IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 25, 2017, accepted July 3, 2017, date of publication July 14, 2017, date of current version August 14, 2017.

Digital Object Identifier 10.1109/ACCESS.2017.2727221

Design Space Exploration of LDPC Decoders Using High-Level Synthesis

JOAO ANDRADE[1], NITHIN GEORGE[2], KIMON KARRAS[3], DAVID NOVO[4], (Member, IEEE), FREDERICO PRATAS[5], (Member, IEEE), LEONEL SOUSA[5], (Senior Member, IEEE), PAOLO IENNE[2], (Senior Member, IEEE), GABRIEL FALCAO[1], (Senior Member, IEEE), AND VITOR SILVA[1]

[1]Instituto de Telecomunicações and Department of Electrical and Computer Engineering, University of Coimbra, 3030-290 Coimbra, Portugal
[2]Processor Architecture Laboratory, School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, 1015 Lausanne, Switzerland
[3]Think-Silicon, Patras Science Park, Rion Achaias 26504, Greece
[4]French National Centre for Scientific Research (CNRS), University of Montpellier, LIRMM, 34090 Montpellier, France
[5]INESC-ID, Instituto Superior Técnico, Universidade de Lisboa, 1000-029 Lisboa, Portugal

Corresponding author: João Andrade (jandrade@co.it.pt)

This work was supported in part by Fundação para a Ciência e a Tecnologia through 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 Today, *high-level synthesis* (HLS) tools are being touted as a means to perform rapid prototyping and shortening the long development cycles needed to produce hardware designs in *register transfer level* (RTL). In this paper, we attempt to verify this claim by testing the productivity benefits offered by current HLS tools by using them to develop one of the most important and complex processing blocks of modern *software-defined radio* systems: the forward error correction unit that uses *low density parity-check* (LDPC) codes. More specifically, we consider three state-of-the-art HLS tools and demonstrate how they can enable users with little hardware design expertise to quickly explore a large design space and develop complex hardware designs that achieve performances that are within the same order of magnitude of handcrafted ones in RTL. Additionally, we discuss how the underlying computation model used in these HLS tools can constrain the microarchitecture of the generated designs and, consequently, impose limits on achievable performance. Our prototype LDPC decoders developed using HLS tools obtain throughputs ranging from a few Mbits/s up to Gbits/s and latencies as low as 5 ms. Based on these results, we provide insights that will help users to select the most suitable model for designing LDPC decoder blocks using these HLS tools. From a broader perspective, these results illustrate how well today's HLS tools deliver upon their promise to lower the effort and cost of developing complex signal processing blocks, such as the LDPC block we have considered in this paper.

INDEX TERMS Error correction codes, reconfigurable architectures, accelerator architectures, reconfigurable logic, high level synthesis.

I. INTRODUCTION

Traditionally, implementing a relatively complex processing algorithm on a *field-programmable gate array* (FPGA) started with developing a *register transfer level* (RTL) description of a digital circuit to perform the computation. However, producing such a RTL description is a tedious task where one needs to detail each low-level circuit operations, such as the movement of data between hardware registers (i.e., flip-flops) and the individual operations performed on this data. Therefore, developing hardware designs was only possible for hardware designers who had the necessary skills. Today, however, there exist *high-level synthesis* (HLS) tools that promise to enable users without such specialized skills to develop complex hardware designs. Additionally, HLS tools enable users to shorten the design development cycles and efficiently explore a large design space and identify designs that achieve the appropriate trade-offs between performance and resource requirements [1]. Furthermore, since these HLS tools use traditional software development languages, e.g., C, C++ and *Open Computing Language* (OpenCL), it enables users to easily migrate existing implementation on platforms such as *central processing unit*s (CPUs) and *graphics processing unit*s (GPUs) to target FPGAs.

2169-3536 © 2017 IEEE. Translations and content mining are permitted for academic research only.
Personal use is also permitted, but republication/redistribution requires IEEE permission.
See http://www.ieee.org/publications_standards/publications/rights/index.html for more information.

Software-defined radio (SDR) systems [2] contain several complex signal processing blocks that must be carefully optimized to achieve the optimum balance between performance, system complexity and development time. One such compute-intensive signal processing block is the *low-density parity-check* (LDPC) *forward error correction* (FEC) codes that are often used as error correcting blocks within advanced communication systems. LDPC codes were invented in the early sixties by R. G. Gallager [3], however, they remained largely unnoticed until the early nineties, when there was enough computational power to harness their capacity-approaching characteristics. Due to the powerful error-correction capabilities of LDPC codes over a noisy channel, today they have been widely adopted by multiple IEEE, ITU-T and ETSI digital communication standards [4]–[11]. Although CPUs and GPUs are often used to simulate these codes, the bulk of deployed LDPC decoders are in the form of dedicated *very large scale integration* (VLSI) devices [7], [12]. However, these implementations are fixed and cannot be modified as standards evolve or error-correction requirements change. On the other hand, reconfigurable substrates, such as FPGAs, are capable of supporting SDR applications [13] and they can be reprogrammed to satisfy different requirements or to deploy multiple communication standards.

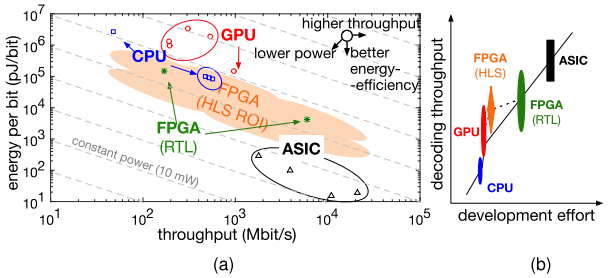


FIGURE 1. Energy efficiency vs. throughput tradeoffs of the LDPC decoders obtained using different computing architectures: (a) operating regions of the different decoder technologies employed; and (b) development effort relative to the potential decoding throughputs. The projected FPGA HLS *region of interest* (ROI) is highlighted in (a).

In this paper, we investigate how one can utilize HLS tools to reduce effort and time (see Figure 1) needed to develop complex LDPC decoding solutions. More specifically, we discuss how a designer can perform design space exploration and develop FPGA solutions that can handle the various intricacies of LDPC codes, such as complexity of decoding algorithms for both binary and non-binary LDPC codes, regularity of code structure, parallel-exposure of the decoding schedules and *Turbo-decoding message-passing* (TDMP) or *two-phased message-passing* (TPMP). We demonstrate how users without specialized hardware design expertise can easily develop LDPC decoders using three state of the art HLS tools: Altera OpenCL, MaxCompiler and Xilinx Vivado HLS. Moreover, the proposed decoder designs produced by the HLS tools achieve performances that are comparable to those of RTL designs.

The main contributions of this paper can be summarized as follows:

- identification of the key challenges in using HLS approaches for designing distinct LDPC decoder architectures;
- assessment of the attainable LDPC decoding performance (i.e., throughput and latency) for HLS-based decoder designs;
- discussion on how the underlying architecture constrains the LDPC decoder design space and, therefore, attainable decoding performance;
- discussion and ranking of the HLS proposed decoders against RTL-based approaches available in the literature.

II. LDPC CODES AND DECODING ALGORITHMS

Introduced in the early sixties [3], and left untamed until the early nineties due to insufficient computational power to prove their capacity-approaching abilities, LDPC codes are linear block codes defined by sparse parity-check matrices $\mathbf{H}$ that verify the condition

$$\mathbf{c}\times\mathbf{H}^{\mathrm{T}} = 0, \quad \mathbf{c} \in \mathbf{C}, \tag{1}$$

with $\mathbf{c}$ a codeword belonging to the set of codewords $\mathbf{C}$ lying in the null-space of $\mathbf{H}$. The parity-check matrix defines the adjacency matrix to the Tanner graph (see Fig. 2), a bipartite graph that assigns a *check node* (CN) to each row in $\mathbf{H}$, and *variable node*s (VNs) to $\mathbf{H}$ columns. Whenever a non-null element h_{cv} exists, there is an edge connecting CN_c to VN_v [14]. The codeword $\mathbf{c}$ can be a binary value, in which case a VN corresponds to a single bit, or it can be a non-binary symbol in the Galois Field GF(q), typically an m-tuple of bits in digital communication systems—the symbol is defined over the binary extension field GF(2^m).[1]

A. DECODING ALGORITHMS

Decoding transmitted codewords can be performed by hard decoding, but is mostly performed using soft-decoding methods that present improved coding gains over the former [14]. Soft-decoding algorithms are based on message-passing between CNs and VNs that compose the Tanner graph. In Figure 2, each VN corresponds to a codeword symbol and each CN to a parity-check equation [14]. The channel demodulator computes an *a-priori* stochastic measure m_v to each symbol in the *log-likelihood ratio* (LLR) or *probability mass function (pmf)* domain, $L(m_v)$ or $\mathbf{m}_v(x)$, respectively, where x is a symbol defined over the binary extension field GF(2^m). Then, according to the connections defined in the Tanner graph of the LDPC code, each VN broadcasts the initial estimate as $L(m_{vc})$ or $\mathbf{m}_{vc}(x)$ messages across their edges towards the set of adjacent CNs $C(v)$. At the CN level, new messages $L(m_{cv})$ or $\mathbf{m}_{cv}(x)$ are computed and sent back to the set of adjacent VNs $V(c)$ which compute new $L(m_{vc})$ or $\mathbf{m}_{vc}(x)$

[1] It is usual to define the primitive element of a given Galois Field generically as α. With the exception of the zero element, all remaining symbols in the field, can be written as powers of α [14].

