A Fast Run-Length Algorithm for Wavelet Image Coding with Reduced Memory Usage

Jose Oliver, M.P. Malumbres

Department of Computer Engineering (DISCA),
Technical University of Valencia
Camino de Vera 17, 46017, Spain
{joliver, mperez}@disca.upv.es

Abstract. A new image coder is described in this paper. Since it is based on the Discrete Wavelet Transform (DWT), it yields good Rate/Distortion (R/D) performance. However, our proposal focuses on overcoming the two main problems of wavelet-based image coders: they are typically implemented by memory-intensive and time-consuming algorithms. In order to avoid these common drawbacks, we ought to tackle these problems in the main stages of this type of coder, i.e., both the wavelet computation and the entropy coding of the coefficients. The proposed algorithms are described in such a manner that they can be implemented in any programming language straightforwardly. The numerical results show that while the R/D performance achieved by our proposal is similar to the state-of-the-art coders, such as SPIHT and JPEG2000/Jasper, the amount of memory required in our algorithm is reduced drastically (in the order of 25 to 35 times less memory), and its execution time is lower (three times lower than SPIHT, and more than ten times lower than JPEG 2000/Jasper).

1 Introduction

Wavelet-based image coders have aroused great interest in the last years due to their nice features, such as natural multiresolution and high compactness of the coefficients, which leads to high compression efficiency. However, one of the main drawbacks of current wavelet encoders is their high memory usage, since the regular wavelet transform requires a lot of memory to be computed. In addition, in many wavelet encoders, the subsequent coding process uses some extra lists and introduces memory overhead. The complexity of these algorithms is another usual problem. In this paper, we deal with both problems (memory requirement and complexity) in both stages (wavelet transform and efficient coding).

2 Wavelet Transform for Image Coding with Low Use of Memory

One of the desirable features of the proposed image coder is to have low memory consumption. Since our proposal is a wavelet-based coder, the first bottleneck that

```
function GetLLlineBwd( level )

  1) First base case: No more lines to be 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) = 1D_DWT(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) = 1D_DWT( line )

  3.2) Calculate the WT from the lines in the buffer, then process the result-
        ing subband lines (LL, HL, LH and HH)
    {LLline, HLline} = ColumnDWT_LowPass( buffer_level )
    {LHline, HHline} = ColumnDWT_HighPass( buffer_level )
    EncodeSubLines( {HLline, LHline, HHline}, level )
    set LinesRead_level = LinesRead_level + 1
    return LLline

end of fuction
```

Algorithm 1.1: Backward recursive function

appears in the efficient use of memory is the computation of the DWT. Our encoder could only have low memory consumption if the DWT is performed in an efficient way. In the regular DWT, Mallat decomposition is performed [1]. In this decomposition, the image is transformed first row by row, and then column by column, at every decomposition level. Therefore, it must be kept entirely in memory. In this section we propose a different wavelet transform in which the key idea for saving memory is to get rid of the wavelet coefficients as soon as they have been calculated.

This idea was first used in [2], aiming to reduce the memory requirements of the 1D DWT. In [3], this transform is extended to image wavelet transform (2D), and other issues related to the order of the data are solved. However, in this 2D version, the authors do not propose a direct algorithm to implement their proposal, and it cannot be easily implemented due to some unclear aspects. In [4], we presented a general-purpose recursive algorithm that we will use in the image coder presented in this

program Code_Image (*nlevel, Q , rplanes*)

set $LinesRead_{level} = run\_length_{level} = 0 \quad \forall level \in nlevel$

set $MaxLines_{level} = \frac{height}{2^{level}} \quad \forall level \in nlevel$

set $buffer_{level} = EncBufferHL_{level} = EncBufferLH_{level} = EncBufferHH_{level} = empty \ \forall level \in nlevel$

repeat $\frac{height}{2^{nlevel}}$ **times**

LLline = GetLLlineBwd(*nlevel*)

EncodeLLSubLine(*LLline*)

end of program

Algorithm 1.2: Perform the DWT and encode the image by calling a backward recursive function (see Algorithm 1.1)

paper. In this section, this wavelet transform is outlined, while the reader is referred to [4] for a more complete and exhaustive description.

The proposed algorithm relies on a line-based strategy. In this strategy, we only keep in memory those image lines that we are dealing with. This way, there is a buffer in each level that is able to keep 2N+1 lines for the low-frequency subband (LL) at that level (2N+1 is the length of the filter bank). These buffers are filled so that, when they are full, one-step of a column wavelet transform is performed. This operation generates a line of every wavelet subband (HH, HL and LH at that level), and a LL line. The HH, HL and LH lines can be directly encoded, while the LL line is passed to the following level in order to fill its buffer up.

