conf. Signals, Syst. Comput.*, Oct. 2008, pp. 1137–1142.

[6] D. B. Kirk and W. H. Wen-Mei, *Programming Massively Parallel Processors: A Hands-On Approach*. San Mateo, CA, USA: Morgan Kaufmann, 2012.

[7] K. K. Abburi, "A scalable LDPC decoder on GPU," in *Proc. 24th Int. Conf. VLSI Design (VLSI Design)*, 2011, pp. 183–188.

[8] K. K. Abburi, "Cell processor based LDPC encoder/decoder for WiMAX applications," in *Proc. Int. Conf. Soft Comput. Problem Solving (SocProS)*, vol. 131. India, Dec. 2011, pp. 781–790.

[9] J. Andrade, G. Falcão, and V. Silva, "Flexible design of wide-pipeline-based WiMAX QC-LDPC decoder architectures on FPGAs using high-level synthesis," *Electron. Lett.*, vol. 50, no. 11, pp. 839–840, 2014.

[10] J. Andrade, G. Falcão, V. Silva, and K. Kasai, "FFT-SPA non-binary LDPC decoding on GPU," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2013, pp. 5099–5103.

[11] J. Andrade, F. Pratas, G. Falcão, V. Silva, and L. Sousa, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era," in *Proc. IEEE 25th Int. Conf. Appl.-Specific Syst., Archit. Process. (ASAP)*, Jun. 2014, pp. 264–269.

[12] J. Andrade, G. Falcão, V. Silva, and K. Kasai, "Flexible non-binary LDPC decoding on FPGAs," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 1936–1940.

[13] M. Beermann, E. Monró, L. Schmalen, and P. Vary, "High speed decoding of non-binary irregular LDPC codes using GPUs," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2013, pp. 36–41.

[14] F. L. Blasco and C. Tang, "Implementation of a multi-user detector for satellite return links on a GPU platform," in *Proc. 7th Adv. Satellite Multimedia Syst. Conf., 13th Signal Process. Space Commun. Workshop (ASMS/SPSC)*, Sep. 2014, pp. 66–72.

[15] C. H. Chan and F. C. M. Lau, "Parallel decoding of LDPC convolutional codes using OpenMP and GPU," in *Proc. IEEE Symp. Comput. Commun. (ISCC)*, Jul. 2012, pp. 000225–000227.

[16] C. H. Chan and F. C. M. Lau, "Simulation of LDPC convolutional decoders with CPU and GPU," in *Proc. 2nd Int. Conf. Consum. Electron., Commun. Netw. (CECNet)*, Apr. 2012, pp. 2854–2857.

[17] C.-C. Chang, Y.-L. Chang, M.-Y. Huang, and B. Huang, "Accelerating regular LDPC code decoders on GPUs," *IEEE J. Sel. Topics Appl. Earth Observ. Remote Sens.*, vol. 4, no. 3, pp. 653–659, Sep. 2011.

[18] C.-C. Chang, M.-Y. Huang, and Y.-L. Chang, "Design of GPU-based platform for LDPC decoder," in *Proc. IEEE Int. Geosci. Remote Sens. Symp.*, Jul. 2011, pp. 3429–3432.

[19] Y.-L. Chang, C.-C. Chang, M.-Y. Huang, and B. Huang, "High-throughput GPU-based LDPC decoding," *Proc. SPIE*, vol. 7810, pp. 781008-1–781008-8, Aug. 2010.

[20] H.-P. Cheng, Y.-C. Shen, J.-L. Wu, and K. Aizawa, "High efficient distributed video coding with parallelized design for cloud computing," in *Proc. 19th ACM Int. Conf. Multimedia (MM)*, New York, NY, USA, Nov. 2011, p. 1257.

[21] A. D. Copeland, N. B. Chang, and S. Leung, "GPU accelerated decoding of high performance error correcting codes," in *Proc. 13th Workshop High Perform. Embedded Comput.*, 2009.

[22] A. Diavastos, P. Petrides, G. Falcão, and P. Trancoso, "LDPC decoding on the Intel SCC," in *Proc. 20th Euromicro Int. Conf. Parallel, Distrib. Netw.-Based Process.*, Feb. 2012, pp. 57–65.

