Proyección del método de segmentación del conjunto de nivel en GPU

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

Resumen— **El método de conjunto de nivel es una herramienta utilizada en la segmentación de imágenes y volúmenes. En este trabajo presentamos dos propuestas en CUDA del método de segmentación rápida FTC basado en conjuntos de nivel para su aplicación a imágenes 3D y que modifican la estructura del algoritmo original de modo que se reducen computaciones innecesarias dentro de cada región de la imagen y se evita computar regiones que se pueden considerar inactivas. El mantenimiento de una calidad de segmentación similar a la del algoritmo original demuestra la viabilidad de las propuestas, las cuales obtienen para un volumen MRI del cerebro valores de aceleración cercanos a 4x.**

Palabras clave— **Conjuntos de nivel, segmentación, GPU, CUDA.**

I. Introducción

El método del conjunto de nivel *(level-set method)* [1] es una técnica numérica utilizada frecuentemente para realizar tareas de segmentación dentro de una imagen o un volumen. El método se basa en la propagación de curvas o superficies a partir de factores intrínsecos (por ejemplo, la curvatura de las regiones segmentadas) y extrínsecos (por ejemplo, la intensidad o la textura) de la imagen [2]. La segmentación basada en conjuntos de nivel tiene multitud de aplicaciones en el campo de visualización de imágenes médicas [3].

Los métodos de conjunto de nivel suelen clasificarse en métodos de banda estrecha *(narrow band)* o de campo disperso *(sparse field)*. Los algoritmos de banda estrecha [4] actualizan el dominio computacional en una banda de unos pocos vóxeles alrededor del área de interés. Las soluciones de campo disperso [5] mantienen una lista de elementos activos en el dominio que se actualiza en cada iteración del algoritmo.

La primera implementación documentada de un método de conjunto de nivel en GPU fue propuesta por Rumpf y Strzodka en 2001 [6]. Esta implementación se ejecutaba completamente en el *pipeline* gráfico. Varios años más tarde, combinando ideas de los métodos de banda estrecha y de campo disperso, se desarrollaría la primera solución que divide el dominio en regiones bidimensionales que son cargadas en la memoria de la GPU según requieren procesamiento [7]. A partir de este trabajo, se presentó una de las primeras implementaciones del conjunto de nivel en CUDA [2], que se caracterizaba por mantener una lista de elementos activos en el dominio. Al mismo tiempo, se desarrolló una solución de banda estrecha en CUDA [8] basada en dividir el dominio en regiones activas que se actualiza en paralelo (aunque requería el uso de operaciones atómicas en GPU, lo que degradaba el rendimiento). Más recientemente, se ha publicado en [9] una implementación en GPU del método de conjunto de nivel *Sorted Tile List* (STL).

En este trabajo presentamos dos propuestas de implementación en CUDA del método de segmentación rápida basado en conjunto de nivel denominado *Fast Two Cycle* (FTC) [10]. Este algoritmo de banda estrecha se caracteriza por utilizar listas enlazadas para almacenar la banda de elementos activos, y utilizar operaciones con enteros para aproximar la resolución de las ecuaciones diferenciales presentes en la definición original del método de conjunto de nivel [1]. Nuestras propuestas adaptan el FTC a las características de la GPU dividiendo el dominio computacional en regiones que pueden ser procesadas en paralelo, modificando la distribución de iteraciones para evitar computaciones innecesarias sobre las regiones y, en el caso de la segunda propuesta, reduciendo el número de regiones sobre las que se realiza procesamiento. En [11] se describe una implementación en GPU de este método, pero está diseñada para imágenes 2D y sólo realiza un paso de uno de los ciclos del algoritmo. Nuestra solución ha sido diseñada para volúmenes 3D e implementa el método FTC en su totalidad facilitando así la comparación con la bibliografía.

Este trabajo se organiza de la siguiente manera: la sección II describe la arquitectura de la GPU y CUDA; la sección III introduce las bases fundamentales de los métodos de conjunto de nivel y, en particular, del método FTC; las propuestas para GPU están descritas en la sección IV. Las secciones V y VI se dedican a la discusión de resultados y principales aportaciones.

