A FAST 3D-DWT VIDEO ENCODER WITH REDUCED MEMORY USAGE SUITABLE FOR IPTV

O. López[a], M. Martínez-Rach[a], P. Piñol[a], M.P. Malumbres[a], J. Oliver[b]

[a]Miguel Hernández University, [b]Universidad Politécnica de Valencia
[a]Avda. Universidad s/n 03202 Elche - Spain, [b]c/ Camino de Vera s/n 46222 Valencia - Spain
Email: [a](otoniel,mmrach,pablop,mels)@umh.es, [b]joliver@disca.upv.es

ABSTRACT

The 3D-DWT is a mathematical tool of increasing importance in those applications that require an efficient processing of volumetric info. Other applications like professional video editing, IPTV video surveillance applications, live event IPTV broadcast, multi-spectral satellite imaging, HQ video delivery, etc, would rather use 3D-DWT encoders to reconstruct a frame as fast as possible. However, the huge memory requirement of the algorithms that compute the 3D-DWT is one of the main drawbacks in practical implementations. In this paper, we introduce a fast frame-based 3D-DWT video encoder with low memory usage. In addition, there is no need to divide the input video sequence into group of pictures (GOP), and it can be applied in a continuous manner, so that no boundary effects between GOPs appear.

Keywords— 3D-DWT, wavelet-based video coding, IPTV surveillance

1. INTRODUCTION

In recent years, three-dimensional wavelet transform (3D-DWT) has focused the attention of the research community, most of all in areas such as video watermarking [1] and 3D coding (e.g., compression of volumetric data [2] or multispectral images [3], 3D model coding [4], and specially, video coding). These encoders are good candidates for some applications like professional video editing, IPTV video surveillance applications (Traffic cameras, child/day care, mall cctv surveillance), live event IPTV broadcast, multi-spectral satellite imaging, HQ video delivery, etc., where a specific frame of a video sequence must be reconstructed as fast as possible and with high visual quality.

In video compression, some early proposals were based on merely applying the wavelet transform on the time axis after computing the 2D-DWT for each frame [5]. Then, an adapted version of an image encoder can be used, taking into account the new dimension. For instance, instead of the typical quad-trees of image coding, a tree with eight descendants per coefficient is used in [5] to extend the SPIHT image encoder to 3D video coding. A more efficient strategy for video coding with time filtering is Motion Compensated Temporal Filtering (MCTF) [6, 7]. In these techniques, in order to compensate object (or pixel) misalignment between frames, and hence avoid the significant amount of energy that appears in high-frequency subbands, a motion compensation algorithm is introduced to align all the objects (or pixels) in the frames before being temporally filtered.

In all these applications, the first problem that arises is the extremely high memory consumption of the 3D wavelet transform if the regular algorithm is used, since a group of frames must be kept in memory before applying temporal filtering, and in the case of video coding, we know that the greater temporal decorrelation, the greater number of frames are needed in memory. Another drawback is the necessity of grouping images in small Group Of Pictures (GOP) to prevent very high memory usage, because the 3D-DWT must be computed along a set of images which are held in memory. This video sequence division into GOPs containing only a few images hinders the decorrelation of the temporal dimension and causes boundary effects between GOPs.

Even though several proposals have been made to avoid the aforementioned problems, most of them are not general (for any wavelet transform) and/or complete (the wavelet coefficients are not the same as those from the usual dyadic wavelet transform). In addition, software implementation is not always easy. In this paper, we propose a video encoder based on a frame-by-frame 3D-DWT scheme which does not require a GOP division, significantly reduces the memory usage and performs the 3D-DWT much faster than traditional algorithms.

2. 3D-DWT WITH LOW MEMORY USAGE

In this section we propose an extension to a three-dimensional wavelet transform of the classical line-based approach [8], which computes the 2D-DWT with reduced memory consumption. In the new approach, frames are continuously input with no need to divide the video sequence into GOPs. Moreover, the algorithm yields slices of wavelet subbands (which we call subband frames) as soon as it has enough frames to compute them. This approach works as follows:

For the first decomposition level the algorithm directly receives frames one by one. On every input frame, a one-level 2D-DWT is applied. Then, this transformed image is stored in a

Thanks to Spanish Ministry of education and Science under grant DPI2007-66796-C03-03 for funding.