The drawback of this algorithm is the synchronization among the buffers. Before a buffer can produce lines, it must be 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.

To solve the synchronization problem, we define a recursive function called GetLLlineBwd (*level*), which obtains the next LL line from a contiguous level. This algorithm is formally described in the frame *Algorithm 1.1*, while *Algorithm 1.2* defines the main program that sets up some variables and performs the image transform by calling the recursive function. Let us see the first algorithm more carefully.

The first time that the recursive function is called at every level, its buffer ($buffer_{level}$) is empty and it has to be filled up. So, its upper half (from N to 2N) is recursively filled with lines from the previous level. When a line is received, it must be transformed using a 1D DWT before it is stored. The lower half part is filled using symmetric extension (the N+1 line is copied into the N-1 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 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 fill the buffer up using symmetric extension again.

```
function EncodeSubLines( {HLline, LHline, HHline} , level )
   AddToBuffer ( EncBufferHL_level , HLline)
   AddToBuffer ( EncBufferLH_level , LHline)
   AddToBuffer ( EncBufferHH_level,, HHline)
   if IsFull (EncBufferHL_level )
         RLW_Code_Subband ( EncBufferHL_level , level)
         RLW_Code_Subband ( EncBufferLH_level , level)
         RLW_Code_Subband ( EncBufferHH_level,, level)
         EncBufferHL_level = EncBufferLH_level = EncBufferHH_level =empty
end of function
```

Algorithm 2.1: Store the subband lines in the encoder buffer and call the run-length coding function when they are full

Once the buffer is filled or updated, both high-pass and low-pass filter banks are applied to every column in the buffer. This way, we get a line of every wavelet subband at this level, and a LL line. The wavelet coefficients are passed to the coder so that they can be compressed, and the function returns the LL line.

Notice that this function has two base cases. In the first one, all the lines at this level have been read. It is detected by keeping an account of the number of lines read, and it returns EOL. In the second one, the variable *level* reaches 0 and then no further recursive call is need since an image line can be read directly. Moreover, the maximum recursion depth is given by the decomposition level (which is usually 5 or 6), and so the memory usage for recursion is negligible compared with the buffer sizes.

3 Run-Length Coding of the Wavelet Coefficients

In order to have low memory consumption, once a wavelet subband line is calculated, it has to be encoded as soon as possible to release memory. However, we cannot encode independent lines if we want good R/D performance, since entropy coders need to exploit local similarities in the image to be efficient. *Algorithm 2.1* stores the subband lines in encoder buffers so that when they are full, there are enough lines to perform an efficient compression, and the coding function is called.

The encoder cannot use global image information since it does not know the whole image. Moreover, we aim at fast execution, and hence no R/D optimization or bit-plane processing can be made, because it would turn it slower. In the next subsection, a Run-Length Wavelet (RLW) encoder with the aforementioned features is proposed.

3.1 Fast Run-Length Coding

In the proposed algorithm, the quantization process is performed by two strategies: one coarser and another finer. The finer one consists on applying a scalar uniform quantization to the coefficients using the *Q* parameter (see *Algorithm 1.2*). The

coarser one is based on removing bit planes from the least significant part of the coefficients. We define *rplanes* as the number of less significant bits to be removed, and we call significant coefficient to those coefficients $c_{i,j}$ that are different to zero after discarding the least significant *rplanes* bits, in other words, if $c_{i,j} \geq 2^{rplanes}$.

The wavelet coefficients are encoded as follows. The coefficients in the buffer are scanned column by column (to exploit their locality). For each coefficient in that buffer, if it is not significant, a run-length count of insignificant symbols at this level is increased (run_length_L). However, if it is significant, we encode both the count of insignificant symbols and the significant coefficient, and run_length_L is reset.

The significant coefficient is encoded by means of a symbol indicating the number of bits required to represent that coefficient. An arithmetic encoder with two contexts is used to efficiently store that symbol. As coefficients in the same subband have similar magnitude, an adaptive arithmetic encoder is able to represent this information in a very efficient way. However, we still need to encode its significant bits and sign. They are raw encoded to speed up the execution time.