II. Arquitectura GPU

Una GPU programable consta de múltiples procesadores multinúcleo capaces de ejecutar cientos de hilos en paralelo. En esta sección presentamos brevemente la arquitectura Fermi, disponible en la tarjeta NVIDIA GeForce GTX 580 que hemos utilizado. Puede encontrarse un análisis más completo en [12].

La arquitectura CUDA de NVIDIA está organizada en un conjunto de *multiprocesadores streaming* (SMs), cada uno con varios núcleos o *procesadores streaming* (SPs) [13]. El modelo de programación está orientado hacia un grano fino de paralelismo [14]. Un programador lanza cientos de hilos que se

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

agrupan en bloques independientes, organizados en una malla. Cuando se ejecuta un *kernel* (código ejecutado en GPU), los bloques de hilos son asignados a los SMs. Un multiprocesador puede ejecutar más de un bloque de hilos a la vez si dispone de recursos suficientes.

La jerarquía de memoria está organizada en una *memoria global*, una *memoria de constantes* de solo lectura y una *memoria de texturas*. Estas memorias están accesibles para todos los hilos. Además, existe un espacio de *memoria compartida* privado para cada bloque de hilos con una velocidad de lectura/escritura unas 100 veces mayor que la memoria global, aun disponiendo esta de una jerarquía con dos niveles de memoria caché. Por ello, priorizar el acceso a esta memoria es lo más recomendable [15].

La comunicación entre hilos puede realizarse a través de *barreras de sincronización* o del espacio de memoria compartida. Como cada bloque de hilos es independiente de los demás, no es posible compartir datos entre hilos de diferentes bloques a través de la memoria compartida.

III. Método del conjunto de nivel

En esta sección introducimos brevemente la base matemática de los métodos de segmentación de conjunto de nivel, y describimos el método de segmentación rápida (FTC) [10], diseñado para realizar segmentaciones del conjunto de nivel en tiempo real con una calidad similar a otras soluciones basadas en la resolución de ecuaciones diferenciales parciales.

A. Fundamentos del conjunto de nivel

Los métodos de conjunto de nivel se basan en calcular la propagación de una curva o superficie. Sea $\phi(\mathbf{x}, t) : \mathbb{R}^n \to \mathbb{R}$, con $\mathbf{x} \in \mathbb{R}^n$, una función continua n-dimensional de Lipschitz. El conjunto de nivel k de ϕ define implícitamente Γ, una hipersuperficie $(n-1)$-dimensional, también denominada frente, curva o interfaz:

$$\Gamma(t) = \{\mathbf{x} \in \mathbb{R}^n \mid \phi(\mathbf{x}, t) = k\}. \tag{1}$$

Sea $\Omega \subset \mathbb{R}$ la región encerrada por Γ. Los métodos del conjunto de nivel calculan y analizan el movimiento de Γ y, en consecuencia, la deformación de Ω. La función ϕ suele definirse como una función de distancia respecto a Γ cuya evolución está determinada por la siguiente ecuación diferencial parcial de primer orden [16]:

$$\frac{d\phi(\mathbf{x}(t), t)}{dt} + \vec{F}|\nabla\phi| = 0, \tag{2}$$

donde $\mathbf{x}(t)$ es el conjunto de puntos en el instante t para los cuales ϕ vale 0, y $\vec{F}$ es la función de velocidad en la dirección de la normal. Así, la función de actualización del conjunto de nivel queda definida como:

$$\phi(\mathbf{x}, t + \Delta t) = \phi(\mathbf{x}, t) + \Delta t F|\nabla\phi(\mathbf{x}, t)|. \tag{3}$$

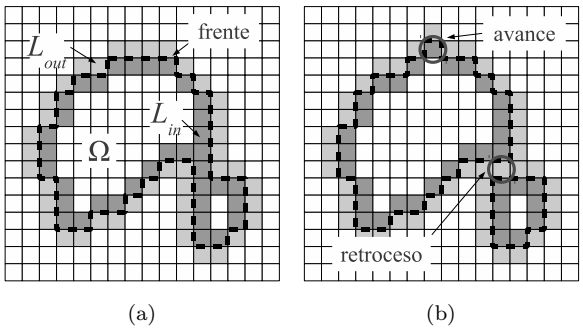

Fig. 1. Representación del conjunto de nivel en un espacio 2D (a) y ejemplo de evolución del frente según el algoritmo FTC (b).

