HARDWARE-BASED LINEAR PROGRAMMING DECODING VIA THE ALTERNATING DIRECTION METHOD OF MULTIPLIERS

Mitchell Wasson, Mario Milicevic, Stark C. Draper, Glenn Gulak

University of Toronto
Edward S. Rogers Sr. Department of Electrical and Computer Engineering
m.wasson@mail.utoronto.ca, {mario.milicevic, stark.draper}@utoronto.ca, gulak@eecg.toronto.edu

ABSTRACT

We detail a field-programmable gate array (FPGA) based implementation of linear programming (LP) decoding. LP decoding frames error correction as an optimization problem. This is in contrast to variants of belief propagation (BP) that view error correction as a problem of graphical inference. LP decoding, when implemented with standard LP solvers, does not easily scale to the blocklengths of modern error-correction codes. This is the main challenge we surmount in this paper. In earlier work we demonstrated how to draw on decomposition methods from optimization theory to build an LP decoding solver competitive with BP, in terms of both performance and speed, but only in double-precision floating point. In this paper we translate the novel computational primitives of our new LP decoding technique into fixed-point. Using our FPGA implementation, we demonstrate that error-rate performance very close to double-precision is possible with 10-bit fixed-point messages.

Index Terms— Linear programming decoding, alternating direction method of multipliers, field-programmable gate arrays, low-density parity-check codes, lowering error floors

1. INTRODUCTION AND BACKGROUND

In the early 2000's, Feldman et al. [1, 2] realized that maximum likelihood (ML) decoding of a binary linear code $\mathcal{C}$ is accomplished by the integer program $\min_{x\in\mathcal{C}} \gamma^\top x$. Here γ is the length-n vector of log-likelihood ratios (LLR) defined component-wise by $\gamma_i = \log\left(\frac{p(y_i|x_i=0)}{p(y_i|x_i=1)}\right)$ where y_i is the i^{th} channel output symbol. By applying the "parity polytope" relaxation to the code's check constraints, one obtains a linear program (LP). For a regular low-density parity-check (LDPC) code of check degree d, the vector of variables neighboring a check must be an even-weight binary vector. In the relaxation, neighboring variables can be some convex combination of even-weight binary vectors. The convex hull of even-weight binary vectors is called the parity polytope, denoted $\mathbb{PP}_d$. In this paper, we assume (t, d)-regular codes for notational simplicity and space. However, this work is easily generalized to irregular codes [3].

LP decoding generated much excitement, but standard LP solvers do not scale to the blocklengths of modern error-correcting codes. A number of attempts to build low-complexity LP decoders followed, either through "softening" [4, 5] or solving a sequence of simpler LPs [6]. Barman et al. developed an application-specific LP decoder that is computationally competitive with BP and has the same message-passing schedule as BP [7]. The LP decoding problem was solved in [7] by applying the alternating direction method of multipliers (ADMM), a decomposition technique from large-scale optimization [8], which further enabled the study of LP decoding performance for long blocklengths. It was then observed empirically, and later confirmed theoretically, that LP decoders outperform BP in the high-SNR regime [7, 9, 10].

To improve ADMM-LP decoding further, Liu and Draper augmented the linear objective with a penalty term designed to discourage fractional solutions or "pseudocodewords" [11]. Additionally, centering ADMM-LP about the origin via a simple variable substitution removes some asymmetries [3]. We let $x_{\mathcal{N}_j}$ be the length-d vector containing the components of x connected check j. Then the centered and ℓ_1-penalized ADMM-LP optimization problem is

$$\begin{aligned} \min \quad & \gamma^\top x - \alpha\|x\|_1 \\ \text{subject to} \quad & z_j = x_{\mathcal{N}_j} \text{ and } z_j \in \mathbb{PP}_d^c \quad j = 1, \ldots, m \\ & x \in \left[-\tfrac{1}{2}, \tfrac{1}{2}\right]^n \end{aligned} \tag{1}$$

where $\alpha > 0$ is the penalty parameter. Because of the centering transformation, (1) uses the centered parity polytope $\mathbb{PP}_d^c$, obtained by subtracting the all-$\frac{1}{2}$ vector from every point in $\mathbb{PP}_d$. The z_j's are replica variable vectors used to enforce the parity polytope constraints in check updates. An important difference from BP is that ADMM-LP's check updates maintain an internal state. These states are dual variables that softly enforce the $z_j = x_{\mathcal{N}_j}$ constraints.

