Accelerating FPGA-based Emulation of Quasi-Cyclic LDPC Codes with Vector Processing

Xiaoheng Chen, Jingyu Kang, Shu Lin and Venkatesh Akella
Department of Electrical and Computer Engineering
University of California
Davis, California 95616, USA
Email: {xhchen, jykang, shulin, akella}@ucdavis.edu

***Abstract*—FPGAs are widely used for evaluating the error-floor performance of LDPC (low-density parity check) codes. We propose a scalable *vector* decoder for FPGA-based implementation of quasi-cyclic (QC) LDPC codes that takes advantage of the high bandwidth of the embedded memory blocks (called Block RAMs in a Xilinx FPGA) by packing multiple messages into the same word. We describe a vectorized overlapped message passing algorithm that results in 3.5X to 5.5X speedup over state-of-the-art FPGA implementations in literature.**

I. Introduction

LDPC codes, discovered by Gallager in 1962 [1], were rediscovered and shown to approach Shannon capacity in the late 1990s. Today LDPC codes are being considered for a wide variety of emerging applications such as high-density flash memory, satellite broadcasting and mobile WiMAX. Rapid performance evaluation of LDPC codes is very desirable to design better codes for these applications. For some applications (such as storage systems), it is necessary to prove the absence of error-floor down to 10^{-12} which is almost impractical on a conventional CPU.

As a result, FPGAs are being widely used to emulate the performance of LDPC codes. Zhang et. al [2], [3] and [4] proposed techniques to realize partially parallel architectures to implement LDPC decoders efficiently on an FPGA. As a result, FPGAs are being widely used to emulate the performance of LDPC codes. Recently, researchers at Berkeley [5], [6], [7] describe a Virtex-II Pro based hardware emulation platform LDPC decoders. Saunders et. al [8] merge multiple messages into a single SISO message to save total memory bits of FPGA. As described in the next section, the message passing algorithm [9] is often used to decode the LDPC codes. Variable-to-check messages and check-to-variable messages are computed by variable nodes and check nodes processing units and are passed along the edges in the Tanner graph. One iteration of message passing contains two steps: check node update and variable node update. The two steps can be overlapped to increase decoding throughput, which is referred to as overlapped message passing (OMP) [4], [10]. The conventional scheduling without overlapping the two steps is referred to as non-OMP method. In a FPGA-based implementation of the message passing algorithm (both OMP and non-OMP methods), the variable to check node and check node to variable node messages are stored in the embedded memory blocks, called Block RAMs in a Xilinx FPGA. We call these *scalar* decoders because a single message is stored in each Block RAM word.

Scalar decoders do not take the full advantage of the high bandwidth capability of Block RAMs in modern FPGA, especially given that typically messages do not require more than 5 or 6 bits of precision [2]. For example, in a Virtex-5 FPGA, Block RAMs can have 18, 36, or 72 bit wide ports, which means that a scalar implementation potentially waste a significant fraction of the available memory bandwidth. In this paper, we propose a technique called VMP (vector message passing) that treats the wide word in a Block RAM as a *short vector*, that holds multiple messages, which are loaded and stored simultaneously. VMP introduces new challenges for data packing and alignment which can be solved by taking advantage of the configurable logic resources of an FPGA. This allows us to build a *customized vector processor* for a given LDPC code and desired data precision to improve the decoding throughput over conventional scalar decoders. However, this improvement comes at the expense of additional logic resources (flip-flops and slices) to implement the additional functional units to support vector processing and the required data alignment and addressing. For large LDPC codes this might result in a given decoder not fitting inside an FPGA. So, a careful design space exploration is needed to select the appropriate vector length for VMP, to maximize the decoding throughput given the resource constraints of the FPGA available for emulation. We built a tool, called QCSyn, to facilitate the exploration of the design space of vector LDPC decoders on a given FPGA platform. QCSyn is used to derive the results described in this paper.

The rest of the paper is organized as follows. We start with an overview of the design, construction, and decoding of QC-LDPC codes using the iterative message-passing algorithm. This is followed by the details of the vector decoding architecture and the VMP algorithm. We conclude with the results of our implementation and directions for future work.

