Performance Balanced General Decoder Design for QC-LDPC Codes Using Vivado HLS

Bingbing Wang[1,2], Jing Kang[1,2] and Yan Zhu[1]
[1]*National Science Space Center, Chinese Academy of Sciences, Beijing, China*
[1]*School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing, China*
bingice.w@gmail.com, k_naive@163.com, zhuyan@nssc.ac.cn

2021 IEEE 11th International Conference on Electronics Information and Emergency Communication (ICEIEC) | 978-1-6654-0385-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICEIEC51955.2021.9463813

***Abstract*—Field-programmable gate array are frequently used to implement high-speed Low-density Parity-check (LDPC) decoders. The conventional practice is to develop a register transfer level (RTL) description of a digital circuit, which requires considerable simulation and verification, resulting in a long development cycle. High-level synthesis (HLS) tools significantly accelerate the hardware design process by using software programming languages such as C/C++. Well optimized code can be synthesized by the HLS tools into efficient hardware circuits. In this paper, we designed a Xilinx FPGA-based LDPC decoder using Vivado HLS. The decoder takes advantage of the cyclic shift of the submatrix in the parity check matrix of the quasi-cyclic LDPC (QC-LDPC) codes and uses a wide-pipeline architecture to process one row/column of all non-zero submatrices per clock cycle, with decoding throughput of hundreds of Mbps. By using different arrays to store data on multiple diagonals in a submatrix can easily avoid the performance bottleneck caused by access conflicts. In addition, the decoder has strong generality and flexibility, which has a high practical value.**

***Index Terms*—QC-LDPC; LDPC decoder; high-level synthesis; FPGA.**

I. Introduction

Low-density parity-check (LDPC) codes are nowadays widely adopted by digital communication standards due to their powerful error correction capability [1-4]. The implementation of LDPC decoders on field-programmable gate arrays (FPGAs) traditionally commences with the elaboration of a register transfer level (RTL) description of a digital circuit to perform the computation [5-9]. Producing such an RTL description, however, is a laborious task where one has to know the details of the hardware. High-level synthesis (HLS) tools enable users without such expertise to develop complex hardware designs using conventional software development languages, such as C/C++. Moreover, HLS tools enable users cost-effectively explore a large design space that achieve the suitable balance between performance and resource consumptions.

Design of high-speed LDPC decoders using HLS tools has become a hot topic in recent years [10-13]. [10] proposed a layered hardware decoder architecture that utilized the advantage of intra-frame parallelization for HLS implementations. The decoders in [11] explored dataflow and wide pipeline design approaches with a high flexibility. [12] generated a wide-pipeline architecture for 802.16e LDPC codes. [13] explored in more detail the potential of designing LDPC decoders with HLS. These designs either do not adequately balance performance and resource consumptions, or are designed for a particular code and thus suffer from a lack of generality. In this paper, we make full use of the quasi-cyclic nature of parity check matrix (PCM) of the QC-LDPC codes to store the decoding messages compactly so as to minimize the consumption of hardware resources, and process the decoding messages in all sub-matrices in parallel with the optimization directives provided by the Vivado HLS to maximize the throughput. The distributed storage strategy is also used to avoid performance bottlenecks caused by access conflicts due to multiple diagonal sub-matrices. Meanwhile, the parameterized design enhances the generality of the architecture. As can be seen in the implementation results, the decoder designed in this paper achieves higher throughput and implementation efficiency compared to other HLS based decoders, and overall performance can even be comparable to some RTL designs.

The rest of the paper is organized as follows. Section II introduces the min-sum algorithm and Vivado HLS. The architecture of the decoder proposed in this paper and its main features will be presented in Section III. Section IV gives the implementation results and discussions. Section V concludes this paper.

