From Low-architectural Expertise Up to High-throughput Non-binary LDPC Decoders: Optimization Guidelines using High-level Synthesis

Joao Andrade*, Nithin George†, Kimon Karras‡, David Novo†, Vitor Silva*, Paolo Ienne† and Gabriel Falcao*
*Instituto de Telecomunicações, Dept. Electrical and Computer Engineering, University of Coimbra, Portugal
†École Polytechnique Fédérale de Lausanne, School of Computer and Communication Sciences, Lausanne, Switzerland
‡Xilinx Research Labs, Dublin, Ireland

***Abstract*—HLS tools have been introduced with the promise of easening and shortening the design cycle of tedious and error-prone RTL-based development of hardware accelerators. However, they do so either by concealing meaningful hardware decisions which model the computing architecture—such as OpenCL compilers—or by abstracting them away into a high-level programming language—usually C-based. In this paper, we show that although Vivado HLS is sufficiently mature to generate a functionally correct FPGA accelerator from a naive description, reaching an accelerator which optimizes the FPGA resource utilization in a way that conveys maximum performance is a process for a hardware architect mindset. We use a highly demanding application, that requires real-time operation, and develop a non-binary LDPC decoder on a state-of-the-art Virtex 7 FPGA, using the Vivado HLS framework. Despite using the same programming syntax as a C-language software compiler, the underlying programming model is not the same, thus, the optimizations required in code refactoring are distinct. Moreover, directive-based optimizations that tweak the synthesized C description hardware must be used in order to attain efficient architectures. These processes are documented in this paper, to guide the reader on how an HLS-based accelerator can be designed, which in our case can come close to the performance achieved with dedicated hand-made RTL descriptions.**

***Keywords*-fast hardware design; high-level synthesis; design space exploration; Vivado HLS; non-binary LDPC codes**

I. Introduction

Field-programmable Gate Arrays (FPGAs) are appearing as main components in the most demanding heterogeneous computer systems. Smartphones, which have to conciliate a very high performance with the high energy efficiency of a portable battery powered device, start to incorporate FPGAs to add versatility to their system on chip platforms [1]. Datacenters are a completely different segment, which has to provide a very high performance while keeping a moderate infrastructural cost. Still, FPGAs have shown to be useful under high load, improving the ranking throughput of each server by a factor of 95% for a fixed latency distribution [2].

However, the strength of FPGA design is also its Achilles heel: the fact that the computer architecture (i.e., number of processing units, interconnect between processing units, memory hierarchy, etc.) is largely unconstrained, enables an efficient tailoring of the architecture to the particular application. This freedom is difficult to maintain when offering a productive programming model [3]. Although current *high-level synthesis (HLS)* flows support a high level application description that can be effectively synthesised in efficient hardware, finding that description is a complex design problem requiring solid hardware understanding. While HLS tools have been proven worthy as testing and validation tools, we find them, backed on the designs herein proposed, sufficiently mature nowadays to synthesize efficient hardware accelerators.

In this paper, we use the implementation of an advanced non-binary *Low-density Parity-check (LDPC)* decoder as a case study to illustrate the design problems that need to be solved in order to achieve an efficient utilization of the reconfigurable substrate. Despite using state-of-the-art HLS tools, such application mapping is tedious and complex, and when used from a software development stance yields inefficient accelerator architectures. We show that in order to obtain an efficient design one should carefully consider (1) the limitations of the target board and device (FPGA) (e.g., bandwidth to external memory, number of DSP units, etc.) and (2) the characteristics of the application (e.g., memory dependencies, parallelization possibilities, operations granularity, scheduling, etc.) to perform targeted optimizations. We show that while a naïve implementation of the LDPC decoder, taken entirely from a pure software description, can only achieve a limited performance, the attentive tuning of the high level application description can unleash performances that are in the same ballpark as those that are painstakingly developed using *Register Transfer Level (RTL)* descriptions. The source code resulting from this work is available to the scientific community for download at https://github.com/andradx/fftspa_fpl2015.

II. Non-binary LDPC codes

LDPC codes are a class of capacity-approaching linear block codes defined by sparse parity-check matrices [4]. They are widely employed for digital communication and storage systems [4]. Typically, LDPC codes are binary but their performance improves with the code length, posing numerous challenges to the design of low latency decoder architectures. On the other hand, non-binary LDPC codes, defined over the

*Authors acknowledge Instituto de Telecomunicações and the Portuguese FCT under grants UID/EEA/50008/2013 and SFRH/BD/78238/2011

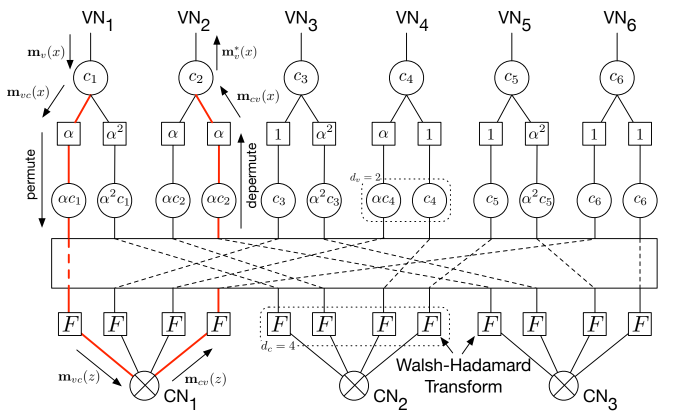


Fig. 1. LDPC code Tanner graph representation of (1) and message-passing decoding. $\mathbf{m}$ messages are exchanged between CNs and VNs. Whenever they traverse a square or a circle, a certain function is applied to it. In the FFT-SPA case, Hamadard products are applied both as CN and VN update rules, due to the introduction of the Walsh-Hadamard Transform (F) prior to the CN update. Furthermore, *pmf*s are permuted due to the non-binary field [5].

binary extension field (GF(2^m)), show a better performance for shorter lengths [5].

