Architectures and Implementations of Low-Density Parity Check Decoding Algorithms

Engling Yeo, Borivoje Nikolić, and Venkat Anantharam

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720-1770, USA

ABSTRACT

Architectures for low-density parity-check (LDPC) decoders are discussed, with methods to reduce their complexity. Serial implementations similar to traditional microprocessor datapaths are compared against implementations with multiple processing elements that exploit the inherent parallelism in the decoding algorithm. Several classes of LDPC codes, such as those based on irregular random graphs and geometric properties of finite fields are evaluated in terms of their suitability for VLSI implementation and performance as measured by bit-error rate. Efficient realizations of low-density parity check decoders under area, power, and throughput constraints are of particular interest in the design of communications receivers.

1. Introduction

Low Density Parity Check (LDPC) codes have been demonstrated to achieve information rates very close to the Shannon limit when iteratively decoded [1]. In general, LDPC decoders are known to require an order of magnitude less arithmetic computations than Turbo decoders [2] that provide similar bit-error performance. Furthermore, decoding algorithms for LDPC codes also have the benefit of being inherently parallel. In principle, this permits exploiting the common approach of using multiple parallel processing elements to increase the throughput of the decoder. However implementation of a fully parallel LDPC decoder is impeded by the complexity of the interconnect between the processing elements. This is due to the inherently sparse nature of the underlying bipartite graph [2].

This paper discusses tradeoffs between various LDPC decoder implementations. The desire is to emphasize the implications that code construction techniques have on decoder implementation, beyond just bit-error rate performance. The paper has a tutorial nature.

2. Soft Decoding of LDPC Codes

For each received bit, x_n for $n = 1, 2, \ldots, N$, in an N-bit block, an LDPC decoder accepts as input the log-likelihood ratio (α_n) of probability of possible values for x_n, as defined in (1):

$$\alpha_n = \ln\left[\frac{\Pr(x_n = 1)}{\Pr(x_n = 0)}\right]. \qquad (1)$$

LDPC codes are often represented by bipartite graphs made up of two families of nodes, variable nodes and check nodes. Variable nodes represent the transmitted bits in the code, including both information bits and parity bits. Each variable node is connected to a few check nodes through a sparse array of edges. Each check node represents a parity check constraint on the set of adjacent bit nodes.

LDPC decoders implement a message-passing algorithm, which specifies the computation of messages and their communication between variable nodes and check nodes as defined by the edges in the graph. An iteration of LDPC decoding consists of a round of message passing from each variable node to all adjacent check nodes, followed by another round of message passing from each check node to its adjacent variable nodes. Decoding performance is achieved through repeated iterations of message passing along edges in the graph, with some stopping criterion.

Let $\boldsymbol{H}$ be the $M\times N$ parity check matrix of an LDPC code comprising N variable nodes and M check nodes. The set $\nu(m) = \{n\text{: } \boldsymbol{H}_{m,n} = 1\}$ defines the variables that are connected to check m. $\mu(n) = \{m\text{: } \boldsymbol{H}_{m,n} = 1\}$ is the set of checks that are connected to variable n. $Q_{n,m}$ and $R_{m,n}$ refer to the messages that are passed between variable n and check m, as defined in (2) and (3) respectively.

Message from variable n to check m:

$$Q_{n,m} = \alpha_n + \left(\sum_{m'\in\mu(n)} R_{m',n}\right) - R_{m,n} \qquad (2)$$

Message from check m to variable n:

$$R_{m,n} = \Phi^{-1}\left\{\left(\sum_{n'\in N(m)} \Phi(Q_{n',m})\right) - \Phi(Q_{n,m})\right\} \times \left(\operatorname{sgn}(Q_{n,m})\cdot \prod_{n'\in\nu(m)} \operatorname{sgn}(Q_{n',m})\right) \qquad (3)$$

$$\Phi(x) = -\log\left(\tanh(\tfrac{1}{2}|x|)\right) = \Phi^{-1}(x); \qquad x \geq 0 \qquad (4)$$

In order to simplify the hardware, for computation of the first term in (3), a lookup table is used to approximate (4). The second term is computed by applying an exclusive-or function on the most-significant bits of the input messages. The messages are naturally represented as signed-magnitude values, such that $\Phi(x)$ is a function of the magnitude value of its operand.

