Proceedings of the 11th International Conference on Computational and Mathematical Methods in Science and Engineering, CMMSE 2011 26–30 June 2011.

Improving the discrete wavelet transform computation from multicore to GPU-based algorithms

V. Galiano[1], O. López[1], M.P. Malumbres[1] and H. Migallón[1]

[1] *Physics and Computer Architecture Department, Miguel Hernández University, 03202 Elche, Spain*

emails: vgaliano@umh.es, otoniel@umh.es, mels@umh.es, hmigallon@umh.es

Abstract

In this work we analyze the behavior of some parallel algorithms when computing the two dimensional discrete wavelet transform (2D-DWT) using both OpenMP over a multicore platform and CUDA (Compute Unified Device Architecture) over a GPU (Graphics Processing Unit). The proposed algorithms are based on both regular filter-bank convolution and lifting transform. Finally we will also compare our algorithms against other recently proposed algorithms.

Key words: CUDA, OpenMP, wavelet transform, image coding, parallel algorithms

Introduction

During the last decade, 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. 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 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. These problems are not as noticeable in other image transforms as in the DWT. For example, when the DCT is used for image compression, it is applied in small block sizes, and thus a large amount of memory is not specifically needed for the transformation process.

The lifting scheme [5, 6] is probably the best-known algorithm to calculate the wavelet transform in a more efficient way. Since it uses less filter coefficients than the equivalent convolution filter, it provides a faster implementation of the DWT. This scheme also provides memory reduction through in-place computation of wavelet coefficients. However, if in-place computation is applied, the low-frequency coefficients are interleaved with the wavelet coefficients, and the subsequent wavelet processing can be non-optimal (specially in cache-based systems), requiring a more careful process. We can overcome this problem with coefficient reordering, at the cost of increasing the complexity of the algorithm.

Other fast wavelet transform algorithms has been proposed in order to reduce both memory requirements and complexity, like line-based [7] and block-based [8] wavelet transform approaches that performs wavelet transformation at image line or block level. These approaches increases flexibility when applying wavelet transform and significantly reduce the memory requirements. At the other hand, in [9], authors present a novel way of computing the wavelet transform called Symmetric Mask-based Discrete Wavelet Transform (SMDWT). This new wavelet transform is computed as a matrix convolution, using four matrix masks, one for each subband type, that are built in order to reduce the repetitive computations found in the classical convolution approach. In this scheme, the 2D-DWT is performed in only one pass, avoiding multiple-layer transpose decomposition operations. One of the most interesting advantages of this method is that the computation of each wavelet subband is completely independent.

When designing fast wavelet-based image/video encoders, one of the most computational intensive tasks is the 2D-DWT, which in some cases may take up between 30% and 50% of the overall encoding time (depending of image size and the number of decompositions levels). So, it is very important to reduce 2D-DWT computation time to develope fast real-time image/video encoders. To do that, we will take profit of the available hardware resources that are present in current off-the-shelf computers, in particular multicore processing and GPU co-processing units.

In this paper, we perform optimized parallel algorithms based on the methods introduced in [4] and [5]. The main goals of the performed optimizations are to obtain low memory requirements as well as good computational behavior, exploiting multicore architectures, i.e. shared memory platform. After that, we will apply the same scheme introduced in the multicore algorithm to develop CUDA-based DWT algorithms on GPU. Algorithms developed on Graphics Processing Units (GPU) require an efficient use of memory to exploit the GPU architecture in an efficient way. The developed algorithms are focused in the use of the GPU shared memory. We have also compared the CUDA based algorithms developed with the algorithms proposed in [10], in both computation performance and memory requirements.