II. QC LDPC Codes

A. Code Construction

A (γ, ρ) binary quasi-cyclic (QC) LDPC code is defined as the null space of a sparse parity-check matrix $\mathbf{H}$ over $\mathtt{GF}(2)$ with the following structural properties: (1) each row has

978-3-9810801-5-5/DATE09 © 2009 EDAA

constant weight (regular) or multiple weights no greater than ρ (irregular). (2) each column has constant weight (regular) or multiple weights no greater than γ (irregular). (3) no two rows (or two columns) can have more than one position where they both have 1-components. (4) the $\mathbf{H}$ matrix is composed of $\gamma \times \rho$ block matrices showed below:

$$\mathbf{H} = \begin{bmatrix} \mathbf{A}_{0,0} & \mathbf{A}_{0,1} & \cdots & \mathbf{A}_{0,\rho-1} \\ \mathbf{A}_{1,0} & \mathbf{A}_{1,1} & \cdots & \mathbf{A}_{1,\rho-1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}_{\gamma-1,0} & \mathbf{A}_{\gamma-1,1} & \cdots & \mathbf{A}_{\gamma-1,\rho-1} \end{bmatrix}$$

where $\mathbf{A}_{i,j}$ refers to either a $m \times m$ all-zero matrix or a circulant permutation matrix (CPM) for $0 \leq i \leq \gamma - 1$ and $0 \leq j \leq \rho - 1$. A CPM is a circularly shifted identity matrix with certain offset. The size of the $\mathbf{H}$ matrix is $(\gamma m) \times (\rho m)$. The code rate is no less than $1 - (\gamma/\rho)$.

Binary QC LDPC codes are constructed based on algebraic and combinatorial tools, such as finite geometries, balanced incomplete block designs, Reed-Solomon codes with two information symbols, and finite fields [11]. The density of an $\mathbf{H}$ matrix can be reduced by replacing a set of CPMs by zero matrices. The replacement of CPMs with zero matrices is referred to as *masking* [11]. Masking results in a new array whose Tanner graph has fewer edges and hence has fewer short cycles and possibly larger girth.

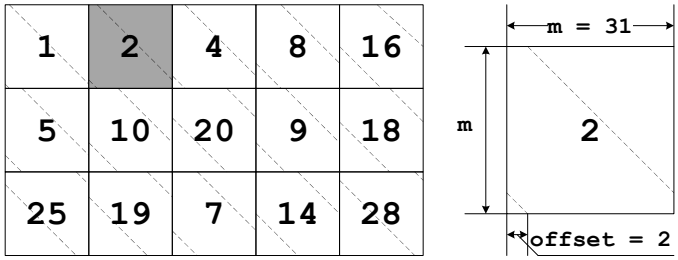

Fig. 1. Code structure of a (3,5) regular QC-LDPC code

Figure 1 shows the code structure of a $(3, 5)$ regular QC-LDPC codes. The $\mathbf{H}$ matrix has 3 block rows and 5 block columns. Each block matrix is a 31×31 CPM with certain offset displayed in the square symbol. The concept of CPM is further explained by a CPM with offset 2. The entries along the dashed line of the CPM are 1, while other entries are 0.

B. Message Passing Decoding

An LDPC code can be decoded under iterated message passing algorithm for Shannon capacity approaching error performance. The sum-product algorithm (SPA) and log-likelihood-ratio (LLR)-based SPA gives the best performance among soft-decision decoding. However, they require complex computations including nonlinear functions. An approximated version of LLR-SPA, called the min-sum algorithm (MSA) [12] requires less decoding complexity at the expense of some performance degradation. However, for LDPC codes with large check node degrees, the performance degradation of MSA compared to SPA could be as large as 1 dB. To address this problem, the normalized min-sum algorithm (NMSA) was introduced [12] and shown to be a better approximation of SPA. Besides, NMSA does not need any channel statistics. Therefore, we use NMSA for evaluating the performance of LDPC codes.

