R-LDPC: Refining Behavior Descriptions in HLS to Implement High-throughput LDPC Decoder

Yifan Zhang[⋆], Qiang Cao[⋆✉], Jie Yao[†], Hong Jiang[‡]
[⋆]Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology,
[†]School of Computer Science and Technology, Huazhong University of Science and Technology,
[‡]Department of Computer Science and Engineering, University of Texas at Arlington,
[✉]Corresponding Author: caoqiang@hust.edu.cn

***Abstract*—High-Level Synthesis (HLS) translates high-level behavior-description to Register-Transfer Level (RTL) implementation in modern Field-Programmable Gate Arrays (FPGAs), accelerating domain-specific hardware developments. Low-Density Parity-Check (LDPC), as a powerful error-correction code family, has been widely implemented in hardware for building a reliable data channel over a noisy physical channel in communication and storage applications. Leveraging HLS to fast prototype high-performance LDPC decoder is intriguing with high scalability and low hardware-dependence, but generally is sub-optimal due to the lack of accurate and precise behavior descriptions in HLS to characterize iteration- and circuit-level implementation details. This paper proposes an HLS-based QC-LDPC decoder with scalable throughput by precisely refining the LDPC behavior descriptions, R-LDPC for short. To this end, R-LDPC first adopts an HLS-based LDPC decoder microarchitecture with a module-level pipeline. Second, R-LDPC offers a multi-instance-sharing one (MSO) description to explicitly define shared parts and non-shared parts for an array of check-node updating-units (CNU), eliminating redundant function modules and addressing circuits. Third, R-LDPC designs efficient single-stage and multi-stage shifters to eliminate unnecessary bit-selection circuits. Finally, R-LDPC provides invalid-element aware loop scheduling before the compile phase to avoid some unnecessary stalls at runtime. We implement an R-LDPC decoder, compared to the original HLS-based implementation, R-LDPC reduces the hardware consumption up to 56%, the latency up to 67%, and the decoding throughput up to 300%. Furthermore, R-LDPC is adapted to different scales, LDPC standards, and code rates, and can achieve 9.9Gbps decoding throughput in Xilinx U50.**

***Index Terms*—Low-density parity-check (LDPC), High-level synthesis (HLS), Field-programmable gate array (FPGA)**

I. Introduction

Low-Density Parity-Check codes (LDPC) are a popular and powerful family of linear error-correction codes close to the Shannon limit [1], which have become safeguards for building reliable cyberspace upon noise-filled physical space. Quasi-cyclic (QC) LDPC codes are a subclass of LDPC codes with regular code structures facilitating the hardware implementation [2], widely adopted in many wireless standards (e.g., WiFi and CCSDS) and in data storage (e.g., solid-state drives).

LDPC decoders with high computation complexity have been extensively studied in code structures, decoding algorithms and hardware implementation [3] [4]. Generally, a Register-Transfer Level (RTL) LDPC decoder implementation is dedicated to a given code structure and decoding algorithm, which is generally long and prone-error process even for RTL experts [5].

Fortunately, High-Level Synthesis (HLS) has been advancing to automatically compile untimed behaviors described in high-level languages such as C/C++ or OpenCL into the RTL, thus dramatically improving hardware-development productivity, especially for hardware accelerators [6]–[8]. Xilinx Vivado HLS (renamed Vitis HLS now) has been the most popular commodity HLS [9] used to implement FPGA applications according to its corresponding C/C++ algorithms.

However, there still exists a considerable Quality-of-Result (QoR) gap in the LDPC-decoder between HLS-generated and RTL-expert-generated designs. By analyzing the RTL implementation of the QC-LDPC decoder based on a straightforward algorithm description of HLS (H-LDPC), we find that there exists significant hardware inefficiency largely due to a lack of HLS detailed description to accurately characterize specified behaviors and functions in three key decoding modules. First, H-LDPC does not distinguish between shared and non-shared parts for multiple hardware instances. Second, H-LDPC realizes a general bit-selection-based shifter. Third, H-LDPC implements loop iteration and does not consider whether the iteration is valid.

We present an HLS-based refining behavior-description QC-LDPC decoder (R-LDPC). Specifically, R-LDPC first offers a multi-instance-sharing-one expression to avoid repetitive shared-part hardware. Second, R-LDPC characterizes element-wise vector processing and further designs single/multi-stage shifters. Finally, R-LDPC reduces unnecessary element-invalid iterations before the compiling phase and decoding delay.