II. Preliminaries

A. Min-Sum Algorithm

The min-sum algorithm (MSA) is an iterative decoding algorithm allows updating any check node or variable node in parallel and contains two phases. The message $m_{c_i \to v_j}$ from the check node c_i to variable node v_j (C2V) can be calculated according to

$$m_{c_i \to v_j} = \min_{v_b \in N(c_i) \backslash v_j} \left(\left| L_{v_b \to c_i} \right| \right) \times \prod_{v_b \in N(c_i) \backslash v_j} \operatorname{sign} \left(L_{v_b \to c_i} \right) \tag{1}$$

where $N(c_i) \backslash v_j$ denotes the neighboring variable nodes of c_i except for v_j. This equation can be expressed as finding the minimum and the second minimum of the absolute value of the sequence $L_{N(c_i) \to c_i}$. The messages $L_{v_j \to c_i}$ is propagated from the variable node v_j to check node c_i (V2C), it can be calculated as follows

$$L_{v_j \to c_i} = \sum_{c_a \in N(v_j) \backslash c_i} m_{c_a \to v_j} + C_{v_j} \tag{2}$$

978-0-7381-1135-3/21/$31.00©2021 IEEE

where C_{v_j} is log-likelihood ratio (LLR) of variable node v_j, and $N(v_j)\backslash c_i$ denotes the neighboring check nodes of v_j except for c_i.

After initializing the $L_{v_j \to c_i} = C_{v_j}$, and after all check nodes are updated using (1), the newly obtained messages $m_{c_i \to v_j}$ are used as input to (2) for V2C update. The decoding result is output when the maximum number of iterations is reached.

B. Vivado HLS

Vivado HLS is an FPGA development tool from Xilinx that automatically analyzes and exploits the concurrency in algorithms, maps data to memory units to balance resource usage and bandwidth. It also provides a variety of optimization directives, such as pipeline, loop unroll, etc., to help users to optimize their designs.

The *pipeline* directive exploits hardware parallelism to speed up the implementation efficiency. *Unroll* is an optimization method to reduce the number of clock cycles required to perform several iterations each cycle. It is worth noting that if the upper loop is pipelined or completely unrolled, all its nested loops will be automatically unrolled completely. The array in C/C++ is synthesized as a single-port BRAM by default, and access conflicts can become a performance bottleneck when elements in different locations of the same array are accessed simultaneously. The directive *array partitioning* can partition an array into several slices, with each slice stored on a different memory unit, thus allowing the array to be accessed in parallel.

III. Proposed Performance Balanced Decoder

A. Basic Architecture

The PCM of QC-LDPC codes consist of multiple cyclic shifted non-zero submatrices (NZSs) and multiple all-zero submatrices. The size of all submatrices is $Z \times Z$, where *Z* is the lifting size. The base matrix H_b with size $M_b \times N_b$ stores the position of the non-zero elements in the first row of each submatrix. In the proposed architecture, the parallelism of C2V update and V2C update are M_b and N_b, respectively, thus one iteration can be completed with $2 \times Z$ clock cycles. Assume that the number of NZSs is *T*, and there is only single diagonal in all non-zero submatrices. We store the log-likelihood ratio and output in arrays $L_{N_b \times Z}$ and $Out_{N_b \times Z}$, respectively. The decoding messages generated during the iterative process store in the array $Mes_{T \times Z}$ for compact storage. The *t*th row of Mes_t store the messages on the *t*th NZS. The row degree *dc* and column degree *dv* of the code need to be stored in advance, and their sizes are $1 \times M_b$ and $1 \times \mathrm{N}_b$ respectively. We assume that the array *dv* contains *P* different values. In order to store the structure information of PCM, two arrays Mlab and Pos of size $M_b \times \max(dv)$ and $1 \times T$, respectively, also need to be stored. $Mlab_{i,j}$ denotes the label of the *j*th NZSs in *i*th row of H_b, and Pos_t denotes the position where the non-zero element of the first row of the NZSs with the label *t*.