1 Discrete Wavelet Transform (DWT)

DWT is a multiresolution decomposition scheme for input signals, see detailed description in [4]. The original signals are firstly decomposed into two frecuency 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 of digital filters are derived using the scaling function $\Phi(t)$ and the corresponding wavelets $\Psi(t)$. The system downsamples the signal to half of the filtered results in decomposition process. If the four-tap and non-recursive FIR filters with length L are considered, the transfer functions of H and G can be represented as follows:

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

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

1.1 Lifting-based Wavelet Transform (LDWT)

One of the drawbacks of the DWT is that it doubles the memory requirements because it is implemented as a filter. A proposal that reduces the amount of memory needed for the computation of the 1D DWT is the lifting scheme [5]. Despite this disadvantage, the main benefit of this scheme is the reduction in 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 [11].

The lifting scheme implements the DWT decomposition as an alternative algorithm to the classical filtering algorithm introduced in the previous section. In the filtering algorithm, in-place processing is not possible because each input sample is required as incoming data for the computation of its neighbor coefficients. Therefore, an extra array is needed to store the resulting coefficients, doubling the memory requirements. On the other hand, the lifting scheme provides in-place computation of the wavelet coefficients and hence, it does not need extra memory to store the resulting coefficients. In addition, the lifting scheme can be computed on an odd set of samples, while the regular transform requires the number of input samples to be even.

The Euclidean algorithm can be used to factorize the poly-phase matrix of a DWT filter into a sequence of alternating upper and lower triangular matrices. In 3, the variables $h(z)$ and $g(z)$ denote the low-pass and high-pass inverse filters, respectively, which can be divided into even and odd parts to form a poly-phase matrix $P(z)$ as in 4.

$$g(z) = g_e\left(z^2\right) + z^{-1} g_o\left(z^2\right), \quad h(z) = h_e\left(z^2\right) + z^{-1} g_o\left(z^2\right) \tag{3}$$

$$P(z) = \begin{pmatrix} h_e(z) & g_e(z) \\ h_o(z) & g_o(z) \end{pmatrix} \tag{4}$$

Using the Euclidean algorithm, it recursively finds the greatest common divisors of the even and odd parts of the original filters. Since $h(z)$ and $g(z)$ form a complementary

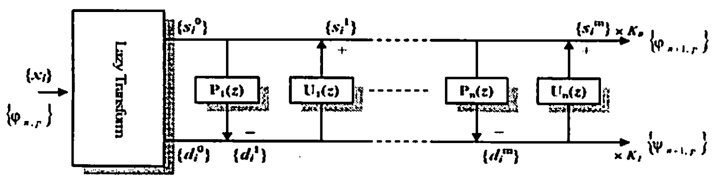


Figure 1: General diagram for a wavelet decomposition using the lifting scheme.

filter pair, $P(z)$ can be factorized into three lifting steps as below.

$$P(z) = \prod_{i=1}^{m} \begin{pmatrix} 1 & s_i(z) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ t_i(z) & 1 \end{pmatrix} \begin{pmatrix} k & 0 \\ 0 & 1/k \end{pmatrix} \tag{5}$$

where $s_i(z)$ and $t_i(z)$ denote the Laurent polynomials corresponding to prediction and update steps, respectively, and k is a nonzero constant.

The whole process consists of a first lazy transform, one or several prediction and update steps, and coefficient normalization. In the lazy transform, the input samples are split into two data sets, one with the even samples and the other one with the odd ones. Thus, if we consider $\{X_i\} = \{\Phi_{n,p}\}$ the input samples at a level n, we define:

$$\{s_i^0\} = \{X_{2i}\} \tag{6}$$
$$\{d_i^0\} = \{X_{2i+1}\} \tag{7}$$

Then, in a prediction step (sometimes called dual lifting), each sample in $\{d_i^0\}$ is replaced by the error committed in the prediction of that sample from the samples in $\{s_i^0\}$:

$$d_i^1 = d_i^0 - P\left(\{s_i^0\}\right) \tag{8}$$

while in an update step (also known as primal lifting), each sample in the set $\{s_i^0\}$ is updated by $\{d_i^1\}$ as:

$$s_i^1 = s_i^0 + U\left(\{d_i^1\}\right) \tag{9}$$

After m successive prediction and update steps, the final scaling and wavelet coefficients are achieved as follows:

$$\{\Phi_{n+1,p}\} = K_0 \times \{s_i^m\} \tag{10}$$
$$\{\Psi_{n+1,p}\} = K_1 \times \{d_i^m\} \tag{11}$$