We implement an R-LDPC decoder in Xilinx U50. Compared to the original HLS-based implementation, R-LDPC reduces the hardware consumption by up to 56% and the latency by up to 67% and increases the decoding throughput by up to 300% (209Mbps). The throughput of the R-LDPC decoder can linearly scale to 9.9Gbps, outperforming existing HLS-based implementations and exceeding most hand-crafted RTL designs.

The main contributions of this work are as follows:

- We design an HLS-based QC-LDPC decoder microarchitecture and identify the inefficiency of origin HLS implementation.
- We refine three key precise behavior descriptions to design three corresponding optimizations.
- We implement and evaluate the R-LDPC decoder in real hardware to manifest its effectiveness, scalability, and flexibility.

II. Background

A. FPGA and High-level Synthesis

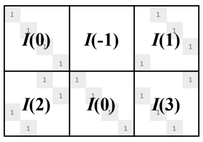
Field-Programmable Gate Array (FPGA) is a reconfigurable integrated circuit, ubiquitously used for domain-specific hardware, from IoT devices to cloud data centers. FPGA is mainly composed of Look-Up-Table (LUT), Flip-Flops (FF), and routing resources. Modern FPGA usually uses standard on-chip bus protocols for high-speed communication inside or outside the chip, such as AXI-MM, AXI-Lite, AXI-S, etc.

High-Level Synthesis (HLS) is an automated design process that compiles an algorithmic description of a logic behavior to the corresponding FPGA hardware implementation at the Register-Transfer Level (RTL). Traditional RTL-based manual design generally is error-prone with a long development cycle [5]. In contrast, HLS-based automated hardware development has higher productivity and flexibility, which is broadly used in large-scale and/or new functionality scenarios, e.g. machine learning, graph computing, and domain-specific accelerators.

Mainstream HLS, such as Vivado HLS and Intel HLS, uses C/C++ as development languages [10] and additional pragmas or directives to characterize hardware behaviors. For example, `UNROLL` indicates whether a loop is implemented as multiple hardware instances in parallel or multiple-cycle serial processing. HLS automatically schedules according to the built-in delay model under the frequency requirement. In addition, HLS also largely decouples algorithms, scalability, and hardware platforms.

Although HLS can rapidly achieve the required functionality in the hardware, the implementation generally can be sub-optimal in hardware efficiency due to a high-level abstract behavior corresponding to multiple realizations with different performances. It is generally necessary to optimize with the pragmas provided by HLS and refactor the behavior-description to improve the quality of HLS results (QoR) [11], [12].

B. QC-LDPC and Decoding Algorithm



(a) Check matrix $\boldsymbol{H}$.

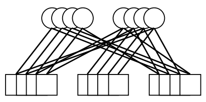

(b) Tanner graph.

Fig. 1: An example of QC-LDPC parity check matrix extending from the base matrix and its tanner graph.

LDPC codes have superior error-correction capability and channel efficiency at the cost of computation complexity. Original LDPC codes with a randomly generated $\boldsymbol{H}$ matrix cause irregular and complex wire routing circuits. Quasi-Cyclic LDPC (QC-LDPC) [13] codes are well-structured and relatively easy to implement in hardware without significant error-correction performance decreases and have been widely used in the communication and storage field. Fig. 1 gives a base matrix and its tanner graph of an example QC-LDPC. Its parity check matrix consist of 2×3 $I(n)$ sub-matrices.The size of $I(n)$ is $Z \times Z$. The $I(n)$ sub-matrix is an n-times circulant permutation matrix from the unit matrix ($I(0)$). The $I(-1)$ represents a zero matrix.

In the parity check matrix of the LDPC code. Each row corresponds to a check node (CN) and each column corresponds to a variable node (VN). 1 represents exist edge between the VN and CN. The decoding algorithm of LDPC is based on message passing. There are two classic node update scheduling of decoding algorithms, namely flood and layered decoding.

The flood decoding first updates messages from CN to VN (R), and then updates messages from VN to CN (Q) with the log-likelihood probability of each bit (P). The layered decoding divides an LDPC check matrix into multiple layers. Only needs to update R and Q in a layer then updates P. The layered approach speeds up the decoding convergence and reduces the complexity of the LDPC decoder interconnection, but introduces additional inter-layer dependencies. Meanwhile, the QC-LDPC decoder can be implemented semi-parallel to exploit sub-matrix level.

III. Design

A. R-LDPC Decoder Microarchitecture

We use HLS to implement a layered semi-parallel QC-LDPC decoder with three key features: value-reuse, intra-block parallel and inter-block serial scheduling. The entire decoding process comprises the following 5 steps:

① Read P and R. At initialization, the value of P is read from $INPUT$ and the value of R is 0.