A QC-LDPC code defined by a $\mathbf{H}$ matrix size of $(\gamma m \times \rho m)$ has ρm bits per codeword and γm parity checks checking the code bits. In the Tanner graph of a QC LDPC code, each bit is represented by a variable node and each parity check is represented by a check node. An edge exists between the variable node v and the check node c if $h_{cv} = 1$ in the $\mathbf{H}$ matrix. Let the set $\mathcal{M}(v)$ denote the set of check nodes connecting the variable node v. Let the set $\mathcal{N}(c)$ denote the variable nodes connected to the check node c. Let $\mathbf{y} = (y_0, y_1, \cdots, y_{\rho m-1})$ be the soft-decision received sequence at the input of the decoder. Let $\mathbf{z} = (z_0, z_1, \cdots, z_{\rho m-1})$ be the posterior soft-decision information. At the i-th iteration, for $0 \leq c < \gamma m$ and $0 \leq v < \rho m$, let q_{vc} and σ_{cv} be the message from variable node v to check node c and the message from check node c to variable node v. In every iteration, the variable-to-check messages and the check-to-variable messages are updated. The normalized min sum algorithm is presented below.

STEP 1 Initialization: Set $i = 0$ and the maximum number of iterations to I_{max}. Set $q_{vc} = y_v$.

STEP 2 Check node update: at check node $0 \leq c \leq \gamma m - 1$, for $0 \leq v < \rho m - 1$, compute $\sigma_{cv}^{(i)}$ by

$$\sigma_{cv} = \alpha \cdot \min_{v' \in \mathcal{N}(c) \backslash v} |q_{cv'}| \cdot \prod_{v' \in \mathcal{N}(c) \backslash v} \text{sign}\left(q_{cv'}\right)$$

STEP 3 Variable node update: At variable node $0 \leq v < \rho m - 1$, for $0 \leq c \leq \gamma m - 1$, compute q_{vc} and z_v by

$$q_{vc} = y_v + \sum_{c' \in \mathcal{M}(v) \backslash c} \sigma_{c'v}$$

$$z_v = y_v + \sum_{c \in \mathcal{M}(v)} \sigma_{cv}$$

STEP 4 Tentative decode: If $\text{sign}\left(\mathbf{z}\right) \cdot \mathbf{H}^T = 0$ or I_{max} is reached, go to (5). Otherwise, $i \leftarrow i + 1$ and go to (2).

STEP 5 Termination: Take $\text{sign}\left(\mathbf{z}\right)$ as the decoded codeword and stop the decoding process.

III. Vector Decoder for QC LDPC Codes

A. Decoder Architecture Overview

In a FPGA-based partially parallel QC-LDPC codes decoder, the intrinsic messages ($\mathbf{y}$ in NMSA), extrinsic messages (q_{vc}, σ_{cv}) and hard decision bit ($\text{sign}(\mathbf{z})$ in NMSA) are grouped based on CPM locality and mapped to embedded memory (Block RAMs in Xilinx). A modulo counter is associated with each memory to generate the corresponding memory address, which always counts from certain initial value and then wraps around to the starting address. $(\gamma + 2)$ embedded memory are required for a regular (γ, ρ) QC-LDPC code decoder, among which $\gamma \times \rho$ memory are used for extrinsic messages, ρ memory are used for intrinsic messages, and ρ memory are used for hard decision bit. To differentiate from

the proposed vector decoder, we name refer to this as a scalar decoder. Given that the scalar decoder uses a memory word to store one message, only one message can be read out or written per memory access.

