UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
POSGRADO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN
INSTITUTO DE INVESTIGACIONES EN MATEMÁTICAS APLICADAS Y SISTEMAS
SEÑALES, IMÁGENES Y AMBIENTES VIRTUALES

Segmentador de volúmenes 3D basado en lógica difusa en GPU

T E S I S

QUE PARA OPTAR POR EL GRADO DE:
MAESTRO EN CIENCIA E INGENIERÍA DE LA COMPUTACIÓN

P R E S E N T A:
J O S É D A N I E L Q U I N T O S L Ó P E Z

Director de Tesis:
Dr. Edgar Garduño Ángeles
Instituto de Investigaciones en Matemáticas Aplicadas y Sistemas

Ciudad Universitaria, Cd. Mx. **enero 2019**

A mi familia y amigos :).

Agradecimientos

Quiero agradecer al Posgrado en Ciencia e Ingeniería de la Computación en su área de Señales, Imágenes y Ambientes Virtuales (SIAV) por haberme brindado la oportunidad de realizar mis estudios. Así mismo, agradezco al Dr. Edgar Garduño Ángeles por aceptarme como su tutorado y asesorarme en el desarrollo de este trabajo, por todos sus aportes y correcciones, pero ante todo, por su paciencia, charlas y consejos.

Agradezco la participación de los jurados de mi comité evaluador, el Dr. Jesús Savage, el Dr. Alfonso Gastélum, el Dr. Miguel Padilla y el Dr. Bruno Carvahlo. Igualmente un agradecimiento a todos los profesores con los que tomé cursos, por transmitirme sus conocimientos y apoyarme en mi crecimiento académico. Gracias a Cecy, Lulú y Amalia por estar siempre al pendiente de los alumnos. Agradezco al Departamento de Ciencias de la Computación del IIMAS por haberme permitido hacer uso de sus instalaciones.

Quiero agradecer al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el apoyo económico sin el cual hubiera sido imposible llevar a cabo mis estudios, mediante la beca asignada con CVU/Becario 778822/612507 a través de la Convocatoria de Becas Nacionales 2016, identificada con el número 291137.

Finalmente pero no menos importante, un agradecimiento a mi familia y amigos, que siempre creyeron en mí aunque yo no lo hiciera. Gracias a mis padres, sin su apoyo afectivo (y económico) no hubiese podido llegar hasta aquí, las palabras no bastan para agradecer todo lo que han hecho por mí. A Germán y José Antonio, mis hermanos, el verlos esforzarse en sus respectivos trabajos hace que yo igualmente quiera dar lo mejor de mí. A mis compañeros de grupo, Cinthya, Rosario, Miguel, Jei

y César, por tantas comidas y charlas compartidas. A mis camaradas de generación, Marco y nuestras eternas discusiones en las materias que tomamos juntos, Scarlett, Ivette, Cecy, David, JC, César y Santi. A mi amigo Lalo Galicia, por tantos consejos y pláticas sobre la academia y la industria. A grandes amigos del equipo Troncos FC, Pedro, Lalo López y en especial a Héctor Alonso (por tantos partidos jugados y sobre todo pláticas amenas). Gracias a Wendy, Karen Ivonne, José Manuel y Migue, amigos que siempre tendré en estima en Puebla, y a la maestra Yalú Galicia y el Dr. Enrique Colmenares, por su amistad y apoyo incondicional desde mi etapa universitaria en la BUAP. A mis maestros de música, Alfonso, Diego y Marvin, sin la música como aditamento en mi preparación hubiese sido más pesado este proceso. Un especial agradecimiento a Martín e Isabel, por su amistad desde mi llegada a CDMX y por tantos recuerdos juntos, por su gran apoyo en la etapa final de este trabajo. A aquellos que no he nombrado y que me han brindado momentos dentro de su valioso tiempo.

Resumen

La segmentación de imágenes es un proceso importante dentro de las ciencias de la computación, ya que permite identificar objetos en una imagen que sean de interés para su análisis e interpretación. La segmentación de imágenes tiene diversas aplicaciones en áreas como la medicina, la astronomía, la visión artificial, entre otras. Dada su importancia, se han desarrollado diversidad de métodos que realicen dicho proceso de segmentación, en base a la problemática que se quiera resolver y al tipo de imágenes que se tengan.

Los algoritmos de segmentación apoyados en los principios de lógica difusa han demostrado ser eficientes bajo diferentes problemas que se presentan en las imágenes, como imágenes con diferentes condiciones de ruido, imágenes con bajo contraste, sombras o texturas. El algoritmo MOFS (multi-object fuzzy segmentation) es un algoritmo de segmentación de éste tipo, que se apoya en la selección de semillas por parte del usuario para caracterizar a los objetos que se quieran segmentar en la imagen. Igualmente el algoritmo MOFS ofrece la opción de volver a ejecutar el algoritmo en base a la selección de nuevas semillas, si la segmentación anterior no fue la deseada. Sin embargo, este algoritmo implementado de forma secuencial es costoso en tiempo de ejecución para conjuntos de datos grandes.

En este trabajo se presenta una propuesta de implementación del algoritmo MOFS en forma paralela, implementada en una GPU; de esta manera se pretende reducir el tiempo de ejecución del algoritmo de segmentación y facilitar al usuario la posibilidad de interactuar de una forma más rápida con el algoritmo.

Índice general

Índice de figuras

Capítulo 1

Introducción

Una *imagen digital* es el resultado de llevar a cabo un proceso de discretización sobre los datos provenientes de campos escalares $f : \mathbb{R}^n \to \mathbb{R}$ o campos vectoriales $f : \mathbb{R}^n \to \mathbb{R}^m$, donde $\mathbb{R}$ es el conjunto de los números reales y n y m son números enteros positivos que representan las dimensiones con las que se trabaja, de tal forma que estos datos puedan ser representados, almacenados y visualizados en una computadora. Un caso especial para formar una *imagen digital* es cuando en lugar de mapear a $\mathbb{R}$ se mapea a $\mathbb{Z}$, donde $\mathbb{Z}$ es el conjunto de los números enteros. El origen de los datos puede ser tan diverso que incluye desde simulaciones hasta mediciones directas en dos o más dimensiones [23, 39, 83]. Generalmente la recolección de estos datos se realiza mediante el uso de cámaras fotográficas [54] o algún dispositivo de imagenología (a través, por ejemplo, de rayos X, ultrasonido, tomografía, PET o resonancia magnética [40, 49, 65, 74, 83]); como las funciones a discretizar provienen de uno de los dispositivos antes mencionados, entonces la función discretizada reflejará tanto el proceso de adquisición como un proceso de alteración y contaminación propio del dispositivo recolector o el ambiente en el que se utilice (por ejemplo, las imágenes digitales de ultrasonido presentan un tipo especial de ruido llamado *speckle* [48, 69]). En este trabajo se hará énfasis en datos tridimensionales (3D) obtenidos por un proceso de discretización de campos escalares de la forma $f : \mathbb{R}^3 \to \mathbb{R}$. Sin embargo, el trabajo presentado no es excluyente para datos en otras dimensiones o de aquellos provenientes de campos vectoriales [30, 66, 78].

Como se mencionó en el parrafo anterior, la información proveniente de un campo

escalar es discretizada para poder ser procesada en una computadora, esta discretización típicamente es llevada a cabo al evaluar el campo escalar en ciertos puntos del espacio [40, 42]. Un punto en el espacio n-dimensional $\mathbb{R}^n$ se puede representar usando una n-tupla de la forma $\boldsymbol{x} = (x_1, x_2, ..., x_n)$ donde $x_i \in \mathbb{R}$; de forma similar, se utiliza la n-tupla $\boldsymbol{k} = (k_1, k_2, ..., k_n)$, donde $k_i \in \mathbb{Z}$, para representar un punto en el espacio n-dimensional de los enteros, o $\mathbb{Z}^n$ [42].

Para llevar a cabo la discretización de una función f se puede realizar la siguiente multiplicación punto a punto

$$\mathrm{f} = f \times \text{Ш}_{G_\Delta}, \tag{1.1}$$

donde Ш_{G_Δ}representa la siguiente distribución

$$\text{Ш}_{G_\Delta} = \sum_{\boldsymbol{y} \in G_\Delta} \delta_{\boldsymbol{y}} \tag{1.2}$$

y $\delta_{\boldsymbol{y}}$ es la *delta de Dirac* trasladada a la posición determinada por el punto $\boldsymbol{y}$, de tal forma qué $\delta_{\boldsymbol{y}} f = f(\boldsymbol{y})$; a la distribución (1.2) se le conoce como *tren de pulsos* sobre el conjunto G_Δ [35, 72]. Para las definiciones anteriores, G_Δ es el siguiente conjunto de puntos (también denominado *rejilla*)

$$G_\Delta = \left\{ \Delta \boldsymbol{k} \,|\, \boldsymbol{k} \in \mathbb{Z}^3 \text{ y } \Delta \in \mathbb{R}_+ \right\}, \tag{1.3}$$

donde Δ se conoce como la *distancia*, o *intervalo*, *de muestreo* [35, 37] y $\mathbb{R}_+$ es el conjunto de los números reales positivos. La distribución de puntos sobre la que se lleva a cabo el muestreo determina el tipo de vecindario de Voronoi y, por lo tanto, el tipo de elementos espaciales (*spels*, por sus siglas en inglés) con los que se conformará la *imagen digital*. Es fácil ver que el vecindario de Voronoi para cada punto en el conjunto (1.3) es un cubo de volumen Δ^3 y por lo tanto el muestreo (1.1) produce *spels* denominados vóxeles (del inglés *volume element*) cúbicos; aunque este tipo de *spels* son con los que trabajaremos de aquí en adelante, el presente trabajo no es excluyente para otro tipo de elementos espaciales, como los presentados en [38]. De igual manera, debido a que en este trabajo utilizaremos solamente el conjunto G_Δ para

colocar los puntos de muestreo, entonces usaremos la palabra vóxel para referirnos a un vóxel cúbico.

Debido al proceso de muestreo (1.1) es posible definir la siguiente función discreta

$$\mathtt{f} = \{f(\boldsymbol{y}) \mid \boldsymbol{y} \in G_\Delta\}, \tag{1.4}$$

donde $\boldsymbol{y} = \Delta\boldsymbol{k}$. El vóxel asociado para la posición $\boldsymbol{y}$ estaría definido como

$$\text{vóxel}(\boldsymbol{y}) = \{\boldsymbol{x} \in \mathbb{R}^3 | \Delta(k_i - \frac{1}{2}) \leq x_i \leq \Delta(k_i + \frac{1}{2}), \text{ para } 1 \leq i \leq 3\}. \tag{1.5}$$

Por lo tanto, se puede uno referir al vóxel centrado en la posición $\boldsymbol{y} \in G_\Delta$ por medio del punto $\boldsymbol{y}$ o, para una Δ fija, por medio del punto $\boldsymbol{k}$.

Para los puntos en G_Δ existen varias adyacencias, pero la más común es

$$(\boldsymbol{c}, \boldsymbol{d}) \in \omega \Leftrightarrow \|\boldsymbol{c} - \boldsymbol{d}\| = \Delta, \tag{1.6}$$

donde $\|\boldsymbol{c}\|$ representa la norma l_2 de $\boldsymbol{c}$, definida como $\|\boldsymbol{c}\| = \sqrt{c_1^2 + c_2^2 + c_3^2}$. Cuando dos puntos $\boldsymbol{c}, \boldsymbol{d} \in G_\Delta$ son ω-adyacentes, entonces sus vóxeles asociados comparten una cara y por lo tanto, a esta adyacencia se le conoce como "adyacencia por cara". Finalmente, el *vecindario* del vóxel $\boldsymbol{c}$ se define como $N_\omega(\boldsymbol{c}) = \{\boldsymbol{d} | \boldsymbol{d} \in G_\Delta \text{ y } (\boldsymbol{c}, \boldsymbol{d}) \in \omega\}$; es fácil ver que son seis los vóxeles vecinos bajo la relación ω, por lo que $N_\omega(\boldsymbol{c})$ también es conocido como el 6-vecindario de $\boldsymbol{c}$. Vale la pena mencionar que el presente trabajo no es excluyente para otro tipo de relaciones que cambiarían el tipo de *vecindario* definido para un vóxel $\boldsymbol{c}$.

Teniendo en consideración lo explicado anteriormente, de aquí en adelante se hablará indistintamente de la función $\mathtt{f}$ definida en (1.4) como imagen digital, o imagen solamente, y de los elementos que la componen como vóxeles, definidos de la forma (1.5); así mismo, sin pérdida de generalidad, se utilizará un conjunto finito G, definido como $G \subset G_\Delta$, para hacer referencia a la rejilla G_Δ. Por otro lado, dado que los datos de una imagen comúnmente son visualizados en una pantalla de computadora, se suele hacer referencia a los datos que se presentan o envían a la pantalla como

niveles de gris o niveles de intensidad, lo cual constituye el *rango de visualización* de la imagen, que es por lo regular de 2^8 niveles o tonos. Finalmente, los valores escalares de los vóxeles que componen la imagen constituyen el *rango de la imagen*, y son conocidos como valores de la función, valores de la imagen o simplemente valores de los vóxeles, por lo que estos últimos tres términos se utilizarán indistintamente en algunas secciones de este trabajo.

1.1 Segmentación

La segmentación es un método que divide un conjunto G en varios subconjuntos de vóxeles (típicamente, se desea que los subconjuntos sean disjuntos), de tal forma que los vóxeles de un subconjunto están conectados entre sí y comparten alguna característica o algunas características en común, por ejemplo, en un caso muy simple y sin pérdida de generalidad, todos aquellos que poseen el mismo valor escalar. A cada uno de estos subconjuntos conectados se le considera como "un objeto" y la conectividad que existe entre los vóxeles que lo forman determinará su topología. De forma general, se puede decir que todos los objetos que comparten características en común pertenecen a una misma "clase". La segmentación, en el ámbito de imágenes, es un método a través del cual se divide una imagen digital, definida por (1.4), en objetos o regiones de interés; se puede ver también como un proceso de "etiquetado" de los distintos vóxeles que componen la imagen, de tal forma que puedan ser analizados posteriormente. La segmentación de imágenes es un proceso fundamental en varias áreas donde se quiere analizar e interpretar una imagen con base en las características de los objetos o regiones que ésta representa [65]. La Figura 1.1 representa de forma sencilla lo que se desea en un proceso de segmentación.

El realizar un proceso de segmentación no es una tarea trivial. El número de objetos en los que se desea subdividir la imagen depende del problema a resolver y, por supuesto, de la imagen que se tenga como entrada [65]. No existe un método general y único que pueda ser utilizado para segmentar cualquier imagen. A decir de muchos autores, la segmentación termina cuando se satisfacen los intereses u objetivos

de la aplicación [65]. Debido a ello, han surgido distintos métodos de segmentación de acuerdo al problema que se desea resolver y el tipo de imágenes que se tengan que segmentar.

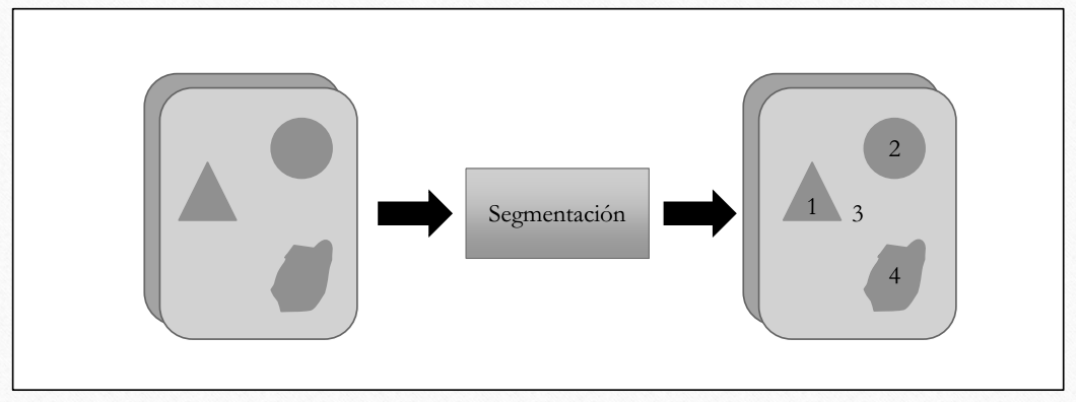

Figura 1.1: Esquema que representa el proceso de segmentación: una imagen digital 3D es procesada, o "segmentada", resultando en cuatro subconjuntos de vóxeles u "objetos".

Se puede realizar una clasificación simple de los métodos de segmentación en tres tipos: manuales, automáticos y semiautomáticos. Los métodos de segmentación manuales son aquellos que proveen una interfaz gráfica de usuario (*GUI*, por sus siglas en inglés) para que el usuario sea el encargado de realizar, en su totalidad, el proceso de segmentación de una imagen [63, 76, 88]. En estos métodos es importante el conocimiento del dominio del problema que tenga el usuario o su *expertise* para identificar los objetos en la imagen [63], así como de las herramientas propias de la interfaz [8, 88]; generalmente, éstas herramientas incluyen la posibilidad de seleccionar regiones a través de rectángulos u otras figuras geométricas, trazar *splines & smooth patches* con vóxeles establecidos por el usuario, etiquetar los vóxeles seleccionados para clasificarlos como miembros de alguna "clase" y borrar las selecciones que ya no sean deseadas [8]. En varias aplicaciones el resultado de estos métodos se usa como punto de comparación o "la verdad", para determinar si el resultado de otros métodos de segmentación (automáticos o semiautomáticos) es correcto o adecuado, por ejem-

plo, en algunos casos donde se realiza segmentación de imágenes médicas, como los presentados en [11, 13, 41]. Una lista de algunas herramientas que usan segmentación manual se puede encontrar en [3]. Por otro lado, los métodos de segmentación automáticos son aquellos en los que la segmentación de la imagen es llevada a cabo en su totalidad por el *software*. Ofrecen un mecanismo para superar el tedio involucrado en la segmentación manual de grandes conjuntos de imágenes [81], aunque muchas veces estos métodos requieren datos de entrenamiento, es decir, datos en donde los vóxeles de imágenes similares a las que se quieren segmentar ya hayan sido clasificados con anterioridad, por ejemplo, con métodos de segmentación manuales para generar un conjunto de datos estadísticos que se puedan utilizar durante el proceso de segmentación automática [41, 71]. Estos métodos han sido basados principalmente en técnicas de inteligencia artificial y reconocimiento de patrones, como el *clustering*, que agrupa vóxeles basándose en sus características [71]; otros ejemplos de estos métodos son aquellos que hacen uso de redes neuronales [67, 79, 81] o la segmentación basada en multi-atlas (*MAS* por sus siglas en inglés), que involucra campos como el aprendizaje de máquina, modelado probabilístico y optimización [45, 51]. Finalmente, los métodos de segmentación semiautomáticos son aquellos que requieren la intervención del usuario ya sea para inicializar el proceso de segmentación, intervenir durante el proceso o corregir el resultado del mismo. Por ejemplo, una modalidad de estos métodos es cuando el usuario selecciona vóxeles iniciales, comúnmente llamados *semillas*, que representan el objeto u objetos de interés, para que posteriormente el algoritmo de segmentación elegido sea el encargado de realizar el seguimiento y clasificación de los vóxeles que restan en la imagen con base en las características (por ejemplo, el valor escalar) de las *semillas* que ha elegido el usuario [24, 50, 60, 70].

Otra clasificación básica de los métodos de segmentación se puede realizar en otras tres categorías o enfoques: métodos basados en umbralización (*thresholding*), métodos basados en la detección de discontinuidades (bordes o fronteras) y métodos basados en la formación de regiones (a través de la similitud que existe entre grupos de vóxeles) [10, 12, 40, 65].

La umbralización es una de las técnicas más antiguas para la segmentación de

imágenes, en donde la imagen se divide en secciones de acuerdo a un umbral de corte en los valores de los vóxeles que forman la imagen. El umbral se puede elegir en función del histograma de la imagen, de la matriz de co-ocurrencia de los valores de la imagen [44] o analizando la homogeneidad dentro y fuera de la región o regiones que se quieren segmentar [9]. En los métodos basados en fronteras lo que se busca por lo general es identificar y localizar discontinuidades que están muy marcadas en la imagen [49]. Éstos métodos hacen uso principalmente de la detección de puntos, bordes y líneas de manera local, y posteriormente realizan una agrupación de dichas características en contornos que separen o delimiten los distintos objetos de interés. Los métodos más establecidos implican detección de bordes, pero también destacan los modelos deformables y los modelos basados en morfología [23]. Por último, la segmentación de imágenes con métodos basados en regiones se basa en el agrupamiento de vóxeles en zonas con características similares; para una región dada, en general, los vóxeles pertenecen a un único objeto. A este conjunto de puntos conectados que pertenecen al mismo objeto se le llama "región" [65]. Dentro de estos métodos se encuentra la técnica de *clustering* mencionada anteriormente, así como métodos de *split-and-merge*, métodos basados en teoría de gráficas, crecimiento de regiones, entre otros [23, 40, 55, 65].