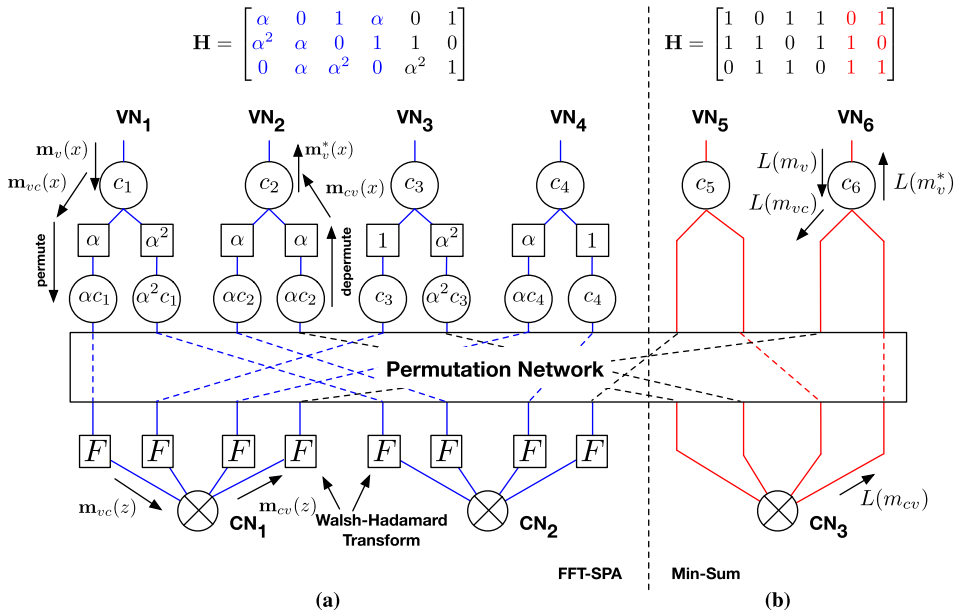


FIGURE 2. Parity-check matrix and Tanner graph representation in (a) GF(2^m) (blue) and (b) GF(2) (red), depicting the message-passing between nodes for the (a) non-binary FFT-SPA and the (b) MSA.

estimates and repeat the procedure. The binary case presents a simpler algorithm with no mathematical operation applied at the edge-level, whereas in the non-binary case, a permutation is applied to all *pmf* traversing the graph from VN to CN and are depermuted on the opposite direction. This is formalized as $\mathbf{m}_{cv}(h_{cv}^{-1}\times h_{cv}\times x)$. With each iteration, an *a-posteriori* likelihood estimate, $L(M_v)$ or $\mathbf{M}_v(x)$, can be computed from which a hard-decision regarding the most likely symbol state can be made.

Among the set of decoding algorithms that can be used, the MSA and the FFT-SPA are particularly interesting for the binary and the non-binary cases, respectively [15], [16]. The MSA is composed of CN processing (2), a new *a-posteriori* estimate on the most likely bit state (3) performed also in conjunction with the VN processing (4), and formalized below with i denoting the current decoding iteration:

$$L(m_{vc}^{(i)}) = \min_{v'\in V(c)\setminus v} |L\left(m_{cv'}^{(i-1)}\right)| \times \prod_{v'\in V(c)\setminus v} \text{sign}\left(m_{cv'}^{(i-1)}\right) \tag{2}$$

$$L(M_v{}^{(i)}) = L(m_v) + \sum_{c'\in C(v)} L(m_{vc'}^{(i)}) \tag{3}$$

$$L(m_{cv}^{(i)}) = L(M_v^{(i)}) - L(m_{cv}^{(i)}). \tag{4}$$

The non-binary decoding is more complex as it can scale with $\sim$O($m\cdot 2^m$), making the practical decoding of non-binary LDPC codes much more challenging than binary ones [14], [17]. Some of the known decoding algorithms with the lowest complexity include *min-max* (MM), *extended min-sum* (EMS), their Trellis versions *trellis min-max* (TMM) and *trellis extended min-sum* (TEMS), and FFT-SPA [16]–[18]. Among them, FFT-SPA is of particular interest since it forgos any operation over GF(2^m) and the logic overhead of finding the required configuration sets needed in TMM and TEMS [17]. The FFT-SPA is composed of CN processing (5) performed in the Fourier domain and *a-posteriori* estimate on the most likely symbol state (6) together with the VN processing (7):

$$\mathbf{m}_{cv}^{(i)}(h_{cv}\times x) = \mathsf{F}\left(\frac{\prod_{v'\in V(c)\setminus v} \mathbf{m}_{v'c}^{(i-1)}(z)}{\mathbf{m}_{v'c}^{(i-1)}(z=0)}\right) \tag{5}$$

$$\mathbf{M}_v^{(i)}(x) = m_v(x) \prod_{c'\in C(v)} \mathbf{m}_{vc'}^{(i)}(x), \tag{6}$$

$$\mathbf{m}_{vc}^{(i)}(x) = \mathbf{M}_v^{(i)}(x)/\mathbf{m}_{cv}^{(i)}(x), \tag{7}$$

with $x \in$ GF(2^m) and the Fourier transform $\mathbf{m}_{vc}(z)$ of the *pmf* $\mathbf{m}_{vc}(x)$ given by $\mathbf{m}_{vc}(z) = \mathsf{F}(\mathbf{m}_{vc}(x))$, where $\mathsf{F}(\cdot)$ is the *Walsh-Hadamard transform* (WHT) operator—the *discrete Fourier transform* (DFT) is also the WHT but in the binary-extension field domain [14].

The workload for the binary and non-binary decoding algorithms include the CN and VN processing and additional edge-level operations in the non-binary case, but the numerical complexity is typically dominated by the CN computation. Therefore, a designer must carefully consider all the arithmetic and memory operations involved in various parts of the algorithm during the implementation. Table 1 gives upper bounds for the complexity of MSA and FFT-SPA algorithms expressed in terms of CN arithmetic and memory instructions required per decoding iteration for the CN, VN and edge-level processing. This data shows that the decoding procedures scale linearly with the LDPC code

TABLE 1. Upper bound on the complexity of arithmetic and memory operations.

Alg.	Operations					
	Arithmetic					Memory
	Comp.	XOR	Add.	Sub.	Mult.	
MSA	$2Md_c + M$	$2Md_c$	$Nd_c + N$	Nd_c	N/A	$2(Nd_v + Md_c)$
FFT-SPA	N/A		$M{\cdot}m(2^m - 1)$	$M{\cdot}m(2^m - 1)$	$2^m(Md_c + Nd_v)$	$(2^{m+1} + 1)(Nd_v + Md_c)$

dimension and with the executed number of decoding iterations, but non-linearly with the order of GF(2^m).

III. ARCHITECTURES FOR HLS-BASED DESIGN

In this work, we use three different state of the art HLS tools and discuss how the features offered by these tools can be efficiently leveraged to model good quality LDPC decoder architectures. We specifically study how decoders can be implemented using a wide-pipeline architecture, a dataflow approach and a loop-annotated method,[2] respectively, provided by the Altera OpenCL, Maxeler MaxCompiler and Vivado HLS, whose main features are summarized in Figure 3 and that are discussed next.

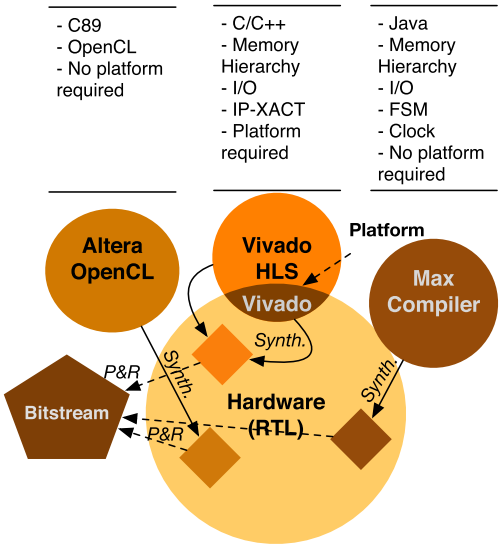


FIGURE 3. HLS approaches relation to RTL development and features supported by each model (on top).

A. WIDE-PIPELINE ARCHITECTURE

The wide-pipeline architecture in this study is modeled using the Altera OpenCL HLS infrastructure [16]. The OpenCL is a parallel programming model that defines specific memory hierarchies and makes it easy for the users to express parallelism in the algorithm. Altera's HLS tool generates kernels from OpenCL code and interconnects them within a template architecture, shown in Figure 4a). Therefore, only a part of the FPGA is used for the OpenCL kernels and the rest is used for other hardware structures, such as the PCIe IP block for moving data between the host (CPU) and the device (FPGA), *dynamic RAM* (DRAM) controllers and clocking interfaces. While it is possible to tailor the architecture to the specific application, this needs the designer to know advanced hardware-level details and deviate markedly from a typical OpenCL-based implementation flow. Without availing these advanced features, the generated architecture might not fully leverage the specialization capabilities of an FPGA.

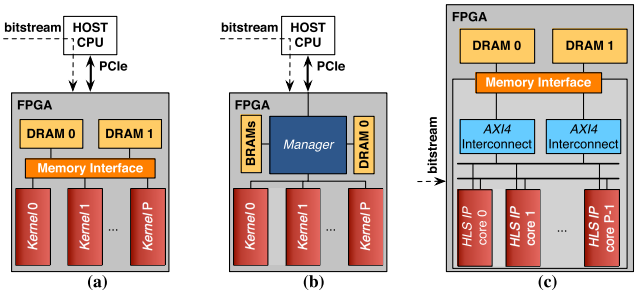


FIGURE 4. Accelerator platform topology: (a) wide-pipeline Altera OpenCL; (b) MaxCompiler; (c) Vivado HLS developed platform. The former two are provided by the HLS infrastructure, the latter is provided by the designer.

1) OpenCL WIDE-PIPELINE MODEL

The *work-item* defines the finest-grained element at which level the computation is defined under OpenCL [19]. Parallelism is exploited through the generation of an accelerator capable of holding hundreds or thousands of active *work-items* in the pipeline, therefore the designation of wide-pipeline. *Work-items*, organized into three dimensional "grids" of computation, called *workgroups*, that essentially form a triple-nested loop structure, with a loop per grid dimension. In the kernels generated by the HLS tool, the entire computation is pipelined so that multiple *work-items* can be processed simultaneously. Additionally, the tool automatically maps the different memories in the OpenCL standard to available on-chip or off-chip memories: `global` memory space is mapped to off-chip DRAM, `local` and `constant` memory spaces are mapped to on-chip *block RAM*s (BRAMs) [16]. Accesses to these memory addressing spaces are managed via interconnection networks to which the kernels generated by the HLS tool are connected. In a typical implementation, most of the optimizations performed by the user directly affect the kernel and the interconnection network is generated by the tool itself.