The parity-check matrix $\mathbf{H}$ with dimensions $M \times N$ describes an LDPC code composed by N symbols with rate $R = \frac{N-M}{N}$, i.e., M symbols represent redundancy and $N-M$ are information. The Tanner graph of parity-check matrix

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

of the code in [5] is exemplified in Fig. 1, and represents parity-check equations (rows in $\mathbf{H}$) as *Check Nodes (CNs)* and symbols (columns in $\mathbf{H}$) as *Variable Nodes (VNs)*, where each non-null element h_{cv} defines an edge connecting CN$_c$–VN$_v$. The set of CNs connected to VN$_v$ and of VNs connected to CN$_c$ are named as $C(v)$ and $V(c)$, with cardinalities d_c and d_v. In the non-binary case, each edge is associated with a non-null element, which can be expressed as a power of the primitive element of GF(2^m) [5]. Herein, $2^m{=}q$ and, thus, GF(2^m)$\equiv$GF(q).

A. Decoding Algorithm Overview

As with binary LDPC codes, non-binary codes can be decoded by the *Sum-Product Algorithm (SPA)*, although its prohibitive numerical complexity makes it necessary to use lower complexity alternatives at the cost of sub-optimality and/or by processing in the Fourier-domain [5]. The latter is particularly advantageous since no sub-optimality is necessarily introduced to the decoding procedure. Therefore, in this work, we target the *Fast Fourier Transform Sum-Product Algorithm (FFT-SPA)* that offers excellent decoding performance at lower computational complexity [5]. The FFT-SPA, formalized in Algorithm 1, consists of the following processing phases:

- at the receiving end of the transmission channel *probability mass functions (pmfs)* $\mathbf{m}_v(x)$ are computed for each VN symbol in GF(2^m) and for each $\mathbf{m}_{vc}(x) = \mathbf{m}_{vc}\left[0, 1, \alpha, \cdots, \alpha^{2^m-2}\right]$ (1), with α the primitive element of the polynomial generating GF(2^m) [5];
- *pmf*s $\mathbf{m}_{vc}(x)$ are permuted to $\mathbf{m}_{vc}(x \times h_{cv})$;
- CN processing, where the CNs process the $\mathbf{m}_{vc}(x)^{(i-1)}$ to compute $\mathbf{m}_{cv}(x)^{(i)}$ (4)-(5), with (i) the i-th iteration;
- *pmf*s $\mathbf{m}_{cv}(x \times h_{cv})$ are depermuted to $\mathbf{m}_{cv}(x \times h_{cv} \times h_{cv}^{-1}) {=} \mathbf{m}_{cv}(x)$;
- the VN processing, where the VN nodes receive their adjacent $\mathbf{m}_{cv}(x)^{(i)}$ and compute compute the *a-posteriori* reliability of VN$_v$, $\mathbf{m}_v^{*(i)}(x)$ (2), from which $\mathbf{m}_{vc}(x)^{(i)}$ (3) can be obtained;
- from each $\mathbf{m}_v^{*(i)}(x)$ the symbol of VN$_v$ may be retrieved;
- the decoder iterates between (2)-(5) until a maximum number of iterations has been reached.

Algorithm 1 GF(2^m) FFT-SPA decoding algorithm.

```
1: Initialize and store data to DRAM:
       $\mathbf{m}_v(x) = \mathbf{m}_{vc}^{(0)}(x) = p(c{=}x|y_v)$
2: Prologue: Fetch channel data from DRAM—(Start FFT-SPA Decoding)
3: while max iterations not executed do
4:     Depermute the pmfs and apply the VN rules (depermute and vn_proc):
           $\mathbf{m}_v^{*(i)}(x) = \mathbf{m}_v(x) \prod_{c' \in C(v)} \mathbf{m}_{c'v}^{(i)}(x \times h_{c'v} \times h_{c'v}^{-1}),$   (2)
           $\mathbf{m}_{vc}^{(i)}(x) = \mathbf{m}_v^{*(i)}(x) / \mathbf{m}_{cv}^{(i)}(x)$   (3)
5:     Apply the CN rules and permute pmfs (cn_proc and permute):
           $\mathbf{m}_{cv}^{(i)}(z) = \prod_{v' \in V_c \backslash v} \mathrm{FWHT}\{\mathbf{m}_{v'c}^{(i-1)}(x \times h_{cv'})\}$   (4)
           $\mathbf{m}_{cv}^{(i)}(h_{cv} \times x) = \mathrm{FWHT}\left\{\mathbf{m}_{cv}^{(i)}(z) / \mathbf{m}_{cv}^{(i)}(z=0)\right\}$   (5)
6: end while                                          —(End FFT-SPA Decoding)
7: Epilogue: Store decoded data to DRAM
```

Note: The FFT can be efficiently computed by the *Fast Walsh-Hadamard Transform (FWHT)* when elements are defined over GF(2^m) [5].

III. LDPC Decoding Hardware

Non-binary LDPC decoding is a complex signal processing application that offers plenty of parallelism for FPGA-based acceleration. However, while targeting a specific application, one needs to tune the architecture based on the specific application requirements and coding parameters resulting thereof. Additionally, to optimally implement this on a given FPGA device, in addition to the necessary algorithmic validation, one needs to perform detailed design space exploration. Hence, developing an FPGA accelerator using the traditional RTL development approach would incur high development time, greatly limiting the design space that can be explored, and thereby lead to high *Non-Recurring Engineering (NRE)*. Recently, HLS tools have greatly improved and are able to provide a convenient environment to speed up the development process and empower the developers to quickly explore a much wider design space. Xilinx Vivado HLS [6] is a state-of-the-art

HLS tool that accepts design inputs in C, C++ or SystemC and enables the designer to quickly perform algorithm verification before automatically translating the high-level design input into an RTL implementation. More importantly, it also permits the designer to use compiler directives, when necessary, to guide the tool to explore some architectural options. Due to this high degree of control on the underlying architecture and the low effort of retargeting different devices, we used Vivado HLS to develop the non-binary LDPC decoder.

