Algoritmos eficientes para la transformada wavelet discreta en multicores y GPUs

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

Resumen— **En este trabajo se analiza el comportamiento del rendimiento para un conjunto de algoritmos utilizados en el cálculo de la transformada *wavelet* discreta 2D (2D-DWT) utilizando tanto OpenMP sobre plataformas de memoria compartida o multinúcleo, como CUDA (Compute Unified Device Architecture) sobre GPUs (Graphics Processing Unit). Los algoritmos propuestos se basan en el uso de filtros de convolución y en la transformada lifting. Además, se compara el rendimiento obtenido por nuestros algoritmos frente al reciente algoritmo SMDWT (Symmetric Mask-based Wavelet Transform) en plataformas de memoria compartida; y frente a algoritmos basados en las propuestas de la SDK de CUDA cuando se usa una GPU como plataforma de computación.**

Palabras clave— **CUDA, OpenMP, transformada wavelet, codificación de imagen, algoritmos paralelos**

Introducción

Durante la última década, se ha profundizado en diferentes algoritmos de compresión de imágenes para evitar las conocidas limitaciones de los algoritmos basados en bloques tales como la DCT [1] (transformada discreta del coseno), siendo ésta la técnica de compresión más utilizada hasta el momento. Algunas de las alternativas propuestas están basadas en técnicas más complejas como la codificación fractal o la cuantización vectorial, mientras otras simplemente proponen el uso de una transformada matemática diferente y más flexible. La transformada DWT (transformada discreta wavelet) se ha consolidado como una herramienta muy potente para la compresión de imágenes y un gran número de codificadores de imágenes actuales, incluyendo el estándar JPEG2000, utilizan esta transformada en sus algoritmos (ver por ejemplo [2], [3]).

Sin embargo, a pesar de los beneficios implícitos del uso de la transformada wavelet, se presentan nuevos problemas relacionados tanto con el incremento de la complejidad, como con el acceso intensivo a memoria, lo que conlleva un incremento en los tiempos de ejecución. De hecho, en la implementación clásica de la transformada DWT [4], la descomposición de la imagen se realiza mediante un filtrado convolucional, cuya complejidad aumenta en función de la longitud del filtro utilizado. Además, en el cálculo de la transformada DWT, para cada nivel de descomposición se realizan dos pasadas a la imagen, una por filas y otra por columnas, por lo que es necesario almacenar toda la imagen en memoria. Por otra parte, en la transformada DCT, al realizarse por bloques, el requisito de memoria no es un problema incluso para imágenes de gran tamaño.

[1]Dpto. Física y Arquitectura de Computadores, Univ. Miguel Hernández, e-mail: {vgaliano,otoniel,mels,hmigallon}@umh.es

El esquema lifting [5], [6] es, probablemente, el algoritmo más estudiado para realizar un cálculo eficiente de la transformada wavelet. Dicho algoritmo utiliza menos coeficientes en los filtros, proporcionando una implementación más rápida de la transformada. Además, este esquema reduce la memoria necesaria, ya que los coeficientes wavelet calculados son almacenados en la matriz de datos original, evitando de esta manera la necesidad de otra matriz para almacenar dichos coeficientes wavelet. Hay que tener en cuenta que los coeficientes wavelet calculados deben almacenarse en posiciones separadas dentro de la matriz de datos, las bajas frecuencias por un lado y las altas por otro, este reordenamiento de coeficientes produce conflictos en el uso de la memoria caché. Por otra parte, se han propuesto otros algoritmos para el cálculo de la transformada wavelet con el fin de reducir los requisitos de memoria, como por ejemplo los algoritmos basados en bloques [7] o los basados en línea [8]. Estas propuestas aumentan la flexibilidad del algoritmo para imágenes de gran tamaño y reducen los requisitos de memoria. Recientemente, en [9], los autores presentan un nuevo método denominado *Symmetric Mask-based Discrete Wavelet Transform* (SMDWT). Este algoritmo realiza el cálculo de la transformada como si se tratase de una convolución matricial, de manera que utiliza cuatro matrices, una para cada subbanda (LL, HL, LH y HH), con la intención de reducir los cálculos repetitivos necesarios en el algoritmo tradicional. En este esquema, la transformada 2D-DWT se realiza en una sola pasada. Además, este algoritmo permite el cálculo independiente de una única subbanda.