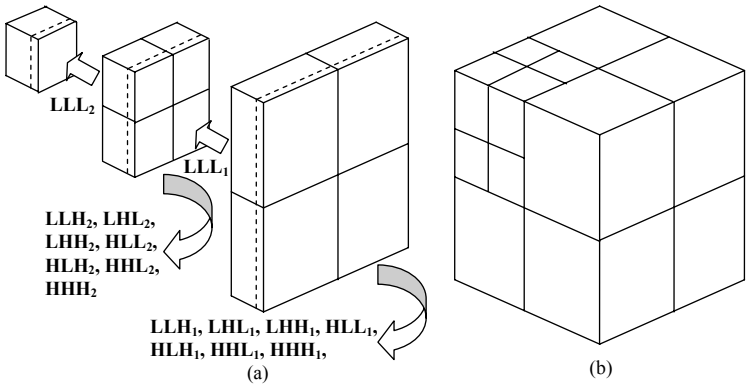


Fig. 1. Overview of the 3D-DWT computation in a two-level decomposition, (a) following a frame-by-frame scheme as shown in Figure 2; or, (b) the regular 3D-DWT algorithm

buffer associated to the first decomposition level. This buffer must be able to keep 2N+1 frames, where 2N+1 correspond with the number of taps for the largest analysis filter bank in the temporal direction. 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 frames in the buffer to perform one step of a wavelet transform in the temporal direction (z-axis), the convolution process is calculated twice, first using the low-pass filter and then the high-pass filter. The result of this operation is the first frame of each high-frequency subbands (the HHL_1, HLH_1, HHH_1, HLL_1, LHL_1, LLH_1 and LHH_1 wavelet subbands), and the first frame of the LLL_1 subband. At this moment, for a dyadic wavelet decomposition, we can process and release the first frame of the wavelet subbands. However, the first frame of the LLL_1 subband does not belong to the final result, since it represents the incoming data for the following decomposition level. On the other hand, once the frames at the first level buffer have been used, this buffer is shifted twice (using a rotation operation) so that two frames are discarded while another two frames are inputted at the other end. Once the buffer is updated, the process can be repeated and more subband frames are obtained.

At the second decomposition level, its buffer is filled with the LLL_1 frames 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 frames of the second level wavelet subbands are achieved, and the low-frequency frames from LLL_2 are passed to the third level. As depicted in Figure 1(a), this process can be repeated until the desired decomposition level (*nlevel*) is reached.

In this algorithm a major problem arises when it is implemented. This drawback is the synchronization among buffers. Before a buffer can produce frames, it must be completely filled with frames 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 delays and rhythms becomes a hard task. To solve the synchronization problem, the algorithm depicted at Figure 2 defines a recursive function that obtains the next low-frequency subband frame (*LLL*) from a contiguous level in a similar way as authors in [9] proposed for the 2D-DWT.

```
function LowMemUsage3D_FWT(nlevel)
  set FramesRead_level = 0 ∀level ∈ nlevel
  set FramesLines_level = Nframes / 2^level ∀level ∈ nlevel
  set buffer_level = empty ∀level ∈ nlevel
  repeat
      LLL = GetLLLframe(nlevel)
      if (LLL != EOF) ProcessLowFreqSubFrame(LLL)
  until LLL = EOF
end of fuction
```

Fig. 2. Perform the 3DFWT by calling GetLLLFrame recursive function

The algorithm starts requesting *LLL* frames to the last level (*nlevel*). As seen in Figure 1, the *nlevel* buffer must be filled with subband frames from the *nlevel*-1 level before it can generate frames. In order to get them, this function recursively calls itself until level 0 is reached. At this point, it no longer needs to call itself since it can return a frame from the video sequence, which can be directly read from the input/output system.