There has been interest in moving ADMM-LP toward a hardware implementation. Several groups have made progress in creating an efficient algorithm for performing Euclidean projection onto the parity polytope [12, 13, 14], the main computational primitive of ADMM-LP decoding. In particular, Wasson and Draper investigated mapping this operation to hardware [14]. Additionally, several implementation papers have considered ADMM-LP decoding. Debbabi et al. investigated how to more efficiently schedule messages and developed a multicore implementation [15, 16]. Jiao et al. modified the penalization scheme to better error-rate performance [17]. Finally, Wei et al. implemented ADMM-LP avoiding parity polytope projection when possible [18].

While useful investigations, these studies do not demonstrate whether or not ADMM-LP decoding is viable in hardware. In this paper, we present an FPGA-based ADMM-LP decoder implementation. Through our RTL Verilog implementation, we show that frame error rate (FER) performance very close to double-precision BP and ADMM-LP is possible with a fixed-point hardware implementation. Additionally, we analyze resource usage. While our implementation can be synthesized for many codes [3], due to space constraints we focus on an ensemble of length-1002 (3, 6)-regular Quasi-Cyclic (QC) LDPC codes.

This work was supported by the National Science and Engineering Research Council (NSERC) of Canada, in part through a Discovery Research Grant, and by the National Science Foundation (NSF) under Grant CCF-1217058.

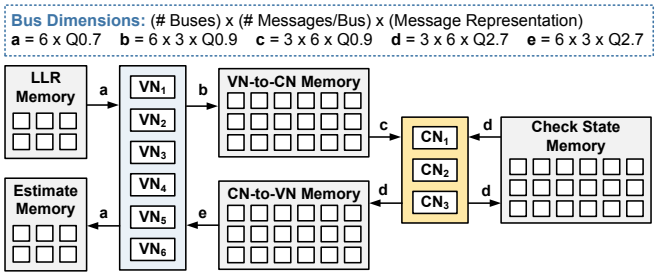

Fig. 1: Decoder architecture for a $(3, 6)$-regular QC-LDPC code.

2. IMPLEMENTATION

2.1. Decoder

Our goal is to develop a hardware proof-of-concept that can be used to study the error-correction performance of ADMM-LP. An FPGA-based platform provides a re-programmable and cost-effective solution. Given that ADMM-LP has the same message-passing structure as BP, we can draw upon existing hardware architectures. We implement a partially-parallel architecture to increase simulation speed, while adhering to the fixed resource constraints of an FPGA [19].

A central challenge in implementing hardware-based decoders is the scalability of the message-passing network. The network requires resource-intensive wiring and memory interconnect resources to pass messages between check node (CN) and variable node (VN) processing units. We restrict ourselves to QC codes [20, 21] in order to simplify message routing and memory interfacing. QC codes have parity-check matrices formed by tilings of $p \times p$ circulant matrices. The tilings naturally divide the parity-check matrix into $s = \frac{n}{p}$ "proto" columns and $r = \frac{m}{p}$ proto-rows.

The partially-parallel architecture combined with the QC code restriction allows us to minimize FPGA routing complexity by implementing the message-passing network with regularly-distributed FPGA block RAMs. Figure 1 presents an overview of our partially-parallel architecture. The architecture is comprised of multiple memory types to store LLRs, intermediate messages, and output codewords, as well as pipelined CN and VN processing units. The LLR and codeword estimate memories consists of s depth-p RAMs. The VN-to-CN message, CN-to-VN message, and check state memories each have a depth-p RAM for each circulant matrix in the code's parity check matrix. Multiple RAMs are used in each memory to facilitate simultaneous reads or writes from all VNs or CNs. In our implementation, VN and CN execution alternates until a maximum number of iterations is reached, without early termination. VNs and CNs are pipelined such that a VN or CN computation can start every clock cycle.