En el diseño de codificadores de imagen y vídeo basados en la transformada wavelet, una de las tareas computacionalmente más costosa es la transformada wavelet puesto que suele emplear entre el 30% y el 50% del tiempo total de codificación (dependiendo del tamaño de la imagen y del número de niveles de decomposición wavelet). Por lo tanto, es vital reducir el tiempo de cálculo de la transformada 2D-DWT desarrollando codificadores eficientes que aprovechen recursos computacionales disponibles en cada computador. En este sentido, la mayor parte de computadores actuales incluyen procesadores multinúcleo de manera que un algoritmo eficiente debe poder aprovechar la capacidad de cálculo en paralelo utilizando simultáneamente varios o todos los núcleos de dichos procesadores. Por otro lado, las GPUs se encuentran cada vez más presentes en los equipos de consumo, por lo que también deben considerarse para aprovecharlas como plataformas de cómputo.

En este artículo, realizaremos unas optimizaciones

para el cálculo en paralelo sobre los métodos descritos en [4] y [5]. Nuestro principal objetivo es la optimización del uso de la memoria y la mejora del rendimiento mediante la utilización de arquitecturas multinúcleo, es decir utilizando plataformas de memoria compartida. Además, se diseñarán para su ejecución en GPUs mediante CUDA, los mismos algoritmos desarrollados para arquitecturas multinúcleo. Hay que remarcar que los algoritmos desarrollados para GPUs requieren de un uso eficiente de la memoria, en particular los algoritmos diseñados se basan en el uso de la memoria compartida de la GPU. Además, se presentarán nuevos métodos basados en las propuestas incluidas en [10], comparándolos tanto a nivel de rendimiento como a nivel de requisitos de memoria.

I. Transformada Discreta Wavelet (DWT)

La transformada DWT obtiene un esquema de descomposición multirresolución para señales de entrada. La señal original se transforma inicialmente en dos subbandas de frecuencia (bajas y altas frecuencias). En la transformada clásica, la descomposición se realiza mediante un filtro digital paso bajo H y un filtro digital paso alto G. Ambos filtros se diseñan utilizando la función de escalado $\Phi(t)$ y los correspondientes coeficientes wavelet $\Psi(t)$. El algoritmo reduce el nivel de muestras de la señal a la mitad. En filtros FIR no recursivos y con longitud L, la función de transferencia de H y G se puede representar del siguiente modo:

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

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

A. Transformada Wavelet Lifting (LDWT)

Uno de los inconvenientes de la DWT es el incremento de los requisitos de memoria debido a su algoritmo basado en la convolución de filtros. Una propuesta que reduce el tamaño de memoria necesaria en el cálculo de la transformada es el esquema lifting [5]. El principal beneficio en este esquema es la reducción del número de operaciones necesarias para realizar la transformada wavelet, comparándolo con el algoritmo convolucional clásico. El orden de reducción en el esquema lifting depende del tipo de transformada wavelet a realizar, tal y como se indica en [11].

En el esquema clásico, el procesado in-situ de los datos no es posible ya que cada muestra original se necesita para el cálculo de los coeficientes en sus vecinos. Por lo tanto, se necesita una nueva matriz para almacenar los coeficientes resultantes, duplicando, de esta manera, los requisitos de memoria. No obstante, en el esquema lifting se implementa una computación in-situ sin necesidad de memoria adicional alguna. Además, el esquema lifting se puede ejecutar sobre un número impar de muestras mientras que en el algoritmo clásico se necesita de un número par de muestras.

