J Supercomput
DOI 10.1007/s11227-012-0750-5

# Parallel strategies for 2D Discrete Wavelet Transform in shared memory systems and GPUs

V. Galiano · O. López · M.P. Malumbres ·
H. Migallón

**Abstract** In this work, we analyze the behavior of several parallel algorithms developed to compute the two-dimensional discrete wavelet transform using both OpenMP over a multicore platform and CUDA over a GPU. The proposed parallel algorithms are based on both regular filter-bank convolution and lifting transform with small implementations changes focused on both the memory requirements reduction and the complexity reduction. We compare our implementations against sequential CPU algorithms and other recently proposed algorithms like the SMDWT algorithm over different CPUs and the *Wippig&Klauer* algorithm over a GTX280 GPU. Finally, we analyze their behavior when algorithms are adapted to each architecture. Significant execution times improvements are achieved on both multicore platforms and GPUs. Depending on the multicore platform used, we achieve speed-ups of 1.9 and 3.4 using two and four processes, respectively, when compared to the sequential CPU algorithm, or we obtain speed-ups of 7.1 and 8.9 using eight and ten processes. Regarding GPUs, the GPU convolution algorithm using the GPU shared memory obtains speed-ups up to 20 when compared to the CPU sequential algorithm.

**Keywords** Wavelet transform · Image coding · Parallel algorithms · CUDA · GPU · OpenMP

## 1 Introduction

During the last years, several image compression schemes emerged in order to overcome the known limitations of block-based algorithms that use the Discrete Cosine Transform (DCT) [1], the most widely used compression technique at that moment.

V. Galiano (✉) · O. López · M.P. Malumbres · H. Migallón
Physics and Computer Architecture Department, Miguel Hernández University, 03202 Elche, Spain
e-mail: vgaliano@umh.es

Published online: 22 March 2012

Some of these alternative proposals were based on more complex techniques, like vector quantization and fractal image coding, while others simply proposed the use of a different and more suitable mathematical transform, the Discrete Wavelet Transform (DWT). Wavelet transforms have proven to be very powerful tools for image compression and many state-of-the-art image codecs, including the JPEG2000 image coding standard, employ a wavelet transform in their algorithms (see, for example, [2, 3]).

Unfortunately, despite the benefits that the wavelet transform entails, some other problems are introduced. Wavelet-based image processing systems are typically implemented by memory-intensive algorithms, with higher execution time than other transforms. In the usual DWT implementation [4], the image decomposition is computed by means of a convolution filtering process and so, its complexity rises as the filter length increases. Moreover, in the regular DWT computation, the image is transformed at every decomposition level first row by row and then column by column, and hence it must be kept entirely in memory.

The lifting scheme [5, 6] is probably the best-known algorithm to compute the wavelet transform in a more efficient way. Since it uses less computations than the equivalent convolution filter, it provides theoretical faster implementation of the DWT. This scheme also provides memory reduction through in-place computing of wavelet coefficients.

Other wavelet transform algorithms has been proposed in order to reduce memory requirements such as line based [7] and block-based [8] wavelet transform approaches that perform wavelet transformation at image line and block level, respectively. These approaches increase flexibility when applying wavelet transform and significantly reduce the memory requirements. In this work, we dismiss both line based and block-based algorithms because these algorithms are focused on the memory requirements reduction but not on the complexity reduction. Moving on to another proposal, in [9] authors present a novel way of computing the wavelet transform called Symmetric Mask-based Discrete Wavelet Transform (SMDWT) where wavelet transform is computed as a matrix convolution, using a matrix masks for each wavelet subband type.

In this paper, we develop optimized parallel algorithms based on the methods introduced in [4] and [5], and we analyze their performance when implemented over both multicore and GPU architectures. The main goals of the performed optimizations are to obtain low memory requirements, due to the nearly in-place computation of the DWT, as well as good computational times, exploiting multicore architectures, i.e., shared memory platforms. After that, we will adapt the scheme introduced in the multicore algorithms in order to develop CUDA-based DWT algorithms. Algorithms developed on Graphics Processing Units (GPU) require an efficient use of memory to exploit their architecture in an efficient way. The developed algorithms are focused in the use of the different memories of the GPU. We have also compared our CUDA proposals against the algorithm proposed in [10], based in the use of textures and developed using OpenGL [11], in both computation performance and memory requirements.

## 2 Discrete Wavelet Transform

The DWT is a multiresolution decomposition scheme for input digital signals, see detailed description in [4]. The source signal is firstly decomposed into two frequency subbands, low-frequency (low-pass) subband, and high-frequency (high-pass) subband. For the classical DWT, the forward decomposition of a signal is implemented by a low-pass digital filter $H$ and a high-pass digital filter $G$. Both digital filters are derived using the scaling function $\Phi(t)$ and the corresponding wavelet functions at different frequency scales $\Psi(t)$. The system downsamples the signal to half of the filtered results in the decomposition process. If four-tap and non-recursive FIR filters are considered, the transfer functions of $H$ and $G$ can be represented as follows:

$$H(z) = h_0 + h_1 z^{-1} + h_2 z^{-2} + h_3 z^{-3} \tag{1}$$

$$G(z) = g_0 + g_1 z^{-1} + g_2 z^{-2} + g_3 z^{-3} \tag{2}$$

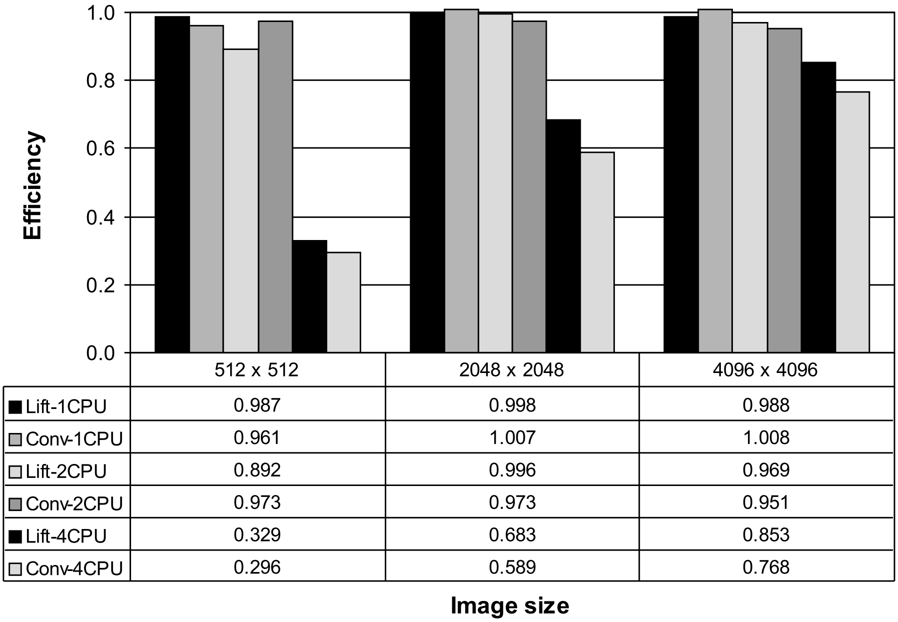
One of the drawbacks of the DWT is that it doubles the memory requirements because it is implemented as a filter. The lifting scheme [5] reduces the memory requirements and the number of operations needed to perform the wavelet transform if compared with the usual filtering algorithm (also known as convolution algorithm). The order of this reduction depends on the type of wavelet transform, as shown in [12].

A special case of wavelet filter is the Daubechies 9/7 filter. This filter has been widely used in image compression [3, 13], and it has been included in the JPEG2000 standard [2]. The algorithms presented in this paper are focused on this filter.

Recently, in [9], authors presented a novel way of computing the wavelet transform trying to reduce the computational complexity for the wavelet filtering process, which the authors have called Symmetric Mask-based Discrete Wavelet Transform (SMDWT). This algorithm computes the wavelet transform as a matrix convolution, using the four matrix derived from the 2D-DWT of Daubechies 9/7 floating point lifting-based coefficients. The 2D lifting-based Wavelet Transform (LDWT) scheme requires vertical and horizontal 1D LDWT calculations, and each of the 1D LDWT requires four steps: splitting, prediction, updating, and scaling. Conversely, the four subband 2D SMDWT can be yielded using four independent matrices of size 7 × 7, 7 × 9, 9 × 7, and 9 × 9 for the Daubechies 9/7 filter. The interest in this algorithm is based on both the way it computes the DWT which unlike the traditional DWT and LDWT algorithms, it computes each subband independently through a four matrix convolution, and the theoretical low computation complexity.

## 3 Implementation of the wavelet transform on a multicore CPU

In order to develop the optimized parallel 2D-DWT, we have used the regular filterbank convolution, based on Daubechies 9/7 filter, proposed in [4]. On the other hand, we have used the lifting scheme proposed by Sweldens in [5] to develop the optimized parallel 2D LDWT. Typical convolution-based wavelet transform implementations

