Influencia de las mesetas en la implementación de watershed sobre GPUs

Pablo Quesada-Barriuso, Julián Lamas-Rodríguez, Dora B. Heras y Francisco Argüello[1]

Resumen— **La transformada *watershed* es una técnica no supervisada para la segmentación de imágenes, especialmente adecuada para imágenes con bajo contraste. En este trabajo introducimos la implementación asíncrona sobre GPU de un algoritmo de watershed basado en autómatas celulares, la comparamos en cuanto a eficiencia con otra síncrona y analizamos la influencia sobre dicha eficiencia del tamaño de las regiones de nivel constante de gris llamadas mesetas. También mostramos cómo escala la eficiencia al evolucionar entre generaciones las tarjetas gráficas. La implementación asíncrona desarrollada muestra buenas aceleraciones y características que indican el potencial de este tipo de algoritmos para las tarjetas gráficas de consumo actuales.**

Palabras clave— **Segmentación de imagen, Watershed, Autómatas Celulares, GPGPU, CUDA.**

I. Introducción

La transformada *watershed* (WT) es una herramienta para la segmentación de imágenes. Si representamos una imagen en escala de grises como un relive topográfico, de tal forma que la altura de cada pixel corresponde a su valor de gris, los valles representan valores mínimos y los picos valores máximos. Sumergiendo el terreno en agua, podemos inundar los valles formando cuencas de agua. Si levantamos una presa en las zonas de trasvase de agua entre diferentes cuencas, al final del proceso de segmentación tendremos la imagen dividida en regiones separadas por dichas presas. Las divisiones se conocen como líneas de watershed [1].

Una de las principales ventajas de WT es que todas las regiones de la imagen están definidas al final de proceso de segmentación, incluso si el contraste de la imagen es bajo. Hay que señalar que la segmentación mediante WT es una tarea computacionalmente intensiva. Se pueden encontrar varias definiciones, algoritmos e implementaciones de WT en la literatura, que se clasifican básicamente en dos grupos: basados en la especificación del algoritmo recursivo de Vincent & Soille [2], y basados en la definición de funciones de distancia de Meyer [3]. En el campo del procesamiento paralelo la mayoría de las publicaciones se han centrado en multiprocesadores [4], y en menor medida en arquitecturas específicas [5] como FPGAs (Field Programmable Gate Arrays). Más recientemente se han publicado algunas implementaciones de WT en GPU considerando el soporte para computación de propósito general disponible en dichas tarjetas [6–8].

Los autómatas celulares (CAs) constituyen un modelo de computación que ha sido ampliamente utilizado en inteligencia artificial para reconocimiento de patrones o procesamiento de imágenes. La popularidad de los CAs se debe principalmente a su simplicidad para modelar problemas complejos simplemente con la ayuda de información local. El concepto de paralelismo está implícito en los CAs y coincide con el modelo de computación de las GPUs, ya que están basados en la evolución individual de diferentes celdas. Por esta razón se están desarrollando distintos algoritmos para GPU basados en CAs [9, 10].

Las primeras propuestas de watershed basadas en CAs fueron implementadas en procesadores vectoriales [5], y programando el cauce clásico de la GPU mediante shaders [6]. Estas implementaciones actualizan todos los píxeles de la imagen de forma síncrona. Galilée et al. [11] propone un algoritmo de watershed asíncrono basado en CAs para una arquitectura VLSI, donde cada pixel se actualiza tan pronto haya información nueva disponible. Este algoritmo asíncrono es particularmente adecuado para CUDA, ya que diferentes regiones de la imagen se pueden actualizar de forma simultánea e independiente.

En este artículo presentamos una implementación asíncrona en CUDA del algoritmo de watershed basado en un CA definido en [11], que explota la jerarquía de memoria de forma eficiente y reduce respecto de una implementación síncrona el número de sincronizaciones necesarias entre bloques.

Este trabajo se organiza de la siguiente manera: la sección II describe el algoritmo de watershed basado en CAs y la sección III la arquitectura de la GPU y CUDA. La implementación del algoritmo está descrita en la sección IV y los resultados se discuten en la sección V. Por último, la sección VI presenta las conclusiones.

II. Watershed basado en CA

En esta sección se presenta el algoritmo de watershed introducido en [11], que puede ser implementado de forma síncrona o asíncrona. Antes de explicar el algoritmo es necesario introducir algunos conceptos y notaciones sobre topografía, así como conceptos básicos asociados a watershed y CAs.