[23] G. Falcão, S. Yamagiwa, V. Silva, and L. Sousa, "Stream-based LDPC decoding on GPUs," in *Proc. 1st Workshop General Purpose Process. Graph. Process. Units (GPGPU)*, 2007, pp. 1–7.

[24] G. F. P. Fernandes, V. M. M. da Silva, M. A. C. Gomes, and L. A. P. S. de Sousa, "Edge stream oriented LDPC decoding," in *Proc. 16th Euromicro Conf. Parallel, Distrib. Netw.-Based Process. (PDP)*, Feb. 2008, pp. 237–244.

[25] G. Falcão, S. Yamagiwa, V. Silva, and L. Sousa, "Parallel LDPC decoding on GPUs using a stream-based computing approach," *J. Comput. Sci. Technol.*, vol. 24, no. 5, pp. 913–924, Sep. 2009.

[26] G. Falcão, J. Andrade, V. Silva, S. Yamagiwa, and L. Sousa, "Stressing the BER simulation of LDPC codes in the error floor region using GPU clusters," in *Proc. Int. Symp. Wireless Commun. Syst. (ISWCS)*, Aug. 2013, pp. 1–5.

[27] G. Falcão, V. Silva, J. Marinho, and L. Sousa, "Parallel LDPC decoding on the Cell/B.E. processor," in *High Performance Embedded Architectures and Compilers* (Lecture Notes in Computer Science), vol. 5409, A. Seznec, J. Emer, M. O'Boyle, M. Martonosi, and T. Ungerer, Eds. Berlin, Germany: Springer, 2009, pp. 389–403.

[28] G. Falcão, L. Sousa, and V. Silva, "Embedded multicore architectures for LDPC decoding," in *Proc. Int. Conf. Embedded Comput. Syst., Archit., Modeling Simulation (SAMOS)*, Jul. 2010, pp. 349–356.

[29] G. Falcão, J. Andrade, V. Silva, and L. Sousa, "GPU-based DVB-S2 LDPC decoder with high throughput and fast error floor detection," *Electron. Lett.*, vol. 47, no. 9, pp. 542–543, Apr. 2011.

[30] G. Falcão, J. Andrade, V. Silva, and L. Sousa, "Real-time DVB-S2 LDPC decoding on many-core GPU accelerators," in *Proc. Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2011, pp. 1685–1688.

[31] G. Falcão, L. Sousa, and V. Silva, "Massive parallel LDPC decoding on GPU," in *Proc. 13th ACM SIGPLAN Symp. Principles Pract. Parallel Program. (PPoPP)*, Feb. 2008, pp. 83–90.

[32] G. Falcão, L. Sousa, and V. Silva, "Massively LDPC decoding on multicore architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 22, no. 2, pp. 309–322, Feb. 2011.

[33] G. Falcão *et al.*, "Shortening design time through multiplatform simulations with a portable OpenCL golden-model: The LDPC decoder case," in *Proc. IEEE 20th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, Apr./May 2012, pp. 224–231.

[34] G. Falcão, V. Silva, L. Sousa, and J. Andrade, "Portable LDPC decoding on multicores using OpenCL [applications corner]," *IEEE Signal Process. Mag.*, vol. 29, no. 4, pp. 81–109, Jul. 2012.

[35] G. Falcão, V. Silva, L. Sousa, and J. Marinho, "High coded data rate and multicodeword WiMAX LDPC decoding on Cell/BE," *Electron. Lett.*, vol. 44, no. 24, pp. 1415–1416, Nov. 2008.

[36] S. Grönroos and J. Björkqvist, "Performance evaluation of LDPC decoding on a general purpose mobile CPU," in *Proc. IEEE Global Conf. Signal Inf. Process.*, Dec. 2013, pp. 1278–1281.

[37] S. Grönroos, K. Nybom, and J. Björkqvist, "Efficient GPU and CPU-based LDPC decoders for long codewords," *Analog Integr. Circuits Signal Process.*, vol. 73, pp. 583–595, Jun. 2012.