② Calculate Q from $P - R$.

③ Uses Q as input, R_{new} as output executes CN updating.

④ Generate P_{new} from $R_{new} + Q$

⑤ Update P and R

The entire decoding process includes multiple iterations with multiple sub-iterations, each of sub-iteration executes the complete ①②③④⑤ process.

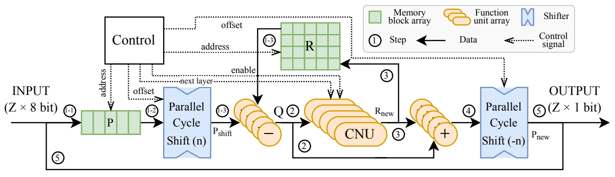
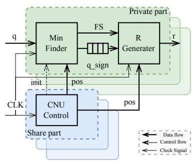


Fig. 2: R-LDPC decoder microarchitecture

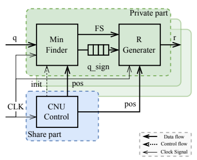
We design a decoder microarchitecture, as shown Fig. 2, and use HLS to describe its entire decoding behavior. P and R are stored in the memory block arrays, which are implemented by C++ arrays. In addition, P_{shift}, R_{new} and P_{new} are added as C++ arrays for more accurately express decoding behavior. ① is implemented as three sub-steps: read $INPUT$, initialize P with $INPUT$, and executes parallel cyclic shift by assignment to P_{shift} from P in a loop; ② executes a subtraction from R and P_{shift} to Q in a loop; ③ defines a CNU class, declare an object array of CNU and use Q call the member function of CNU in a loop to get R_{new}; ④ executes an addition in a loop from R_{new} and Q to P_{new}; ⑤ executes an assignment from P_{new} to P in a loop. Finally, the `PIPELINE` pragma is added to the sub-iteration loop, achieving a module-level pipeline.

However, compared to the representative RTL design [3], the HLS-based decoder has a lower decoding throughput (44% down) but higher hardware consumption (584% up), as detailed in Section IV. By analyzing RTL implementation details, we find that there exists hardware inefficiency in the CNU array and parallel cyclic shifter, while existing HLS cannot accurately define detailed behaviors such as multi-instances sharing one, single/multi-stage shifter and invalid-element aware loop scheduling. This motivates us to refine the behavior descriptions to implement an optimized QC-LDPC decoder.

B. *Multi-instance Sharing One (MSO)*



(a) Original HLS



(b) MSO

Fig. 3: Structures of CNU-array

The CN updating unit (CNU)-array is the compute heart of QC-LDPC decoder to perform the value-reuse feature. As shown in Fig. 3, the CNU-array consists of N CNUs, which are independent but can process the same set of edges in the $I(n)$ sub-matrices simultaneously. All CNUs share the CNU control but have their internal state (PS), final state (FS) and q_sign as non-shared members.

```
 1  class CNU{
 2     bit3 pos; // ....shared part
 3     CNU_State PS; //private part
 4     CNU_State FS; //private part
 5     FIFO q_signs; //private part
 6     bit6 f(bit6 q){
 7  #pragma HLS PIPELINE
 8        bit6 ret;
 9        if(CNU_control(pos))
10
11           FS = PS;
12
13
14        min_finder(PS, q, pos);
15        q_signs << q.sign;
16        ...
17        ret = Rgenerate(
18           FS, q_signs.pop(), pos);
19        return ret;
20     }
21  };
22  ...
23  void decoder(...){
24     CNU cnua[N];
25     ...
26     for (int i=0; i<N; i++){
27  #pragma HLS UNROLL
28        ... = cnua[i].f(...);
29     }
30     ...
31  }
```

(a) Original CNU-array

```
 1  class CNUA{
 2     bit3 pos;
 3     CNU_State PS[N];
 4     CNU_State FS[N];
 5     FIFO q_signs[N];
 6     V<bit6> f(V<bit6> q){
 7  #pragma HLS PIPELINE
 8        V<bit6> retV;
 9        if(CNU_control(pos))
10           for (int i=0; i<N; i++){
11              FS[i] = PS[i];
12           }
13        for (int i=0; i<N; i++){
14        min_finder(PS[i],q[i],pos);
15        q_signs[i] << q[i].sign;
16        ...
17        ret[i] = Rgenerate(FS[i],
18           q_signs[i].pop(), pos);}
19        return retV;
20     }
21  };
22  ...
23  void decoder(...){
24     CNUA cnua;
25     ...
26
27
28     cnua.f(...);
29
30     ...
31  }
```

(b) MSO CNU-array

Fig. 4: CNU-array HLS codes

As shown in Fig. 4a, a CNU is defined as a C++ class on lines 1-22 containing its members and functions. Line 24 defines N CNU-instances. The CNU-array is implemented with loop and `UNROLL` pragma (lines 26 to 29) to generate N CNU hardware independently. However, the shared parts of the CNU are also implemented as multiple copies as Fig. 3a shows.