Los métodos mencionados en el párrafo anterior pueden ser tanto automáticos como semiautomáticos. Hablando específicamente de los métodos basados en regiones, aquellos que son automáticos deben de ser capaces de determinar, dado algún criterio, las regiones en las cuales la imagen debe de ser segmentada. En los métodos semiautomáticos, por otro lado, el usuario es el que determina el número de objetos o regiones de la imagen a segmentar [65]. Uno de los enfoques básicos dentro de los métodos semiautomáticos basados en regiones es el de empezar el proceso de segmentación con un conjunto de *semilla*s, proporcionadas por el usuario, a partir de las cuales las regiones se irán expandiendo o creciendo. Los demás vóxeles en la imagen serán agrupados a su correspondiente *semilla* de acuerdo a algún criterio de similitud [40]; se parte del hecho de que al menos cada región que se quiere segmentar en la imagen cuenta con una *semilla* que distingue a dicha región de las demás. La

selección del criterio de similitud depende no solo del problema, sino del tipo de datos de imagen disponibles [23]. Los algoritmos de segmentación de imágenes basados en regiones deben tomar en consideración también la proximidad que exista entre vóxeles (es decir, considerando su vecindario, como el definido en (1.6)) para poder producir una segmentación con regiones conectadas. Los problemas principales que presenta el crecimiento de regiones son la selección de las *semillas* que representen correctamente las regiones de interés, la selección de un criterio de paro para detener el crecimiento de una región, cuando no haya más vóxeles que satisfagan el criterio de similitud en esa región, y la creación de un criterio de similitud [40] que contemple los problemas que pueden tener las imágenes, mencionados al inicio de este capítulo, como el bajo contraste, diferentes condiciones de ruido, artefactos, tipos de iluminación, sombras o texturas, que afectan en la decisión para incluir un vóxel en una u otra región. Una de las técnicas emblemáticas de segmentación basada en el crecimiento de regiones es la Transformada Watershed [15, 16, 73], cuya analogía básica consiste en considerar una imagen como una superficie topológica, en donde los valores escalares de los componentes de la imagen representan alturas, y por lo tanto las regiones homogéneas se pueden considerar como cuencas rodeadas por elevaciones [23]. El principal inconveniente de este método de segmentación es que produce muchas regiones y resultados sobre-segmentados [23], lo cual podría provocar que el resultado no sea útil [40, 73, 80].

Se han propuesto nuevos métodos para realizar segmentación de imágenes que, en muchos casos, han logrado tener mejor respuesta que los métodos tradicionales mencionados anteriormente, por ejemplo, los métodos de segmentación basados en Ecuaciones Diferenciales Parciales (*PDE*s, por sus siglas en inglés) [53]. Estos métodos han mostrado tener buenos resultados debido a que muchos fenómenos físicos pueden ser descritos por *PDEs* [53, 84], y la teoría que hay detrás tanto de los conceptos como de las técnicas de solución de estos fenómenos está bien establecida, lo que ayuda a tener una mejor representación del procedimiento que se está realizando y lograr la segmentación que se desea [84]; el método de contornos activos (*snakes*) [85], el método de flujo del vector gradiente [52] y la segmentación por conjuntos de nivel (*Level Sets*) [47, 58] son ejemplos de este tipo de métodos. Otro tipo de algoritmos

de segmentación son aquellos que están basados en lógica difusa. En la lógica difusa es esencial la utilización de una función de pertenencia por medio de la cual se puede decir que un elemento pertenece a un conjunto difuso con cierto grado [26, 27]. La definición de la función de pertenencia determinará como se asignan elementos a los conjuntos difusos. De manera similar, para una imagen digital, la pertenencia de un vóxel a un conjunto difuso estará determinada por dos factores, una "medida local", que es el grado de *afinidad* de un vóxel con respecto a sus vecinos, y una "medida general", que es una medida de *conectividad* (que depende de la función de *afinidad*) de un vóxel hacia vóxeles que están más allá de su vecindario.

La segmentación de conectividad difusa, o simplemente segmentación difusa, es una metodología para encontrar uno o más objetos de interés en una imagen digital, basada en un conjunto de puntos *semilla* especificados por el usuario y también en las funciones de *afinidad* que mapean cada par de puntos en la imagen a un valor real en el intervalo $[0, 1]$, y que ayudan a determinar la *conectividad* que hay entre ellos [26]. Los algoritmos de segmentación basados en los principios de la lógica difusa, y más específicamente en el concepto de conectividad difusa, han demostrado ser eficientes bajo diferentes condiciones de ruido, textura y artefactos; este conjunto de algoritmos es conocido comúnmente como métodos de segmentación difusa [22, 23, 26].

En la segmentación difusa existen principalmente dos escuelas de desarrollo. La primera, propuesta por Udupa [78], define el proceso de conectividad difusa relativa (RFC, por sus siglas en inglés) y el proceso de conectividad difusa relativa iterativa (IRFC, por sus siglas en inglés); la otra escuela, propuesta por Herman [43], define el proceso de segmentación difusa de múltiples objetos (MOFS, por sus siglas en inglés). En las segmentaciones RFC e IRFC, la segmentación de los objetos de interés es disjunta, es decir, cada elemento de la imagen que se está segmentando pertenece a un y sólo un objeto, solo existe una función de *afinidad* (sin importar el número de objetos de interés que sean) y también ésta función de *afinidad* es simétrica [26]. Por otro lado, en la segmentación MOFS, un elemento de la imagen puede pertenecer a uno o más objetos, siempre y cuando tenga en cada uno de ellos el mismo valor de *conectividad*, puede haber distintas funciones de *afinidad* (en donde cada clase podría

tener asociada su propia función de *afinidad*) y, a diferencia de los otros métodos como RFC e IRFC, éstas *afinidades* utilizadas no necesitan ser simétricas [22, 26].

El uso del algoritmo MOFS para la segmentación de imágenes ha producido resultados robustos con base a las semillas elegidas por el usuario y las funciones de afinidad utilizadas [20, 22, 26]. Debido a que hay una gran cantidad de experiencias exitosas utilizando este algoritmo en la comunidad de imágenes médicas y en nuestro grupo de trabajo [20, 22, 23, 35, 39, 43, 55], en el presente trabajo se hará uso del algoritmo MOFS, que se explica con más detalle en el Capítulo 2.

Por el momento, solamente mencionaremos que uno de los principios del algoritmo MOFS es ser supervisado, es decir, dado que el resultado final de la segmentación es dependiente de las semillas que elige el usuario, si la segmentación no es la deseada, se debe poder volver a ejecutar el algoritmo con la selección de nuevas semillas (y por ende, esperar un resultado distinto); debido a esto, es de importancia la velocidad de ejecución del algoritmo.

Al igual que en otros métodos de segmentación, el tiempo de ejecución (así como el consumo de memoria) utilizando el algoritmo MOFS depende, en su mayoría, del número de vóxeles con los que se está trabajando. Recientemente, las nuevas tecnologías de adquisición de imágenes se han vuelto capaces de producir conjuntos de datos cada vez más grandes, lo que impacta en el tiempo de ejecución de la implementación secuencial del algoritmo MOFS que se tiene actualmente en nuestro grupo de trabajo. Así mismo, la ejecución del algoritmo MOFS sería lenta para aplicaciones en las que es requerido el procesamiento de imágenes en tiempo real o procesamiento de video [30, 64] debido, nuevamente, a la cantidad de datos a procesar, ahora en el dominio 4D. Por estas razones, en este trabajo se propone una implementación paralela del algoritmo MOFS bajo una GPU, para optimizar el tiempo de ejecución en el que es llevado a cabo el proceso de segmentación.

El Capítulo 3 aborda algunos conceptos de paralelismo así como el trabajo realizado, es decir, la implementación paralela del algoritmo MOFS bajo el lenguaje de programación OpenCL®; se presenta también en el Capítulo 3 una breve discusión del por qué de la elección de este lenguaje de programación así como la comparación

con otras propuestas de paralelización. En el Capítulo 4 se presentan los resultados obtenidos en este trabajo. Finalmente, en el Capítulo 5 se presentan las conclusiones del proyecto y las propuestas para trabajo futuro.

Capítulo 2

Segmentación difusa de múltiples objetos

Como se mencionó al final del Capítulo 1, el algoritmo MOFS es un algoritmo de segmentación basado en los principios de lógica difusa que ha tenido buenos resultados. Como cualquier método de este tipo, el algoritmo MOFS requiere de una intervención mínima del usuario para indicar puntos (vóxeles), comúnmente llamados *semillas*, dentro de los objetos de interés que se quieren identificar, para después calcular la similitud entre las semillas y el resto de los vóxeles de la imagen [22, 23, 43, 55]. La similitud entre cualquier semilla y otro vóxel de la imagen se calcula usando una *función de conectividad difusa* (mencionada sólo como función de conectividad o conectividad de aquí en adelante), que es una medida global para asignar la afinidad que existe entre la pareja de vóxeles; a su vez, la conectividad depende de una medida local entre dos vóxeles adyacentes, definida por (1.6), comúnmente conocida como una relación de *afinidad difusa* (mencionada sólo como función de afinidad o afinidad de aquí en adelante). La función de *afinidad* en el contexto de segmentación difusa típicamente tiene dos componentes, uno para capturar las características de los vóxeles vecinos y otro para capturar su homogeneidad [23].

De la forma en que fue presentado en [43], y mejorado en [22], el algoritmo MOFS puede ser entendido inicialmente a través de una analogía con un ejercicio militar de control y expansión de territorios, que se describe a continuación. Supongamos que existen M ejércitos que luchan entre sí para conquistar N territorios. En este

ejercicio, M corresponde al número de clases de interés en las que se quiere segmentar una imagen (en una clase puede haber 1 o más objetos) y N corresponde al número de vóxeles que la constituyen. A su vez, cada pareja de territorios adyacentes está conectada por un camino, y dicho camino tiene una *afinidad* establecida para cada uno de los ejércitos; esta *afinidad* se puede medir con un entero no negativo, entre menor sea este entero, al ejército en cuestión le resultará más difícil viajar por ese camino.

Al inicio del ejercicio, los M ejércitos son colocados estratégicamente en algunos territorios *semilla* e inician con su fuerza máxima de expansión, representada por el valor real 1. Posteriormente, todos los ejércitos tratarán de incrementar sus dominios, marchando desde los territorios que ocupan hacia otros territorios vecinos, por los caminos que los unen. La fuerza de un ejército se verá reducida conforme avance el ejercicio, debido al desplazamiento del ejército por los caminos hacia otros territorios; dicha fuerza será equivalente al valor mínimo entre la fuerza que tenía el ejército en el territorio de donde salió y la *afinidad* del ejército respecto al camino en donde está transitando.

La fuerza del ejército que está ocupando un territorio en algún momento dado se conoce también como la *resistencia* que tiene dicho territorio a ser ocupado por otros ejércitos. Un territorio puede estar ocupado por uno o más ejércitos, siempre y cuando éstos ejércitos tengan la misma fuerza; por otro lado, si ningún ejército está ocupando el territorio, la *resistencia* de dicho territorio estará asignada a 0. La *resistencia* de cada territorio puede aumentar a medida que los ejércitos se movilicen y lo ocupen. Un ejército ocupará un territorio sí y solo sí tiene una fuerza mayor que venza a la *resistencia* del territorio. Si un ejército ocupa un territorio, podrá enviar unidades desde él hacia otros territorios cercanos.

El ejercicio finalizará cuando los ejércitos ya no puedan continuar expandiendo sus territorios, debido a que ya no pueden derrotar a ejércitos defensores en otros territorios o su fuerza ha disminuido a 0, debido al desplazamiento realizado por los caminos. El objetivo del ejercicio militar de control y expansión de territorios es determinar cómo la configuración final de los territorios ocupados depende de la

asignación inicial de los territorios a los distintos ejércitos y de la afinidad, que dichos ejércitos tienen, para viajar por los caminos que unen a los territorios.

2.1 Teoría

Como se mencionó en el Capítulo 1, por simplicidad y sin pérdida de generalidad, explicaremos la teoría y el algoritmo MOFS con base en el conjunto G. Sin embargo, es importante mencionar que tanto la teoría como el algoritmo presentados en este trabajo no trabajan solamente con conjuntos de datos del mismo tipo que el conjunto G, sino que también es posible utlizar cualquier otro conjunto de datos finito V, cuyos elementos pueden ser, por ejemplo, píxeles de una imagen 2D [21, 78], puntos en el plano [82] o vectores de caracterísitcas [32]; además, tanto la teoría como el algoritmo son independientes del área de aplicación de este trabajo (es decir, la segmentación de imágenes 3D), y por lo tanto, se pueden aplicar al *clustering* de datos en general [46].

En un enfoque general, siguiendo con las definiciones (1.3) y (1.4), se tiene que `f` es una función discreta, que al estar definida en el conjunto G, representa los valores escalares que se pueden asignar a un vóxel $\boldsymbol{c} \in G$. Un objeto se puede definir como un conjunto de vóxeles conectados entre sí y que comparten alguna característica; este objeto a su vez pertenece a una clase en particular. Cuando se habla de segmentación de imágenes, lo que se desea es dividir el conjunto de voxéles G en distintos objetos, para un número M de clases de interés, que representan a los M ejércitos de la analogía explicada en la sección anterior. Esta división se realizará de forma difusa, esto es, como parte del proceso de decidir si un vóxel pertenece o no a un objeto de una clase en particular, se le asignará a este vóxel un grado de pertenencia o conectividad a dicha clase (un número real entre 0 y 1, en donde 0 indica que el vóxel definitivamente no pertenece a la clase, y 1 indica que definitivamente sí). Para este propósito, el algoritmo de segmentación difusa MOFS ocupa una función σ que, para M clases, mapea cada elemento $\boldsymbol{c} \in G$ a un vector de $(M+1)$ dimensiones, definido como

$$\sigma^{\boldsymbol{c}} = (\sigma_0^{\boldsymbol{c}}, \sigma_1^{\boldsymbol{c}}, ..., \sigma_M^{\boldsymbol{c}}), \tag{2.1}$$

donde $\sigma_m^{\boldsymbol{c}}$, para el rango $1 \leq m \leq M$, representa el grado de pertenencia de un voxél $\boldsymbol{c}$ a la "m-ésima" clase y $\sigma_0^{\boldsymbol{c}}$ es el máximo de todos los valores $\sigma_m^{\boldsymbol{c}}$, de forma tal que $\sigma_0^{\boldsymbol{c}} \in [0, 1]$ y dicho valor corresponde a la clase con la que el vóxel $\boldsymbol{c}$ tiene mayor conectividad. En el vector de (2.1) existe, por lo menos, algún valor de m para el cual $\sigma_m^{\boldsymbol{c}} = \sigma_0^{\boldsymbol{c}}$, y para cualquier otra posición m del vector, $\sigma_m^{\boldsymbol{c}}$ tiene valor igual a 0 o $\sigma_0^{\boldsymbol{c}}$. El vector de (2.1) también es conocido como una M-semisegmentación del conjunto G. Cabe destacar que en una M-semisegmentación se permite a un vóxel $\boldsymbol{c}$ pertenecer a más de una clase (equivalente a que un territorio pueda estar ocupado por más de un ejército en la analogía del ejercicio militar), siempre y cuando éste tenga el mismo grado de pertenencia en cada una de ellas.

El concepto básico a partir del cual se construye $\sigma^{\boldsymbol{c}}$ es el de *conectividad*, el cual permite asignar a cada par de elementos del conjunto G un valor real en el rango $[0, 1]$. A su vez, la segmentación difusa asume que el valor de un vóxel $\boldsymbol{c}$ puede definir si éste es miembro o no de una clase m, y que esa propiedad de pertenencia cambia suavemente. De igual forma, la segmentación difusa supone que las imágenes 2D o 3D pueden representarse como una gráfica con nodos y aristas [23, 55], tal y como se muestra en la Figura 2.1.

La teoría de segmentación difusa indica que, en una imagen digital, el grado de pertenencia de un vóxel a un conjunto puede ser determinado por una función de conectividad, que depende a su vez de la afinidad existente entre vóxeles adyacentes [23, 74]. En este caso particular, esa relación difusa puede representarse por medio de una gráfica como se muestra en la Figura 2.2, que es conocida como mapa de conectividad, en donde los vóxeles representados en la Figura 2.1 son mapeados a un conjunto de nodos cuyo valor indica el grado de pertenencia que cada vóxel (nodo) tiene a las clases correspondientes, y el peso de las aristas representa la afinidad entre vóxeles adyacentes.

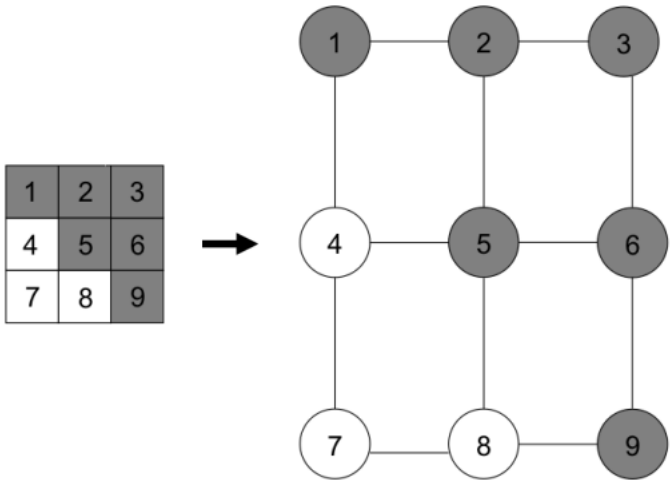

Figura 2.1: Representación de una imagen de tamaño 3×3 como una gráfica, donde cada vóxel corresponde a un nodo y las aristas de un nodo hacia otros nodos representan un tipo de adyacencia. En este caso, la adyacencia es "por cara". (Cortesía de [23].)

En la representación de la Figura 2.2, se puede definir una *cadena* como una secuencia $(\boldsymbol{c}^{(0)}, ..., \boldsymbol{c}^{(J)})$ de vóxeles, donde sus enlaces, o ligas, son dos vóxeles consecutivos $(\boldsymbol{c}^{(j-1)}, \boldsymbol{c}^{(j)})$ en la secuencia, para $0 \leq j \leq J$, y, para ser precisos, el peso $\psi_{\boldsymbol{c}^{(j-1)},\boldsymbol{c}^{(j)}}$ de cada arista corresponde a la *afinidad* entre la pareja de elementos $(\boldsymbol{c}^{(j-1)}, \boldsymbol{c}^{(j)})$, que también puede ser referida como la ψ-fuerza del enlace que comparten. La afinidad ψ mencionada se puede definir como una función difusa de la forma

$$\psi : G^2 \to [0, 1]. \tag{2.2}$$

Se hablará de la función de afinidad con más detalle en la siguiente sección de este capítulo. En la analogía del ejercicio militar, $(\boldsymbol{c}^{(j-1)}, \boldsymbol{c}^{(j)})$ representa el camino que une a los territorios $\boldsymbol{c}^{(j-1)}$ y $\boldsymbol{c}^{(j)}$, y la afinidad ψ es la que medirá la fuerza con la que un ejército m transitará por ese camino.

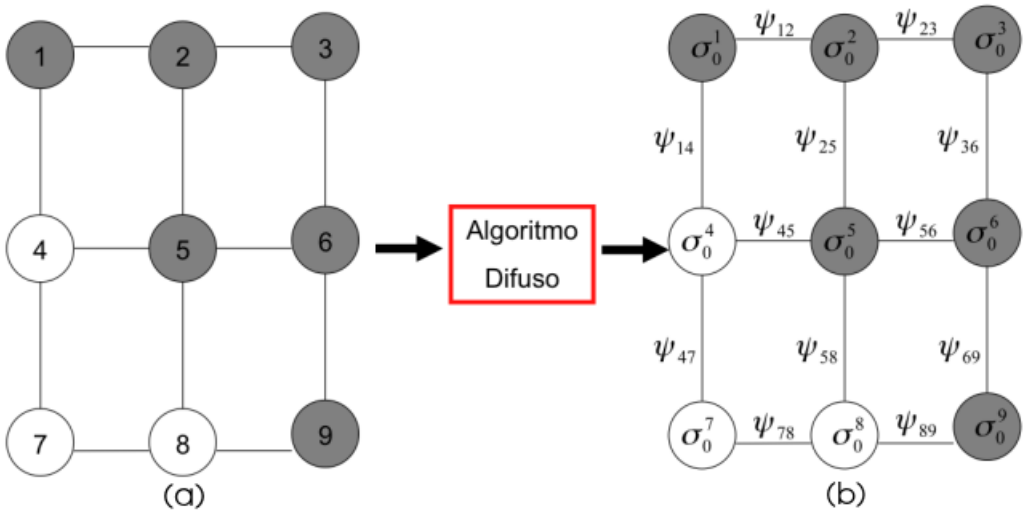

Figura 2.2: El algoritmo de segmentación difusa realiza un mapeo de la gráfica (a), que representa la imagen original, a una gráfica difusa (b), conocida como mapa de conectividad, en donde a cada nodo corresponde el valor $\sigma_0^{\boldsymbol{c}}$ y cada arista tiene un peso $\psi_{\boldsymbol{c},\boldsymbol{d}}$, que representa la afinidad existente entre nodos adyacentes. (Cortesía de [23].)

La μ-fuerza de una cadena está dada por la ψ-fuerza de su enlace más débil. Un conjunto $F \subseteq G$ está *conectado* bajo la función μ si, para cada pareja de vóxeles en F, existe una cadena de μ-fuerza positiva del primer vóxel al segundo. Con estas definiciones, la *conectividad* de cualquier pareja de vóxeles $\boldsymbol{c}$ y $\boldsymbol{d}$ del conjunto F se define como

$$\mu(\boldsymbol{c},\boldsymbol{d}) = \max_{\substack{<\mathbf{c}^0,\mathbf{c}^1,\ldots,\mathbf{c}^J> \\ \forall \mathbf{c}^0=\boldsymbol{c}\ y\ \mathbf{c}^J=\boldsymbol{d}}} [\, \min_{1\leq j\leq J} (\psi_{c(j-1),\mathbf{c}(j)}) \,], \tag{2.3}$$

donde la función μ representa, en otros términos, la μ-fuerza de la cadena más fuerte que conecta a los dos voxéles $\boldsymbol{c}$ y $\boldsymbol{d}$.

Ahora bien, si el objetivo es segmentar múltiples objetos, resulta razonable tener

para cada uno de ellos una propia definición de *afinidad*, así como un conjunto propio de *semillas* (esto corresponde a la idea, en la analogía del ejercicio militar, de que cada ejército tiene su propia afinidad para cada uno de los caminos por los que transita). De esta manera, se puede definir una gráfica M-difusa con la tripleta $(G, \mathbf{\Psi}, \mathcal{V})$, donde G es la imagen de entrada, $\mathbf{\Psi}$ es el conjunto de definiciones de afinidad para las M clases, esto es, $(\psi_1, \psi_2, ..., \psi_M)$, y $\mathcal{V}$ contiene los conjuntos de *semillas* para cada una de las clases, $(\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_M)$, con $\mathcal{V}_M \subseteq G$ para $1 \leq m \leq M$. Esta representación es *conectable* si se cumple que

1. El conjunto de entrada G está *conectado* bajo la función ψ como se definió en (2.3).

2. $\mathcal{V}_m \neq \emptyset$, para al menos un conjunto semilla en el rango $1 \leq m \leq M$.