[2]This designation stems from the exposure of parallelism through loop directives that drive the pipelining and unrolling the iterations scheduled within.

2) PARALLELISM IN THE WIDE-PIPELINE

In OpenCL, a *workgroup*, essentially, models computation within a triple-nested loop structure with explicit parallelism across *work-items*. In the case of the FPGA, as noted earlier, this parallelism is exploited by performing computation in a pipelined fashion. Therefore, a designer must write C code which is functionally correct and at the same time, exposes enough parallelism to reach high levels of pipelining. The efficiency of generated pipeline is measured in terms of the *initiation interval* (II) of *work-items*, which is the number of cycles that must elapse after accepting a *work-item* before it can accept the next one. Although the designer has explicit no control over the II, the HLS tool automatically tries to process 1 *work-item* per clock cycle at a given clock frequency. Therefore, complex algorithms or poorly constructed code will result in lower clock frequencies of operation.

From a designers' perspective, the performance of a design can be improved by 1) setting a fixed *workgroup* size, this removes all loop-guard constraints and facilitates more aggressive pipelining; 2) directing the tool to generate several *compute unit*s (CUs), k_{CUs} to handle the execution of *work-items* in the pipeline; 3) driving the level of *single-instruction multiple-data* (SIMD) computation upwards (in powers of 2) to k_{SIMD}-way SIMD execution of *work-items*.

B. DATAFLOW ARCHITECTURE

Maxeler's MaxCompiler provides a HLS infrastructure that enables the high-level description of dataflow hardware accelerators [20]. This is facilitated by a number of Java-based classes that abstract away the underlying FPGA platform which contains the necessary memory controllers that enable the communication of the host computer system with the FPGA chip—enabling constructs such as *finite-state machine* (FSM) control. The model is inspired in stream-based approaches that decouple arithmetic operations from memory accesses and data movement. The underlying platform, seen in Figure 4b), interconnects a given set of *kernel* accelerators to a *manager* that is responsible for serving the *kernels* with data and for all the data movement to and from the dataflow accelerator [20].

Under the dataflow model, a *kernel* is defined as a hardware datapath performing the arithmetic and logical computations as the data flows through it. A *manager* is responsible for orchestrating the kernel calls and feeding the kernel with the data needed for the computation via off-chip I/Os, in a streaming fashion. The compiler also uses a streaming model for off-chip I/O to the PCIe, to implement so-called *dataflow engine*s (DFEs) via the MaxRing interconnection [20], and to the DRAM memory. The objective is to keep the utilization of the available off-chip communication bandwidth high, without the need for users to dig deeper onto low-level FSMs that control the flow of data. With this approach, by keeping communication and computation separate, *kernels* can be deeply pipelined without encountering synchronization issues—both communication and computation occur concurrently.

C. LOOP-ANNOTATED ARCHITECTURE

Vivado provides a HLS toolchain that enables users to design hardware from C/C++ or SystemC description [21]. In addition to this description, the user can provide additional inputs, such as inline compiler directives using the `#pragma` construct or using a separate `TCL` script, to further optimize the generated hardware. The HLS tool compiles the inputs and synthesizes a hardware module, in the IP-XACT format, which implements the computation inside a top-level function that is marked within the user description. During this C-synthesis, all the functions, logic and arithmetic, inside this top-level are mapped onto hardware primitives. At this stage, the behavior of the generated design can be analyzed for functional correctness at the clock cycle level. Then, the generated hardware module is connected within a larger platform design, such as the one shown in Figure 4c), before performing the circuit synthesis. Among the available directives, we can highlight the following that are of interest to the generation of efficient LDPC decoders: 1) loop directives that deal with unrolling, pipelining and how complex loop structures can be flattened or merged; 2) memory directives that influence how memory is mapped, e.g., arrays can be partitioned or reshaped in block or cyclically; 3) resource directives that integrate specific hardware blocks onto the HLS description such as BRAMs, multipliers, FIFOs or protocol-specific I/O blocks.

Concurrency can be hard to express in C/C++ without suitable extensions, such as the parallelism defined at the *work-item* level in OpenCL kernels, thus, loop pipelining and loop unrolling are responsible for the bulk of parallelism exposed by the algorithm description.

IV. HLS PARALLEL LDPC DECODERS

Herein, we discuss the development of LDPC decoders for each of the HLS approaches.

A. WIDE-PIPELINE BINARY LDPC DECODER

Two distinct approaches for wide-pipeline decoders can be followed i) one by exploiting pipelining of *work-items* to the fullest [5] and ii) by defining multiple *kernels*, the latter an approach closer to typical programmable OpenCL approaches [22], as seen on Fig. 5. We refer to the former as pipelined decoder, as presented previously in [5], and the designated multi-kernel decoder is discussed below.

1) THREAD-PER-NODE MULTI-KERNEL LDPC DECODER

In this case, the node- and edge-level functions are synthesized into separate kernels. Algorithm 1 defines the multi-kernel approach that can be used to implement an LDPC decoder for the binary case. The multi-kernel approach enables us to specify parallelism at a very fine-grained level. Otherwise, the unbalanced quantity of work performed at the node- and edge-level within the same kernel can lead to sub-optimal designs. In fact, for kernels that compute the CN or the VN, higher IIs are achieved if they are left without

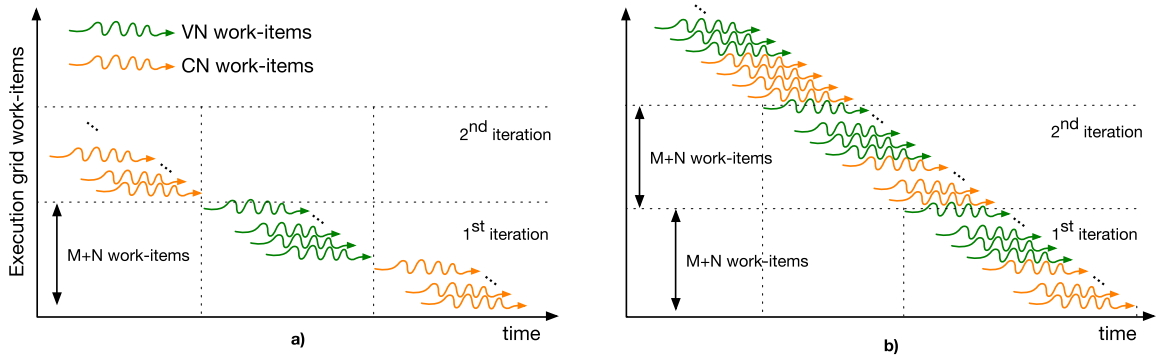


FIGURE 5. Altera OpenCL TpN execution and work-item scheduling: following *a)* a multi-kernel strategy, where an execution grid per iteration per processing stage is issued, and completely flushed, and using *b)* a pipelined approach where work-items are scheduled at once and the single-kernel pipeline is never flushed [5].

performing the update procedures at the edge-level that precede and follow them. Likewise, higher IIs are accomplished for edge-level kernels if no node-level computation is defined.

Algorithm 1 Multi-Kernel TpN MSA Decoding Using the OpenCL Wide-Pipeline Approach

1: **Launch OpenCL multi-kernel TpN decoder kernels**
2: **repeat**
3: *CN processing kernel*
4: **for** $wk_i = 0$ to $M - 1$ **do**
5: Load all $L(m_{vc})$ from DRAM (d_c LLRs per work-item)
6: Execute CN update (2)
7: **end for**
8: Wait for work-items in the CN pipeline to be flushed
9: *VN processing kernel*
10: **for** $wk_i = 0$ to $N - 1$ **do**
11: Load all $L(m_{cv})$ from DRAM (d_v LLRs per work-item)
12: Load $L(m_v)$ from DRAM (1 LLR per work-item)
13: Execute VN updates *a-posteriori* (3) and (4)
14: **end for**
15: Wait for all work-items to be flushed
16: **until** all i iterations are executed
17: Copy $L(m_v^*)$ from FPGA DRAM back to host

B. DATAFLOW BINARY LDPC DECODER

Using the dataflow model for the LDPC decoder provides the ability to more freely define the architecture in HLS, although, as a consequence, it also puts more responsibility on the designer. The additional freedom avoids limitations such as the need to map the physical addressing spaces to logical ones, each with different scopes and variable lifetime. Moreover, we are now able to define *functional unit*s (FUs) at the node-level which can be used to express varying degrees of partially parallel designs [23].

1) *M*-FUNCTIONAL UNIT LDPC DECODER

Given the popularity of *quasi-cyclic LDPC* (QC-LDPC) and *LDPC Irregular-Repeat-Accumulate* (LDPC-IRA) codes in communication standards, we developed a binary LDPC decoder for LDPC-IRA codes with M FUs that utilizes a partially-parallel architecture [23] (see Fig.6). This architecture exploits the modular M properties within the Tanner graph of LDPC-IRA codes. Additionally, by utilizing the streaming model we divide the dataflow accelerator onto a *manager* that handles all data communications from the front- and back-end and the processing block that is connected to the managers and contains one or more *kernels*.

The front-end and back-end interface the input and output streams, respectively, from the external interfaces (e.g., PCIe or DRAM) to BRAM units in the FPGA. The processing block performs the processing of data coming over the input streams. A double-buffering mechanism ensures that at any given time there can be 1) data being read over the input streams, 2) data being processed by processing block and 3) processed data being written to the output stream [24]. The precise control facilitated by the extended-Java language allows the definition of a processing rate for the computation which loads one message per FU per clock cycle.

The actual number, M, of FUs in the system is specified by the designer before synthesis and it is implemented as DFE-array of the defined LDPC FUs. Usually, M is a sub-multiple of the expansion factor z_f of QC-LDPC codes [23] or a sub-multiple of the regularity factor of LDPC-IRA codes [6]. Thus, we can decide between assigning more or less FUs to a decoder based on the required throughput and/or the available resources on the FPGA fabric. Furthermore, the M-modulo architecture assigns a separate memory bank to each FU so that stalls are minimized—this enables simultaneous reading and storing of messages that can be as high as a message per clock cycle.