In order to encode the count of insignificant symbols, we encode a *RUN* symbol. After encoding this symbol, the run-length count is stored in a similar way as in the

function RLW_Code_Subband(*Buffer*, *L*)
Scan *Buffer* in horizontal raster order (i.e., in columns)
for each $c_{i,j}$ in *Buffer*
$nbits_{i,j} = \lceil \log_2 (|c_{i,j}|) \rceil$
if $nbits_{i,j} \leq rplanes$
increase run_length_L
else
if $run\_length_L \neq 0$
if $run\_length_L < enter\_run\_mode$
repeat run_length_L **times**
arithmetic_output *LOWER*
else
arithmetic_output *RUN*
$rbits = \lceil \log_2 (run\_length_L) \rceil$
arithmetic_output *rbits*
output $\text{bit}_{rbits-1}(run\_length_L) \ldots \text{bit}_1(run\_length_L)$
$run\_length_L = 0$
arithmetic_output $nbits_{i,j}$
output $\text{bit}_{nbits_{(i,j)}-1}(|c_{i,j}|) \ldots \text{bit}_{rplane+1}(|c_{i,j}|)$
output $\text{sign}(c_{i,j})$
end of function
Note: $\text{bit}_n(c)$ is a function that returns the n^{th} bit of *c*

Algorithm 2.2: Run-length coding of the wavelet coefficients

significant coefficients. First, the number of bits needed to encode the run value is arithmetically encoded (with a different context), afterwards the bits are raw encoded.

Instead of using run-length symbols, we could have used a single symbol to encode every insignificant coefficient. However, we would need to encode a larger amount of symbols, and therefore the complexity of the algorithm would increase (most of all in the case of large number of insignificant contiguous symbols, which usually occurs in moderate to high compression ratios).

Despite of the use of run-length coding, the compression performance is increased if a specific symbol is used for every insignificant coefficients, since an arithmetic encoder stores more efficiently many likely symbols than a lower amount of less likely symbols. So, for short-run lengths, we encode a *LOWER* symbol for each insignificant coefficient instead of coding a run-length symbol for all the sequence. The threshold to enter the run-length mode and start using run-length symbols is defined by the *enter_run_mode* parameter. The formal description of the depicted algorithm can be found in the frame entitled *Algorithm 2.2*.

3.2 Tradeoff between R/D Performance and Speed and Memory Requirements

The proposed algorithm can be tuned according to the final application. Thus, some parameters can be adjusted to improve the compression performance at the cost of slightly higher memory requirements or execution time. This way, the size of the encoder buffer can be 8 subband lines for a good R/D performance, but compression efficiency can be improved with 16 lines, increasing the memory requirements. Another parameter that can be tuned is the *enter_run_mode* variable in *Algorithm 2.2*. When this parameter is increased, larger run-lengths are encoded by successive *LOWER* symbols, which results slower but a bit more efficient in R/D performance. Another tradeoff between compression and complexity is the use of an arithmetic encoder (with nine contexts) for the sign of the coefficients. In general, each of these improvements may increase the PSNR of an image encoded at 1bpp in about 0.1 dB, while the two latter improvements increase the execution time in about 20% each one.

4 Numerical Results

We have implemented the proposed coder in ANSI C language. In this section we will compare it with the state-of-the-art wavelets coders SPIHT [5] and JPEG 2000 [6]. For JPEG 2000, we do not consider image tiling since it degrades the image quality a lot. The results for JPEG 2000 have been obtained using Jasper [7], an official implementation included in the ISO/IEC 15444-5 standard. All of them use the same wavelet filter bank (Daubechies' B7/9) and have been written and compiled with the same level of optimization. In our comparison, we will use the standard images Lena and Barbara (monochrome, 8bpp, 512x512), and the larger and less blurred images Café and Woman (monochrome, 8bpp, 2560x2048, equiv. 5-Megapixel), from the JPEG 2000 testbed. For more tests, the reader can download an implementation of the coder at the authors' web site http://www.disca.upv.es/joliver/LowMemRLW.

Table 1. PSNR (dB) with different bit rates and coders for the evaluated images. The numbers in parenthesis for our proposal correspond to the decrease of performance if the R/D improvements discussed in subsection 3.2 are not applied.

	Lena (512x512)			Barbara (512x512)		
Codec\ rate	SPIHT	Jasper/ JP2K	Proposed Run Length	SPIHT	Jasper/ JP2K	Proposed Run Length
1	40.41	40.31	40.37 (-0.14)	36.41	37.11	36.82 (-0.35)
0.5	37.21	37.22	37.15 (-0.10)	31.39	32.14	31.90 (-0.29)
0.25	34.11	34.04	34.03 (-0.08)	27.58	28.34	28.12 (-0.22)
0.125	31.10	30.84	30.97 (-0.04)	24.86	25.25	25.19 (-0.08)
	Woman (2560x2048)			Café (2560x2048)		
Codec\ rate	SPIHT	Jasper/ JP2K	Proposed Run Length	SPIHT	Jasper/ JP2K	Proposed Run Length
1	38.28	38.43	38.49 (-0.21)	31.74	32.04	31.89 (-0.26)
0.5	33.59	33.63	33.72 (-0.15)	26.49	26.80	26.67 (-0.16)
0.25	29.95	29.98	30.04 (-0.08)	23.03	23.12	23.10 (-0.12)
0.125	27.33	27.33	27.40 (-0.04)	20.67	20.74	20.67 (-0.06)

Table 2. Total memory requiered (in KB) to encode the Woman image with the compared algorithms. The numbers in parenthesis correspond to the memory that is saved if the R/D improvements are not used (it can be applied in both columns of our proposed algorithm).