A special case of wavelet filter is the Daubechies 9/7 filter. This filter has been widely used in image compression [3, 12], and it has been included in the JPEG2000 standard [2]. In this paper, all the DWT algorithms will be focused on this filter because

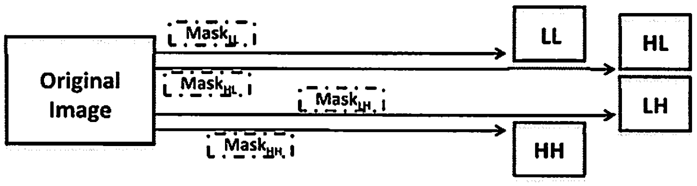


Figure 2: 2D SMDWT structure.

of its goodness behavior. The coefficients of the Daubechies 9/7 decomposition filters, $h[n]$ and $g[n]$ are:

$$\begin{aligned} h\,[n] &= 0.026749, -0.016864, -0.078223, 0.266864, 0.602949, \\ &\quad 0.266864, -0.078223, -0.016864, 0.026749 \\ g\,[n] &= 0.091272, -0.057544, -0.591272, 1.115087, \\ &\quad -0.591272, -0.057544, 0.091272 \end{aligned}$$

while the result of the lifting-based decomposition is:

$$P(z) = \begin{pmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \beta\left(1+z\right) & 1 \end{pmatrix} \begin{pmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \delta\left(1+z\right) & 1 \end{pmatrix} \begin{pmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{pmatrix} \tag{12}$$

where $\alpha = -1.5861343420$, $\beta = -0.052980118$, $\gamma = 0.882911075$, $\delta = 0.443506852$ and $\zeta = 1.230174105$.

1.2 Symmetric Mask-based Wavelet Transform (SMDWT)

In [9], authors present a novel way of computing the wavelet transform trying to reduce the computational complexity for the wavelet filtering process. The Symmetric Mask-based Discrete Wavelet Transform (SMDWT) is performed as a matrix convolution, using four matrix derived from the 2D DWT of 9/7 floating point lifting-based coefficients (see Figure 2). The 2D LDWT lifting scheme require 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 9/7 filter.

2 Multicore Wavelet Transform

We have used the regular filter-bank convolution based on Daubechies 9/7 filter, in order to develop the optimized parallel 2D discrete wavelet transform (DWT), proposed in [4]. On the other hand, we have used the lifting scheme proposed by Sweldens in [5], in order to develop the optimized parallel 2D lifting wavelet transform (LWT). As we have previously mentioned, we require the image size memory space to store the computed

Image Size	Cores	Extra memory size		Image Size	Cores	Extra memory size	
		Conv.	Lifting			Conv.	Lifting
512 x 512	1	520	1024	2048 x 2560	1	2568	4608
	2	1040	2048		2	5136	9216
	4	2080	4096		4	10272	18432
2048 x 2048	1	2056	4096	4096 x 4096	1	4104	8192
	2	4112	8192		2	8208	16384
	4	8224	16384		4	16416	32768

Table 1: Amount of extra memory size using four-tap filter (pixel size).

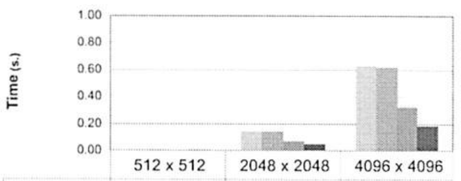
wavelet coefficients. In the convolution based wavelet transform, an extra memory space to store the current image row/column is required. On the other hand, in lifting wavelet based transform, we need the memory space to store a copy of both one row and one column. Remark that, the SMDWT algorithm requires twice the image size space to perform the four mask filtering.