[38] S. Grönroos, K. Nybom, and J. Björkqvist, "Complexity analysis of software defined DVB-T2 physical layer," *Analog Integr. Circuits Signal Process.*, vol. 69, nos. 2–3, pp. 131–142, 2011.

[39] H. Ji, J. Cho, and W. Sung, "Massively parallel implementation of cyclic LDPC codes on a general purpose graphics processing unit," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, Oct. 2009, pp. 285–290.

[40] H. Ji, J. Cho, and W. Sung, "Memory access optimized implementation of cyclic and quasi-cyclic LDPC codes on a GPGPU," *J. Signal Process. Syst.*, vol. 64, no. 1, pp. 149–159, 2011.

[41] B. Jiang, J. Bao, and X. Xu, "Efficient simulation of QC LDPC decoding on GPU platform by CUDA," in *Proc. IEEE Int. Conf. Wireless Commun. Signal Process. (WCSP)*, Oct. 2012, pp. 1–5.

[42] J. Cui, Y. Wang, and H. Yu, "Systematic construction and verification methodology for LDPC codes," in *Wireless Algorithms, Systems, and Applications* (Lecture Notes in Computer Science), vol. 6843, Y. Cheng, D. Y. Eun, Z. Qin, M. Song, and K. Xing, Eds. Berlin, Germany: Springer, 2011.

[43] S. Kang and J. Moon, "Parallel LDPC decoder implementation on GPU based on unbalanced memory coalescing," in *Proc. IEEE Int. Conf. Commun. (ICC)*, Jun. 2012, pp. 3692–3697.

[44] J. A. Kennedy and D. L. Noneaker, "Decoding of a quasi-cyclic LDPC code on a stream processor," in *Proc. MILITARY Commun. Conf. (MILCOM)*, Oct./Nov. 2010, pp. 2062–2067.

[45] J. A. Kennedy and D. L. Noneaker, "Scheduling parity checks for increased throughput in early-termination, layered decoding of QC-LDPC codes on a stream processor," *EURASIP J. Wireless Commun. Netw.*, vol. 2012, no. 1, pp. 1–10, 2012.

[46] F. C. M. Lau and L. Shi, "Programming graphics processing units for the decoding of low-density parity-check codes," in *Proc. 14th Int. Conf. Adv. Commun. Technol. (ICACT)*, 2012, pp. 1002–1005.

[47] Y. Zhao and F. C. M. Lau, "Implementation of decoders for LDPC block codes and LDPC convolutional codes based on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 3, pp. 663–672, Mar. 2014.

[48] B. Le Gal, C. Jego, and J. Crenne, "A high throughput efficient approach for decoding LDPC codes onto GPU devices," *IEEE Embedded Syst. Lett.*, vol. 6, no. 2, pp. 29–32, Jun. 2014.

[49] R. Li, J. Zhou, Y. Dou, S. Guo, D. Zou, and S. Wang, "A multi-standard efficient column-layered LDPC decoder for software defined radio on GPUs," in *Proc. IEEE 14th Workshop Signal Process. Adv. Wireless Commun. (SPAWC)*, Jun. 2013, pp. 724–728.

[50] Y. Lin and W. Niu, "High throughput LDPC decoder on GPU," *IEEE Commun. Lett.*, vol. 18, no. 2, pp. 344–347, Feb. 2014.

[51] F. J. Martínez-Zaldívar, A. M. Vidal-Maciá, A. Gonzalez, and V. Almenar, "Tridimensional block multiword LDPC decoding on GPUs," *J. Supercomput.*, vol. 58, no. 3, pp. 314–322, Mar. 2011.

[52] Y.-S. Pai, H.-P. Cheng, Y.-C. Shen, and J.-L. Wu, "Fast decoding for LDPC based distributed video coding," in *Proc. Int. Conf. Multimedia (MM)*, New York, NY, USA, 2010, pp. 1211–1214.

[53] Y.-S. Pai, Y.-C. Shen, and J.-L. Wu, "High efficient distributed video coding with parallelized design for LDPCA decoding on CUDA based GPGPU," *J. Vis. Commun. Image Represent.*, vol. 23, no. 1, pp. 63–74, 2012.

