# A FAST WAVELET TRANSFORM FOR IMAGE CODING WITH LOW MEMORY CONSUMPTION

*Jose Oliver, M. P. Malumbres*

Department of Computer Engineering (DISCA)
Universidad Politécnica de Valencia
Camino de Vera 17, 46022 Valencia, Spain
e-mail: {joliver, mperez}@disca.upv.es

## ABSTRACT

In this paper, a new wavelet transform algorithm for image coding is presented. The proposed algorithm requires less memory than the regular transform, since it processes the image line by line, such as other proposals do. However, it clearly specifies how to perform the synchronization among different buffer levels, so that an implementation can be written straightforward. Experimental results show that, using a 5-Megapixel image, our recursive algorithm requires 125 times less memory and it is more than 3 times faster than the usual algorithm. Standards such as JPEG 2000 and MPEG 4 can benefit from these improvements.

## 1. INTRODUCTION

The discrete wavelet transform (DWT) is a new mathematical tool that has aroused great interest in the last years due to its nice features (multiresolution, space and frequency domains, high compactness, etc). However, one of its major drawbacks is the high memory requirements of the regular algorithms that compute it. In [1] it is introduced a first solution to overcome this drawback for the 1D DWT. In [2], this transform is extended to image wavelet transform (2D) and other issues related to the order of the data are solved. However, it is not easy to implement this algorithm due to some unclear aspects. We will address them in Section 2, while Section 3 describes the proposed solution and Section 4 shows some results.

## 2. THE LINE-BASED APPROACH

The basic idea of our proposal is the use of a line-based strategy such as that used in [2], where the key idea for saving memory is to get rid of the wavelet coefficients as soon as they have been calculated. It is clearly different to the regular DWT, in which at every decomposition level, the image is transformed first line by line, and then row by row, and so it must be kept entirely in memory.

In order to keep in memory only the part of image strictly necessary, and therefore reduce the amount of memory required, the order of the regular wavelet algorithm must be changed.

For the first decomposition level, the algorithm directly receives image lines, one by one. On every input line, a one-level 1D wavelet transform algorithm is applied so that it is divided into two parts, representing the horizontal details and a low-frequency, smaller version of this line. Then, these transformed lines are stored in a buffer associated to the first decomposition level. This buffer must be able to keep 2N+1 lines, where 2N+1 is the number of taps for the largest analysis filter bank. We only consider odd filter lengths because they have higher compression efficiency, however this analysis could be extended to even filters as well.