B. Método de segmentación rápida (FTC)

El método de segmentación del conjunto de nivel de dos ciclos (en adelante, *Fast Two Cycle* o FTC) [10] analiza el movimiento del frente en un espacio discreto y uniforme. Dos listas de puntos vecinos determinan la situación del frente Γ:

$$L_{in} = \{\mathbf{x} \mid \mathbf{x} \in \Omega \text{ y } \exists \mathbf{y} \in \mathcal{N}(\mathbf{x}) \text{ tal que } \mathbf{y} \notin \Omega\}$$
$$L_{out} = \{\mathbf{x} \mid \mathbf{x} \notin \Omega \text{ y } \exists \mathbf{y} \in \mathcal{N}(\mathbf{x}) \text{ tal que } \mathbf{y} \in \Omega\},$$

donde $\mathcal{N}$ es el conjunto de puntos vecinos de $\mathbf{x}$. En nuestro caso, el vecindario está formado por los 6 puntos inmediatamente adyacentes a $\mathbf{x}$ en un espacio tridimensional.

La figura 1(a) muestra el frente como una línea discontinua localizada entre los conjuntos de puntos L_{in} y L_{out}, en gris oscuro y gris claro, respectivamente, para un ejemplo de espacio en dos dimensiones.

El método FTC define ϕ como una aproximación de la función de distancia a Γ. Así, ϕ es una función entera que puede tomar los valores -3, -1, $+1$ o $+3$ dependiendo de si el punto pertenece al interior de Ω, a L_{in}, a L_{out} o al exterior de Ω, respectivamente.

El frente avanza o retrocede en este espacio discreto añadiendo o quitando elementos a L_{in} y L_{out}, y actualizando los valores de ϕ de los correspondientes vecinos para mantener una representación continua del frente. La figura 1(b) muestra el resultado de avance y retroceso del frente en dos posiciones diferentes del ejemplo presentado en la figura 1(a).

El algoritmo, tal y como se puede ver en el pseudocódigo de la figura 2, está dividido en dos partes diferenciadas, ciclo uno y ciclo dos. N_a y N_s son, respectivamente, el número de iteraciones que se ejecutan el ciclo uno y el ciclo dos por cada iteración completa del algoritmo.

El primer ciclo del algoritmo hace avanzar o retroceder el frente basándose en los datos de entrada, esto es, la imagen o el volumen sobre el que se va a realizar una segmentación. El ciclo dos aplica un suavizado gaussiano al frente con el objetivo de hacerlo evolucionar en base a su curvatura. En ambos ciclos el frente evoluciona siguiendo los mismos pasos, resumidos en la ecuación (3), pero el ciclo uno utiliza una función de velocidad derivada de los datos de la imagen y de las propiedades geométricas del frente, F_d,

```
para i = 1 → N_a:
    ejecuta el ciclo uno
    comprueba la condición de parada
    si condición de parada satisfecha entonces:
        fin del bucle
para i = 1 → N_s:
    ejecuta el ciclo dos
si condición de parada no satisfecha entonces:
    repite de nuevo desde el ciclo uno
```

Fig. 2. Pseudocódigo del método FTC.

mientras que el ciclo dos utiliza una función de suavizado que es proporcional a la curvatura media, F_{int}. Así, el método FTC aproxima la función de velocidad de la ecuación (3) como $F = F_d + F_{int}$. Dependiendo del signo que tome la función de velocidad para un elemento dado se decidirá si el frente debe avanzar o retroceder: L_{out} "avanzará" si el signo de la función de velocidad es positivo, y L_{in} "retrocederá" si el signo de la función de velocidad es negativo.

El proceso de segmentación consiste así en la ejecución iterativa de los ciclos uno y dos, y solo se detiene cuando se ha verificado la condición de parada:

$$\forall \mathbf{x} \in L_{out}, F_d(\mathbf{x}) \leq 0 \quad \wedge \quad \forall \mathbf{x} \in L_{in}, F_d(\mathbf{x}) \geq 0,$$

o se ha alcanzado un número máximo de iteraciones.

IV. Propuestas para ejecución en GPU

En esta sección presentamos nuestras propuestas de implementación en CUDA del método FTC en la tarjeta gráfica. La primera propuesta modifica el número de iteraciones de los ciclos uno y dos para adaptar el algoritmo a las características particulares de la GPU. La segunda realiza además una selección más estricta de las regiones activas, consiguiendo así reducir el tiempo de computación sin perder calidad.