Usaremos el algoritmo euclídeo para factorizar la matriz en varias fases mediante una secuencia alternativa de matrices triangulares superiores e inferiores. En (3), las variables $h(z)$ y $g(z)$ representan la inversa de los filtros paso bajo y paso alto respectivamente. Dichos filtros se dividen en una parte impar y otra par para generar una matriz $P(z)$, como se muestra en (4).

$$\begin{aligned} g(z) &= g_e\left(z^2\right) + z^{-1} g_o\left(z^2\right) \\ h(z) &= h_e\left(z^2\right) + z^{-1} g_o\left(z^2\right) \end{aligned} \quad (3)$$

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

Mediante el algoritmo euclídeo, de manera recursiva, encontraremos los máximos comunes divisores de la parte par e impar de los filtros originales. De este modo, $h(z)$ y $g(z)$ forman un filtro complementario que se puede factorizar en tres pasos tal y como se muestra a continuación,

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

donde $s_i(z)$ y $t_i(z)$ representan los polinomios de Laurent correspondientes a los pasos de predicción y actualización respectivamente, y k es una constante distinta de cero.

El proceso completo consiste en una primera transformación aproximada, uno o más pasos de predicción y una posterior actualización y normalización de los coeficientes. En la primera transformación, las muestras de entrada se dividen en dos conjuntos de datos, las muestras pares y las impares. De este modo, si consideramos $\{X_i\} = \{\Phi_{n,p}\}$ como las muestras de entrada en el nivel de descomposición n, definimos:

$$\begin{aligned} \left\{s_i^0\right\} &= \{X_{2i}\} \\ \left\{d_i^0\right\} &= \{X_{2i+1}\} \end{aligned} \quad (6)$$

Por tanto, en un paso de predicción, cada muestra en $\left\{d_i^0\right\}$ se reemplaza por el error cometido en la predicción de esa muestra con respecto a las muestras $\left\{s_i^0\right\}$:

$$d_i^1 = d_i^0 - P\left(\left\{s_i^0\right\}\right) \quad (7)$$

mientras que en el paso de actualización, cada muestra en $\left\{s_i^0\right\}$ se actualiza por $\left\{d_i^1\right\}$:

$$s_i^1 = s_i^0 + U\left(\left\{d_i^1\right\}\right) \quad (8)$$

Después de m sucesivos pasos de actualización y predicción, se obtienen los coeficientes de escalado y wavelet del siguiente modo:

$$\begin{aligned} \{\Phi_{n+1,p}\} &= K_0 \times \{s_i^m\} \\ \{\Psi_{n+1,p}\} &= K_1 \times \{d_i^m\} \end{aligned} \quad (9)$$

Un caso especial de filtro wavelet es el filtro Daubechies 9/7. Este filtro se utiliza frecuentemente

en compresión de imagen (ver por ejemplo [3], [12]), y se ha incluido en el estándar JPEG2000 [2]. En este trabajo, todos los algoritmos desarrollados para cálculo de la transformada DWT utilizan este filtro. Los coeficientes de los filtros de descomposición Daubechies 9/7 $h[n]$ y $g[n]$ son:

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

mientras que la descomposición basada en lifting resulta:

$$P(z) = \begin{pmatrix} 1 & \alpha\left(1+z^{-1}\right) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \beta(1+z) & 1 \end{pmatrix}$$

$$\begin{pmatrix} 1 & \gamma\left(1+z^{-1}\right) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ \delta(1+z) & 1 \end{pmatrix} \begin{pmatrix} \zeta & 0 \\ 0 & 1/\zeta \end{pmatrix} \quad (10)$$

donde $\alpha = -1.586134342$, $\beta = -0.052980118$, $\gamma = 0.882911075$, $\delta = 0.443506852$ y $\zeta = 1.230174105$.

B. *Symmetric Mask-based Wavelet Transform (SMDWT)*

En [9], los autores presentan una novedosa forma de calcular la transformada wavelet tratando de reducir la complejidad computacional. La transformada SMDWT se realiza mediante una convolución matricial utilizando cuatro matrices derivadas del filtro Daubechies 9/7 con coeficientes en coma flotante. En el esquema lifting 2D LDWT se requiere de un cálculo en sentido vertical y otro en sentido horizontal, y para cada uno de estos cálculos se deben realizar cuatro pasos: división, predicción, actualización y escalado. Por el contrario, las cuatro bandas en la transformada 2D SMDWT se pueden obtener de forma independiente mediante cuatro matrices de tamaños 7×7, 7×9, 9×7 y 9×9 en el caso del filtro Daubechies 9/7.