Se puede considerar una imagen en escala de grises como un grafo $G = (V, A)$ formado por un conjunto finito V de vértices (píxeles) y un conjunto de arcos $A \subseteq V \times V$ que definen la conectividad. Dos píxeles u y v están conectados si $(u, v) \in A$. Los píxeles conectados a u, llamados vecinos, se expresan por $\mathcal{N}(u)$. Lo más común es usar una conectividad de cuatro, considerando los vecinos a la izquierda, derecha, arriba y abajo.

[1]Centro de Investigación en Tecnoloxías da Información. Universidad de Santiago de Compostela. Santiago de Compostela, España. E-mail: {pablo.quesada, julian.lamas, dora.blanco, francisco.arguello}@usc.es.

La *pendiente* entre dos vecinos está definida por:

$$\forall u \in V, \forall v \in \mathcal{N}(u), \quad \text{slope}(u,v) = h(u) - h(v), \quad (1)$$

donde $h(u)$ es la altitud (valor de gris) del pixel u. La *pendiente descendente* (LS) se define como:

$$LS(u) = \max(\ h(u) - h(v)\ ) \mid v \in \Gamma(u), \quad (2)$$

siendo $\Gamma(u)$ el conjunto de vecinos v tal que $h(v) < h(u)$. Si $\Gamma(u)$ tiene más de un elemento, $\mathcal{N}^F(u)$ representa un elemento arbitrario de dicho conjunto. En el resto del artículo usaremos el término pendiente para hacer referencia a LS.

Una meseta (denotada mediante P) es una región con un valor constante de gris dentro de la imagen, definida como un subgrafo $P = (V_P, A_P) \subseteq G$, donde $\forall u \in V_P, h(u) = c$, siendo c la altitud de la meseta. El conjunto $\mathcal{N}^=(u)$ representa los pixeles $v \in \mathcal{N}(u)$ tal que $h(u) = h(v)$.

Por último, el *borde inferior* de una meseta P se define como:

$$\partial_P^- = \{\ u \in V_P \mid \exists v \in \mathcal{N}(u), h(v) < h(u)\ \}. \quad (3)$$

Si $\partial_P^- = \emptyset$, la meseta se denomina *meseta mínima*. Todos los píxeles dentro de una meseta mínima se consideran mínimos. Por contra, si $\partial_P^- \neq \emptyset$, la meseta se denomina *meseta no-mínima*.

La transformada watershed es una técnica de segmentación basada en regiones. Para computarla seguimos el algoritmo *Hill-Climbing* basado en las funciones de distancia de Meyer [3], considerando cuatro vecinos. El algoritmo de *Hill-Climbing* comienza detectando y marcando con diferentes etiquetas todos los mínimos de la imagen. El proceso continúa propagando las etiquetas hacia arriba, *subiendo la colina*, por el camino definido por la pendiente de cada pixel. El resultado de la segmentación es un conjunto de regiones (cuencas de agua), cada una identificada con la etiqueta del mínimo a partir del cual fue generada. Todos los píxeles pertenecen a una región, y las líneas de watershed quedan representadas por los límites entre regiones. La precisión de la segmentación será mayor cuanto mayor sea la conectividad usada [12].

Los CAs son modelos de computación compuestos por un conjunto de celdas de modo que cada celda está unida a sus vecinos adyacentes. Se caracterizan por un conjunto de estados y unas reglas de transición. Cada celda evoluciona de forma síncrona y en intervalos de tiempo discretos, dependiendo de su estado actual y el estado de sus vecinos. Si cada celda puede evolucionar de forma independiente un número indefinido de veces hablamos de autómatas asíncronos [13].

La principal ventaja del algoritmo de watershed basado en un CA es que la detección de los mínimos, el etiquetado y la propagación de las etiquetas por la pendiente se realizan de forma local y simultánea para los distintos píxeles [11]. Este algoritmo está basado en un autómata celular de tres estados, como se muestra en la figura 1, que computa el algoritmo

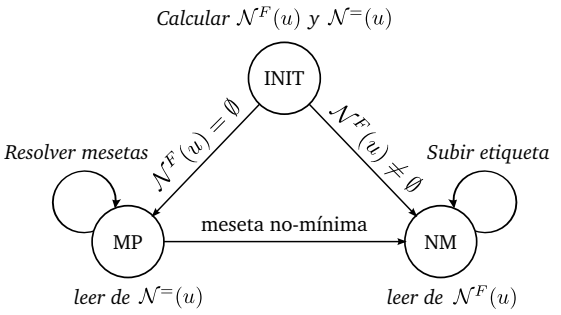

Fig. 1. Autómata de tres estados realizando el algoritmo *Hill-Climbing* [11].