El algoritmo de segmentación difusa de múltiples objetos, MOFS, ocupa los términos definidos anteriormente y hace uso del siguiente teorema como base de su funcionamiento

Teorema 1. Si $(G, \mathbf{\Psi}, \mathcal{V})$ es una gráfica M-difusa donde $\mathbf{\Psi} = (\psi_1, \psi_2, ..., \psi_M)$ y $\mathcal{V} = (\mathcal{V}_1, \mathcal{V}_2, ..., \mathcal{V}_M)$, entonces

(i) Existe una M-semisegmentación σ de G con las siguientes propiedades: para cada $\boldsymbol{c} \in G$, si para $1 \leq n \leq M$

$$s_n^{\boldsymbol{c}} = \begin{cases} 1, & si\, \boldsymbol{c} \in \mathcal{V}_n, \\ \max\limits_{\boldsymbol{d} \in G}(min(\mu(\boldsymbol{c}, \boldsymbol{d}), \psi_n(\boldsymbol{c}, \boldsymbol{d}))), & cualquier\, otro\, caso. \end{cases} \tag{2.4}$$

Luego para $1 \leq m \leq M$

$$\sigma_m^{\boldsymbol{c}} = \begin{cases} s_m^{\boldsymbol{c}}, & si\, s_m^{\boldsymbol{c}} \geq s_n^{\boldsymbol{c}},\, para\, 1 \leq n \leq M, \\ 0, & cualquier\, otro\, caso. \end{cases} \tag{2.5}$$

(ii) Esta M-semisegmentación es única, y

(iii) Esta es una M-segmentación, dado que $(G, \boldsymbol{\Psi}, \mathcal{V})$ es conectable.

La prueba de la validez de este teorema se puede encontrar en [22].

2.2 Función de afinidad

Es fácil ver que de (2.3), y su correspondiente uso en (2.1), la función de afinidad ψ es importante para el funcionamiento del algoritmo de segmentación difusa, ya que de la definición de ésta pueden depender de manera crítica los resultados entregados por el algoritmo.

En la literatura se menciona que para definir la función de *afinidad* ψ solo es necesario que esta sea monotónica y normal [27], además de que sean consideradas en su definición las características y homogeneidad de los objetos que representa. Hay quien afirma que esta función también debe de ser reflexiva [78] y simétrica [28].

Según [77, 78] la función de afinidad ψ para cualquier pareja de vóxeles $\boldsymbol{c}$ y $\boldsymbol{d}$ tiene como forma general

$$\psi(\boldsymbol{c}, \boldsymbol{d}) = \psi_{_A}(\boldsymbol{c}, \boldsymbol{d})[\omega_{_H}\psi_{_H}(\boldsymbol{c}, \boldsymbol{d}) + \omega_{_G}\psi_{_G}(\boldsymbol{c}, \boldsymbol{d})]. \tag{2.6}$$

Así mismo, se habla de una afinidad $\psi_m(\boldsymbol{c}, \boldsymbol{d})$, o simplemente ψ_m, para $1 \leq m \leq M$, cuando se hace referencia a la función de afinidad entre cualquier pareja de vóxeles $\boldsymbol{c}$ y $\boldsymbol{d}$ para una clase m en particular.

En (2.6), el término $\psi_{_A}$ es una relación de adyacencia entre los vóxeles $\boldsymbol{c}$ y $\boldsymbol{d}$, definida de la forma

$$\psi_{_A}(\boldsymbol{c}, \boldsymbol{d}) = \begin{cases} 1, & si\,(\boldsymbol{c}, \boldsymbol{d}) \in \omega, \\ 0, & de\,otra\,forma. \end{cases} \tag{2.7}$$

El término $\psi_{_G}$ permite medir el grado de similitud de las características de intensidad de los vóxeles a segmentar, mientras que el término $\psi_{_H}$ permite medir el grado de homogeneidad de los vóxeles $\boldsymbol{c}$ y $\boldsymbol{d}$ [23, 55]. Las funciones $\psi_{_G}$ y $\psi_{_H}$ han sido definidas

en [20, 21, 22, 43, 78] como

$$\psi_{_G}(\boldsymbol{c},\boldsymbol{d}) = e^{-\frac{(\boldsymbol{c}+\boldsymbol{d}-\eta_G)^2}{2\zeta_G^2}} \tag{2.8}$$

y

$$\psi_{_H}(\boldsymbol{c},\boldsymbol{d}) = e^{-\frac{(|\boldsymbol{c}-\boldsymbol{d}|-\eta_H)^2}{2\zeta_H^2}}, \tag{2.9}$$

donde η_G y ζ_G representan la media y la desviación estándar, respectivamente, de la sumatoria $\boldsymbol{u}+\boldsymbol{v}$ de cada pareja de vóxeles $\boldsymbol{u}$ y $\boldsymbol{v}$ en el vecindario $N_\omega(\boldsymbol{c})$, con $\boldsymbol{c} \in \mathcal{V}_m$ para $1 \leq m \leq M$; es fácil ver que η_G y ζ_G serán diferentes para cada clase m. De manera similar, η_H y ζ_H representan el cálculo de la media y la desviación estándar de la sumatoria de las diferencias absolutas $|\boldsymbol{u}-\boldsymbol{v}|$.

Para el propósito de la segmentación difusa de imágenes, los valores de los componentes de la función de afinidad (2.6) mencionados en el párrafo anterior a menudo pueden ser definidos automáticamente, basándose en propiedades estadísticas de los enlaces de los voxéles dentro de las regiones de interés identificadas por el usuario [22]. La idea de definir de esta forma dichos componentes de la función de *afinidad* ψ es que el usuario no tenga la tarea de definir de forma matemática las características del objeto a segmentar, sino que el *software* que implemente el algoritmo de segmentación sea el encargado de calcular algunas medidas estadísticas, de acuerdo al vecindario (definido por la relación (1.6)) de las *semillas* seleccionadas por el usuario.

Por último, en (2.6), $\omega_{_H}$ y $\omega_{_G}$ son constantes para ponderar a las funciones (2.8) y (2.9). En este trabajo su valor es asignado normalmente a 0.5, aunque se han investigado los efectos de estas constantes y también las formas de ajustarlas durante el proceso de segmentación [62].

2.3 Algoritmo de segmentación difusa

De acuerdo con el escenario de expansión militar presentado al inicio de este capítulo, se puede ver que lo que se obtiene al final de ese ejercicio militar es la M-segmentación (distribución de los M ejércitos en los territorios) del Teorema 2.1,

que corresponde a los valores de σ para cada uno de los vóxeles en la imagen, es decir, el mapa de conectividades. Con el objetivo de que el presente documento tenga la característica de ser auto-contenido, en la mayor medida posible, se presenta a continuación el algoritmo *fast*-MOFS descrito en [22], que es una optimización del algoritmo MOFS propuesto en [43]. El algoritmo recibe como entradas una función discretizada (o imagen) G, el conjunto de funciones de afinidad $\mathbf{\Psi} = \{\psi_m\}$ y el conjunto de semillas $\mathcal{V} = \{\mathcal{V}_m\}$, para $1 \leq m \leq M$, donde M es el número de clases en los que se quiere segmentar la imagen. Como se mencionó anteriormente, $\sigma_m^{\boldsymbol{c}}$ representa el grado de pertenencia o afinidad del vóxel $\boldsymbol{c}$ a la clase m, mientras que $\sigma_0^{\boldsymbol{c}}$ es el grado de pertenencia máximo que ha sido asignado al vóxel $\boldsymbol{c}$ para alguna de las clases.

Para hacer más rápido el algoritmo presentado en [43], se hace la suposición de que el conjunto de valores que pueden tomar las funciones de conectividad difusas es siempre un subconjunto de un conjunto fijo A, donde K es su cardinalidad y sus elementos son $1 = a_1 > a_2 > ... > a_K$. Lo anterior se basa en que en muchas aplicaciones la calidad de la segmentación no se ve significativamente afectada si los valores de conectividad se redondean a un número fijo de cifras decimales. En [22] tres cifras decimales son suficientes, lo que resulta en una cardinalidad $K = 1000$, con el conjunto de valores de conectividad definido como $A = \{1.000, 0.999, 0.998, ..., 0.002, 0.001\}$, siendo sus elementos $a_k = 1.001 - k/1000$, para $1 \leq k \leq K$; de esta manera, el valor 1 representa el grado de conectividad máximo de un vóxel $\boldsymbol{c}$ para alguna clase m. Adicionalmente, se dispone de una matriz U de tamaño $M \times K$, en donde cada elemento $U[m][k]$ contiene el conjunto de vóxeles que están ocupados por un objeto de la clase m y cuya conectividad actual hacia esa clase es $K + 1 - k$.

El Algoritmo 1 muestra el proceso de segmentación *fast*-MOFS. El proceso se inicializa en las líneas 1-9, en donde se asigna el valor de 0 a cada $\sigma_m^{\boldsymbol{c}}$, para cada vóxel $\boldsymbol{c} \in G$ y para $0 \leq m \leq M$, además de garantizarse que los conjuntos $U[m][k]$ estén vacíos. Posteriormente, cada vóxel $\boldsymbol{c} \in \mathcal{V}_m$ es incluido en el conjunto $U[m][1]$, realizando al mismo tiempo la asignación $\sigma_0^{\boldsymbol{c}} = \sigma_m^{\boldsymbol{c}} = 1$. Esta última asignación es equivalente al estado inicial en el escenario de expansión militar, en donde es fácil

ver que los vóxeles *semilla* seleccionados por el usuario tendrán asignado el valor de conectividad mayor hacia sus respectivas clases.

Algoritmo 1 Algoritmo *fast*-MOFS descrito en [22].

1. **para** $\boldsymbol{c} \in G$ **hacer**
2. **para** $m \leftarrow 0$ a M **hacer**
3. $\sigma_m^{\boldsymbol{c}} \leftarrow 0$
4. **para** $m \leftarrow 1$ a M **hacer**
5. **para** $\boldsymbol{c} \in \mathcal{V}_m$ **hacer**
6. $\sigma_0^{\boldsymbol{c}} \leftarrow \sigma_m^{\boldsymbol{c}} \leftarrow 1$
7. $U\,[m]\,[1] \leftarrow \mathcal{V}_m$
8. **para** $k \leftarrow 2$ a K **hacer**
9. $U\,[m]\,[k] \leftarrow \emptyset$
10. **para** $k \leftarrow 1$ a K **hacer**
11. **para** $m \leftarrow 1$ a M **hacer**
12. **mientras** $U\,[m]\,[k] \neq \emptyset$ **hacer**
13. **remover** un spel $\boldsymbol{d}$ del conjunto $U\,[m]\,[k]$
14. $C \leftarrow \{\, \boldsymbol{c} \in G \mid \sigma_m^{\boldsymbol{c}} < min\,(a_k, \psi_m\,(\boldsymbol{d}, \boldsymbol{c}))\ y\, \sigma_0^c \leq min\,(a_k, \psi_m\,(\boldsymbol{d}, \boldsymbol{c}))\}$
15. **mientras** $C \neq \emptyset$ **hacer**
16. **remover** un spel $\boldsymbol{c}$ de C
17. $t \leftarrow min\,(a_k, \psi_m\,(\boldsymbol{d}, \boldsymbol{c}))$
18. **si** $\sigma_0^{\boldsymbol{c}} < t$ **entonces hacer**
19. **remover** $\boldsymbol{c}$ de cada conjunto en U que lo contenga
20. **para** $n \leftarrow 1$ a M **hacer**
21. $\sigma_n^{\boldsymbol{c}} \leftarrow 0$
22. $\sigma_0^{\boldsymbol{c}} \leftarrow \sigma_m^{\boldsymbol{c}} \leftarrow t$
23. **insertar** $\boldsymbol{c}$ en el conjunto $U\,[m]\,[l]$, donde l es el entero tal que $a_l = t$

Una vez realizado el proceso de inicialización se ejecuta el ciclo principal (líneas 11-23), en donde, para cada valor de conectividad $k \in K$ y para cada objeto $m \in M$, son actualizados los valores de $\sigma_m^{\boldsymbol{c}}$. El valor actual de $\sigma_m^{\boldsymbol{c}}$ es reemplazado por uno mayor si existe una cadena de un objeto de la clase m, que parte de un vóxel en el conjunto $\mathcal{V}_m$, hasta el vóxel $\boldsymbol{c}$ con una fuerza (conectividad) mayor a la anterior. Por otro lado, el valor de $\sigma_m^{\boldsymbol{c}}$ es reemplazado por el valor de 0 si, para todo $n \neq m$, existe una cadena desde un vóxel en el conjunto $\mathcal{V}_n$ hasta $\boldsymbol{c}$ con una fuerza mayor al valor

actual de $\sigma_m^{\boldsymbol{c}}$, siendo este el caso en el que la clase n se apropia o "reclama" el vóxel $\boldsymbol{c}$. Una clase m puede apropiarse o "reclamar" al vóxel $\boldsymbol{c}$ si, y sólo si, $\sigma_m^{\boldsymbol{c}} > 0$ y $\sigma_m^{\boldsymbol{c}} \geq \sigma_0^{\boldsymbol{c}}$.

Al inicio de cada iteración k, para cada índice l en el rango $k \leq l \leq K$, el conjunto $U[m][l]$ contiene todos aquellos vóxeles que están ocupados por la clase m y cuya fuerza de conectividad actual es $K + 1 - l$. Durante la k-ésima iteración, retomando la analogía del escenario de expansión militar, los ejércitos de la clase m son enviados desde cada vóxel del conjunto $U[m][k]$ a todos los vóxeles adyacentes (líneas 11-14) para tratar de ocuparlos. Cuando un ejército de la clase m tiene éxito en ocupar un vóxel $\boldsymbol{c}$, y la fuerza resultante de $\boldsymbol{c}$ es igual a $K + 1 - l$, entonces $\boldsymbol{c}$ es insertado en el conjunto $U[m][l]$ (línea 23). Sin embargo, si $\boldsymbol{c}$ es tomado porque su fuerza anterior era menor a $K + 1 - l$, entonces es necesario remover al vóxel $\boldsymbol{c}$ de cualquier otro conjunto que lo contenga (línea 19).

Finalmente, pueden obtenerse como resultado las segmentaciones $\sigma_1, \sigma_2, ..., \sigma_M$ recorriendo, respectivamente, los elementos en los conjuntos $U[1], U[2], ..., U[M]$, en donde cada conjunto $U[m]$, para $1 \leq m \leq M$, representa los vóxeles conquistados por la clase m con los diferentes valores de conectividad dentro del conjunto A.

2.4 Aumento en el uso de los recursos de cómputo

Como se mencionó al final del Capítulo 1, se cuenta con conjuntos de datos cada vez más grandes, debido al avance en las tecnologías de adquisición de imágenes. Por ejemplo, las imágenes obtenidas por tomografía computarizada han evolucionado, desde su introducción clínica en 1971 en donde se obtenían imágenes de 160×160 píxeles únicamente del cerebro, hasta obtener imágenes tridimensionales de cualquier área anatómica, con imágenes entre 512×512 y 1024×1024 píxeles [18]; esto se puede apreciar en la Figura 2.3, en donde es notorio que la calidad de las imágenes ha mejorado considerablemente con los equipos actuales. Otro ejemplo claro es la evolución de las cámaras fotográficas digitales de los teléfonos celulares, que en sus inicios capturaraban imágenes de 640×480 píxeles y actualmente pueden capturar imágenes de 7228×4354 píxeles aproximadamente [2].

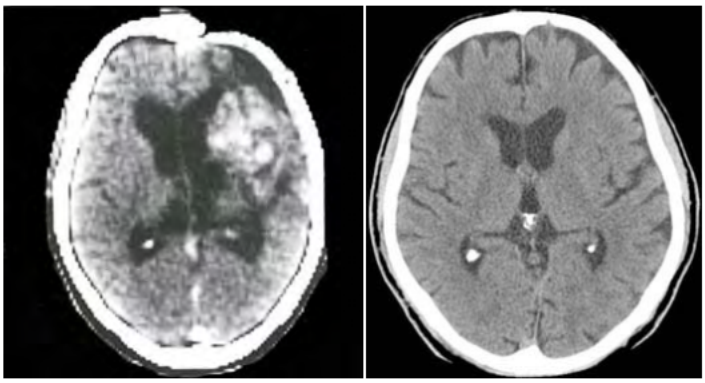

Figura 2.3: A la izquierda, imagen de 160×160 píxeles de una tomografía cerebral obtenida en los inicios de la técnica, a la derecha, imagen de un plano similar de 512×512 píxeles, obtenida con un tomógrafo actual. (Cortesía de [18].)

Sin embargo, el tamaño de los conjuntos de datos que se tienen actualmente involucra también un mayor uso de los recursos de cómputo que se utilizan para procesarlos. Aunque es posible realizar una segmentación de una imagen 3D de 7,000,000 vóxeles utilizando el algoritmo MOFS en 35 segundos aproximadamente [22], otros conjuntos de datos más grandes, por ejemplo de entre 30,000,000 y 70,000,000 vóxeles, pueden tomar entre 400 y 900 segundos respectivamente [36]; esto puede ser un tiempo no razonable para algunas aplicaciones, por ejemplo, aquellas en las que se desea realizar segmentación de video (esto es, imágenes en el dominio 4D), segmentación en tiempo real o simplemente aquellas en las que se cuente con conjuntos de datos aún más grandes [19, 30, 64]. Aunado al tiempo de ejecución, de igual manera se ve incrementado el consumo de memoria, debido a que las estructuras de datos utilizadas por la implementación del algoritmo MOFS deberán ser capaces de mantener en memoria los vóxeles y demás datos con los que se esté trabajando.

Otro factor importante que aumenta el uso de los recursos de cómputo, en especial el tiempo de ejecución, es la incorporación de medidas de textura en el proceso de segmentación; éstas son importantes especialmente para detectar información en donde los defectos y anormalidades en las estructuras de los objetos se caracterizan por las diferencias de propiedades de textura que estos tienen [7, 23]. Aunque en este

trabajo se ha descrito el cálculo de una función de afinidad con base en características estadísticas de los vóxeles semilla seleccionados por el usuario, podrían utilizarse otras funciones de afinidad que incorporen medidas de textura como las presentadas en [25]; así mismo, se pueden utilizar también funciones de afinidad que tomen en cuenta información de color [55]. Un proceso de segmentación que utilice medidas de textura u otra información adicional necesita el cómputo de varias dependencias espaciales, interacciones y asociaciones entre los vóxeles de los objetos que se quieren segmentar en la imagen [57], lo que impacta en el tiempo en el que estos cálculos son llevados a cabo.

Aunque se ha mencionado el consumo de memoria como un factor negativo debido al tamaño de los conjuntos de datos que se tienen actualmente, este trabajo se centrará solamente en optimizar el tiempo de ejecución del algoritmo MOFS presentado en este capítulo; una forma de lograr dicha optimización es a través del uso de computadoras paralelas.

Una computadora paralela se puede definir como un conjunto de procesadores que trabajan en conjunto para resolver un problema computacional; esta definición es lo suficientemente amplia como para incluir supercomputadoras que tienen cientos o miles de procesadores, clústers, redes de *workstations*, computadoras multiprocesador y sistemas integrados [33]. El cómputo paralelo (llamado de aquí en adelante como paralelismo) es una forma de cómputo en la que muchas instrucciones se ejecutan simultáneamente, gracias al uso de una computadora paralela. El paralelismo trata de hacer uso de todos los recursos de cómputo disponibles, dividiendo la cantidad de datos que se tienen que procesar y/o el proceso en sí a través de los distintos procesadores, lo que generalmente incrementa el rendimiento de un programa respecto a su tiempo de ejecución, en comparación con la ejecución de este mismo programa en un solo procesador, es decir, de forma secuencial [33, 59]. Parece razonable, entonces, utilizar este enfoque de cómputo para mejorar el rendimiento en tiempo de ejecución de la implementación secuencial (escrita en C++) del algoritmo MOFS que se tiene actualmente en nuestro grupo de trabajo; el trabajo realizado se presenta con más detalle en el siguiente capítulo.

Capítulo 3

Algoritmo MOFS en paralelo

Antes de entrar a detalle en la discusión de la implementación paralela del algoritmo MOFS que se realizó en este trabajo, es conveniente mencionar algunos conceptos de paralelismo que ayudarán a comprender la teoría y explicar el trabajo realizado, más allá de las justificaciones que se mencionaron al final del Capítulo 2 sobre la utilización del paralelismo para optimizar el tiempo de ejecución.

El proceso de convertir un programa secuencial o cualquier algoritmo en un programa paralelo es conocido como "paralelización". La paralelización se basa en el principio de que los problemas de gran tamaño o complejos (ya sea en el número de instrucciones a realizar o en la cantidad de datos a manipular) se pueden descomponer en conjuntos de problemas más pequeños o simples [75]. Una vez que se ha decidido realizar un proceso de paralelización, se debe elegir el enfoque con el que este proceso se llevará a cabo.

3.1 Paralelismo basado en tareas y paralelismo basado en datos

El paralelismo requiere dividir la cantidad de datos que se tienen y/o el proceso a realizar entre las distintas unidades de procesamiento de una computadora paralela; nos referiremos a estas unidades como "núcleos" a partir de ahora. En el enfoque de paralelismo basado en tareas (*task-parallelism*, en inglés), se dividen las diversas tareas (instrucciones) llevadas a cabo para resolver un problema entre los núcleos que

se tengan; en este enfoque cada uno de los núcleos ejecutará instrucciones diferentes, ya sea sobre el mismo conjunto de datos o sobre conjuntos de datos diferentes [59, 75]. Por otro lado, en el enfoque de paralelismo basado en datos (*data-parallelism*, en inglés), se dividen los datos utilizados para resolver un problema entre los distintos núcleos, y cada núcleo llevará a cabo instrucciones iguales o similares en la parte correspondiente de los datos que le han sido asignados [59, 75].