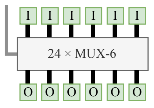
To solve this problem, we propose Multi-instance-Sharing-One (MSO) description. MSO explicitly defines a CNU-array class containing a set of member arrays. Each element of an array is associated with a non-shared member/function of an instance. A shared member/function is only generated once as shown in Fig. 3b.

Specifically, as shown in Fig. 4b, we first define a CNU-array class (lines 1, 24). Second, the members corresponding to the non-shared part in the CNU-array class are implemented as arrays (lines 3-5), Third, the loop with `UNROLL` is used to express multiple instances of the non-shared part inside the member function of the CNU-array class (lines 10-18). The read of the shared part is inside the loop (lines 14, 18), and the write of the shared part is outside the loop (lines 9). Therefore, multiple CNU instances have their non-shared members but share the no-array members in the CNU-array class.

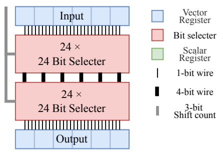
C. *Parallel Cyclic Shifter*

```
for(i=0;i<size;++i)
{
#pragma HLS UNROLL
 outa[i]=
 ina[(i+count)%size];
}
```

(a) Original HLS behavior code



(b) Original HLS (array_partition)



(c) Original HLS (array_remap)

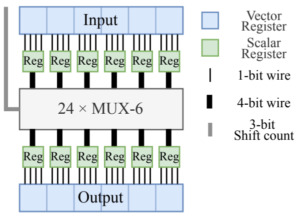
Fig. 5: Parallel cyclic shifter structures in HLS

The parallel cyclic shifter is another important component of the QC-LDPC decoder, which is implemented by a simple loop as Fig. 5a shows. We find that when the ina and $outa$ arrays are applied with different pragmas, the loop is implemented as different structures in HLS.

Specifically, when the input/output array is specified as an array_partition, the implemented structure (Fig. 5b) is similar to a barrel shifter with some redundant addressing circuits. When the input/output array is specified as a vector or array_remap (Fig. 5c), the addressing circuit is no longer redundant but implemented structure has a very high hardware consumption (about 277k LUTs). The reason is, i becomes a constant after the loop is unrolled, $count$ is a runtime variable, so $(i{+}count)\%size$ is a runtime variable. In HLS, a vector means a high-width integer, so the behavior of those codes is to use a variable as an address to get some bits from a high-width integer and obtain the structure as shown in Fig. 5c.

```
 1  void vshifter(...){
 2  #pragma HLS PIPELINE
 3     T in_reg[size];
 4  #pragma HLS array_partition ...
 5     T out_reg[size];
 6  #pragma HLS array_partition ...
 7     for (int j=0;j<size;++j){
 8        in_reg[j]=ina[j];}
 9     for (int j=0;j<size;++j){
10        out_reg[j]=
11        in_reg[(j+count)%size];}
12     for (int j=0;j<size;++j){
13        outa[j]=out_reg[j];}
14  }
```

(a) HLS code



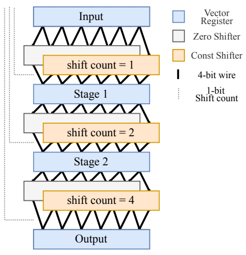
(b) Single-stage shifter

Fig. 6: Parallel cyclic shifter structures in HLS

1) Single-stage Shifter: To improve hardware efficiency, we first design a single-stage shifter behavior description. Specifically, as shown in Fig. 6a, first in_reg and out_reg array added to this function with array_partition pragma at lines 3 and 5. Second, every element is assigned in the input vector to the in_reg array at line 8. Third, the in_reg and out_reg are used to perform a parallel cyclic shift at lines 10-11. Finally, every element is assigned in the out_reg to the output vector at line 13. This approach can obtain an expected barrel shifter structure (Fig. 6b) without redundant addressing circuits.

```
template<int size, typename T>
void logshifter(
   V<T> &ina, V<T> &outa,
   3bit count){
#pragma HLS PIPELINE
   V<T> s1;
   V<T> s2;
   if(count[0])
        cshifter<T, 1>(ina, s1);
   else cshifter<T, 0>(ina, s1);
   if(count[1])
        cshifter<T, 2>(s1, s2);
   else cshifter<T, 0>(s1, s2);
   if(count[2])
        cshifter<T, 4>(s2, outa);
   else cshifter<T, 0>(s2, outa);
}
```