The LLRs, codeword estimates, and messages passed between VNs and CNs are signed fixed-point numbers implemented in the Q format [22]. Through experimentation, we found that 10-bit internal messages and 8-bit LLRs are required to obtain FER performance close to double-precision implementations [3]. Experimentation also showed that maximizing the number of LLR fraction bits provides the best FER performance [3]. This results in Q0.7 LLRs. As we will see, VN-to-CN messages and codeword estimates are guaranteed to be between $-\frac{1}{2}$ and $\frac{1}{2}$. Therefore, Q0.9 VN-to-CN messages and Q0.7 codeword estimates are used. CN-to-VN messages and check state values are implemented with Q2.7 since additional dynamic range is required to override LLRs in the VN addition computation.

2.2. Variable Node Compute Module

Figure 2 illustrates the architecture and execution of a VN. The LLR γ_i is first subtracted from the addition of all incoming CN-to-VN messages. To avoid overflow, we use a Q4.7 adder tree output, which is then penalized by adding α, 0, or $-\alpha$ based on the selection of a 3-to-1 multiplexer. Penalization pushes variable estimates farther in the direction of their current belief, thus discouraging pseudocodewords. Another integer bit is added here to avoid overflow. Note that the α in Fig. 2 is a normalized version of the α in (1). See [3] for the full algorithm derivation and discussion.

The next VN step is to normalize the penalized sum by the variable degree t. Division by t is performed by finding its reciprocal during synthesis and executing the normalization using a multiplier. A hard-wired DSP block is used to perform the multiplication unless t is a power of 2. To form the new variable estimate x_i, the above normalization must be projected onto the centered unit interval, which guarantees that the estimate is between $-\frac{1}{2}$ and $\frac{1}{2}$. Therefore a Q0.9 format is used. To form the variable estimate, it is crucial to *round* when discarding excess fraction bits, rather than *truncate*, since the downward bias of truncation results in different FER performance among codewords. The new variable estimate x_i is sent to *all* connected checks. This is in contrast to BP, where a different message is sent to each connected check.

The adder tree is the most resource intensive VN component, and scales as $\mathcal{O}(t)$ in area and $\mathcal{O}(\log t)$ in delay.

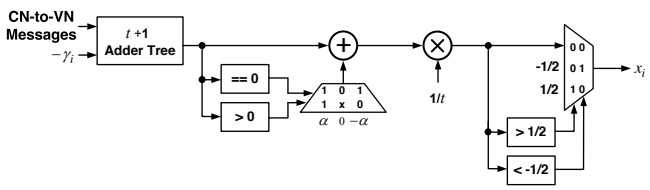

Fig. 2: Variable node compute module.

2.3. Check Node Compute Module

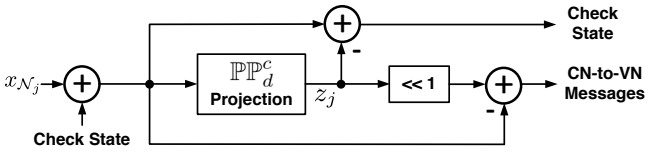

Fig. 3: Check node compute module.

Figure 3 illustrates the architecture and execution of a CN. Execution starts by performing a length-d vector addition with the check state vector and VN-to-CN messages $x_{\mathcal{N}_j}$. An extra integer bit is added to each component to prevent overflow.

The output of the vector addition is fed into the parity polytope projection module, which produces a new value for the variable replica vector z_j. The components of z_j are guaranteed to be between $-\frac{1}{2}$ and $\frac{1}{2}$, hence a Q0.12 format is used. At convergence, z_j will be equal to the incoming vector of VN-to-CN messages.