Para ejemplificar el funcionamiento de estos dos enfoques, se puede considerar el pseudocódigo que se muestra en la Figura 3.1, en donde para un número total T de datos, se realizan cuatro tareas diferentes (*tareaA*, *tareaB*, *tareaC* y *tareaD*), sumando el resultado obtenido de cada tarea en una variable general.

```
desde i=1 hasta i=T hacer
      resultadoA = tareaA(i);
      resultadoB = tareaB(i);
      resultadoC = tareaC(i);
      resultadoD = tareaD(i);
      general = general +
                resultadoA+resultadoB+resultadoC+resultadoD
fin
```

Figura 3.1: Pseudocódigo que hace la suma de los resultados de cuatro tareas diferentes, que operan sobre T datos distintos.

Consideremos, por simplicidad, una computadora paralela de cuatro núcleos, aunque este ejemplo no es excluyente para un número menor o mayor de núcleos. Si se decidiera hacer la paralelización con un enfoque de paralelismo basado en tareas, se podría delegar la ejecución de una sola tarea diferente a cada núcleo, en donde cada núcleo trabajaría con los T datos para la tarea que le fue asignada, siendo comúnmente el primer núcleo el encargado de realizar la acumulación de los resultados. Por otro lado, si se realiza la paralelización con un enfoque de paralelismo basado

en datos, cada núcleo ejecutaría las cuatro tareas por sí mismo, pero sólo sobre un subconjunto particular de los T datos, en donde también el primer núcleo sería el encargado de realizar la acumulación de los resultados. La representación gráfica de estos dos enfoques, para este ejemplo en particular, se puede apreciar en la Figura 3.2.

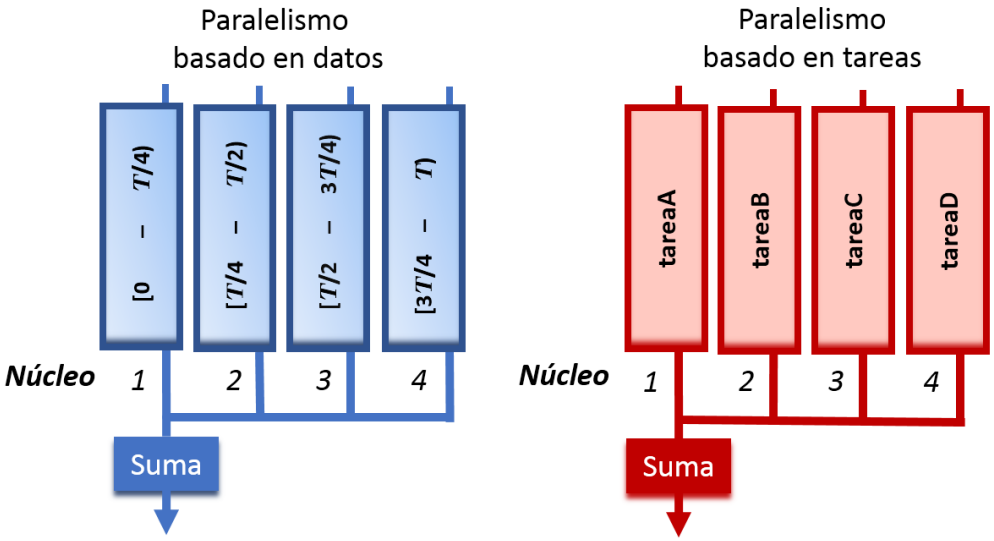

Figura 3.2: Comparación de los enfoques de paralelismo basado en datos y en tareas. En el enfoque de paralelismo basado en datos, se puede apreciar la subdivisión de los datos que se tienen a través de los cuatro núcleos, mientras que en el enfoque de paralelismo basado en tareas se da prioridad a la división de tareas. En ambos enfoques, comúnmente el primer núcleo es el encargado de acumular los resultados.

Es importante señalar que, sin importar el enfoque de paralelismo que se escoja, en un proceso de paralelización comúnmente se utiliza una arquitectura denominada maestro-esclavo (*master-slave* o *manager-worker*, en inglés), en la cual hay varios núcleos (los trabajadores o esclavos) que procesan subconjuntos de instrucciones y/o datos, así como también hay un núcleo especial (el administrador o maestro) que controla cómo los otros núcleos llevan a cabo su trabajo y por lo general también establece instrucciones de inicialización y finalización del proceso [31, 39]. En el

ejemplo descrito en el párrafo anterior, y como se puede apreciar también en la Figura 3.2, el primer núcleo es el que cumple la función de administrador, a la vez que también realiza operaciones para apoyar en la resolución del problema, mientras que los otros núcleos serían los trabajadores.

Se dice que un problema es perfectamente o adecuadamente paralelo cuando éste se puede dividir fácilmente entre los distintos núcleos de una computadora paralela. Sin embargo, un proceso de paralelizacción se vuelve mucho más complejo cuando los núcleos necesitan coordinar su trabajo, debido a la dependencia que existe entre los datos que manejan o porque la ejecución y resultado de una tarea A depende de la ejecución y resultado de otra tarea B. Los tres factores a considerar durante un proceso de paralelización en el que los núcleos requieren coordinación son la comunicación, el balance de carga y la sincronización [59]. La comunicación permite compartir entre los núcleos la información que sea requerida para su funcionamiento, el balance de carga trata de garantizar que todos los núcleos tengan la misma cantidad de trabajo a realizar y la sincronización permite que la ejecución del programa sea ordenada y el resultado final sea determinista [33, 59].

3.2 Tendencias, aceleradores y *GPUs*

Los sistemas de procesamiento paralelo que se desarrollan comúnmente involucran conectar *CPUs* genéricas y, generalmente, hacen uso de tecnologías como OpenMP^{TM} y MPI (*Message-Passing Interface*) [75] para realizar paralelismo basado en datos y paralelismo basado en tareas, respectivamente.

OpenMP^{TM} es una interfaz de programación de aplicaciones (*API* por sus siglas en inglés), diseñada para programar sistemas paralelos de memoria compartida, es decir, con una memoria general para uso de todos los núcleos; proporciona mecanismos para acceder a las ubicaciones de la memoria compartida, principalmente a través de un conjunto de directivas de compilador, funciones propias de la biblioteca y variables de entorno [59, 64]. OpenMP^{TM} permite añadir paralelismo a programas secuenciales escritos en C, C++ y Fortran. El área de procesamiento digital de

imágenes no es ajena al uso de esta tecnología, inclusive, en lo que respecta a la segmentación de imágenes con el algoritmo MOFS presentado en el Capítulo 2, en nuestro grupo de trabajo se cuenta con una implementación con OpenMP^{TM}, con un enfoque de paralelismo basado en datos, cuya teoría y resultados de ejecución pueden ser revisados en [36]. OpenMP^{TM} se basa en la creación de hilos de ejecución, en donde un hilo se puede definir como una tarea que puede ser ejecutada al mismo tiempo que otra tarea por el sistema operativo.

MPI, por otro lado, es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes, diseñada para programar sistemas paralelos de memoria distribuida, es decir, aquellos en los que cada núcleo tiene su propia memoria privada, y los núcleos deben comunicarse explícitamente enviando mensajes, generalmente a través de una red de computadoras [64]. MPI define la lógica del sistema, pero no especifica su implementación; la implementación se puede realizar de manera eficiente en una amplia gama de arquitecturas (generalmente bajo programas escritos en C y en Fortran), por lo que MPI ofrece portabilidad entre las mismas [59, 64].

Aunque OpenMP^{TM} y MPI han permitido construir sistemas paralelos eficientes, sin embargo, presentan algunos inconvenientes. Por ejemplo, el rendimiento en la ejecución de un programa que utiliza OpenMP^{TM} está ligado a las optimizaciones que pueda hacer el propio compilador del código fuente del programa; así mismo, aunque OpenMP^{TM} es capaz de crear muchos hilos de ejecución para ejecutar un código en modo paralelo, estos hilos de ejecución no necesariamente estarán ligados a núcleos físicos de la computadora paralela, lo cuál puede impactar en la sincronización y rendimiento del programa, por lo que se debe de ser cuidadoso a la hora de establecer las opciones o características de las directivas de compilador [36]. Por otro lado, aunque con MPI el programador tiene mayor control sobre la optimización y uso de recursos, el costo económico de construir un entorno en el cual sea posible ejecutar un programa bajo MPI puede ser elevado, por ejemplo, una red o un clúster de computadoras, aunado a que la latencia en la red impactará en las comunicaciones entre núcleos y por ende en el rendimiento general del sistema.

El uso de las dos tecnologías mencionadas anteriormente parece intuitivo para realizar paralelismo, sin embargo, existe otro enfoque para realizar cómputo paralelo, que es utilizar *hardware* especializado en la realización de ciertas tareas como un co-procesador; el *hardware* que cumple con estas características es conocido como un "acelerador" [6, 75]. Los aceleradores típicamente cuentan con núcleos optimizados para realizar operaciones aritméticas de punto flotante, y dado que estos núcleos son relativamente simples y no ocupan mucho espacio en un chip, generalmente son puestos en cantidades numerosas [6, 75]. Algunos aceleradores populares son los *Cell Broadband Engine* (*Cell/B.E.*) y las unidades de procesamiento de gráficos (*GPUs*, por sus siglas en inglés, que están integradas en las tarjetas gráficas de cualquier computadora actual y que también se venden como unidades externas).

La unidad de procesamiento de gráficos es un componente muy parecido a la *CPU*, solo que el tipo de procesamiento al que se dedica es al de gráficos, para generar imágenes visuales sintéticamente así como integrar o cambiar la información visual y espacial de las mismas [61, 75]. La diferencia entre una *GPU* y una *CPU* está en su arquitectura. Las *GPUs* están construidas de modo que sea mucho más eficiente realizar algunos cálculos de punto flotante y manejar grandes cantidades de datos [34]. Esto se logra gracias a que las *GPUs* están altamente segmentadas, lo que quiere decir que poseen una gran cantidad de unidades de procesamiento, que son el equivalente a los núcleos de una computadora paralela y que están agrupadas a su vez en lo que se conoce como "unidades de cómputo"; los núcleos de una *GPU* se dividen en grupos de igual tamaño que residen en una sola unidad de cómputo, y comparten parte de la memoria en el chip para un rápido intercambio y almacenamiento de datos [17, 56]. Los núcleos de una *GPU* son capaces de ejecutar el mismo flujo de instrucciones sincronizadamente, cada uno de ellos sobre un conjunto de datos distinto, por lo que el paralelismo basado en datos surge de manera natural [17]. La jerarquía de memoria es otra diferencia de la *GPU* con respecto a la *CPU*, cada núcleo puede ser manipulado a través de un hilo de ejecución, lo que quiere decir que tiene su propia *memoria privada*, a su vez, las unidades de cómputo tienen una memoria denominada *memoria local*, que es compartida por todos los núcleos que se encuentran en dicha

unidad de cómputo. Finalmente, todas las unidades de cómputo pueden acceder a una *memoria global*, en donde el ancho de banda a esta memoria suele ser mayor que el ancho de banda de la *CPU* a la memoria principal [56]. A diferencia de las *CPUs*, que están diseñadas para trabajar en general a altas velocidades de reloj (entre 3 GHz y 4 GHz y que repercuten en un alto consumo de energía), los núcleos de la *GPU* funcionan con velocidades de reloj relativamente bajas (1 GHz aproximadamente), lo que al final involucra un ahorro de energía y menor generación de calor [6, 34, 61]. La Figura 3.3 muestra gráficamente la arquitectura general de una *GPU*.

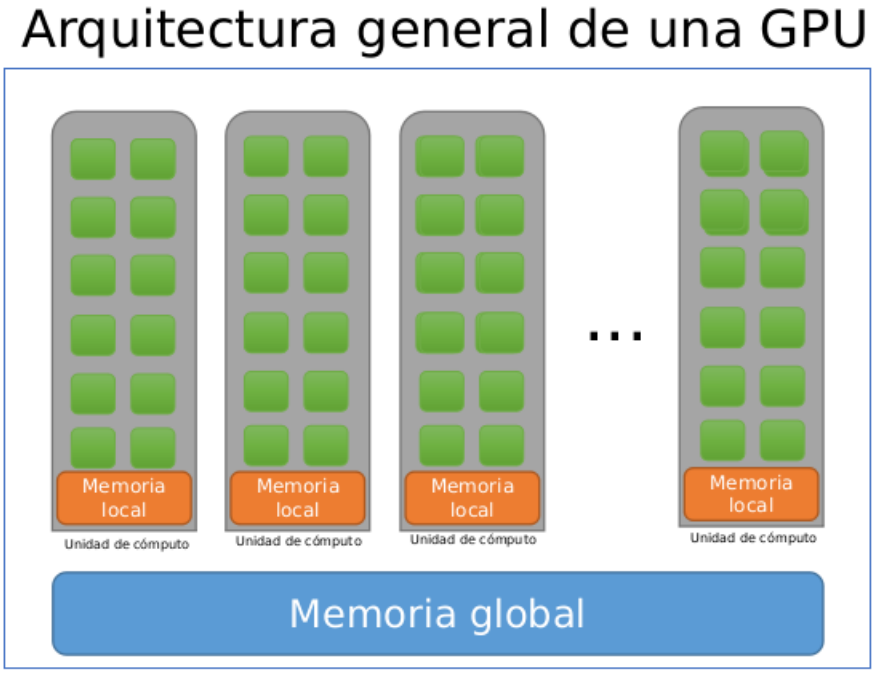

Figura 3.3: Arquitectura general de una *GPU*. Los núcleos son divididos en grupos de igual tamaño, denominados unidades de cómputo. Cada unidad de cómputo tiene una memoria local que es compartida por los núcleos de dicha unidad. A su vez, todas las unidades de cómputo tienen acceso a una memoria global.

Sin embargo, una desventaja de las *GPUs* es que actualmente su arquitectura no permite realizar cómputo multitarea, lo que impide utilizar sus recursos hasta que la tarea que estén ejecutando no haya sido completada; así mismo, las *GPUs* no so-

portan acceso a disco o funcionalidades de red [56]. Otra desventaja más es que la memoria global de una *GPU* suele ser menor a la memoria principal de los sistemas basados en *CPU*. Por estas razones, actualmente se suelen ocupar sistemas híbridos (*CPU*+*GPU*) para la resolución de problemas; aunque una *GPU* fue diseñada principalmente para realizar tareas de procesamiento de gráficos como el *rendering* en tiempo real o la generación de imágenes 3D, se puede utilizar también para resolver otro tipo de problemas en conjunto con la *CPU*, en un enfoque conocido como computación acelerada [6, 34]. La computación acelerada por *GPU* permite asignar parte del trabajo de un programa a la *GPU*, especialmente en donde la computación es más intensiva, mientras que el resto del código se ejecuta en la *CPU* [6]; de este modo, la *GPU* puede aligerar la carga de información que debe de ser procesada por la *CPU*, y en conjunto ambas pueden hacer un trabajo de manera más eficiente [61]. Una representación gráfica de este enfoque se puede ver en la Figura 3.4. Desde la perspectiva del usuario, el programa se ejecuta de forma mucho más rápida.

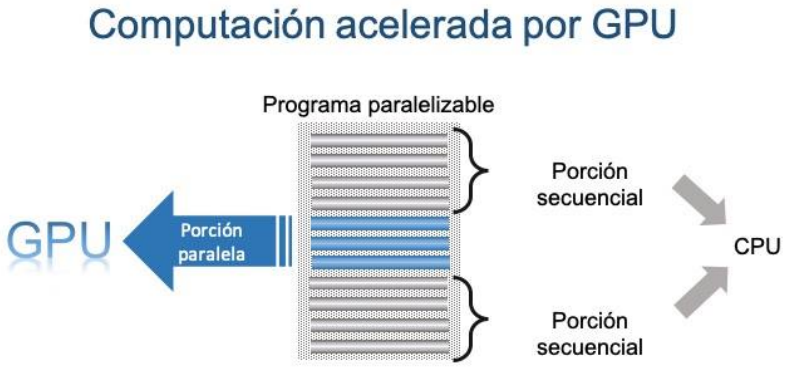

Figura 3.4: Aceleración de una aplicación por *GPU*. Parte del trabajo se envía a la *GPU*, mientras el resto se ejecuta en la *CPU*.

En resumen, un acelerador permite un sistema de bajo costo económico, bajo consumo de energía y alto rendimiento, por lo que su utilización para hacer paralelismo

es factible; hoy en día, muchas computadoras del Top500 (*ranking* de las 500 supercomputadoras con mayor rendimiento del mundo) ya incorporan en sus sistemas tarjetas gráficas para aumentar su rendimiento [5].

Con el uso de aceleradores, en específico de *GPUs*, surge lo que es conocido como cómputo de propósito general en *GPU* (*GPGPU*, por sus siglas en inglés), que, como su nombre sugiere, trata de la resolución de problemas de cómputo aprovechando el potencial de las *GPUs* para ejecutarse en ellas [34]. Una herramienta ampliamente utilizada para realizar *GPGPU* es CUDA™ (siglas de *Compute Unified Device Architecture*), que es una arquitectura de cómputo paralelo que incluye un compilador y un conjunto de herramientas para la programación de *GPUs* [61]; en el ámbito de la segmentación de imágenes con algoritmos de lógica difusa existen trabajos relacionados con esta tecnología, como los presentados en [86, 87]. Sin embargo, CUDA™ sólo se limita a la programación de tarjetas gráficas manufacturadas por la compañia nVidia®, por lo que el presente trabajo se ha concentrado en el uso de otra tecnología de propósito general: OpenCL™.

3.3 ¿Por qué OpenCL™?

OpenCL™ es un estándar para realizar cómputo paralelo que define un conjunto de tipos de datos, estructuras y funciones que aumentan las capacidades del lenguaje C/C++. OpenCL™ se basa en el lema en inglés "*Write once, run on anything*" (escriba una vez, ejecute en cualquier plataforma/dispositivo), esto es, garantiza portabilidad y compatibilidad entre múltiples plataformas [4]; a diferencia de CUDA™, no se limita a ejecutar código en dispositivos de un solo vendedor, sino que puede correr en procesadores Intel®, procesadores AMD®, tarjetas gráficas nVidia®, tarjetas gráficas Radeon®, FPGAs así como en cualquier otro dispositivo que soporte el estándar OpenCL™. Esta es la característica principal por la que se decidió realizar el presente trabajo con esta tecnología, ya que complementa nuestra implementación secuencial escrita en C++ (para añadirle paralelismo) y no se es dependiente de un solo ambiente de ejecución.

Como parte de sus características técnicas, OpenCL^{TM} incluye un procesamiento estandarizado de vectores (debido a que las instrucciones de vectores usualmente son específicas de cada vendedor) y, sobre todo, un funcionamiento orientado a la programación paralela, lo que lo hace ideal para realizar cualquier proceso de paralelización [4, 68]. En general, el funcionamiento de OpenCL^{TM} se basa en tareas llamadas *kernels*, que son funciones específicamente codificadas para que sean ejecutadas en uno o más dispositivos que tengan soporte para el estándar OpenCL^{TM}.

Inicialmente, un programa *anfitrión* (que regularmente es un sistema secuencial programado con el lenguaje C o C++) es el encargado de dividir el trabajo a realizar, ya sea en un enfoque basado en datos o en un enfoque basado en tareas, y construye los *kernels* necesarios para realizar dicho trabajo. Posteriormente, el *anfitrión* accede a un contenedor llamado "contexto", que es el espacio en donde residen todos los dispositivos que soportan el estándar OpenCL^{TM}, para obtener uno o más dispositivos que puedan ejecutar los *kernels*. Finalmente, el *anfitrión* se comunica con cada dispositivo obtenido a través de una "cola de comandos"; una cola de comandos es el mecanismo a través del cual el *anfitrión* le manda al dispositivo los *kernels* a ejecutar, y a su vez mantiene comunicación con él [68]. La Figura 3.5 representa de forma gráfica la arquitectura y el funcionamiento de una aplicación bajo OpenCL^{TM}.

Como se puede apreciar, con el uso de OpenCL^{TM} el paralelismo surge de manera natural, ya sea en el enfoque de paralelismo basado en tareas o en el paralelismo basado en datos; esta es otra justificación más por la que apostamos por utilizar esta tecnología, ya que podemos aprovechar cualquier enfoque de paralelismo a diferencia de las tecnologías mencionadas anteriormente, que manejan un solo enfoque. A su vez, haciendo una analogía con lo explicado en secciones anteriores sobre la arquitectura maestro-esclavo, en un programa OpenCL^{TM} el sistema *anfitrión* actuará como el administrador, mientras que los diferentes dispositivos serán los trabajadores.

Por otro lado, es preciso decir que aunque la versión actual de OpenCL^{TM} es la 2.0, la versión con la que se realizó este trabajo fue la 1.2 (versión inmediatamente anterior a la 2.0 [4]), debido a que ésta es la versión que soportan las computadoras

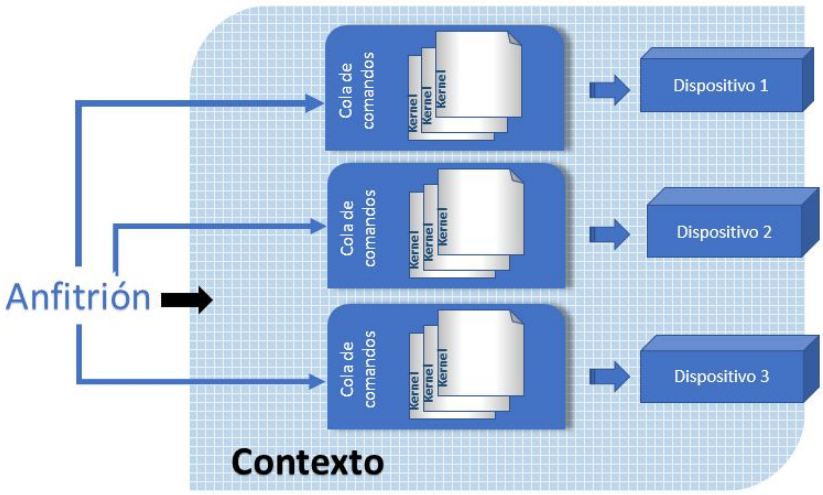

Figura 3.5: Arquitectura de una aplicación bajo OpenCL^{TM}. Los *kernels* se distribuyen hacia los diferentes *dispositivos* vistos por el *anfitrión* en el *contexto*, a través de *colas de comandos*.

que se tienen en nuestro grupo de trabajo. Ésta versión de OpenCL^{TM} tiene algunas desventajas y cuenta con algunas restricciones, mismas que pueden ser consultadas en el Apéndice A; por ejemplo, no se pueden utilizar arreglos de dos o más dimensiones, así como tampoco se tiene soporte para utilizar estruturas de datos dinámicas o complejas, como listas ligadas y árboles. Sin embargo, en este trabajo se han propuesto algunas estrategias para afrontar dichas restricciones.