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 frames from the previous level. Recall that once a frame is received, it must be transformed using a 2D-DWT before being stored. Once the upper half is full, the lower half is filled by using symmetric extension. 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 frame contained in the first position is discarded and a new frame can be introduced in the last position (2N) by using a recursive call. This operation is repeated twice.

However, if there are no more frames in the previous level, this recursive call will return End Of Frame (EOF). 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 by using

symmetric extension again.

Once the buffer is filled or updated, both high-pass and low-pass filter banks are applied to the frames in the buffer. As a result of the convolution, we get a frame of every wavelet subband at this level, and an *LLL* frame. The high-frequency coefficients are compressed and this function returns the LLL frame (see Figure 3).

The inverse DWT algorithm is similar to the forward DWT, but applied in reverse order. The decoding process begins immediately by filling up the highest-level buffer (*nlevel*) with the information received from the bit-stream. During this process, other information from the bit-stream is ignored. Afterwards, once this buffer is full, we also begin to accept information from the previous level, and so forth, until all the buffers are full. At that moment, the video can be sequentially decoded as usual. The latency of this process is deterministic and depends on the filter length and the number of decomposition levels (the higher they are, the higher latency). However, for the regular 3D algorithm, the latency depends on the remaining number of frames in the current group when the process begins, and the GOP size. 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 by using some buffers at both ends, so that data are supplied in the right order [8]. Other simpler solutions are to save every level in secondary storage separately so that it can be read in a different order or to keep the compressed bit-stream in memory if the 3D-DWT is used for compression.

function GetLLLFrame (*level*)
1) First base case: No more frames to read at this level
  if $FramesRead_{level} = MaxFrames_{level}$
    return EOF
2) Second base case: The current level belongs
to the space domain and not to the wavelet domain
  else if $level = 0$
    return InputFrame()
  else
3) Recursive case
3.1) Recursively fill or update the buffer for this level
  if *buffer*$_{level}$ is empty
    for $i = N \ldots 2N$
      buffer$_{level}(i) = 2DFWT(GetLLframe(level-1))$
    FullSymmetricExtension(*buffer*$_{level}$)
  else
    repeat twice
      Shift(*buffer*$_{level}$)
      frame = GetLLLframe($level-1$)
      if *frame* = EOF
        buffer$_{level}(2N)$ = SymmetricExt(*buffer*$_{level}$)
      else
        buffer$_{level}(2N)$ = 2DFWT(*frame*)
3.2) Calculate the WT for the time direction from the frames
in buffer, then process the resulting high frequency subband frames
  $\{LLL, LLH, LHL, LHH\}$ =Z-axis_FWT_LowPass(*buffer*$_{level}$)
  $\{HLL, HLH, HHL, HHH\}$ =Z-axis_FWT_HighPass(*buffer*$_{level}$)
  ProcessSubFrames($\{LLH, LHL, LHH, HLL, HLH, HHL, HHH\}$)
  set $FramesRead_{level}$=$FramesRead_{level}+1$
  return LLL
end of fuction

Fig. 3. GetLLLFrame Recursive function

3. RUN-LENGTH ENCODER

In order to have low memory consumption, once a wavelet subband is calculated, it has to be encoded as soon as possible to release memory. The encoder cannot use global video information since it does not know the whole video. Moreover, we aim at fast execution, and hence no R/D optimization or bitplane processing can be made, because it would turn it even 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. 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 subband buffer are scanned row by row (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 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 coefficient, 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 Figure 4.

function RLW_Code_Subband(*Buffer*, L)
  Scan Buffer in horizontal raster order
  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 $run\_length_L \leq enter\_run\_mode$
      repeat run_length_L **times**
        arithmetic_output *LOWER*
    else
      arithmetic_output *RUN*
      $rbits = \lceil log_2 (run\_length_L) \rceil$
      arithmetic_output $rbits$
      output $bit_{nbits_{(i,j)}-1} (|C_{i,j}|) \ldots bit_{rplane+1} (|C_{i,j}|)$
      output sign($c_{i,j}$)
end of fuction
Note: bit_n (C) is a function that returns the n^{th} bit of C