II. Transformada Wavelet multinúcleo

Hemos utilizado la convolución basada en el filtro Daubechies 9/7 con el objetivo de desarrollar una computación de la transformada 2D-DWT propuesta en [4] de forma paralela y optimizada. Por otro lado, hemos utilizado el algoritmo lifting propuesto por Sweldens en [5], para optimizar la transformada lifting (2D-LWT).En los filtros basados en la convolución, necesitaremos un espacio de memoria adicional para almacenar la fila o columna de la actual convolución, mientras que en la transformada lifting necesitaremos un espacio de memoria adicional para almacenar una fila y una columna. Debemos destacar que el algoritmo SMDWT requiere el doble del tamaño de la imagen.

Hemos utilizado OpenMP [13] como herramienta de desarrollo de algoritmos paralelos en arquitecturas multinúcleo. La plataforma multinúcleo utilizada es un Intel Core 2 Quad Q6600 2.4 GHz, con 4 núcleos,

Tamaño de imagen	Número de núcleos	Tamaño extra de memoria Conv.	Lifting
512×512	1	520	1024
	2	1040	2048
	4	2080	4096
2048×2048	1	2056	4096
	2	4112	8192
	4	8224	16384
4096×4096	1	4104	8192
	2	8208	16384
	4	16416	32768

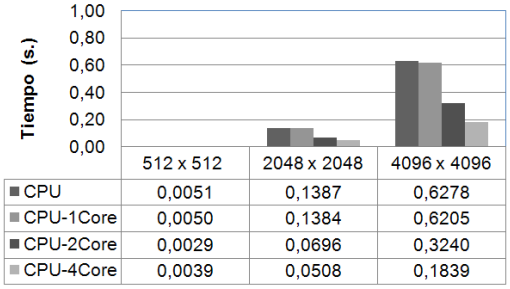
TABLA I
Número de píxeles en memoria necesarios utilizando filtros de cuatro "taps".

denominado SULLI. Cada proceso calcula la transformada wavelet de un bloque de filas y de un bloque de columnas, hay que tener en cuenta que se crean tantos procesos como núcleos hay disponibles. Por tanto, cada proceso (o núcleo) requiere un extra de memoria para realizar la transformada, lógicamente el tamaño total de memoria adicional necesaria aumenta al aumentar el número de núcleos utilizados. Debemos destacar que el resultado de los coeficientes wavelet se puede almacenar en el espacio de memoria ocupada por la imagen original, evitando, de este modo, duplicar los requisitos de memoria. La Tabla I muestra la memoria adicional en píxeles utilizada por cada uno de los algoritmos dependiendo del número de núcleos utilizados. El peor caso se da para imágenes pequeñas que, no obstante, requieren menos del 2% de memoria adicional, siendo para el resto de casos inferior al 1%. Hay que remarcar que la memoria adicional necesaria en el algoritmo SMDWT es el tamaño de la imagen. Los datos mostrados en la Tabla I ilustran una imagen con niveles de grises representados mediante coma flotante.

El sistema operativo de SULLI es Ubuntu 9.04 (Jaunty Jackalope) para sistemas de 64 bits. Hemos utilizado el compilador GNU `gcc` incluido en `gcc 4.3.3`, las opciones de compilación utilizadas para arquitecturas multinúcleo han sido "-O3 -m64 -fopenmp". Para CUDA el compilador utilizado es `nvcc` incluido en el *CUDA Toolkit 3.2 RC*, siendo las opciones de compilación utilizadas "-O3 -m64".