3.4 Propuesta de paralelización

Retomando el contexto del algoritmo MOFS explicado en el Capítulo 2, podemos darnos cuenta de que su funcionamiento es intrínsecamente paralelo, debido a que el crecimiento de regiones a partir de las semillas seleccionadas por el usuario es independiente por cada clase a segmentar, al menos hasta el punto en el que una región no solape o intersecte a otra (siendo éste el caso en donde se tendría que verificar la región dominante, es decir, aquella con mayor nivel de conectividad); sin embargo, la implementación que se tiene y utiliza actualmente en nuestro grupo de trabajo es secuencial.

Una opción intuitiva de utilizar el paralelismo para rediseñar la implementación secuencial del algoritmo MOFS es dividir el proceso en un enfoque de paralelismo basado en tareas, en donde delegaríamos el crecimiento de cada región de forma independiente a cada uno de los núcleos de la computadora paralela que utilicemos, en donde cada núcleo trabajaría con todos los vóxeles a segmentar. Sin embargo, el factor de optimización al hacer la implementación de esta manera estaría ligado directamente al número M de clases que se quieran segmentar, en donde, de tener sólo dos clases, se alcanzaría (teóricamente) un factor de sólo $2\times$. Una forma más cuidadosa de abordar el problema nos permitiría alcanzar un factor de optimización un poco mayor.

La forma en la que se decidió realizar el proceso de paralelización del algoritmo MOFS fue analizar la existencia de dos tareas principales que se realizan en el proceso de segmentación, la primera de ellas corresponde al cómputo de los valores de conectividad desde las semillas elegidas por el usuario hacia los demás vóxeles de la imagen, y la segunda corresponde al cómputo de los valores de afinidad necesarios para dicho cálculo de conectividades. Mientras que para el cálculo de conectividad establecido por el punto (i) en el Teorema 2.1 se utiliza una implementación dinámica del algoritmo de Dijkstra [29], el valor de afinidad calculado por la ecuación (2.6) (y utilizado en las líneas 14 y 17 del Algoritmo 1) se basa en el cálculo local de afinidad de un vóxel con respecto a su vecindario, para alguna clase m. Debido a este principio de localidad, es preciso notar que el cómputo de los valores de afinidad es independiente para todos los vóxeles de la imagen, es decir, el cómputo de la afinidad en un vóxel $\boldsymbol{u}$ no depende del cómputo de la afinidad en un vóxel $\boldsymbol{v}$, aún si dichos vóxeles tienen vecinos en común, debido a que el cómputo de la afinidad no modifica los valores originales de los vóxeles de la imagen. Podemos aprovechar esta independencia entre vóxeles para realizar un enfoque de paralelismo basado en datos, en donde encarguemos la tarea de realizar el cómputo de los valores de afinidad a los núcleos que tengamos disponibles, cada uno de ellos trabajando sobre un subconjunto de los vóxeles de la imagen, y almacenar este cálculo en una tabla de consulta/búsqueda o *lookup table* (LUT) [29] que sea consultada cuando sea requerido.

3.5 Uso de memoria de la *lookup table*

Una de las desventajas de usar paralelismo es el consumo de memoria ligado a las estructuras de datos que se tengan que utilizar para su ejecución [33, 36, 59]. En la propuesta que hemos descrito en la sección anterior, la utilización de una LUT permite ahorrar el tiempo de procesamiento que se requiere para calcular los valores de afinidad durante el proceso de segmentación, pero a costa de un mayor consumo de memoria, al almacenar dichos valores de afinidad. El uso de memoria para esta estructura de datos en el algoritmo MOFS estará directamente relacionado con el número de vóxeles de la imagen que estemos segmentando, con el vecindario considerado para un vóxel y con el número de clases en las que se quiera segmentar la imagen. Para un vóxel $\boldsymbol{u}$ dado, sólo utilizaremos su 6-vecindario. Así mismo, el número de clases M estará limitado en este trabajo a 5 (en otros proyectos de nuestro grupo de trabajo se ha utilizado una cantidad menor o igual a esta [22, 23, 36, 55]).

Si consideramos que el número de niveles de conectividad está limitado a 1000 elementos como se explicó en el Capítulo 2, sólo serán necesarios 10 bits para almacenar un valor de afinidad de un vóxel hacia alguno de sus vecinos, debido a que el nivel máximo de afinidad que puede haber, 1000, puede ser representado con la secuencia de bits 1111101000b. Si consideramos los 6 vecinos de un vóxel, podemos entonces almacenar en un dato de 64 bits (*unsigned long* en nuestra implementación en C++) los 6 niveles de afinidad (60 bits), "desperdiciando" 4 bits en este proceso de codificación, pero teniendo en un solo dato la información de afinidad necesaria de un vóxel con su vecindario, para una sola clase m; esto se puede apreciar de forma gráfica en la Figura 3.6. La memoria utilizada en bits estará dada entonces por la ecuación

$$memoria_{lookuptable} = númeroVóxeles \times 64 \times M\,[bits]. \tag{3.1}$$

Si bien esta codificación consume y debe mantener aproximadamente 900 MB de memoria para alamacenar la información de afinidad de una imagen de $60,000,000$ de vóxeles, ayudará a optimizar el cálculo de conectividad durante el proceso de

Figura 3.6: Codificación de valores de afinidad para un vóxel con respecto a sus 6 vecinos. Se puede apreciar que en un dato del tipo *unsigned long*, se pueden almacenar los 6 niveles de afinidad, ocupando cada uno de ellos 10 bits, desperdiciando los 4 bits restantes.

segmentación, y no será necesario almacenar en datos independientes los niveles de afinidad (por ejemplo, en seis datos del tipo `int`, es decir, un dato por cada vecino).

Un caso especial que se debe de considerar en esta codificación es el del vecindario de los vóxeles en los bordes, ya que por definición, los vóxeles pertenecientes a los bordes no tienen vecinos más allá de las dimensiones de la imagen; se decidió que el valor de afinidad correspondiente a un vecino fuera de rango fuése el de 1023 ($1111111111b$ en secuencia de bits), y posteriormente, en el proceso de segmentación, no procesar los vecinos con este valor de afinidad.

3.6 Aplanado de las estructuras de datos

Una de las principales desventajas mencionadas en el Apéndice A del lenguaje OpenCL^{TM} es el uso restringido de apuntadores, ya que impide manejar estructuras bidimensionales (matrices) o de más dimensiones. Considerando que este trabajo utiliza imágenes 3D y que su manejo en la implementación secuencial involucra el uso de apuntadores triples (operador `***`, en el lenguaje C), la estrategia para manejar estas imágenes 3D en un *kernel* de OpenCL^{TM} consistió en aplanar dichos volúmenes y manejarlos a través de un solo apuntador (operador `*`, dentro de un *kernel* de OpenCL^{TM}). Si *width*, *height* y *depth* son las dimensiones de nuestra imagen en ancho, alto y profundidad, respectivamente, el Algoritmo 2 muestra la forma básica de aplanar las imágenes 3D y almacenarlas en un arreglo unidimensional de tamaño $width \times height \times depth$. Cabe mencionar que el proceso de aplanado se realiza dentro del proceso de lectura de la imagen 3D, por lo que no requiere tiempo de cómputo adicional para su elaboración. La desventaja de aplanar el volúmen

nuevamente es el consumo de memoria, ya que debemos mantener en ella tanto la imagen original como la imagen aplanada, al menos durante la parte de inicialización o lectura de la imagen 3D con la que vamos a trabajar.

Algoritmo 2 Algoritmo para aplanar una imagen 3D.

```
1. flattenArray = nuevo array[width × height × depth]
2. auxCounter = 1
3. para z ← 1 a depth hacer
4.     para y ← 1 a height hacer
5.         para x ← 1 a width hacer
6.             flattenArray[auxCounter] = imagen[x][y][z]
6.             auxCounter = auxCounter + 1;
```

La memoria necesaria para alamacenar el volúmen aplanado estará dada por la siguiente fórmula, considerando que en un dato de tipo `int` (32 bits) se puede almacenar el valor escalar de un vóxel de la imagen

$$memoria_{flattenArray} = númeroVóxeles \times 32\,[bits]. \tag{3.2}$$

Una vez que se ha realizado el aplanado de la imagen 3D, el índice para acceder al valor escalar de uno de sus vóxeles dentro de la nueva estrutura unidimensional está determinado por la ecuación

$$índiceU = coordX + (coordY \times width) + (coordZ \times width \times height), \tag{3.3}$$

en donde $coordX$, $coordY$ y $coordZ$ representan la posición 3D del vóxel que se desea analizar.

Teniendo en cuenta lo explicado en este apartado, la LUT descrita en la sección anterior igualmente debe de estar aplanada para su uso dentro de un *kernel*, sin embargo, la forma de acceso a esta estructura difiere un poco con respecto al índice de la ecuación (3.3). Debido a que la LUT soportará los valores de afinidad de todos los vóxeles con respecto a sus vecinos, para cualquier clase m, se debe de considerar

la clase m que se quiera analizar para calcular el índice correspondiente; dicho de otra forma, la clase m determinará el desplazamiento requerido con respecto al tamaño total de la imagen (número de vóxeles) para acceder a los valores de afinidad de un vóxel dado. La ecuación para calcular el índice de un vóxel en la LUT con respecto a una clase m es

$$índiceLUT = índiceU + (m \times númeroVóxeles), \tag{3.4}$$

en donde $númeroVóxeles$ es la cantidad total de vóxeles de la imagen 3D, m es la clase que se quiere analizar e $índiceU$ es el índice calculado por la ecuación (3.3).

3.7 Paso de parámetros a un *kernel*

Una vez que se han decidido y construido las estructuras de datos con las que va a operar un *kernel*, otro paso importante dentro de una aplicación OpenCL^{TM} es el paso de parámetros desde la aplicación *anfitrión* hacia el *kernel*; en nuestro trabajo no solo se necesitan enviar las estructuras de datos explicadas en este capítulo, sino que también se requieren otro tipo de datos para trabajar dentro del *kernel*, como lo son el tamaño de la imagen (*width*, *height* y *depth*), el número de clases m así como los valores estadísticos de las ecuaciones (2.8) y (2.9) para el cálculo del valor de afinidad (que son inicialmente calculados por el *anfitrión*).

Se ha explicado que la forma de comunicación que tiene el *anfitrión* con un dispositivo que ejecutará un *kernel* es a través de una cola de comandos. Sin embargo, el paso de parámetros a través de la cola de comandos requiere ejecutar la siguiente secuencia de acciones para cada dato que se quiera enviar:

1. Se inicializa el dato que se quiere pasar como parámetro, desde el lado del *anfitrión*.

2. Se obtiene la cola de comandos asociada con el dispositivo que ejecutará el *kernel*.

3. Se informa al *kernel* el tamaño en bytes del dato que se le va a enviar como parámetro.

4. Se crea un objeto propio del lenguaje OpenCL^{TM} , en el cual se almacena el parámetro a enviar y que se utiliza también para reservar e inicializar memoria en el dispositivo que ejecutará el *kernel*.

5. Se envía el parámetro al *kernel* con las funciones propias del lenguaje OpenCL^{TM}.

6. Se recibe el parámetro en el *kernel* como un apuntador (*).

7. Se decodifica en el *kernel* el apuntador recibido, para transformarlo al tipo de dato que se quiera manipular.

Como se puede apreciar, ésta secuencia de acciones es sencilla, sin embargo, el tener que realizarla por cada parámetro que se envía al *kernel* puede ser un proceso laborioso y extenso respecto al número de líneas de programación requeridas para ello, aunado a la obtención inicial del contexto OpenCL^{TM}, la búsqueda del dispositivo adecuado para ejecutar el *kernel* y la compilación del mismo *kernel* (desde el *anfitrión*) para su ejecución en el dispositivo.

La estrategia para el paso de parámetros en nuestra implementación fue la de utilizar las capacidades del lenguaje OpenCL^{TM} para la construcción de un *kernel*, en donde es posible incluir bibliotecas mediante la directiva *#include* (una directiva de preprocesamiento del compilador de C/C++). Gracias a esto, se pueden construir archivos que contengan las variables necesarias para la ejecución del *kernel*, sin necesidad de pasar dichas variables como parámetros mediante la secuencia de acciones descrita anteriormente.

3.8 Arquitectura del sistema y función *clEnqueueNDRangeKernel*

Para la ejecución de la propuesta paralela descrita anteriormente se decidió utilizar la arquitectura que se muestra en la Figura 3.7. El programa principal (*main*)

seguirá teniendo la responsabilidad de leer la imagen 3D con la que se trabajará, así como inicializar las variables correspondientes, por ejemplo, el cómputo de los valores estadísticos (a partir de las semillas elegidas por el usuario) para el cálculo de los valores de afinidad. Posteriormente, se pasarán todos los datos a una clase controladora, nombrada "*ParallelManager*", que será la encargada de controlar la comunicación con la *GPU* así como de la construcción de los *kernels* y los archivos compartidos (bibliotecas) descritos en la sección anterior. A su vez, *ParallelManager* hará uso de una clase llamada *GPUObject*, que será la encargada de obtener el contexto OpenCL^{TM}, el dispositivo *GPU* en donde se ejecutará algún *kernel* y la cola de comandos correspondiente.

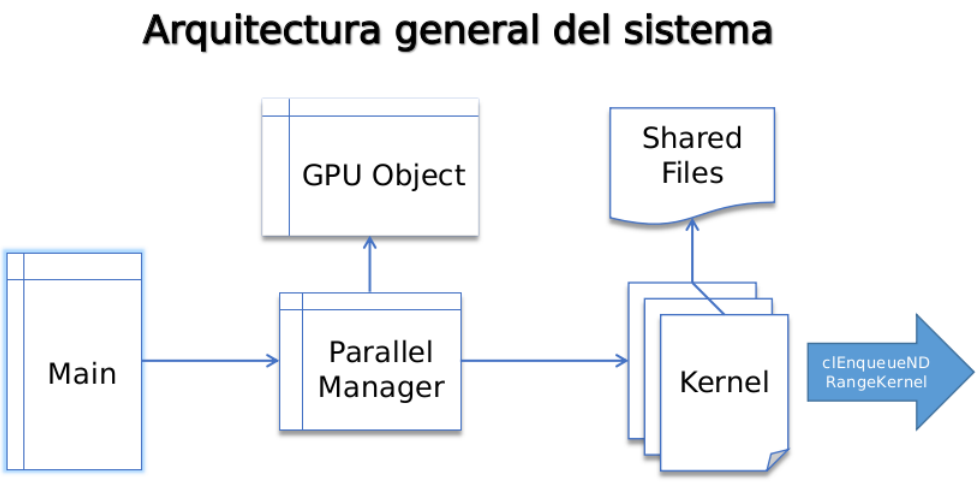

Figura 3.7: Arquitectura general del sistema, en donde se establece una clase controladora (*ParallelManager*) para el manejo de recursos y comunicación con la *GPU*.

Finalmente, para enviar y ejecutar el *kernel* en la *GPU*, se hizo uso de la función *clEnqueueNDRangeKernel* del lenguaje OpenCL^{TM}. Dicha función permite poner un *kernel* en la cola de comandos para su ejecución en la *GPU*; una de sus principales características es que permite indicar cómo se distribuirá el trabajo (la ejecución del *kernel*) a través de los núcleos de la *GPU*, es decir, permite controlar el número de hilos de ejecución que se crearán así como el número de dimensiones con las cuales se está trabajando. Cada hilo de

ejecución que se genere ejecutará una instancia del *kernel*, bajo algún índice en particular que lo distinga de los demás; se puede aprovechar este principio para realizar el enfoque de paralelismo basado en datos de nuestra propuesta, en donde de acuerdo al índice(s) propio del hilo de ejecución se manipule la parte correspondiente de nuestra estructura de datos aplanada.

Aprovechando las opciones de la función *clEnqueueNDRangeKernel*, podemos indicar que queremos trabajar en el espacio de 3 dimensiones (x,y,z), y que el número de hilos de ejecución que se tienen que crear estará determinado por el conjunto $\{width, height, depth\}$; definiendo la ejecución de esta forma, se aprovechará el máximo potencial de la *GPU* para ejecutar el *kernel* de cómputo de afinidades, cuyo funcionamiento se muestra en el Algoritmo 3, ya que se crearán $width \times height \times depth$ hilos de ejecución, es decir, un hilo de ejecución por cada vóxel a procesar. Cabe destacar que como el cómputo de afinidad en un vóxel es independiente al de los demás vóxeles, no se requieren barreras de sincronización que controlen los hilos de ejecución, sólo basta poner una barrera general para esperar la finalización de la ejecución total del *kernel*.

Por último, la clase *ParallelManager* lee los resultados de ejecución del *kernel* (es decir, los datos de la LUT que se calcularon de forma paralela) y los devuelve a la clase *main*, que será la encargada de utilizarlos en la tarea del cómputo de conectividades del algoritmo MOFS y generar los resultados correspondientes.

Algoritmo 3 Algoritmo del *kernel* de cómputo de afinidades.

```
1. coordX = índice(1)
2. coordY = índice(2)
3. coordZ = índice(3)
4. índiceU = coordX + ( coordY × width ) + ( coordZ × width × height)
5. vóxel u = flattenArray[índiceU]
6. Vecinos V = vecinos(u)
7. para m ← 1 a M hacer
8.     índiceLUT = índiceU + ( m × númeroVóxeles)
9.     para cada v en V hacer
10.        afinidad = calculaAfinidad(u,v)
11.        almacenaEnLUT(afinidad,índiceLUT)
```

Capítulo 4

Experimentos y resultados

En el presente trabajo se buscó explorar la utilidad de realizar el proceso de paralelización descrito en el Capítulo 3, y ver el factor de optimización (respecto al tiempo) que se puede alcanzar en la ejecución del algoritmo MOFS.

Gracias al uso del lenguaje OpenCL^{TM} y su flexibilidad de poder ejecutarse bajo cualquier plataforma que soporte el estándar, se pudieron realizar experimentos en dos ambientes de ejecución. El primero de ellos, que llamaremos de aquí en adelante ambiente A1, fue una computadora con procesador Intel Core i5® a 2.50 GHz, 16 GB de memoria RAM y una tarjeta gráfica Intel HD Graphics 3000®, que cuenta con un reloj de 850 MHz y memoria compartida con el sistema anfitrión; el segundo ambiente de ejecución, que llamaremos de aquí en adelante ambiente A2, fue una computadora con procesador Intel Core i5® a 3.20 GHz, 32 GB de memoria RAM y una tarjeta gráfica nVida GTX TITAN X®, que cuenta con un reloj de 1000 MHz y 12 GB de memoria interna.

4.1 Conjuntos de imágenes

Para nuestros experimentos se utilizaron tres conjuntos de datos, cortesía de [36], que representan volúmenes de microscopía electrónica tridimensional (extensión de archivo MRC [36]), y que llamaremos MRC1, MRC2 y MRC3, por simplicidad. Las dimensiones y cantidad de vóxeles de estos conjuntos de datos se muestran en la Tabla 4.1; el valor escalar de cada vóxel representa un solo valor de intensidad y está almacenado en un dato de tipo int de 32 bits. Así mismo, el conjunto de semillas para cada volúmen fue proporcionado por [36], en donde se definen las semillas solamente

de dos clases a discriminar (el fondo y un objeto de interés).

Conjunto	Ancho	Alto	Profundidad	Vóxeles
MRC1	310	860	186	49,587,600
MRC2	464	862	141	56,395,488
MRC3	359	764	245	67,197,620

Figura 4.1: Dimensiones de los conjuntos de datos ocupados en este trabajo.

En la Figura 4.2 se pueden apreciar cortes representativos (imágenes 2D) de los volúmenes MRC utilizados en este trabajo.