We have used OpenMP [13] paradigm in order to develop the parallel algorithms. The multicore platform used is an Intel Core 2 Quad Q6600 2.4 GHz, with 4 cores, where a block of rows and a block of columns has been assigned to one process in each core to compute the wavelet transform, therefore each process (or core) requires the above mentioned 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 in this manner to double the memory requirements. Table 1 shows the amount of extra memory in pixels (i.e. floats) used by each algorithm depending on the number of cores used. As it can be seen, 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. Note that we work with grayscale images where a pixel is represented by a float, therefore the data shown in Table 1 are pixels or floats.

The operating system used by the multicore platform is Ubuntu 9.04 (Jaunty Jackalope) for 64 bit systems. We have used the GNU compiler gcc included in gcc 4.3.3. Compiler flags used to exploit the multicore architecture are: "-O3 -m64 -fopenmp", while the ones used to avoid multicore architecture are: "-O3 -m64".

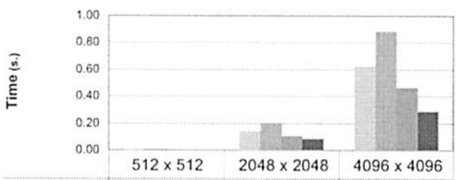
We have considered two scenarios for the parallel algorithms. In the first one, we assign a set of consecutive rows/columns to each processor, while in the second scenario the compiler perform the distribution of computational load. We will not present different results for both scenarios because the computational times obtained are quite similar.

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. In figure 3 we show the computational times obtained for both convolution-based and lifting wavelet transform, for different images sizes: 512 × 512, 2048 × 2048, and 4096 × 4096 pixels. Although the memory access bottleneck is the major obstacle



	512 x 512	2048 x 2048	4096 x 4096
░ CPU	0.0051	0.1387	0.6278
▒ CPU-1Core	0.0050	0.1384	0.6205
▓ CPU-2Core	0.0029	0.0696	0.3240
█ CPU-4Core	0.0039	0.0508	0.1839

(a) Convolution



	512 x 512	2048 x 2048	4096 x 4096
░ CPU	0.0051	0.1387	0.6278
▒ CPU-1Core	0.0037	0.2066	0.8870
▓ CPU-2Core	0.0020	0.1055	0.4626
█ CPU-4Core	0.0033	0.0871	0.2865

(b) Lifting

Figure 3: Computational times for multicore fast wavelet transform algorithms.

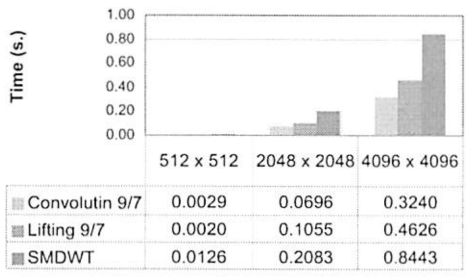
to obtain ideal efficiencies, in Figure 3 we can observe that the computational time decreases, except for small images, as we increase the number of processes. Note that each core executes only one process. Working with small pictures do not achieve good performance due to the relationship between the computational load and the memory accesses degrading the inherent parallelism. Note that each column and/or row has few elements, hence the work performed over each row or column stored in the buffer is not significant. If we calculate efficiency between multicores algorithms, we obtain an efficiency closely ideal one using 2 cores, while we obtain a good efficiency using 4 cores. Note that the memory access bottleneck get worse as the number of cores increase because the number of entities that use the memory is greater.