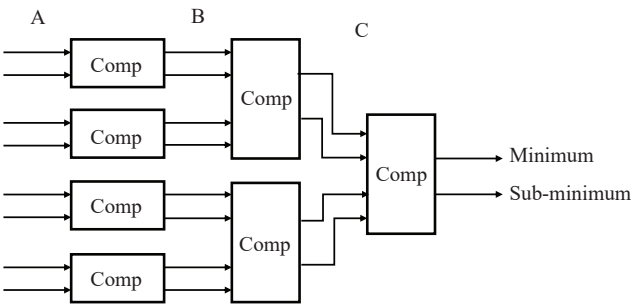

Fig. 1. Find the minimum and the sub-minimum values among the eight elements. The first level of comparison compares the elements in A in pairs and puts the smallest value in front, while the second and third levels of comparators are to obtain the minimum and the sub-minimum values among the four elements.

The proposed performance balanced decoder architecture is described in Algorithm 1. Before starting the iterative process, the array *Mes* needs to be initialized according to $L_{v_j \to c_i} = C_{v_j}$, and then two temporary arrays $tmMes_1$ and $tmMes_2$ of size $M_b \times \max(dv)$ are created for the temporary storage of the messages, where $\max(dv)$ is the maximum column degree of the code. Line 3 to 18 represent the C2V update. Note that we use the *pipeline* directive for the loop in line 3, which means that the loops in line 5, 6 and 11 will be completely unrolled and Vivado HLS will automatically analyze the parallelism between the operations included in these loops. The loop in line 6 reads the messages to be updated from *Mes* into array $tmMes_1$. The array $tmMes_1$ is used as input for the C2V update in line 10, and the updated messages are stored in the array $tmMes_2$. The message is then written to *Mes* again in the loop in line 11. As described in the previous section, the C2V update of MSA can be converted into finding the minimum and the sub-minimum values. We employ a comparison tree that is convenient for pipeline processing, and the structure is shown in Fig. 1. It is worth noting that the number of inputs to the comparison tree is a multiple of four. When the row degree does not meet the requirement, the remaining inputs to the comparison tree need to be initialized to the maximum in the range of quantization. When the number of inputs of the comparison tree is *X*, it takes $\lceil log_2 X \rceil$ clock cycles to get the output. The initiation interval (II) can be shortened to one clock cycle by using the *pipeline* directive in line 4, which means that M_b C2V update results can be output per clock cycle. Line 19 to 30 represent the V2C update. The variable *acd* denotes the accumulation of column degrees, i.e., $acd = \sum_{i=0}^{n_b} d_i$ $(n_b = 0, 1, \ldots, N_b - 1)$. Since there are *P* different column degrees, there should be *P* different V2C update strategies accordingly. When performing an update, an update strategy is chosen based on the column degree. To enable updating N_b variable nodes in parallel, the unroll directive is utilized for the loop in line 20. The V2C update according to (2) is shown in Algorithm 2. The posterior probability is computed in line 3 and the value is used for hard decision. line 4 to 6 perform message update by subtracting

Algorithm 1 Basic Architecture of Proposed Decoder

Initialize according to $L_{v_j \to c_i} = C_{v_j}$.
Creat array $tpMes1_{M_b \times \max(dv)}$,$tpMes2_{M_b \times \max(dv)}$
for i from 0 to $Z-1$ **do**
  #pragma HLS pipeline
  for m_b from 0 to $M_b - 1$ **do**
    for d from 0 to $\max(dc-1)$ **do**
      $mp = matpos_{m_b,d}$
      $tpMes1_{m_b,d} = Mes_{mp,\ \mathrm{mod}\ (idx_{mp}+i,Z)}$
    end for
    $tpMes2_{m_b} = \mathrm{C2V}\,(tpMes1_{m_b})$
    for d from 0 to $\max(dc-1)$ **do**
      if $d < dc_{m_b}$ **then**
        $mp = matpos_{m_b,d}$
        $Mes_{mp,\ \mathrm{mod}\ (idx_{mp}+i,Z)} = tpMes2_{m_b,d}$
      end if
    end for
  end for
end for
$acd = 0$
for n_b from 0 to $N_b - 1$ **do**
  #pragma HLS unroll
  switch (dv_{n_b})
  case dv_1**:**
    $V2C_{dv1}\left(L_{n_b}, Out_{n_b}, Mes_{acd}, \cdots, Mes_{acd+dv_{n_b}-1}\right)$
    $\vdots$
  case dv_P**:**
    $V2C_{dvP}\left(L_{n_b}, Out_{n_b}, Mes_{acd}, \cdots, Mes_{acd+dv_{n_b}-1}\right)$
  end switch
  $accd = accd + dv_{n_b}$
end for
if the termination condition is not satisfied **then**
  Position 3
end if