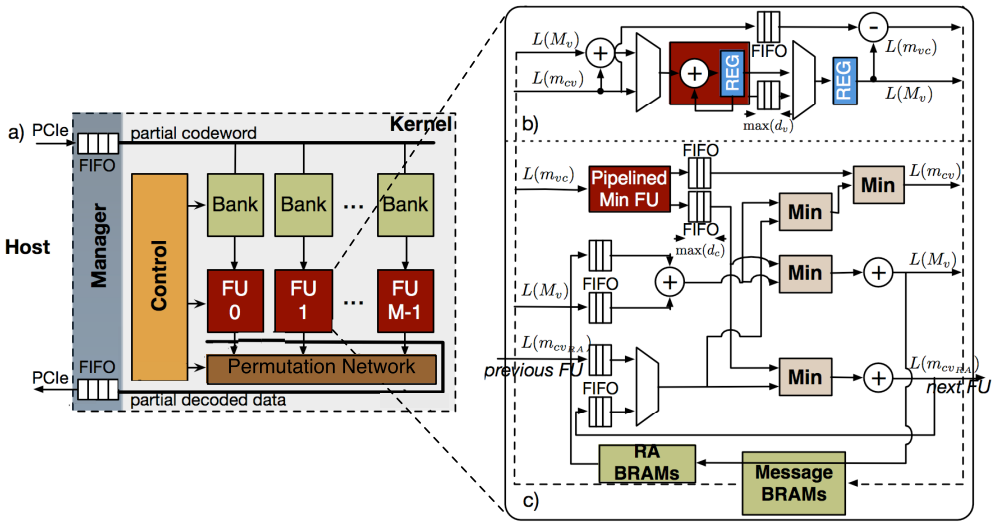


FIGURE 6. Dataflow pipelined MSA FU VN and CN datapaths. a) Host, *manager* and *kernel*, and pipelined FU. The b) VN datapath and c) CN datapath allow for concurrent execution. The shown example instantiates a single decoder, but there is enough bandwidth to instantiate a k_{dec} number of decoders before the PCIe link is saturated.

Other considerations regarding Tanner graph indexing performed by the permutation network (e.g., cyclic-shifters, Benes networks or barrel shifters) are expressed directly in the input specifications to the HLS tool [20]. Increasing the number P of the accelerator creates a larger design, which due to the impaired routing, will have a lower operation frequency. Therefore, after a certain threshold of P, it is desirable to have a structure with multiple *kernels* where the unused FPGA resources are utilized to instantiate additional FUs that have high operation frequency [24]. This modularity is made possible due to the detailed and fine-grained descriptions at the FU-, and at the array-of-FUs-level.

2) PIPELINED FUNCTIONAL UNIT AND DECODER

Fully parallel processing at the node-level would involve computing d_c or d_v messages per clock cycle. Managing the ensuing unbalanced memory accesses and the demand for high bandwidth can be addressed by defining the CN and the VN processing to update sequentially, such that only one BRAM bank is required per VN or CN being processed in a batch of P decoders. This makes the demand for BRAM memory ports scale with P instead of $P\times d_v$ or $P\times d_c$ for a fully parallel design. As seen in Figure 6, each arithmetic macro-function is connected to FIFO units that enables sequential reading and writing of data at the rate of one message per clock cycle. The tool generates appropriate counters which push or pop data on the FIFOs as well as associated registers to perform this operation. The internal CN and VN update operation is formalized in Algorithm 2 at the granularity of the FU-level. The complete decoder is composed of an array of M FUs [20]. Moreover, the pipelined FU supports a dual-mode of operation that supports two groups of active streams at the same time—one group in the CN datapath, and the other in the VN datapath. This feature significantly helps to reduce the logic overhead incurred by the FUs, since the CN and VN datapaths share the control and clock signals, and allows for simultaneous and coherent computation of a new set of CNs along with a trailing set of VNs, and the other way around.

C. LOOP-ANNOTATED NON-BINARY LDPC DECODER

To study the loop-annotated design approach, we use it to implement a more complex case: the non-binary LDPC decoder. In this design, front-end and back-end units stream data from the off-chip DRAM to BRAMs on the FPGA so that data is brought closer to where the computation is performed. The computation unit is generated using the HLS tool from a high-level description. This description details nested-loop structures that perform the computations shown in Figure 2a) and formalized in (5), (6) and (7).

1) LOOP-ACCELERATION

Although we consider non-binary decoding case here, the binary decoding case can be derived from it by trimming out some of the computation. Figure 7 a) shows the loop-structure for the FFT-SPA non-binary LDPC decoding algorithm. The trip count of each loop and its relative position in the nested-loop structure would determine what optimizations are applied on it produce an efficient design using Vivado HLS. We can use the loop-pipelining and loop-unrolling directives to improve the parallelism in the loop computation. However, the effective parallelism in the design will depend on other factors, such as having enough bandwidth to serve data to all the parallel computation in the datapath [25].

To optimize loop structures, we can use loop-unrolling to either unroll it completely and perform all its operations simultaneously, or unroll it only by a factor of k_{unroll},

Algorithm 2 MSA Decoding Using the Dataflow Approach

1: **Host data and DFE management**
2: Initialize data and move it to FPGA DRAM over PCIe
3: **Launch DFE execution**
4: Reset FIFOs, registers and counters (Figure 6)
5: Stream $L(m_{vc})$, $L(m_{cv})$ and $L(m_v)$ from DRAM to BRAMs
6: **repeat**
7: d-th FU (P FUs in parallel)
8: *CN operation*
9: **for** All CUs in the DFE in CN mode **do**
10: Load $L(m_{vc})$ from d-th BRAM bank and push to FIFO
11: FIFO output is streamed to arith. units (Figure 6c))
12: Execute CN update (2)
13: Store updated $L(m_{c-k_{\text{CUs}}v})$ to d-th BRAM bank
14: **end for**
15: *VN operation*
16: **for** All CUs in the DFE in VN mode **do**
17: Load $L(m_{cv})$ from d-th BRAM bank and push to FIFO
18: FIFO output is streamed to arith. units (Figure 6b))
19: Execute VN updates (3) and (4)
20: Store updated $L(m_{v-k_{\text{CUs}}c})$ to d-th BRAM bank
21: **end for**
22: **until** all i iterations are executed
23: **Move data from DFE over PCI**

When commuting operation from CN to VN ($c-k_{CUs}<0$) or from VN to CN ($v-k_{CUs}<0$) the elements in the former mode of operation are still being flushed while the latter are commencing to be updated mantaining coherence.

where k_{unroll} loop iterations are performed simultaneously. Another loop optimization is loop-pipelining which enable the subsequent loop iteration to begin before the previous one has completed. When applying pipelining, the tool aims to achieve a certain II for that loop, which is the number cycles between the start of consecutive loop iterations. When applying this optimizations, perfect inner loop structures are merged into a single loop. In the case of imperfect inner loops, the loop structure is kept the same, i.e., $k_{unroll} = 1$ and the II is set as the loop latency—effectively, there is no pipelining. Naturally, the ability to effectively schedule all k_{unroll} iterations at once or meet the requested II is limited by data dependencies within the loop iterations, logic available to instantiate a higher number of arithmetic resources and bandwidth available to serve a higher memory load.

When applying both unrolling and pipelining together, unrolling outer loops and pipelining the inner loops will instantiate multiple pipelined FUs that process the inner loops. On the other hand, pipelining the outermost loop requires the unrolling of all the inner loops, and results in the generation of a wide-pipeline FU. The former results in lower performance and higher logic utilizations. However, the latter results in a smaller design with higher overall IIs. Therefore, we pipeline the outermost loops (O) and unroll the innermost one (I).

2) MEMORY MAPPING

Similar to the dataflow approach, we can fully tailor the way data flows in the FPGA decoder. Data is initialized and streamed from the FPGA DRAM to BRAMs at the accelerator front-end. BRAMs are physically two-port memories, which can be split onto two single-port half-size BRAMs, and the C-synthesizer's default behavior is to store arrays using the minimum number of required BRAMs with array elements stored sequentially. Typically, one of the BRAM ports is used for writing and the other for reading, limiting in-order data accesses to a rates of a single element per clock cycle. However, there are memory optimizations (e.g., array partitioning and reshaping) that can be used to overcome this limitation.

Partitioning and reshaping directives can be used to split data arrays across multiple BRAM modules such that more ports are physically exposed to the computation units, or more words can be read simultaneously with each read operation. These optimizations increase the data bandwidth to the arithmetic units. However, exposing BRAM ports without instantiating a sufficiently high number of arithmetic units to exploit it does not translate into higher computational performance. In fact, it is the correct combination of the loop and memory directives that leads to the optimal accelerator performance. Using these optimizations, small width data arrays can be reshaped such that each word from the BRAM contain multiple data elements from the array. Additionally, by applying the partitioning optimization, the array elements can be cyclically divided across k_{cyclic} BRAM banks [25] so that non-consecutive elements can be read simultaneously [25], as shown in Figure 7 b). In Algorithm 3, the assumed number of banks is 2^m, allowing fully unrolled accesses by the inner loop iterating over the field dimension to the elements therein stored.

V. EXPERIMENTAL EVALUATION

In this section, we discuss the experimental results obtained using the experimental setup listed in Table 2 and for the dataset detailed in Table 3.

