

# A Scalable Architecture of a Structured LDPC Decoder\*

Jason Kwok-San Lee, Benjamin Lee, Jeremy Thorpe,  
Kenneth Andrews, Sam Dolinar, Jon Hamkins<sup>†</sup>

{kwoklee, leeb, jeremy}@caltech.edu {andrews, sam, hamkins}@shannon.jpl.nasa.gov  
Jet Propulsion Laboratory, California Institute of Technology

## Abstract

We present a scalable decoding architecture for a certain class of structured LDPC codes. The codes are designed using a small  $(n, r)$  protograph that is replicated  $Z$  times to produce a decoding graph for a  $(Z \times n, Z \times r)$  code. Using this architecture, we have implemented a decoder for a (4096, 2048) LDPC code on a Xilinx Virtex-II 2000 FPGA, and achieved decoding speeds of 31 Mbps with 10 fixed iterations. The implemented message-passing algorithm uses an optimized 3-bit non-uniform quantizer that operates with 0.2dB implementation loss relative to a floating point decoder.

## 1 Introduction

Low-Density-Parity-Check (LDPC) codes[1] have recently received a lot of attention because of their excellent error-correcting capability. LDPC codes have been shown to be able to perform close to the Shannon limit[2]. They also can achieve very high throughput because of the parallel nature of their decoding algorithms. In the past decade or so, much of the research on LDPC codes has focused on the analysis and improvement of codes under decoding algorithms with floating point precision. However, to make LDPC codes practical in the real world, the design of an efficient hardware architecture is crucial.

## 2 Structured LDPC codes

Given unlimited hardware resources, a well-understood strategy is to allocate one processing element to each check and variable node in the Tanner graph[3] of an LDPC code. However, for the sake of error-correcting capability, it may be desirable to use a code with many more nodes than can be instantiated with limited hardware resources. To this end, we have developed an architecture for decoding structured LDPC codes in which computations are scheduled in space and time.

### 2.1 Protograph Construction

By "structured", it is meant that the code is constructed via a specific construction called a "protograph"[4]. The protograph is typically a small  $(n, r)$  graph that is used as a template for a large  $(Z \times n, Z \times r)$  code graph.

The code graph is constructed from the protograph by making  $Z$  copies of each variable and check node. Each edge in the small protograph represents a set of edges in the larger code graph which connect  $Z$  copies of a variable node with  $Z$  copies of a check node via an arbitrary permutation (see figure 1).

As a matter of terminology, although we use the protograph formalism in [4], other researchers have referred to a "projected graph"[5] or "base graph"[6], which are mostly functionally equivalent.

---

\*The work described was funded by the IND Technology Program and performed at the Jet Propulsion Laboratory, California Institute of Technology, under contract with the National Aeronautics and Space Administration.

<sup>†</sup>Communications Systems & Research Section, Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA.

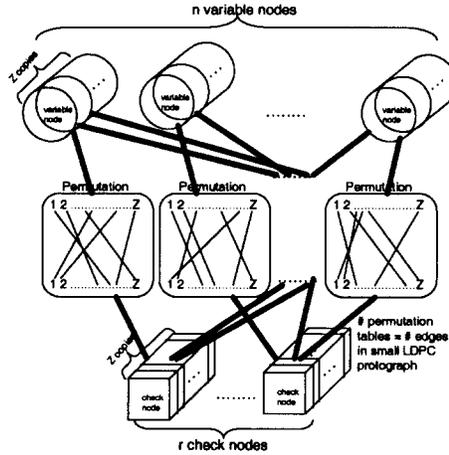


Figure 1: Protograph architecture

## 2.2 Decoder Architecture

The basic computation performed in message-passing decoding is a message-update, in which a node computes its set of outgoing messages from its set of incoming messages. In our hardware architecture, the processing elements are variable node units and check node units, each of which computes its respective message updates. These processing elements can be highly decentralized and distributed across the available area. Our strategy is to instantiate hardware units for each of the  $n$  variable nodes and  $r$  check nodes in the small LDPC protograph. All  $n$  variable node units or all  $r$  check node units decode synchronously and in parallel. The  $Z$  copies of the identical small protograph share this hardware and are operated on serially. The fundamental unit of time is called a "computation cycle", in which a processing element can read the incoming messages from a memory and compute and store outgoing messages. Messages are stored in memory modules, which each correspond to an edge in the small protograph. Each memory module consists of two memory banks capable of storing  $Z$  messages and a permutation table. One memory bank stores variable-to-check messages, and is writable by an associated variable node unit and readable by an associated check node unit. The other memory bank stores the check-to-variable messages and is writable by an associated check node unit and readable by an associated variable node unit. The permutation table specifies a permutation  $\pi_e : \{1, 2, \dots, Z\} \rightarrow \{1, 2, \dots, Z\}$  such that if  $\pi_e(i) = j$ , then the  $i^{\text{th}}$  variable node is connected to the  $j^{\text{th}}$  check node.