A. High-Level System Architecture

Vivado HLS can be used to develop *HLS IP cores* that implement the LDPC decoding functionality. However, to realize the FPGA accelerator, one still needs to integrate these cores into a high-level architecture that contains essential system components, such as the interconnection bus and external memory controller, and the clock and control circuitry. This is needed because the storage space required for larger values of m can exceed the local storage capacity of typical FPGAs.

In order to develop a high-performance decoder, we need a high-bandwidth access to the data. Since our target platform, the Xilinx VC709 board, has two DRAM SODIMMs, we have two DRAM controllers in the system. In order to leverage this bandwidth from the *HLS IP cores*, we store the input data, comprising of $\mathbf{m_{cv}}(x)$, $\mathbf{m_{vc}}(x)$ and $\mathbf{m_v}(x)$, in one DRAM and the output, $\mathbf{m_v^*}(x)$, in the other.

While this design choice provides high data bandwidth to the computation cores, the performance of the cores will still suffer due to the DRAM memory high response latency. To overcome this, we have also provision local storage. Prior to starting each decoding phase, the data is copied to the local storage using a burst transfer. The core then executes the FFT-SPA using the data from the local storage, which offers minimum access latency. Once the computation is completed, the decoded data is copied, again using a burst transfer, into the DRAM. This implies that the DRAM controller is only utilized intermittently during the prologue and epilogue of the actual decoding procedure, during the burst transfers to and from the HLS core. We can hence add multiple cores, that can access the DRAM in a staggered fashion to provide higher throughputs. Naturally, when the number of cores is high, there will not be enough bandwidth to prevent the cores from starving for data, although the utilization of the FPGA logic elements by each individual core is too high to ever see this happen for the tested cases.

IV. HLS-GENERATED ARCHITECTURE

The design of the GF(2^m) LDPC FFT-SPA decoder space exploration and corresponding optimizations performed are discussed next. The latter come in two ways, either by proper code writing, and in the cases where an existing C application is ported code refactoring is a must, or by annotating the code with optimization directives. These directives can be inlined with C code using the `#pragma` construct, or systematized in a `Tcl` script with equivalent syntax, both of which are mutually exclusive methods to instruct the HLS tool.

A. Mapping the FFT-SPA to HLS C

From equations (1)–(5), in Algorithm 1, we can observe that each expression is applied to certain subsets of data, $C(v)$ or $V(c)$ within larger sets $\mathbf{m}_{cv}(x)$ or $\mathbf{m}_{vc}(x)$, so they represent computations inside a loop structure in our C-design input. Essentially, a non-binary LDPC code can be described in three fundamental dimensions: (1) how many *pmf*s or edges need to be updated; (2) how many elements compose the *pmf*, i.e., the GF(2^m) dimension; and (3) how many *pmf*s participate in the update of an edge, i.e., d_c and d_v, respectively for CN (4),(5) and VN (3) processing phases of the FFT-SPA.

For a *Single Instruction Multiple Thread (SIMT)*-like architecture [7], expressing the non-binary LDPC code dimensions can be efficiently performed by linearizing all dimensions to a single one in a thread execution grid [8]. Translating this to a single iterator variable code whose trip count is enough to cover all three dimensions is straightforward, and the loop structure in the top of Fig. 2 wil generate a bit-true C testbench in Vivado HLS. However, optimizations instructed in this fashion are not picked up easily by the compiler when we attempt to explore loop unrolling or pipelining in a particular dimension. In fact, most times this will draw a too long compile time and inefficient resource utilization by the C-synthesized accelerator. The code transformation show in Fig. 2 is required for each decoding algorithm structure that could be defined by a loop. Hence, in order to enable

```
//flat loop unsuitable for Vivado HLS optimizations
for(int i = 0; i < edges*q; i++){
  int e = i/(d_v*q);          //get VN id
  int g = i%q;                //get GF(q) element
  int t = (i/q)%d_v;          //get d_v element
}

//nested loop suitable for Vivado HLS optimizations
for(int e = 0; e < edges; e++)
  for(int g = 0; g < q; g++)
    for(int t = 0; t < d_v; t++)
```

Fig. 2. Loop structures suitable and unsuitable for Vivado HLS optimizations.

our subsequent optimizations, we express computation within loop-nests constituting of loops labelled (1) **E**, (2) **GF**, (3) **LogGF**, (4) $\mathbf{D_v}$ and (5) $\mathbf{D_c}$ that iterate over the dimensions, (1) number of edges in the code, (2) Galois Field dimension $q{=}2^m$, (3) Galois Field order $m{=}\log_2 q$, (4) d_v and (5) d_c. Also, this structure exposes the parallelism in the application in a form that is more amenable for easy analysis and subsequent optimization, but more importantly, as previously mentioned, it becomes easier for the HLS tool to leverage such parallelism in order to produce better results. The architecture resulting from such a code structuring is shown in Fig. 3, herein designated as Solution I, and is the starting point for further design exploration.

*1) **vn_proc** and **cn_proc**:* These kernels perform Hadamard multiplications. In Solution I, as seen in Fig. 3, each kernel is composed of a triple-nested loop structure **E–GF–**$\mathbf{D_v}$**/**$\mathbf{D_c}$. In the **E** loop it computes over different messages, with trip count $edges{=}N{\times}d_v$. In the **GF** loop it operates over different

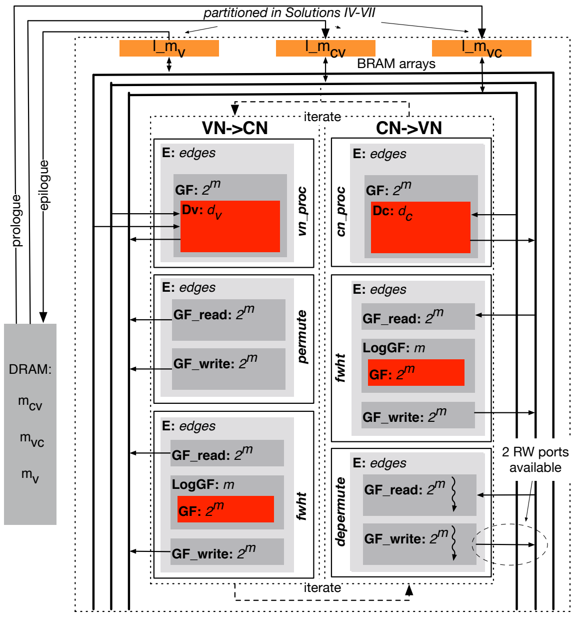


Fig. 3. LDPC decoder architecture base version (Solution I). Each kernel is defined as a triple- or double-nested loop body function—loop trip counts are shown after the *colon*—consuming and producing data to BRAM defined arrays. As the tool default behaviour does not optimize the BRAM array physical structure, typically only two ports are exposed per module utilized, and, thus, data requests are served at the rate of two elements per clock cycle.