Hill-Climbing en dos etapas: *inicialización y actualización*. Primero los píxeles son etiquetados secuencialmente por filas. En el estado inicial (INIT), todos los píxeles calculan a partir de sus cuatro vecinos sus conjuntos $\mathcal{N}^F(u)$ y $\mathcal{N}^=(u)$, que se corresponden con un elemento arbitrario de la pendiente (un elemento con menor valor de gris) y con el conjunto de los vecinos con el mismo valor de gris respectivamente. Si un pixel no tiene pendiente, es decir, $\mathcal{N}^F(u) = \emptyset$, su estado cambia a mínimo o meseta (MP) y su etiqueta se modifica de la siguiente forma:

$$l(u) = min\ (\ l(v)\ ) \quad \text{tal que} \quad v \in \mathcal{N}^=(u). \quad (4)$$

De lo contrario, $\mathcal{N}^F(u)$ contiene el pixel desde el que se propagará la etiqueta. En este último caso, el estado del pixel cambia a no-mínimo (NM) y su valor de gris y etiqueta se actualizan de la siguiente forma:

$$f(u) = f\left(\ \mathcal{N}^F(u)\ \right), \quad (5)$$

donde $f(u)$ es el par $(h(u), l(u))$, siendo $h(u)$ el valor de gris del pixel u y $l(u)$ su etiqueta.

Una vez que se han inicializado todos los píxeles comienza la etapa de actualización, que procesa los estados MP y NM de forma iterativa. Un pixel en estado MP espera por datos de cualquier vecino $v \in \mathcal{N}^=(u)$, y dependiendo del valor de gris $h(v)$, el pixel u se actualiza de una de las dos formas siguientes:

$$l(u) = min\ (\ l(u), l(v)\ ) \quad \text{si} \quad h(v) \geq h(u), \quad (6)$$

lo que equivale a extender la etiqueta más pequeña por cada meseta mínima. De lo contrario, el pixel u pertenece al borde inferior de una meseta, es decir, a una meseta no-mínima, y su estado cambia a NM de la siguiente forma:

$$\left.\begin{array}{r} \mathcal{N}^F(u) = v \\ f(u) = f(v) \\ \text{estado} = \text{NM} \end{array}\right\} \quad \text{si} \quad h(v) < h(u). \quad (7)$$

Por otro lado, un pixel en estado NM se actualiza de la forma:

$$f(u) = f(\mathcal{N}^F(u)), \quad (8)$$

esperando nuevos datos del pixel $\mathcal{N}^F(u)$, simulando que las etiquetas ascienden por la pendiente.

Este proceso iterativo continúa mientras se produzcan nuevas actualizaciones. El algoritmo es no

determinista y, por tanto, puede producir resultados de segmentación diferentes. Una demostración formal de la convergencia hacia una solución correcta de segmentación por watershed fue presentada en [11].

III. Arquitectura GPU

Las GPUs más recientes proporcionan capacidades de procesamiento masivamente paralelo basadas en una arquitectura *data-parallel*, mediante programas escritos en lenguajes de alto nivel como CUDA [14] para tarjetas de NVIDIA, o OpenCL [15] para plataformas heterogéneas. La arquitectura de CUDA está organizada en un conjunto de *multiprocesadores streaming* (SMs), cada uno de los cuales con varios cores llamados *procesadores streaming* (SPs). Estos cores pueden manejar miles de hilos en un modelo de programación SIMD.

En un programa escrito en CUDA, el código que se ejecuta en la GPU se conoce como *kernel*, y es ejecutado por miles de hilos agrupados en bloques. La memoria DRAM de la GPU está organizada en una memoria global (conocida genéricamente como memoria del dispositivo), una memoria de texturas de sólo lectura y una memoria constante. Todas ellas están disponibles para todos los hilos. También existe una *memoria compartida* de baja latencia accesible por parte de los hilos de un bloque pero con el tiempo de vida del bloque, por lo que no permite la compartición de datos entre ellos. Por último, cada hilo tiene su propia *memoria local* y *registros*. Estas características, disponibles en una tarjeta como la GTX295, se han mejorado en las tarjetas de consumo más actuales, conocidas como Fermi. Las más destacadas son un aumento en el número de cores y una jerarquía de memoria caché, compuesta por una caché L1 por SM y una caché unificada L2 de 768 KB para todos los SMs. Además, la caché L1 es configurable, ya que los 64 KB disponibles en el chip de memoria compartida, el usuario puede configurar 16 KB para la caché L1, quedando 48 KB para la memoria compartida, o viceversa. Una tarjeta de consumo de gama alta perteneciente a esta generación es la GTX580.