(a) HLS code



(b) Multi-stage shifter

Fig. 7: Structure of logarithmic shifter

2) Multi-stage Shifter: We further implement a multi-stage parallel shifter, shift 0 or 2^i elements ($i = 0, 1, 2, 3...$) in each stage, as shown in Fig. 7a. Since the hardware consumption of a parallel cyclic shift with a constant shift count is small, the shifter can reduce the hardware consumption with the same decoding throughput. The multi-stage shifter structure is similar to the logarithmic shifter as shown in Fig. 7b, which reducing the hardware complexity from $O(N^2)$ to $O(NlogN)$.

D. Invalid-element Aware Loop Scheduling

In many LDPC communication standards, the base matrix is irregular and has a large number of invalid elements ('-1'). For example, in the 802.16e standard with 1/2 code, about 70% elements in the base matrix are invalid.

```
 1
 2  int base_m[][] = {...};
 3
 4  for (int i=0; i<col; i++){
 5     for (int j=0; j<row; j++){
 6        if(base_m[i][j] != -1){
 7           ...
 8           P = Parray[j];
 9           logshfter(
10           P, Pshift,base_m[i][j]);
11           ...
12  } } }
```

(a) Original HLS code

```
 1  int jMAX = ...;
 2  int c_matrix[][jMAX] = {...};
 3  int c_index[] = {...};
 4  for (int i=0; i<col; i++){
 5     for (int j=0; j<jMAX; j++){
 6
 7           ...
 8           P = Parray[c_index[j]];
 9           logshfter(
10           P, Pshift,c_matrix[i][j]);
11           ...
12  } }
```

(b) R-LDPC HLS code

Fig. 8: Base matrix and usage

For a loop implemented with a `PIPELINE` in HLS, the decoding delay mainly depends on the number of loop iterations as shown in Fig. 8a at line 5. For R-LDPC, the number of loop iterations is the element number in the base matrix. because the pipelined decoder does not handle invalid elements, the corresponding handling cycles cannot be eliminated at runtime.

We reduce the actual number of iterations, as shown in Fig. 8b. We first traverse the base matrix and only keep valid elements in the base matrix, and then compact each column of the matrix. Second, all compacted rows construct a compacted matrix (c_matrix at line 2) and use an additional index array (c_index at line 3) to record the position of each element in the original array. The compacted matrix with valid elements has a shorter columns-length ($jMAX$ at line 5) which determines the number of iterations.

IV. Implementation and Evaluation

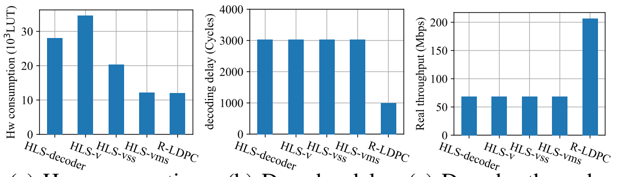
A. Implementation and Experimental Setup

We use the Xilinx Vitis hardware development platform for implementing a set of LDPC decoders on FPGA and conduct a set of experiments to analyze the performance and hardware consumption. The experimental platform employs a server with Intel Xeon CPU E5-2660 and 173GB memory. The FPGA used for the experiment is a Xilinx Alveo U50 data center accelerator card (A-U50-P00G-PQ-G) with official firmware (Xilinx_u50_gen3x16_xdma_201920_3). The version of Vitis is 2022.2, and the version of XRT is 2.6.655.

The decoders use the 802.16e and the CCSDS standard with different code rates (0.5-0.9), the expansion factor is 64 and the decoding iterations number is 5.

B. Overall Performance

We implement five version decoders, including the original-HLS as the baseline (HLS-decoder), HLS-decoder with the MSO (HLS-v), HLS-v decoder with the single-stage shifter (HLS-vss), HLS-v decoder with the multi-stage shifter (HLS-vms), and HLS-vms decoder with invalid-element aware loop scheduling (R-LDPC).