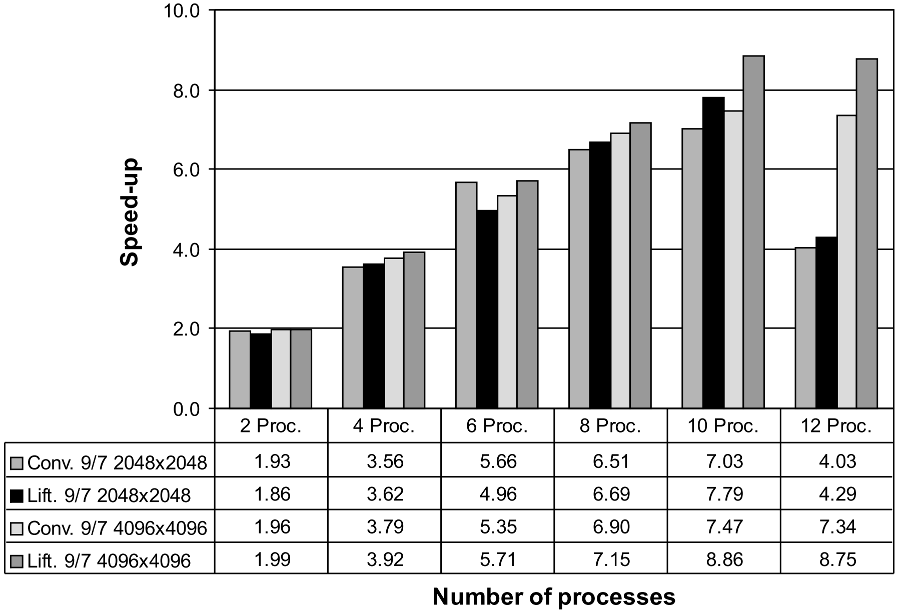


| | 512 x 512 | 2048 x 2048 | 4096 x 4096 |
|---|---|---|---|
| ■ Lift-1CPU | 0.987 | 0.998 | 0.988 |
| ▨ Conv-1CPU | 0.961 | 1.007 | 1.008 |
| □ Lift-2CPU | 0.892 | 0.996 | 0.969 |
| ▨ Conv-2CPU | 0.973 | 0.973 | 0.951 |
| ■ Lift-4CPU | 0.329 | 0.683 | 0.853 |
| □ Conv-4CPU | 0.296 | 0.589 | 0.768 |

**Fig. 1** Efficiency for multicore wavelet transform algorithms. Multicore Q6600

require twice the image size to store the resulting coefficients. In our convolution-based implementation, an extra memory space to store the current image row/column is required, which allows us a nearly in-place computation, also due to the way we perform the decimation process. We compute it just in the filtering process, applying the low pass filter to the even pixels and the high pass filter to the odd ones. In the lifting-based wavelet transform, we need the memory space to store a copy of both one row and one column. Note that, the SMDWT algorithm requires twice the image size space to perform the four mask filtering process, and to store the resulting coefficients.

We have used OpenMP [14] paradigm in order to develop the parallel algorithms. The multicore platforms used are: (1) an Intel Core 2 Quad Q6600 2.4 GHz, with four cores and (2) a HP Proliant SL390 G7 with two Intel Xeon X5660, each CPU with six cores at 2.8 GHz. In this algorithm, a block of rows and a block of columns has been assigned to one thread in each core to compute the wavelet transform. Therefore, each thread would require the aforementioned amount of extra memory. Remark that the objective of this buffer is to compute the wavelet transform, so we could store the final wavelet coefficients in the same memory space occupied by the image, avoiding doubling the memory requirements. The amount of extra memory used by each algorithm depends on the number of cores used. We have worked in our tests with different grayscale image sizes: 512 × 512, 2048 × 2048, and 4096 × 4096 pixels, using floating point precision for each pixel. The worst case is for the smallest image, requiring less than 2 % of extra memory overhead, being for the rest of the images less than 1 %. As mentioned, the extra memory size needed by the SMDWT algorithm is the size of the image. The operating system running on both multicore platforms is



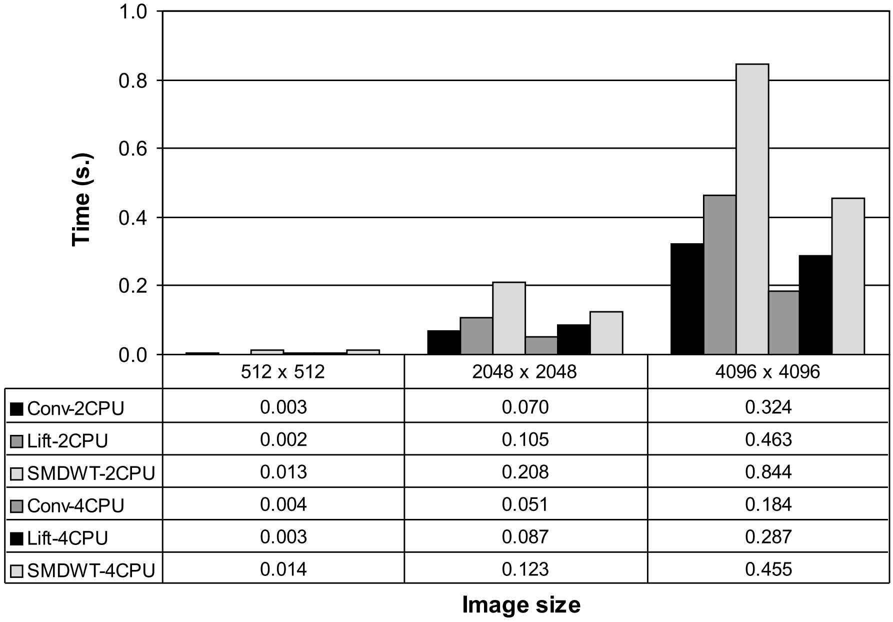
| | 2 Proc. | 4 Proc. | 6 Proc. | 8 Proc. | 10 Proc. | 12 Proc. |
|---|---|---|---|---|---|---|
| ▨ Conv. 9/7 2048x2048 | 1.93 | 3.56 | 5.66 | 6.51 | 7.03 | 4.03 |
| ■ Lift. 9/7 2048x2048 | 1.86 | 3.62 | 4.96 | 6.69 | 7.79 | 4.29 |
| □ Conv. 9/7 4096x4096 | 1.96 | 3.79 | 5.35 | 6.90 | 7.47 | 7.34 |
| ▨ Lift. 9/7 4096x4096 | 1.99 | 3.92 | 5.71 | 7.15 | 8.86 | 8.75 |

**Fig. 2** Speed-up for multicore wavelet transform algorithms. Multicore HP Proliant SL390 G7

a Linux-based one for 64 bit systems. The compiler flags used in GNU compiler to exploit the multicore architectures are: “-O3 -m64 -fopenmp,” while the ones used to avoid multicore architecture are: “-O3 -m64.”