probability values (*pmf*s) and in $\mathbf{D_v/D_c}$ it uses data read from different arrays. Hence, the optimization for these kernels will be to leverage the parallelism from all three loop bodies (Fig. 4).

```
//nested loop loop structure of vn_proc
E:for(int e = 0; e < edges; e++){
  GF:for(int g = 0; g < q; g++){
    Dv:for(int t = 0; t < d_v; t++){
      //computation follows
}}}
```

Fig. 4. Nested loop structure of **vn_proc** and **cn_proc** (replace $d_v \leftarrow d_c$).

*2) **permute** and **depermute**:* These kernels deal with the permutation and depermutation of probabilities inside each *pmf* and are described as a double nested loop structure, shown in Fig. 3. In the **GF_read** loop, it loads data and shuffles it according to the permutation/depermutation into a local copy. Then, loop **GF_write** writes data back contiguously to the correct BRAM location. Since the shuffling is performed in-place, as shown in Fig. 5, the available parallelization potential is limited and is, in fact, related to m.

```
E:for(int e = 0; e < limit; e++){
 GF_read:for(int g = 0; g < GF; g++)
    //load data into temporary buffer
  GF_write:for(int g = 0; g < GF; g++)
    //permute and store back to memory
}
```

Fig. 5. Nested loop structured of **permute** and **depermute**.

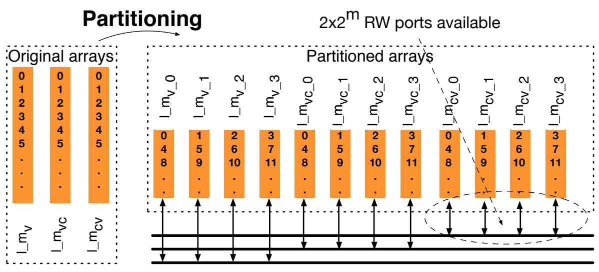


Fig. 6. Partitioning of the BRAM-defined arrays, by a cyclic factor of 4 (a factor of 2^m was applied for each GF(2^m) solution) exposing 4 ports with double read-write capacity per clock cycle. The cyclic factor spreads contiguous data elements across different BRAMs, therefore exposing parallel accesses to those elements in the 2^m dimension (**GF** loops).

*3) **fwht**:* This kernel implements the FWHT, a special case of the FFT where the twiddle factors are always -1 or 1, thus only additions and subtractions are executed by this kernel. In the loop nest shown in Fig. 3, **E** iterates over all the *pmf*s whose transforms are computed. Since performing the radix-2 factorization of the FWHT in-place would entail tremendous pressure accessing BRAMs, we utilize a temporary array—a scratchpad—to hold the working data. Loops **GF_read** and **GF_write** copy the data to and from this scratchpad. The **LogGF** loop cycles through the FWHT stages and **GF** iterates over each transform element. Here, we achieve parallelism among the different messages—transform batches—in **E** and also **GF**, among different message elements, that can be exploited during optimizations.

```
E:for(int e = 0; e < edges; e++){
  G_read:for(int g = 0; g < q; g++){
    //load data into temporary array
  }
  LogGF:for(int c=0;c<m;c++){
    GF:for(int g = 0; g < q; g++){
      //perform Radix-2 computation
  }}
  G_write:for(int g = 0; g < q; g++){
    //store data back to memory
}}
```

Fig. 7. Nested loop structure of **fwht**.

B. Architecture Optimization Guidelines

The unoptimized decoder in Solution I achieves only a modest performance since the decoding operations are performed sequentially. This is because the tool does not automatically apply necessary optimizations to leverage the available parallelism. Hence, to achieve high performance, one needs to explicitly direct the tool to apply the necessary optimizations. Moreover, one needs to carefully consider the hardware implications of the specific optimization and often apply one or more optimization together in order to achieve the intended result. The optimizations carried out, and described next, can be visualized in Figure 8.

1) Loop unrolling: Loop unrolling tries to schedule multiple iterations in parallel to leverage parallelism and improve

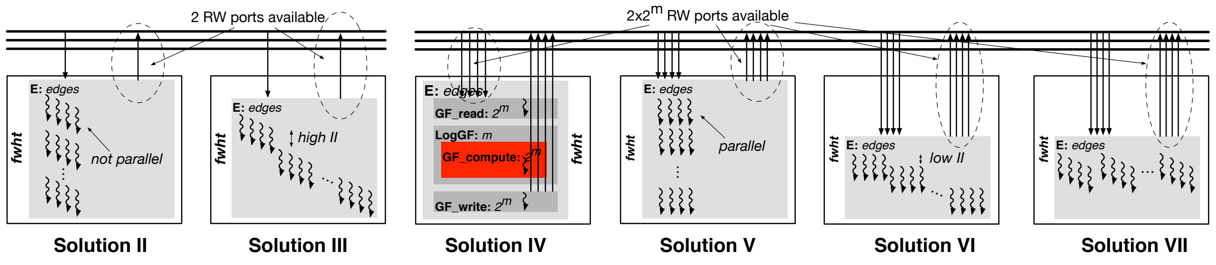


Fig. 8. Solutions II–VII and expected behaviour of the iteration scheduling of the FWHT kernel. Solutions II, III, V and VI have all the inner loops of **E** unrolled, and loop **E** pipelined, respectively. Solutions II–III access BRAM arrays through a double Read/Write (RW) port per array, while Solutions IV–VI access them via 2^m double RW ports. As a consequence, true parallel execution of unrolled iterations and minimum *IIs* for pipelined execution is only achieved through the higher bandwidth exposed by 2^m double RW ports of Solutions V and VI. Solution IV is an example of how higher bandwith available without scheduling optimizations yields no improvement of performance and contributes only to a lower efficiency of design.

processing throughput. In our unoptimized decoder, as aforementioned, we have data parallelism in the **E**, **GF**, $\mathbf{D_v}$ and $\mathbf{D_c}$ loops. Additionally, we can also unroll the **LogGF** loop to remove the control flow overhead associated with the loop structure, which can be useful when m is small. However, while enough logic resources exist on the FPGA to schedule the operations within each loop in parallel, the limiting factor is the number of memory ports available to BRAM memory.