Respecto a la sincronización entre hilos, puede realizarse entre hilos dentro del mismo bloque pero no entre diferentes bloques, por lo cual no es posible compartir datos entre bloques y la comunicación debe realizarse a través de la memoria global. Esto supone un reto en cuanto a realizar computaciones eficientes.

IV. Implementación en GPU

En esta sección presentamos la implementación asíncrona en CUDA del algoritmo de watershed basado en un CA, que reúsa la información dentro de un bloque explotando de forma eficiente la memoria compartida y la memoria caché. Para esta implementación hemos elegido conectividad cuatro.

El algoritmo, tal como hemos analizado en la sección II, está dividido en dos etapas: una para inicializar y otra para actualizar el autómata. En la primera

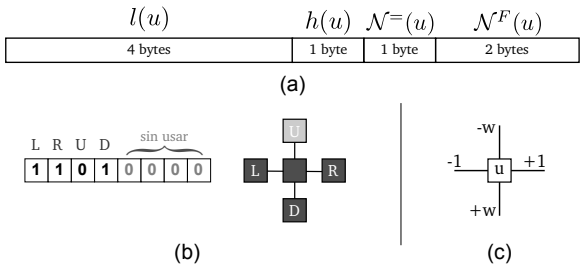

Fig. 2. Estructuras de datos para cada pixel. (a) Datos empaquetados en 64 bits, (b) estructura de la variable $\mathcal{N}^{=}(u)$ y (c) posibles valores de $\mathcal{N}^{F}(u)$.

etapa se calculan, para cada pixel, $\mathcal{N}^{F}(u)$, es decir, la dirección de la pendiente, y el conjunto de vecinos con el mismo valor de gris, $\mathcal{N}^{=}(u)$. En la segunda etapa un proceso iterativo propaga las etiquetas, segmentando la imagen en diferentes regiones, y con un número de sincronizaciones más bajo que en una implementación síncrona, que requeriría una sincronización por cada iteración de actualización. Cada una de estas dos etapas se realiza mediante un kernel.

Con el objetivo de incrementar la localidad en el acceso a los datos y reducir el número de transacciones, hemos empaquetado la información necesaria para cada pixel en 64 bits: 4 bytes para $l(u)$, 1 byte para $h(u)$, 1 byte para $\mathcal{N}^{=}(u)$ y 2 bytes for $\mathcal{N}^{F}(u)$. No es necesario almacenar el estado de cada pixel, ya que se puede deducir de la variable $\mathcal{N}^{F}(u)$. Si un pixel no tiene pendiente, su estado es MP, y de lo contrario, su estado es NM. La figura 2(a) muestra un ejemplo del empaquetado de datos para un pixel. El conjunto $\mathcal{N}^{=}(u)$ se ha comprimido en 1 byte usando 1 bit para cada vecino, como se muestra en la figura 2(b), donde "1" representa un vecino con el mismo valor de gris, y un "0" representa un vecino con un valor diferente. Los cuatro bits menos significativos se ignoran, aunque podrían usarse para conectar hasta ocho vecinos. $\mathcal{N}^{F}(u)$ almacena un desplazamiento relativo a la posición del pixel u dentro de la región que es procesada por cada bloque, teniendo en cuenta que las posiciones de cada pixel dentro de la imagen se almacenan por filas. Para regiones de $w \times w$ píxeles, los valores posibles son ± 1 y $\pm w$ como indica la figura 2(c).

El primer kernel inicializa el autómata según las ecuaciones (4) y (5). Los valores de la imagen se leen de memoria de texturas, y los datos de salida se empaquetan en 64 bits antes de copiarlos a memoria global. Con el fin de actualizar todos los píxeles de forma síncrona en la siguiente etapa, utilizamos un *buffer de lectura* y un *buffer de escritura* en memoria global. Al final de la inicialización, cada pixel ha cambiado su estado a MP o NM.

La segunda etapa consiste en un bucle en CPU que ejecuta el kernel de actualización. La etapa de actualización se ha adaptado para ejecutar en memoria compartida tantas actualizaciones como sean posibles con los datos disponibles para cada región de la imagen, antes de realizar una sincronización global

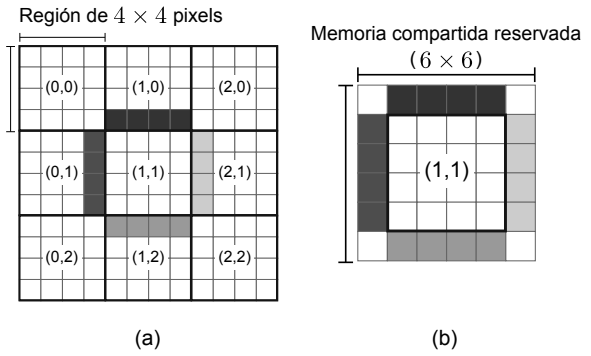

Fig. 3. Una imagen dividida en regiones de 4×4 píxeles (a) y la memoria compartida de 6×6 reservada para la región (1, 1) (b).

entre bloques. Esto se traduce en un proceso híbrido que incluye actualizaciones de tipo *inter-bloque* e *intra-bloque*.