We have tuned the algorithms to obtain the best performance on multicore architectures, taking into account that these algorithms are characterized by an intensive use of memory. We have tested several OpenMP-based strategies in order to develop the multicore algorithm such us varying the schedule option, the use of parallel sections, and to assign the block size depending on the thread rank. The best results are obtained by assigning the block size depending on the thread rank, achieving a slight improvement. In Fig. 1, we show the efficiency obtained in the multicore Q6600 platform, for both convolution-based and lifting-based 2D-DWT using 1, 2, and 4 cores, for three different image sizes and six wavelet decomposition levels. The reference computational time to compute the efficiency showed in Fig. 1 is the sequential computational time of each parallel algorithm. As it can be seen, we obtain a good behavior for high resolution images using 4 cores and close to the ideal using 2 cores. However, for small images (low resolution images), the behavior is not good due to the relationship between the computational load and the memory accesses. Note that by increasing the number of threads, the access to memory becomes a bottleneck because the number of entities that use the memory is higher.

Figure 2 presents the speed-up obtained using the multicore HP Proliant SL390 G7 and up to 12 cores. Note that, in this case, the architecture provides a high-bandwidth memory access, through the *Intel QPI Speed* 64 $GT/s$. We are able to avoid, using that architecture and for a more number of processes, the bottleneck observed in the Q6600 multicore platform. The multicore HP Proliant SL390 G7 offers greater



| | 512 x 512 | 2048 x 2048 | 4096 x 4096 |
|---|---|---|---|
| ■ Conv-2CPU | 0.003 | 0.070 | 0.324 |
| ▨ Lift-2CPU | 0.002 | 0.105 | 0.463 |
| □ SMDWT-2CPU | 0.013 | 0.208 | 0.844 |
| ▨ Conv-4CPU | 0.004 | 0.051 | 0.184 |
| ■ Lift-4CPU | 0.003 | 0.087 | 0.287 |
| □ SMDWT-4CPU | 0.014 | 0.123 | 0.455 |

**Fig. 3** Computational times for multicore wavelet transform algorithms. Multicore Q6600

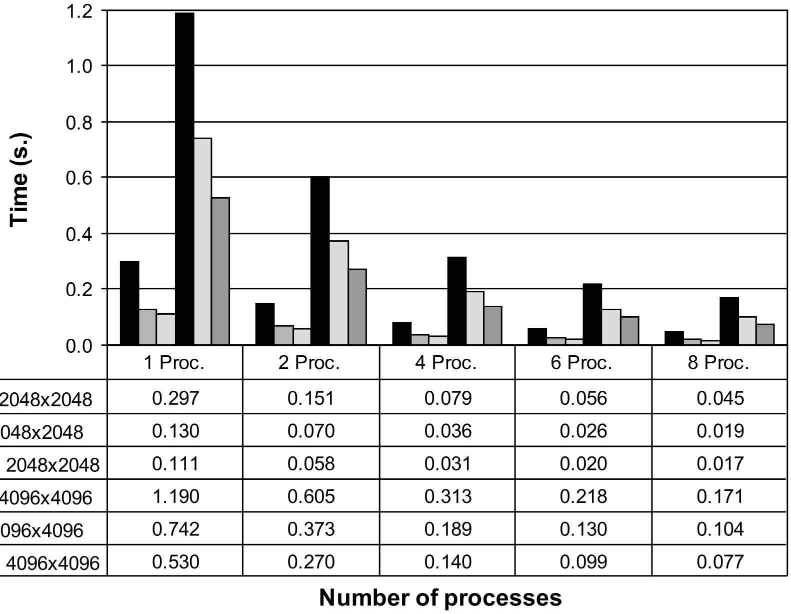
computing power, but it especially offers a significant higher bandwidth memory access. As Fig. 2 shows, we obtain good values of speed-up even using 8 and 10 cores, depending on the image resolution.

Moreover, we have compared our algorithms against the SMDWT proposal introduced in Sect. 2, on both multicore platforms. We have developed a parallel algorithm of this reference algorithm. In Fig. 3, we present a comparison, using the Q6600 multicore platform, between our convolution-based and lifting-based algorithms, and the SMDWT algorithm, using 2 and 4 cores. As it can be seen, our convolution implementation has better times than the lifting one, and both algorithms are 2.5 times as fast as the SMDWT algorithm.

Figure 4 presents the comparison showed in Fig. 3 using the HP Proliant SL390 GT. We can extend the conclusions obtained, up to the maximum number of available cores, with the multicore Q6600, however, if we increase the number of cores, the efficiency of the SMDWT algorithm decreases faster than using our algorithms.