Finally, we have compared our algorithms with a recent and not classical implementation of the fast DWT called "symmetric mask-based DWT" (SMDWT) [9]. We have developed the method introduced in [9] and also, we have parallelized its reference algorithm. In Figure 4 we present a comparison between convolution, lifting and the SMDWT algorithm, using two and four cores. As it can be seen, our convolution and lifting implementations are 2.5 times as fast as the SMDWT algorithm. Note that the authors in [9] propose the SMDWT algorithm to improve the computational complexity of the lifting scheme and also for the ability of the SMDWT algorithm to compute the four subbands (LL, LH, HL and HH) independently.

Some applications only require computing the LL subband, in Figure 5 we present the same comparison as the one in Figure 4, only computing the LL subband when SMDWT algorithm is used, and computing all subbands in our algorithms. Note that the behavior of our algorithms computing the four subbands is similar to the SMDWT behavior only computing the LL subband.

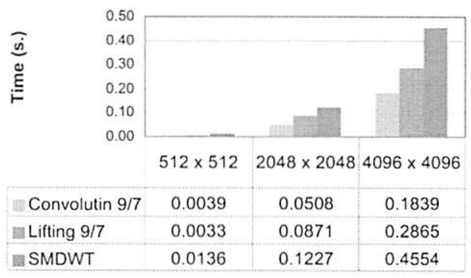
3 CUDA GPU-based Wavelet Transform Algorithm

In the previous section, we have confirmed that our shared memory parallel algorithm for computing the 2D DWT presents a good behavior. Moreover, we question in this section if better performance can be achieved with other architecture. The Graphical Processor



	512 x 512	2048 x 2048	4096 x 4096
▨ Convolutin 9/7	0.0029	0.0696	0.3240
▨ Lifting 9/7	0.0020	0.1055	0.4626
▨ SMDWT	0.0126	0.2083	0.8443

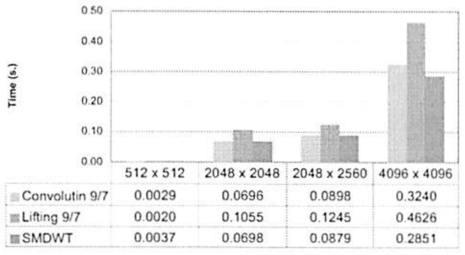
(a) 2 Cores



	512 x 512	2048 x 2048	4096 x 4096
▨ Convolutin 9/7	0.0039	0.0508	0.1839
▨ Lifting 9/7	0.0033	0.0871	0.2865
▨ SMDWT	0.0136	0.1227	0.4554

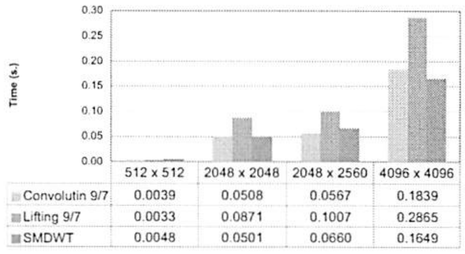
(b) 4 Cores

Figure 4: Comparison between Convolution, Lifting and SMDWT algorithms.



	512 x 512	2048 x 2048	2048 x 2560	4096 x 4096
▨ Convolutin 9/7	0.0029	0.0696	0.0898	0.3240
▨ Lifting 9/7	0.0020	0.1055	0.1245	0.4626
▨ SMDWT	0.0037	0.0698	0.0879	0.2851

(a) 2 Cores



	512 x 512	2048 x 2048	2048 x 2560	4096 x 4096
▨ Convolutin 9/7	0.0039	0.0508	0.0567	0.1839
▨ Lifting 9/7	0.0033	0.0871	0.1007	0.2865
▨ SMDWT	0.0048	0.0501	0.0660	0.1649

(b) 4 Cores

Figure 5: Comparison between Convolution, Lifting and LL subband computation using SMDWT algorithm.