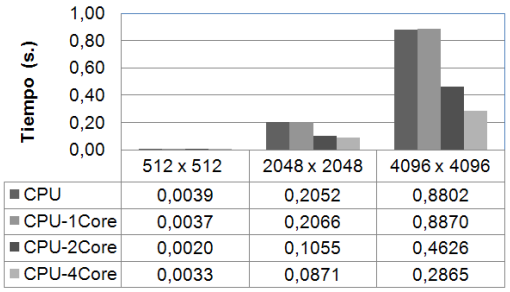
Hemos adaptado los algoritmos para obtener el mejor rendimiento en arquitecturas multinúcleo, teniendo en cuenta que estos algoritmos realizan un uso intensivo de memoria. En la figura 1 se muestran los tiempos de cálculo necesarios para obtener la transformada wavelet mediante convolución y lifting para imágenes de diferentes tamaños: 512×512, 2048×2048, y 4096×4096. No obstante, el cuello de botella en el acceso a memoria provoca pérdidas de eficiencia en el cálculo, ya que para imágenes pequeñas el cómputo asociado a cada fila y a cada columna es muy pequeño. Sin embargo, con imágenes grandes se obtienen eficiencias casi ideales para el caso de 2 núcleos y muy buenas en el caso de 4 núcleos.

Además, hemos comparado nuestros algoritmos con la reciente propuesta para el cálculo de la



	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

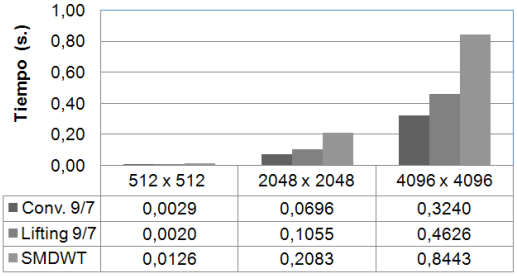
(a) Convolución



	512 x 512	2048 x 2048	4096 x 4096
CPU	0,0039	0,2052	0,8802
CPU-1Core	0,0037	0,2066	0,8870
CPU-2Core	0,0020	0,1055	0,4626
CPU-4Core	0,0033	0,0871	0,2865

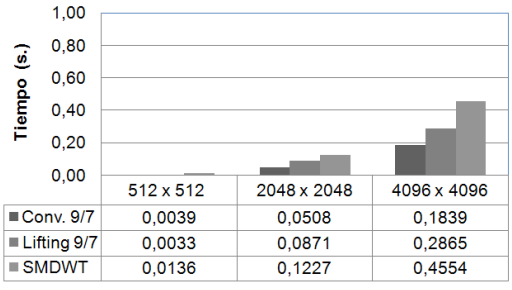
(b) Lifting

Fig. 1. Tiempo de computación de los algoritmos convolución y lifting para arquitecturas multinúcleo.



	512 x 512	2048 x 2048	4096 x 4096
Conv. 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 Núcleos



	512 x 512	2048 x 2048	4096 x 4096
Conv. 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 Núcleos

Fig. 2. Comparación entre los algoritmos basados en la convolución y en la transformada lifting y el algoritmo SMDWT.

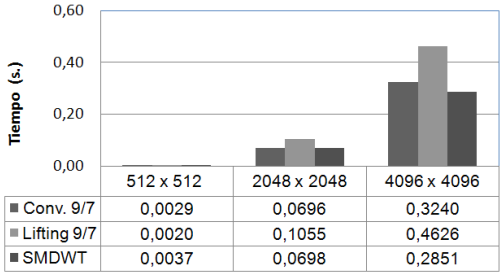
DWT, denominada "Symmetric Mask-based DWT" (SMDWT), introducida en [9] y que propone un enfoque novedoso. Hemos desarrollado el método descrito en [9] y además, hemos paralelizado dicho algoritmo. En la figura 2 mostramos una comparación de los tiempos de cálculo para los algoritmos de convolución, lifting y SMDWT utilizando 2 y 4 núcleos. Como se puede observar, nuestros algoritmos, tanto el basado en la convolución como el basado en la transformada lifting, presentan mejor rendimiento que el algoritmo SMDWT, obteniendo un factor de mejora de hasta $2,5$. Hay que remarcar, que los autores en [9] proponen el algoritmo SMDWT tanto para reducir la complejidad computacional como por la capacidad de dicho algoritmo para obtener de forma independiente cualquiera de las cuatro subbandas (LL, LH, HL o HH).