Algorithm 2 V2C Update

#pragma HLS pipeline
for For k from 0 to $Z-1$ **do**
  $s = L_{n_b,k} + Mes_{acd,k} + \cdots + Mes_{acd+dv_{n_b}-1,k}$
  $Mes_{acd,k} = s - Mes_{acd,k}$
  $\vdots$
  $Mes_{acd+d_v-1,k} = s - Mes_{acd+dv_{n_b}-1,k}$
  $Out_{n_b,k} = 1/2(1 - sign(s))$
end for

irrelevant information.

It is worth noting that to enable simultaneous access, arrays *L*, *Out* and *Mes* should be partitioned completely in the first dimension, arrays $tmMes_1$ and $tmMes_2$ should be partitioned in all dimensions.

B. Update Strategy for Multiple Diagonal Submatrices

Some submatrices contain multiple diagonals, and storing the messages on these diagonals in the same space will limit the parallel access and cause performance bottlenecks. Therefore, the messages on different diagonals need to be stored in different arrays. Suppose that the submatrix containing the maximum number of diagonals in the PCM has *Q* diagonals, and the number of submatrices with at least q ($q = 1, 2, \cdots, Q$) diagonals is n_q. Consider the original PCM as a superposition of *Q* sub-PCMs, each of which has only one diagonal in NZSs. The decoding messages of the *q*th sub-PCM are stored in array $Mes(q)$ with size $n_q \times Z$, where the array n_q of length N_b denotes the column degree of the *q*th sub-PCM. The line 10 and line 24 to line 27 in Algorithm 1 should be modified accordingly as (3).

where idx_q, d_{vq}, $acdq$ denote the position of non-zero elements in the first row, column degree and accumulation column degree corresponding to *q*th diagonal, respectively. The only thing that changes is the number of parameters entered by the update module, the processing remains the same.

$$\begin{cases} C2V\left(Mes(1)_{t,idx_1}, Mes(2)_{t,idx_2}, \ldots, Mes(Q)_{t,idx_Q}\right) \\ V2C\left(L_{n_b}, \mathrm{Out}_{n_b}, Mes(1)_{acd_1}, \cdots, Mes(1)_{acd_1+dv(1)_{n_b}-1}, \ldots, Mes(Q)_{acd_Q}, \ldots, Mes(Q)_{acd_q+dv(Q)_{n_b}-1}\right) \end{cases} \tag{3}$$

IV. Implementation and Discussion

A. Implementation Results

To understand and evaluate the performance benefits of our design over existing solutions, we used the LDPC codes from 802.11n [1], 802.16e [2], and Consultative Committee for Space Data Systems (CCSDS) standards [3-4]. The NZSs in both 802.11n and 802.16e are single diagonal, and the other two codes contain double NZSs. The software version is Vivado HLS 2019.1. We targeted a Kintex-7 (xc7k325tffg900-2) FPGA part and target clock frequency is 330 MHz. We employ 6-bits quantization (one symbol bit, three integer bits and two fractional bits) to quantize the messages to balance the performance. Vivado HLS takes several seconds to several minutes on a notebook computer to generate LDPC decoders from the C++ model. We perform initialization and code word output with parallelism N_b, and use ping-pong operations (i.e., initialization, iterative decoding and code word output are performed in pipeline way) to improve implementation efficiency. The throughput (Γ) of the proposed decoder is consequently consistent with the following equation:

$$\Gamma = \frac{(N_b - M_b) \times Z \times f_{clk}}{(2 \times Z + \theta) \times iter} \quad (Mbps) \tag{4}$$

where f_{clk} denotes the operation frequency, θ denotes the initiation interval, and *iter* denotes the number of iterations. As described in previous sections, the throughput of the proposed