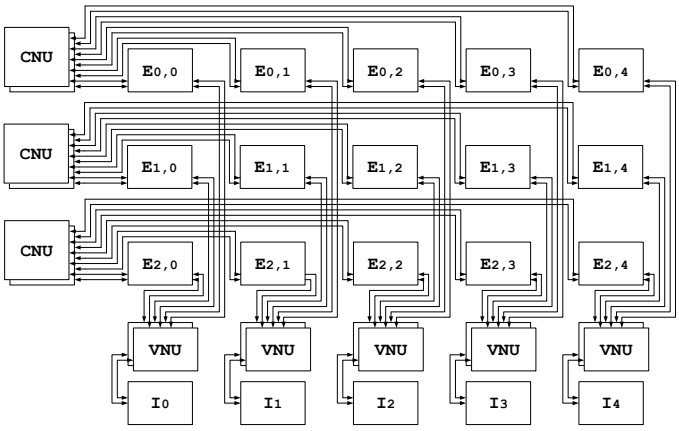

Fig. 2. A vector decoder for code depicted in figure 1 when $K = 2$

We exploit the wider bitwidth and configurability of embedded memory blocks to propose a vector decoder architecture. For example, in a Xilinx Virtex FPGA, a 18K Block RAM (BRAM) can be configured in different aspect ratios such as 8k 2-bit words or 512 36-bit words. For normalized min sum algorithm, the intrinsic and extrinsic message is usually 6-8 bit wide, thus up to six 6-bit messages can be packed in one memory word in the 512 36-bit aspect ratio. We define the number of messages packed into one memory word as K. Figure 2 shows a vector decoder for our example (3,5) code when $K = 2$. CNU denotes the check node unit, which perform the check node update and parity check (Step 2 and Step 4 in NMSA). VNU denotes the variable node unit, which perform the variable node update (Step 3 in NMSA). I denotes the intrinsic memory, which stores the intrinsic messages. E denotes the extrinsic memory, which stores the extrinsic messages and hard decision bit. The hard decision bit is computed by VNU and appended at the head of the variable-to-check messages. The vector decoder for a regular $(\gamma \times \rho)$ code requires $K \times \gamma$ CNUs, $K \times \rho$ VNUs, and $(\gamma + 1) \times \rho$ embedded memory. The decoding steps in NMSA are mapped to the vector decoder architecture as below.

STEP 1 Initialization Load the received intrinsic messages to both the ρ intrinsic message memory I and the $\gamma \times \rho$ extrinsic message memory E. $K \times \rho$ messages are load at one clock cycle.

STEP 2 Check node update and tentative decode: The $K \times \gamma$ CNUs read the variable-to-check messages and update the check-to-variable messages simultaneously. Each CNU read ρ messages from ρ extrinsic memory of the same block row. For example, the 1st and the 2nd CNUs read messages from and update messages to extrinsic memory $E_{0,0}, E_{0,1}, E_{0,2}, E_{0,3}, E_{0,4}$ in Figure 2. Besides, the CNUs extract hard decision bits from the variable-to-check messages and check the parity.

STEP 3 Variable node update: The $K \times \rho$ variable node units (VNUs) read the intrinsic messages and the check-to-variable messages, and update the variable-to-check messages simultaneously. Each VNU read γ messages from γ extrinsic memory of the same block column. For instance, the 3rd and the 4th VNUs read messages from $I_1, E_{0,1}, E_{1,1}, E_{2,1}$ and update the corresponding messages in Figure 2.

STEP 4 Termination: The hard decision bits are extracted from the extrinsic memory at the same block row and stored to data sink.

The vector decoder architecture outperforms the scalar decoder in two ways. First, the embedded memory port can be configured wide to read and write multiple message per port access without sacrificing clock frequency, thus we can exploit higher memory bandwidth for throughput gain by incorporating K-times processing units. Second, the vector decoder embeds the hard decision bits in the variable-to-check messages, and thus save ρ embedded memory and routing resources.

B. Message Packing and Alignment

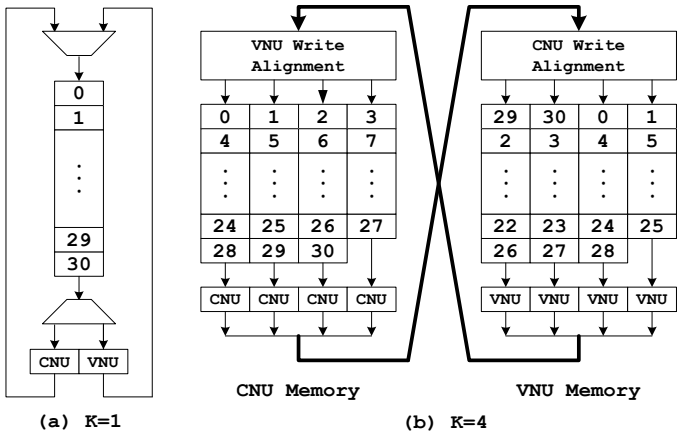

Fig. 3. Message packing when $K = 1$ and $K = 4$ for CPM depicted in figure 1

In the scalar decoder, which is same as a vector decoder with $K = 1$, a message is packed in one memory word. The memory structure can be modeled as a one dimensional array $L[m]$, as shown in Figure 3(a). For simplicity, all the indices, addresses, and pointers are assumed to start at 0. The message is packed in the row major order of the CPM, i.e., the message in the i-th CPM row is packed in $L[i]$. The location for message $L[i]$ stores σ_{cv} at step (2) and q_{vc} at step (3), i.e., the extrinsic messages are updated in place. For standard decoding scheduling, the CNU starts from the 0-th CPM row, i.e., in the order of $L[0], L[1], \cdots, L[m-1]$. The VNU starts from the 0-th CPM column, i.e., in the order of $L[m - \text{offset}], \cdots, L[m-1], L[0], \cdots, L[m - \text{offset} - 1]$.