A. APPARATUS AND DATASET

The details of Virtex and Stratix FPGA families utilized in this work are detailed in Table 2. The HLS-based LDPC decoders were developed and analyzed for codes dataset defined in Table 3. These codes were chosen due to their codeblock length, regularity (QC-LDPC and LDPC-IRA), and applicability—the codes used are Wi-Fi and

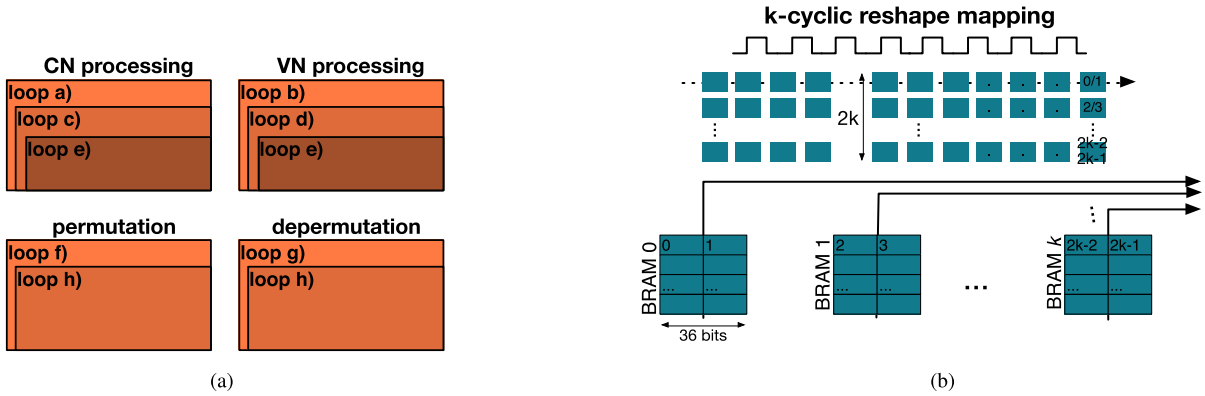


FIGURE 7. Two of the transformations applied in the loop-annotated decoder case: (a) loop nest structure definition to allow effective unroll and pipeline directives compounded by (b) array reshaping for improving bandwidth and scheduling of multiple iterations in parallel.

TABLE 2. Utilized HLS tools and FPGA boards characteristics.

Board	Chip	HLS Tool	ALMs	Slices	FFs	ELBs
Nallatech PCIe 385	Stratix V 5SGSD5	Altera OpenCL SDK 13.0sp1	172K	N/A	690K	690K
MAX3412A	Virtex 6 SX 475T	Maxeler 2012.2.1 MaxCompiler	N/A	74K	595K	595K
VC709	Virtex 7 VX 690T	Xilinx Vivado HLS 2014.4	N/A	108K	866K	866K

1 ALM: 6LUT, 4×FFs; 1 Slice: 4×6LUTs,8×FFs; ELB defined in Section V-B

TABLE 3. Dataset utilized for the LDPC decoders.

		LDPC Code						
		N (bits)	Rate	m	d_c	d_v	z_f	Standard
Dataset	I	64 800	1/2	1	{6, 7}	{2, 3, 8}	N/A	DVB-S2
	II	1944	1/2	1	{7, 8}	{2, 3, 4, 11}	81	WiFi
	III-a)	768		2				
	III-b)	852	1/3	3	3	2	N/A	[26]
	III-c)	1536		4				

DVB - satellite 2nd gen. (DVB-S2) standard codes. The place and route (P&R) results obtained for the logic utilization, operating clock frequency, decoding figures of merit, decoding throughput and decoding latency (for 10 decoding iterations) are summarized in Table 4. The parallelism nomenclature introduced earlier can be summarized also in terms of a number of *processing unit*s (PUs), where PUs = $k_{dec} \times k_{CUs}$.

B. LOGIC UTILIZATION NORMALIZATION

Due to the differences in the FPGA architectures and families, which apply different combinations for number and type of LUTs and number of FFs each slice of logic element possesses, we borrow on the normalizing methodology discussed in the survey work [4]. This rationale adopts a so-called *equivalent logic block* (ELB) as the fined-grained logic element of normalization, composed of one 4LUT and one FF (c.f. Table 2). Thus, the actual logic utilization results are converted onto ELBs so that they can be crossed compared. First, the 6LUT and *adaptive logic module*s (ALMs) of Xilinx and Altera families must be converted into a number of 4LUTs. It is considered that each Xilinx 6LUT is equivalent to two 4LUTs and that each Altera ALM is equivalent to two 4LUTS. To actually obtain the number of ELBs the maximum number of required FFs or converted 4LUTs is then used [4].

TABLE 4. FPGA utilization and performance of the decoders after P&R for the LDPC accelerator supported.

	Dataset	Par. (PUs)		Logic Util.	Performance (10 iter.)		
		k_{dec}	k_{CUs}	ELBs	Clock (MHz)	Thr. (Mbit/s)	Lat (μs)
A)	II	1	1	201300	240	16.00	121.50
		1	2	322852	157	21.00	185.14
B)	I	1	45	32000	260	234.00	276.92
	I	2	45	54000	250	345.00	375.65
	I	4	45	96000	200	720.00	360.00
	I	8	45	188000	150	1080.00	480.00
	I	1	90	54000	260	468.00	138.46
	I	2	90	96000	150	540.00	240.00
	I	4	90	186000	150	1080.00	240.00
	I	8	90	376000	90	1296.00	400.00
	I	1	180	98000	200	720.00	90.00
	I	2	180	190000	170	1224.00	105.88
	I	1	360	190000	175	1260.00	51.43
	I	2	360	382000	80	1152.00	112.50
C)	III-a)	1	1	124000	250	1.17	656.41
	III-a)	14	1	688000	219	14.54	739.48
	III-b)	1	1	182000	250	0.95	1697.69
	III-b)	6	1	698000	210	4.81	1437.01
	III-c)	1	1	258000	216	0.66	2327.27
	III-c)	3	1	632000	201	1.85	2490.81

A) Nallatech PCIe 385; B) MAX3412A; C) VC709

C. WIDE-PIPELINE DECODER

In this section, we present and discuss the experimental results for the wide-pipeline architecture LDPC decoders, for both the pipelined single-kernel and the multi-kernel *thread-per-node* (TpN) approaches.

Algorithm 3 Loop-Annotated FFT-SPA Decoder. The Decoder Below Assumes Pipelined Outer Loops (O) With II = 1, Fully Unrolled Inner Loops (I) and BRAM-Array Reshaping as Detailed in Solution VI (Table 5)

1. **Stream $\mathbf{m}_{vc}(x)$, $\mathbf{m}_{cv}(x)$ and $\mathbf{m}_{v}(x)$ from DRAM to BRAMs**
2. **Launch loop-annotated kernels on the FPGA**
3. **repeat**
4. *VN processing kernel*
5. **for** (O) All VNs **do**
6. **for** (I) All symbols in GF(2^m) **do**
7. **for** (I) All m_{cv} **do**
8. Load $m_{cv}(x = q)$ and $m_{v}(x = q)$ from q-th BRAM bank
9. Execute VN update (7)
10. **end for**
11. Store $m_{vc}(x = q)$ to q-th BRAM bank
12. **end for**
13. **end for**
14. *Permutation/Depermutation*
15. **for** (O) *pmf* in the Tanner Graph **do**
16. **for** (I) All symbols in GF(2^m) **do**
17. Permute *pmf*
18. **end for**
19. **end for**
20. *FWHT kernel*
21. **for** (O) *pmf* in the Tanner Graph **do**
22. **for** (I) All symbols in GF(2^m) **do**
23. Load one *pmf* (VN) or Fourier *pmf* (CN)
24. **end for**
25. Perform radix-2 butterfly computation
26. **for** (I) All symbols in GF(2^m) **do**
27. Store the Fourier *pmf* (VN) or *pmf* (CN)
28. **end for**
29. **end for**
30. *CN processing kernel*
31. Similar to described in lines: 5 to 13, but in the opposite direction of Tanner Graph traversal (5)
32. *FWHT kernel* (execute 20:29)
33. *Depermutation kernel* (execute 15:19)
34. **until** all i iterations are executed
35. **Copy $\mathbf{m}_{v}^{*}(x)$ from BRAMs back to DRAM**

1) THREAD-PER-NODE MULTI-KERNEL DECODER

We employed the multi-kernel approach to design a decoder for the MSA decoding algorithm and used the dataset II to evaluate it. As shown in Table 4, we also instantiated multiple CUs, however, due the higher resource requirements of the design, we were only able to instantiate up to 2 CUs. As noted earlier, the complexity of the design resulted in a reduction in the maximum operating clock frequency from 240MHz to 157MHz. This reduction in the clock frequency hampered the decoding performance of this decoder design.

2) INEFFICIENCY OF THE OpenCL MEMORY MODEL

The OpenCL memory model when mapped to the FPGA uses BRAMs to implement the `local` memory. However, adhering to this standard implies that the same lifetime and scope of the memory must be maintained. This is a significant disadvantage since data that must be kept close to computation in iterative algorithms is required to flow through the `global` addressing space (i.e., off-chip DRAM) twice for every kernel call, ingressing and egressing the computation logic. Therefore, this increases contention to the memory interface and does not take advantage of the fact that data can be stored on-chip and closer to the CUs. Counterintuitively, the HLS tool makes extensive use of the BRAMs the generated design for its internal operation, even though data coherence cannot be guaranteed across multiple iterations.

D. DATAFLOW DECODER

To evaluate the dataflow approach saw a M-modulo decoder based on designs optimized for LDPC-IRA codes [6] and benchmarked using dataset I (c.f. Table 3). This design can scale up or down based on the M FUs that are related to the Tanner graph regularity factor. In the dataset I, the normal frame DVB-S2 codes are expanded by a factor of 360 allowing the number of instantiated FUs to be any sub-multiple of it, $M \in \{2^i \times 3^j \times 5^k\}$, with $0 \leq i \leq 3$, $0 \leq j \leq 2$, $0 \leq k \leq 1$. In the benchmarked design, we studied sub-multiple factorizations of $M \in \{45, 90, 180, 360\}$ FUs, and instantiated multiple decoders based on the logic resources available in the FPGA chip (detailed in Table 2). In fact, a maximum of 720 FUs have been instantiated for up to $k_{\text{dec}} \in \{1, 2, 4, 8\}$ decoders.

As seen in Table 4, a low number of P FUs leads to lower logic utilization and higher clock frequencies of operation. Interestingly, memory elements scale only sub-linearly since the BRAMs units can be shared among the different units. Considering the utilization of LUTs, the main factor limiting the number of instantiated decoders k_{dec}, designs having a utilization greater than 70% were not successfully mapped by the tool chain. Therefore, the decoder with the largest area ($k_{\text{dec}} = 2$, $M = 360$ FUs) leaves 35% of the LUTs unused. However, it is worth noting that the expected operating clock frequency limits the gains attained from defining a higher number of decoder systems. Hence, we can speculate that this trend would still be observed with a higher number of FUs. The increase in FUs usually does not correspond to a gain in decoding throughput. Also, due to the design of M-module architecture, the latency will increase with larger values of k_{dec}.