TABLE I
IMPLEMNATIONS AND PERFORMANCE COMPARISONS WITH RELATED WORKS

LDPC codes		I_{max}	Related Works							Proposed (Kintex-7, 330MHz)			
			Target	Type	f_{clk} (MHz)	Γ (Mbps)	BRAMS	LUTs	FFs	Γ (Mbps)	BRAMs	LUTs	FFs
802.11n	(648,324) Z=27	10	Virtex-7 [10]	HLS	330	34	22	5140	5928	178	88	26729	2797
		3	Spartan-6 [13]		123	27	56	9072	4272	594			
		6	Virtex-6 [5]	RTL	100	281	81	35668	-	297			
		5	Virtex-2 Pro [6]		194	77	3	19332	16184	356			
	(1944,972) Z=81	8	Kintex-7 [7]		200	608	102	33351	26945	235	86	25600	3028
		10	Stratix-5 [11]	HLS	240	21	*1349	$^{\Delta}$187000	-	189			
	(648,540) Z=27	10	Virtex-7 [10]		330	64	22	5175	5930	297	88	21643	19519
	(1944,1620) Z=81	10	Virtex-7 [10]		330	149	67	17047	17306	320	79		
802.16e	(1152,576) Z=48	10	Virtex-7 [10]		330	59	38	8178	10307	188	76	25621	15954
		10	Stratix-5 [12]		222	104	*1703	$^{\Delta}$161000	-				
	(1920,960) Z=80	10	Virtex-7 [10]		330	69	64	17819	17061	192	76	24974	14744
		10	Stratix-5 [12]		202	81	*1516	$^{\Delta}$144000	-				
CCSDS	(8176,7154) Z=511	15	Virtex-2 [8]	RTL	193	172	128	28229	28269	153	80	26926	29871
		14	Virtex-5 [9]		250	2000	132	56778	86942	164			

*:BRAM 20K, others are BRAM 18K. $^{\Delta}$: ALMs

TABLE II
IMPLEMNATIONS WITH THREE MODES

LDPC Code	Z	Iter.	Γ Mbps	BRAMs	LUTs	FFs
(128,64)	16					
(256,128)	32	10	57∼65	32	16731	5960
(512,256)	64					

decoder mainly depends on the lifting size *Z*, while M_b, N_b, row degree dc and column degree dv are closely related to the resource consumption and initiation interval. Hardware complexity and decoding performances are provided in Table I. As can be seen in Table I, we have decoding throughputs between 153 to 320 megabit per second (Mbps).

The results reported in Table I show the well balanced performance of the proposed decoder model when compare to similar works, i.e., it has higher throughput with similar resource consumption. In FPGA development, consuming multiple times of resources does not necessarily increase the throughput by the same times. Because the increase in resource consumption will increase the complexity of placement and routing, and result in lower clock frequencies. Therefore, the throughput of our designed decoder is increased by a factor of 2 to 5 compared to [10]. Although the resource usage is increased by a factor of 2 to 5 at the same time, it is still evident that our designed decoder has higher efficiency. In addition, our design is more capable of meeting the requirements of high speed data transmission. Compared to [11], throughput improves by a factor of 9 while LUTs usage is only 1/60 of it. Compared to [12], throughput improves by a factor of 2.3. Compared to [13], throughput improves by a factor of 22 while resource usage is only 5 times of it. As many practices have shown, HLS takes up greater logic utilization than RTL-based designs with similar performance [14]. The resource utilization in [8] is similar to ours but the throughput is almost 30% higher. the throughput in [7] is similar to ours but the resource utilization (LUT) is 20% less. However, our decoder still superior to other state-of-the-art HLS designs. In Table II, we use this architecture to implement three modes of an LDPC code whose throughput still matches the calculation in (3), which exhibits a degree of flexibility in our design.