A. Propuesta 1

El algoritmo comienza cargando el volumen de entrada y los datos iniciales de la función de distancia ϕ en la memoria global. Ambos se dividen en regiones tridimensionales que pueden ser procesadas en paralelo y de forma independiente. Así, el trabajo se reparte entre diferentes bloques de $8^3 = 512$ hilos y se asigna una región por bloque (cada vóxel es asignado a un hilo de cada bloque).

La figura 3 muestra el pseudocódigo de la implementación en GPU. Antes de iniciar la segmentación, es necesaria una identificación inicial de qué regiones son activas. Una región se considera activa si es atravesada por el frente. El frente está formado por todos los elementos de L_{in} o L_{out}, donde ϕ toma el valor -1 o $+1$, respectivamente. Las regiones restantes no necesitan ningún procesamiento, por lo que solo se lanzarán tantos bloques de hilos como regiones activas existan. A medida que el frente va evolucionando, el número de regiones activas cambia.

Se han implementado tres kernels que ejecutan respectivamente el ciclo uno, el ciclo dos y la comprobación de la condición de parada. La CPU controla

```
identifica las regiones activas iniciales
para i = 1 → N_b:
    lanza kernel GPU del ciclo uno (N_a iters.)
    identifica las regiones activas
    comprueba la de cond. de parada en GPU
    si condición de parada satisfecha entonces:
        fin del bucle
lanza el kernel GPU del ciclo dos (N_s iters.)
identifica las regiones activas
si condición de parada no satisfecha entonces:
    repite de nuevo desde el ciclo uno
```

Fig. 3. Pseudocódigo de la implementación del algoritmo FTC en GPU.

la ejecución del algoritmo, pero a diferencia del código presentado en la figura 2, la CPU ya no itera por cada uno de los pasos internos del proceso de segmentación, sino que son los kernels los que ejecutan las N_a y N_s iteraciones requeridas por los ciclos uno y dos, respectivamente.

Con el objetivo de aprovechar al máximo la capacidad computacional de la tarjeta y, por tanto, evitar computaciones que no producen evolución del frente, se ha modificado la estructura del algoritmo para su ejecución en GPU. Se ha añadido un nuevo parámetro, N_b, que determina el número de veces que se ejecuta el ciclo uno por cada iteración completa del algoritmo, tal como se puede apreciar en el pseudocódigo. El kernel del ciclo uno ejecuta N_a iteraciones dentro de cada región activa para evolucionar el frente. Debido al pequeño tamaño de las regiones asignadas a cada bloque de hilos, a partir de un número de iteraciones igual a las dimensiones de cada región (8 en nuestro caso) las iteraciones no tienen efecto, ya que el frente habrá llegado al borde de dicha región. Por ello hemos escogido valores de N_a menores que el tamaño de la región. Por otro lado, el ciclo dos de suavizado sólo es necesario ejecutarlo cuando ha habido cambios apreciables en el frente, por lo que es conveniente recalcular estas N_a iteraciones del ciclo uno N_b veces antes de ejecutar el ciclo dos. En este segundo ciclo no se da la misma situación ya que N_s suele ser lo suficientemente pequeño en relación con el tamaño de las regiones como para que el frente pueda evolucionar durante todas las iteraciones.

El proceso de ejecución de los kernels de los ciclos uno y dos es muy similar, con la salvedad del cálculo de la función de velocidad. En ambos casos, cada hilo comienza realizando la carga de datos identificando las coordenadas de la región del volumen y de ϕ asignados a su bloque de hilos. A continuación, los elementos de ϕ pertenecientes a la región se almacenan en la memoria compartida del bloque, junto con un borde exterior. Este borde es necesario porque las computaciones se realizan enteramente en memoria compartida y requieren para cada elemento la presencia de sus vecinos. El grosor del borde depende del ciclo que se esté ejecutando: en el ciclo uno, un grosor de un elemento es suficiente; en el ciclo dos, depende del tamaño del filtro de suavizado, N_g, que