Codec \ rate	Compressed Image File	SPIHT	Jasper/ JP2K	Proposed Run Length	Proposed with bit-stream in memory	
1	640	42,888	62,768	1,256	1,896	(-180)
0.5	320	35,700	62,240	1,192	1,512	(-180)
0.25	160	31,732	61,964	1,192	1,352	(-180)
0.125	80	28,880	61,964	1,176	1,256	(-180)

Table 1 shows a compression comparison for the evaluated images and coders. In general, our proposal performs as well as SPIHT does for less detailed images (Lena and Woman) and better than it for more complex images (Barbara and Café). It is due to the fact that SPIHT is based on coefficients trees, and fewer trees can be established in images with many details. On the contrary, JPEG 2000 is more efficient than our proposal in highly detailed images, since it defines more contexts and uses R/D optimization. However, our coder and JPEG 2000 are similar in low detailed images.

The comparison in which our encoder clearly outperforms both SPIHT and JPEG 2000 is in memory consumption. Table 2 shows that, for a 5-Megapixel image, our proposal requires between 25 and 40 times less memory than SPIHT, and more than 45 times less memory than Jasper/JPEG 2000. In this table, the last column refers to the case in which the complete bitstream (i.e., the compressed image) is kept in memory while it is generated. Due to the computation order in the proposed wavelet transform, the coefficients from different subband levels are interleaved. Thus, instead of a single bitstream, we generate a different bitstream for every level. These different streams can be kept in memory or saved in secondary storage. In addition, having a different bitstream for each level eases the decompression process, since the order in the inverse transform is just the reverse of the order in the forward one.

In this table, the memory estimated for executing a single process is about 650 KB. Hence, we can consider that the remaining memory is the data memory. Moreover, for our RLW coder, 180 KB can be saved if we use 8 lines per buffer instead of 16.

Since JPEG 2000 has more contexts and uses R/D optimization, it is more complex than our proposal. SPIHT is also more complex because it performs several image scans handling a different bit-plane each scan. Moreover, in cache-based systems, the proposed DWT makes better use of the cache. The last table shows an execution time comparison for two image sizes. Due to the former reasons, our algorithm clearly outperforms Jasper/JPEG 2000, and it is several times faster than SPIHT. In addition, we can speed it up in about 30% if no compression improvements are performed.

Table 3. Execution time (in Million of CPU Cycles) needed to encode images of different size. The numbers in parenthesis correspond to time reduction if no R/D improvements are applied.

	Woman (2560x2048)			Lena (512x512)		
Codec \ rate	SPHIT	Jasper / JP2K	Proposed Run Length	SPHIT	Jasper / JP2K	Proposed Run Length
1	3,669	23,974	1,855 (-587)	147	750	98 (-28)
0.5	2,470	23,864	1,291 (-377)	97	734	65 (-21)
0.25	1,939	23,616	970 (-259)	73	726	44 (-11)
0.125	1,651	23,563	783 (-197)	60	717	34 (-7)

5 Conclusions

In this paper, a wavelet image coder with state-of-the-art compression performance has been presented. The main contribution of this image coder is that it requires much less memory to work and thus, it is a good candidate for many embedded systems and other memory-constrained environments (such as digital cameras and PDAs). In addition, it is also several times faster than the other evaluated wavelet image coders.

References

1. S. Mallat: A theory for multiresolution signal decomposition. IEEE Transactions on Pattern Analysis and Machine Intelligence, July 1989
2. M. Vishwanath: The recursive pyramid algorithm for the discrete wavelet transform. IEEE Transactions on Signal Processing, March 1994
3. C. Chrysafis, and A. Ortega: Line-based, reduced memory, wavelet image compression. IEEE Transactions on Image Processing, March 2000
4. J.Oliver, M.P.Malumbres: A fast wavelet transform for image coding with low memory consumption. 24th Picture Coding Symposium, December 2004
5. A. Said, A. Pearlman: A new, fast, and efficient image codec based on set partitioning in hierarchical trees. IEEE Trans. on Circuits and Systems for Video Technology, June 1996
6. ISO/IEC 15444-1: JPEG 2000 image coding system, 2000
7. M. Adams: Jasper software reference manual. ISO 1/SC 29/WG 1 N 2415, October 2002