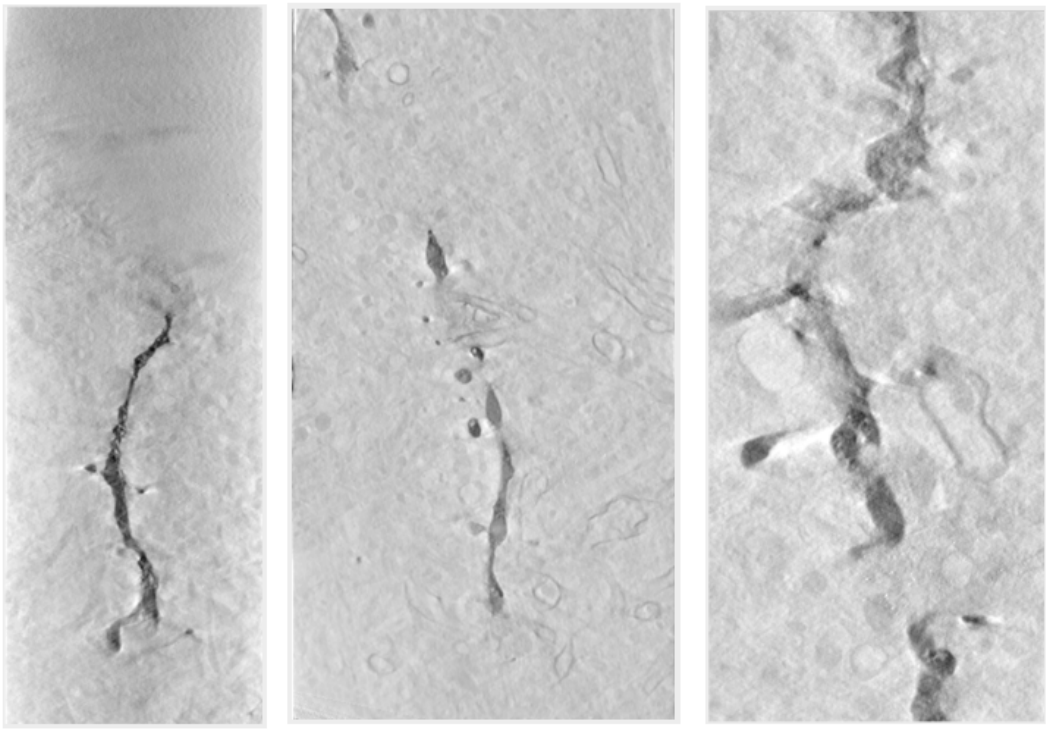

Figura 4.2: Imágenes de los volúmenes MRC, cada imagen representa un solo corte de su volúmen correspondiente; en (a), volúmen MRC1, en (b), volúmen MRC2, en (c), volúmen MRC3.

Se ejecutó la implementación paralela propuesta en este trabajo sobre los tres conjuntos de datos así como la implementación serial, para la comparación de resultados, en los dos ambientes de ejecución mencionados al inicio de este capítulo.

4.2 Ambiente A1

Hay que recordar que en el proceso de segmentación por el algoritmo MOFS analizamos la existencia de dos tareas principales, el cómputo local de afinidades y el proceso de cálculo de conectividad (en donde se utilizan las afinidades locales). Se realizó primeramente la ejecución y medición del tiempo en segundos del cómputo de afinidades, tanto en la implementación serial que se tenía actualmente en nuestro grupo de trabajo como en la implementación paralela que se desarrolló. Se llevaron a cabo diez experimentos en ambos casos, para los tres conjuntos de datos. Los tiempos medidos para los conjuntos MRC1, MRC2 y MRC3 se presentan en las Tablas 4.3, 4.4 y 4.5, respectivamente.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	138	13
2	142	14
3	138	13
4	140	15
5	140	15
6	141	14
7	137	14
8	137	13
9	137	13
10	136	14

Figura 4.3: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC1.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	157	16
2	156	16
3	157	16
4	155	16
5	156	16
6	160	16
7	159	16
8	154	16
9	160	16
10	155	16

Figura 4.4: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC2.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	189	19
2	187	19
3	187	18
4	188	18
5	187	18
6	187	18
7	189	18
8	188	18
9	191	18
10	187	18

Figura 4.5: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC3.

En las Figuras 4.6, 4.7 y 4.8 se pueden apreciar los gráficos correspondientes a las Tablas 4.3, 4.4 y 4.5, respectivamente. Finalmente, la Figura 4.9 muestra el promedio de los tiempos de ejecución de los diez experimentos, por cada conjunto de datos, así como el factor de optimización alcanzado en cada uno de ellos, comparando

el tiempo de ejecución de la implementación serial con el tiempo de ejecución de la implementación paralela.

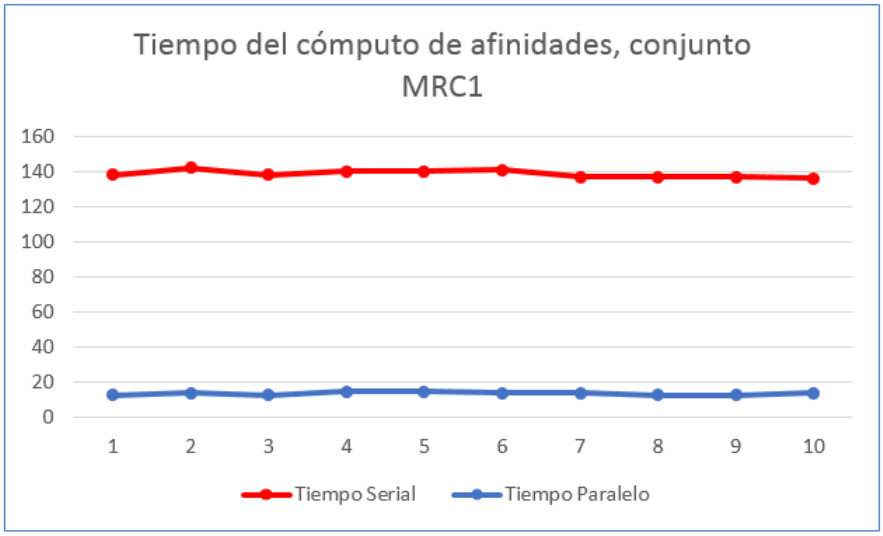

Figura 4.6: Gráfico correspondiente a la tabla mostrada en la Figura 4.6.

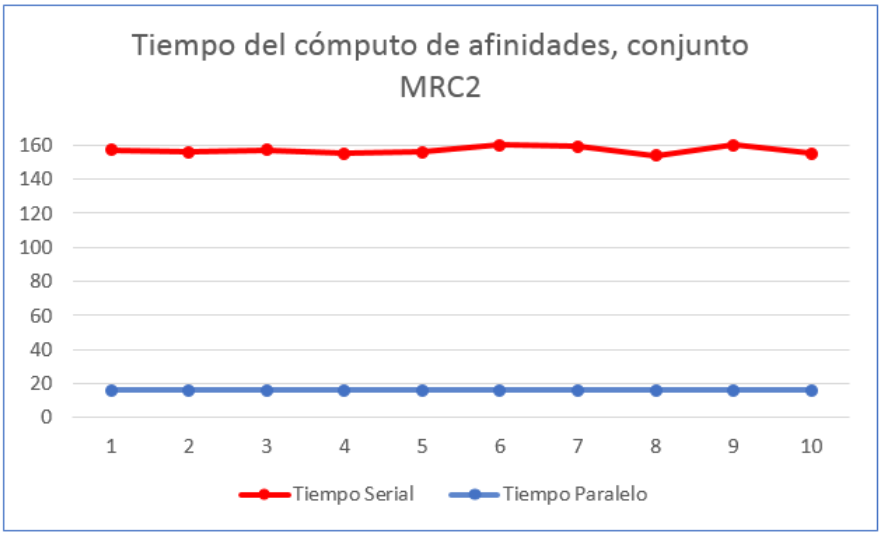

Figura 4.7: Gráfico correspondiente a la tabla mostrada en la Figura 4.7.

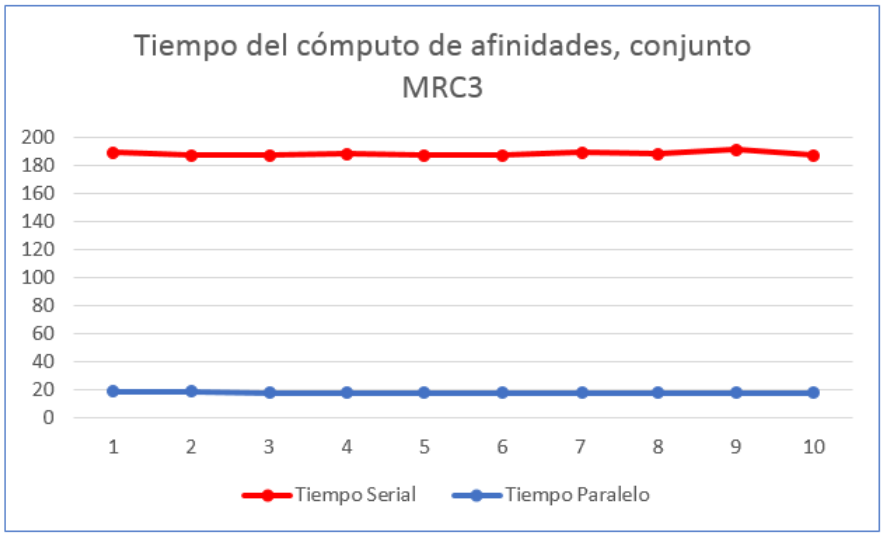

Figura 4.8: Gráfico correspondiente a la tabla mostrada en la Figura 4.8.

Conjunto	Tiempo Serial	Tiempo Paralelo	Optimización
MRC1	138.6	13.8	10.04x
MRC2	156.9	16	9.80x
MRC3	188	18.2	10.32x

Figura 4.9: Promedio en tiempo de la ejecución de los experimentos del cómputo de afinidades, así como el factor de optimización alcanzado.

Después de la ejecución de los experimentos anteriores, se realizó la ejecución y medición de tiempos del algoritmo de segmentación MOFS en su totalidad, es decir, abarcando tanto la tarea del cómputo de afinidades como el cálculo de conectividad. La ejecución serial se realizó con la implementación existente en nuestro grupo de trabajo, mientras que la ejecución paralela abarcó solamente el cómputo de afinidades, para utilizarlas posteriormente en el cálculo serial de conectividades; se puede apreciar que con tan solo hacer la paralelización de la primera tarea se redujeron los tiempos de ejecución totales. Los tiempos medidos de la ejecución del algoritmo MOFS para

los conjuntos MRC1, MRC2 y MRC3 se presentan en las Tablas 4.10, 4.11 y 4.12, respectivamente.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	613	259
2	605	258
3	631	258
4	614	257
5	610	256
6	621	256
7	620	257
8	612	260
9	605	261
10	606	257

Figura 4.10: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC1.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	684	292
2	716	299
3	702	290
4	701	292
5	704	290
6	697	291
7	687	290
8	698	295
9	707	285
10	712	289

Figura 4.11: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC2.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	907	355
2	912	355
3	900	353
4	885	359
5	911	353
6	923	355
7	892	366
8	894	353
9	890	366
10	899	367

Figura 4.12: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC3.

Así mismo, en las Figuras 4.13, 4.14 y 4.15 se pueden apreciar los gráficos correspondientes a las Tablas 4.10, 4.11 y 4.12, respectivamente. Finalmente, la Figura 4.16 muestra el promedio de los tiempos de ejecución de los diez experimentos, por cada conjunto de datos, así como el factor de optimización alcanzado en cada uno de ellos.

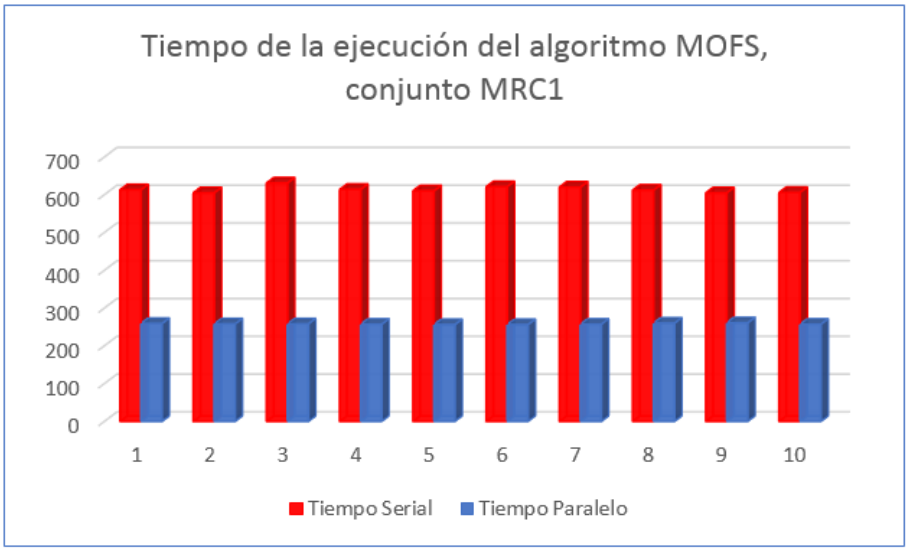

Figura 4.13: Gráfico correspondiente a la tabla mostrada en la Figura 4.10.

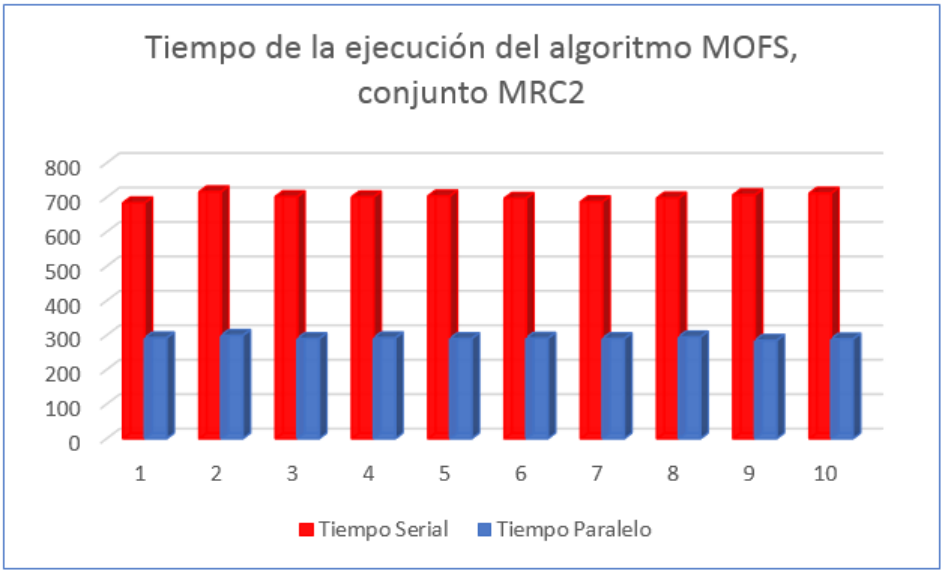

Figura 4.14: Gráfico correspondiente a la tabla mostrada en la Figura 4.11.

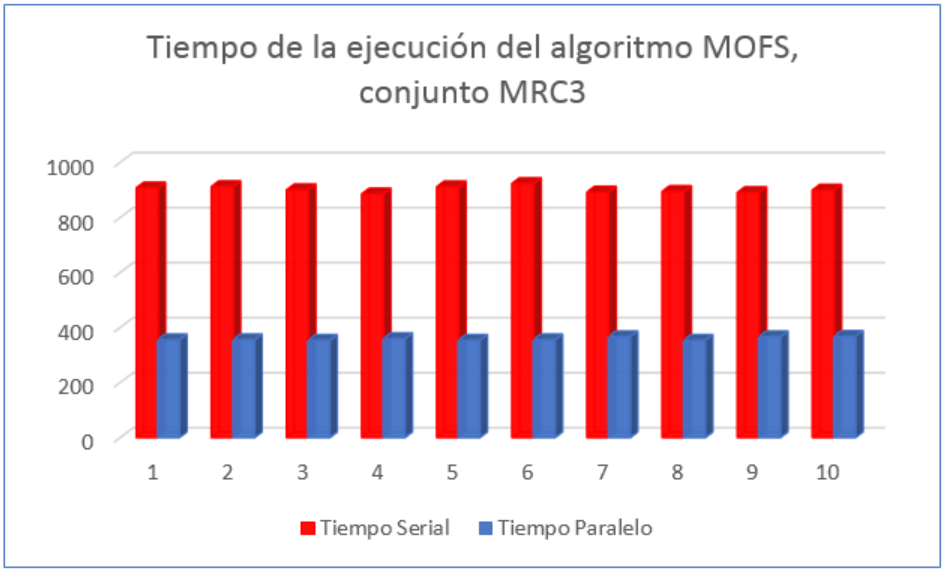

Figura 4.15: Gráfico correspondiente a la tabla mostrada en la Figura 4.12.

Conjunto	Tiempo Serial	Tiempo Paralelo	Optimización
MRC1	613.7	257.9	2.37x
MRC2	700.8	291.3	2.40x
MRC3	901.3	358.2	2.51x

Figura 4.16: Promedio en tiempo de la ejecución de los experimentos de la segmentación por el algoritmo MOFS, así como el factor de optimización alcanzado.

4.3 Ambiente A2

Para el ambiente A2 se realizaron los mismos experimentos que para el ambiente A1, realizando primero la ejecución y medición de tiempos del cómputo de afinidades (tanto en la implementación serial como en la implementación paralela) y después la ejecución y medición de tiempos del algoritmo de segmentación MOFS en su totalidad. Las Figuras 4.17, 4.18 y 4.19 muestran los tiempos medidos para los conjuntos MRC1, MRC2 y MRC3, respectivamente.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	54	8
2	54	8
3	54	8
4	54	8
5	54	8
6	54	8
7	54	8
8	54	8
9	54	8
10	54	8

Figura 4.17: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC1.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	61	9
2	61	9
3	61	9
4	61	9
5	61	9
6	61	9
7	61	9
8	61	9
9	61	9
10	61	9

Figura 4.18: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC2.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	73	11
2	73	11
3	73	11
4	73	11
5	73	11
6	73	11
7	73	11
8	73	11
9	73	11
10	73	11

Figura 4.19: Medición y comparación de los tiempos de ejecución (en segundos) del proceso de cálculo de afinidades, para el conjunto MRC3.

Así mismo, en las Figuras 4.20, 4.21 y 4.19 se pueden apreciar los gráficos correspondientes a las Tablas 4.17, 4.18 y 4.19, respectivamente. Finalmente, la Figura 4.23 muestra el promedio de los tiempos de ejecución de los diez experimentos, por cada conjunto de datos, así como el factor de optimización alcanzado en cada uno de ellos.

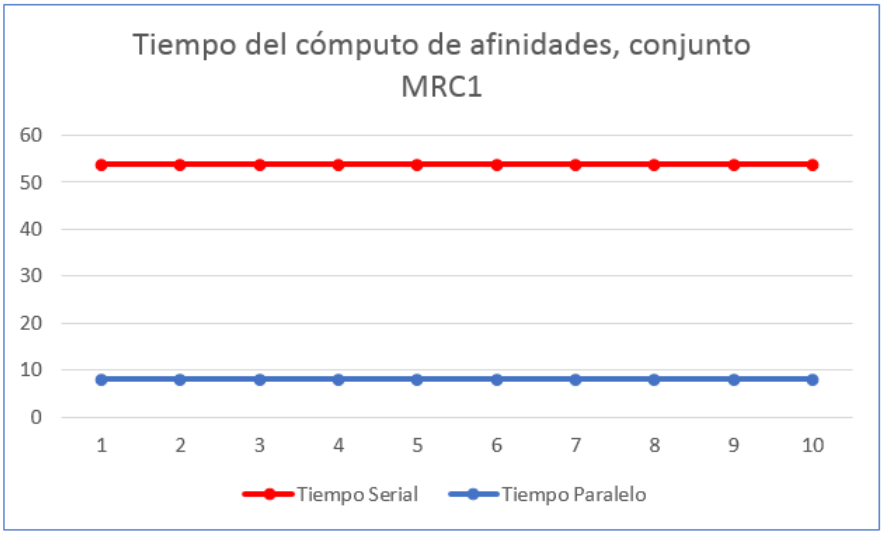

Figura 4.20: Gráfico correspondiente a la tabla mostrada en la Figura 4.17.

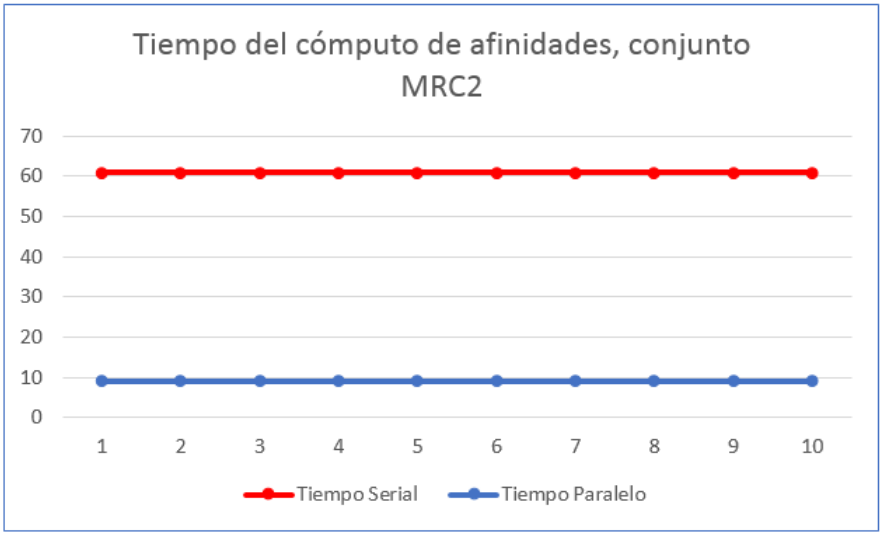

Figura 4.21: Gráfico correspondiente a la tabla mostrada en la Figura 4.18.

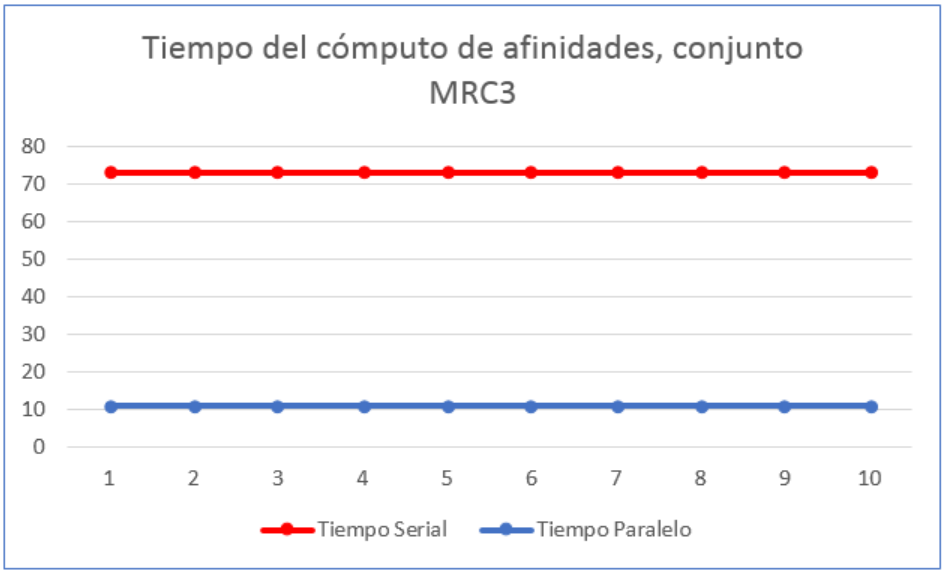

Figura 4.22: Gráfico correspondiente a la tabla mostrada en la Figura 4.19.