Figure 1 shows an example structure which computes the messages $R_{m,n}$ for $n = n_1, n_2, \ldots, n_{|\nu(m)|}$ with a wordlength of b. The structure is divided into two portions. The top half evaluates the magnitude values of the check-to-variable messages with a collection of adders. The lower half evaluates the marginalized parity checksums through an XOR exchange. The size of the adders and the XOR exchange depends on the edge degree of the particular node, defined as the number of edges connected to it. A (x,y) regular code is defined as one whose variable and check nodes have uniform edge degrees of x

0-7803-7523-8/02/$17.00 ©2002 IEEE

and y respectively.

An LDPC decoder performs the message computations (2)-(3) and provides for proper relaying of messages between the two classes of nodes. The computational complexity required for evaluation of either variable-to-check or check-to-variable messages is low, compared to Turbo decoders [2]. However, the sparse structure of the underlying bipartite graph leads to complex interconnect in parallel architectures, or excessive memory requirements in the alternative serial implementations.

3. Parallel vs. Serial Architectures

3.1 Parallel Architecture

The message passing algorithm described above is inherently parallel because there is no dependency between computation of either $Q_{n,m}$ for $n = 1, 2, \ldots, N$ or $R_{m,n}$ for $m = 1, 2, \ldots, M$. Parallel decoder architectures directly map the nodes of a bipartite graph onto message computation units known as processing elements, and the edges of the graph onto a network of interconnect.

A fully parallel architecture (Figure 2) provides potentially the fastest decoding throughput. Since it relies on concurrent computation of messages, the number of processing elements required is the same as the number of nodes in the underlying graph. Parallel architectures are also subject to complex interconnect because a hard-wired route is required between every pair of adjacent processing elements. An implemented 1024-bit LDPC decoder [5] with an average variable edge degree of 3.25 and message wordlength of 3 has 9750 wires. The design has logic density of only 50% in order to accommodate the complexity of the interconnect fabric. Area of implementation and interconnect routing are the two major issues related to parallel architectures. Due to irregularity in the parity check matrix, it is difficult to partition a fully parallel design into smaller sub-blocks.

3.2 Serial Architecture

An alternative approach is to serialize and distribute an inherently parallel algorithm among a small number of processing elements [2],[6] as shown in Figure 3. The messages $Q_{n,m}$ and $R_{m,n}$ are stored temporarily in memory between their generation and consumption. This architecture results in less area and much less routing, but dramatically increases memory requirements.

By traversing multiple steps through the bipartite graph, it can be shown that the computation of messages in the decoding algorithm has data dependencies on messages corresponding to a large number of edges. For example, the computation of a message $Q_{n,m}$ in (2) has data dependencies on messages $R_{m',n}$ for $m' \in v(n) \backslash m$. These messages depend on a larger set of $Q_{n',m'}$ $n' \in \mu(m') \backslash n$ from the last iteration. In LDPC codes with good asymptotic performance, the trace of dependencies grows rapidly through each step. This behavior is related to the requirement that the girth of the graph be large [13], and implies that the decoder will be required to have written most of the computed $Q_{n,m}$ messages into memory, before the computation of $R_{m,n}$ messages can proceed (and vice-versa). The size of the memory required is therefore dependent on the total number of edges in the particular code design. A serial implementation of a rate-8/9 4608-bit LDPC decoder described in [2] has more than 18000 edges in the underlying graph, and would have to perform 37000 memory read or write operations for each iteration of decoding.

Serial architectures that use more than one processing element require memory devices with operating frequencies that are faster than the datapath, or multiple I/O ports. Memory access is approximately 2ns (general-purpose single-ported 32kb memories in 0.13 μm CMOS technology), significantly more than the sub-400ps required to add four 5-bit numbers in the LDPC decoding logic (0.13μm CMOS ASIC design). The memory access is therefore in the critical path. On the other hand, the area of memories is subject to quadratic growth with number of I/O ports. The transistor widths in the SRAM cells of a multi-ported memory have to be increased in order to provide greater noise immunity. Thus, both options are unsuitable solutions for high throughput decoder implementation.

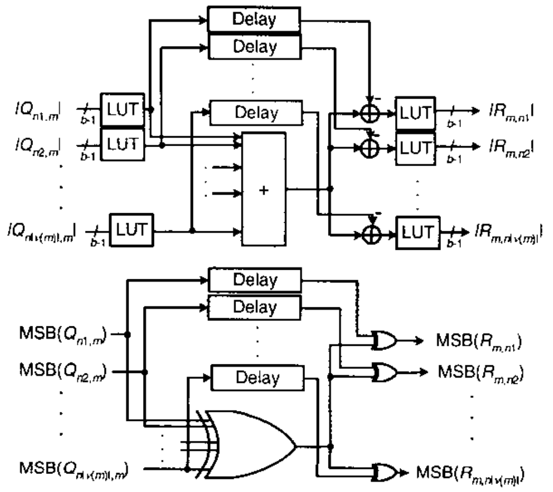

Figure 1. Structure for computing check-to-variable messages $R_{m,n}$ for $n = n_1, n_2, \ldots, n_{|v(m)|}$ with wordlength b.