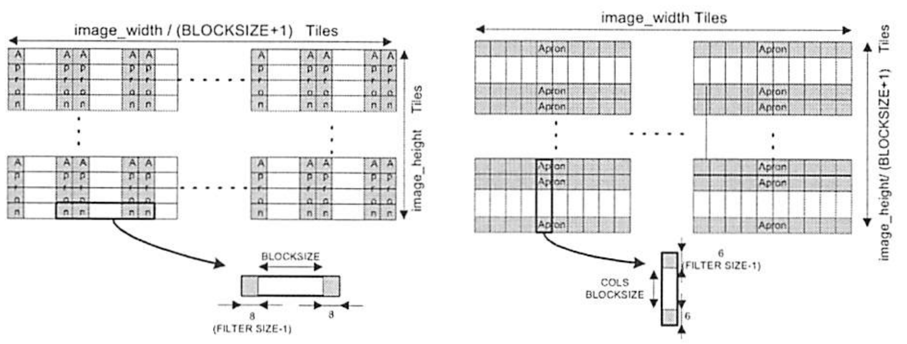
Unit (GPU) architecture is based on a set of multiprocessor units called streaming multiprocessors (SM), containing each one a set of processor cores called streaming processors (SP). CUDA is a heterogeneous computing model that involves both the CPU and the GPU. In the CUDA parallel programming model [14, 15], an application consists of a sequential host program, that may execute parallel programs known as kernels on a parallel device, i.e. a GPU. Note that the CPU could be a multicore processor running an OpenMP model program, but in this case only one of the available cores can call a kernel, i.e. kernel calls must be serialized, therefore we do not use both models in an single algorithm. A kernel is an Single Program Multiple Data (SPMD) computation that is executed using a potentially large number of parallel threads. Each thread runs the same scalar sequential program. The programmer organizes the threads of a kernel into a grid of thread blocks. The threads of a given block can cooperate among themselves using a barrier synchronization. The main kind of memory units available in GPUs are: the global memory, which has the highest latency; the constant and the texture memory units, which are read only units and, the shared memory and the registers, which both are on-chip memory units. The shared memory is owned by a block while the registers are owned by a thread.

So, in order to implement a GPU-based algorithm with the same scheme that the one presented in Section 2, the key element is the use of shared memory to store the buffer that contains a copy of the working data row, and the constant memory to store the filter taps $h[n]$ and $g[n]$. We call each CUDA kernel with a one-dimensional number of blocks NBLOCKS and a one-dimensional number of threads NTHREADS. The number of blocks (NBLOCKS) must be equal to or greater than the maximum size of a row or a column. Each block computes a single row or a single column, the copy of the row or column to be computed is stored in the GPU shared memory. Remark that the available size of shared memory in a GTX 280 is 16KB.

Note that, one of the main goals, in the proposed CUDA based methods is to minimize memory requirements, so we will store the resulting wavelet coefficients in the image memory space. On the other hand, the methods included in the CUDA SDK [10] use three times the image size. These methods perform two steps; in the first step, compute and store, the convolution of rows in GPU global memory, and, in the second step, compute and store the convolution of columns. Remark that, the memory requirements of these methods can be easily reduced using the image memory space to store the wavelet coefficients of the second step. Nevertheless, the memory requirements, using this improvement, is twice the size of the image. We have developed two methods based on the naive implementation described in the SDK (see [10, 16]), the first one using global memory (*CUDA-Mem 9/7*), and the second one using texture memory (*CUDA-Text 9/7*).

