Overlapped Scheduling for Folded LDPC Decoding Based on Matrix Permutation

In-Cheol Park and Se-Hyeon Kang
Department of Electrical Engineering and Computer Science, KAIST
{icpark, shkang}@ics.kaist.ac.kr

Abstract – *The fully parallel LDPC decoding architecture can achieve high decoding throughput, but it suffers from large hardware complexity caused by a large set of processing units and complex interconnections. A practical solution to achieve area-efficient decoders is to use the folded architecture in which a PU is time-multiplexed for several row or column operations. In the folded architecture, the rows or columns to be processed in a PU and their processing order should be carefully determined by analyzing their dependencies among rows and columns to enable overlapped processing that leads to increased performance. Finding an optimum grouping of rows and columns for each PU, however, takes considerable amount of time, since the number of rows and columns can be several thousands. This paper proposes an efficient scheduling algorithm that can be applied to general LDPC codes, which is based on the concept of the matrix permutation. Experimental results show that the proposed scheduling achieves a reduction of 36.7% processing time, on the average, for various LDPC codes, leading to a higher decoding rate.*

Keywords: Channel coding, decoder, factor graph, LDPC code, matrix permutation, scheduling

1 Introduction

Low density parity check (LDPC) codes are originally devised to exploit low decoding complexity by constructing sparse parity check matrices. Though the LDPC code does not have a maximized minimum distance due to the randomly generated sparse parity check matrix, the typical minimum distance increases linearly as the block length increases. Moreover, the error probability decreases exponentially for a sufficiently long block length, whereas the decoding complexity is linearly proportional to the code length. Recent simulation results show that the LDPC code can achieve a performance that is within 0.04 dB of Shannon limit and the performance is close to that of the turbo code if the block length is larger than 1000 bits [2].

Despite of these advantages, when the LDPC code was first introduced, it made a little impact on the information theory community because of the storage requirements in encoding and the computational complexity in decoding. Modern VLSI technology is so advanced that it enables parallel architectures exploiting the benefit of inherently parallel LDPC decoding algorithms. Blanksby et al. implemented an 1-Gb/s fully parallel decoder in which the message passing algorithm is directly mapped [3]. This architecture, however, requires a large number of complex routings between concurrent processing units (PUs) each of which corresponds to a node of the message passing algorithm, leading to the average net length of 3mm and the total die size of 52.5mm^2. On the other side, Yeo et al. proposed an area-efficient architecture that serializes the computations by sharing PUs [4]. Consequently, one iteration takes 9216 cycles and wide-input multiplexers are required to select one of 18432 intermediate values to be fed into the shared PUs. These two counter examples show that high throughput LDPC decoding architectures should exploit the benefit of parallel decoding algorithms while reducing the interconnection complexity.

The folded architecture is a good trade-off between throughput and hardware cost. Since a PU is shared for a number of rows or columns, the number of PUs becomes much smaller than that of the fully parallel architecture. As decoding operations are parallel in nature, it is important to determine which rows or columns are processed in a PU. In the grouping, the dependencies between rows and columns should be considered to minimize the overall cycles by overlapping the decoding operations. There has been a heuristic scheduling algorithm proposed for quasi-cyclic LDPC codes [9], but it cannot be applied to general LDPC codes. This paper proposes an efficient scheduling algorithm that can be applied to general LDPC codes. The proposed algorithm is based on the concept of the matrix permutation.

The rest of this paper is organized as follows. Section 2 explains briefly on the low density parity check code and the decoding algorithm. Section 3 proposes a new scheduling algorithm which helps the folded LDPC decoder achieve high throughput using matrix permutation. Experimental results of the proposed algorithm are shown in Section 4. Finally, Section 5 addresses some concluding remarks.

2 LDPC Decoder Architecture

The LDPC code, which was first introduced by Gallager in 1962 [1], is defined by a binary linear block

code of length n and a parity check matrix **H** with a column weight γ and a row weight ρ: (n, γ, ρ). The parity check matrix **H** has n columns and m rows that correspond to the block length and the total number of parity check equations of the code, respectively. The column weight γ and the row weight ρ represent the number of 1's in a column and a row, respectively, and they are much smaller than n to achieve the sparse matrix **H**. As the code parameters are related by an equation, $n \times \gamma = m \times \rho$, the code rate can be calculated as $R = 1 - \gamma / \rho$. Fig 1 (a) shows an example matrix **H** of (12, 3, 6) LDPC code with indicating the code parameters. An LDPC code associated with fixed row and column weights is called a regular LDPC code, and an irregular LDPC code allows some variations in row and column weights.