[54] J.-Y. Park and K.-S. Chung, "LDPC decoding for CMMB utilizing OpenMP and CUDA parallelization," in *Proc. 17th Asia-Pacific Conf. Commun. (APCC)*, Oct. 2011, pp. 910–914.

[55] J.-Y. Park and K.-S. Chung, "Parallel LDPC decoding using CUDA and OpenMP," *EURASIP J. Wireless Commun. Netw.*, vol. 2011, no. 1, pp. 1–8, 2011.

[56] F. Pratas, J. Andrade, G. Falcão, V. Silva, and L. Sousa, "Open the gates: Using high-level synthesis towards programmable LDPC decoders on FPGAs," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2013, pp. 1274–1277.

[57] D. L. Romero and N. B. Chang, "Sequential decoding of non-binary LDPC codes on graphics processing units," in *Proc. IEEE ASILOMAR*, Nov. 2012, pp. 1267–1271.

[58] E. Scheiber, G. H. Bruck, and P. Jung, "Implementation of an LDPC decoder for IEEE 802.11n using Vivado high-level synthesis," in *Proc. Int. Conf. Electron., Signal Process. Commun. Syst.*, vol. 4. 2013, pp. 45–48.

[59] T.-C. Su, Y.-C. Shen, and J.-L. Wu, "Real-time decoding for LDPC based distributed video coding," in *Proc. 19th ACM Int. Conf. Multimedia (MM)*, New York, NY, USA, Nov. 2011, pp. 1261–1264.

[60] H. P. Thi, S. Ajaz, and H. Lee, "Efficient Min-Max nonbinary LDPC decoding on GPU," in *Proc. Int. SoC Design Conf. (ISOCC)*, 2014, pp. 266–267.

[61] H. Tiwari, H. N. Bao, and Y. B. Cho, "A parallel IRRWBF LDPC decoder based on stream-based processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 23, no. 12, pp. 2198–2204, Dec. 2012.

[62] T.-H. Chiu, H.-K. Kuo, and B.-C. C. Lai, "A highly parallel design for irregular LDPC decoding on GPGPUs," in *Proc. Asia-Pacific Signal Inf. Process. Assoc. Annu. Summit Conf. (APSIPA ASC)*, 2012, pp. 1–5.

[63] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "GPU accelerated scalable parallel decoding of LDPC codes," in *Proc. Asilomar Conf. Signals, Syst., Comput. (ASILOMAR)*, 2011, pp. 2053–2057.

[64] G. Wang, H. Shen, B. Yin, M. Wu, Y. Sun, and J. R. Cavallaro, "Parallel nonbinary LDPC decoding on GPU," in *Proc. IEEE ASILOMAR*, Nov. 2012, pp. 1277–1281.

[65] G. Wang, M. Wu, Y. Sun, and J. R. Cavallaro, "A massively parallel implementation of QC-LDPC decoder on GPU," in *Proc. IEEE Symp. Appl. Specific Process. (SASP)*, Washington, DC, USA, Jun. 2011, pp. 82–85.

[66] G. Wang, M. Wu, B. Yin, and J. R. Cavallaro, "High throughput low latency LDPC decoding on GPU for SDR systems," in *Proc. IEEE Global Conf. Signal Inf. Process. (GlobalSIP)*, Dec. 2013, pp. 1258–1261.

[67] S. Wang, S. Cheng, and Q. Wu, "A parallel decoding algorithm of LDPC codes using CUDA," in *Proc. 42nd Asilomar Conf. Signals, Syst. Comput.*, 2008, pp. 171–175.

[68] S. Wang, L. Cui, S. Cheng, and R. C. Huck, "GPU acceleration for particle filter based LDPC decoding," in *Proc. nVidia Res. Summit GPU Technol. Conf. (GTC)*, San Jose, CA, USA, 2009, pp. 1–6.

[69] X. Wen *et al.*, "A high throughput LDPC decoder using a mid-range GPU," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, May 2014, pp. 7515–7519.

[70] S. F. Yau, T. L. Wong, and F. C. M. Lau, "Extremely fast simulator for decoding LDPC codes," in *Proc. 13th Int. Conf. Adv. Commun. Technol. (ICACT)*, 2011, pp. 635–639.