When there are enough lines in the buffer to perform

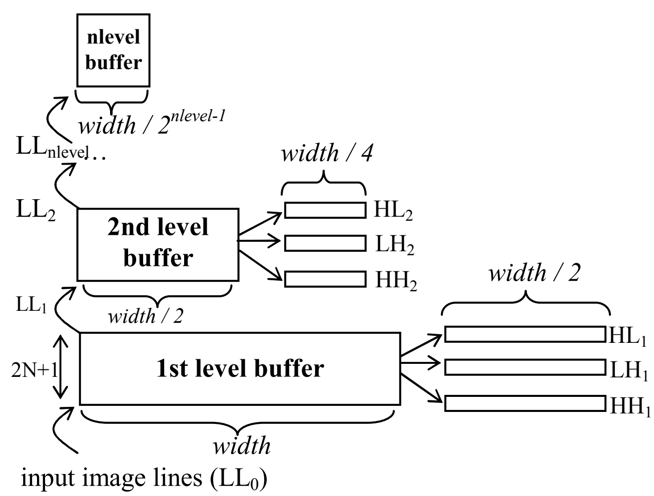


Figure 1: Overview of a line-based forward wavelet transform

```
function GetLLlineBwd( level )
1) First base case: No more lines to read at this level
    if LinesRead_level = MaxLines_level
        return EOL
2) Second base case: The current level belongs to the space domain and not to the wavelet domain
    else if level = 0
        return ReadImageLineIO( )
    else
3) Recursive case
3.1) Recursively fill or update the buffer for this level
    if buffer_level is empty
        for i = N...2N
            buffer_level(i) = 1DFWT(GetLLlineBwd( level-1))
        FullSymmetricExtension( buffer_level )
    else
        repeat twice
            Shift( buffer_level )
            line = GetLLlineBwd( level-1 )
            if line = EOL
                buffer_level(2N) = SymmetricExt( buffer_level )
            else
                buffer_level(2N) = 1DFWT( line )
3.2) Calculate the WT from the lines in the buffer, then process the resulting subband lines (LL, HL, LH and HH)
    {LLline, HLline} = ColumnFWT_LowPass( buffer_level )
    {LHline, HHline} = ColumnFWT_HighPass( buffer_level )
    ProcessHighFreqSubLines( {HLline, LHline, HHline} )
    set LinesRead_level = LinesRead_level + 1
    return LLline
end of fuction
```

Algorithm 1.1: Backward recursive function

```
function LowMemUsageFWT( nlevel )
    set LinesRead_level = 0   ∀level ∈ nlevel
    set MaxLines_level = height / 2^level   ∀level ∈ nlevel
    set buffer_level = empty   ∀level ∈ nlevel
    repeat height / 2^nlevel times
        LLline = GetLLlineBwd( nlevel )
        ProcessLowFreqSubLine( LLline )
end of function
```

Algorithm 1.2: Perform the FWT by calling a backward recursive function (see Algorithm 1.1)

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. The result of this operation is the first line of the $HL_1$, $LH_1$ and $HH_1$ wavelet subbands, and the first line of the $LL_1$ subband.

At this moment, for a dyadic wavelet decomposition, we can process and release the first line of the wavelet subbands. However, the first line of the $LL_1$ subband does not belong to the final result, but it is needed as incoming data for the following decomposition level. On the other hand, once the lines in the first level buffer have been used, this buffer is shifted twice (using a rotation operation) so that two lines are discarded while another two image lines are input at the other end. Once the buffer is updated, the process can be repeated and more lines are obtained.

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 in the very same way as we have described for the first level. In this manner, the lines of the second wavelet subbands are achieved, and the low-frequency lines from $LL_2$ are passed to the third level.

As it is depicted in Figure 1, this process can be repeated until the desired decomposition level (*nlevel*) is reached.

Although this algorithm might seem quite simple, a major problem arises when it is implemented. This drawback is the synchronization among the buffers. Before a buffer can produce lines, it must be completely filled with lines from previous buffers, therefore they start working at different moments, i.e., they have different delays. Moreover, all the buffers exchange their result at different intervals, according to their level.

Handling several buffers with different delay and rhythm becomes a hard task. The next section proposes a recursive algorithm that clearly specifies how to perform this communication between buffers.

## 3. A RECURSIVE ALGORITHM FOR BUFFER SYNCHRONIZATION

In this section, we present both Forward and Inverse Wavelet Transform algorithms (FWT and IWT), which automatically solves the synchronization problem among levels that has been addressed in Section 2. In order to solve this problem, both algorithms define a recursive function that obtains the next low-frequency subband (LL) line from a contiguous level.

### 3.1. FWT with Backward Recursion

The FWT starts requesting LL lines to the last level (*nlevel*). As seen in Figure 1, the *nlevel* buffer must be filled with lines from the *nlevel*-1 level before it can generate lines. In order to get them, the function recursively call itself until the level 0 is reached. At this point, it no longer needs to call itself since it can return an

```
function GetLLlineFwd( level )
1) First base case: Buffer ready for another wavelet step
    if lineinbuffer_level
            set lineinbuffer_level = false
            set LinesRead_level = LinesRead_level + 1
            return ColumnIWT_HighPass( buffer_level )
2) Second base case: All the lines have been read
    else if LinesRead_level = MaxLines_level
            return EOL
3) Third base case: The last level is accessed directly
    else if level = nlevel
            return RetrieveLowFreqSubLine ( )
    else
4) Recursive case
4.1) Recursively fill or update the buffer for this level
    if buffer_level is empty
        for i = N'...2N'
            buffer_level(i) = 1DIWT( BuildLine(level ) )
        FullSymmetricExtension( buffer_level )
    else
        repeat twice
            Shift( buffer_level )
            line = BuildLine( level)
            if line = EOL
                buffer_level(2N') = SymmetricExt( buffer_level )
            else
                buffer_level(2N') = 1DIWT( line )
4.2) Calculate one IWT step from the lines in the buffer
    set lineinbuffer_level = true
    set LinesRead_level = LinesRead_level + 1
    return ColumnIWT_LowPass( buffer_level )
end of fuction
```

Algorithm 2.1: Forward recursive function

image line that can be read directly from the input/output system. Notice that although we are calculating a forward wavelet transform, we do it by means of a backward recursion, since it goes from *nlevel* to 0.