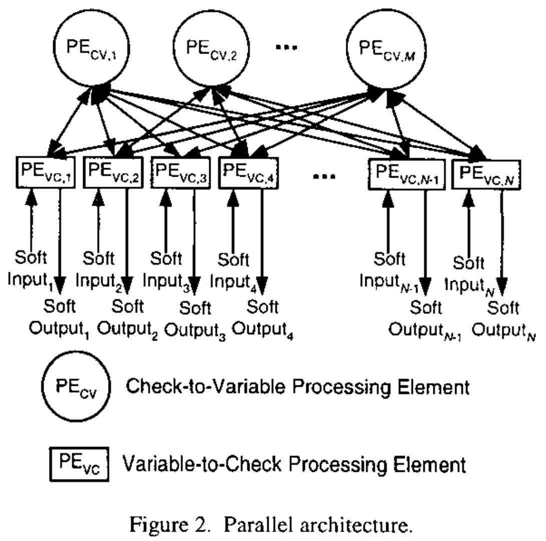

Figure 2. Parallel architecture.

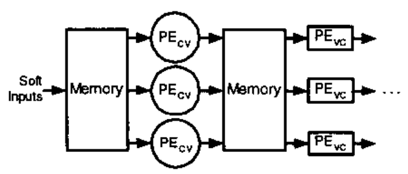

Figure 3. Serial architecture.

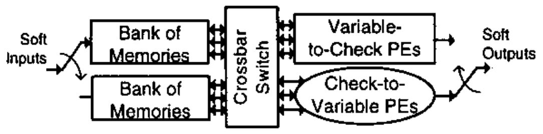

Figure 4. Reducing routing congestion by pipelined and serial implementation.

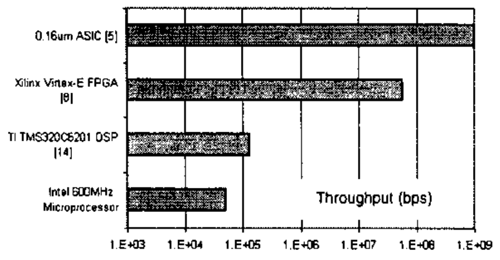

Figure 5. Various platforms vs. realistic throughput rates of rate-½ decoders.

To improve the speed of multiple-memory access, a solution [6] uses a crossbar switch to facilitate communication between *P* processing elements and a bank of memories consisting of *P* independent SRAMs. Each processing element selects an input from one of the SRAMs in each memory cycle. However, inadvertent memory access collisions will cause the processing elements to stall. The solution in [6] proposed an ad-hoc scheduling method to avoid such conflicts. This schedule, though, may require an access pattern that is difficult to compute on the fly and therefore has to be stored as ROM data.

4. Platforms for LDPC Decoding

LDPC codes are applicable to wireless, wired and optical communications. The type of application dictates the particular class of platforms suitable for implementation of an LDPC decoder. Wireless applications are focused on low power implementation with rates at a few hundreds of kbps to several Mbps. Wireline access technologies such as VDSL have envisaged data rates up to 52 Mb/s downstream. Wireless LANs require data rates of the order of 100Mb/s. Storage applications require about 1Gbps, while optical communication throughputs can be above 10Gbps.

Microprocessors and digital signal processors (DSPs) have a limited number of execution units but provide the most flexibility. These platforms naturally implement the serial architecture for LDPC decoding. Although an optimized program may decode at throughput rates of a few hundreds of kbps, practical use of microprocessors have to address operating system overhead. As a result, sustained decoding throughputs up to 100kbps are more realistic. Microprocessors and DSPs are used as tools for the majority of researchers in this field to design, simulate, and perform comparative analysis of LDPC codes. Performing simulations with bit error rates below 10^{-6}, however, is a lengthy process.