## 4 Implementation of the wavelet transform on a GPU

In Sect. 3, we have compared the execution times and the speed-up obtained by our algorithms against several relevant and recent algorithms. Our shared memory parallel convolution-based algorithm presents the best performance according to the results shown in Sect. 3. We question in this section if a better performance of this algorithm can be achieved using a Graphical Processor Unit (GPU). The GPU architecture is



| | 1 Proc. | 2 Proc. | 4 Proc. | 6 Proc. | 8 Proc. |
|---|---|---|---|---|---|
| ■ SMDWT 2048x2048 | 0.297 | 0.151 | 0.079 | 0.056 | 0.045 |
| ▨ Lift. 9/7 2048x2048 | 0.130 | 0.070 | 0.036 | 0.026 | 0.019 |
| □ Conv. 9/7 2048x2048 | 0.111 | 0.058 | 0.031 | 0.020 | 0.017 |
| ■ SMDWT 4096x4096 | 1.190 | 0.605 | 0.313 | 0.218 | 0.171 |
| □ Lift. 9/7 4096x4096 | 0.742 | 0.373 | 0.189 | 0.130 | 0.104 |
| ▨ Conv. 9/7 4096x4096 | 0.530 | 0.270 | 0.140 | 0.099 | 0.077 |

**Fig. 4** Computational times for multicore wavelet transform algorithms. Multicore HP Proliant SL390 G7

based on a set of multiprocessor units called streaming multiprocessors (SM), containing each one a set of processor cores called streaming processors (SP). CUDA is a heterogeneous computing model that involves both the CPU and the GPU. In the CUDA parallel programming model [15, 16], an application consists of a sequential host program that may execute parallel programs known as kernels on a parallel device, i.e., a GPU. A kernel is an Single Program Multiple Data (SPMD) computation that is executed using a large number of parallel threads organized into a grid of blocks. The threads of each block can cooperate among themselves using a barrier synchronization. Threads may access data from multiple memory spaces. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory. There are also two additional read only memory spaces accessible by all threads: the constant and texture memory spaces. Texture memory also offers different addressing modes.

So, in order to develop our algorithms presented in Sect. 3 to run on GPU architecture, we must consider the GPU model we are going to use. We use the NVIDIA GTX 280 GPU that contains 30 multiprocessors with 8 cores in each multiprocessor, 1 GB of global memory, and 16 KB of shared memory by block (or SM).

Firstly, we will define our GPU-based algorithm, named as *CUDA-Conv 9/7*, as the reference algorithm. It will only use shared memory space to store the buffer that will contain a copy of the working row/column data. The constant memory space is used to store the filter taps $h[n]$ and $g[n]$.

We call each CUDA kernel with a one-dimensional number of thread blocks, NBLOCKS, and a one-dimensional number of threads by block, NTHREADS. In

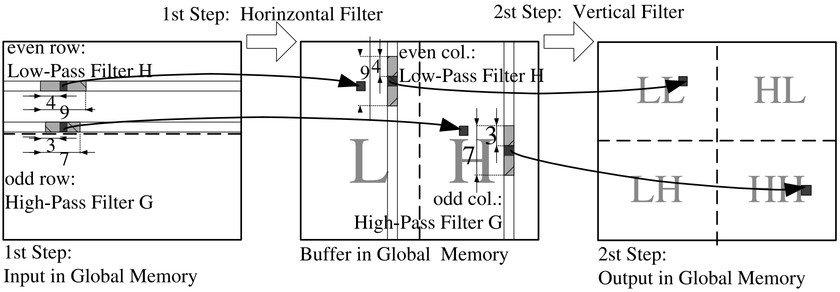


**Fig. 5** *CUDA-Mem 9/7* method (global memory)

the horizontal DWT filtering process, each image row is stored in the shared memory by the threads. After that, in the vertical filtering, each column is processed in the same way. The row or column size determines the NBLOCKS parameter, which must be greater or equal to the image width in the horizontal step or the image height in the vertical step. One of the main goals in the proposed CUDA-based methods is to reduce memory requirements, so we will store the resulting wavelet coefficients in the original image memory space.

As there are different memory spaces available in GPUs, it would be of interest to compute the DWT using different memory locations to determine their impact in the algorithm runtime performance. In this sense, we have also implemented the DWT convolution using different ways of accessing to the three memory spaces from the threads: global memory, texture memory, and shared memory.

In the *CUDA-Mem 9/7* called method, the original image is stored in the GPU global memory and the buffer required to perform the filtering process of columns is stored in the GPU global memory too. In Fig. 5, we illustrate the algorithm implemented in two steps. First, each thread computes only one pixel applying the horizontal convolution and saving the output in the buffer memory. The low pass filter and the high pass filter are applied to the even and odd rows, respectively. The results are stored in the low and high frequency area of the image. In the second step, the vertical convolution is computed and the results are stored in the original memory space; the results are also split in low and high frequencies by columns.

In the second method called *CUDA-Text 9/7*, whose behavior is shown in Fig. 6, the original image is stored in the GPU texture memory space while the buffer required to perform the filtering process of rows/columns is stored in the GPU global memory. We must remark that texture memory is a read only memory, so an intermediate step is needed. After the horizontal convolution, the output data must be copied to the texture memory space. This copy must be done transferring data from GPU to CPU and after that, from CPU to GPU.