For the vector decoder architecture, $K > 1$ messages are packed in one memory word. We use double buffering technique to eliminate memory conflict for CNU and VNU access. The extrinsic memory is divided into two parts: one for CNU to read out and VNU to write into (denoted as CNU memory), another for VNU to read out and CNU to write into

(denoted as VNU memory). Although double memory is being used, it *does not* change the number of Block RAMs used, which is important because the key limitation to the size of the LDPC code that can be accommodated on a given FPGA is limited by the *number* of Block RAMs not the memory capacity.

When $K > 1$, each CNU or VNU memory has $\lceil m/K \rceil$ memory words and K messages per word. The CNU and VNU memory can be modeled as two dimensional arrays, denoted by L_c and L_v. Figure 3(b) shows the packing details for a CPM with $m = 31$ and offset 2 when $K = 4$. The variable-to-check messages are packed in the CNU memory by the CNU access order, i.e., the message $L[0]$ is packed as $L_c[0][0]$. The check-to-variable messages are packed in the VNU memory by the VNU access order, i.e., the message $L[m - \text{offset}]$ is packed as $L_v[0][0]$. Generally, the message $L[k]$ is packed to the location $L_c[\lfloor k/K \rfloor][k \bmod K]$ in the CNU memory and $L_v[\lfloor((k+\text{offset}) \bmod m)/K\rfloor][((k+\text{offset}) \bmod m) \bmod K]$ in the VNU memory.

As $L[k]$ is mapped to different locations in the CNU memory and VNU memory except when offset $= 0$, the messages need to be aligned before they are written into the memory. As shown in Figure 3(b), messages $L[0], L[1], L[2], L[3]$ are read out and processed by four CNUs simultaneously. However, $L[0], L[1]$ and $L[2], L[3]$ are aligned by *CNU Write Alignment* unit and written into different memory words in the VNU memory. Besides, when $L[0]$ is not mapped to the 0^{th} column of the VNU memory, or $L[m-\text{offset}]$ is not packed in the 0^{th} column of the CNU memory, the starting row for write will be accessed twice. For instance, the 0^{th} row of the VNU memory in Figure 3(b) will be accessed twice in the check node update step, since $L[29], L[30]$ and $L[0], L[1]$ are packed into the same row. The *CNU Write Alignment* and *VNU Write Alignment* units are implemented using registers and multiplexors.

C. Vector Message Passing

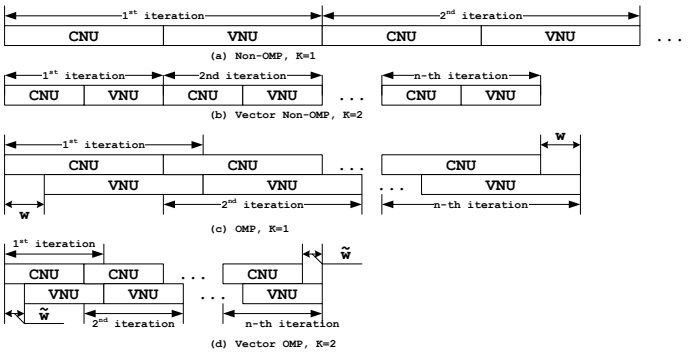

Fig. 4. Comparison of Scheduling Schemes

Overlapped message passing (OMP) method was proposed by [4] and further optimized by [10] to reach the maximized concurrency and avoid memory access conflicts. As showed in Figure 4(a,b), the non-OMP-based vector decoder reduces sub-iteration time by vector processing, and thus increases throughput. Based on the OMP method, we propose a vector overlapped message passing (VMP) scheme to exploit the vector processing architecture. Let $\tilde{w}$ denote the number of waiting clock cycles between CNU update and VNU update of the same iteration. We minimize $\tilde{w}$ as below.