FPGAs and custom ASICs are suitable for direct mapping of the message-passing algorithm, and offer more parallelism with reduced flexibility. Each computational logic block (CLB) in an example Xilinx™ Virtex-E FPGA can implement a 4-bit adder, or two 5-input XORs, or four 4-bit table lookups. The array of 104 × 156 CLBs in a XCV3200E is sufficient to execute the decoding logic of a fully parallel decoder for a 1024-bit, rate-½, (3,6) regular LDPC code. The implementation of each variable-to-check (5 adders) and check-to-variable (eleven adders, six 5-input XORs, and twelve table lookups) processing element requires 5 CLBs and 17 CLBs respectively. However, fully parallel LDPC decoding architectures will face mismatch between the routing requirements of the programmable interconnect fabric and bipartite graph. FPGAs are intended for datapath intensive designs, and thus have an interconnect grid optimized for local routing. The sparse nature of the LDPC graph, however, requires global and significantly longer routing. Existing implementations [7] [8] eluded this problem by using time-shared hardware and memories in place of interconnect. This serial method limits the internal throughput to 56Mbps.

A direct-mapped custom ASIC implementation has been demonstrated on a rate-½, 1024-bit parallel LDPC decoder [5] in 0.16μm technology. It dissipates 690mW at 1Gbps decoding throughput, and has an area of 7mm×7mm. Unfortunately, the high throughput and low power dissipation of a parallel design is not easily scalable to codes with larger block sizes. For decoding near the capacity bound, block sizes with tens of thousands of bits are required. With at least 10 times more interconnect wires, a parallel implementation will face imminent routing congestion, and exceed viable chip areas.

An approach to avoid the routing congestion is through time-sharing of hardware units; with hardware pipelining (through segmenting the check-to-variable and variable-to-check stages) to sustain the high throughput rates. Full utilization of all processing elements in the pipeline is only achievable if the computation of each class of messages is operating on an independent block of data. Figure 4 illustrates the use of separate banks of memories to store messages corresponding to consecutive blocks. The messages are routed through a crossbar switch to two groups of processing elements, divided by their functionality. The amount of memory required remains a linear function of the total number of edges in the bipartite graph. An LDPC decoder core that exemplifies this approach has become available as a commercial IP [9]. It supports a maximum parallelism factor of 128, though details of the particular LDPC code have not been published.

Additional reduction of the memory requirement has been proposed through a staggered decoding schedule [10]. This approach does not perform marginalization of the variable-to-check messages. By not computing the last term in (2), it has a memory requirement that is dependent only on the total number of variable nodes in the block. Decoders with area or power constraints that limit the number of iterations to five or less will benefit from more than 75% reduction in memory requirement, while yielding to less than 0.5dB loss in BER performance. It is noted that the staggered decoding will not achieve the same asymptotic results as LDPC decoding under belief propagation.

The decoding throughputs of several platforms implementing rate-½ codes are compared in Figure 5.

5. Effects of Code Construction on Implementation

Most research has focused on the design of LDPC codes with the best possible bit-error-rate performance while the suitability of these codes for parallel implementation has received little attention. In terms of implementation-related issues, LDPC code construction techniques can be differentiated along the lines of whether the code has a structured graph, a uniform edge degree (regular codes), and whether the maximum edge degree (of both check and variable nodes) is relatively large or not.

The method of LDPC construction based on density evolution [3] [11] has one of the best reported performances being only 0.0045dB away from the Shannon bound. An example of this construction method yields a rate-½ irregular code with a maximum variable degree of 100 and block size of 10^7 bits. It also requires an average of more than 1000 iterations to achieve the above decoding results. Practical implementations of decoders, particularly parallel ones, however, benefit from a regular code with a small maximum edge degree in order to avoid detrimental arithmetic precision effects, and the complexity of collating a large number of inputs and outputs at the processing elements. A parallel decoder implementation with 10^7 processing elements will exceed realistic area constraints. These codes are thus much more easily mapped onto a serial architecture, but will result in extended decoding latencies.

Codes based on Cayley graphs and Ramanujan graphs [12][13] have an unstructured graph representation with maximum edge degrees that are usually less than 10. The implementation of decoders for these codes continue to face the primary difficulty of routing unstructured interconnects. However, the lowered maximum edge degrees make the decoder implementation more feasible. An implementation [7] was able to use a bus to multiplex a number of neighboring interconnects in order to reduce routing congestion. These codes do suffer some degradation in SNR performance. A rate-½ (3,6) regular code with 4896 bits achieved a bit error rate of 10^{-5} at 1.7dB away from the theoretical bound [12].