The MaxCompiler flow requires the developer to introduce a tentative operating clock frequency for which the synthesis and implementation procedures will try to produce a compliant design. Therefore, the results for the decoder solutions presented in Table 4 are shown only for the highest achieved operating clock frequency. Compared to the approach of the wide-pipeline tool flow which delivers the highest clock frequency using different implementation

TABLE 5. Optimization levels of the proposed loop-annotated approach (see also Fig. 7).

Sol.	Description of the solution architecture optimizations
I	Base version without `#pragma HLS` C-directives
II	I + Inner loops c), d) e) and h) are tentatively unrolled completely
III	II + Outer loops a), b), f) and g) are tentatively pipelined to II=1
IV	I + BRAM arrays reshaped by a 2^m factor
V	IV + Inner loops c),d) e) and h) are tentatively unrolled completely
VI	III + IV Unrolling c),d) e) and h), pipelining a), b), f) and g) and reshaping)
VII	IV + Pipelining of inner loops c), d) e) and h) to II=1 and unrolling outer loops a), b), f) and g) by a factor 2^m

strategies, the dataflow approach involves a trial and error process for the user. This implies that tighter design parameters (e.g., high operating frequency and large logic utilization) will increase the time needed for the tool flow and a working design is not guaranteed. Similar to wide-pipeline case, the dataflow decoder is connected by a platform design that is automatically generated by the tool [20].

E. LOOP-ANNOTATED DECODER

As discussed previously, this design benefits from multiple optimizations; therefore, we need to combine different optimizations regarding memory, loop and dataflow to achieve the best results. Table 5 lists the designs that were obtained by applying the optimizations in different combinations [25] and the best design, solution VI, has also been included in Table 4.

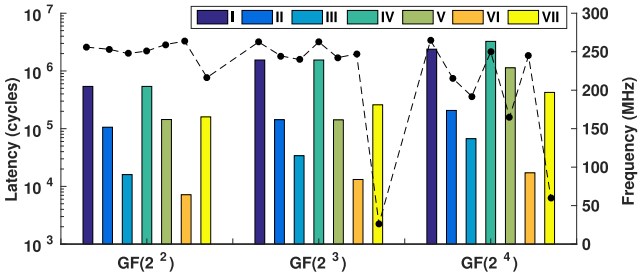


FIGURE 8. Loop-annotated accelerator decoding latency (*bars*, left axis) vs. clock frequency (*points*, right axis) showing the tradeoff between latency and frequency of operation for solutions I–VII.

Figure 8 plots how the different optimization impacts the operating clock frequency and latency of the decoder design using dataset III-a),b),c). As seen in the Figure, the optimizations, applied either by using the appropriate `#pragma HLS`, or by using `TCL` directives, have a profound impact of on the latency of the decoder system. There is a two orders of magnitude difference in latency between solutions I and VI, while there is only a small difference in operating clock frequency for most solutions. The unoptimized decoder in solution I achieves only a modest performance since the decoding operations are performed sequentially. This is mainly because the tool does not automatically apply necessary optimizations to leverage the available parallelism. However, by carefully combining the different loop-level directives supported by the tool, we can drive the HLS tool and produce a design that is capable of much higher performance (e.g., I and VI).

The optimizations at the loop-level should be accompanied by the additional ones at the memory side. First, to improve the effective data bandwidth, data is streamed from DRAM onto op-chip memories (i.e., BRAMs) to keep it close the computation. Despite this optimization, the effective computational throughput will not improve unless the data bandwidth from the BRAMs is also increased. This is accomplished by using the reshaping directive which has been applied in solutions IV and VI. In solution IV, however, this optimization is not applied in conjunction with the streaming optimization (discussed above). Solution VI combines all the loop-level and memory-level optimizations discussed here and, consequently, achieves the best performance. Note that the memory reshaping essentially remaps the memory indexes i enabling the computation to use a two-dimensional addressing space with indexes $(x, y) = (mod(i, 2^m), \lfloor i/2^m \rfloor)$ that can span across multiple memories instead using only one memory. While this optimization can improve the data bandwidth, the additional index computation increases the latency of each memory transaction and significantly impacts non-pipelined designs. However, in a pipelined design (i.e., those using the loop-pipelining directive) the HLS tool is able to efficiently ensure a certain II inside a loop structure. The additional latency to the overall loop design is the number of clock cycles of the II itself.

An interesting issue that arises when combining directives (e.g., unrolling and pipelining of loop) is about the order in which they must be applied. Solutions VI and VII were generated to help answer that question. First, only a single loop structure is generated in the solution VI case, while in VII the number of pipeline loops generated corresponds to the ratio of the loop trip count by the unroll factor. This leads to long running times for the C-synthesis and in the end, to lower clock frequencies and latencies not even at par with solution II, as observed in Figure 8.

1) REPLICATION OF COMPUTE UNITS

As seen in Table 4, the individual decoders have a low logic utilization and our platform design (Figure 4) is able to utilize multiple decoders ($k_{\text{dec}} > 1$) to improve performance. Therefore, after generating a single decoder from the HLS tool (as an *HLS IP core*), we instantiate multiple copies of this decoder while creating the final design. The number of decoders used will depend on the logic resources available on the FPGA and it entails a trial and error process to find the most suitable number. Our observation is that designs will likely fail to meet timing if logic utilization of the LUTs goes beyond 80%. Within this constraint, we were able to create decoding systems with as many as 14 decoders. Using this approach, we were able to set $k_{\text{dec}} \in \{14, 6, 3\}$, respectively for III-a), III-b) and III-c. The designs with multiple decoders achieved higher overall performance, but due to their increased routing complexity, their operating clock frequency reduced by $\{12.4\%, 16.0\%, 6.94\%\}$.

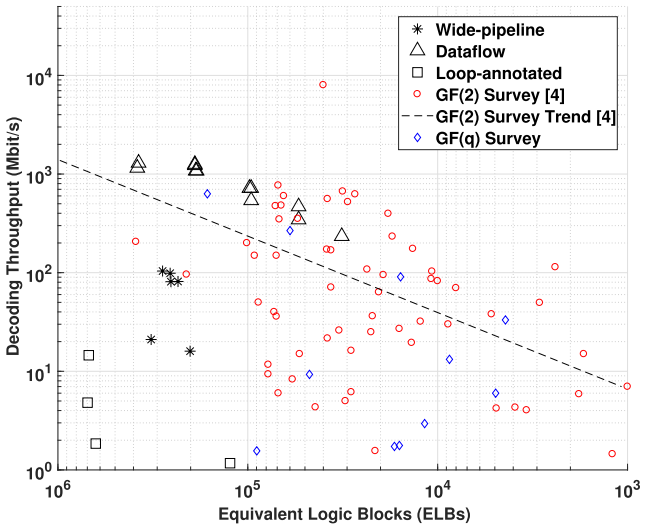


FIGURE 9. Decoding throughput vs. ELBs of the proposed decoders and surveyed RTL-decoders [4].

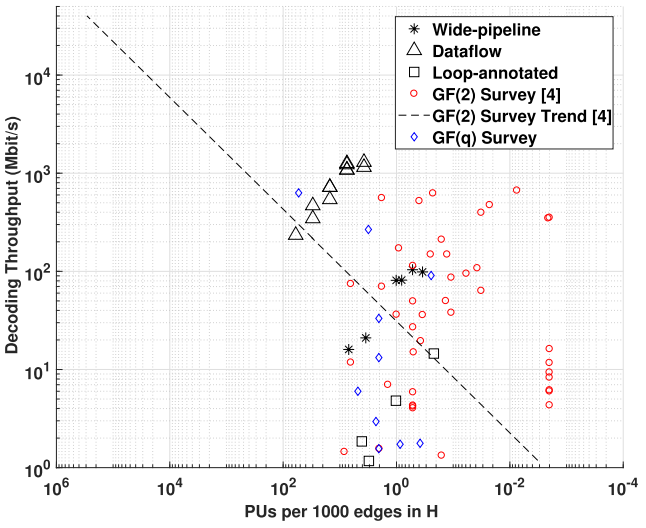


FIGURE 10. Decoding throughput vs. PUs per 1000 edges in **H** of the proposed decoders and surveyed RTL-decoders [4].

VI. RELATED WORK AND DISCUSSION

We compare the proposed decoders with an exaustive survey of RTL-based FPGA LDPC decoders in the literature [4]. To provide a fair comparison across the HLS and RTL implementations, data concerning both logic utilization, number of PUs and the edges in the LDPC code is provided in Figures 9, 10, and 11. Therein, we plot our proposed HLS-based decoders[3] with the RTL-based ones. For other LDPC decoders in the binary domain, we utilized the dataset available in the survey [4], while for the non-binary case, we surveyed a dataset from the literature [27]–[34].

A. THROUGHPUT PER LOGIC UTILIZATION

In Figure 9, we assess the decoding throughput obtained to the number of ELBs required for the LDPC decoders. The dataflow decoders, are able to surpass the trend of the surveyed RTL-based approaches [4], while the wide-pipeline decoders fall behind. Dataflow decoders obtain equivalent decoding throughputs to what is reported in the literature [4], as well as the wide-pipeline ones, the latter, however, at greater logic utilization. On the other hand, non-binary decoders, while on a par in throughput, when compared to the other decoders surveyed, achieve so at a much higher logic utilization levels. Furthermore, the Galois Field dimensions which we were able to synthesize are lower than those compared against. The main motivation for this is the inability of the HLS tool to scale to logic utilization levels which can fit the FPGA architecture.

B. THROUGHPUT PER PUs AND CODE COMPLEXITY

In Figure 10, the throughput to level of parallelism, expressed in the number of PUs per thousand edges in the LDPC code parity-check matrix **H** is plotted. This representations aims at a comparison of the decoding throughput to the LDPC code complexity and the level of parallelism at the same time. As seen, both the dataflow and wide-pipeline approaches fare well with the RTL-based implementations surveyed [4]. Furthermore, the loop-annotated non-binary designs are well within the scattered cloud of the other non-binary designs reported, i.e. while their required logic utilization levels is much higher than RTL-based approaches, their ability to reach within equivalent throughputs (although for lower Galois Field dimensions) is not impaired by the HLS tool.

