On the Efficient Memory Usage in the Lifting Scheme for the Two-Dimensional Wavelet Transform Computation

Jose Oliver, Elena Oliver, Manuel P. Malumbres
Department of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera 17, 46022 Valencia, Spain
e-mail: joliver@disca.upv.es

Abstract—**In this paper, a new algorithm to efficiently implement the two-dimensional lifting scheme is presented. The 1D lifting-scheme performs in-place processing of the input samples, and hence it provides reduction in memory requirements. However, for image processing (2D), in-place computation is not enough, resulting in a memory-intensive algorithm, since it has to keep the whole image in memory. We propose the use of line-by-line processing algorithm for the lifting scheme, and we address some issues on how to perform synchronization among different buffer levels, so that an implementation can be easily written. Experimental results show that, for a 5-Megapixel image, our algorithm requires 200 times less memory and it is more than 3 times faster than the usual one.**

Keywords: Lifting scheme, line-based, Efficient memory usage

I. INTRODUCTION

The discrete wavelet transform (DWT) is a mathematical tool that has aroused great interest in the last years. However, one of its major drawbacks is the high memory requirements of the regular algorithms that compute it. A proposal that reduces the amount of memory need for the computation of the 1D WT is the lifting scheme [1]. It implements the DWT decomposition as an alternative algorithm to the classical filter bank algorithm. This scheme provides in-place computation of the wavelet coefficients and hence, it does not need extra memory to store the resulting coefficients. A disadvantage of the lifting scheme for the 1D DWT is that the high frequency and low frequency coefficients are interleaved in memory. Thus, the later reordering of the wavelet coefficients would need extra memory.

For the image wavelet transform (2D), the use of the lifting scheme shows little benefit, since the entire image has to be kept in memory. Therefore, it has to be applied along with other strategies that allow us to avoid keeping the entire image in memory. The line-based approach [2] can help us to overcome this problem. In the line-based approach, for the first decomposition level, we receive directly image lines, one by one. On every input line, a one-level 1D DWT is applied. Then, these lines are stored in a buffer associated to the first decomposition level. When there are enough lines in the buffer to calculate a line of each wavelet subband, we compute them. Then, the wavelet subband lines can be processed and released. However, the first line of the LL_1 subband does not belong to the final result, and is needed as incoming data for the following decomposition level. In order to get more lines, we have to update the buffer, filling it with more lines and discarding those that are no longer needed. At the second level, its buffer is filled with the LL_1 lines that have been computed in the first level. Once the buffer is completely filled, it is processed as we have described for the first level. As it is depicted in Figure 1, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

Several hardware implementations of this line-based strategy have been proposed and they can be found in the literature [3] [4] [5] [6].

However, this algorithm cannot be easily implemented in software without a control unit, since a buffer must be completely filled with lines from previous buffers before it can