Finally, the third method called *CUDA-Sep 9/7* stores the original image in the GPU global memory but computes the filtering steps from the shared memory. As it can be seen in Fig. 7, a block of the image is loaded into a shared memory array with BLOCKSIZE pixels. The number of thread blocks, NBLOCKS, or tiles depends on BLOCKSIZE and image dimensions. Note that, around the loaded image block there is an apron of neighbor pixels of the width of the filter radius (where filter radius is

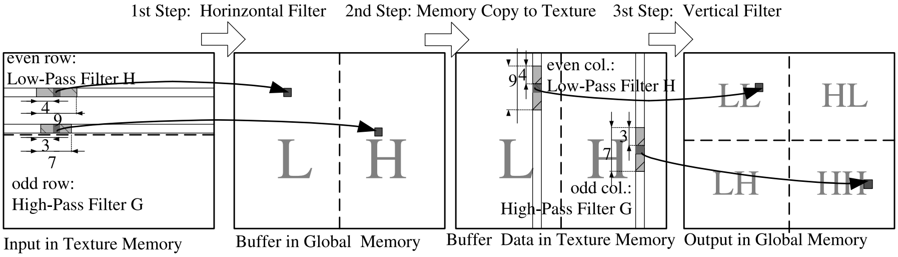
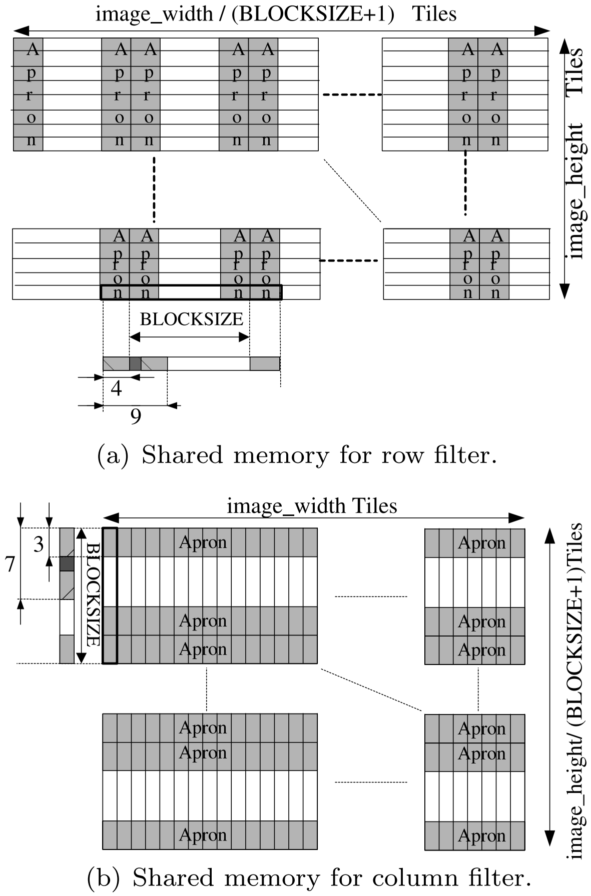


**Fig. 6** *CUDA-Text 9/7* method (texture memory)

**Fig. 7** Shared memory for Daubechies 9/7 filter



(a) Shared memory for row filter.

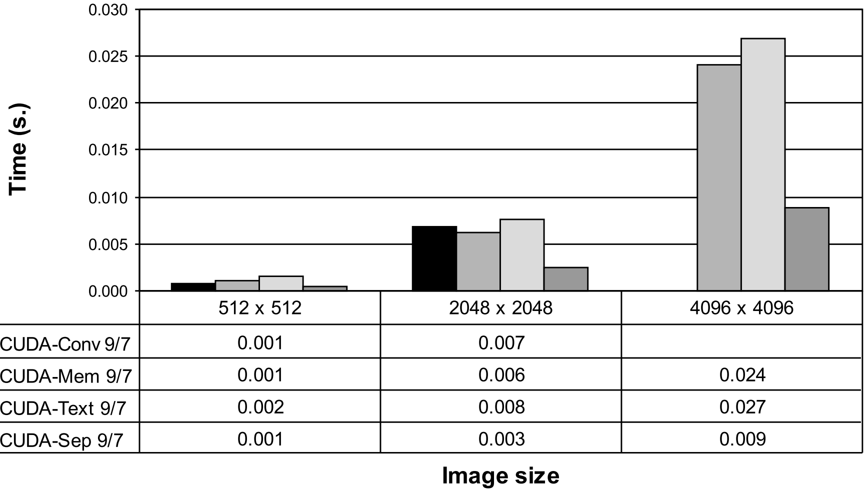
(b) Shared memory for column filter.

the half of the filter length minus 1) that is required in order to properly filter the image block. These regions are showed in Figs. 7(a) and 7(b) as shaded pixels. In both subfigures, the values of the filter radius and the filter length corresponding to the Daubechies 9/7 filter are represented. We can reduce the number of idle threads by reducing the total number of threads per block and also using each thread to load multiple pixels into shared memory. This ensures that all threads are active during the computation stage. Note that the number of threads in a block must be a multiple of the warp size (32 threads on GTX 280) for optimal efficiency.