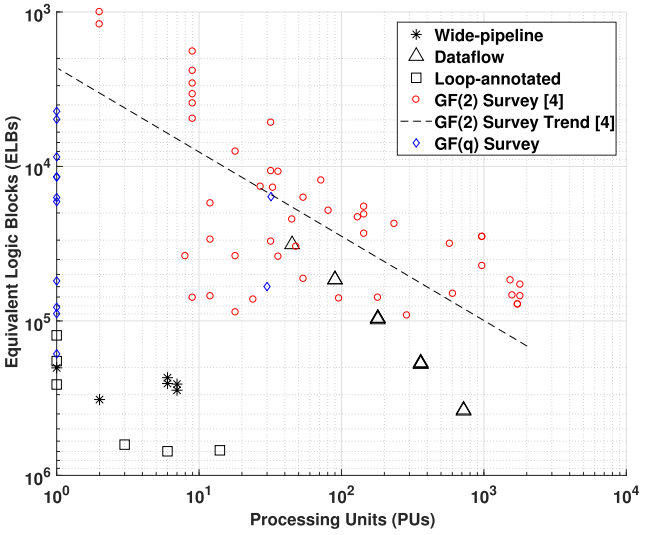


FIGURE 11. ELBs vs. PUs of the proposed decoders and surveyed RTL-decoders [4].

C. ELBs REQUIRED PER PU

In Figure 11, the complexity of each PU is depicted. As expected, the efficiency of HLS approaches generating low logic utilization PUs is lower than that of

[3] In Figs. 9-11 we include the wide-pipeline decoders in [5].

TABLE 6. Summary of features of each HLS approach for LDPC decoding and its advantages and limitations.

	Altera OpenCL	MaxCompiler	Vivado HLS
Program. model and language	· C89-based kernel · C/C++ host program · Stream computing wide-pipeline	· JAVA hardware description · C/C++ for host application · Dataflow computing	· C/C++, SystemC description · Loop-annotated, dataflow, FSM
Platform	· Fixed to 1- or 2-DRAM banks · Not customizable from HLS	· Fixed template · Customizable from HLS	· Platform not generated · Full flexibility for RTL platform
Design Flow	· Offline OpenCL compilation of partial reconfiguration bitstream · P&R performance figures of merit	· Offline MaxCompiler generates partial reconfiguration bitstream · Cycle-accurate behavior of design · P&R performance figures of merit	· Vivado HLS offline generation of *HLS IP core* available to RTL · Cycle-accurate co-simulation
Optim.	· Adjustable no. of CUs k_{CUs} · Adjustable SIMD processing k_{SIMD} · 1- or 2-bank DRAM SODIMMs · Fixed workgroup for improved II	· Parameterizable no. of dec/CUs · Arithmetic floating-/fixed-point based on acceptable error · Easy stream control from DRAM or PCI to the CUs	· Loop unrolling/pipelining · Arithmetic and I/O primitives · BRAM reshaping, partitioning · Latency, data dependencies, and trip count control
Disadv.	· Pipeline flushing penalties · Fixed platform model · Rigid streaming model	· JAVA-based dataflow constructs · Fixed platform model · Dataflow computation model	· Absence of a template requires RTL knowledge to build a fully-functioning platform · Tighter optimizations control behavior at times

RTL-based ones, both for binary decoders and for the non-binary case. These results are hereby explained by a two-fold effect. For the one, HLS approaches cannot minimize the number of ELBs required for a certain logic or arithmetic function, as they are built and developed for general-purpose utilization and are not fined-tuned for the particular tasks required by an LDPC decoder. For the other, the platform generated by the HLS tool has been included in all the ELB logic utilization results aforementioned. This way, a more complex platform than that of RTL designs has been accounted for. For instance, the wide-pipeline decoders require modules for access to a PCIe interface to connect to their host in the OpenCL model, and also DRAM modules for external memory access, which are inexistent, to the best of our knowledge in the majority, if not all, RTL-tuned architectures [4], [27]–[34]. Likewise, the dataflow approach also requires such functionality, while the proposed loop-annotated design requires access to external DRAM. RTL-based designs can have their memory spaces fully implemented in BRAM units and provide a certain pinout interface which does not introduce the ELB overhead of the HLS platforms. It is nevertheless, noteworthy that the dataflow LDPC decoder sees some of its configuration well within the scattered cloud of binary LDPC decoders depicted in Figure 11.

1) HIGH-LEVEL SYNTHESIS STATUS

According to the defined taxonomy, HLS tools are currently in their third generation [35]. Most, are C-based efforts, led by academia and industry. BlueSpec, is SystemVerilog-based tool for both FPGA and *application-specific integrated circuit* (ASIC) design [36], extending the FSM through *guarded atomic actions*. LegUp is a C-based tool, which generates an accelerator system from a C specification, separating data management and control into a MIPS processor from computation that occurs in the circuits [37]. Another academia tool is the ROCCC, providing a C to VHDL compilation tool [38]. Moreover, Cadence C-to-silicon, uses SystemC to raise the abstraction level and introduces *transaction level models*, targeting both FPGA and ASIC accelerators [39]. OpenCL models are also been getting traction from both industry and academia alike, mainly due to the fact that an existing code base can be ported without syntax modifications from a CPU or GPU architecture onto reconfigurable circuits. *Silicon-to-OpenCL* (SOpenCL) is one such tool [19], generating a wide-pipeline custom accelerator for OpenCL kernels. In addition to Altera OpenCL, used in this work, Xilinx also provides similar wide-pipeline concepts for their PCIe-connected FPGAs and FPGA SoC [40]. Furthermore the Vivado HLS suite also accepts C++ and SystemC algorithmic descriptions and, most recently, the introduced HLx suite also aims at allowing the exported *HLS IP cores* to be easily connected on a suitable platform. FCUDA and FASTCUDA generate a custom accelerator from CUDA kernel descriptions [41], [42].

2) PROS AND CONS OF THE STUDIED APPROACHES

Based on the experiments described in this work, we put forward a description of the features and limitations of each tool and its underlying design space exploration in Table 6. The design approach delivering a functional solution in the least development time is based on the wide-pipeline HLS model. However, cross-platform optimization is not granted. To achieve optimal performance a reworking of the OpenCL kernels had to be made. Furthermore, the dataflow approach also offers the designer a ready-made platform, but in this case, the HLS description must follow a dataflow approach. A defined decoder architecture has to be provided in this case to allow for this approach to reach within the

high decoding throughputs realized, one order of magnitude beyond those obtained with the wide-pipeline and loop-annotated designs. Finally, the latter approach allows the highest number of optimizations, with regards to directives that instruct the hardware generation process. Nevertheless, in this case the designer must integrate the decoder into a host platform. The majority of the explored optimizations in the first and second approaches are done so through algorithmic reworking and code refactoring, while the third approach observes limited code refactoring, as the greater share of optimizations is carried out through annotations in the code.

VII. CONCLUSIONS

HLS tools that enable users without hardware design expertise to generate FPGA implementations from high-level language descriptions, e.g., in C, C++, OpenCL and Java. While such approaches produce functionally correct designs, it is of little value unless meet the requirements of target applications. In this paper, we studied how well current generation HLS tools can enable users to perform design space exploration and develop hardware implementations for LDPC decoders. Our experimental results show that decoders generated using HLS, either as wide-pipeline or as dataflow designs, are able to reach within RTL-based ones decoding throughputs, although at greater logic utilization. These results suggest that for SDR system that do not have tight constrains on area and power, HLS-based approaches can reduce the development effort and time needed to develop FPGA implementations. In the future, with the development of languages, optimization and domain-specific solutions, we believe the quality of the HLS tools will improve and they will play a vital role in design of complex communication systems.

REFERENCES

[1] M. Owaida *et al.*, "Enhancing design space exploration by extending CPU/GPU specifications onto FPGAs," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 2, pp. 33:1–33:23, Feb. 2015.

[2] D. Carey, R. Lowdermilk, and M. Spinali, "Testing software defined and cognitive radios using software defined synthetic instruments," *IEEE Trans. Instrum. Meas.*, vol. 18, no. 2, pp. 19–24, Apr. 2015.

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

[4] P. Hailes, L. Xu, R. G. Maunder, B. M. Al-Hashimi, and L. Hanzo, "A survey of FPGA-based LDPC decoders," *IEEE Commun. Surveys Tuts.*, vol. 18, no. 2, pp. 1098–1122, 2nd Quart., 2016.

[5] J. Andrade, G. Falcao, 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.

[6] G. Falcao *et al.*, "Configurable M-factor VLSI DVB-S2 LDPC decoder architecture with optimized memory tiling design," *EURASIP J. Wireless Commun. Netw.*, vol. 2012, p. 98, Mar. 2012.

[7] S. Muller, M. Schreger, M. Kabutz, M. Alles, F. Kienle, and N. Wehn, "A novel LDPC decoder for DVB-S2 IP," in *Proc. IEEE Conf. Des., Autom. Test Eur.*, Apr. 2009, pp. 1308–1313.

[8] C. Marchand, L. Conde-Canencia, and E. Boutillon, "Architecture and finite precision optimization for layered LDPC decoders," in *Proc. IEEE Int. Conf. Appl.-Specific Syst., Archit. Process.*, Oct. 2010, pp. 350–355.

[9] A. Balatsoukas-Stimming and A. Dollas, "FPGA-based design and implementation of a multi-GBPS LDPC decoder," in *Proc. IEEE Int. Conf. Field-Program. Logic Appl.*, Aug. 2012, pp. 262–269.

[10] D. C. Alves, E. De Lima, and J. E. Bertuzzo, "A pipelined semiparallel LDPC Decoder architecture for DVB-S2," in *Proc. 3rd Workshop Circuits Syst. Des. (WCAS)*, 2013, pp. 1–4.

[11] O. Boncalo, A. Amaricai, A. Hera, and V. Savin, "Cost-efficient FPGA layered LDPC decoder with serial AP-LLR processing," in *Proc. IEEE 24th Int. Conf. Field-Programm. Logic Appl.*, Sep. 2014, pp. 1–6.