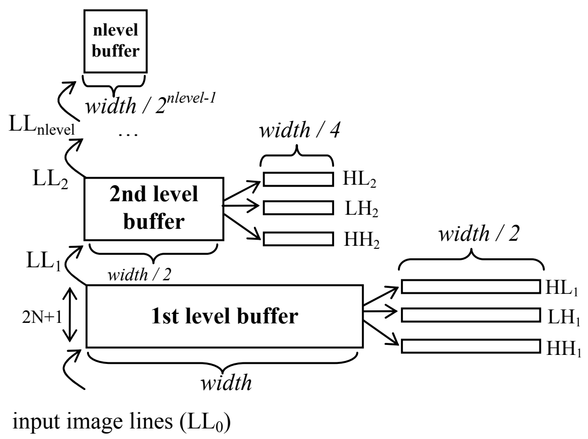

Figure 1: Overview of a line-based wavelet transform

This work was supported by the Spanish Ministerio de Ciencia y Tecnología, funded with the CICYT project under grant TIC2003-0052, and by la Junta de Comunidades de Castilla la Mancha under grant PBC-03-001

```
function GetLLlineBwd( level )
1) First base case:
    If there are no more lines to return at this level
        return EOL
2) Second base case:
    If $level = 0$
        return ReadImageLineIO( )
3) Recursive case
3.1) If $buffer_{level}$ is empty
        Fill up $buffer_{level}$ calling GetLLlineBwd(level-1)
        Get subband lines from $buffer_{level}$
3.2) else if no more lines can be read from the level-1 level
        Start cleaning $buffer_{level}$
        Get subband lines from $buffer_{level}$
3.3) else
        Update $buffer_{level}$ calling GetLLlineBwd(level-1)
        Get subband lines from $buffer_{level}$
    Process the high freq. subband lines $\{HLline, LHline, HHline\}$
    return $LLline$
end of fuction
```

```
function LowMemUsageDWT( nlevel )
    set $buffer_{level} = empty \quad \forall level \in nlevel$
    repeat
        $LLline$ = GetLLlineBwd( nlevel )
        if (LLline!=EOL) Process the low freq. line( $LLline$ )
    until LLline=EOL
end of function
```

Algorithm 1.1: Backward recursive DWT computation

```
3.1) if $buffer_{level}$ is empty
        for $i = N \dots 2N$
            $buffer_{level}(i)$ = 1DFWT(GetLLlineBwd( level-1))
        FullSymmetricExtension( $buffer_{level}$ )
3.2) if not( more_lines( level-1) )
        repeat twice
            Shift( $buffer_{level}$ )
            $buffer_{level}(2N)$ = SymmetricExt( $buffer_{level}$ )
3.3) else
        repeat twice
            Shift( $buffer_{level}$ )
            $buffer_{level}(2N)$ = 1DFWT(GetLLlineBwd( level-1 ) )
For 3.1), 3.2) and 3.3)
    $\{LLline, HLline\}$ = ColumnFWT_LowPass( $buffer_{level}$ )
    $\{LHline, HHline\}$ = ColumnFWT_HighPass( $buffer_{level}$ )
```

Algorithm 1.2: Filter-bank implementation, recursive case

produce lines, and therefore they have different delays, and this control is hard to be performed. Moreover, all the buffers exchange their result at different intervals. In the next section we propose a general recursive algorithm that clearly specifies how to perform this communication between buffers. Then, we will present two implementation options, using a filter algorithm and the lifting scheme.

II. A RECURSIVE LINE-BASED ALGORITHM

The function that implements this recursive algorithm is called GetLLlineBwd() (see *Algorithm 1.1*). This function receives a decomposition level as a parameter, and calculates a line of each wavelet subband (LH, HL and HH) at that level, and returns a line from the low-frequency (LL) subband at that level.

The first time that this recursive function is called at every level, it has its buffer ($buffer_{level}$) empty (case 3.1). Therefore, the buffer has to be recursively filled with lines from the previous level. On the other hand, if the buffer is not empty, it simply has to be updated by discarding some lines and introducing additional lines from the contiguous level (by means of a recursive call again) (case 3.3). However, if there are no more lines in the previous level, this recursive call would return *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still have to calculate some subband lines from the lines in the buffer (case 3.2). We will give more details of each recursive case in Sections 3 and 4, since they are treated in a different way depending on whether we are dealing with a convolution algorithm or with lifting. In both convolution and lifting, at the end of the recursive case, we have a wavelet subband line that is processed and released depending on the application purpose (e.g., compressed), and it returns an LL line.

This function has two base cases. The first case is when all the lines at this level have been read. In this case, the function returns EOL. The second base case is achieved when the level reaches 0 and then no further recursive call is needed since an image line can be read directly from the I/O system and returned.

The inverse DWT algorithm is similar to the forward DWT, but applied in reverse order. A drawback that has not been considered yet is the need to reverse the order of the subbands, from the forward DWT to the inverse one. This problem can be solved using some buffers at both ends, so that data are supplied in the right order [2]. Other simpler solutions are: to save every level in secondary storage separately so that it can be read in a different order and, if the WT is used for compression, to keep the compressed bitstream in memory.