Las actualizaciones inter-bloque se ejecutan en un bucle en CPU que se encarga de lanzar el kernel de CUDA. En cada iteración inter-bloque, se cargan los datos del buffer de entrada y se desempaquetan en memoria compartida. Y una vez desempaquetados se ejecuta el bucle intra-bloque sobre cada región de la imagen. En cada iteración de la actualización intra-bloque cada pixel se actualiza según las ecuaciones (6) y (7) si su estado es MP, y (8) si su estado es NM. Este proceso iterativo continúa mientras se produzcan nuevas actualizaciones en el bloque. Al final, el resultado se vuelve a empaquetar en 64 bits y se almacena en el buffer de salida. Los buffers de entrada y salida se intercambian en la CPU y una nueva iteración inter-bloque se realiza desde CPU si la última ejecución del kernel intra-bloque ha producido nuevas actualizaciones.

Con el fin de actualizar los píxeles en el borde de la imagen, la memoria compartida reservada para cada región tiene que ser extendida con un borde de tamaño uno. Por lo tanto, el borde de una región se solapa con las regiones adyacentes, tal como se puede observar en la figura 3. Los hilos en el borde del bloque hacen un trabajo extra cargando datos en el espacio de memoria ampliado.

V. Resultados

Hemos evaluado nuestros algoritmos en un PC Intel Core i7 con cuatro procesadores a 2,80 GHz, 8 GB de memoria RAM, 64 KB de memoria caché L1 y 256 KB de caché L2 por procesador, y 8192 KB de caché L3 compartida por todos los procesadores. Hemos utilizado dos GPUs diferentes, una GPU de arquitectura Fermi, una NVIDIA GeForce GTX580, con 16 SMs y 32 SPs (512 cores), configurada para usar 48 KB de memoria caché, y una tarjeta de la generación anterior, una NVIDIA GeForce GTX295, equipada con 30 SMs y 8 SPs (240 cores). Hemos elegido tarjetas de consumo para evaluar la eficiencia de los algoritmos en sistemas al alcance de muchos sectores profesionales y que no disponen de infraestructuras de más altas prestaciones para realizar análisis de imagen. El código ha sido compilado en linux, usando el compilador gcc-4.4.3 con soporte para OpenMP, en la implementación en CPU, y el *toolkit* de CUDA 4.0 para las implementaciones en GPU.

TABLA I
Tiempos de transferencia entre CPU y GPU.

512^2	1024^2	2048^2
0,0015s	0,0052s	0,0196s

Los resultados se expresan en tiempos de ejecución y aceleraciones. Las aceleraciones se calculan, salvo en el caso de comparativas entre GPUs, con respecto a una implementación paralela, usando OpenMP, del autómata celular síncrono. Este código está optimizado para usar un hilo en cada procesador de la CPU, con planificación estática para distribuir uniformemente la carga de trabajo entre los procesadores, de modo que cada uno computa una banda horizontal de la imagen. Incluye una barrera de sincronización en cada paso de la etapa de actualización. En este trabajo también analizamos cómo influye el tamaño de las mesetas en la eficiencia de los algoritmos y comparamos la implementación asíncrona en GPU con una implementación síncrona también en GPU. También analizamos cómo cambia el rendimiento al cambiar entre generaciones de tarjetas gráficas, al pasar de la GPU GTX295 a la GTX580.

Los tiempos de ejecución se miden como la suma de los tiempos de transferencia de datos entre CPU y GPU más los tiempos de computación de cada etapa. La transferencia de datos CPU–GPU tiene lugar al principio del algoritmo para enviar la imagen de entrada a memoria de texturas, y al final de la etapa de actualización para enviar el resultado a la CPU. Los kernels de las implementaciones en GPU se han configurado para trabajar en bloques de 16×16 hilos. Los tests se ejecutan sobre dos imágenes a diferentes resoluciones. La tabla I muestra los tiempos de transferencia en segundos para las diferentes resolu-