## 2.3 Computation Scheduling

The cornerstone of our hardware architecture is the scheduling of message-updates in space and time. One iteration consists of a check node phase, followed by a variable node phase. In each phase, there are  $Z$  computation cycles.

In the check node phase, all check node modules read messages from the edge memory in ascending order, update the messages, and write their results back to the edge memory in ascending order. This computation across all  $r$  check node units occurs in parallel.

In the variable node phase, all variable node modules read messages from the edge memory in permuted order, update the messages, and write back the edge memory in permuted order. The computation across all  $n$  variable node units also occurs in parallel. The decoding stops at the maximum iteration number, or when a stopping rule is satisfied.

Although this work was underway before the Flarion decoder patent was published, we can now make a useful comparison to that architecture. Flarion's design operates on all  $Z$  copies of the template LDPC graph in parallel and processes the individual nodes serially. In this manner, memory and processing can be centralized and a Single-Instruction-stream-Multiple-Data-stream (SIMD) instruction is used to access all  $Z$

messages[5].

In contrast, our system has multiple decentralized processing elements with multiple separate memories (see figure 2). All nodes in the template LDPC graph are operated on simultaneously in parallel and each of the  $Z$  copies are processed serially (see figure 3).

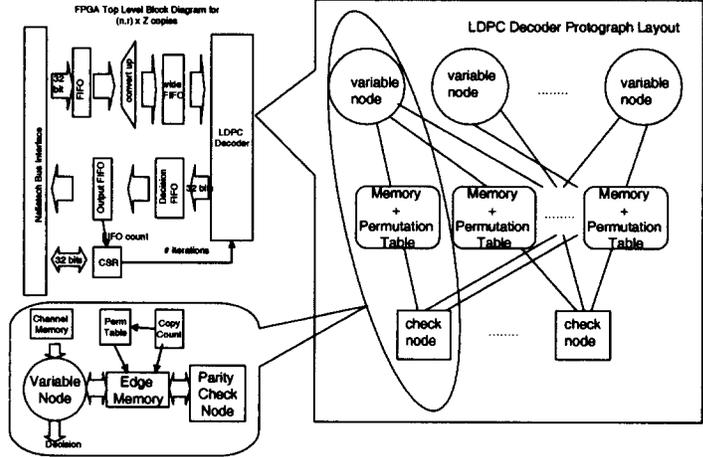


Figure 2: Our FPGA decoder architecture

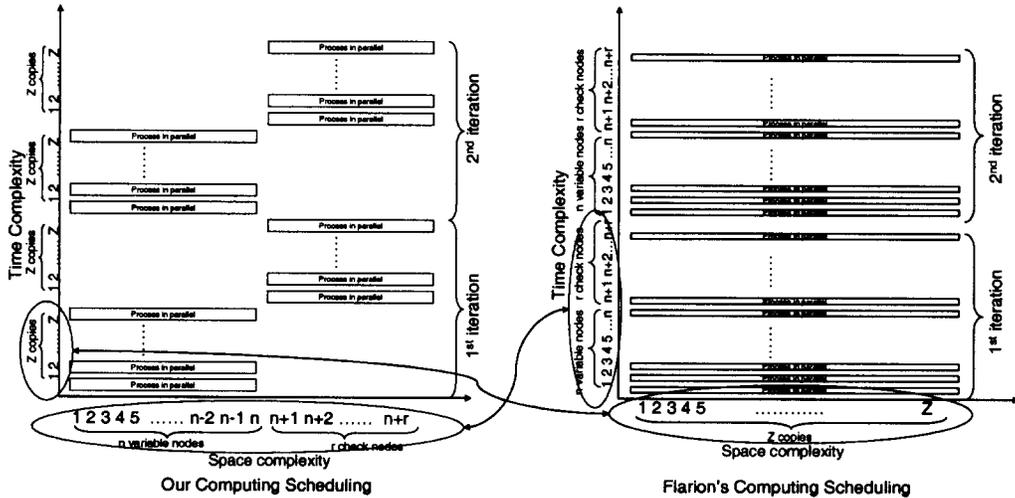


Figure 3: Our Computation Scheduling vs. Flarion's Computation Scheduling

## 2.4 Structured LDPC Implementation Methodology

1. Choose a small  $(n, r)$  protograph by some methods (e.g. [7]).
2. Replicate the protograph  $Z$  times and apply a "girth conditioning" algorithm such as Progressive-Edge-Growth (PEG)[8] to permute the end points of each set of edges to obtain a large  $(Z \times n, Z \times r)$  code graph that does not contain short cycles.
3. Generate a decoder design by applying the protograph and the chosen permutations to parameterized Verilog HDL

#### 4. Automatically synthesize, place and route design using Xilinx XST

Particular attention is given to the degree distribution of the small protograph chosen, as the larger code graph will have the same degree distribution. For all our protographs implemented, regular (3, 6) protograph was used.