(a) Hw consumption (b) Decoder delay (c) Decoder throughput

Fig. 9: Decoder hardware consumption and performance

Fig. 9 shows the performance and resource consumption of these five versions of the decoders. The R-LDPC decoder with 305% higher decoding throughput at a 56% lower LUT usage than the H-LDPC.

Compared with the HLS-decoder, HLS-v consumes more hardware resources. The reason is that the MSO CNU array has a vectorized memory interface, without the element-aware vector, HLS will generate an extra bit-selection circuit. However, by applying a single-stage shifter and multi-stage shifter, the HLS-vb decoder reduces hardware consumption by 27%, HLS-vl further decreases hardware consumption by 56%. These four decoders do not dramatically change either decoding latency or throughput. Nevertheless, when using invalid-element aware matrix scheduling, the R-LDPC decoder gains improvements in all three metrics as decoding latency, throughput, and hardware consumption.

C. Sensitivity Study

Next, we further analyze the effect of MSO and single-/multi-stage shifter optimization in detail.

The kernel is the smallest runnable unit of the Vitis platform. To independently evaluate the optimization effect of R-LDPC, a kernel containing only one function module and bus interface module is implemented in this experiment. The INLINE-off pragma is applied to the function module to retain the module boundary for easier analysis of the hardware consumption of itself.

1) Multi-instance sharing one (MSO): In this experiment, a kernel contains only the CNU array and bus interface modules. The CNU array has 64 CNUs with 2-bit calculation bit-width. The baseline CNU array uses the code as shown in Fig. 4a, while the MSO CNU array uses the code as shown in Fig. 4b.

TABLE I: The hardware consumption of CNU array

Type of CNU array	Hardware consumption (LUT)	Hardware consumption (FF)
Baseline	1345	1740
MSO	1165	1554

Table. I shows the result of the CNU array, indicating that the hardware consumption of the MSO CNU array is 13% lower than that of the baseline CNU array.

2) Parallel Cyclic Shifter: Three versions of shifters are implemented in this experiment. All shifters shift 64 elements in parallel, with each element being an 8-bit integer. The baseline shifter is the original HLS version with the code as shown in Fig. 5a and the single-/multi-stage shifter.

TABLE II: Delay and hardware consumption of three types of shifters

Shifter type	Delay (Cycles)	Hardware consumption (LUT)
baseline	1	8599
single-stage	0	6627
multi-stage	1	797

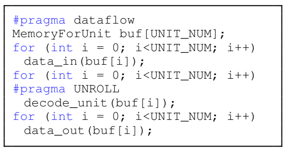
As shown in Table. II, compared to the baseline shifter, the single-stage shifter reduces the hardware consumption and the delay by eliminating the bit-selection circuit. Compared with the single-stage shifter, the multi-stage shifter further greatly reduces the hardware usage due to its lower hardware complexity, and only increases the delay by one cycle. Because the circuit delay of the const cyclic shifter in HLS is significantly less than a one-cycle delay, multiple shifting operations can be completed within one cycle without increasing pipeline delay.

D. Scalability and Flexibility

To manifest the scaling advantages of R-LDPC, we use the R-LDPC decoder as a decoding unit (DU), simply implement the coarse-grained pipeline across DU as shown in Fig. 10, and adjust the number of DUs to increase the decoding throughput of the decoder.

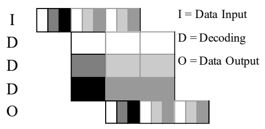
In addition, we adopt the different LDPC code standards, code rate, and calculation bit-width of the LDPC decoder by simply replacing the key arrays and constants in the R-LDPC behavior description HLS code.

Fig. 11 shows the hardware and decoding throughput scalability in the coarse-grained pipelined manner. The result



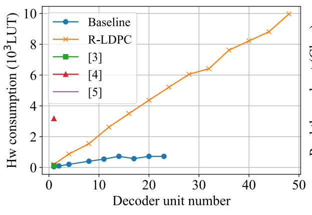
```
#pragma dataflow
MemoryForUnit buf[UNIT_NUM];
for (int i = 0; i<UNIT_NUM; i++)
  data_in(buf[i]);
for (int i = 0; i<UNIT_NUM; i++)
#pragma UNROLL
  decode_unit(buf[i]);
for (int i = 0; i<UNIT_NUM; i++)
  data_out(buf[i]);
```