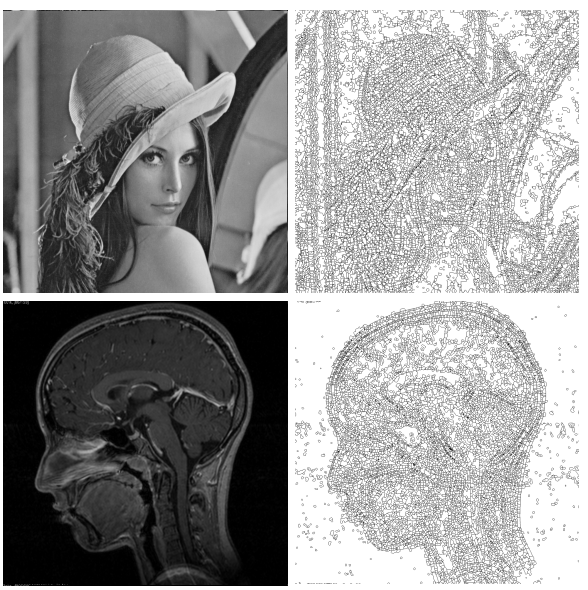

Fig. 4. Imágenes usadas en los test (izquierda) y resultado de watershed (derecha).

TABLA II

Número de regiones generadas por los algoritmos.

	512^2	1024^2	2048^2
Lena	24958	25139	28521
CT Scan	6221	7300	13381

ciones usadas en los experimentos. Las imágenes usadas han sido *Lena* y una tomografía computarizada de una cabeza humana a la que llamaremos *CT Scan*, como imágenes representativas de procesamiento de mesetas pequeñas y grandes respectivamente (figura 4 (izquierda)). Procesar mesetas grandes, o lo que es lo mismo, imágenes con amplias zonas de valor de gris constante, requiere propagar las etiquetas para esas zonas entre regiones de la imagen que son computadas por diferentes bloques de hilos, lo que requiere más tiempo de computación que procesar mesetas pequeñas, las cuales se pueden computar íntegramente mediante actualizaciones intra-bloque en una única región. Por tanto, las imágenes seleccionadas representan dos casos diferentes respecto al coste computacional de la implementación asíncrona de watershed.

La figura 4 (derecha) muestra el resultado de aplicar watershed a las imágenes de prueba. Para validar los resultados comparamos el número de regiones segmentadas por la implementación en OpenMP con las generadas por las implementaciones en GPU, obteniendo siempre el mismo resultado, que se muestra en la tabla II. La diferencia entre el número de regiones a diferentes resoluciones se debe al proceso de escalar la imagen. La imagen CT Scan presenta mesetas grandes y, por tanto, se generan menos regiones que para la imagen de Lena.

La tabla III muestra un resumen de los resultados obtenidos para la implementación asíncrona y, a efectos comparativos, también para una implementación síncrona en GPU, que sigue los mismos pasos de resolución pero donde solamente hay actualizaciones inter-bloque, ya que se requiere una sincronización global tras cada actualización sobre los píxeles de la imagen. Las aceleraciones de las implementaciones

TABLA III

Tiempos de ejecución y aceleraciones para las imágenes de Lena y CT Scan a distintas resoluciones.

Lena	512^2	1024^2	2048^2
T. OpenMP	$0,0350$s	$0,1990$s	$1,2452$s
T. GPU Sinc.	$0,0044$s	$0,0202$s	$0,0966$s
T. GPU Asinc.	$0,0035$s	$0,0150$s	$0,0619$s
Ac. GPU Sinc.	7,9x	9,8x	12,8x
Ac. GPU Asinc.	**10.0x**	**13.2x**	**20.1x**
CT Scan			
T. OpenMP	$0,4940$s	$2,8792$s	$15,0919$s
T. GPU Sinc.	$0,0298$s	$0,1431$s	$0,7013$s
T. GPU Asinc.	$0,0099$s	$0,0406$s	$0,1725$s
Ac. GPU Sinc.	16,5x	20,1x	21,5x
Ac. GPU Asinc.	**49.8x**	**70.9x**	**87.4x**

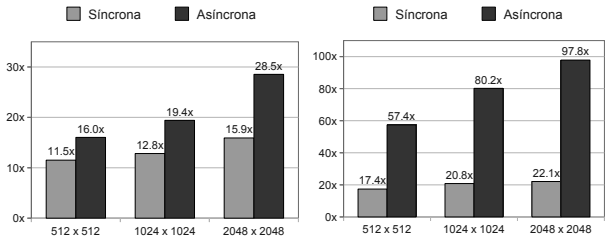

Fig. 5. Aceleración de las implementaciones en GPU usando la imagen de Lena (izquierda) y CT Scan (derecha), sin incluir tiempos de transferencia.

en GPU respecto de la implementación en CPU se representan en la figura 5, considerando únicamente tiempos de computación. Los mejores resultados obtenidos son 28,5x y 97,8x respectivamente para Lena y CT Scan, procesando 2048×2048 píxeles mediante la implementación asíncrona, que son siempre mejores que para la versión síncrona; aproximadamente el doble mejores para la imagen de Lena, y cinco veces mejores para la CT Scan. El motivo es que la implementación asíncrona explota mejor los recursos disponibles en la GPU, como la memoria compartida, reutilizando los datos cargados de memoria global, y reduce el número de sincronizaciones entre bloques, gracias al esquema de actualización inter e intra-bloque.