### 3 Quantized Belief Propagation Algorithm

We use the non-uniform quantization scheme proposed in [9], which applies to regular (3, 6) LDPC codes.

Initially, variable nodes read the channel memory, compute initial variable-to-check messages  $v_{i \rightarrow j}(0)$ , and directly deposit into corresponding edge memory according to the permutation tables:

$$v_{i \rightarrow j}(0) = Q_{ch}(channel_i), i \in \{1..Zn\} \quad (1)$$

where  $Q_{ch}(channel_i)$  is the quantization rule for the channel.

At the  $t^{th}$  iteration, the parity check phase occurs first. All  $r$  check node units read the variable-to-check messages  $v_{i \rightarrow j}$  from edge memory connecting the  $i^{th}$  variable node to the  $j^{th}$  check node in the large code graph, update the message by equation (2), then write the check-to-variable messages  $u_{j \rightarrow i}$  back to the edge memory according to the permutation tables.  $r$  check node units are running in parallel, while  $Z$  copies of messages are being updated serially.

$$u_{j \rightarrow i}(t) = Q_c\left(\sum_{i'} \phi_c(v_{i' \rightarrow j}(t-1))\right), j \in \{1..Zr\} \quad (2)$$

where  $i'$  ranges over all edges connected to the  $j^{th}$  check node excluding  $i$ ,  $Q_c$  is the quantization rule for the check-to-variable message  $u_{j \rightarrow i}$ , and  $\phi_c$  is the reconstruction function for the variable-to-check message  $v_{i \rightarrow j}$ .

Next, the variable phase occurs.  $n$  variable node units read the check-to-variable messages  $u_{j \rightarrow i}$  from edge memory, update the message by equation (3), then write the variable-to-check messages  $v_{i \rightarrow j}$  back to edge memory according to the permutation tables.

$$v_{i \rightarrow j}(t) = Q_v\left(\phi_{ch}(Q_{ch}(channel_i)) + \sum_{j' \neq j} \phi_v(u_{j' \rightarrow i}(t))\right), i \in \{1..Zn\} \quad (3)$$

where  $j'$  ranges over all edges connected to the  $i^{th}$  variable node excluding  $j$ ,  $Q_v$  is the quantization rule for the variable-to-check message  $v_{i \rightarrow j}$ ,  $\phi_v$  is the reconstruction function for the check-to-variable message  $u_{j \rightarrow i}$ , and  $\phi_{ch}$  is the reconstruction function for the channel message  $Q_{ch}(channel_i)$ .

At the final  $K^{th}$  iteration, hard decisions  $X_i$  are made in variable nodes following:

$$X_i = \begin{cases} 0, & \sum_j u_{j \rightarrow i}(K) \geq 0 \\ 1, & \sum_j u_{j \rightarrow i}(K) < 0 \end{cases} \quad (4)$$

$x$	$\phi_{ch}(x)$	$\phi_v(x)$	$\phi_c(x)$
-4	-21	-20	-1
-3	-15	-12	-2
-2	-9	-6	-6
-1	-3	-2	-26
0	3	2	26
1	9	6	6
2	15	12	2
3	21	20	1