2.1 Message Passing Algorithm

The LDPC code is often represented by a factor graph to make it easy to understand the message passing decoding algorithm, which is a bipartite graph that expresses how a global function of many variables is factored into a product of local functions [7]. Fig 1 (b) shows the factor graph of a (12, 3, 6) LDPC code, which consists of two sets of nodes: i.e. variable nodes, $\{v_j\}$, and check nodes, $\{c_i\}$. The edge between a variable node v_j and a check node c_i is constructed if there is 1 at (i, j) in the parity check matrix. Therefore each check node represents a check equation used in generating parity check bits and each variable node represents one bit in the codeword. The variable nodes that are connected to a check node are called the neighbor variable nodes of the check node. For a variable node, the neighbor check nodes are defined similarly.

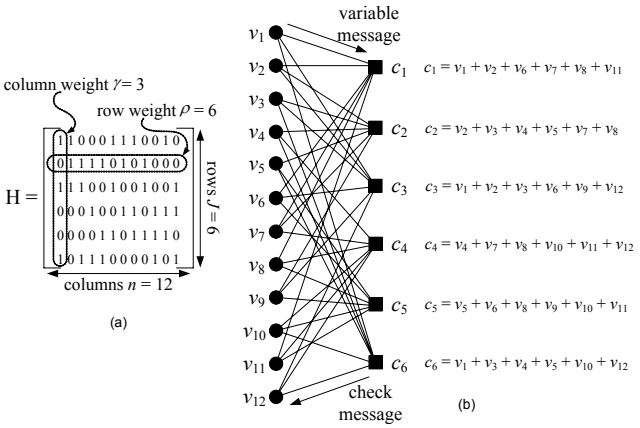

Fig 1. A (12, 3, 6) LDPC code. (a) Parity check matrix. (b) Factor graph.

Unlike the general parity check codes, a LDPC code cannot be optimally decoded since its decoding is NP-complete. An iterative approximate algorithm called a message-passing algorithm is used instead, which is also known as sum-product [8] or belief propagation [5]. The message-passing algorithm can be expressed easily based on the factor graph as below.

1) Initialize each variable node v_j to the probability ratio of the corresponding received bit.

$$\Delta_j = \frac{p(r_j \mid x_j = 1)}{P(r_j \mid x_j = 0)} = \exp(\frac{2r_j}{\sigma^2}) \tag{1}$$

2) Variable-to-check (VTC) step: Each check node c_i computes the likelihood ratio, Λ_{ij}, by using (2) and sends it to variable node v_j.

$$\Lambda_{ij} = \prod_{j' \in N(i) \setminus j} \frac{1 - \Delta_{ij'}}{1 + \Delta_{ij'}} \tag{2}$$

3) Check-to-variable (CTV) step: Each variable node v_j computes probability ratios, denoted by Δ_{ij}, by using (3) and sends it to check node c_i.

$$\Delta_{ij} = \frac{p(r_j \mid 1)}{P(r_j \mid 0)} \prod_{i' \in M(j) \setminus i} \frac{1 - \Lambda_{i'j}}{1 + \Lambda_{i'j}} \tag{3}$$

4) For each variable node, create a tentative bit-by-bit decoding xj using the pseudo-posteriori probability Δj.

$$\Delta_j = \frac{p(r_j \mid 1)}{P(r_j \mid 0)} \prod_{i \in M(j)} \frac{1 - \Lambda_{ij}}{1 + \Lambda_{ij}} \tag{4}$$

$$x_j = \begin{cases} 0 & when\ \Delta_j \le 1 \\ 1 & when\ \Delta_j > 1 \end{cases} \tag{5}$$

5) Check if $\mathbf{x} \cdot \mathbf{H}^T = 0$ is satisfied. If it is satisfied or the maximum number of iterations is reached, the decoding algorithm finishes. Otherwise, the algorithm repeats from step 2).