B. Discussion

When designing decoders for different QC-LDPC codes using the design proposed in this paper, only some constant arrays *matpos*, *idx*, *dc* and *dv* and constant *Z*, M_b, N_b need to be modified. These constants can be generated using auxiliary programs. According to our practice, this modification process can be completed in several minutes, which reflects the generality of decoders that can be easily designed using HLS. Therefore, using HLS can greatly improve development efficiency. However, if the attempt is made to make the decoder support more modes (e.g., 12 modes of 802.11n at the same time) and not at the cost of reduced throughput, it is inevitable to completely partition many large arrays. Vivado HLS takes a lot of time (hours or even almost one day) to find the optimal storage strategy and scheduling strategy. Despite the cost of heavy labor, hand-coded RTL often avoids these problems. This shows that there is still a long way to go in the development of current HLS tools.

V. CONCLUSION

In this paper, we proposed a performance balanced LDPC decoder using Vivado HLS. The decoder takes advantage of the quasi-cyclic properties of the PCM of QC-LDPC codes and uses compressed storage and distributed storage strategies to reduce resource usage and avoid access conflicts. It also improves throughput by optimizing the architecture and increasing parallelism with optimized directives. In addition, the parameterized design improves the generality of the decoder. The implementation results show that the overall performance

of the decoder outperforms the state-of-the-art HLS based decoder and has great practical value.

REFERENCES

[1] IEEE P802.11 Wireless LANs WWiSE Proposal: High Throughout Extension to the 802.11 Standard, IEEE 11-04-0886-00000n, Aug.2004.

[2] IEEE Std 802.16e-2005 and IEEE Std 802.16-2004/Cor 1-2005(Amendment and Corrigendum to IEEE Std 802.16-2004), IEEE Std802.16e-2005, 2006.

[3] TM synchronization and channel coding. Recommended Standard (Blue Book), Issue 3, CCSDS 131.0-B-3, Washington, DC, USA, September 2017.

[4] Short Block Length LDPC Codes for TC Synchronization and Channel Coding. Experimental Specification. Issue 1. Recommendation for Space Data System Standards (Orange Book), CCSDS 231.1-O-1, Washington, D.C.: CCSDS, April 2015.

[5] S. A. Zied, A. T. Sayed, and R. Guindi, "Configurable low complexity decoder architecture for quasi-cyclic LDPC codes," in 2013 21st SoftCOM, Sept 2013, pp. 1–5.

[6] I. Tanyanon and S. Choomchuay, "A hardware design of MS/MMSbased LDPC decoder," in 2012 IEEE International Conference on Electron Devices and Solid State Circuit (EDSSC), Dec 2012.

[7] Mhaske S, Kee H, Ly T, et al. High-Throughput FPGA-Based QC-LDPC Decoder Architecture[C]// 2015 IEEE 82nd Vehicular Technology Conference (VTC Fall). IEEE, 2015.

[8] Wang Z, Cui Z. Low-Complexity High-Speed Decoder Design for Quasi-Cyclic LDPC Codes[J]. IEEE Transactions on Very Large Scale Integration Systems, 2007, 15(1):104-114.

[9] Xie T, Li B, Yang M, et al. Memory compact high-speed QC-LDPC decoder[C]// 2017 IEEE International Conference on Signal Processing, Communications and Computing (ICSPCC). IEEE, 2017.

[10] Delomier Y, Gal B L, Crenne J, et al. Model-based Design of Efficient LDPC Decoder Architectures[C]// 2018 IEEE 10th International Symposium on Turbo Codes and Iterative Information Processing (ISTC). IEEE, 2018.

[11] J. Andrade et al, "Combining flexibility with low power: Dataflow and wide-pipeline LDPC decoding engines in the Gbit/s era," in Proceedings of the ASAP Conference, June 2014, pp. 264–269.

[12] J. Andrade, G. Falcao, , and V. Silva, "Flexible design of wide-pipeline based WiMAX QC-LDPC decoder architectures on FPGAs using high level synthesis," IET Electronics Letters, 2014.

[13] E. Scheiber et al., "Implementation of an LDPC decoder for IEEE 802.11n using Vivado high-level synthesis," in Proc. of ESPCO, 2013.

[14] Andrade J , George N , Karras K , et al. Design Space Exploration of LDPC Decoders using High-Level Synthesis[J]. IEEE Access, 2017, PP(99):1-1.