[71] Y. Wang, H. Yu, and Y. Xu, "Quasi-cyclic low-density parity-check convolutional code," in *Proc. IEEE 7th Int. Conf. Wireless Mobile Comput., Netw. Commun. (WiMob)*, Oct. 2011, pp. 351–356.

[72] L. Yuan, Z. Xing, Y. Zhang, and X. Chen, "An optimizing strategy research of LDPC decoding based on GPGPU," in *Proc. 12th IEEE Int. Conf. Trust, Secur. Privacy Comput. Commun.*, Jul. 2013, pp. 1901–1906.

[73] Y. Zhao, X. Chen, C.-W. Sham, W. M. Tam, and F. C. M. Lau, "Efficient decoding of QC-LDPC codes using GPUs," in *Algorithms and Architectures for Parallel Processing* (Lecture Notes in Computer Science), vol. 7016, Y. Xiang, A. Cuzzocrea, M. Hobbs, and W. Zhou, Eds. Berlin, Germany: Springer, 2011, pp. 294–305.

[74] A. Mink and A. Nakassis, "LDPC error correction for Gbit/s QKD," *Proc. SPIE*, vol. 9123, p. 912304, May 2014.

[75] B. Le Gal and C. Jego, "High-throughput multi-core LDPC decoders based on x86 processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 5, pp. 1373–1386, May 2015.

[76] J. D. Owens *et al.*, "A survey of general-purpose computation on graphics hardware," *Comput. Graph. Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[77] J. Nickolls and W. J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, Mar./Apr. 2010.

[78] R. Vuduc and K. Czechowski, "What GPU computing means for high-end systems," *IEEE Micro*, vol. 31, no. 4, pp. 74–78, Jul./Aug. 2011.

[79] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Boston, MA, USA: Addison-Wesley, 2000.

[80] *OpenCL 2.0 Specification*, Khronos Group, Beaverton, OR, USA, 2014.

[81] *TOP500 The List*, accessed on May 2016. [Online]. Available: http://www.top500.org

[82] W. S. Carter *et al.*, "A user programmable reconfigurable logic array," in *Proc. IEEE Custom Integr. Circuits Conf.*, May 1986, pp. 233–235.

[83] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proc. IEEE*, vol. 103, no. 3, pp. 332–354, Mar. 2015.

[84] A. DeHon, "The density advantage of configurable computing," *Computer*, vol. 33, no. 4, pp. 41–49, Apr. 2000.

[85] J. M. Rabaey, "Reconfigurable processing: The solution to low-power programmable DSP," in *Proc. IEEE Int. Conf. Acoust., Speech, Signal Process. (ICASSP)*, vol. 1. Apr. 1997, pp. 275–278.

[86] J. Varghese, M. Butts, and J. Batcheller, "An efficient logic emulation system," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 1, no. 2, pp. 171–174, Jun. 1993.

[87] Altera Corp. (2013). *Altera SDK for OpenCL Optimization Guide*. [Online]. Available: http://www.altera.com/literature/hb/opencl-sdk/aocl_optimization_guide.pdf

[88] O. Pell and V. Averbukh, "Maximum performance computing with dataflow engines," *Comput. Sci. Eng.*, vol. 14, no. 4, pp. 98–103, Jul. 2012.

[89] Xilinx. (2015). *Vivado Design Suite User Guide; High-Level Synthesis*. [Online]. Available: http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_3/ug902-vivado-high-level-synthesis.pdf

[90] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE J. Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, Oct. 1974.

[91] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge, U.K.: Cambridge Univ. Press, 2008.

[92] C. Roth, C. Benkeser, C. Studer, G. Karakonstantis, and A. Burg, "Data mapping for unreliable memories," in *Proc. 50th Annu. Allerton Conf. Commun., Control, Comput. (Allerton)*, Oct. 2012, pp. 679–685.

[93] H. Jin, A. Khandekar, and R. McEliece, "Irregular repeat-accumulate codes," in *Proc. 2nd Int. Symp. Turbo Codes Rel. Topics*, 2000, pp. 1–8.