2) Loop pipelining: Loop pipelining tries to improve loop execution performance by having multiple loop iterations execute on the same hardware. *Initiation Interval (II)* is a metric that signifies how soon the loop structure can initiate the execution of a new iteration after having begun the execution of the previous one, (*II*=1 in the optimal situation). In our decoder, we utilize pipelining to exploit the parallelism that has remained untapped in the **E** loop, when its innermost loops are unrolled, or to exploit parallelism of the innermost loops. As with loop unrolling, the *II* resulting of pipelining the loops is limited by the number of memory ports that serve the BRAM-allocated arrays from which data is fetched. Unlike loop unrolling, pipelining in Vivado HLS is an optimization that constrains all inner loops to be unrolled prior to pipelining.

3) Loop unrolling before or after pipelining?: A question that remains is what is the most efficient way to combine unrolling and pipelining. Given the nested-loop structure of the LDPC decoder, educated assumptions can be made regarding the best approach. For a decoder whose innermost loops are pipelined, unrolling of the outermost loop will generate an accelerator composed of small pipelined cores. In its turn, this creates extra overhead due to several units managing their own pipelines. While this is not directly a limiting factor to the performance obtained, higher utilization of resources by control units will reduce the slack for better routing and higher clock frequencies. Whereas pipelining of the outermost loop, with the innermost loops unrolled, will generate a single core per decoding kernel which aggregates control logic in a single pipeline. This way, the FPGA logic resources are devoted in a larger fraction to arithmetic and exploitation of parallel instructions and less to logic control. This comes with increased slack in terms of routing and clock and gives margin to replicate more blocks of the LDPC decoder as explained next. Given that Vivado HLS optimizations are directives and the decoding kernels loop-nested structures, testing between both cases is a simple matter of interchanging the unroll and the pipeline directives, as shown in Fig. 9 for the **vn_proc**.

```
#a) pipeline outermost and unroll innermost
set_directive_pipeline "vn_proc/E" -II 1 -rewind
set_directive_unroll   "vn_proc/GG"
set_directive_unroll   "vn_proc/Dv"
#b) unroll outermost and pipeline innermost
set_directive_unroll   "vn_proc/E" -factor U
set_directive_pipeline "vn_proc/GF" -II 1 -rewind
```

Fig. 9. Pipeline and unroll optimizations `Tcl` directives for **vn_ proc**. In a), complete unrolling is instructed and pipeline *II* is tentatively set at 1. In b), unrolling by a U factor is instructed, and pipeline *II* is tentatively set at 1.

4) Array Partitioning: In order to benefit from the unrolling and pipelining optimizations, we must provide sufficient bandwidth to the design. However, the default strategy of the tool is to sequentially allocate all the data elements into a BRAM unit until a new one is needed. This implies that contiguous data accesses often need to be served by the same BRAM which has only a limited bandwidth from the single or, sometimes, double read-write (RW) port. To alleviate this issue, we instruct Vivado HLS to instantiate dual-ported BRAM memories and, additionally, to partition each BRAM array with a 2^m cyclic factor to expose $2{\times}2^m$ ports per data array. It partitions each array into 2^m new ones, where contiguous elements of the original one, are spread across the multiple BRAMs as seen in Fig. 6. This partitioning enables us to achieve an *II*=1 for the most complex loops in the design. While array partitioning is useful, it comes with the overhead of computing the indices where an index i in the original array must be mapped to a 2-D address $(x,y){=}(mod(i,2^m),\lfloor i/2^m \rfloor)$, with x the BRAM bank and y the index of i in x bank. Nevertheless, it is also a directive optimization that recomputes the indices automatically for the developer and breaks the array into several BRAM banks, as seen in Fig. 10.

5) Floating- vs fixed-point: In FPGA design, we are not constrained by micro-architecture defined data types, such as single-precision floating-point, and can configure the data-

```
#store l_mcv of top-level fftspa in 2-port BRAM
set_directive_resource -core RAM_T2P_BRAM
   "fftspa" l_mcv
#partition array l_mcv cyclically by a factor of 4
set_directive_array_partition -type cyclic
   -factor 4 -dim 1 "fftspa" l_mcv
```

Fig. 10. `Tcl` directives that assign array `l_mcv` to be stored in BRAM units with two R/W memory ports and cyclically partition it across its first dimension by a factor of four.

path to use the most convenient one given the application requirements. For our application, we have found that $Q8.7$ fixed-point representation used for the messages exchanged in the FFT-SPA, with the intermediate operations performed in $Q16.13$, lead to simpler synthesized circuits and reduced latency of fixed-point arithmetic—$QX.Y$ standing for $X-Y$ sign and magnitude bits, and Y for decimal bits. This is a design optimization that must be done after carefully considering the characteristics of the specific application, but whose code refactoring can be performed with the inclusion of the `ap_cint.h` library and the `typedef` definition (Fig. 11).

```
#include<ap_cint.h>
//data is stored in llr type variables
//computation is performed in llr_ type variables
//use floating-point
typedef float llr;
typedef float llr_;
//use Q8.7 fixed-point
typedef ap_fixed< 8, 1, AP_RND_INF, SC_SAT > llr;
typedef ap_fixed< 16, 3, AP_RND_INF, SC_SAT > llr_;
```

Fig. 11. Code refactoring performed for synthesis of the decode design with fixed-point (`llr` with $Q8.7$ and `llr_` with $Q16.13$).