Conjunto	Tiempo Serial	Tiempo Paralelo	Optimización
MRC1	54	8	6.75x
MRC2	61	9	6.77x
MRC3	73	11	6.63x

Figura 4.23: Promedio en tiempo de la ejecución de los experimentos del cómputo de afinidades, así como el factor de optimización alcanzado.

Las Figuras 4.24, 4.25 y 4.26 muestran los tiempos medidos para los conjuntos MRC1, MRC2 y MRC3, respectivamente, para la ejecución del algoritmo de segmentación MOFS en su totalidad, teniendo en cuenta las consideraciones descritas en los experimentos del ambiente A1.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	389	187
2	388	191
3	392	186
4	388	186
5	390	188
6	390	185
7	389	191
8	391	187
9	389	186
10	389	188

Figura 4.24: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC1.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	444	211
2	443	211
3	442	218
4	444	212
5	444	214
6	443	211
7	441	212
8	443	211
9	444	211
10	444	212

Figura 4.25: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC2.

N° de experimento	Tiempo Serial	Tiempo Paralelo
1	536	260
2	539	262
3	540	260
4	538	260
5	538	260
6	536	261
7	537	261
8	537	260
9	536	259
10	539	259

Figura 4.26: Medición y comparación de los tiempos de ejecución (en segundos) de la ejecución del algoritmo MOFS, para el conjunto MRC3.

Finalmente en las Figuras 4.27, 4.28 y 4.26 se pueden apreciar los gráficos correspondientes a las Tablas 4.24, 4.25 y 4.26, respectivamente, mientras que en la Figura 4.30 muestra el promedio de los tiempos de ejecución de los diez experimentos, por cada conjunto de datos, así como el factor de optimización alcanzado en cada uno de ellos.

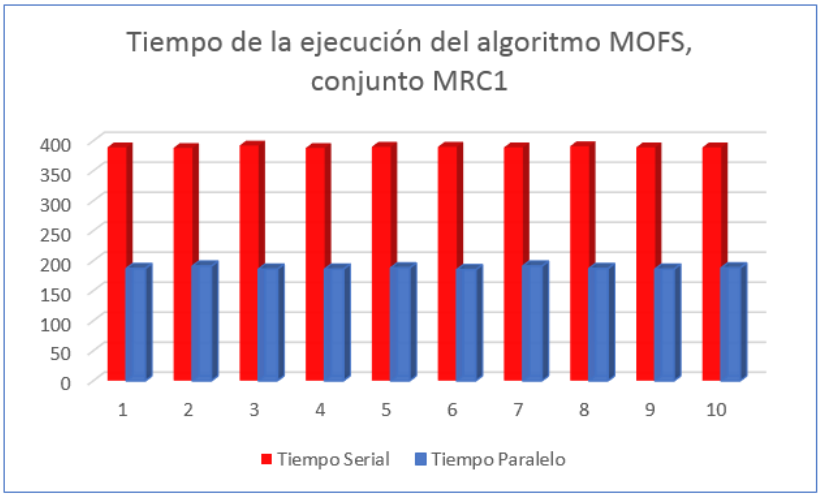

Figura 4.27: Gráfico correspondiente a la tabla mostrada en la Figura 4.24.

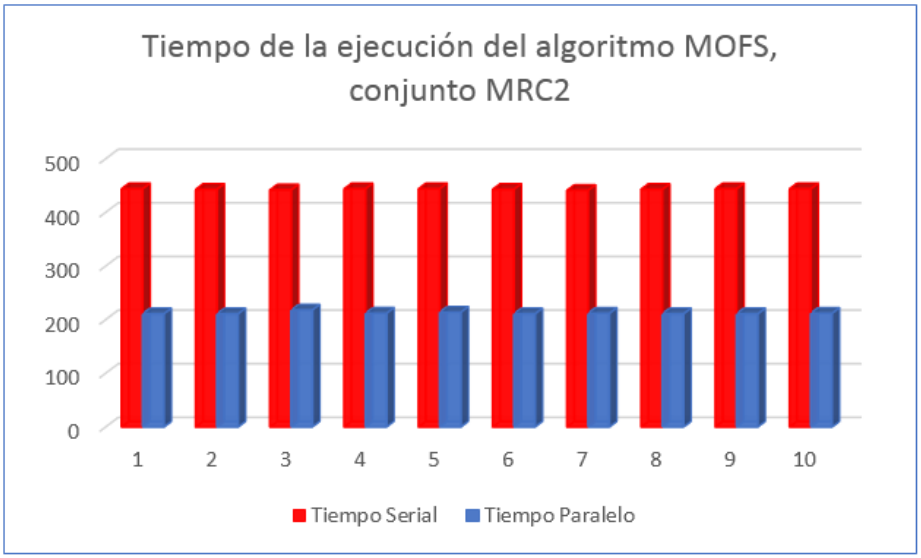

Figura 4.28: Gráfico correspondiente a la tabla mostrada en la Figura 4.25.

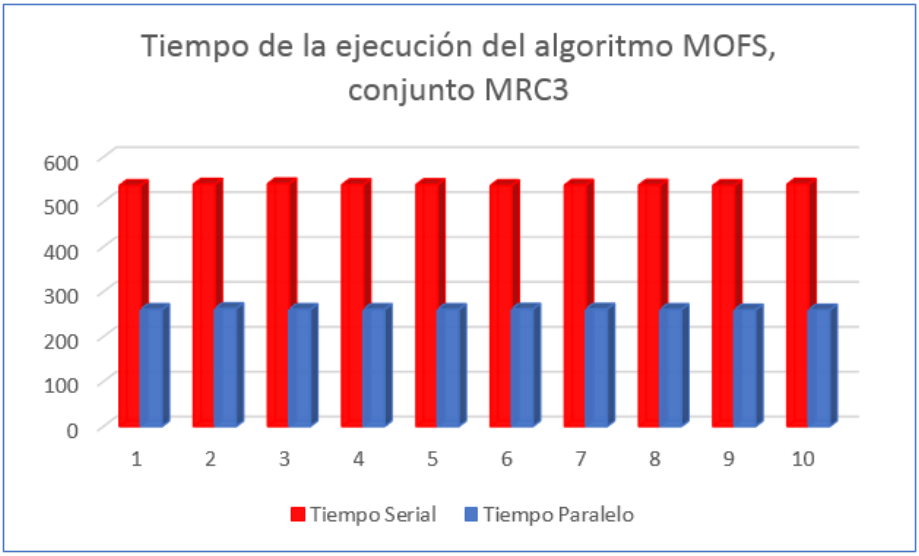

Figura 4.29: Gráfico correspondiente a la tabla mostrada en la Figura 4.26.

Conjunto	Tiempo Serial	Tiempo Paralelo	Optimización
MRC1	389.5	187.5	2.07x
MRC2	443.2	212.3	2.08x
MRC3	537.6	260.2	2.06x

Figura 4.30: Promedio en tiempo de la ejecución de los experimentos de la segmentación por el algoritmo MOFS, así como el factor de optimización alcanzado.

4.4 Validación de resultados

Por último, se evaluaron los resultados obtenidos de realizar el proceso de segmentación mediante el algoritmo MOFS sobre los tres conjuntos de datos, tanto con la implementación serial como con la implementación paralela en los ambientes de ejecución A1 y A2.

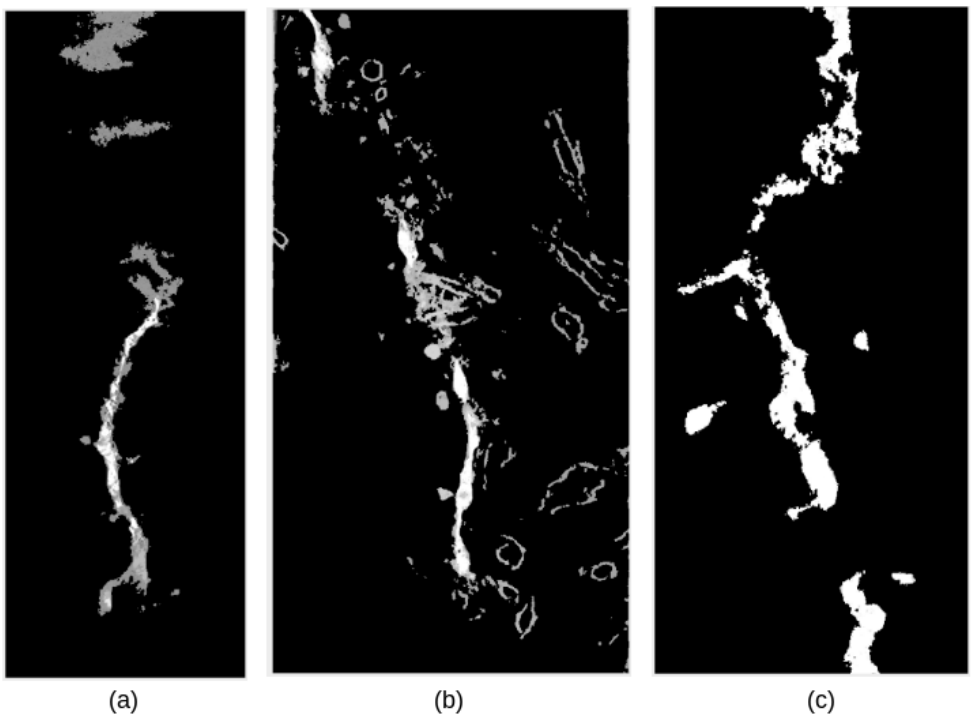

Figura 4.31: Segmentación de los volúmenes MRC1, MRC2 y MRC3, respectivamente, mediante la implementación serial del algoritmo MOFS.

El algoritmo de segmentación MOFS genera como resultado el mapa de conectividad descrito en el Capítulo 2; el mapa de conectividad generado por la implementación serial fue considerado como nuestro *Ground-Truth*, debido a que ésta implementación se ha utilizado en diversos proyectos de nuestro grupo de trabajo y sus resultados han sido satisfactorios. No se encontraron diferencias al comparar elemento por elemento el mapa de conectividad generado por la implementación paralela y el mapa de conectividad generado por la implementación serial. La Figura 4.31 muestra visualmente los resultados del proceso de segmentación sobre los conjuntos de datos MRC1, MRC2 y MRC3, generados por la implementación serial, en los que se distingue un objeto de interés (en tonalidades de gris) de un fondo (en negro); la Figura 4.32 muestra los resultados correspondientes a la implementación paralela. Los cortes presentados en cada Figura son equivalentes a los cortes presentados en la Figura 4.2, con lo cual se puede apreciar que los resultados son los esperados.

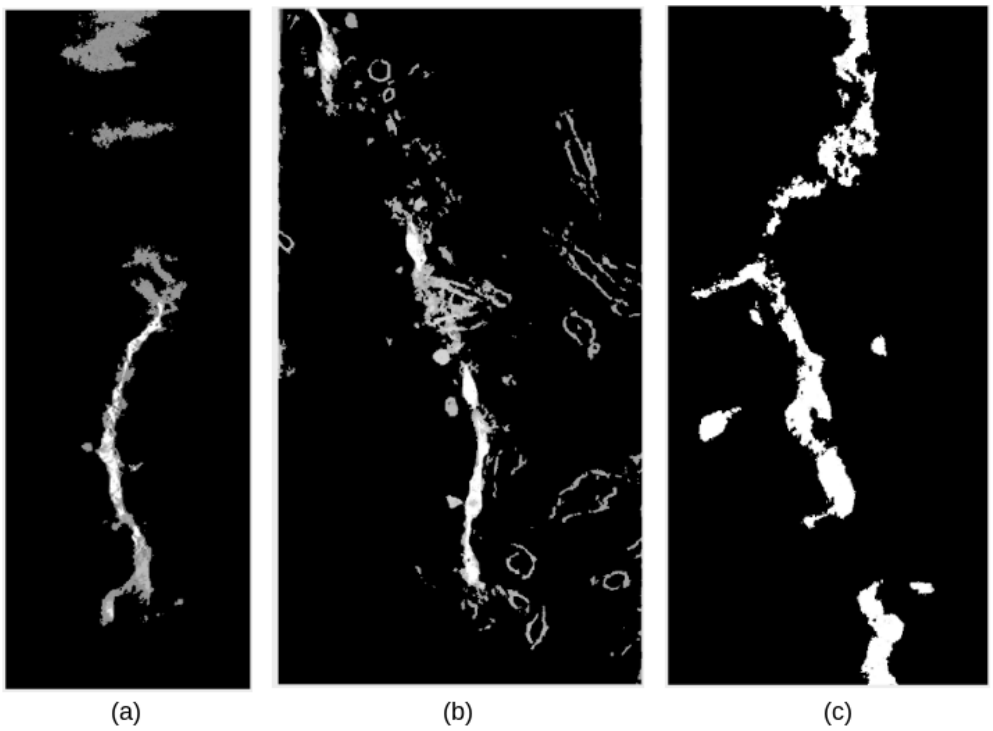

Figura 4.32: Segmentación de los volúmenes MRC1, MRC2 y MRC3, respectivamente, mediante la implementación paralela del algoritmo MOFS.

4.5 Discusiones finales

De los resultados presentados en este Capítulo se puede observar que en el ambiente A1 se obtuvo un factor de optimización mejor que en el ambiente A2. Esto se puede apreciar en las Tablas 4.33 y 4.34, que muestran los tiempos de ejecución (en segundos) del cómputo de afinidades y de la ejecución del algoritmo MOFS, respectivamente. Ambas tablas muestran el tiempo serial y el tiempo paralelo en los dos ambientes, sobre el conjunto MRC3 (se decidió discutir sobre este conjunto ya que es el de mayor número de vóxeles; los otros conjuntos presentan resultados bajo el mismo comportamiento).

	Tiempo Serial	Tiempo Paralelo	Factor
Ambiente A1	188	18.2	10.32x
Ambiente A2	73	11	6.63x

Figura 4.33: Tiempo de ejecución del cómputo de afinidades en ambos ambientes, para el conjunto de datos MRC3, así como el factor de optimización alcanzado en cada uno de ellos.

	Tiempo Serial	Tiempo Paralelo	Factor
Ambiente A1	901.3	358.2	2.51x
Ambiente A2	537.6	260.2	2.06x

Figura 4.34: Tiempo de ejecución del algoritmo MOFS en ambos ambientes, para el conjunto de datos MRC3, así como el factor de optimización alcanzado en cada uno de ellos.

Estos resultados parecen ser contraintuitivos respecto al factor de optimización

alcanzado en cada ambiente, debido a que como se mencionó al inicio del Capítulo, el ambiente A2 tiene mejores recursos de cómputo que el ambiente A1. Sin embargo, aunque el factor de optimización es mejor en el ambiente A1, dicho factor sólo muestra el ratio entre el tiempo serial y el tiempo paralelo, y no se debe perder de vista que lo importante es el tiempo de ejecución total (o velocidad) con la que se lleva a cabo una tarea dada. Si se comparan los tiempos de ejecución de ambas tareas, el ambiente A2 (al contar con mejores recursos de cómputo) sigue siendo más rápido que el ambiente A1, tanto en la ejecución serial como en la ejecución paralela; esto se puede observar también en las Figuras 4.35 y 4.36. Así mismo, al tratarse de un ratio, el factor de optimización por sí solo no es comparable de un ambiente de ejecución a otro, debido a que se están comparando los tiempos de ejecución serial y paralelo de forma independiente para cada ambiente de ejecución.

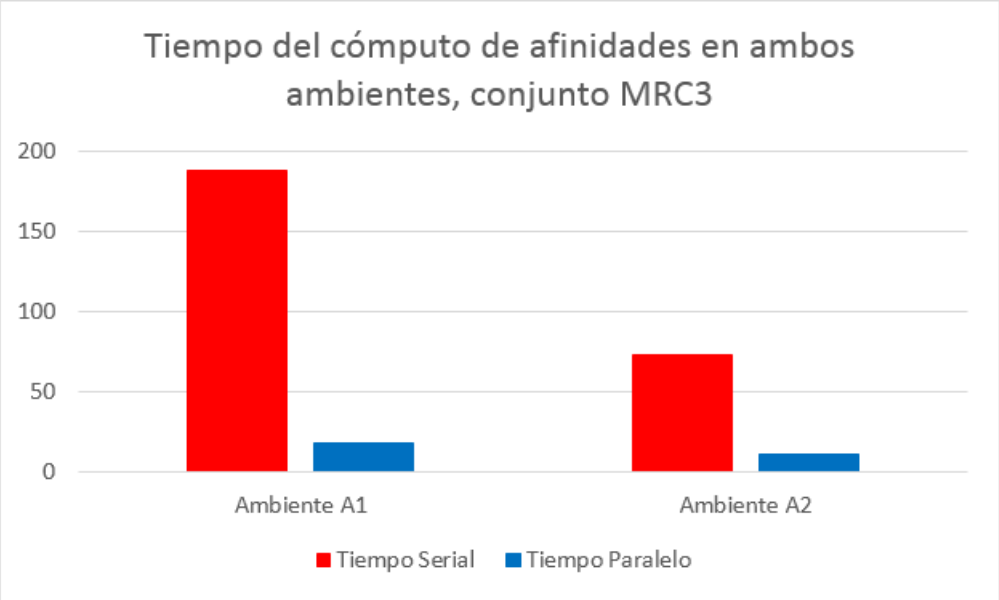

Figura 4.35: Tiempo de ejecución del cómputo de afinidades en ambos ambientes, para el conjunto de datos MRC3.

Este trabajo demuestra que se puede optimizar (realizando paralelismo) la eje-

cución del algoritmo MOFS en diferentes ambientes de ejecución, bajo el lenguaje OpenCL^{TM}. Con las descripciones de ambos ambientes, se puede esperar que el ambiente A2 sea más rápido que el ambiente A1, y los resultados obtenidos tanto en la ejecución serial como en la paralela confirman este hecho.

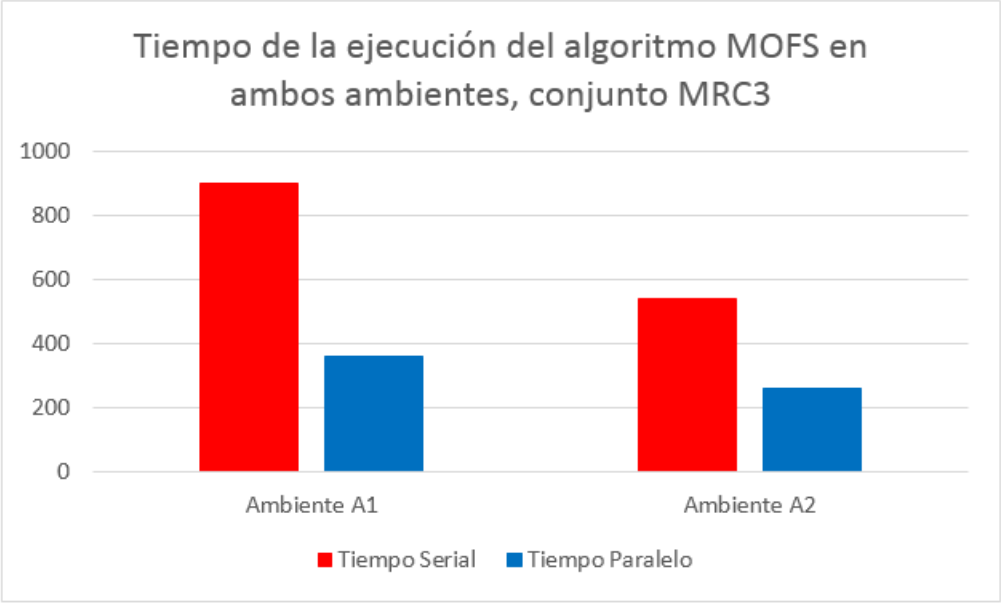

Figura 4.36: Tiempo de ejecución del cómputo de afinidades en ambos ambientes, para el conjunto de datos MRC3.

Capítulo 5

Conclusiones

En el presente trabajo se vio la utilidad de realizar un proceso de paralelización sobre el algoritmo de segmentación MOFS. El trabajo se enfocó en analizar la existencia de las tareas principales que se llevan a cabo en el proceso de segmentación, para poder reestructurarlas y paralelizarlas de tal forma que se obtuviera un factor de optimización respecto a su tiempo de ejecución. Se aprovechó la implementación serial que se tenía en nuestro grupo de trabajo, para empezar el desarrollo con la arquitectura de ésta implementación y agregar los componentes necesarios para la ejecución en paralelo. Se decidió trabajar con el lenguaje OpenCL^{TM} dada su flexibilidad de ejecución en múltiples plataformas, lo cual se pudo corroborar con la ejecución del trabajo propuesto en dos ambientes diferentes, ya que no se necesitó realizar alguna configuración específica para alguno de ellos.

Como parte de este trabajo fue construida una arquitectura paralela, acoplada a la implementación serial, que facilita la construcción y compilación de cualquier *kernel* OpenCL^{TM}, aunado al paso de parámetros (de una forma más sencilla) que sean requeridos para su funcionamiento y la búsqueda del dispositivo adecuado para su ejecución. La utilidad de esta arquitectura permitirá extender las funcionalidades en modo paralelo que se deseen incorporar posteriormente, siendo de gran ayuda para el programador del sistema. Así mismo, se prestó especial atención al uso de memoria y a las restricciones del lenguaje OpenCL^{TM}, ya que ello derivó a buscar estrategias para aprovechar al máximo los recursos del dispositivo que ejecutaría el *kernel* OpenCL^{TM} y buscar la forma de unificar los datos que se iban a manipular.