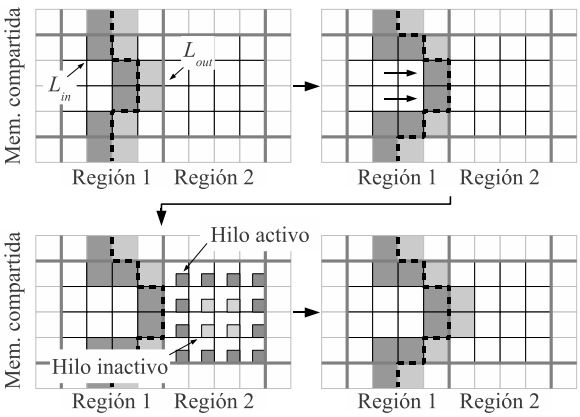

Fig. 4. Corrección de incoherencias en memoria compartida.

es definido por el usuario.

Como parte del proceso de carga de datos en memoria compartida es necesario corregir las incoherencias que se detecten. Dividir el trabajo en regiones donde el frente evoluciona de forma independiente tiene el problema de generar incoherencias, es decir, elementos con valores de ϕ que no tienen continuidad con los obtenidos en las regiones vecinas. La figura 4 muestra un ejemplo de corrección de incoherencias en memoria compartida entre dos regiones adyacentes en un espacio 2D. El frente evoluciona moviéndose a la derecha, situándose a mitad de las regiones 1 y 2, aunque solo la región 1 fue actualizada correctamente. En la siguiente iteración del algoritmo la región 2 se activa, y tras ser cargada en memoria compartida, los hilos que tienen asignados elementos en los límites de la región (identificados en la figura en color gris oscuro como hilos activos) verifican la existencia de incoherencias con las regiones vecinas. En este caso, se detectan incoherencias con la región 1, por lo que se corrigen en el último paso mostrado en la figura.

Continuando con el pseudocódigo de la figura 3, al final de la ejecución de los kernels del ciclo uno y del ciclo dos, después de las líneas 4 y 9, la lista de regiones activas debe ser actualizada para reflejar los cambios en el conjunto de nivel. Cada bloque de hilos comprueba si su región actual y las regiones adyacentes pueden ser consideradas activas.

Una vez terminada la ejecución del ciclo uno, se lanza el kernel de comprobación de la condición de parada. Este kernel lanza un hilo por cada región activa, y evalúa la condición de parada en cada elemento de las regiones activas. Si existe al menos un elemento en la región activa que no satisface la condición de parada, entonces la segmentación aún está incompleta (suponiendo que no se haya alcanzado todavía el número máximo de iteraciones).

B. Propuesta 2

La propuesta que describimos en este apartado modifica respecto a la anterior el criterio para seleccionar las regiones activas que se utilizan en la líneas 4 y 9 del pseudocódigo mostrado en la figura 3. El objetivo es reducir el número de regiones activas evitando que se lancen bloques de hilos que computarían regiones sobre los que no se produciría evolución y, por tanto, aprovechando mejor los recursos computacionales de la tarjeta.

Durante el proceso de segmentación del algoritmo FTC, el frente crece en tamaño y complejidad, por lo que el número de regiones activas aumenta. Sin embargo, no todo el frente está evolucionando en todo momento, ya que hay partes que se estabilizan antes de que la segmentación termine. En nuestra anterior propuesta para GPU, las regiones pertenecientes al frente estabilizado se siguen considerando activas, aunque el frente en ellas no evolucione. En la actual propuesta una región solamente se considera activa si no se ha estabilizado, es decir, si es atravesada por el frente y además todos sus elementos verifican la condición de parada. Sobre estas nuevas regiones activas se ejecutan los ciclos uno y dos.

Al final del algoritmo se fuerza la ejecución de una iteración adicional que realiza un suavizado sobre todas las regiones que se consideran activas según el criterio original, es decir, que son atravesadas por el frente. De este modo, el suavizado se aplica también sobre las regiones que contienen el frente, aunque no hayan evolucionado durante las últimas iteraciones. El coste computacional de este último proceso es mayor que el de cada una de las iteraciones anteriores. Sin embargo, como veremos en la sección de resultados, el rendimiento del algoritmo completo aumenta y se mantiene la calidad de la segmentación.