V. Evaluation

We used Vivado HLS 2014.2 to develop our LDPC decoder and targeted a Xilinx VC709 development board with a Virtex-7 XC7VX690T device. To understand the benefit of our optimizations, we evaluated the decoder at different design points, Solutions I-VII in Table I, using a rate $1/3$ $N{=}384$ LDPC code with $d_c{=}3$ and $d_v{=}2$. Additionally, we also varied the field order $m{=}\{2,3,4\}$ to study its impact on the design.

TABLE I
Solutions tested and corresponding optimizations.

Solution	Description of the solution architecture optimizations
I	Base version without C-directives
II	I + Full unrolling of inner loops **LogGF**and **GF**
III	II + Pipelining of outer loops E to II=1
IV	I + Cyclic partition of all BRAM arrays by a factor of 2^m
V	IV + Full unrolling of inner loops **LogGF** and **GF**
VI	III + IV (Unrolling, pipelining and partitioning)
VII	IV + Pipelining of inner loops **LogGF** and **GF** to II=1 and unrolling of outer loop **E** by a factor $U{=}2^m$

A. Methodology

During the decoder development, the C-synthesis provides preliminary results to drive the design space exploration. The functional correctness of this synthesized design is then ascertained through RTL co-simulation, which provides a fairly accurate estimate of the overall decoding performance in clock cycles. Finally, before integrating the decoder into the high-level system architecture, we *place and route* (P&R) the decoder design standalone to obtain a more accurate values for the hardware utilization and clock frequency. This enables us to estimate how many decoders can be instantiated in the high-level system architecture. Now, after performing P&R on this complete system, we compute the decoding throughput for 10 decoding iterations from the post-P&R clock frequency of this system, the number of decoders instantiated and the decoding latency based on the co-simulation.

B. Experimental results

To gain insight on the efficiency of each solution, we analyze the overall decoding latency obtained in C-synthesis and co-simulation as well as the clock frequencies of operation obtained by the C-synthesis and place and route of the *HLS IP core*. This is shown in Tables II and III.

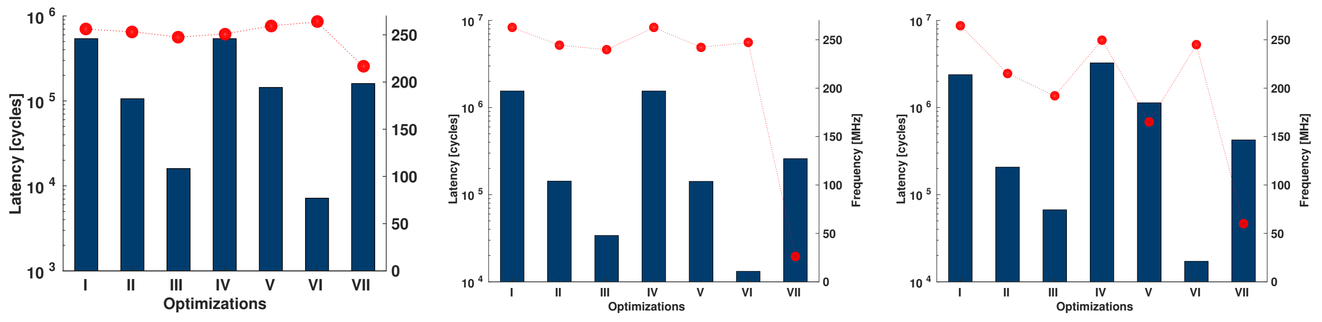
TABLE II
FPGA utilization for the standalone LDPC decoder IP core.

FPGA	$\mathrm{GF}(2^2)$							$\mathrm{GF}(2^3)$						
Util.[%]	I	II	III	IV	V	VI	VII	I	II	III	IV	V	VI	VII
LUTs	0.76	1.48	2.98	1.07	1.52	4.62	7.67	0.64	2.17	5.04	1.13	5.20	10.4	37.4
FF	0.34	0.71	1.42	0.48	0.81	1.89	2.73	0.28	1.16	2.48	0.53	2.52	3.94	7.69
DSP	0.06	0.44	0.44	0.14	0.44	0.44	0.33	0.06	0.89	0.89	0.06	0.89	0.89	0.66
BRAM	0.31	0.24	0.24	0.41	0.41	0.41	0.41	0.44	0.48	0.68	0.82	0.82	0.82	0.85
	$\mathrm{GF}(2^4)$							$\mathrm{GF}(2^3)$ (floating-point)						
LUTs	0.84	3.95	9.80	1.70	10.4	13.5	41.5	0.65	N/A	N/A	1.48	11.7	17.3	N/A
FF	0.42	2.25	4.86	0.97	4.95	5.14	12.9	0.29	N/A	N/A	0.51	3.30	7.56	N/A
DSP	0.44	1.78	1.78	0.22	2.00	1.11	1.33	0.06	N/A	N/A	0.06	1.78	1.78	N/A
BRAM	1.36	0.85	1.09	1.36	1.36	1.36	2.78	0.78	N/A	N/A	1.63	2.72	1.63	N/A

1) Base version: The LDPC decoder base version provided by Solution I exploits no parallelism and, therefore, has low resource utilization and achieves a very modest throughput, well within the Kbit/s range. Moreover, the number of clock cycles taken by this design roughly doubles for each increment of m. This version was used for algorithmic validation and served as a baseline to evaluate the other design optimizations.

2) Loop unrolling: Solutions II, V, VI and VII are the cases that employ loop unrolling, which leads to a reduction in the overall latency of the decoder design, independent of any other optimizations carried out, as seen in Table III. Naturally, unrolling is best applied in conjunction with other optimizations. Both II and V have limited potential to reduce the decoder latency, as these solutions only expose parallelism to the inner loops. Solution II, counterintuitively, has lower decoding latency than V due to the fact that the re-indexing caused by the cyclic partitioning interferes with the in-place permutations carried out by the ***permute*** and ***depermute*** kernels.