(1) Apply the OMP method: Let $\mathbf{c} = \{c_0, c_1, \cdots, c_{\gamma-1}\}$ denote the starting CPM rows for CNUs. Let $\mathbf{v} = \{v_0, v_1, \cdots, v_{\rho-1}\}$ denote the starting CPM columns for VNUs. Let w denote the waiting time between intra-iteration CNU and VNU computations. We apply the OMP method described in [10] to get $\mathbf{c}$, $\mathbf{v}$, and w.

(2) Message packing: We use the same memory model and notation system described in the previous subsection to explain the packing scheme. For an extrinsic memory $E_{i,j}$, the CNU starts from the CPM row c_i, and the VNU starts from the CPM column v_j. The variable-to-check messages are packed in the CNU memory by the CNU access order, i.e., the message $L[c_i]$ is packed as $L_c[0][0]$. The check-to-variable messages are packed in the VNU memory by the VNU access order, i.e., the message $L[v_j - \text{offset} \bmod m]$ is packed as $L_v[0][0]$. Generally, the message $L[k]$ is packed to the location $L_c[\lfloor(k - c_i)/K\rfloor][(k - c_i) \bmod K]$ in the CNU memory and to $L_v[\lfloor((k - v_j + \text{offset}) \bmod m)/K\rfloor][((k - v_j + \text{offset}) \bmod m) \bmod K]$ in the VNU memory.

(3) Compute the starting read and write word address at the 1st iteration. The starting CNU or VNU read address is 0. The starting write message for CNU is $L[c_i]$, which is packed in $L_v[x_c(i,j)][y_c(i,j)]$ in the VNU memory, where

$$\begin{aligned} x_c(i,j) &= \lfloor((c_i - v_j + \text{offset}) \bmod m)/K\rfloor \\ y_c(i,j) &= ((c_i - v_j + \text{offset}) \bmod m) \bmod K \end{aligned}$$

Likewise, the starting write message for VNU is $L[v_j - \text{offset} \bmod m]$, which is packed in $L_c[x_v(i,j)][y_v(i,j)]$ in the CNU memory, where

$$\begin{aligned} x_v(i,j) &= \lfloor(v_j - c_i - \text{offset}) \bmod m)/K\rfloor \\ y_v(i,j) &= (v_j - c_i - \text{offset}) \bmod m) \bmod K \end{aligned}$$

(4) Compute the minimum waiting time for VMP by

$$\tilde{w} = \max_{0 \le i < \gamma, 0 \le j < \rho} \{\lceil m/K \rceil - x_c(i,j), \lceil m/K \rceil - x_v(i,j)\} + 1$$

(5) Compute the number of clock cycles for sub-iteration time by

$$\tilde{m} = \begin{cases} \lceil m/K \rceil & \text{if } \sum_{i=0}^{\gamma} \sum_{j=0}^{\rho} (y_c(i,j) + y_v(i,j)) = 0 \\ \lceil m/K \rceil + 1 & \text{otherwise} \end{cases}$$

The above steps (2-5) can be applied to vector decoders with non-OMP scheduling, for which $c_i = v_j = 0$. The starting read and write address for CNUs and VNUs increases cyclically with the increment of $\tilde{m} - 1$ for every new iteration for data dependency.

For OMP or VMP scheduling scheme, CNUs and VNUs may read and write the extrinsic memory at the same time, thus quad port memory is required. As the block RAM runs at the frequency as high as 500 MHz, which is much larger than twice the clock rate of the critical path within our decoder,

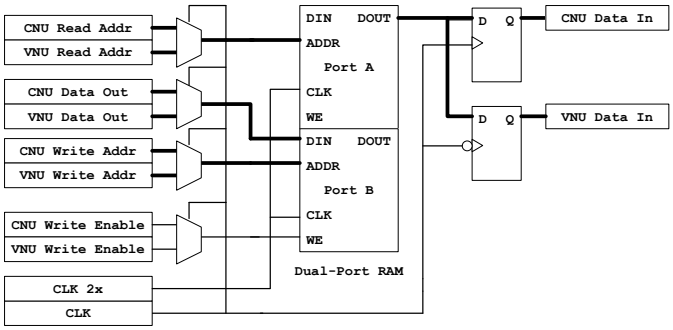

Fig. 5. Transform dual port memory to quad-port memory

we emulate quad-ported memory by clocking the memories at twice the clock frequency and time-multiplexing without frequency loss. As shown in figure 5, 'Port A' of the dual-port RAM emulates a dual-port read-only memory, while 'Port B' emulates a dual-port write-only memory. Hence we have two read ports and two write ports. The clock signal 'CLK 2x' is generated by the clock management unit within FPGA. The main clock signal 'CLK' is used as the select signal for multiplexors.