The complete recursive algorithm is formally described in the frame entitled *Algorithm 1.1*, while *Algorithm 1.2* sets the variables up and performs the FWT by calling the recursive algorithm. Let us see the first algorithm more carefully.

The first time that the recursive function is called at every level, it has its buffer ($buffer_{level}$) empty. Then, its upper half (from N to 2N) is recursively filled with lines from the previous level. Recall that once a line is received, it must be transformed using a 1D FWT (either convolution [3] or lifting [4]) before it is stored. Once the

```
subfunction BuildLine(level )
    if OddAccess_level
        OddAccess_level = false
        LLline = GetLLlineFwd(level+1)
        HLline= RetrieveHL_SubLine(level)
        return LLline + HLline
    else
        OddAccess_level = true
        {LHline, HHline}=RetrieveLH_HH_SubLines(level)
        return LHline + HHline
end of subfunction
```

Algorithm 2.2: Subfuction used for Algorithm 2.1

```
function LowMemUsageIWT( nlevel )
    set LinesRead_level = 0    ∀level ∈ nlevel
    set MaxLines_level = height / 2^level    ∀level ∈ nlevel
    set buffer_level = empty    ∀level ∈ nlevel
    set lineinbuffer_level = false    ∀level ∈ nlevel
    set OddAccess_level = true    ∀level ∈ nlevel
    repeat height times
            imageLine = GetLLlineFwd( 0 )
            WriteImageLineIO( imageLine )
end of function
```

Algorithm 2.3: Perform the IWT by calling a forward recursive function (see Algorithm 2.1)

upper half is full, the lower half is filled using symmetric extension (the N+1 line is copied into the N-1 position, …, the 2N is copied into the 0 position).

On the other hand, if the buffer is not empty, it simply has to be updated. In order to update it, it is shifted one position so that the line contained in the first position is discarded and a new line can be introduced in the last position (2N) using a recursive call. This operation is repeated twice.

However, if there are no more lines in the previous level, this recursive call will return *End Of Line* (EOL). That points out that we are about to finish the computation at this level, but we still need to continue filling the buffer. We fill it using symmetric extension again.

Once the buffer is filled or updated, both high-pass and low-pass filter banks are applied to every column in the buffer. As result of the convolution, we get a line of every wavelet subband at this level, and a LL line. The wavelet coefficients are processed according to the application (compressed, saved to secondary storage, etc.) and this function returns the LL line.

Every recursive function needs at least one base case to stop backtracking. This function has two base cases.

The first case is when all the lines at this level have been read. It is detected by keeping an account of the number of lines read and the maximum number of lines that can be read at every level. 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 need since an image line can be read directly.

### 3.2. IWT with Forward Recursion

The IWT algorithm is described in Algorithms *2.1, 2.2* and *2.3*. This algorithm receives the subband lines that have been calculated in the FWT, and it recovers the original image lines. Both algorithms 2.1 and 2.3 are similar to the corresponding FWT 1.1 and 1.2, however some differences need to be explained.

The main difference is that the recursion is carried out forward, starting from 0 to *nlevel*. Then, at that level, the $LL_{nlevel}$ subband lines are retrieved directly.

Moreover, in Algorithm 2.1, the lines for the buffers are obtained using a subfunction described in Algorithm 2.2. This function iteratively returns the concatenation of a line from the LL and from the HL subbands, or the concatenation of a line from LH and from HH. Notice that the lines from HL, LH and HH are retrieved directly from the input data associated to that level, but the LL line has to be achieved recursively from the following level.

The last difference is that once a buffer is full, two LL lines can be returned, a line from every column convolution. For this reason, $lineinbuffer_{level}$ shows if there is any line in the buffer that has been calculated previously, and therefore it can be returned directly. It becomes the first base case in *Algorithm* 2.1.