The symbols, Δ and Λ, correspond to the outgoing messages of the variable and check nodes, respectively, and $N(i)$ and $M(j)$ represent the set of neighbor nodes of check node c_i and variable node v_j, respectively. The not-notation (') marked as superscript means that the product is calculated over the indexes excluding its own index. The VTC operation calculates the check messages $\{\Lambda_{ij}\}$ of check node c_i using the variable messages $\{\Delta_{ij}\}$ for variable nodes identified by the i-th row of the matrix **H**. A detailed explanation of the algorithm can be found in [6].

As the message passing algorithm is projected on the parity check matrix, a check node corresponds to a row and a variable node corresponds to a column in the parity check matrix. The Δ_{ij} and Λ_{ij} messages are updated at 1's positions after each step. Thus a VTC operation has dependencies to CTV operations which will update Δ_{ij} messages at the 1's position of corresponding row. But there is no dependency between VTC operations.

2.2 Folded Architecture

The fully parallel architecture is inherited from the message-passing algorithm and all the operations required in a check/variable node are implemented as a check/variable PU. The number of PUs is as many as the number of the variable and check nodes. Therefore each iteration consists of two cycles; One is for VTC operations

and another for CTV operations. All the check PUs and variable PUs are operated simultaneously in their corresponding clock cycle. Although the fully parallel implementation of the message passing algorithm is straightforward and results in a high throughput decoder, it faces with complex interconnections required to sum up the Λ_{ij} and Δ_{ij} messages that are calculated in the PUs spread over the chip area. The serial architecture, in which a shared check/variable PU computes all the rows /columns one after another, requires not only many cycles, but also large multiplexers to read and write the messages in serial order.

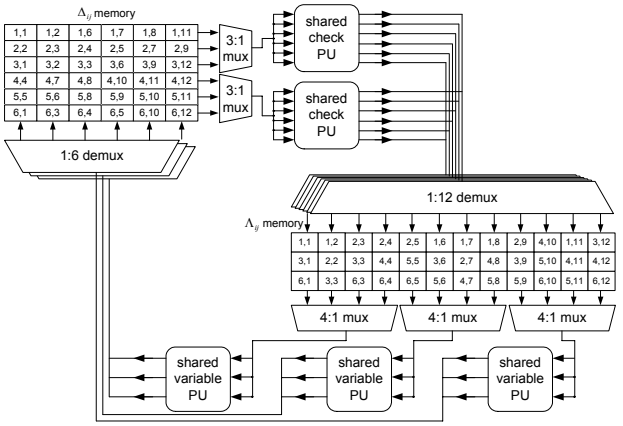

Fig 2. Folded LDPC decoding architecture.

In the folded architecture, each PU takes in charge of several numbers of rows or columns as shown in Fig 2. In this case, there are 2 check PUs each of which take charge of 3 rows and 3 variable PUs each of which processes 4 columns. As a PU is shared for a number of rows or columns, the number of PUs becomes much smaller than that of the fully parallel architecture. The number of VTC (CTV) operations processed in a cycle is as many as the number of check (variable) PUs. Therefore each VTC (CTV) step takes several cycles as many as the number of rows (columns) that a PU takes in charge of. The check messages calculated row-by-row by a check PU can be grouped and stored into a local memory to save area, and variable PUs access them later.

The VTC and CTV operations are parallel in nature and thus a PU can process any rows or columns regardless of their order in matrix **H**. Since a PU takes in charge of several rows or columns in the folded architecture, we have to determine which rows or columns are processed in the PU. In the grouping, the dependencies between rows and columns should be considered to minimize overall cycles by overlapping the VTC and CTV operations. For example, the CTV operation for v1 in the (12, 3, 6) LDPC code requires the VTC operations for c_1, c_3 and c_6 to be completed beforehand. In addition, it is important to determine the processing order of rows or columns for a PU, because the dependence of the Λ_{ij} and Δ_{ij} can hinder the overlapped processing.