Algunas aplicaciones requieren únicamente del cálculo de la subbanda LL. En la figura 3 presentamos la misma comparación que en la figura 2 pero calculando únicamente la subbanda LL cuando se utiliza el algoritmo SMDWT. Hay que remarcar que el comportamiento de nuestros algoritmos computando las cuatro subbandas es similar al comportamiento del algoritmo SMDWT computando únicamente la subbanda LL.

III. Transformada Wavelet basada en GPUs

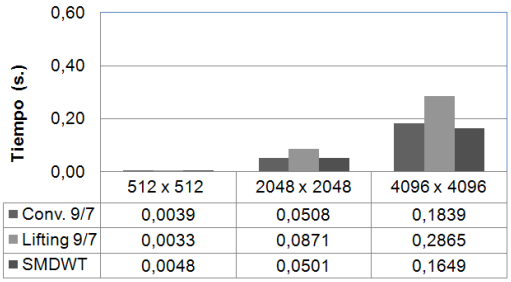
En las secciones anteriores hemos comprobado que los algoritmos desarrollados para plataformas de memoria compartida que calculan la transformada 2D DWT obtienen buenos rendimientos. A continuación, nos preguntamos en esta sección si se puede mejorar ese rendimiento con otro tipo de arquitectura, en particular con el uso de GPUs. Los procesadores gráficos (GPU) están basados en un conjunto de unidades multinúcleo llamadas multiprocesadores de streaming (SM) que contienen cada una de ellas un conjunto de procesadores de streaming (SP). CUDA es un modelo de computación heterogéneo que involucra tanto a la CPU como a la GPU. En la programación paralela con CUDA [14], [15], una aplicación consiste en un programa secuencial, ejecutado en el procesador *h*ost, que puede ejecutar programas, conocidos como *kernels*, en el dispositivo paralelo, es decir en la GPU. Además, el procesador host puede ser un sistema multinúcleo ejecutando códigos paralelos, aunque en este caso únicamente uno de los núcleos podrá realizar llamadas a los kernel de la GPU, o más específicamente las llamadas a los kernels deben serializarse. Un kernel es un programa SPMD (Single Program Multiple Data) que se ejecuta con un número elevado de hilos o threads. Cada hilo ejecuta el mismo programa secuencial. El programador organiza los hilos en una malla de bloques de hilos. Los hilos de un bloque determinado pueden colaborar entre ellos mediante mecanismos de sincronización y mediante los diferentes niveles de memoria de los que dispone una GPU: la memoria global, que es la de mayor latencia; la memoria constante de sólo lectura; la memoria de texturas; la memoria compartida; y los registros. La memoria compartida es visible por todos los hilos de un bloque, mientras que los registros son propios de cada hilo. Hay que tener en cuenta que CUDA no proporciona mecanismos globales de sincronización.

Con el objetivo de implementar el algoritmo basado en la convolución presentado en la sección II sobre una GPU, debemos tener en cuenta que el elemento clave es la memoria compartida. Usaremos esta memoria compartida para almacenar la copia de datos de la fila o la columna con la que estén



	512 x 512	2048 x 2048	4096 x 4096
Conv. 9/7	0,0029	0,0696	0,3240
Lifting 9/7	0,0020	0,1055	0,4626
SMDWT	0,0037	0,0698	0,2851