III. A FILTER-BANK IMPLEMENTATION

We are going to use the general description given in Section 2 to implement the DWT computation using a filter-bank algorithm. For convolution, each buffer at every level must be able to keep 2N+1 lines, where 2N+1 is the number of taps for the largest analysis filter bank.

Since the base cases are completely defined in *Algorithm 1.1*, we only have to describe the recursive case. For this convolution implementation, when the buffer is empty (case 3.1), its upper half (from N to 2N) is recursively filled with lines from the previous level. Once the upper half is full, the lower half is filled using symmetric extension. On the other hand, if the buffer is not empty, we have to update it (case 3.3). Thus, we shift it so that a new line can be introduced in the last

```
3.1) if $buffer_{level}$ is empty
        for $i = W \ldots 0$
            $buffer_{level}(i)$ = 1DFWT(GetLLlineBwd( level-1))
        Successively predict and update the lines in $buffer_{level}$,
                provided that the required lines are in buffer.
        $\{LLline, HLline\} = buffer_{level}(W) * (1/K)$
3.2) if not( more_lines( level-1) )
        repeat twice Shift( $buffer_{level}$ )
        Update and predict the remaining lines in the buffer
        $\{LHline, HHline\} = buffer_{level}(W+1) * K$
        $\{LLline, HLline\} = buffer_{level}(W) * (1/K)$
        if there are no more LL lines to be calculated at this level
            $\{LHline, HHline\} = buffer_{level}(W-1) * K$
3.3) else
        repeat twice
            Shift( $buffer_{level}$ )
            $buffer_{level}(0)$ = 1DFWT(GetLLlineBwd( level-1 ) )
        $buffer_{level}(1) = (buffer_{level}(0) + buffer_{level}(2))p_1 + buffer_{level}(1)$
        $buffer_{level}(2) = (buffer_{level}(1) + buffer_{level}(3))u_1 + buffer_{level}(2)$
        …
        $buffer_{level}(W) = (buffer_{level}(W-1) + buffer_{level}(W+1))u_{W/2} + buffer_{level}(W)$
        $\{LHline, HHline\} = buffer_{level}(W+1) * K$
        $\{LLline, HLline\} = buffer_{level}(W) * (1/K)$
```

Algorithm 1.3: Lifting implementation, recursive case

position (2N) using a recursive call. This operation is repeated twice. However, if there are no more lines in the previous level (case 3.2), we fill it using symmetric extension again. In all the cases, once there are enough lines in the buffer to perform one step of a column wavelet transform, the convolution process is calculated vertically twice, first using the low-pass filter and then the high-pass filter. This way, we get a line of every wavelet subband. The whole process is described in *Algorithm 1.2*.

IV. 2D DWT COMPUTATION WITH LIFTING SCHEME AND EFFICIENT MEMORY USAGE

The convolution implementation that has been presented in the previous section introduces wide benefits in memory usage, since we only keep in memory a few (2N+1) low-frequency lines for each decomposition level. However, we can still reduce the amount of memory required by using the lifting scheme.

In the lifting scheme, the wavelet coefficients are computed by means of several steps on the input samples (see Figure 2). In the first step, the samples in odd positions (black squares in the figure) are processed from the contiguous even samples (the white ones). This way, we try to predict each odd sample as a linear combination of the even ones, and thus this step is called prediction step. In the second step, the even values are computed from the contiguous odd ones, and it is called update step. This way, we compute successive prediction and update steps. The total number of steps depends on the DWT transform that is being computed. Finally, the odd values calculated in the last prediction step are normalized by a constant factor (*K*), to achieve the high-frequency wavelet coefficients, and the values from the last update step are normalized by 1/*K* to get the low-frequency coefficients. The lifting scheme depicted in Figure 2 is for the popular B7/9. The derivation from the filter bank to the weighting factors of every prediction and update step is given in [7].