To achieve higher efficiency and higher memory throughput, the GPU attempts to coalesce accesses from multiple threads into a single memory transaction. If all threads within a warp (32 threads) simultaneously read consecutive words then single large read of the 32 values can be performed at optimum speed. In the *CUDA-Sep 9/7* algorithm, the convolution process is separated in two stages: (1) the row filtering stage and (2) the column filtering stage. Each row/column filtering stage is separated into two sub-stages: (a) the threads load a block of pixels of one row/column from the global memory into the shared memory, and (b) each thread computes the filter over the data stored in the shared memory and stores the results in the global memory. We must not forget about the cases when a row or column processing tile becomes clamped by image borders, and initialize clamped shared memory array indices with correct values. In this case, threads also must load in shared memory the values of adjacent pixels in order to compute the pixels located in borders (named as *Apron* area in Fig. 7).

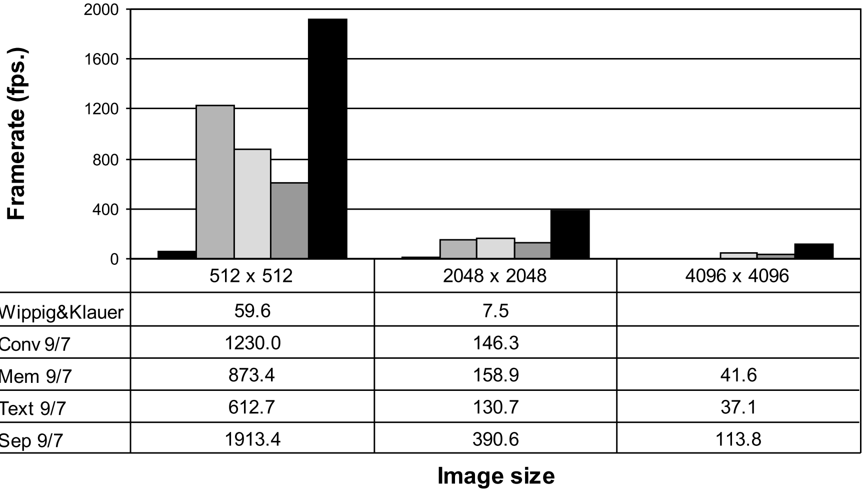
In Fig. 8, we compare computational times to obtain the 2D-DWT using the four proposed CUDA-based algorithms. We want to remark that in all experiments reported we have used floating point precision. We can observe that the results obtained by the *CUDA-Conv 9/7* algorithm are similar to the results obtained by *CUDA-Mem 9/7* and *CUDA-Text 9/7* algorithms, but the *CUDA-Conv 9/7* algorithm has the lowest memory requirements because the image is overwritten with the wavelet coefficients while the *CUDA-Mem 9/7* and *CUDA-Text 9/7* algorithms need to double memory requirements. On the other hand, the best performance is achieved by the *CUDA-Sep 9/7* algorithm because in this algorithm we optimize the memory access and we achieve optimal memory throughput in the shared memory. As we can observe, efficient use of shared memory access let us to achieve the best performance. Data for the 4096 × 4096 image size is not presented because the shared memory is not large enough to store a row/column of the image including the symmetric extension. The available shared memory in the GTX280 GPU is 16384 B, the needed shared memory for a 4096 × 4096 image size is 4104 pixels, that is, 16416 B. Note that we use the symmetric extension in order to avoid the border effects, therefore we need to extend the maximum size of filter divided by 2 at both ends, being for the particular case of the Daubechies 9/7 filter 4 pixels at both ends. In the 4096 × 4096 case, it would be necessary to split the row/column, which in fact is the same scheme as the *CUDA-Mem 9/7* algorithm.

In order to compare our CUDA DWT implementations with other proposals, in Fig. 9 we show the frame rate obtained by our algorithms and by the algorithm proposed in [10], illustrated as *Wippig&Klauer*. The *Wippig&Klauer* algorithm is based on the definition of fragment shader and the use of a texture of twice the image size for the temporary and final results. Authors aim to achieve better performance using low level graphics programming through the OpenGL API, being in this case most of graphics hardware supported. The *Wippig&Klauer* algorithm is a direct implementation of the filter banks by a block (fragment shader). Furthermore, the border effects are prevented by indirect accessing to image positions stored in the texture memory and using separate kernels for that purpose. All these algorithms have been



| | 512 x 512 | 2048 x 2048 | 4096 x 4096 |
|---|---|---|---|
| ■ CUDA-Conv 9/7 | 0.001 | 0.007 | |
| ▩ CUDA-Mem 9/7 | 0.001 | 0.006 | 0.024 |
| □ CUDA-Text 9/7 | 0.002 | 0.008 | 0.027 |
| ▩ CUDA-Sep 9/7 | 0.001 | 0.003 | 0.009 |