New check state and CN-to-VN messages are computed in parallel using vector additions from the parity polytope projection input and output. Pipeline registers store the parity polytope projection input while the projection is computed, and CN outputs are also rounded to avoid codeword bias.

Parity polytope projection dominates CN resource utilization with $\mathcal{O}\left(d(\log d)^2\right)$ area scaling (primarily from the storage of intermediate values), and $\mathcal{O}\left((\log d)^2\right)$ delay scaling.

2.4. Parity Polytope Projection

Our method for computing the Euclidean projection of a vector onto $\mathbb{PP}_d^c$ is depicted in Fig. 4. This method creates a projection onto the shell of $\mathbb{PP}_d^c$ and a projection onto the centered hypercube [14, 3]. If the unit hypercube projection is within the parity polytope, then that projection is chosen as the module output. Otherwise, the parity polytope shell projection is used. This 2-step procedure is followed because an efficient general parity polytope membership test is not known. However, with the assumption of unit hypercube membership, parity polytope membership is easily tested.

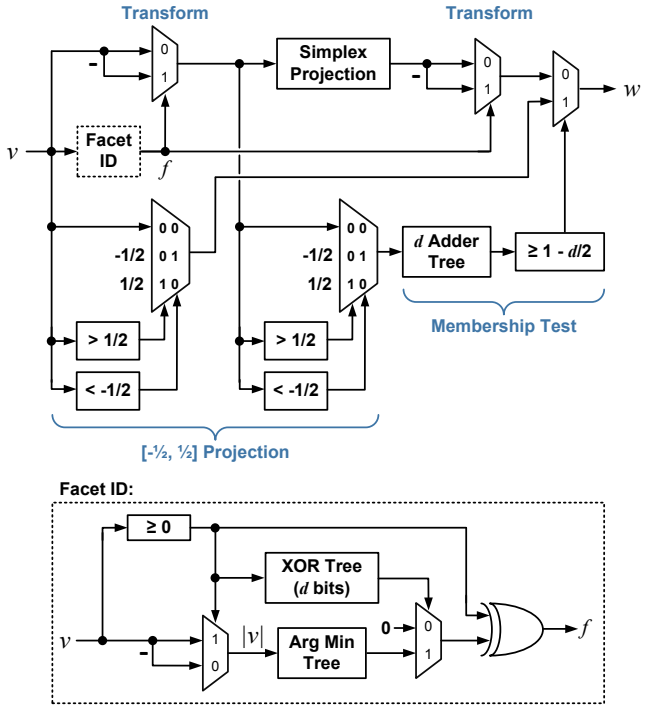

Fig. 4: Parity polytope projection. N.B., the computation block for facet identification in the upper figure is presented in the dotted box.

Projecting onto the shell of $\mathbb{PP}_d^c$ is accomplished via a projection onto the centered probability simplex $\mathbb{S}_d^c$. $\mathbb{S}_d^c$ is created by subtracting the all-$\frac{1}{2}$ vector from every point in the d-dimensional probability simplex. First, the relevant parity polytope facet is identified by checking the sign of each input vector component. Next, a similarity transform is performed to align the identified facet with $\mathbb{S}_d^c$, followed by the simplex projection. Finally, the transform is inverted to obtain the projection onto the shell of $\mathbb{PP}_d^c$. The unit hypercube projection is implemented by component-wise saturation at $-\frac{1}{2}$ and $\frac{1}{2}$.

The parity polytope membership test takes the transformed input, projects it onto the unit hypercube, and determines which side of $\mathbb{S}_d^c$ this point lies on. Alternatively, one could transform the input's unit hypercube projection and test which side of $\mathbb{S}_d^c$ it lies on.

Simplex projection is most resource intensive, scaling in area as $\mathcal{O}\left(d(\log d)^2\right)$ and $\mathcal{O}\left((\log d)^2\right)$ in delay. Again, storage of intermediate values in pipeline registers has a large impact on area usage.