Based on the formula in [10], we derive a general formula for computing the throughput of the vector decoder as follows -

$$T_p = \begin{cases} \frac{\rho \times m \times f}{M+(M \times N_{iter}+W)} & \text{if } W \leq \lfloor M/2 \rfloor \\ \frac{\rho \times m \times f}{M+(W \times (2N_{iter}-1)+M)} & \text{if } W > \lfloor M/2 \rfloor \end{cases}$$

where f is the clock frequency, N_{iter} is the average iteration number, W is the number of clock cycles for intra-iteration wait, and M is the number of clock cycles for loading or updating messages. The denominator denotes the number of clock cycles to decode a codeword, and contains two parts: overlapped load/store time M, and iteration time. The formula applies to four cases: (1) scalar decoder with non-OMP scheduling, $W = M = m$. (2) scalar decoder with OMP scheduling, $W = w$, $M = m$. (3) vector decoder with non-OMP scheduling, $W = M = \tilde{m}$. (4) vector decoder with OMP scheduling, $W = \tilde{w}$, $M = \tilde{m}$.

IV. Results and Discussion

We have developed and implemented a tool, called QCSyn, in python to automate the creation of vector decoder architecture. The QCSyn tool takes the description of a regular or irregular QC-LDPC code and the desired vector width (K) writes synthesizable decoder description file in verilog. We use Xilinx 10.1 for synthesis and implementation. ModelSim 6.3f is used for simulation.

First, we implement a regular (3,6) QC LDPC decoder for a code with $m = 256$ using the same setting as [10]: Xilinx Virtex 2 XC2V6000-5 FPGA, 8-bit quantization scheme, and min-sum algorithm to benchmark our implementation against the best known technique from literature. Table I shows that although the decoder in [10] uses a deeper pipeline and thus has slightly higher clock rate, our vector architecture has throughput gain up to 5.5 when $K = 4$. Furthermore, our implementation uses only 24 Block RAMs instead of 36 BRAMs used by the implementation in [10]. This is because of our optimization of embedding hard decision bit in the extrinsic messages.

Method	Lehigh [10]	K=1	K=2	K=3	K=4
Slices	1616	1784	3256	6354	6828
Flip Flops	1073	1613	2996	6367	6828
4-input LUTs	2887	3026	5185	9716	10911
BRAMs	36	24	24	24	24
f (MHz)	148.7	116.9	109.9	99.6	98.4
N_{iter} at 4.5 dB	3	3	3	3	3
W	63	63	33	22	17
M	256	256	129	87	65
T_p (Mbps)	99.1	165.2	307.5	413.5	545.6
T_p Gain	1	1.67	3.1	4.17	5.5

TABLE I
Regular (3,6) code with $m = 256$

Next, we evaluate our technique on a regular and an irregular QC-LDPC code of moderate code length which is representative of codes being considered for storage and wireless communications. $(8192, 7171)$ code is a $(\gamma = 4, \rho = 128, m = 256)$ regular code, which is constructed on the prime field $\mathrm{GF}(257)$ [11] where as $(3369, 3213)$ is a $(\gamma = 12, \rho = 63, m = 63)$ irregular code, which is constructed by masking the $\mathrm{GF}(2^6)$-based base matrix. We use NMSA ($\alpha = 0.5$) and 6-bit quantization scheme to design the decoder when $K = 1, 2, 3, 4$. The synthesis result of $(8192, 7171)$ code using Xilinx Virtex 4 XC4LX160-10 FPGA is listed in the Table II. The synthesis result of $(3369, 3213)$ code using Xilinx Virtex 5 XC5VSX240T-2 FPGA is listed in the Table III. (Different FPGAs were used because Virtex 4 does not have sufficient number of Block RAMs for the more complex irregular code). The results indicate that the decoding performance increases linearly with vector length (K). The logic resources increase proportionally to support the additional functional units and alignment logic. However, with each successive generation of FPGAs, the number of resources in terms of slices, flip-flops and LUTs is increased significantly, so the proposed technique can be used to scale the decoding performance as bigger FPGAs become available. QCSyn makes this effortless by automating the selection of the appropriate vector length given a code and a target FPGA.