### 3.3. Some theoretical considerations

The main advantage of line-based algorithms is its lower memory requirement compared to the regular wavelet transform. In these algorithms, every buffer needs to store $(2N+1)\times BufferWidth$ coefficients. If the image width is $w$, then the memory requirements for all the buffer is $(2N+1)\times w+(2N+1)\times w/2+\ldots+(2N+1)\times w/2^{nlevel}$, which is asymptotically $2\times(2N+1)\times w$, considerably lower than the $width\times height$ coefficients required by the regular WT. This reduction in the amount of memory has another beneficial side effect when the algorithm is implemented in a computer. The subband buffers are more likely to fit in the cache memory than the whole image, and thus the execution time is substantially reduced.

A drawback that has not been considered yet is the need to reverse the order of the subbands, from the FWT to the IWT. The former starts generating lines from the first levels to the last ones, while the latter needs to get lines from the last levels before getting lines from the first ones. This problem can be resolved 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 image compression, to keep the compressed coefficients in memory.

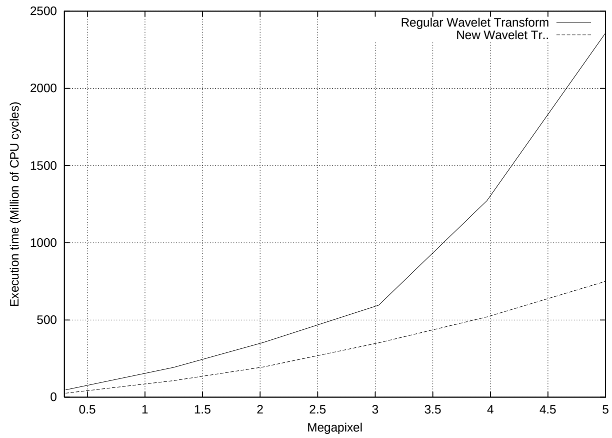

Figure 2: Execution time comparison (excluding I/O)

| Image size (megapixel) | Regular WT | Proposed WT |
|---|---|---|
| 5 (2048 x 2560) | 20510 | 162 |
| 4 (1856 x 2240) | 16266 | 146 |
| 3 (1600 x 1984) | 12423 | 126 |
| 2 (1280 x 1664) | 9339 | 101 |
| 1 (1024 x 1280) | 5135 | 80 |
| VGA (512x640) | 1287 | 40 |

Table 1: Memory requirement (KB) comparison

## 4. EXPERIMENTAL RESULTS

We have implemented the regular Wavelet Transform and our proposal with the B7/9 filter bank [5], using standard ANSI C language on a regular PC computer with 256 KB L2 cache. This implementation can be downloaded at http://www.disca.upv.es/joliver/WT.

We have used the standard Lena (512x512) and Woman (2048x2560) images. With six decomposition levels, our algorithm requires 40 KB for Lena and 162 KB for Woman, while the regular WT needs 1030 KB for Lena and 20510 KB for Woman, i.e., it uses 25 and 127 times more memory. In addition, Table 1 shows that our algorithm is much more scalable than the usual WT.

An execution time comparison between both algorithms can be seen in Figure 2. It shows that, while our algorithm has a linear behavior, the regular WT approaches to an exponential curve. This behavior happens because our algorithm fits in cache for all the image sizes (162 KB for the 5-megapixel image). On the contrary, the usual WT rapidly exceeds the cache limits (1287 KB for the VGA resolution).

Further experiments have shown that the IWT has similar results in memory requirement and execution time.

## 5. CONCLUSIONS

A line-by-line transform algorithm has been presented that solves the existing problem about different delay 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.

## 6. REFERENCES

[1] M. Vishwanath, “The recursive pyramid algorithm for the discrete wavelet transform,” *IEEE Tr. Signal Proc.*, Mar. 1994

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

[3] S. Mallat, “A theory for multiresolution signal decomposi-tion,” *IEEE Tr. Patter Anal. and Machine Intelligence*,July 1989

[4] Sweldens, “The lifting scheme: a custom-design construction of biorthogonal wavelets,” *Appl. Comput. Harmon. Anal.*, 1996

[5] M.Antonini, Barlaud, P.Mathieu, and I.Daubechies, “Image coding using wavelet transform,” *IEEE Tr. Image Proc.*, 1992