Fig. 4. Run-length coding of the wavelet coefficients

4. RESULTS

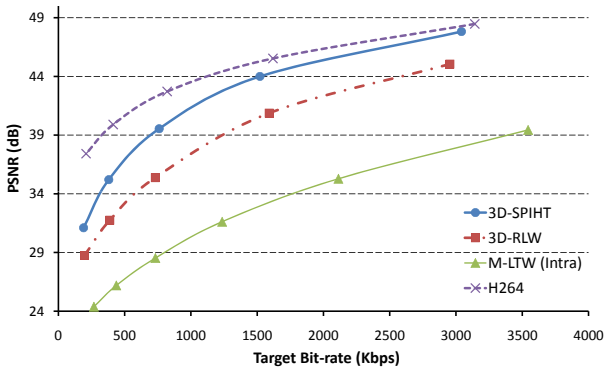
In this section we analyze the behavior of the proposed encoder (3D-RLW). We will compare the 3D-RLW encoder versus the fast M-LTW Intra video encoder[10], 3D-SPIHT [11] and H.264 (JM16.1 version), in terms of R/D performance, coding and decoding delay and memory requirements. All the evaluated encoders have been tested on an Intel PentiumM Dual Core 3.0 GHz with 1 Gbyte RAM memory.

Codec/Format	H.264	3D-SPIHT	3D-RLW	M-LTW
QCIF	35824	10152	3556	**1104**
CIF	86272	34504	11616	**1540**

Table 1. Memory requirements for evaluated encoders (KB) (results obtained with Windows XP task manager, peak memory usage index)

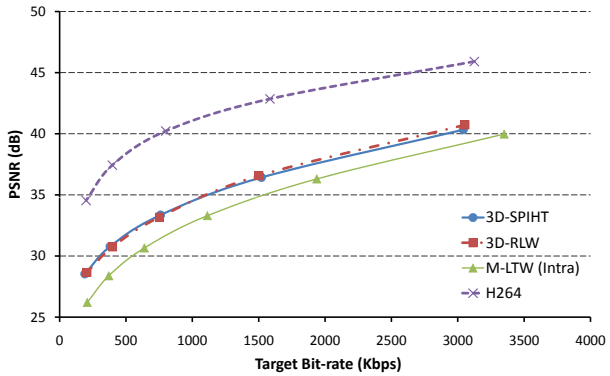
In Table 1, the memory requirements of different encoders under test are shown. Obviously, the M-LTW encoder only uses the memory needed to store one frame. The 3D-RLW encoder (using Daubechies 9/7F time filter) uses 3 times less memory than 3D-SPIHT and up to 10 times less memory than H.264 for QCIF sequence size.

Regarding R/D, in Figures 5 and 6 we can see the R/D behavior of all evaluated encoders. As shown, H.264 is the one that obtains the best results, mainly due to the motion estimation/motion compensation (ME/MC) stage included in this encoder, contrary to 3D-SPIHT and 3D-RLW that do not include any ME/MC stage. It is interesting to see the improvement of 3D-SPIHT and 3D-RLW when compared to an INTRA video encoder. As mentioned, no ME stage is included in 3D-SPIHT and 3D-RLW, so this improvement is accomplished by exploiting only the temporal redundancy among video frames. The



(a) Container

Fig. 5. PSNR (dB) for all evaluated encoders for Container sequence in CIF format



(b) Foreman

Fig. 6. PSNR (dB) for all evaluated encoders for Foreman sequence in CIF format

R/D behavior of 3D-SPIHT and 3D-RLW is similar for images with moderate-high motion activity, but for sequences with low movement, 3D-SPIHT outperforms 3D-RLW, showing the power of tree encoding system.

Regarding coding delay, in Figure 7 we can see that the 3D-RLW encoder is the fastest one, being up to 8 times faster than 3D-SPIHT for QCIF size sequences, 3 times faster than the M-LTW INTRA video encoder and up to 3500 times faster than H.264. The decoding process is also faster in 3D-RLW than in the other encoders.