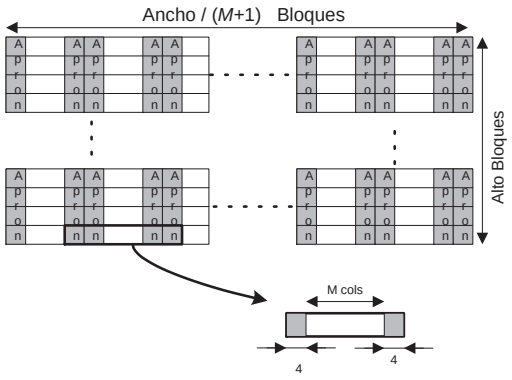
(a) 2 Núcleos



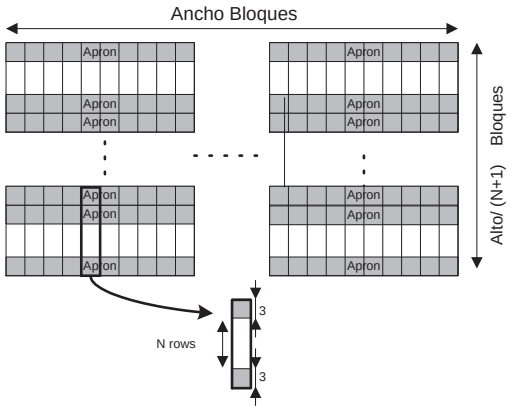
	512 x 512	2048 x 2048	4096 x 4096
Conv. 9/7	0,0039	0,0508	0,1839
Lifting 9/7	0,0033	0,0871	0,2865
SMDWT	0,0048	0,0501	0,1649

(b) 4 Núcleos

Fig. 3. Comparación entre los algoritmos basados en la convolución y en la transformada lifting y el cálculo de la subbanda LL con el algoritmo SMDWT



(a) Convolución por filas



(b) Convolución por columnas

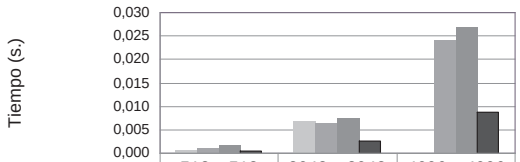
Fig. 4. Distribución por bloques en la memoria compartida de la GPU

trabajando los hilos de un bloque. Por otra parte, almacenaremos en la memoria constante los coeficientes $h[n]$ y $g[n]$ descritos anteriormente. En este algoritmo cada kernel de CUDA es llamado con un número de bloques, y con un número de hilos por bloque. El número de bloques debe ser igual o superior a la dimensión mayor de la imagen, que será o bien el número de filas o bien el número de columnas. Cada bloque calcula una sola fila o columna, copiando dicha fila o columna en la memoria compartida de la GPU. El tamaño de la memoria compartida en la NVIDIA GTX 280 es únicamente de 16KB.

Uno de los principales objetivos conseguidos mediante este algoritmo propuesto ha sido minimizar los requisitos de memoria, de este modo almacenaremos los coeficientes wavelet resultantes en el espacio de memoria de la imagen. Por otro lado, los métodos incluidos en la SDK de CUDA [10] utilizan tres veces el tamaño de la imagen. Estos métodos realizan la convolución en dos pasos: en el primer paso calculan la convolución por filas y almacenan los coeficientes obtenidos en la memoria global de la GPU; y en el segundo paso calculan y almacenan la convolución por columnas en otro espacio de la memoria global de la GPU. Se puede reducir los requisitos de memoria en un tercio si los coeficientes wavelet obtenidos en el segundo paso son almacenados en el espacio ocupado por la imagen original. Por otra parte, basándonos en la SDK de CUDA hemos desarrollado dos métodos de la implementación descrita como convolución clásica (véase [10], [16]), el primero de ellos, utilizando la memoria global de la GPU (*CUDA-Mem 9/7*), y el segundo, utilizando la memoria de tipo textura (*CUDA-Text 9/7*).

Tal y como se propone en [10], el comportamiento en estos métodos se puede mejorar considerablemente optimizando el acceso a memoria para evitar los conflictos (*memory coalescence*). Para poder optimizar el acceso a memoria, los filtros de la convolución deben ser separables en dos pasos, es decir, una convolución por filas y otra por columnas, por tanto la convolución SMDWT descrita en la sección I-B no podría optimizarse con este sistema. El filtro Daubechies 9/7 se divide en una convolución por filas y una posterior convolución por columnas, por tanto es un filtro separable. Hemos implementado una convolución (*CUDA-Sep 9/7*) que mejora el rendimiento mediante a) la reducción de las lecturas de un mismo píxel, b) acceso coalescente a memoria, c) alto rendimiento de la memoria compartida, y d) la reducción del número de hilos en espera.