$Q_{ch}(ch)/$ $Q_v(v)/ Q_c(c)$	$ch$	$v$	$c$
-4	$ch < -3.3$	$v < -18$	$-5 \leq c < 0$
-3	$-3.3 \leq ch < -2.2$	$-18 \leq v < -12$	$-9 \leq c < -5$
-2	$-2.2 \leq ch < -1.1$	$-12 \leq v < -6$	$-26 \leq c < -9$
-1	$-1.1 \leq ch < 0$	$-6 \leq v < 0$	$c < -26$
0	$0 \leq ch \leq 1.1$	$0 \leq v \leq 6$	$c > 26$
1	$1.1 < ch \leq 2.2$	$6 < v \leq 12$	$9 < c \leq 26$
2	$2.2 < ch \leq 3.3$	$12 < v \leq 18$	$5 < c \leq 9$
3	$ch > 3.3$	$v > 18$	$0 \leq c \leq 5$

## 4 Performance

### 4.1 FPGA utilization

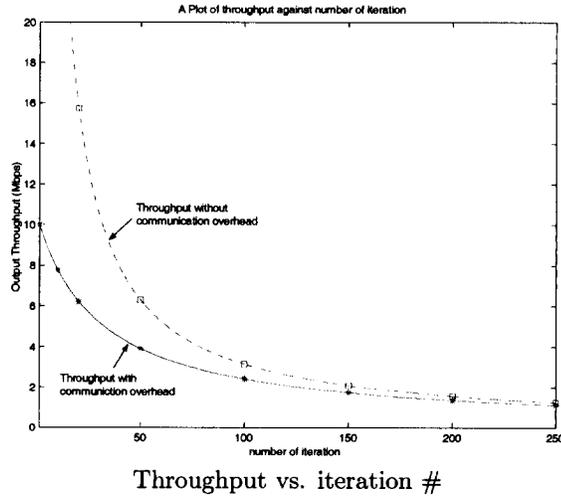
The performance of decoder can always be improved by increasing the block length. However, the block length of the LDPC code is limited by the area constraints on the FPGA chip. The LDPC decoder consists of processing units and edge memory. The area consumed by the processing units is proportional to the size of  $(n, r)$  protograph, which is proportional to the throughput. The area consumed by the edge memory is proportional to the size of  $(n \times Z, r \times Z)$  code graph, which is proportional to the error-correcting capability. We implemented several size of LDPC codes, and measured the utilization of the decoder on a Xilinx Virtex-II 2000 FPGA.

LDPC template $(n, r)$	Copies $Z$	Block length $(n \times Z, r \times Z)$	Slice Utilization %
(64, 32)	16	(1024, 512)	85
(64, 32)	32	(2048, 1024)	99
(32, 16)	32	(1024, 512)	53
(32, 16)	64	(2048, 1024)	66
(32, 16)	128	(4096, 2048)	97

### 4.2 Speed/Throughput

We measured the real decoding throughput by the FPGA decoder of a  $(128 \times 32, 128 \times 16)$  LDPC code at fixed iteration numbers without stopping rules.

# iteration	Throughput without communication overhead (Mbps)	Throughput with communication overhead (Mbps)
1	314.47	10.01
10	31.45	7.74
20	15.72	6.20
50	6.29	3.91
100	3.14	2.41
150	2.10	1.74
200	1.57	1.37
250	1.26	1.12



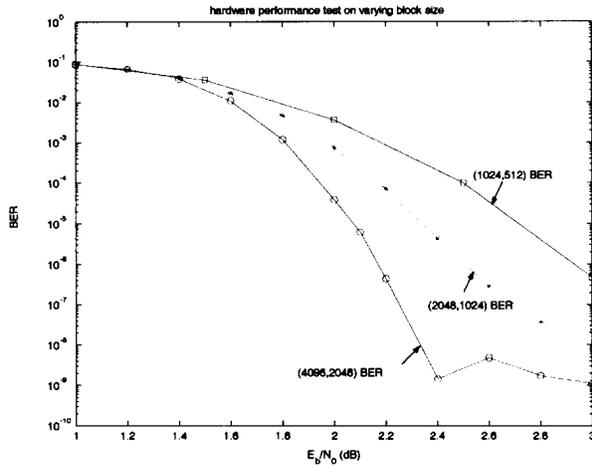
The measured delay consists of communication overhead and decoder latency, in which decoder latency is proportional to the number of iterations. The decoder latency is 3.18 ns/bit/iteration. The communication overhead is 97.1 ns/bit in our tests. Communication overhead includes the buffer delay outside decoder module, and the time delay writing to and reading from the FPGA board.