The main advantage of the use of the lifting scheme instead of convolution in our recursive algorithm is the extra reduction of memory achieved. Let us define W as the total number of weighting factors (prediction and update) for a DWT. Then, the buffer height in the lifting scheme has to be W+2, so it can perform the W prediction and update steps needed to compute a low and a high-frequency line in a segmented way, as we will see later. The two additional lines are needed for the first and last computed steps, and they are read but not modified. In general W+2 is lower than 2N+1 (see [7] for details) and hence we need less lines in the buffers. For example, for B7/9, 2N+1 is 9 while W+2 is only 6.

In *Algorithm 1.3* we describe how to implement the recursive case of *Algorithm 1.1* using the lifting scheme. In this algorithm, when the buffer is empty (case 3.1), we fill it from W to 0 (W+1 is left empty), using a recursive call. Then, we compute the successive prediction and update steps, using only the lines in the buffer. So, in every step, we can compute fewer lines, since the rest of lines rely on information that still has not been input. Finally, we get a low frequency line (the first line

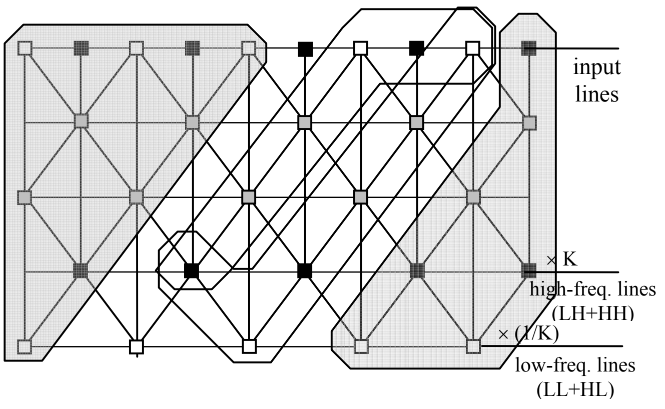

Figure 2: Overview of the lifting scheme for the B7/9 DWT

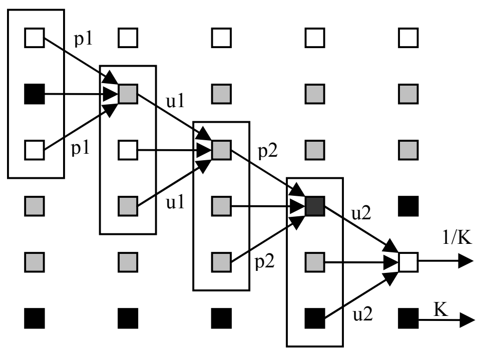

Figure 3: Line processing in a buffer for a lifting scheme

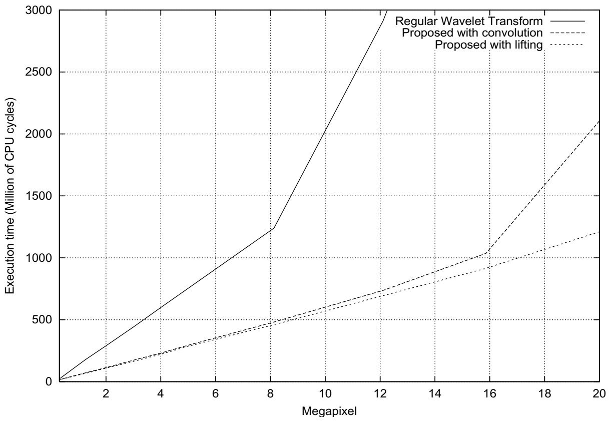

Figure 4: Execution time comparison (excluding I/O)