Esta convolución separable se realiza en dos pasos, uno por filas y otro por columnas. Cada paso se compone de dos etapas, una carga inicial de los datos desde la memoria global de la GPU a la memoria compartida del bloque, y una etapa posterior en la cual cada hilo calcula y almacena los resultados obtenidos en la memoria global. En la etapa inicial de carga de datos se almacenan en la memoria compartida los píxeles asignados a cada bloque, M en convolución por filas y N en columnas, además son necesarios un determinado número de píxeles contiguos al bloque considerado, el número de píxeles contiguos necesarios viene dado por ($filtersize$ –



	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

Fig. 5. Tiempos de ejecución sobre GPUs con CUDA

1)/2, en particular para el filtro Daubechies 9/7 son necesarios 4 píxeles contiguos en cada extremo para las filas y 3 píxeles para las columnas. En la figura 4, estos bloques contiguos y pertenecientes a otros bloques se han representado en gris y señalizados como "Apron".En la etapa de cálculo, tal y como se puede ver en la figura 4, cada hilo lee los datos desde la memoria compartida y los almacena en su correspondiente posición de la imagen final. De este modo, hilos consecutivos acceden a posiciones de memoria consecutivas por lo que no existen conflictos en el acceso a la memoria compartida (una descripción más detallada puede verse en [10]).

En la figura 5, comparamos los tiempos de ejecución obtenidos para la transformada 2D-DWT utilizando las cuatro implementaciones propuestas en CUDA: la implementación en CUDA del algoritmo basado en la convolución descrito en la sección I (nombrado como *CUDA-Conv 9/7*); la implementación del algoritmo básico descrito en la SDK de CUDA utilizando tanto la memoria global (nombrado como *CUDA-Mem 9/7*) como utilizando texturas (nombrado como *CUDA-Text 9/7*); y el algoritmo, descrito en esta sección, que optimiza el acceso a memoria basado en filtros separables (nombrado como *CUDA-Sep 9/7*). En la figura 5 podemos observar que los tiempos obtenidos con el algoritmo propuesto *CUDA-Conv 9/7* son similares a los obtenidos con las implementaciones *CUDA-Mem 9/7* y *CUDA-Text 9/7*. Sin embargo, sí obtenemos una mejora considerable al optimizar el acceso a la memoria compartida usando filtros separables. Por ejemplo, el factor de mejora obtenido por el algoritmo *CUDA-Sep 9/7* es $2,7$ para un tamaño de imagen de 4096×4096, .

IV. Conclusiones

Hemos presentado varios algoritmos para el cálculo de la transformada discreta wavelet, basados tanto en la convolución como en la transformada lifting, en sistemas multinúcleo y en arquitecturas GPU. Además, hemos comparado nuestras propuestas con otras propuestas recientes como la citada SMDWT. El *speed-up* obtenido en sistemas multinúcleo es de $1,9$ utilizando dos procesadores y entre $2,4$ y $3,4$ usando cuatro procesadores. Hemos querido trasladar estos resultados obtenidos en sistemas multinúcleo a arquitecturas con procesadores gráficos, trasladando la memoria temporal de almacenamiento de la fila o columna a la memoria compartida de la GPU. El *speed-up* en la GPU respecto a un sistema multinúcleo ha sido superior a 20. Por otro lado, hemos comparado el rendimiento obtenido en la GPU con otras propuestas de implementaciones similares en CUDA.

Como conclusión, nos gustaría destacar que a) el uso de una arquitectura multinúcleo mejora el rendimiento considerablemente en el cálculo de la DWT, y b) obtenemos una ganancia muy considerable en GPUs que puede ser incluso mejorada con un acceso optimizado a la memoria compartida.

Acknowledgements

El presente trabajo ha sido financiado por el Ministerio de Educación y Ciencia mediante el proyecto DPI2007-66796-C03-03 y por el Ministerio de Educación y Ciencia mediante el proyecto TIN2008-06570-C04-04.

Referencias

[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] 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.

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

[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.