Pipelining the computation produces designs that achieve the lowest latency—Solutions III, VI—and also Solution VII. Naturally, pipelining also increases the resource utilization of the FPGA. Among these two first design points utilizing pipelining, Solution VI achieves better performance since the array-partitioning exposes additional memory ports to serve



a) GF(2^2)   b) GF(2^3)   c) GF(2^4)

Fig. 12. Decoding kernels latency (bars, left axis), and clock frequency of operation (points, right axis) of each solution. Compared to the base decoder (I), unrolling (II) and pipelining (III) allow for ten-fold improvements in decoder latency, despite being constrained by the BRAM available bandwidth. Improving the bandwidth by partition of the arrays, exposes more BRAM ports, but without parallel scheduling of operations latency actually increases (IV) and unrolling (V) is not sufficient to mask the indexing overhead. Only the sensible combination of the former is able to minimize latency (VI). Furthermore, it is shown that pipelining the outermost loop provides a more efficient design than to unroll a structure containing pipelined loops (VII).

the iterations inside loop **E**. After partitioning the arrays by a 2^m-factor, ***vn_proc*** and ***cn_proc*** can be optimized to 1 and 2 cycles of *II* and the ***fwht*** can achieve an *II* of 1 cycle. The ***permute*** and ***depermute*** kernels have data dependencies that cause its minimum *II* to grow with the field dimension. An interesting observation in Table III is that the clock frequency estimates for the most complex Solution after C-synthesis can turn out to be, in some cases, grossly estimated, than that obtained after P&R, a potential pitfall in relying on only C-synthesis estimates for evaluation.

TABLE III
LDPC *HLS IP Core* DECODER LATENCY AND CLOCK FREQUENCY.

Sol.	C-Synth'd Design		RTL Co-Sim.	*HLS IP core* P&R'd
	E Lat. [Kcycles]	Clk [MHz]	Lat. [Kcycles]	Clk [MHz]
	GF$\left(2^2\right)$			
I	540	266	607	263
II	121	266	129	253
III	20	266	28	248
IV	508	269	691	251
V	157	266	165	259
VI	8	117	16	264
VII	180	29	187	216
	GF$\left(2^3\right)$			
I	3096	266	1232	262
II	182	266	198	244
III	42	266	58	239
IV	3096	266	1581	263
V	284	266	226	242
VI	26	57	30	247
VII	299	19	315	26
	GF$\left(2^4\right)$			
I	4768	266	2651	265
II	285	266	315	215
III	83	266	114	192
IV	6512	266	3499	249
V	2268	266	1355	165
VI	34	25	50	244
VII	526	117	558	127

3) Loop pipelining: Solutions VI and VII, that combine unrolling with pipelining must be analyzed in their own light. As the outermost loops **E** are prevented to be fully unrolled by the tool, which imposes a limit to the trip count of loops to be unrolled, only low unroll factors could be set ($U{=}2^m$), in order for the tool to synthesize the decoder accelerator in a reasonable timespan. Furthermore, as seen in Fig. 13, Solution VII is a non-optimal Pareto point, using a high number of *Lookup Tables (LUTs)* for a low reduction in decoder latency.

4) Array partitioning: Array partitioning increases the data bandwidth to the computation units by exposing more data that can be consumed in parallel. However, it also comes with a non-negligible cost of re-indexing that consumes resources and increases latency. This effect is visible in Solutions I-IV, and a more pronounced effect in Solutions II-V because of the unrolling that was previously applied, as seen in Figure 13. However, when this is applied in conjunction with pipelining, we obtain a 42–75% reduction in latency, as seen while moving from Solution III to VI, but not for the Solution VII.

5) Replication of Compute Units: As seen in Table II, our individual decoding units are fairly small. Moreover, as discussed in Section III-A, our high-level system architecture facilitates using multiple decoders to achieve higher throughput. Therefore, in our final design, we utilize multiple HLS IP cores to develop a design that targets an FPGA LUT utilization of 80%. We depend on the resource utilization estimates produced from applying P&R on the standalone *HLS IP core* to guide this step. Using this approach, we were able to instantiate $K{=}\{14, 6, 3\}$ decoders for $m{=}\{2, 3, 4\}$. In this multi-decoder design, due to the large design size, we observed a drop in post-P&R clock frequency of operation—dropping by $\{12.4\%, 16.0\%, 6.94\%\}$—compared to the a single decoder design. But, this is well compensated by the improvement in decoding throughput due to the multiple kernels.

VI. Related Work

A handful of publications address the complex design space exploration of non-binary LDPC code architectures on FPGAs. The use of HLS is uncommon for the case of LDPC codes and other signal processing applications. An 802.11n LDPC decoder was designed using Vivado HLS achieving 13.4 Mbit/s throughput for a Spartan 6 LX150T, operating at 122 MHz and for a frame length of 648 symbols [9].

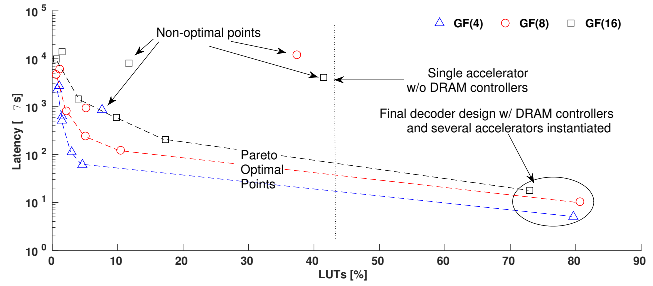


Fig. 13. Pareto plotting of the design space for the LDPC accelerator in latency (μs) vs. LUT utilization (%). On the rightmost side, the design points correspond to the final decoder with replicated accelerators (14, 5 and 3, respectively for GF($2^2, 2^3, 2^4$)). Latency is per instantiated accelerator.

TABLE IV
PROPOSED DECODERS AND RELATED WORK DECODING THROUGHPUT, FPGA UTILIZATION AND FREQUENCY OF OPERATION*.