3 Scheduling By Permutation

Traditionally, the CTV step starts only after the entire VTC step finishes completely, and vice versa. A well-scheduled sequence of the message calculations can reduce the latency, because a CTV operation can start in the course of the VTC step if the corresponding row summation values are available, and vice versa. The overlapped processing of the VTC and CTV steps results in a reduced number of processing cycles for an iteration.

3.1 Overlapped Processing

Fig 3 shows an example of overlapped processing for the (12, 3, 6) LDPC code, assuming that the numbers of check PUs and variable PUs are two and three, respectively. Each arrow denotes a clock cycle and the numbers above an arrow indicate the row or column indices processed at that cycle. If the VTC and CTV steps are performed according to the increasing order of indices without scheduling, no overlapped processing is possible. Though two CTV operations for v_2 and v_7 can start at the last cycle of the VTC step, the CTV step does not start until three CTV operations are enabled for easy control design. If the VTC operations are scheduled as shown in Fig 4 (b), three CTV operations for v_2, v_6 and v_9 can start processing at the last cycle of the VTC step. Furthermore, the CTV operations can be scheduled to enable the VTC operations of the next iteration to start earlier. In this example, the scheduling of the VTC and CTV operations saves two cycles. It is called overlapped processing because the VTC and CTV operations are executed concurrently at the end of each step, whereas the VTC and CTV steps are executed one after another in the conventional message passing algorithm.

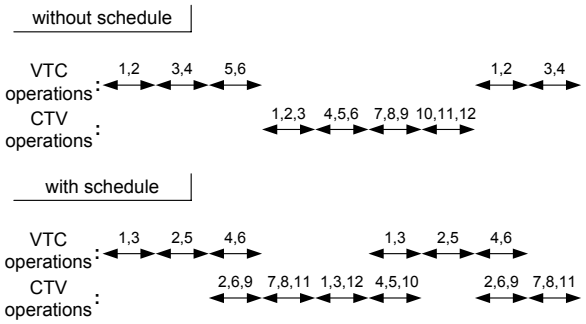

Fig 3. An example of overlapped processing.

The scheduling process can be viewed as the row and column permutation of the parity check matrix **H**. Fig 4 (a) is the original parity check matrix **H** whose size is $M \times N$. In this case, the size of **H** is 6×12 and we assume 2 check PUs and 3 variable PUs. If there are m check PUs, m rows at the top of the matrix are processed in the first cycle, the next m rows in the second cycle, and so on. Thus the VTC step takes M/m cycles. Similarly, the CTV step takes N/n cycles if there are n variable PUs. Therefore the row and

column indices on the top and bottom side of the matrix indicate the processing orders and thick line divides the rows and columns to be processed in a cycle. The permutation changes the sequence of the VTC and CTV operations to enlarge the empty slots at the lower left and upper right corners as shaded in Fig 4 (b).

The connected empty slots in a column, which start from the bottom of the matrix, mean that the CTV operation of the column can start as long as the VTC operations of the rows above the empty slots are processed beforehand. If there are k empty slots in a column, the column processing can start $\lfloor k/m \rfloor$ cycles earlier, and if the first n columns have more than m empty slots, all the variable PUs can start their processing one cycle earlier. The early starting is possible if the $m \times n$ slots that are at the lower left part surrounded by thick lines are all empty. A partial activation of some operations in a group is not allowed because it requires a complex control design and does not lead to a noticeable cycle reduction. For example, as three columns have empty slots larger than two in Fig 4 (b), they can start after two cycles of the VTC step. Similarly, the connected empty slots in a row, which start from the left side of the matrix, mean that the VTC operation of the row can start after the CTV operations for the columns at the left hand side of the empty slots are finished. In this example, two VTC operations can start after three cycles of the CTV step, leading to one cycle reduction. Now, the scheduling problem is translated to an ordering problem which makes both the upper right and lower left corners empty as many as possible.

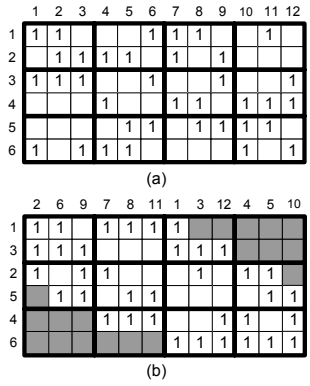

Fig 4. Scheduling by the permutation of matrix **H**. (a) Matrix **H**. (b) Permuted matrix.