K	1	2	3	4
Slices	6926	13282	26465	28603
Flip Flops	3945	9005	21993	19722
4 input LUTs	12813	24419	48388	52811
BRAMs	160	160	160	160
f (MHz)	92.6	92.2	93.2	92.8
N_{iter}	10	10	10	10
W	150	76	51	38
M	256	129	87	65
T_p (Mbps)	225.6	443.8	668.0	892.3
T_p Gain	1	1.97	2.96	3.96

TABLE II
Regular (8192,7171) codes results

K	1	2	3	4
Slices	44047	65942	76766	103367
Flip Flops	23511	33059	36974	46926
6 input LUTs	42270	60968	71154	95319
BRAMs	134	134	134	134
f (MHz)	124.2	124.5	124.5	124.5
N_{iter}	10	10	10	10
M	63	33	22	17
W	63	33	22	17
T_p (Gbps)	0.332	0.636	0.953	1.234
T_p Gain	1	1.91	2.87	3.71

TABLE III
IRREGULAR (3369,3213) CODES RESULTS.

V. CONCLUSIONS AND FUTURE WORK

We have developed a vector decoding architecture and vector message passing algorithm to take advantage of the configurable Block RAMs available in modern FPGAs to improve the throughput of QC-LDPC code. We have showed that the proposed implementation can improve the throughput by a factor of 3.5X to 5.5X for a variety of codes of moderate to high complexity for regular and irregular QC-LDPC codes. We are developing techniques to virtualize the Block RAMs to handle arbitrary length QC-LDPC codes.

Acknowledgement: This research was supported by NSF under the grants of CCF-0429154 and CCF-0727478.

REFERENCES

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

[2] T. Zhang and K. Parhi, "A 54 MBPS (3, 6)-regular FPGA LDPC decoder," *IEEE Proc. of SIPS*, pp. 127–132, 2002.

[3] T. Zhang, "Efficient VLSI Architectures for Error-Correcting Coding," Ph.D. dissertation, University of Minnesota.

[4] Y. Chen, K. Parhi, D. Center, T. Inc, and T. Dallas, "Overlapped message passing for quasi-cyclic low-density parity check codes," *IEEE Trans. Circuits and Syst. I*, vol. 51, no. 6, pp. 1106–1113, 2004.

[5] Z. Zhang, L. Dolecek, B. Nikolic, V. Anantharam, and M. Wainwright, "Investigation of error floors of structured low-density parity-check codes by hardware emulation," *IEEE Proc. of Globecom*, 2006.

[6] E. Yeo, B. Nikolic, and V. Anantharam, "Architectures and Implementations of Low-Density Parity Check Decoding Algorithms," in *Proceedings of the IEEE Midwest Symposium on Circuits and Systems*, vol. 45, 2002.

[7] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "High throughput low-density parity-check decoder architectures," in *Global Telecommunications Conference.s IEEE*, vol. 5, 2001.

[8] P. Saunders and A. Fagan, "A High Speed, Low Memory FPGA Based LDPC Decoder Architecture for Quasi-Cyclic LDPC Codes," in *IEEE Proc. of FPL*, 2006, pp. 1–6.

[9] J. Chen and M. Fossorier, "Near optimum universal belief propagation based decoding of low-density parity check codes," *IEEE Trans. Commun.*, vol. 50, no. 3, pp. 406–414, 2002.

[10] Y. Dai, Z. Yan, and N. Chen, "Optimal Overlapped Message Passing Decoding of Quasi-Cyclic LDPC Codes," *IEEE Trans. on VLSI Syst.*, vol. 16, no. 5, pp. 565–578, 2008.

[11] L. Lan, L. Zeng, Y. Tai, S. Lin, and K. Abdel-Ghaffar, "Constructions of quasi-cyclic LDPC codes for the AWGN and binary erasure channels based on finite fields and affine mappings," *Proc. Int. Symp. Info. Theory, ISIT*, pp. 2285–2289, 2005.

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