V. Resultados

Hemos evaluado nuestra implementación en una NVIDIA GeForce GTX 580 con 512 cores agrupados en 16 SMs de 32 SPs cada uno, con una frecuencia de reloj de 1,544 GHz y 1,5 GB de memoria global. Cada SM tiene 64 kB de RAM con un particionado configurable de memoria compartida y caché L1: 16 kB de memoria compartida y 48 kB de caché L1, o viceversa. En nuestras pruebas hemos seleccionado la última de estas configuraciones. Además, una caché L2 unificada de 768 kB está disponible para todos los SMs [13]. La versión original del método FTC fue evaluada en una CPU Intel Core i7 con cuatro cores a 2,8 GHz y 8 GB de RAM [17].

Las pruebas de segmentación se realizaron sobre una de las imágenes MRI 3D de cabeza generadas en el simulador BrainWeb Simulated Brain Database [18] (modelo sin lesiones cerebrales, modalidad T1, grosor de lámina 1 mm, sin ruido y 20 % de heterogeneidad de intensidades). El volumen generado tiene un tamaño de $181 \times 217 \times 181$ bytes. La figura 5 muestra una selección de cortes de dicho volumen.

Hemos seleccionado una función de velocidad idéntica a la usada en [10], calculada a partir de los valores de interés de la imagen, con el objetivo de facilitar la comparación con los resultados allí publicados. En nuestras pruebas hemos segmentado las materias gris y blanca del cerebro, seleccionando segmentar valores de intensidad en el rango $(98, 209)$, con una semilla inicial de radio de 40 puntos situada en la posición $(90, 108, 90)$.

TABLA I

Parámetros de segmentación de nuestras pruebas y resultados obtenidos sobre un volumen de BrainWeb de tamaño $181 \times 217 \times 181$ segmentando materias gris y blanca.

Algoritmo	N_a	N_b	N_s	N_g	σ	Tiempo	Aceleración	Coef. Dice
CPU	30	-	3	3	2	2,346 s	1,0x	0,95
GPU Implem. 1	6	5	3	3	2	0,961 s	2,4x	0,96
GPU Implem. 2	6	5	3	3	2	0,622 s	3,8x	0,96

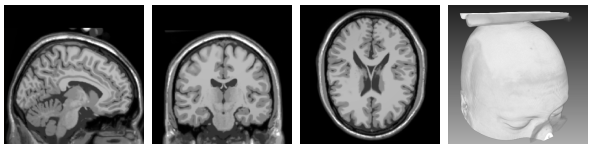

Fig. 5. Cortes sagital, coronal y transversal y renderizado de la imagen de BrainWeb.

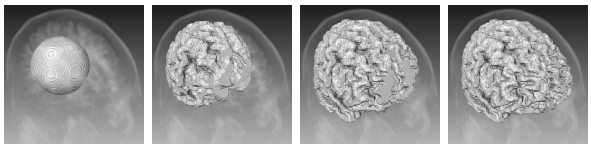

Fig. 6. Fases de la segmentación de la imagen de BrainWeb.

La figura 6 muestra diferentes etapas del progreso de segmentación en GPU de la imagen de BrainWeb. La segmentación fue implementada en Amira [19] y todas las capturas de pantalla fueron tomadas dentro de dicha aplicación. Inicialmente, el volumen de segmentación es una semilla esférica cuya posición y tamaño son configurados por el usuario. Según progresa la segmentación, la semilla inicial crece hasta que la segmentación se detiene.

La tabla I contiene los valores de los parámetros de segmentación usados en las pruebas para las implementaciones CPU y GPU del método FTC, junto con los resultados obtenidos. N_a y N_b hacen referencia al número de iteraciones del ciclo uno, y N_s hace referencia al número de iteraciones del ciclo dos, en cada iteración del algoritmo (véase Sección IV). N_g y σ son, respectivamente, el tamaño y la varianza del filtro gaussiano de suavizado. Hemos escogido valores similares a los usados en [10].

En cuanto a los resultados de rendimiento, en la tabla I se muestran el tiempo consumido en cada una de nuestras pruebas, la aceleración de las implementaciones en GPU respecto a la de CPU, y como medida de calidad, el valor de coeficiente Dice [20] respecto a la clasificación de referencia disponible en BrainWeb. El coeficiente Dice compara dos volúmenes vóxel a vóxel y ofrece una medida de semejanza que puede tomar valores entre 0 y 1, donde 1 indicaría que ambos volúmenes son completamente iguales.