(a) HLS code

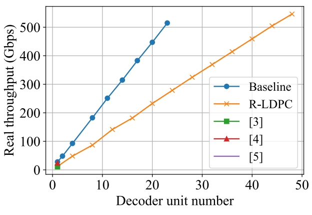


(b) Space-time diagram

Fig. 10: Coarse-grained pipeline



(a) Hardware consumption



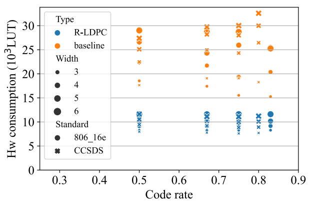
(b) Decoding throughput

Fig. 11: Scalability of R-LDPC decoder

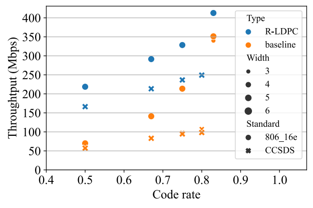
indicates that the R-LDPC decoder scales linearly with the number of decoder units. It also suggests that HLS is convenient and effective for designing large-scale hardware.

Due to limited resources on FPGA, the design with excessive hardware consumption cannot complete synthesis. As a result, the baseline decoder cannot expand the scale beyond 24 decoding units, while R-LDPC can scale to 48 decoding units and obtain maximum decoding throughput up to 9.9Gbps.

In addition, R-LDPC has advantages over the latest RTL-based QC-LDPC decoders [3], [4], [14] in the peak decoding throughput as shown in Fig. 11. Although R-LDPC decoders consume slightly more hardware than these RTL-based expert designs at the same scale. By more accurately describing the latest RTL-based expert design behavior, R-LDPC can further improve the decoding throughput.



(a) Hardware consumption



(b) Decoding throughput

Fig. 12: Flexibility of R-LDPC decoder

Fig. 12 shows the flexibility of H-LDPC and R-LDPC can effectively adapt to different LDPC algorithms, code rates, calculation bit-widths, and the number of iterations, which is mainly due to the functionality of HLS. In all cases, R-LDPC outperforms H-LDPC in reducing the hardware consumption up to 66% and increasing the decoding throughput up to 213%, which indicates that it is still beneficial to refine the behavior description as R-LDPC.

V. Related Work

A. Improve QoR of HLS by Refining Behavior Description

While HLS already supports many syntaxes of high-level languages, which can greatly shorten the development cycle of hardware development. But its semantics and automatic optimizations are still limited, which leaves room for improvement in QoR.

Winterstein and Homsirikamol et al. [15], [16] conducts a study on how to restructure C codes to improve QoR with HLS. They convert the code manually, which effectively improves the performance of the hardware. However, such conversion requires developers to have rich hardware development experience, and the optimized code is not intuitive and portable.

Young-kyu and Cong [11] propose an automated source code conversion, but this work can only deal with three specific variable boundary loops. However, the QoR of the HLS-based LDPC decoder cannot benefit from these automated methods.

B. HLS-based LDPC Decoder Design

Some excellent HLS-based LDPC decoder designs achieve performance close to RTL designs.

Mhaske et al. [17], [18] propose a layered decoder based on LabVIEW-based HLS. Although the performance is excellent, the HLS used in this work is designed to describe hardware through low-level circuit diagrams, which has a big semantic gap from the mainstream HLS based on high-level programming languages.

Wang et al. [19] implemented a performance-balanced general-purpose QC-LDPC decoder design using Vivado HLS, which is similar to the expert design in 2007 [20], However, this microarchitecture cannot support layered decoding algorithms, which have faster decoding iterative convergence and lower complexity of interconnection. Therefore, it consumes more hardware resources (the closest design is about 25K LUTs) than the R-LDPC decoder (14K LUTs) under similar decoding throughput.

VI. Conclusion

We design an HLS-based QC-LDPC decoder microarchitecture and identify the inefficiency of H-LDPC. We propose three refined behavior descriptions to realize the corresponding optimizations. These optimizations just use standard C++ and common HLS semantics, without relying on specific HLS and new compilers. The experiments show that the R-LDPC has effectiveness, scalability, and flexibility.

R-LDPC reduces hardware consumption up to 66%, increases decoding throughput up to 213% and reaches the maximum decoding throughput up to 9.6 Gbps on the U50.

VII. Acknowledgments

This work was supported in part by the National key research and development program of China under Grant 2018YFA0701800, NSFC No.62172175, Creative Research Group Project of NSFC No.61821003, Key Research and Development Project of Hubei No.2022BAA042, and by the US National Science Foundation grant CNS-2008835 and CCF-2226117.

References

[1] R. Gallager, "Low-density parity-check codes," *IEEE Transactions on Information Theory*, vol. 8, no. 1, pp. 21–28, Jan. 1962.