2.5. Simplex Projection

Figure 5 presents our implementation of the probability simplex projection method developed by Duchi et al. [23] and modified as in [3] to project a point onto $\mathbb{S}_d^c$. The basic premise is to shift the point to be projected along the all-1 vector, and then to clip (saturate) components at a lower bound. The shift along the all-1 vector follows from geometry: we minimize the Euclidean norm and the all-1 vector is orthogonal to the simplex.

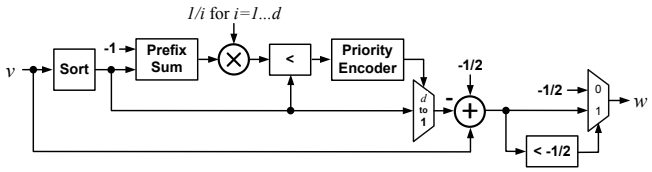

Fig. 5: Simplex projection.

This projection starts by sorting the input vector into descending order. Sorting networks, described in [24], accomplish sorting in hardware. Specifically, delay optimal networks from [24] are used for $d \leq 16$. Batcher's general merge sort construction can be used for larger dimensions [25, 14, 3].

Next, a prefix sum operation is performed using Ladner and Fischer's minimum delay construction [26]. Normalization by the number of sorted vector components in each prefix sum component follows. Similar to VNs, this is accomplished with multiplication.

The result of the normalization is a set of possible shift values to be subtracted from the input vector. The final shift value is chosen as the largest indexed component that is greater than the corresponding component in the sorted vector. Comparison operations produce a vector indicating which components satisfy this condition. The indicator vector is then fed into a priority encoder to produce a one-hot vector identifying the shift value we want. The shift is then subtracted from all components of the input vector, and vector components are clipped at $-\frac{1}{2}$ if necessary.

Batcher's sorting method has $\mathcal{O}\left(d(\log d)^2\right)$ area scaling and $\mathcal{O}\left((\log d)^2\right)$ delay scaling. As before, storage of intermediate values in pipeline registers has a large effect on area usage.

3. RESULTS

We perform experiments on an ensemble of five [1002, 503] QC-LDPC codes created by randomly generating "base" matrix [21] values while ensuring a minimum girth of six, using methods from [27]. An example 3×6 base matrix for one of these codes is [115 13 25 166 17 129; 124 38 137 13 160 136; 75 152 89 73 0 145].

Channel simulation was performed on the FPGA using a Gaussian random number generator [28]. The channel output must be saturated to produce LLRs in the decoder's input range. We found the optimal saturation value to be the channel input symbol magnitude (usually ± 1 for BI-AWGN model) plus one standard deviation of channel noise [3]. In all simulations, we set a maximum of 60 iterations. All data points correspond to 100 frame errors per code.

Figure 6 shows the FER performance averaged across all codes. Double-precision results were generated with Liu's code [29], also used in [7]. We can see that the fixed-point ADMM-LP implementation maintains performance close to the double-precision implementation. Furthermore, penalized ADMM-LP decoding has FER performance close to Butler and Siegel's non-saturating BP implementation [30] without displaying an error floor.

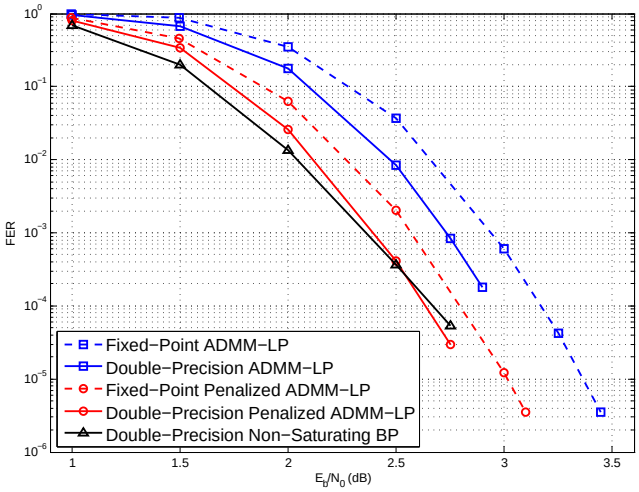

Fig. 6: Code ensemble performance on the BI-AWGN channel.