La mejora de la implementación asíncrona es mayor para la imagen CT Scan, que representa imágenes con mesetas grandes. Como se muestra en la tabla III, procesar esta imagen lleva más tiempo, 0,1725s frente a sólo 0,0619s para la imagen de Lena, considerando imágenes a una resolución de 2048×2048. Sin embargo, se obtienen mejores aceleraciones para la imagen CT Scan. La razón de esta mejora es que para la ejecución asíncrona en GPU, la actualización intra-bloque permite que las etiquetas se propaguen más rápido entre regiones, lo que supone menos sincronizaciones entre bloques. Por ejemplo, si un región está contenida dentro de una meseta, las etiquetas tienen que propagarse de lado a lado en dicha región. La implementación asíncrona sólo necesita 1 actualización inter-bloque y w actualizaciones intra-bloque, tomando w como el tamaño del bloque, mientras que la implementación síncrona en CPU necesita en el mismo caso w actualizaciones inter-bloque, con la consecuente penalización por la

TABLA IV

Tiempos de computación y aceleraciones para la implementación asíncrona en dos tarjetas gráficas diferentes.

Lena	512^2	1024^2	2048^2
GTX295	$0,0044$s	$0,0236$s	0,1234s
GTX580	$0,0022$s	$0,0102$s	$0,0436$s
Aceleración.	2,0x	2,3x	2,8x
CT Scan			
GTX295	$0,0409$s	0,1654s	0,6513s
GTX580	$0,0086$s	$0,0359$s	$0,1542$s
Aceleración	4,7x	4,6x	4,2x

barrera de sincronización implícita en cada paso.

La tabla IV muestra los resultados de la implementación asíncrona en una GTX295 y una GTX580. La mejora en el tiempo de computación obtenida para la tarjeta GTX580 respecto de la GTX295 es de media 2,3x para la imagen de Lena y 4,5x para la imagen CT Scan. Estas mejoras se deben principalmente a que la GTX580 tiene más del doble de cores que la GTX295, así como el doble de registros por SM. Además, la arquitectura Fermi de la GTX580 cuenta con memoria caché [16]. Respecto al número máximo de registros, en la tarjeta GTX295 tenemos como máximo 16384 registros disponibles por cada SM, lo que permite que para nuestra implementación asícnrona tengamos como máximo cuatro bloques activos por SM, mientras que para la GTX580 el total disponible de registros es 32768 que permite tener hasta seis bloques activos por SM. El hecho de tener más bloques activos nos permite propagar las etiquetas más rápido entre regiones. Por ello la mejora es mayor al procesar la imagen CT Scan, que tiene mesetas más grandes que la imagen de Lena.

En los últimos años se han publicado diferentes algoritmos de watershed en CUDA [7, 8]. En [7], los experimentos se llevan a cabo en volúmenes de datos, mediante un nuevo algoritmo basado en una función cromática para establecer el orden de procesamiento de los vóxeles. Los autores consiguen una aceleración de 7x en una NVIDIA GTX295 cuando se comparan con una versión secuencial en CPU. El algoritmo presentado en [8] se compara en una GTX295 con un autómata celular en GPU [6], aunque éste no está implementado en CUDA. Dado que los experimentos en [8] se ejecutan en una tarjeta más antigua, los hemos ejecutado en nuestra GTX580 consiguiendo aceleraciones similares a las obtenidas con nuestra implementación asíncrona de watershed.

VI. Conclusiones

En este trabajo presentamos una implementación asíncrona para calcular la transformada watershed en GPU que explota el paradigma de computación de CUDA y mostramos la influencia del tamaño de las mesetas en el tiempo de computación y la aceleración obtenida. El algoritmo de watershed seleccionado está basado en un autómata celular de tres estados que puede ejecutarse de forma asíncrona. Mediante la implementación asíncrona se explota al máximo la capacidad computacional de la tarjeta. La clave de la implementación asíncrona está en un esquema de actualización de estados híbrido que permite que cada bloque de hilos realice todas las actualizaciones posibles dentro de una región de la imagen antes de que una sincronización global sea necesaria.