[2] M. Mansour and N. Shanbhag, "Low-power VLSI decoder architectures for LDPC codes," in *Proceedings of the International Symposium on Low Power Electronics and Design*, 2002, pp. 284–289.

[3] K. K. Gunnam, G. S. Choi, W. Wang, E. Kim, and M. B. Yeary, "Decoding of Quasi-cyclic LDPC Codes Using an On-the-Fly Computation," in *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, Oct. 2006, pp. 1192–1199.

[4] E. A. Papatheofanous, D. Reisis, and K. Nikitopoulos, "LDPC Hardware Acceleration in 5G Open Radio Access Network Platforms," *IEEE Access*, vol. 9, pp. 152 960–152 971, 2021.

[5] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A Survey and Evaluation of FPGA High-Level Synthesis Tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 10, pp. 1591–1604, Oct. 2016.

[6] J. Cong and J. Wang, "PolySA: Polyhedral-based systolic array auto-compilation," in *Proceedings of the International Conference on Computer-Aided Design*. San Diego California: ACM, Nov. 2018, pp. 1–8.

[7] A. Özeloğlu and İ. San, "Acceleration of Neural Network Training on Hardware via HLS for an Edge-AI Device," in *2020 Innovations in Intelligent Systems and Applications Conference (ASYU)*. IEEE, 2020, pp. 1–6.

[8] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong, "FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, 2017, pp. 152–159.

[9] E. Homsirikamol and K. G. George, "Toward a new HLS-based methodology for FPGA benchmarking of candidates in cryptographic competitions: The CAESAR contest case study," in *2017 International Conference on Field Programmable Technology (ICFPT)*, 2017, pp. 120–127.

[10] J. Andrade, N. George, K. Karras, D. Novo, F. Pratas, L. Sousa, P. Ienne, G. Falcao, and V. Silva, "Design Space Exploration of LDPC Decoders Using High-Level Synthesis," *IEEE Access*, vol. 5, pp. 14 600–14 615, 2017.

[11] Y.-k. Choi and J. Cong, "HLS-based optimization and design space exploration for applications with variable loop bounds," in *Proceedings of the International Conference on Computer-Aided Design*. San Diego California: ACM, Nov. 2018, pp. 1–8.

[12] J. Cong, P. Wei, C. H. Yu, and P. Zhang, "Automated accelerator generation and optimization with composable, parallel and pipeline architecture," in *Proceedings of the 55th Annual Design Automation Conference*, ser. DAC '18. New York, NY, USA: Association for Computing Machinery, Jun. 2018, pp. 1–6.

[13] R. Townsend and E. Weldon, "Self-orthogonal quasi-cyclic codes," *IEEE Transactions on Information Theory*, vol. 13, no. 2, pp. 183–195, Apr. 1967.

[14] V. L. Petrović, M. M. Marković, D. M. E. Mezeni, L. V. Saranovac, and A. Radošević, "Flexible High Throughput QC-LDPC Decoder With Perfect Pipeline Conflicts Resolution and Efficient Hardware Utilization," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 12, pp. 5454–5467, Dec. 2020.

[15] F. Winterstein, S. Bayliss, and G. A. Constantinides, "Separation Logic-Assisted Code Transformations for Efficient High-Level Synthesis," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, 2014, pp. 1–8.

[16] E. Homsirikamol and K. Gaj, "Can high-level synthesis compete against a hand-written code in the cryptographic domain? A case study," in *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, 2014, pp. 1–8.

[17] S. Mhaske, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "FPGA-Based Channel Coding Architectures for 5G Wireless Using High-Level Synthesis," *International Journal of Reconfigurable Computing*, vol. 2017, pp. 1–23, 2017.

[18] S. Mhaske, D. Uliana, H. Kee, T. Ly, A. Aziz, and P. Spasojevic, "A 2.48Gb/s FPGA-based QC-LDPC decoder: An algorithmic compiler implementation," in *2015 36th IEEE Sarnoff Symposium*, Sep. 2015, pp. 88–93.

[19] B. Wang, J. Kang, and Y. Zhu, "Performance Balanced General Decoder Design for QC-LDPC Codes Using Vivado HLS," in *2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC)*, Jun. 2021, pp. 26–30.

[20] Z. Wang and Z. Cui, "A Memory Efficient Partially Parallel Decoder Architecture for Quasi-Cyclic LDPC Codes," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 4, pp. 483–488, Apr. 2007.