### 4.3 Error Correcting Capability

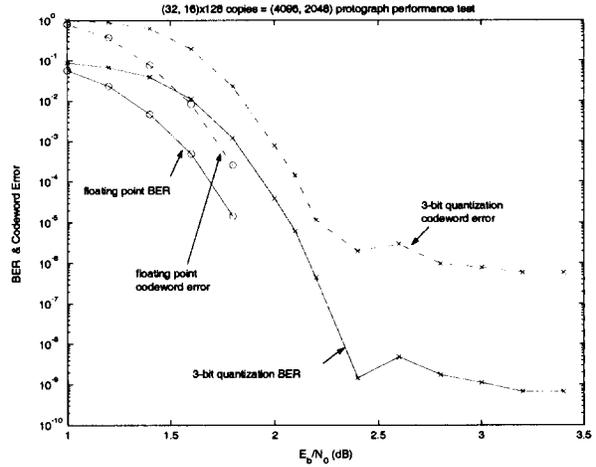
We implemented several codes of different block lengths, and ran performance tests to compare their performance differences. The largest block length code we can implement to fit into a Xilinx Virtex-II 2000 FPGA chip is a  $(32, 16) \times 128$  copies =  $(4096, 2048)$  code. The results demonstrate that doubling the block length can improve the performance by about 0.5 dB.

The performance of 3-bit non-uniform quantization is another interesting topic to investigate. Compared to the full floating point simulation done in software, hardware 3-bit non-uniform quantization is only off about 0.2 dB, with drastically smaller hardware implement requirements. The speed advantage of the FPGA

over software simulation allows the detection of errors down to  $10^{-9}$  BER. The error floor at  $10^{-9}$  BER is resulted from the quantization error.



Performance curves at varying block size



Full floating point vs. 3-bit non-uniform quantization

## 5 Conclusion

We have presented a scalable decoding architecture for a certain class of structured LDPC codes protograph, and demonstrated a FPGA implementation of a (4096, 2048) regular (3, 6) structured LDPC code. Partially parallel structure allows high throughput, while the serial processing of multiple copies of the protograph allows a large block length in implementation to improve the performance. Our use of three-bit non-uniform quantization allows near floating point performance in the waterfall region. As demonstrated by this work, an FPGA implementation of LDPC codes can have excellent performance, high throughput, low hardware complexity and easy reconfigurability; FPGA implementation of LDPC codes are expected to be employed for many applications in next-generation communication systems.

## References

- [1] R. G. Gallager, Low-Density Parity-Check Codes, MIT Press, Cambridge, MA, 1963.
- [2] S. Chung, G. D. Forney, T. J. Richardson, and R. Urbanke, "On the design of low-density parity-check codes within 0.0045 dB of the Shannon limit," *IEEE Comm. Letters*, vol. 5, pp. 58-60, Feb. 2001.
- [3] R. M. Tanner, "A recursive approach to low complexity codes," *IEEE Trans. Inform. Theory*, vol. 27, pp. 533-547, Sept. 1981.
- [4] J. Thorpe "Low-Density Parity-Check (LDPC) Codes Constructed from Protographs", IPN Progress Reports 42-154, April-June 2003
- [5] T. Richardson, "Methods and Apparatus for Decoding LDPC Codes," United States Patent No.: US 6,633,856 B2, Oct. 14, 2003.
- [6] H. Zhong and T. Zhong, "Design of VLSI Implementation-Oriented LDPC Codes," *IEEE Semiannual Vehicular Technology Conference (VTC)*, Oct. 2003
- [7] J. Thorpe, K. Andrews, S. Dolinar, "Methodologies for Designing LDPC Codes Using Protographs," submitted to 2004 IEEE International Symposium on Information Theory.
- [8] X. Hu, E. Eleftheriou, and D. Arnold, "Progressive edge-growth Tanner graphs," *Global Telecommunications Conference, 2001. GLOBECOM '01. IEEE*, Volume: 2, 25-29 Nov. 2001
- [9] J. Thorpe "Low-Complexity Approximations to Belief Propagation for LDPC Codes", Unpublished