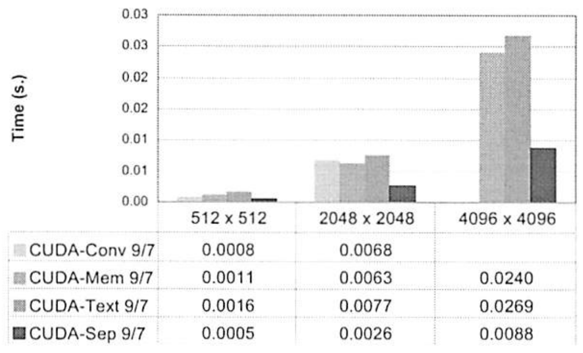
As proposed in [10], the behavior of these methods computed over a GPU can be improved optimizing the memory coalescence. In order to optimize the memory coalescence, separable filters must be used. Using a separable filter allows the convolution of rows and convolution of columns to be computed separately. Based on the properties of separable filters, we have developed the method *CUDA-Sep 9/7*, which uses the Daubechies 9/7 filter. The expected improvement should be based on (a) the reduction of the times the pixels are read, (2) on coalescing access to memory, (3) high memory throughput, and (4) the reduction of the number of idle threads (see [10]). As we have said, the convolution is separated in two stages, 1) the rows stage and 2) the columns stage; each stage is separated into two sub-stages, a) the first sub-stage loads the data from global memory into shared memory, and b) the second sub-stage processes the data and stores the results in the global memory. In the computation stage, as it can be seen in Figure 6, each thread loops over a width of twice the filter radius plus 1 (8 in rows and 6 in columns for Daubechies 9/7 filter), multiplying each pixel by the corresponding filter tap stored in the constant memory. Each thread in a half-warp (a warp is composed by 32 CUDA threads) accesses to the same constant memory address and hence there is no penalty due to constant memory bank conflicts. Also, consecutive threads always access consecutive shared memory addresses so no shared memory bank conflicts occur as well, see [10] for a detailed description.

In Figure 7, we compare execution times to obtain the 2D DWT using the four proposed CUDA based algorithms. The four algorithms considered are: the algorithm based on convolution described in Section 1 (labeled *CUDA-Conv 9/7*); the naive aforementioned algorithms described in the SDK, the first one using global memory (labeled as *CUDA-Mem 9/7*) and the second one using texture memory (labeled as *CUDA-*



(a) Shared memory for row filter (b) Shared memory for column filter

Figure 6: Shared Memory for separable filter 9/7

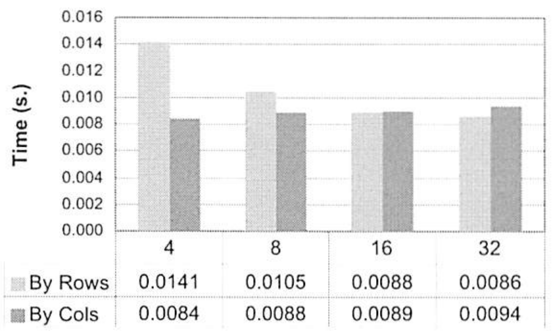


	512 x 512	2048 x 2048	4096 x 4096
▨ CUDA-Conv 9/7	0.0008	0.0068	
▨ CUDA-Mem 9/7	0.0011	0.0063	0.0240
▨ CUDA-Text 9/7	0.0016	0.0077	0.0269
■ CUDA-Sep 9/7	0.0005	0.0026	0.0088

Figure 7: 2D DWT computation over GPUs with CUDA

Text 9/7); and the algorithm developed to exploit the characteristics of the convolution based on a separable filter (labeled as *CUDA-Sep 9/7*). Daubechies 9/7 filter is used in all experiments performed. In Figure 7 we can observe that the results obtained by the proposed algorithm *CUDA-Conv 9/7* are similar to results obtained by algorithms *CUDA-Mem 9/7* and *CUDA-Text 9/7*, note that the memory requirement of algorithm *CUDA-Conv 9/7* is the lowest one, because the image is overwritten with wavelet coefficients. On the other hand, the best results are obtained using the algorithm *CUDA-Sep 9/7*, note that in this algorithm we optimize the memory coalescence using a separable filter. We want to point out that the speed-up obtained is up to 2.7 for 4096×4096 image size.