Table 1 presents a comparison of our FPGA-based implementation of ADMM-LP decoding to a min-sum decoder for a QC-LDPC code with the same code rate and comparable block length, also implemented with a partially-parallel architecture. We synthesized and implemented our decoder on an Altera Stratix V (model 5SGXEA7N2F45C2) FPGA. Based on our FPGA resource results presented in Table 1, CNs account for 85% of all Adaptive Logic Module (ALM) usage, VNs for 6%, and memory modules for 8%. Power estimates from Altera's power analyzer tool based on gate-level simulation report a total decoder power consumption of 863mW, with 797mW of dynamic power and 66mW of static power. When considering power consumption, CNs account for 76%, VNs for 6%, and memories for 17%. Each degree-6 CN uses 4,046 ALMs, 3 DSP blocks, and is 47 pipeline stages deep. Each degree-3 VN uses 140 ALMs, 1 DSP block, and is 9 pipeline stages deep. Inside a CN, 89% of ALM consumption and 94% of power consumption is due to parity polytope projection. Simplex projection accounts for 52% of CN ALM usage and 56% of CN power consumption. Finally, sorting accounts for 21% of both CN ALM usage and power.

4. DISCUSSION AND CONCLUSIONS

In this paper we demonstrate that ADMM-LP decoding can attain excellent error-rate performance in a fixed-point implementation. While our initial implementation requires higher fixed-point precision and more logic resources than min-sum, this study points to numerous possible avenues for future developments; developments that could bring ADMM-LP's resource requirements into line with those of other message passing decoders.

One direction to pursue is algorithmic simplification. Just as min-sum can be viewed as an computationally simple approximation of BP, we can seek approximations of ADMM-LP that preserve ADMM-LP's high-SNR performance. As one such example, in [3] it is observed that implementing partial-sort (rather than full-sort) can result in negligible performance reduction in error rates. As a second example, while we implemented hooks for early termination in our RTL Verilog code, we did not allow early termination in the presented results. A study of the effects of early termination would be quite useful.

Table 1: FPGA-based LDPC decoder comparison

	Chandrasetty [31] 2011	**This Work** 2016
Algorithm	Min-Sum	ADMM-LP
Architecture	Partially parallel	Partially parallel
Block Length	1152	1002
Code Design Rate	1/2	1/2
Code Structure	Quasi-cyclic	Quasi-cyclic
Number Iterations	10	60
BER Performance	1×10^{-5} at 3dB	2×10^{-7} at 3dB *
Target FPGA	Xilinx Virtex 2	Altera Stratix V
Message Width (Bits)	4	LLR: 8 Internal: 10
Early Termination	No	No
Clock Freq. (MHz)	64	224
Throughput (Mb/s)	50	8.52
Throughput Per Iter. (Mb/s/Iter)	5	0.142
Latency / Iter. (μs)	2.30	1.96
Power Est. (mW)	322	863
Logic Resources	2778 Slices	14315 ALMs
Memory (Kbits)	19.5 (29 BRAMs)	106.2 (47 BRAMs)
DSP Blocks	N/A	15

* N.B., the bit error rate (BER) presented here corresponds to that achieved by fixed-point penalized ADMM-LP, the FER of which is plotted in Fig. 6.

A second set of directions are hardware-centric. Numerous interesting challenges yet remain in the design of a hardware-efficient implementation of ADMM-LP. For example, it is not obvious how to implement a CN or a VN unit that can handle multiple node degrees. We believe that this problem can be solved through innovative hardware sharing or algorithmic generalization. As a second example, ADMM-LP also provides an opportunity for simplifying message-passing networks; especially when considering a fully-parallel architecture. This is because the same message is sent from each variable to all connected checks. Such message broadcasting can perhaps be exploited to reduce interconnect complexity. Finally, this study is a first step en-route to the development of a fully custom, in-silicon, application specific integrated circuit (ASIC). An ASIC would allow for high-performance, power-optimized register files and customized message passing resources that would yield significant performance improvements not possible in FPGA realizations.