There is one thing to be considered when overlapping VTC and CTV steps. The earliest cycle found by the permutation is not the real starting time of the CTV (VTC) step. For the sake of easy control design, we assume that the CTV (VTC) step performs continuously without skipping cycles once it starts. If there is a cycle in which any column (row) cannot be processed during the CTV (VTC) step, the former operations are delayed to achieve continuous CTV (VTC) operations. Fig 5 shows an example illustrating the delayed start of the CTV step. First three column groups can be processed 5 and 3 cycles earlier, but the first column group is delayed to start 3 cycles earlier as shown in Fig 5 (b), because the early start of CTV operations of more than 3 cycles can not contribute to the cycle reduction and results in a more complex control design due to the non-continuous processing.

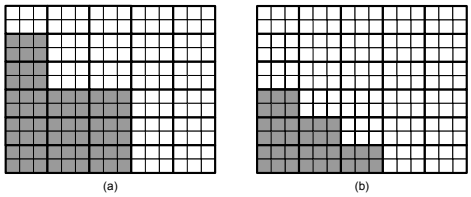

Fig 5. (a) After permutation. (b) Delayed start of CTV operations.

3.2 Scheduling Algorithm

It takes a considerable amount of time to find an optimum schedule that minimizes the overall cycles because of the large number of rows and columns. There has been a heuristic scheduling algorithm proposed for quasi-cyclic LDPC codes [9], but it cannot be applied to general LDPC codes. This paper describes a new scheduling algorithm developed for the folded LDPC decoding architecture. The proposed algorithm is based on the concept of the matrix permutation. The algorithm makes the row sequence and column sequence that can result in empty spaces in the lower left and upper right corners of the permuted matrix when the matrix **H** is rearranged according to the sequences. The row sequence is made by selecting rows one by one in the proposed algorithm. Before describing the proposed algorithm, there are two terms to be defined first.

DEFINITION I. CC (common columns) of an unselected row means the number of 1's that are column with the selected rows.

If we assume that the 1st and 3rd rows in the parity check matrix for the (12, 3, 6) LDPC codes are selected, the CC of the 5th row is four, because the row have 1's in the 2nd, 8th, 9th and 11th columns in common with the selected rows.

DEFINITION II. CTC (Columns To be Completed) of an unselected row means the number of columns whose (γ - 1) 1's are located in the selected rows. This means that all the 1's in those columns are covered if the row is selected.

If all the 1's in a column are included in the already scheduled rows, it can be thought that all the VTC operations which have dependencies with the column are completed. In Fig 4 (b), if we assume that the 1st, 3rd and 2nd rows are already selected, the CTC of the 5th row is two since all the 1's in the 6th and 9th columns are included in the scheduled rows by appending the 5th row to the set of scheduled rows.

The proposed algorithm determines the row and column sequences and then groups rows and columns according to the number of PUs. A row is selected and appended to the row sequence by looking at CC and CTC. This will make enlarged empty space at the upper right corner of the matrix. Then the corresponding columns of the selected row are appended to column sequence and the completed columns are moved forward to make the lower left corner empty. The row and column sequences become the processing order.

Step 1: Initialize row and column sequences with null.

Step 2: Order row indices randomly and pick the first one. Then go to Step 4.

Step 3: Count CC and CTC for unselected rows. Then pick a row r_i that has the maximal CTC. If there are more than two rows associated with the maximal CTC, select the one that has the largest CC.

Step 4: Append columns which have 1 at the just selected row but not selected yet to the column sequence.

Step 5: If there are columns that are newly completed by selecting the rows, move them right after the already completed columns. Uncompleted columns are shifted right by the number of the newly completed columns.

Step 6: If there are unselected rows go to step 3. Otherwise go to Step 7.

Step 7: If there are m check PUs, assign the first m rows at the top of the permuted matrix to the first cycle of the VTC step, the next m rows to the second cycle, and so on. Similarly, for n variable PUs, assign n columns from the left of the permuted matrix to each cycle of the CTV step.

Step 8: Calculate the start point of the CTV and VTC operations for easy control design.