[94] *Encyclopedia of Sparse Graph Codes*, accessed on May 2016. [Online]. Available: http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[95] M. Gomes, G. Falcão, V. Silva, V. Ferreira, A. Sengo, and M. Falcão, "Flexible parallel architecture for DVB-S2 LDPC decoders," in *Proc. IEEE Global Telecommun. Conf. (GLOBECOM)*, 2007, pp. 3265–3269.

[96] A. Morello and V. Mignone, "DVB-S2: The second generation standard for satellite broad-band services," *Proc. IEEE*, vol. 94, no. 1, pp. 210–227, Jan. 2006.

[97] S. Yamagiwa and L. Sousa, "Caravela: A novel stream-based distributed computing environment," *Computer*, vol. 40, no. 5, pp. 70–77, May 2007.

[98] *CUDA C Programming Guide 7.5*, NVIDIA, Santa Clara, CA, USA, Sep. 2015.

[99] J. Fang, A. L. Varbanescu, and H. Sips, "A comprehensive performance comparison of CUDA and OpenCL," in *Proc. Int. Conf. Parallel Process. (ICPP)*, Sep. 2011, pp. 216–225.

[100] T. Richardson and R. Urbanke, "The renaissance of Gallager's low-density parity-check codes," *IEEE Commun. Mag.*, vol. 41, no. 8, pp. 126–131, Aug. 2003.

[101] D. J. Costello, Jr., and G. D. Forney, "Channel coding: The road to channel capacity," *Proc. IEEE*, vol. 95, no. 6, pp. 1150–1177, Jun. 2007.

[102] K. Kasai, R. Matsumoto, and K. Sakaniwa, "Information reconciliation for QKD with rate-compatible non-binary LDPC codes," in *Proc. Int. Symp. Inf. Theory Appl. (ISITA)*, Oct. 2010, pp. 922–927.

[103] K. Kasai and K. Sakaniwa, "Fourier domain decoding algorithm of non-binary LDPC codes for parallel implementation," *IEICE Trans. Fundam. Electron., Commun. Comput. Sci.*, vol. E93-A, no. 1, pp. 1949–1957, 2010.

[104] R. Hazra, "The explosion of petascale in the race to exascale," in *Proc. Int. Supercomput. Conf.*, Hamburg, Germany, Jun. 2012.

[105] M. P. C. Fossorier, M. Mihaljevic, and H. Imai, "Reduced complexity iterative decoding of low-density parity check codes based on belief propagation," *IEEE Trans. Commun.*, vol. 47, no. 5, pp. 673–680, May 1999.

[106] M. M. Mansour, "A turbo-decoding message-passing algorithm for sparse parity-check matrix codes," *IEEE Trans. Signal Process.*, vol. 54, no. 11, pp. 4376–4392, Nov. 2006.

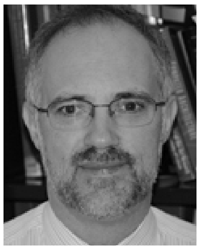
[107] Y. Cai, S. Jeon, K. Mai, and B. V. K. V. Kumar, "Highly parallel FPGA emulation for LDPC error floor characterization in perpendicular magnetic recording channel," *IEEE Trans. Magn.*, vol. 45, no. 10, pp. 3761–3764, Oct. 2009.

[108] J. Ding and M. Yang, "eIRA LDPC codes on FPGA," *IEEE Commun. Lett.*, vol. 15, no. 6, pp. 665–667, Jun. 2011.
[109] F. Verdier and D. Declercq, "A low-cost parallel scalable FPGA architecture for regular and irregular LDPC decoding," *IEEE Trans. Commun.*, vol. 54, no. 7, pp. 1215–1223, Jul. 2006.
[110] Y. Dai, Z. Yan, and N. Chen, "Optimal overlapped message passing decoding of quasi-cyclic LDPC codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 5, pp. 565–578, May 2008.
[111] Xilinx Inc. *The Xilinx SDAccel Development Environment*, accessed on May 2016. [Online]. Available: http://www.xilinx.com/publications/prod_mktg/sdnet/sdaccel-backgrounder.pdf
[112] M. Owaida, N. Bellas, K. Daloukas, and C. D. Antonopoulos, "Synthesis of platform architectures from OpenCL programs," in *Proc. IEEE 19th Annu. Int. Symp. Field-Program. Custom Comput. Mach. (FCCM)*, May 2011, pp. 186–193.
[113] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed. The Netherlands: Springer, 2008.
[114] X.-Y. Hu, E. Eleftheriou, and D. M. Arnold, "Regular and irregular progressive edge-growth tanner graphs," *IEEE Trans. Inf. Theory*, vol. 51, no. 1, pp. 386–398, Jan. 2005.
[115] J. Bao, Y. Zhan, J. Wu, and J. Lu, "Design of efficient low rate QCARA GLDPC codes," in *Proc. IET Int. Commun. Conf. Wireless Mobile Comput. (CCWMC)*, Dec. 2009, pp. 213–216.
[116] J. Lin, J. Sha, Z. Wang, and L. Li, "Efficient decoder design for nonbinary quasicyclic LDPC codes," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 5, pp. 1071–1082, May 2010.