El rendimiento de la segmentación depende en gran medida del tamaño y las características del volumen de datos que va a ser procesado, pero nuestras pruebas demuestran que en cualquier caso la versión en GPU presenta una mejora importante en rendimiento, llegando a obtener un valor de aceleración de 3,8x. En cuanto a la calidad, el valor del coeficiente Dice para las implementaciones en GPU es similar al del algoritmo original en CPU, demostrando que la reducción de regiones activas de la *propuesta 2* en GPU no disminuye la calidad de la segmentación obtenida.

Una de las primeras implementaciones de segmentación mediante el método de conjunto de nivel realizadas en CUDA está descrita en [2]. A partir del trabajo presentado en [7], implementa un algoritmo de banda estrecha haciendo uso de condiciones basadas en derivadas temporales del conjunto de nivel para reducir al mínimo el dominio computacional activo. Un 77 % del tiempo de ejecución se consume en mantener y actualizar este dominio, y la implementación tarda 7 segundos en segmentar un volumen de tamaño 256^3 obtenida de BrainWeb utilizando una tarjeta NVIDIA GeForce GTX 280. Nuestra solución divide el dominio activo en bloques de 8^3 elementos, de forma que el tiempo requerido en mantener el dominio despreciable, requiriendo 0,8 segundos para segmentar un volumen de características similares y tamaño $256 \times 256 \times 181$ en una tarjeta NVIDIA GeForce GTX 580. Dado que la tarjeta GTX 280 tiene aproximadamente la mitad de núcleos que la GTX 580, podemos estimar que su potencia es aproximadamente la mitad [21], por lo que nuestra *propuesta 2* es aproximadamente 5 veces más rápida.

En [9] se describe una implementación basada en CUDA de un algoritmo disperso de segmentación para el cálculo del método de conjunto de nivel aplicado a la reconstrucción de superficies. Se muestran resultados conseguidos para consumir el modelo del dragón de Stanford [22] proyectado en un volumen de tamaño 512^3. Se requieren 10 segundos para realizar el proceso en una NVIDIA GeForce GTX 280, mientras que nuestra solución termina en 2,8 segundos sobre este mismo volumen en una NVIDIA GeForce GTX 580. Por las razones expuestas anteriormente, podemos concluir que nuestra solución es aproximadamente 2 veces más rápida.

VI. Conclusiones

En este trabajo presentamos dos propuestas de implementación en GPU del método FTC de segmentación de conjuntos de nivel y mostramos su eficiencia en términos de rendimiento y calidad de la segmentación. Nuestras propuestas evitan cálculos innecesarios de modo que se aumenta el rendimiento de las computaciones. La primera de las propuestas modifica la estructura iterativa del algoritmo en el ciclo uno evitando computaciones en cada región que no suponen un avance del frente. La segunda propues-

ta aplica un criterio más restrictivo para reducir el número de zonas activas evitando computar regiones en las que el frente se mantiene estable.

Ambas implementaciones se han probado en una NVIDIA GeForce GTX 580 comparando con la versión secuencial ejecutada en CPU. Los resultados obtenidos muestran valores de aceleración en torno a 4x para la imagen de prueba de tamaño $181 \times 217 \times 181$. La evaluación del coeficiente Dice muestra que ambas implementaciones en GPU mantienen la calidad de la segmentación respecto del algoritmo en CPU. Además, los resultados son competitivos respecto a otras implementaciones recientes de métodos de conjunto de nivel en GPU.

Agradecimientos

Julián Lamas y Pablo Quesada agradecen el apoyo económico del Ministerio de Ciencia e Innovación, Gobierno de España, bajo sendas becas FPI-MICINN. Los autores quieren mostrar su agradecimiento al Departamento de Visualización y Análisis de Datos del Konrad-Zuse-Zentrum für Informationstechnik Berlin, y en especial a Stefan Zachow, por facilitar el acceso a la herramienta Amira.

Referencias

[1] Sethian J., *Level Set Methods and Fast Marching Methods: Evolving interfaces in computational fluid mechanics, computer vision, and materials science*, Cambridge University Press, Cambridge, UK, 1996.

[2] Mike Roberts, Jeff Packer, Mario Costa Sousa, and Joseph Ross Mitchell, "A work-efficient GPU algorithm for level set segmentation," in *Proceedings of the Conference on High Performance Graphics*, Saarbrucken, Germany, 2010, HPG '10, pp. 123–132, Eurographics Association.