The random ordering of row indices in Step 2 is to explore more search space by starting from a different row and giving priority to a different row when there are many rows that have the same CTC and CC. Steps 3 to 5 are the main loop of matrix permutation and Steps 7 to 8 assign rows and columns to PUs considering easy control design. An example of the algorithm is illustrated in Fig 6, where dotted circles represent newly completed columns.

3.3 Time Complexity

The time complexity of the proposed algorithm is $O(m^2)$, where m is the number of rows of the parity check matrix. In the parity check matrix, the row and column weights are bounded by a constant. Counting CC and CTC of a row takes a constant time if the row is associated with a list of column indices containing 1's. Therefore, it takes

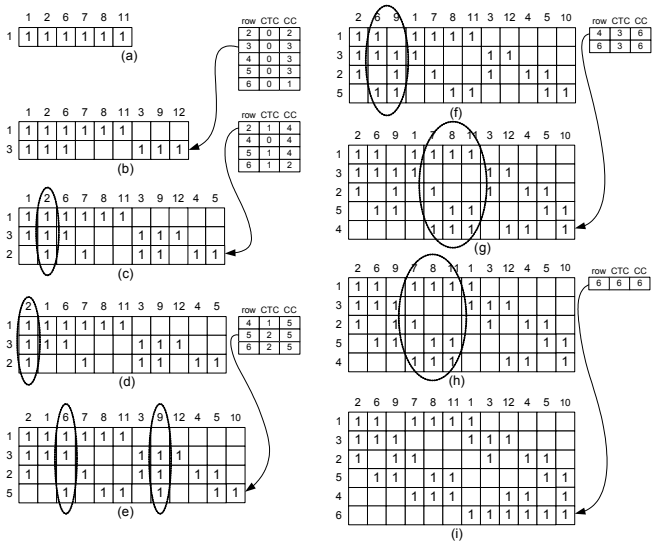

Fig 6. An illustrative example of the proposed scheduling. (a) Select r_1. (b) Append r_3. (c) Append r_2. (d) Move c_2. (e) Append r_5. (f) Move c_6, c_9. (g) Append r_4. (h) Move c_7, c_8, c_{11}. (i) Append r_6.

$O(m)$ time to select a row out of m rows. For a selected row, we have to check whether there are newly completed columns and move them to the right of already completed columns if there are. The checking takes $O(1)$ time because of the bounded row weight, and the moving takes also $O(1)$ time if we keep the last position of already completed columns. As only one row is selected at a time, m iterations are needed to select all the rows. Therefore the time complexity becomes $O(m^2)$.

4 Experimental Results

By applying the proposed scheduling algorithm to various LDPC codes, we can save the number of processing cycles per iteration as summarized in Table 1, where parallelism (m, n) stands for an implementation associated with m check PUs and n variable PUs.

In Table 1, there are three types of LDPC codes. One is from Mackay's Encyclopedia of Sparse Graph Codes [10]. They are regular LDPC codes constructed systematically for the binary symmetric channel [12]. Another is constructed using Neal's software [11]. Basically, it tries to locate 1's randomly on the parity check matrix, maintaining the number of checks per row approximately uniform. Therefore resulting LDPC codes are irregular. The other is also a kind of randomly constructed LDPC codes. The parity check matrix is extended by appending a column whose 1's locations are randomly generated under row and column weight and no 4-cycle constraints. For the three types, we generate several codes by varying code length, column and row weight and code rate.

Table 1. Cycles saved by scheduling algorithm per iteration.

Code	Parallel-ism	Cycles w/o scheduling	Cycles w/ scheduling	Saved cycles	Saved cycles (%)
816.3.6.MK	(17, 34)	48	29	19	39.6
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
816.4.6.MK	(17, 34)	56	35	21	37.5
	(34, 51)	40	27	13	32.5
	(34, 68)	28	18	10	35.7
1008.3.6.MK	(24, 56)	39	24	15	38.5
	(28, 56)	36	22	14	38.9
	(36, 63)	30	19	11	36.7
816.3.6.N	(17, 34)	48	28	20	41.7
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
1024.3.6.N	(16, 32)	64	38	26	40.6
	(32, 32)	48	33	15	31.3
	(32, 64)	32	20	12	37.5
1024.4.8.N	(16, 32)	64	41	23	35.9
	(32, 32)	48	34	14	29.2
	(32, 64)	32	21	11	34.4
816.3.6.r4	(17, 34)	48	28	20	41.7
	(34, 51)	28	18	10	35.7
	(34, 68)	24	15	9	37.5
1024.3.6.r4	(16, 32)	64	38	26	40.6
	(32, 32)	48	32	16	33.3
	(32, 64)	32	20	12	37.5
2048.3.6.r4	(32, 64)	64	39	25	39.1
	(64, 64)	48	33	15	31.3
	(64, 128)	32	20	12	37.5
Average	-	-	-	-	36.7