Referring to Table 1, we point out that while our normalized throughput per iteration is $35\times$ lower than that of the min-sum decoder of [31], our ADMM-LP decoder achieves a bit-error rate (BER) nearly $100\times$ better. This is the crux of the matter. If one is concerned with applications where excellent performance in the high-SNR regime is required, a regime where algorithms such as min-sum or sum-product encounter error-floor problems, then ADMM-LP should be an algorithm of great interest. Our current implementation is already outperforming min-sum with less than an order of magnitude difference in the number of FPGA resources required. Further development, and innovation, could turn ADMM-LP into the algorithm of choice in such regimes of operation.

5. REFERENCES

[1] Jon Feldman, *Decoding Error-Correcting Codes via Linear Programming*, Ph.D. thesis, Massachusetts Institute of Technology, USA, 2003.

[2] Jon Feldman, Martin J. Wainwright, and David R. Karger, "Using linear programming to decode binary linear codes," *IEEE Trans. Inf. Theory*, vol. 51, no. 3, pp. 954–972, Mar. 2005.

[3] Mitchell Wasson, "Hardware-based linear program decoding with the alternating direction method of multipliers," M.S. thesis, University of Toronto, Canada, Aug. 2016.

[4] Pascal O. Vontobel and Ralf Koetter, "Towards low-complexity linear-programming decoding," in *Proc. 4th Int. Symp. Turbo Codes and Related Topics*, Munich, Germany, Apr. 2006, pp. 1–9.

[5] David Burshtein, "Iterative approximate linear programming decoding of LDPC codes with linear complexity," *IEEE Trans. Inf. Theory*, vol. 55, no. 11, pp. 4835–4859, Nov. 2009.

[6] M. H. Taghavi and Paul H. Siegel, "Adaptive methods for linear programming decoding," *IEEE Trans. Inf. Theory*, vol. 54, no. 12, pp. 5396–5410, Nov. 2008.

[7] Siddharth Barman, Xishuo Liu, Stark C. Draper, and Benjamin Recht, "Decomposition methods for large scale LP decoding," *IEEE Trans. Inf. Theory*, vol. 59, no. 12, pp. 7870–7886, Dec. 2013.

[8] S. Boyd, N. Parikh, E. Chu, B. Peleato, and J. Eckstein, "Distributed optimization and statistical learning via the alternating direction method of multipliers," *Foundations and Trends in Machine Learning*, vol. 3, no. 1, pp. 1–122, 2011.

[9] Xishuo Liu and Stark C. Draper, "Instanton search algorithm for the ADMM penalized decoder," in *Proc. Int. Symp. Inf. Theory*, Honolulu, June 2014.

[10] Xishuo Liu and Stark C. Draper, "ADMM decoding on trapping sets," in *Proc. Int. Symp. Inform. Theory*, Hong Kong, June 2015.

[11] Xishuo Liu and Stark C. Draper, "The ADMM penalized decoder for LDPC codes," *IEEE Trans. Inf. Theory*, vol. 62, no. 6, pp. 2966–2984, June 2016.

[12] Xiaojie Zhang and Paul H. Siegel, "Efficient iterative LP decoding of LDPC codes with alternating direction method of multipliers," in *Proc. IEEE Int. Symp. Inf. Theory*, Istanbul, Turkey, July 2013, pp. 1501–1505.

[13] Gouqiang Zhang, Richard Heusdens, and W. Bastiaan Kleijn, "Large scale LP decoding with low complexity," *IEEE Commun. Lett.*, vol. 17, no. 11, pp. 2152–2155, Nov. 2013.