[3] Stanley Osher and Nikos Paragios, *Geometric Level Set Methods in Imaging,Vision,and Graphics*, Springer-Verlag New York, Inc., Secaucus, New Jersey, USA, 2003.

[4] David Adalsteinsson and James A. Sethian, "A fast level set method for propagating interfaces," *Journal of Computational Physics*, vol. 118, no. 2, pp. 269–277, May 1995.

[5] Danping Peng, Barry Merriman, Stanley Osher, Hongkai Zhao, and Myungjoo Kang, "A PDE-based fast local level set method," *Journal of Computational Physics*, vol. 155, no. 2, pp. 410–438, July 1999.

[6] Martin Rumpf and Robert Strzodka, "Level set segmentation in graphics hardware," in *Proceedings of IEEE International Conference on Image Processing*, Thessaloníki, Greece, 2001, ICIP' 01, pp. 1103–1106, IEEE.

[7] Aaron E. Lefohn, Joe M. Kniss, Charles D. Hansen, and Ross T. Whitaker, "A streaming narrow-band algorithm: Interactive computation and visualization of level sets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 10, no. 4, pp. 422–433, 2004.

[8] Won-Ki Jeong, Johanna Beyer, Markus Hadwiger, Amelio Vazquez, Hanspeter Pfister, and Ross T. Whitaker, "Scalable and interactive segmentation and visualization of neural processes in em datasets," *IEEE Transactions on Visualization and Computer Graphics*, vol. 15, no. 6, pp. 1505–1514, November 2009.

[9] Andrei C. Jalba, Wladimir J. van der Laan, and Jos B. T. M. Roerdink, "Fast sparse level-set on graphics hardware," *IEEE Transactions on Visualization and Computer Graphics*, vol. 99, 2012, PrePrints.

[10] Yoggang Shi and W.C. Karl, "A real-time algorithm for the approximation of level-set-based curve evolution," *IEEE Transactions on image processing*, vol. 17, no. 5, pp. 645–656, May 2008.

[11] Gábor J. Tornai and György Cserey, "2D and 3D level-set algorithms on GPU," in *Proceedings of the 12th International Workshop on Cellular Nanoscale and their Applications*, Berkeley, California, USA, 2010, CNNA '10, pp. 1–5, IEEE.

[12] J. Nickolls and W.J. Dally, "The GPU computing era," *IEEE Micro*, vol. 30, no. 2, pp. 56–69, March–April 2010.

[13] NVIDIA, Santa Clara, California, USA, *CUDA C programming guide (version 4.0)*, 2011.

[14] David B. Kirk and Wen-mei W. Hwu, *Programming Massively Parallel Processors: a Hands-on Approach*, Elsevier, Burlington, Massachussets, USA, 2010.

[15] NVIDIA, Santa Clara, California, USA, *CUDA C best practices guide (version 4.0)*, 2011.

[16] S. Osher and J. Sethian, "Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations," *Journal of Computational Physics*, vol. 79, pp. 12–49, December 1988.

[17] Intel Corporation, Santa Clara, California, USA, *Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide*, May 2011.

[18] D.L. Collins, A.P. Zijdenbos, V. Kollokian, J.G. Sled, N.J. Kabani, C.J. Holmes, and A.C. Evans, "Design and construction of a realistic digital brain phantom," *IEEE Transactions on Medical Imaging*, vol. 17, no. 3, pp. 463–468, June 1998.

[19] D. Stalling, H.C. Hege, and M. Zöckler, "Amira: An advanced 3D visualization and modeling system," `http://amira.zib.de`, Accedida el 2 de mayo de 2012.

[20] Alex P. Zijdenbos, Benoit M. Dawant, Richard A. Mangolin, and Andrew C. Palmer, "Morphometric analysis of white matter lesions in MR images: method and validation," *IEEE Transactions on Medical Imaging*, vol. 13, no. 4, pp. 716–724, December 1994.

[21] "GeForce GTX 580 performance," `http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-580/performance`, Accedida el 22 de mayo de 2012.

[22] "The Stanford 3D scanning repository," `http://graphics.stanford.edu/data/3Dscanrep/`, Accedida el 9 de mayo de 2012.