[12] A. J. Blanksby and C. J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-1/2 low-density parity-check code decoder," *IEEE J. Solid-State Circuits*, vol. 37, no. 3, pp. 404–412, Mar. 2002.

[13] A. M. Wyglinski, D. P. Orofino, M. N. Ettus, and T. W. Rondeau, "Revolutionizing software defined radio: Case studies in hardware, software, and education," *IEEE Commun. Mag.*, vol. 54, no. 1, pp. 68–75, Jan. 2016.

[14] R. A. Carrasco and M. Johnston, *Non-Binary Error Control Coding for Wireless Communication and Data Storage*. Hoboken, NJ, USA: Wiley, 2008.

[15] J. Andrade, G. Falcao, V. Silva, J. P. Barreto, N. Goncalves, and V. Savin, "Near-LSPA performance at MSA complexity," in *Proc. IEEE Int. Conf. Commun.*, Jun. 2013, pp. 3281–3285.

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

[17] J. O. Lacruz *et al.*, "Simplified trellis min–max decoder architecture for nonbinary low-density parity-check codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 9, pp. 1783–1792, Sep. 2015.

[18] G. Wang *et al.*, "Parallel nonbinary LDPC decoding on GPU," in *Proc. IEEE Asilomar Conf. Signals, Syst., Comput.*, Nov. 2012, pp. 1277–1281.

[19] M. Owaida *et al.*, "Massively parallel programming models used as hardware description languages: The OpenCL case," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Des.*, Nov. 2011, pp. 326–333.

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

[21] J. Andrade *et al.*, "Fast design space exploration using Vivado HLS: Non-binary LDPC decoders," in *Proc. IEEE 23rd Annu. Int. Symp. Field-Program. Custom Comput. Mach.*, May 2015, p. 97.

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

[23] 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.

[24] J. Andrade, F. Pratas, G. Falcao, 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 ASAP*, Jun. 2014, pp. 264–269.

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

[26] K. Kasai and K. Sakaniwa, "Fourier domain decoding algorithm of non-binary LDPC codes for parallel implementation," in *Proc. IEEE Int. Conf. Accoustics, Speech Signal Process.*, Prague, Czech Republic, May 2011, pp. 3128–3131.

[27] X. Zhang and F. Cai, "Reduced-complexity decoder architecture for non-binary LDPC codes," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 7, pp. 1229–1238, Jul. 2011.

[28] F. García-Herrero, M. J. Canet, and J. Valls, "High-speed nb-ldpc decoder for wireless applications," in *Proc. IEEE Int. Symp. Intell. Signal Process. Commun. Syst.*, Nov. 2013, pp. 215–220.

[29] J. O. Lacruz, F. García-Herrero, M. J. Canet, J. Valls, and A. Pérez-Pascual, "A 630 Mbps non-binary LDPC decoder for FPGA," in *Proc. IEEE Int. Symp. Circuits Syst.*, May 2015, pp. 1989–1992.

[30] W. Sulek, M. Kucharczyk, and G. Dziwoki, "GF(q) LDPC decoder design for FPGA Implementation," in *Proc. IEEE CCNC*, Jan. 2013, pp. 460–465.

[31] F. Liu and H. Li, "Decoder design for non-binary LDPC codes," in *Proc. IEEE 7th Int. Conf. Wireless Commun., Netw. Mobile Comput.*, Sep. 2011, pp. 1–4.

[32] T. Lehnigk-Emden and N. Wehn, "Complexity evaluation of non-binary Galois field LDPC code decoders," in *Proc. IEEE 6th Int. Symp. Turbo Codes Iterative Inf. Process.*, Sep. 2010, pp. 53–57.

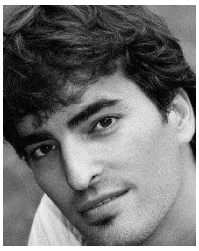
[33] E. Boutillon, L. Conde-Canencia, and A. A. Ghouwayel, "Design of a GF(64)-LDPC Decoder based on the EMS Algorithm," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 60, no. 10, pp. 2644–2656, Oct. 2013.

[34] C. Spagnol, W. Marnane, and E. Popovici, "FPGA implementations of LDPC over GF(2^m) decoders," in *Proc. IEEE Int. Work. Signal Process. Syst.*, Oct. 2007, pp. 273–278.
[35] R. Tessier, K. Pocek, and A. DeHon, "Reconfigurable computing architectures," *Proc. IEEE*, vol. 103, no. 3, pp. 332–354, Mar. 2015.
[36] R. Nikhil, "Bluespec system verilog: Efficient, correct RTL from high level specifications," in *Proc. ACM/IEEE Int. Conf. Formal Methods Models Co-Des.*, Jun. 2004, pp. 69–70.
[37] A. Canis *et al.*, "LegUp: High-level synthesis for FPGA-based processor/accelerator systems," in *Proc. ACM/SIGDA Int. Symp. Field-Programm. Gate Arrays*, 2011, pp. 33–36.
[38] J. Villarreal *et al.*, "Designing modular hardware accelerators in C with ROCCC 2.0," in *Proc. IEEE FCCM*, May 2010, pp. 127–134.
[39] "C-to-silicon compiler high-level synthesis automated high-level synthesis for design and verification," Cadence Des. Syst., Inc., San Jose, CA, USA, White Paper, 2011.
[40] "Xilinx SDAccel, a unified development environment for tommorow's data center," Xilinx, Inc., San Jose, CA, USA, White Paper, 2014.
[41] A. Papakonstantinou *et al.*, "FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs," in *Proc. IEEE Symp. Appl. Specific Process.*, Jul. 2009, pp. 35–42.
[42] I. Mavroidis *et al.*, "FASTCUDA: Open source FPGA accelerator & hardware-software codesign toolset for CUDA kernels," in *Proc. Euromicro Conf. Digit. Syst. Des.*, Sep. 2012, pp. 343–348.

JOAO ANDRADE received the M.Sc. degree in telecommunications and the Ph.D. degree in electrical and computer engineering from the University of Coimbra. From 2010 to 2016, he was a Researcher with the Instituto de Telecomunicações and an Affiliated Member of the HiPEAC. Since 2016, he has been a Research and Development Engineer with Synopsys Porto, Porto, Portugal. His research interests include hardware verification, error resilient architectures, forward error-correction, and reconfigurable computing.

NITHIN GEORGE received the Ph.D. degree in computer science from the École Polytechnique Fédérale de Lausanne in 2016, and the M.Sc. degree in communication engineering from the Technische Universität München in 2009. He is currently a Software Engineer with Intel Technologies. His research interests include high-level synthesis, hardware design targeting FPGAs, and developing domain-specific tools.

KIMON KARRAS received the B.Sc. degree from the Technical Educational Institute of Pireaus, the M.Sc. degree in microelectronics from the University of Athens, and the Ph.D. degree from the Technische Universität München. He was a Research Engineer with Xilinx Research Labs, Ireland. He is currently with Think Silicon S.A., where he is responsible for hardware development. His research interests include high-performance data center platforms, high-level synthesis, and networking for data centers.



DAVID NOVO (M'08) received the M.Sc. degree from the Universitat Autonoma de Barcelona, Spain, in 2005, and the Ph.D. degree in engineering from KU Leuven, Belgium, in 2010. From 2010 to 2016, he was a Post-Doctoral Researcher with the Processor Architecture Laboratory, EPFL, Switzerland, for five years, and with the Adaptive Computing Group, LIRMM, France, for one year. Since 2017, he has been a Tenured Full-Time CNRS Research Scientist with LIRMM. His research interests include hardware and software techniques for increasing computational efficiency in next-generation computers.

FREDERICO PRATAS (S'07–M'13) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Técnico, Universidade de Lisboa, Lisbon, Portugal, in 2012. Until 2013, he was also a Researcher with INESC-ID, Lisbon. In 2013, he was a Researcher Scientist with Intel Labs, Barcelona, involving in the design of future microarchitectures. Since 2014, he has been with the Imagination Technologies' MIPS Group, KL, U.K., where he currently collaborates as a Leading Hardware Design Engineer. His research interests include computer architectures and microarchitectures design and verification, high-performance computing, and reconfigurable computing.

LEONEL SOUSA (M'01–SM'03) received the Ph.D. degree in electrical and computer engineering from the Instituto Superior Tecnico, Universidade de Lisboa, Lisbon, Portugal, in 1996. He is currently a Full Professor with the Universidade de Lisboa. He is also a Senior Researcher with the Instituto de Engenharia de Sistemas e Computadores (INESC-ID). His research interests include parallel computing, VLSI and computer architectures, and computer arithmetic. He is a fellow of the IET and a Distinguished Member of the ACM. He is an Associate Editor of the IEEE TMM, the IEEE TCSVT, and the IEEE Access, and the Editor-in-Chief of the EURASIP JES.

PAOLO IENNE (M'90–SM'10) received the Ph.D. degree in computer science from the École Polytechnique Fédérale de Lausanne. He is currently a Professor with the School of Computer and Communication Sciences, École Polytechnique Fédérale de Lausanne, where he heads the Processor Architecture Laboratory. His research interests include computer and processor architecture, electronic design automation, computer arithmetic, FPGAs and reconfigurable computing, and multiprocessor systems-on-chip. He is an Associate Editor of the ACM CSUR and the ACM TACO.

GABRIEL FALCAO (S'07–M'10–SM'14) received the M.Sc. degree in electrical and computer engineering from the University of Porto and the Ph.D. degree from the University of Coimbra. In 2011 and 2012, he was a Visiting Professor with EPFL, Switzerland. He is currently an Assistant Professor with the University of Coimbra. He is a Researcher with the Instituto de Telecomunicacações. His research interests include parallel computer architectures, GPU- and FPGA-based accelerators, and signal processing. He is a Senior Member of the IEEE, and a member of the IEEE Signal Processing Society and the HiPEAC network of excellence.

VITOR SILVA received the Graduation Diploma in electrical engineering and the Ph.D. degree 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. He is currently the Director of the Instituto de Telecomunicações, Coimbra, coordinating the research activities of 40 collaborators. His research activities include signal processing, image and video compression, and coding theory.