In algorithm *CUDA-Sep 9/7* the shared memory stores a block of pixels of one row or a block of one column, the block data stored in shared memory will be computed by a CUDA block. Due to each block of threads computes a block data, the number of threads by block must be selected according the row block size and column block size. Figure 6 shows this structure for both rows and columns. In Figure 8 we present results varying the row block size and column block size for 4096×4096 image size. The best



	4	8	16	32
░ By Rows	0.0141	0.0105	0.0088	0.0086
▓ By Cols	0.0084	0.0088	0.0089	0.0094

Figure 8: Algorithm *CUDA-Sep 9/7* varying column and row block size

results are obtained with the row block size equal to 16 or 32 and with the column block size equal to 4, 8 or 16. Since shared memory is limited (16 Kbytes in GTX 280), the smaller optimal values of row block size and column block size must be used.

4 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 using two processors and above 2.4 and up to 3.4 using four processors. Since the best results over a multicore platform have been obtained by the convolution algorithm which also requires a 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. We have also compared several CUDA-based algorithms, based on both the proposed multicore-based algorithms and the CUDA SDK proposals. In conclusion, we would like to point out that (1) the use of a multicore platform obtains good performance, and (2) we obtain a good speed-up in a GPU respect to the results obtained in the multicore platform. The CUDA based algorithm to be chosen depends on the parameters to optimize, which can be either the computation time or the memory requirements.

Acknowledgements

This research was supported by the Spanish Ministry of Education and Science under grant DPI2007-66796-C03-03 and the Spanish Ministry of Science and Innovation under grant number TIN2008-06570-C04-04.

References

[1] K. Rao and P. Yip. Discrete cosine transform: Algorithms, advantages, applications. In *Academic Press, USA*, 1990.

[2] ISO/IEC 15444-1. JPEG2000 image coding system, 2000.

[3] A. Said and A. Pearlman. 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, 1996.

[4] S. G. Mallat. A theory for multi-resolution signal decomposition: The wavelet representation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 11(7):674–693, July 1989.

[5] W. Sweldens. The lifting scheme: a custom-design construction of biorthogonal wavelets. *Applied and Computational Harmonic Analysis*, 3(2):186–200, April 1996.

[6] W. Sweldens. The lifting scheme: a construction of second generation wavelets. *SIAM Journal on Mathematical Analysis*, 29(2):511–546, March 1998.

[7] C. Chrysafis and A. Ortega. Line-based, reduced memory, wavelet image compression. *IEEE Transactions on Image Processing*, 9(3):378–389, March 2000.

[8] Y. Bao and C.C. Jay Kuo. Design of wavelet-based image codec in memory-constrined environment. *IEEE Trans. on Circuits and Systems for Video Technology*, 11(5):642–650, May 2001.

[9] Chih-Hsien Hsia, Jing-Ming Guo, Jen-Shiun Chiang, and Chia-Hui Lin. A novel fast algorithm based on smdwt for visual processing applications. In *Circuits and Systems, 2009. ISCAS 2009. IEEE International Symposium on*, pages 762 -765, May 2009.

[10] V. Podlozhnyuk. Image convolution with cuda, June 2007.

[11] I. Daubechies and W. Sweldens. Factoring wavelet transforms into lifting steps. *Fourier Analysis and Applications*, 4(3):247–269, 1998.

[12] J.M. Shapiro. Embedded image coding using zerotrees of wavelet coefficients. *IEEE Transactions on Signal Processing*, 41(12), December 1993.

[13] OpenMP Architecture Review Board. Openmp c and c++ application program interface, version 2.0. March 2002.

[14] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable parallel programming with cuda. In *Queue*, volume 6, pages 40–53, 2008.

[15] NVIDIA Corporation. Nvidia cuda c programming guide. version 3.2.

[16] Ian Buck. Gpu computing with nvidia cuda. In *ACM SIGGRAPH 2007 courses*, SIGGRAPH '07, New York, NY, USA, 2007. ACM.