Finally, a class of highly structured and regular codes is based on properties of finite fields [4]. These constructions allow for both high or low edge degrees, with corresponding implications on their error correcting performance; the performance suffers both if the degree is too small and if it is too large. The demonstrated rate-½ (32,64) code has a block size of 8190 bits, and achieves a bit error rate of 10^{-5} at 1.8dB away from the theoretical bound. Although the edge degree is higher than codes based on Cayley or Ramanujan graphs, these codes have a natural cyclic structure, which can be exploited to allow the use of high-speed shift registers. Column splitting on these codes also yields added parallelism between memory accesses in serial architectures with a limited number of parallel processing elements [10]. A fully parallel implementation, however, still has to cope with complex interconnect because the sparseness of the graph requires a large amount of global routing.

In order to produce viable real-time LDPC decoding, future techniques in code construction will have to address the complexity of routing parallel implementations.

6. Conclusion

When compared with turbo decoders, LDPC decoders need much less computation to achieve a similar performance. However, the sparse nature of the LDPC code presents a serious implementation bottleneck. Serial and parallel implementations of LDPC decoders have to primarily address the issue of large memory requirement and interconnect congestion respectively. The choice between a serial and a parallel implementation is thus tied to the tradeoff between memory or interconnect complexity concerns.

Direct mapping of the decoding algorithm onto an FPGA or custom ASIC offer the ability to exploit higher levels of parallelism, leading to higher throughputs without incurring heavy power penalties. However, it comes at the cost of convoluted interconnect and ultimately, implementation area.

While the topic of LDPC code construction has been a major subject of interest in recent years, it has largely proceeded with the sole purpose of improving the error correcting properties. Methods based on density evolution, Cayley and Ramanujan graphs showed promising BER performance, but future practical implementations of high-throughput LDPC decoders will be trading off the error correcting performance for reduced implementation complexity through the code construction.

7. References

[1] D. J. C. Mackay and R. M. Neal, "Near Shannon limit performance of low density parity check codes," *IEE Electronics Letters*, vol.33, no.6, pp.457-458, Mar. 1997.

[2] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "VLSI architectures for iterative decoders in magnetic recording channels," *IEEE Trans. Magnetics*, vol.37, no.2, pp. 748-755, Mar. 2001.

[3] T. J. Richardson, M. A. Shokrollahi, and R. L. Urbanke, "Design of capacity-approaching irregular low-density parity-check codes," *IEEE Trans. Information Theory*, vol.47, pp.619-637, Feb. 2001.

[4] Y. Kou, S Lin, and M. P.C. Fossorier, "Low-density parity-check codes based on finite geometries: a rediscovery and new results," *IEEE Trans. Information Theory*, vol.47, no.7, pp.2711-2736, Nov. 2001.

[5] A.J. Blanksby and C.J. Howland, "A 690-mW 1-Gb/s 1024-b, rate-½ low-density parity-check code decoder," *IEEE Journal of Solid-State Circuits*, vol.37, no.3, pp.404-412, Mar. 2002.

[6] G. Al-Rawi, J. Cioffi, and M. Horowitz, "Optimizing the mapping of low-density parity check codes on parallel decoding architectures," *Proc. IEEE ITCC*, Las Vegas, NV, USA, Apr. 2-4, 2001, pp.578-586.

[7] M. M. Mansour and N. R. Shanbhag, "Memory-efficient turbo decoder architectures for LDPC codes," to appear in *Proc. IEEE SIPS 2002*, San Diego, CA, USA, Oct 16-18, 2002.

[8] T. Zhang and K. Parhi, "A 56Mbps (3,6)-Regular FPGA LDPC Decoder," to appear in *Proc. IEEE SIPS 2002*, San Diego, CA, USA, Oct 16-18, 2002.

[9] "*Vector-LDPC*™ Core Solutions", Flarion Technologies, Inc., http://www.flarion.com, South Bedminster, NJ 07921, USA.

[10] E. Yeo, P. Pakzad, B. Nikolic, and V. Anantharam, "High throughput low-density parity-check architectures," *Proc. IEEE Globecom*, San Antonio, TX, USA, Nov. 25-29, 2001, pp.3019-3024.

[11] 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.

[12] J. Rosenthal and P. O. Vontobel, "Constructions of regular and irregular LDPC codes using Ramanujan graphs and ideas from Margulis," *Proc. IEEE ISIT*, Washington, DC, USA, Jun. 24-29, 2001, p.5.

[13] M. Sipser and D. A. Spielman, "Expander codes," *IEEE Trans. Information Theory*, vol.42, pp.1710-1722, Nov. 1996.

[14] T. Bhatt, K. Narayanan, and N. Kehtarnavaz, "Fixed Point DSP Implementation of Low-Density Parity Check Codes," *Proc IEEE DSP2000*, Hunt, TX, USA, Oct. 15-18, 2000.