JOAO ANDRADE received the Ph.D. degree in electrical and computer engineering from the Faculty of Science and Technology, University of Coimbra, in 2016. He conducted his Ph.D. research with the MSP-Co Group, Instituto de Telecomunicações, and performed two internships with the LAP Group, EPFL, Switzerland, and Xilinx Research Labs, Dublin, in 2013/14. Recently, he joined the R&D Department, Coriant GmbH, Lisbon, Portugal, where he is a Hardware Engineer. His research activities focus on architectures for error-correction and their resiliency to unreliable memory systems. He is an Affiliated Member of the HiPEAC network.

GABRIEL FALCAO (S'07–M'10–SM'14) received the degree in electrical and computer engineering and the M.Sc. degree in digital signal processing from the University of Porto, and the Ph.D. degree from the University of Coimbra in 2010. He became an Assistant Professor with the University of Coimbra. In 2011 and 2012, he was a Visiting Professor with EPFL, Switzerland. In 2013, he was a recipient of a Google Faculty Research Award and the Altera Europe-Wide University contest 2012-2013. Presently, he is studying efficient parallelization strategies, novel algorithms and architectures for dealing with compute-intensive applications used in medical, ultrasound, and deep neural network imaging contexts, in parallel with continuous work in digital communications. He is a Researcher with the Instituto de Telecomunicações, and a Senior Member of the Signal Processing Society and the HiPEAC Network of Excellence.

VITOR SILVA received the Diploma and Ph.D. degrees in electrical engineering from the University of Coimbra, Portugal, in 1984 and 1996, respectively. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Coimbra, where he lectures digital signal processing, and information and coding theory. His research activities in signal processing, image and video compression, and coding theory are mainly carried out with the Instituto de Telecomunicações, Coimbra, Portugal. He has published more than 140 papers, successfully supervised more than 20 graduation theses, is co-author of a patent in video coding, and has participated in eight funded research projects. Currently, he is the Director of the Coimbra IT-site, coordinating the research activities of 40 collaborators.



LEONEL SOUSA (SM'–) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Tecnico, Universidade de Lisboa (UL), Lisbon, Portugal, in 1996, where he is currently a Full Professor. He is also a Senior Researcher with the R&D Instituto de Engenharia de Sistemas e Computadores. His research interests include VLSI architectures, computer architectures, parallel computing, computer arithmetic, and signal processing systems. He has contributed to more than 200 papers in journals and international conferences, for which he got several awards, such as the DASIP'13 Best Paper Award, the SAMOS'11 'Stamatis Vassiliadis' Best Paper Award, the DASIP'10 Best Poster Award, and several Honorable Mention Awards from the Universidade Tecnica de Lisboa/Santander Totta (2007, 2009) and UL/Santander (2016) for the quality and impact of his scientific publications. He has contributed to the organization of several international conferences, namely, as the Program Chair and General and Topic Chair, and has given keynotes in some of them. He has edited four special issues of international journals, and he is currently an Associate Editor of the IEEE TRANSACTIONS ON MULTIMEDIA, the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY, the IEEE ACCESS, *IET Electronics Letters*, and *Journal of Real-Time Image Processing* (Springer), and the Editor-in-Chief of the *EURASIP Journal on Embedded Systems*. He is fellow of the IET and a Distinguished Scientist of ACM.