[14] Mitchell Wasson and Stark C. Draper, "Hardware based projection onto the parity polytope and probability simplex," in *Proc. 49th Asilomar Conf. Signals, Systems, Computers*, Pacific Grove, CA, Nov. 2015, pp. 1015–1020.

[15] Imen Debbabi, Bertrand Le Gal, Nadia Khouja, Fethi Tlili, and Christophe Jego, "Fast converging ADMM-penalized algorithm for LDPC decoding," *IEEE Commun. Lett.*, vol. 20, no. 4, pp. 648–651, Apr. 2016.

[16] Imen Debbabi, Bertrand Le Gal, Nadia Khouja, Fethi Tlili, and Christophe Jego, "Analysis of ADMM-LP algorithm for LDPC decoding, a first step to hardware implementation," in *IEEE Int. Conf. Electronics, Circuits, and Systems*, Cairo, Egypt, Dec. 2015, pp. 356–359.

[17] Xiaopeng Jiao, Haoyuan Wei, Jianjun Mu, and Chao Chen, "Improved ADMM Penalized decoder for irregular low-density parity-check codes," *IEEE Commun. Lett.*, vol. 19, no. 6, pp. 913–916, June 2015.

[18] Haoyuan Wei, Xiaopeng Jiao, and Jianjun Mu, "Reduced-complexity linear programming decoding based on ADMM for LDPC codes," *IEEE Commun. Lett.*, vol. 19, no. 6, pp. 909–912, June 2015.

[19] Dale E. Hocevar, "A reduced complexity decoder architecture via layered decoding of LDPC codes," in *Proc. Work. Signal Proc. Systems*, Oct. 2004.

[20] Yu Kou, Shu Lin, and Marc P. C. Fossorier, "Low-density parity-check codes based on finite geometries: A rediscovery and new results," *IEEE Trans. Inf. Theory*, vol. 47, no. 7, pp. 2711–2736, Nov. 2001.

[21] Marc P. C. Fossorier, "Quasicyclic low-density parity-check codes from circulant permutation matrices," *IEEE Trans. Inf. Theory*, vol. 50, no. 8, pp. 1788–1793, Aug. 2004.

[22] Shoab Ahmed Khan, *Digital design of signal processing systems: a practical approach*, John Wiley & Sons, 2011.

[23] John Duchi, Shai Shalev-Shwartz, Yoram Singer, and Tushar Chandra, "Efficient projections onto the ℓ_1-ball for learning in high dimensions," in *Proc. Int. Conf. Machine Learning*, San Diego, USA, Dec. 2008.

[24] Donald E. Knuth, *The Art of Computer Programming: Sorting and Searching*, vol. 2, Addison Wesley Longman, Redwood City, CA, USA, 2 edition, 1998.

[25] Kenneth E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Computer Conf.* Apr. 1968, pp. 307–314, ACM.

[26] Richard E. Ladner and Michael J. Fischer, "Parallel prefix computation," *J. of the ACM*, vol. 27, no. 4, pp. 831–838, Oct. 1980.

[27] Yige Wang, Stark C. Draper, and Jonathan S. Yedidia, "Hierarchical and high-girth QC LDPC codes," *IEEE Trans. Inf. Theory*, vol. 59, no. 7, pp. 4553–4583, July 2013.

[28] Guangxi Liu, "Gaussian noise generator," webpage accessed Jan. 2016, http://opencores.org/project,gng.

[29] Xishuo Liu, "ADMM decoder," webpage accessed Jun. 2015, https://sites.google.com/site/xishuoliu/codes.

[30] Brian K. Butler and Paul H. Siegel, "Error floor approximation for LDPC codes in the AWGN channel," *IEEE Trans. Inf. Theory*, vol. 60, no. 12, pp. 7416–7441, Dec. 2014.

[31] Vikram A. Chandrasetty and Syed M. Aziz, "A multi-level hierarchical quasi-cyclic matrix for implementation of flexible partially-parallel ldpc decoders," in *2011 IEEE Int. Conf. Multimedia and Expo*, July 2011, pp. 1–7.