5. CONCLUSIONS

In this paper a fast and low memory demanding 3D-DWT encoder has been presented. The new encoder reduces the memory requirements compared with 3D-SPIHT (3 times less memory) and H.264 (up to 10 times less memory). The new 3D-DWT encoder is very fast (up to 8 times faster than 3D-SPIHT) and it has better R/D behavior than the INTRA video coder M-LTW. In order to improve the coding efficiency, an ME/MC stage could be added. In this manner, the objects/pixels of the input video sequence will be aligned, and so, fewer frequencies would appear

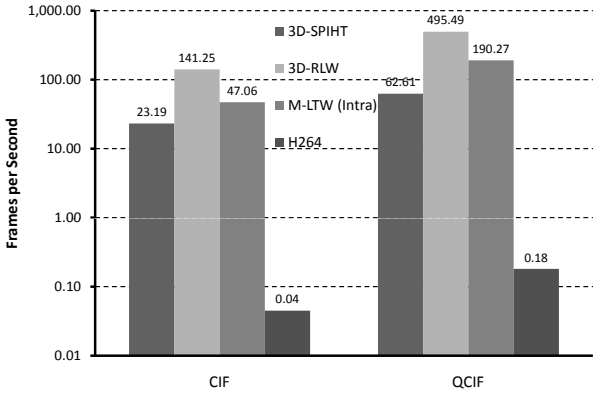


Fig. 7. Execution time comparison of the encoding process

at the higher frequency subbands, improving the compression performance. The low memory requirements and the fast coding/decoding process, makes the 3D-LTW encoder a good candidate for IPTV applications where the coding delay is critical for proper operation.

6. REFERENCES

[1] P. Campisi and A. Neri, "Video watermarking in the 3D-DWT domain using perceptual masking," in *IEEE International Conference on Image Processing*, September 2005, pp. 997–1000.

[2] P. Schelkens, A. Munteanu, J. Barbariend, M. Galca, X. Giro-Nieto, and J. Cornelis, "Wavelet coding of volumetric medical datasets," *IEEE Transactions on Medical Imaging*, vol. 22, no. 3, pp. 441–458, March 2003.

[3] P.L. Dragotti and G. Poggi, "Compression of multispectral images by three-dimensional SPITH algorithm," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 38, no. 1, pp. 416–428, January 2000.

[4] M. Aviles, F. Moran, and N. Garcia, "Progressive lower trees of wavelet coefficients: Efficient spatial and SNR scalable coding of 3D models," *Lecture Notes in Computer Science*, vol. 3767, pp. 61–72, 2005.

[5] B.J. Kim, Z. Xiong, and W.A. Pearlman, "Low bit-rate scalable video coding with 3D set partitioning in hierarchical trees (3D SPIHT)," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 10, pp. 1374–1387, December 2000.

[6] A. Secker and D. Taubman, "Motion-compensated highly scalable video compression using an adaptive 3D wavelet transform based on lifting," *IEEE Internantional Conference on Image Processing*, pp. 1029–1032, October 2001.

[7] P. Cheng and J.W.Woods, "Bidirectional MC-EZBC with lifting implementation," *IEEE Transactions on Circuits and Systems for Video Technology*, pp. 1183–1194, October 2004.

[8] C. Chrysafis and A. Ortega, "Line-based, reduced memory, wavelet image compression," *IEEE Transactions on Image Processing*, vol. 9, no. 3, pp. 378–389, March 2000.

[9] J. Oliver, E. Oliver, and M.P.Malumbres, "On the efficient memory usage in the lifting scheme for the two-dimensional wavelet transform computation," in *IEEE International Conference on Image Processing*, September 2005, pp. 485–488.

[10] O. Lopez, M. Martinez-Rach, P. Piñol, M.P. Malumbres, and J.Oliver, "M-LTW: A fast and efficient intra video codec," *Signal Processing: Image Communication*, , no. 23, pp. 637–648, July 2008.

[11] B.J. Kim, Z. Xiong, and W.A. Pearlman, "Very low bit-rate embedded video coding with 3D set partitioning in hierarchical trees (3D SPIHT)," 1997.