The proposed scheduling algorithm reduces the number of cycles per iteration by 36.7% on the average. In Table 1, the third and forth columns represent the number of cycles per iteration obtained without and with overlapped processing, respectively. The fifth and sixth columns indicate the number of reduced cycles resulted from the proposed algorithm and its percentage to the cycles without scheduling. The proposed algorithm works well for all the three types of LDPC codes and the percentage of the saved cycles is almost uniform if the code length and row and column weights are given regardless of their construction methods.

5 Conclusions

This paper has presented an efficient scheduling algorithm for folded LDPC decoding to minimize the overall processing time by overlapping CTV and VTC steps. As the VTC and CTV operations are parallel in nature, a PU can process any rows or columns regardless of their order in matrix **H**, and thus it is very important to determine which rows or columns are processed in a PU. In the grouping, the dependencies between rows and columns should be considered to overlap the VTC and CTV operations. We have proposed an efficient scheduling algorithm using the concept of the matrix permutation, which leads to empty spaces in the lower left and upper right corners of the permuted matrix. The row sequence and column sequence are the order of row and column indices of the permuted matrix and they are grouped and scheduled under the hardware constraints. Experimental results show that the proposed scheduling achieves a reduction of 36.7% processing time on the average, leading to a higher decoding rate. In addition, the proposed scheduling produces near optimum scheduling results regardless of the LDPC construction methods.

Acknowledgment

This work was supported by Institute of Information Technology Assessment through the ITRC and by IC Design Education Center (IDEC).

References

[1] R. G. Gallager, “Low density parity check codes,” *IRE Trans. Info. Theory*, vol. IT-8, pp. 533-547, Jan. 1962.

[2] S. Chung, D. Forney, T. 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] A. Blanksby and C. Howland, “A 690-mW 1-Gb/s, rate-1/2 low-density parity-check code decoder,” *IEEE J. of Solid-State Circuits*, vol. 37, pp. 404-412, Mar. 2002.

[4] E. Yeo, P. Pakzad, B. Nikolić and V. Anantharam, “VLSI architectures for iterative decoders in magnetic recording channels,” *IEEE Trans. Magnetics*, vol. 37, pp. 748-755, Mar. 2001.

[5] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, Morgan Kaufmann, 1988.

[6] K. Lo, *Layered space time structures with low density parity check and convolutional codes*, MS Thesis, School of EIE, Univ. of Sydney, 2001.

[7] F. R. Kschischang, B. J. Frey, and H. Loeliger, “Factor graphs and the sum-product algorithm,” *IEEE Trans. Info. Theory*, vol. 47, pp. 498-519, Feb. 2001.

[8] N. Wiberg, *Codes and decoding on general graphs*, PhD thesis, Dept. of EE, Linköping Univ. Sweden, 1996.

[9] Y. Chen, K. K. Parhi, “Overlapped message passing for quasi-cyclic low-density parity check codes,” *IEEE Trans. Circuits and Syst. I*, vol. 51, pp. 1106-1113, Jun. 2004.

[10]D. Mackay, *Encyclopedia of Sparse Graph Codes*. Available:http://www.inference.phy.cam.ac.uk/mackay/codes/data.html

[11]R. Neal, Software for Low Density Parity Check (LDPC) codes. Available: http://www.cs.utoronto.ca/~radford/ldpc.software.html

[12]D. Mackay, “Good error-correcting codes based on very sparse matrices,” *IEEE Trans. Info. Theory*, vol. 45, pp. 399-431, Mar. 1999.