of the LL and HL subbands). The lines that have been handled in this step are shown in the area selected on the left of Figure 2. At this moment, all the lines in diagonal are predicted or updated. Hence, if we introduce two new lines, and discard two other lines (W+1 was empty and W was saved), we can compute two more lines in a segmented way (case 3.3). When we introduce two lines in a buffer, their lines are processed as described in Figure 3. The first column in this figure indicates the initial state, in which we have two new lines (white and black squares). Then, the odd line is predicted from its two contiguous even lines (it becomes a grey square). Afterwards, we update the third line in the buffer from the contiguous even lines, and so on. At the end of this process, we have computed two new full lines (four subband lines). The new high frequency line (represented as a black square, since it stems from an odd line) is not released because it is need for the following computation. Thus, we normalize and release the low-frequency line, and the high-frequency line that was computed in the previous stage. In Figure 2, the left dotted area represents the initial state in Figure 3 (first column), while the right dotted area is the final state (last column). Finally, the area selected on the right of Figure 2 shows the lines that are processed when no more lines can be read from the previous level (case 3.2). In this case, we use the remaining intermediate lines to generate more subband lines while we are cleaning the buffer by shifting it two positions in each call.

TABLE I. MEMORY REQUIREMENT (KB) COMPARISON.

Image size (megapixel)	Regular WT	Proposed convolution	Proposed lifting
20 (4096 x 5120)	81,980	324	205
16 (3712 x 4480)	65,013	293	186
8 (2560 x 3328)	33,319	202	128
3 (1600 x 1984)	12,423	127	80
VGA (512 x 640)	1,288	41	26

V. EXPERIMENTAL RESULTS

For the experimental tests, we have used the standard Lena (512x512) and Woman (2048x2560) images. With six decomposition levels, the regular WT needs 1030 KB for Lena and 20510 KB for Woman, while the convolution algorithm requires 41 KB for Lena and 162 KB for Woman, i.e., it uses 25 and 127 times less memory. However, the lifting proposal needs 26 KB for Lena and 102 KB for Woman, which means that it only requires 60% of memory with respect to the convolution algorithm. In addition, Table 1 shows that our proposals are much more scalable than the usual DWT.

An execution time comparison between both proposals and the regular DWT is presented in Figure 4. It shows that our proposals work faster than the regular DWT, since they use the cache memory in a better way. Thus, our algorithms have a linear behavior while the regular DWT approaches to an exponential curve. However, we see that for very big images, the convolution algorithm has an exponential behavior, because it uses more memory than the lifting one, and thus it does not fit in cache memory. For more tests, our implementation is available at http://www.disca.upv.es/joliver/lift.

VI. CONCLUSIONS

A recursive line-by-line lifting algorithm has been presented that solves the existing problems about different delays and rhythm among the buffers. It can be used as a part of compression algorithms, such as JPEG 2000, speeding up its execution time and reducing its memory requirements compared with the usual DWT algorithm.

REFERENCES

[1] Sweldens, “The lifting scheme: a custom-design construction of biorthogonal wavelets,” *Appl. Comput. Harmon. Anal.*, 1996J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.

[2] C. Chrysafis, and A. Ortega, “Line-based, reduced memory, wavelet image compression,” *IEEE Transactions on Image Processing.*, Mar. 2000.

[3] T. Acharya, P. Tsai, “JPEG 2000 Standard for Image Compression: Concepts, Algorithms and VLSI Arquitectures,” Chapter 5, Wiley, October 2005.

[4] G. Dillen, B. Georis, J. Legat, O. Cantineau, “Combined Line-Based Architecture for the 5-3 and 9-7 Wavelet Transform of JPEG 2000,” IEEE Transactions on Circuits and Systems for Video Technology, vol. 13, pp. 944-950, September 2003.

[5] N. Zervas, G. Anagnostopoulos, V. Spiliotopoulos, Y. Andreopoulos, C. Goutis, “Evaluation of Design Alternatives for the 2-D-Discrete Wavelet Transform,” IEEE Transactions on Circuits and Systems for Video Technology, vol. 11, pp. 1246-1262, December 2001.

[6] W. Chang, Y. Lee, W. Peng, C. Lee, “A Line-Based, Memory Efficient and Programmable Architecture for 2D DWT using Lifting Scheme,” International Symposium on Circuits and Systems ISCAS 2001.

[7] I. Daubechies, W. Sweldens, “Factoring wavelet transforms into lifting steps,” *J. Fourier Anal.*, no.3, 1998.