Decoder	m	K	LUT [%]	FF	BRAM	DSP	Thr. [Mbit/s]	Clk [MHz]
This work	2	1	14	7	0.5	0.5	1.17	250
		14	80	35	6	6	14.54	219
	3	1	21	9	0.9	0.9	0.95	250
		6	81	34	5	5	4.81	210
	4	1	30	13	2	2	0.66	216
		3	73	32	5	5	1.85	201
[13]	4	1	48 (Slices)		41	N/A	9.3	N/A
[14]	2		N/A				33.16	100
	4						13.22	
	8						1.56	
[11]	3		13	3	1	N/A	≤4.7	99
[12]	6		19	6	1	N/A	2.95	61
[8]	8		85 (LEs)		62	7	1.1	163
[9]	1		14 (Slices)		21	N/A	13.4	122

* Differences in technology nodes and FPGA are not considered.

Complex non-binary LDPC decoder architectures found in the literature for FPGA devices are usually developed at RTL level. Most non-binary LDPC decoders Sulek *et al.* have exploited the use of DSP blocks of the FPGA to perform the multipliers used in the CNs and adders in the VNs computation reporting 6 Mbit/s of throughput for code (480,240) with column weight $d_v = 2$ in GF(2^5) [10]. Spagnol *et al.* mixed-domain RTL-based decoder for the GF(2^3) Mackay code obtains 4.7 Mbit/s on a Virtex 2 Pro FPGA [11]. Boutillon *et al.* developed a decoder architecture for a GF(2^6)-LDPC decoder based on the EMS algorithm reporting a decoding throughput of 2.95 Mbit/s for an occupied area of 20% of a Virtex 4 FPGA. Although their architecture scales with minimal adaptation of the design to higher order GF(q), with $q \geq 2^{12}$, no throughputs are reported for q other than 2^6 [12]. Zhang *et al.* developed a layered partial-parallel architecture, achieving 9.3 Mbit/s throughput at 15 iterations for a GF(2^5) $(744, 653)$ length non-binary code of rate 0.88, with a frequency of operation of 106 MHz on a Virtex-2 Pro [13]. Emden *et al.* study the scalability of the non-binary decoder with a growing GF(2^m), on a Virtex 5 FPGA, for $m{=}\{2, 4, 8\}$. Their achieved throughputs range from 1.6 up to 33.1 Mbit/s which illustrate the complexity of the algorithm, as the decoder scales with $11\times$ more area and a throughput 95% inferior [14].

While surpassing the performance of RTL-dedicated solutions with HLS-based descriptions may be out of reach now, we show that an efficient accelerator can be designed performing at similar orders of throughput magnitude.

VII. CONCLUSION

As the complexity of designing more sophisticated signal processing algorithms increases, the kind of design space exploration problems addressed in this paper are increasingly becoming a real concern of hardware designers. While RTL-based development can achieve the best performance, it incurs very high development costs for FPGA accelerators. Alternatively, one can target FPGAs through HLS tools that significantly reduce the development time and allow the designer to produce reasonably good results. Although the design space exploration must leverage upon application and hardware design knowledge in order to reach a satisfactory design point, once a solid base version is established, the design cycle requires little code refactoring due to the directive-based modifications of the HLS tool. We have considered a complex scenario of non-binary LDPC decoders used in channel coding and report throughputs of 0.66–1.17 Mbit/s for a single-decoder architecture, and 1.85–14.54 Mbit/s for multi-decoder design, which compare fairly against RTL-based decoders found in the literature, especially when the significant productivity gap is factored inn. Optimized HLS libraries like the ones available for the video space [15] could further help performance convergence while extending the productivity lead. The optimization strategies adopted in this work provide insights on how other complex problems can be efficiently designed in HLS. The LDPC decoder source code is available at https://github.com/andradx/fftspa_fpl2015.

REFERENCES

[1] M. Maxfield, "Google's project ARA smartphones to use lattice ECP5 FPGAs," *EE Times*, April 2014.
[2] A. Putnman *et al.*, "A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services," in *Proc. ACM/IEEE ISCA*, 2014, pp. 13–24.
[3] G. Martin and G. Smith, "High-Level Synthesis: Past, Present, and Future," *IEEE Des. Test*, vol. 26, no. 4, pp. 18–25, Jul. 2009.
[4] T. Richardson and R. Urbanke, "The renaissance of Gallager's low-density parity-check codes," *IEEE Comm. Mag.*, vol. 41, no. 8, 2003.
[5] R. A. Carrasco and M. Johnston, *Non-Binary Error Control Coding for Wireless Communication and Data Storage*. Wiley, Chichester, 2008.
[6] Xilinx Inc., "The Xilinx SDAccel Development Environment."
[7] D. B. Kirk and W. H. Wen-mei, *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.
[8] J. Andrade *et al.*, "Flexible non-binary LDPC decoding on FPGAs," in *Proc. IEEE ICASSP*, May 2014, pp. 1936–1940.
[9] E. Scheiber *et al.*, "Implementation of an LDPC decoder for IEEE 802.11n using Vivado High-Level Synthesis," in *Proc. IEEE ICECS*, 2013, pp. 45–48.
[10] W. Sulek *et al.*, "GF(q) LDPC decoder design for FPGA Implementation," in *Proc. IEEE CCNC*, Jan 2013, pp. 460–465.
[11] C. Spagnol *et al.*, "FPGA Implementations of LDPC over GF(2^m) Decoders," in *Proc. IEEE SiPS*, Oct 2007, pp. 273–278.
[12] E. Boutillon *et al.*, "Design of a GF(64)-LDPC Decoder based on the EMS Algorithm," *IEEE TCS—I*, vol. 60, no. 10, pp. 2644–2656, 2013.
[13] X. Zhang and F. Cai, "Efficient Partial-Parallel Decoder Architecture for Quasi-Cyclic Nonbinary LDPC Codes," *IEEE TCS–I*, vol. 58, no. 2, pp. 402–414, 2011.
[14] T. Lehnigk-Emden and N. Wehn, "Complexity Evaluation of Non-binary Galois Field LDPC Code Decoders," in *Proc. IEEE ISTC*, Sept 2010, pp. 53–57.
[15] Xilinx Inc., "Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries," XAPP1167.