Los resultados fueron ejecutados en una NVIDIA GTX580 y una GTX295 y comparados con una implementación en CPU usando OpenMP. La mejor aceleración obtenida ha sido 97,8x al procesar una imagen con mesetas grandes a una resolución de 2048×2048 píxeles. Los resultados obtenidos son competitivos si se comparan con los últimos trabajos publicados en este campo, con la ventaja de que el método basado en CAs es sencillo y permite incorporar fácilmente etapas de pre y postprocesado de las imágenes manteniendo la estructura del algoritmo.

Agradecimientos

Este trabajo fue financiado en parte por el Ministerio de Ciencia e Innovación, Gobierno de España, cofundador de los fondos FEDER de la Unión Europea, en virtud del contrato TIN 2010-17541, y por la Xunta de Galicia, Programa para la Consolidación de Grupos de Investigación Competitiva ref. 2010/28. Pablo Quesada y Julián Lamas agradecem el apoyo económico del Ministerio de Ciencia e Innovación, Gobierno de España, bajo sendas becas FPI-MICINN.

Referencias

[1] Jos B. T. M. Roerdink and Arnold Meijster, "The watershed transform: definitions, algorithms and parallelization strategies," *Fundam. Inf.*, vol. 41, pp. 187–228, January 2000.

[2] Luc Vincent and Pierre Soille, "Watersheds in digital spaces: An efficient algorithm based on immersion simulations," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 13, pp. 583–598, June 1991.

[3] F. Meyer, "Topographic distance and watershed lines," *Mathematical Morphology and its Applications to Signal Processing*, vol. 38, no. 1, pp. 113–125, July 1994.

[4] A.N. Moga, B. Cramariuc, and M. Gabbouj, "An efficient watershed segmentation algorithm suitable for parallel implementation," in *Proc. of the 1995 Int. Conf. on Image Processing*, Oct. 1995, vol. 2, pp. 101–104 vol.2.

[5] D. Noguet, "A massively parallel implementation of the watershed based on cellular automata," in *Proc. of the IEEE Int. Conf. on Application-Specific Systems, Architectures and Processors*, July 1997, pp. 42–52.

[6] C. Kauffmann and N. Piche, "A cellular automaton for ultra-fast watershed transform on gpu," in *ICPR 2008, 19th Int. Conf. on Pattern Recognition*, Dec. 2008, pp. 1–4.

[7] Björn Wagner, Paul Müller, and Gundolf Haase, "A parallel watershed-transformation algorithm for the gpu," in *Workshop on Applications of Discrete Geometry and Mathematical Morphology*, Aug. 2010, pp. 111–115.

[8] André Körbes, Giovani Vitor, Roberto de Alencar Lotufo, and Janito Ferreira, "Advances on watershed processing on gpu architecture," in *Mathematical Morphology and Its Applications to Image and Signal Processing*, Pierre Soille, Martino Pesaresi, and Georgios Ouzounis, Eds., vol. 6671 of *Lecture Notes in Computer Science*, pp. 260–271. Springer Berlin / Heidelberg, 2011.

[9] S. Rybacki, J. Himmelspach, and A.M. Uhrmacher, "Experiments with single core, multi-core, and gpu based computation of cellular automata," in *Proc. of the 2009 First Int. Conf. on Advances in System Simulation*. sept. 2009, pp. 62–67, IEEE Computer Society.

[10] C. Kauffmann and N. Piche, "A cellular automaton framework for image processing on gpu," in *Pattern Recognition*, Yin Peng-Yeng, Ed., pp. 353–376. inTech, Oct. 2009.

[11] B. Galilee, F. Mamalet, M. Renaudin, and P.-Y. Coulon, "Parallel asynchronous watershed algorithm-architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 1, pp. 44–56, Jan. 2007.

[12] S. Beucher, "Watershed, hierarchical segmentation and waterfall algorithm," *Mathematical morphology and its applications to image processing*, vol. 2, pp. 69–76, 1994.

[13] Chrystopher L. Nehaniv, "Evolution in asynchronous cellular automata," in *Proc. of the eighth Int. Conf. on Artificial life*. 2003, pp. 65–73, MIT Press.

[14] Nvidia Corporation, *NVIDIA CUDA C Programming Guide*, Santa Clara, 2011.

[15] Khronos OpenCL Working Group, *The OpenCL Specification*, 2011.

[16] Nvidia Corporation, "Whitepaper NVIDIA's Next Generation CUDA Compute Architecture: Fermi," 2012.