**Fig. 8** Computational times for proposed algorithms on the GTX280



| | 512 x 512 | 2048 x 2048 | 4096 x 4096 |
|---|---|---|---|
| ■ Wippig&Klauer | 59.6 | 7.5 | |
| ▩ Conv 9/7 | 1230.0 | 146.3 | |
| □ Mem 9/7 | 873.4 | 158.9 | 41.6 |
| ▩ Text 9/7 | 612.7 | 130.7 | 37.1 |
| ■ Sep 9/7 | 1913.4 | 390.6 | 113.8 |

**Fig. 9** Framerate for proposed algorithms on the GTX280

tested in the same GPU (GTX 280) and with six wavelet decomposition levels. For high resolution images, no data are described in the cited paper [10]. However, for a 2048 × 2048 image size all our algorithms achieve better performance than the *Wippig&Klauer* proposal, in particular the *CUDA-Sep 9/7* algorithm which obtains a framerate 52 times as fast as the *Wippig&Klauer* algorithm. Remark that the *Wippig&Klauer* algorithm performs twice the operations than our algorithms, because of the translation-invariant DWT used in that algorithm.

## 5 Conclusions

We have presented both multicore-based (convolution and lifting) and CUDA-based algorithms (convolution) that perform the two-dimensional discrete wavelet transform. We have analyzed the behavior of the proposed algorithms over a shared-memory multiprocessor and a GPU architecture. Furthermore, we have compared our multicore-based proposals against a recent algorithm called SMDWT. The multicore-based algorithms obtain a speed-up above 1.9 when using two processors and above 2.4 and up to 3.4 when using four processors, running on a relatively low computing power platform as the Q6600 multicore platform, when compared to the sequential CPU algorithm. But when running on the HP Proliant SL390 G7, we obtain good speed-ups even using the maximum number of available cores, depending on the image resolution. Since the best results over a multicore platform have been obtained by the convolution algorithm which also requires the smaller buffer size, we have developed the corresponding GPU-based algorithm using CUDA and implemented the row/column buffer in the GPU shared memory. The speed-up achieved by the GPU-based algorithm is up to 20 relative to the sequential implementation in one core (data transfer time is not included). Note that wavelet transform is only a single first step in an image/video codec and the wavelet coefficients obtained must be processed according to the final application. Also, we have developed several CUDA-based algorithms using the different available kind of memories in a GPU. Between them, the *CUDA-Sep 9/7* algorithm using the GPU shared memory is the fastest one, taking advantage of the separable properties of the Daubechies 9/7 filter to optimize the memory coalescence. In conclusion, we would like to point out that (1) the use of a multicore platform obtains good performance, and (2) we obtain a high speed-up in a GPU compared to the results obtained in the multicore platform. The CUDA-based algorithm to chose depends on the parameter to be optimized, which can be either the computational time or the memory requirements.

**Acknowledgements** This research was partially supported by the Spanish Ministry of Science and Innovation under grant numbers TIN2011-27543-C03-03 and TIN2011-26254.

## References

1. Rao K, Yip P (1990) Discrete cosine transform: algorithms, advantages, applications. Academic Press, Boston
2. ISO (2000) ISO/IEC 15444-1. JPEG2000 image coding system
3. Said A, Pearlman A (1996) A new, fast and efficient image codec based on set partitioning in hierarchical trees. IEEE Transactions on Circuits, Systems and Video Technology 6(3):243–250
4. Mallat SG (1989) A theory for multi-resolution signal decomposition: The wavelet representation. IEEE Trans Pattern Anal Mach Intell 11(7):674–693
5. Sweldens W (1996) The lifting scheme: a custom-design construction of biorthogonal wavelets. Appl Comput Harmon Anal 3(2):186–200
6. Sweldens W (1998) The lifting scheme: a construction of second generation wavelets. SIAM J Math Anal 29(2):511–546
7. Chrysafis C, Ortega A (2000) Line-based, reduced memory, wavelet image compression. IEEE Trans Image Process 9(3):378–389
8. Bao Y, Jay Kuo CC (2001) Design of wavelet-based image codec in memory-constrained environment. IEEE Trans Circuits Syst Video Technol 11(5):642–650

9. Hsia C-H, Guo J-M, Chiang J-S, Lin C-H (2009) A novel fast algorithm based on SMDWT for visual processing applications. In: IEEE international symposium on circuits and systems, ISCAS 2009, pp 762–765
10. Wippig D, Klauer B (2011) GPU-based translation-invariant 2d discrete wavelet transform for image processing. Int J Comput 5(2):226–234
11. Rost RJ (2006) OpenGL© shading language, 2nd edn. Addison-Wesley, Reading
12. Daubechies I, Sweldens W (1998) Factoring wavelet transforms into lifting steps. J Fourier Anal Appl 4(3):247–269
13. Shapiro JM (1993) Embedded image coding using zerotrees of wavelet coefficients. IEEE Trans Signal Process 41(12):3445–3462
14. OpenMP Architecture Review Board (2002) OpenMP C and C++ application program interface, version 2.0
15. Nickolls J, Buck I, Garland M, Skadron K (2008) Scalable parallel programming with CUDA. Queue 6:40–53
16. Corporation NVIDIA (2010) NVIDIA CUDA C programming guide, version 3.2