El proceso de segmentación MOFS se realizó sobre tres conjuntos de datos 3D,

de 50, 60 y 70 millones aproximadamente, tanto con la implementación serial con la que se contaba como con la implementación paralela propuesta en este trabajo, de tal forma que se pudieran comparar los tiempos de ejecución de cada una de ellas. Se pudo observar que el factor de optimización entre ambos tiempos de ejecución fue un poco mayor a 2×, como se presentó en los resultados del Capítulo 4, para el proceso de segmentación MOFS en su totalidad. Sin embargo, al analizar de forma individual la tarea a la cual se le dedicó el mayor esfuerzo de refactorización para su paralelización (tarea del cómputo de afinidades), se puede apreciar que se alcanzaron factores de optimización de hasta 10×, siendo 6× el factor mínimo alcanzado; esto tiene que ver mucho con la independencia de datos que hay en dicho proceso, ya que gracias a esto es posible alcanzar un mejor desempeño evitando procesos de sincronización o comunicación. Por otro lado, en la segunda tarea (tarea del cómputo de conectividades) sólamente se hizo uso de la LUT propuesta como estructura de datos y calculada de forma paralela, pero no se llevó a cabo un proceso de paralelización mediante *kernels* de OpenCL^{TM}, como en la primera tarea, por lo que el factor de 2× alcanzado sólo por el uso de la nueva estructura de datos parece adecuado.

Para corroborar que los resultados de la segmentación MOFS fueran los correctos, se tomó como *ground-truth* los resultados generados por la implementación serial, como se muestra en la sección final del Capítulo 4. Los resultados generados por la implementación paralela se compararon elemento por elemento con el *ground-truth*, para los tres conjuntos de datos utilizados en este trabajo, y no se observó diferencia alguna; esto se puede apreciar también visualmente en las Figuras 4.31 y 4.32, por lo que se puede concluir que el trabajo y desempeño del sistema propuesto es correcto.

Finalmente, se pudo comprobar que no se necesita realizar una inversión económica mayor para adquirir o construir un sistema de alto procesamiento como un clúster o red de computadoras y realizar cómputo paralelo, sino que gracias al uso de *GPUs* y el lenguaje OpenCL^{TM} podemos ocupar el equipo que se tiene en nuestro grupo de trabajo.

5.1 Trabajo futuro

Como se mencionó anteriormente, la tarea en la que se dedicó un esfuerzo de paralelización mayor fue la del cómputo de afinidades; una vez analizados los resultados, se puede observar que una de las primeras tareas que se desprende del presente trabajo es la paralelización del cómputo de conectividades, lo que requerirá de mecanismos de sincronización y comunicación entre los distintos núcleos que se tengan contemplados para su ejecución, debido a la dependencia de datos que surge en este proceso; actualmente se tiene en progreso dicha tarea, cuya implementación se está trabajando también con el lenguaje OpenCL^{TM}.

El presente trabajo solamente utiliza conjuntos de datos cuyos valores escalares representan un solo valor de intensidad, por lo que es importante también extender la implementación del mismo para aceptar representaciones de color (por ejemplo, el espacio de color *RGB*). Así mismo, es importante probar otras funciones de afinidad, por ejemplo, aquellas que incorporan medidas de textura o consideran un vecindario más grande para generar las características y estadísticas de las clases de interés.

Finalmente, un trabajo interesante sería realizar el proceso de migración de la versión de OpenCL^{TM} ocupada en este trabajo (versión 1.2) a la versión más reciente (versión 2.0), que incorpora otras características técnicas que serían de utilidad, como el manejo dinámico de memoria en los *kernels*. Así mismo, sería importante realizar una implementación con la arquitectura CUDA^{TM} para comparar los resultados que se puedan generar específicamente en dispositivos *GPU*.

Apéndice A

Restricciones del lenguaje OpenCL

De acuerdo con la especificación oficial de la versión 1.2 de OpenCL publicada por el *Khronos OpenCL Working Group* [4], OpenCL cuenta con las restricciones listadas a continuación.

1. El uso de apuntadores es restringido. Las siguientes reglas aplican en la creación de un programa con OpenCL:

   (a) Los argumentos de un *kernel* que son declarados como un apuntador deben de estar precedidos por el calificador *__global*, *__constant* o *__local*.

   (b) Un apuntador declarado con el calificador *__global*, *__constant* o *__local* sólo puede ser asignado a otro apuntador declarado en el mismo espacio de memoria (*__global*, *__constant* o *__local*, respectivamente).

   (c) Los apuntadores a funciones no están permitidos.

   (d) Los argumentos de un *kernel* no pueden ser declarados como un apuntador de apuntadores.

2. Los tipos de datos para el manejo de imágenes sólo son válidos en la declaración de argumentos (no pueden ser declarados dentro del cuerpo de un *kernel*) y sólo son accedidos en modo lectura, por las funciones propias del lenguaje.

3. Los argumentos cuyo tipo de dato es para el manejo de imágenes no pueden estar precedidos por ningún calificador de memoria (*__global*, *__constant*, *__local* o *__private*), ya que estos son almacenados por default en la memoria global.

4. A diferencia del lenguaje C, no está permitido declarar miembros del tipo *bit-field* dentro de una *struct.*

5. El uso de arreglos dinámicos o *structs* que tengan como miembros arreglos de este tipo no está soportado.

6. Las macros y funciones variadicas (aquellas que admiten un número variable de argumentos) no están soportadas.

7. Las funciones de las bibliotecas *assert.h*, *ctype.h*, *complex.h*, *errno.h*, *fenv.h*, *float.h*, *inttypes.h*, *limits.h*, *locale.h*, *setjmp.h*, *signal.h*, *stdarg.h*, *stdio.h*, *stdlib.h*, *string.h*, *tgmath.h*, *time.h*, *wchar.h* y *wctype.h* no están disponibles y no pueden ser incluidas en un *kernel*.

8. Los especificadores *auto* y *register* no están soportados.

9. La recursión no está soportada.

10. El valor de retorno de un *kernel* debe de ser del tipo *void*.

11. Los argumentos de un *kernel* no pueden ser declarados del tipo *bool*, *half*, *size_t*, *ptrdiff_t*, *intptr_t*, o *uintptr_t*, ya que el tamaño en bytes de estos tipos es dependiente de la implementación del estándar OpenCL hecha por el dispositivo, y al programa *host* le resultaría difícil asignar los buffers correspondientes para pasar como parámetros datos de estos tipos.

12. Los argumentos de un *kernel* no pueden ser declarados del tipo *event_t*.

13. Los elementos de una *struct* o una *union* deben de ser declarados con el mismo calificador de memoria.

14. Los calificadores *const*, *restrict* y *volatile* están soportados, sin embargo, no pueden usarse con tipos de datos especializados para el manejo de imágenes, como el *image2d_t* o *image3d_t*.

Bibliografía

[1] "Dijkstra's greedy shortest path algorithm," https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/, accedido: 22-08-2018.

[2] "Evolución de la fotografía móvil: de añadido extra a motivo principal de compra," https://www.xatakamovil.com/movil-y-sociedad/evolucion-de-la-fotografia-movil-de-anadido-extra-a-motivo-principal-de-compra, accedido: 22-08-2018.

[3] "List of manual image annotation tools," https://en.wikipedia.org/wiki/List_of_manual_image_annotation_tools, accedido: 22-08-2018.

[4] "OpenCL reference pages," https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/, accedido: 22-08-2018.

[5] "Top500," https://www.top500.org/, accedido: 22-08-2018.

[6] "¿Qué es la computación acelerada por GPU?" https://la.nvidia.com/object/what-is-gpu-computing-la.html, accedido: 22-08-2018.

[7] R. Acharya, R. Menon, A. Singh, D. Goldgof, and D. Terzopoulos, "A review of biomedical image segmentation techniques," *Deformable Models in Medical Image Analysis*, pp. 140–161, 1998.

[8] A. Ahmad, R. Mohd, H. Gan, and K. Sayuti, "A mini review on the design of interactive tool for medical image segmentation," in *2017 International Conference on Engineering Technology and Technopreneurship (ICE2T)*, Kuala Lumpur, Malaysia, 18-20 September 2017, pp. 1–5.

[9] H. Akramifard, R. Askari, and M. Firouzmand, "Edge detecting and partitioning by comparative gradient," *IACSIT International Journal of Engineering and Technology*, vol. 4, no. 2, pp. 192–197, 2012.

[10] A. Anjna and K. Rajandeep, "Review of image segmentation technique," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 4, pp. 36–40, 2017.

[11] E. Ashton, L. Molinelli, S. Totterman, and K. Parker, "Evaluation of reproducibility for manual and semi-automated feature extraction in CT and MR images," in *Proceedings. International Conference on Image Processing*, vol. 3, Rochester, NY, USA, 22-25 September 2002, pp. 161–164.

[12] A. Bali and S. Narayan, "A review on the strategies and techniques of image segmentation," in *2015 Fifth International Conference on Advanced Computing & Communication Technologies*, Haryana, India, 21-22 February 2015, pp. 113–120.

[13] A. Bazille, M. Guttman, E. McVeigh, and E. Zerhouni, "Impact of semiautomated versus manual image segmentation errors on myocardial strain calculation by magnetic resonance tagging," *Invest Radiol*, vol. 29, no. 4, pp. 427–433, 1994.

[14] K. E. Bett, J. S. Rowlinson, and G. Saville, *Thermodynamics for Chemical Engineers*. MITP, 2003.

[15] S. Beucher and C. Lantuejoul, "Use of watersheds in contour detection," in *International Workshop on Image Processing: Real-time Edge and Motion Detection/Estimation*, Rennes, France, 17-21 September 1979, pp. 1–12.

[16] S. Beucher and C. Mathmatique, "The watershed transformation applied to image segmentation," *Scanning Microscopy*, vol. 6, p. 299–314, 2000.

[17] M. C. Calatrava and T. Auzinger, "General-purpose graphics processing units in service-oriented architectures," in *2013 IEEE 6th International Conference on*

Service-Oriented Computing and Applications, Koloa, HI, USA, 16-18 December 2013, pp. 260–268.

[18] A. Calzado and J. Geleijns, "Tomografía computarizada. evolución, principios técnicos y aplicaciones," *Revista de Física Médica*, vol. 11, no. 3, pp. 163–180, 2010.

[19] B. Carvalho, L. Oliveira, and G. Andrade, "Fuzzy segmentation of color video shots," *Discrete Geometry For Computer Imagery*, vol. 4245, pp. 494–505, 2006.

[20] B. M. Carvalho, E. Garduño, and G. T. Herman, "Multiseeded fuzzy segmentation on the face centered cubic grid," in *Advances in Pattern Recognition: Second International Conference - ICAPR 2001*, vol. 2013, Rio de Janeiro, Brazil, 11-14 March 2001, pp. 339–348.

[21] B. M. Carvalho, C. J. Gau, G. T. Herman, and T. Y. Kong, "Algorithms for fuzzy segmentation," *Pattern Analysis and Applications*, vol. 2, no. 1, pp. 73–81, 1999.

[22] B. M. Carvalho, G. T. Herman, and T. Y. Kong, "Simultaneous fuzzy segmentation of multiple objects," *Discrete Applied Mathematics*, vol. 151, no. 1, pp. 55–77, 2005.

[23] C. L. Ceja, "Segmentación difusa como generador de funciones de transferencia para visualización directa de volúmenes," Tesis de maestría, Universidad Nacional Autónoma de México, Posgrado en Ciencia e Ingeniería de la Computación, mayo 2012.

[24] Q. Chen, B. Xue, Q. Sun, and D. Xia, "Interactive image segmentation based on object contour feature image," in *2010 IEEE International Conference on Image Processing*, Hong Kong, China, 26-29 September 2010, pp. 3605–3608.

[25] P. Chiranjeevi and S. Sengupta, "New fuzzy texture features for robust detection of moving objects," *IEEE SIGNAL PROCESSING LETTERS*, vol. 19, no. 10, pp. 603–606, 2012.

[26] K. C. Ciesielski, G. T. Herman, and T. Y. Kong, "General theory of fuzzy connectedness segmentations," *Journal of Mathematical Imaging and Vision*, vol. 55, no. 3, pp. 304–342, 2016.

[27] K. C. Ciesielski and J. K. Udupa, "Affinity functions in fuzzy connectedness based image segmentation (i): Equivalence of affinities," *Computer Vision and Image Understanding*, vol. 114, no. 1, pp. 146–154, 2010.

[28] ——, "Affinity functions in fuzzy connectedness based image segmentation (ii): Defining and recognizing truly novel affinities," *Computer Vision and Image Understanding*, vol. 114, no. 1, pp. 155–166, 2010.

[29] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to algorithms*. The MIT Press, 2009.

[30] J. Czajkowska, J. Kawa, Z. Czajkowski, and M. Grzegorzek, "4-D fuzzy connectedness-based medical image segmentation technique," in *Proceedings of the 20th International Conference Mixed Design of Integrated Circuits and Systems - MIXDES 2013*, vol. 1, Gdynia, Poland, 20-22 June 2013, pp. 519–524.

[31] M. Dubreuil, "Grid computing for parallel bioinspired algorithms," *Journal of Parallel and Distributed Computing*, vol. 66, no. 8, pp. 1052–1061, 2006.

[32] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*. Wiley, 1973.

[33] I. Foster, *Designing and Building Parallel Programs*. Addison-Wesley, 1995.

[34] I. Fumihiko, "The past, present and future of GPU-Accelerated computing," in *2013 First International Symposium on Computing and Networking*, Matsuyama, Japan, 4-6 December 2013, pp. 17–21.

[35] E. Garduño, "Visualization and extraction of structural components from reconstructed volumes," Tesis de doctorado, Universidad de Pensilvania, 2002.

[36] E. Garduño and G. T. Herman, "Parallel fuzzy segmentation of multiple objects," *International journal of imaging systems and technology*, vol. 18, no. 5, p. 336–344, 2008.

[37] E. Garduño, G. T. Herman, and R. Davidi, "Reconstruction from a few projections by l_1-minimization of the Haar A. transform," *Inverse Problems*, vol. 27, no. 5, 2011.

[38] E. Garduño, G. T. Herman, and H. Katz, "Boundary tracking in 3D binary images to produce rhombic faces for a dodecahedral model," *IEEE Transactions on Medical Imaging*, vol. 17, no. 6, pp. 1097–1100, 1998.

[39] E. Garduño, M. Wong-Barnum, N. Volkmann, and M. Ellisman, "Segmentation of electron tomographic data sets using fuzzy set theory principles," *Journal of Structural Biology*, vol. 162, no. 3, pp. 368–379, 2008.

[40] R. González and R. E. Woods, *Digital Image Processing.* Pearson / Prentice Hall, 2008.

[41] E. Heijman, J. Aben, C. Penners, P. Niessen, R. Guillaume, K. Nicolay, and G. Strijkers, "Evaluation of manual and automatic segmentation of the mouse heart from cine MR images," *Journal of Magnetic Resonance Imaging*, vol. 27, no. 1, pp. 86–93, 2008.

[42] G. T. Herman, *Geometry of Digital Spaces.* Birkhaüser Boston, 1998.

[43] G. T. Herman and B. M. Carvalho, "Multiseeded segmentation using fuzzy connectedness," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 5, pp. 460–474, 2001.

[44] R. Huifeng, H. Guyu, X. Jun, and P. Zhisong, "Fuzzy co-occurrence matrix fusion based texture feature extraction," in *The 27th Chinese Control and Decision Conference (2015 CCDC)*, Qingdao, China, 23-25 May 2015, pp. 3622–3626.

[45] J. Iglesias and M. Sabuncu, "Multi-atlas segmentation of biomedical images: A survey," *Medical Image Analysis*, vol. 24, no. 1, pp. 205–219, 2015.

[46] A. Jain, M. Murty, and P. Flynn, "Data clustering: A review," *ACM Computing Surveys*, vol. 31, no. 3, pp. 264–323, 1999.

[47] X. Jian, R. Zhang, and S. Nie, "Image segmentation based on level set method," *Physics Procedia*, vol. 33, pp. 840–845, 2012.

[48] C. Kranthi, K. Manjunathachari, and G. V. Subba, "Speckle noise reduction in 3D ultrasound images — a review," in *2015 International Conference on Signal Processing and Communication Engineering Systems*, Guntur, India, 2-3 January 2015, pp. 257–259.

[49] L. K. Lee, S. C. Liew, and W. J. Thong, *A Review of Image Segmentation Methodologies in Medical Image*. Springer, 2015.

[50] C. Li, X. Jia, and Y. Sun, "Improved semi-automated segmentation of cardiac CT and MR images," in *2009 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, Boston, MA, USA, 28 June - 01 July 2009, pp. 25–28.

[51] J. Liu, D. Narayan, K. Chang, L. Kim, E. Turkbey, L. Lu, J. Yao, and R. Summers, "Automated segmentation of the thyroid gland on CT using multi-atlas label fusion and random forest," in *2015 IEEE 12th International Symposium on Biomedical Imaging (ISBI)*, New York, USA, 16-19 April 2015, pp. 1114–1117.

[52] T. Liu, H. Zhou, F. Lin, Y. Pang, and J. Wu, "Improving image segmentation by gradient vector flow and mean shift," *Pattern Recognition Letters*, vol. 29, no. 1, pp. 90–95, 2008.

[53] G. López and F. Martínez, *Introducción a las Ecuaciones Diferenciales Parciales*. UAM, 2014.

[54] T. B. Moeslund, *Introduction to Video and Image Processing*. Springer, 2012.

[55] J. Méndez, "Segmentación de crio-rebanadas de ratón usando principios de lógica difusa," Tesis de maestría, Universidad Nacional Autónoma de México, Posgrado en Ingeniería Eléctrica, julio 2017.

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

[57] J. Oakley and E. Hancock, "Texture segmentation using fuzzy clustering," in *IEE Colloquium on Texture Classification: Theory and Applications*, vol. 5, London, UK, 07 October 1994, pp. 1–4.

[58] J. Olveres, R. Nava, E. Moya, B. Escalante, J. Brieva, G. Cristóbal, and E. Vallejo, "Texture descriptor approaches to level set segmentation in medical images," in *Proceedings, SPIE Photonics Europe*, vol. 9138, Brussels, Belgium, 15 May 2014, pp. 1–12.

[59] P. Pacheco, *An Introduction to Parallel Programming.* Morgan Kaufmann, 2011.

[60] S. Park, H. S. Lee, and J. Kim, "Seed growing for interactive image segmentation with geodesic voting," in *2016 IEEE International Conference on Image Processing (ICIP)*, Phoenix, AZ, USA, 25-28 September 2016, pp. 2564–2568.

[61] D. Pawar, "GPU based background subtraction using CUDA: State of the art," in *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*, Chennai, India, 22-24 March 2017, pp. 1201–1205.

[62] A. S. Pednekar and I. A. Kakadiaris, "Image segmentation based on fuzzy connectedness using dynamic weights," *IEEE Transactions on Image Processing*, vol. 15, no. 6, pp. 1555–1562, 2006.

[63] S. Plaza, L. Scheffer, and M. Saunders, "Minimizing manual image segmentation turn-around time for neuronal reconstruction by embracing uncertainty," *PLoS ONE*, vol. 7, no. 9, pp. 1–14, 2012.

[64] H. Prajapati and S. Vij, "Analytical study of parallel and distributed image processing," in *2011 International Conference on Image Information Processing (ICIIP 2011)*, Shimla, India, 3-5 November 2011, pp. 1–6.

[65] R. Rodríguez and J. Sossa, *Procesamiento y Análisis Digital de Imágenes*. Alfaomega, 2012.

[66] F. Salzenstein and C. Collet, "Fuzzy markov random fields versus chains for multispectral image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 28, no. 11, pp. 1753–1767, 2006.

[67] S. Sanjay, B. Sahiner, H. Chan, and N. Petrick, "Neural network based segmentation using a priori image models," in *Proceedings of International Conference on Neural Networks*, vol. 4, Houston, TX, USA, 12 June 1997, pp. 2455–2459.

[68] M. Scarpino, *OpenCL in Action*. Manning, 2012.

[69] G. Shan, Z. Boyu, Y. Cihui, and Y. Lei, "Speckle noise reduction in medical ultrasound image using monogenic wavelet and laplace mixture distribution," *Digital Signal Processing*, vol. 72, pp. 192–207, 2018.

[70] P. ShanmugaSundaram and N. Santhiyakumari, "Interactive segmentation of capsule endoscopy images using grow cut method," in *2014 International Conference on Computational Intelligence and Communication Networks*, Bhopal, India, 14-16 November 2014, pp. 190–192.

[71] N. Sharma and L. Aggarwal, "Automated medical image segmentation techniques," *Journal of Medical Physics*, vol. 35, no. 1, pp. 3–14, 2010.

[72] J. Smith, *Spectral Audio Signal Processing*. W3K Publishing, 2011.

[73] J. Smolka, "Watershed based region growing algorithm," *Annales UMCS Informatica AI*, vol. 3, no. 1, p. 169–178, 2005.

[74] J. Suri, D. Wilson, and S. Laxminarayan, *Handbook of biomedical image analysis: Segmentation models, part B*. Kluwer Academic / Plenum Publishers, 2005.

[75] R. Tsuchiyama, T. Nakamura, T. Iizuka, A. Asahara, J. Son, and S. Miki, *The OpenCL Programming Book*. Fixstars, 2012.

[76] J. K. Udupa, V. LeBlanc, Y. Zhuge, C. Imielinska, H. Schmidt, L. Currie, B. Hirsch, and J. Woodburn, "A framework for evaluating image segmentation algorithms," *Computerized Medical Imaging and Graphics*, vol. 30, no. 2, pp. 75–87, 2006.

[77] J. K. Udupa and P. K. Saha, "Fuzzy connectedness and image segmentation," *Proceedings of the IEEE*, vol. 91, no. 10, pp. 1649–1669, 2003.

[78] J. K. Udupa, P. K. Saha, and R. A. Lotufo, "Relative fuzzy connectedness and object definition: theory, algorithms and applications in image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 24, no. 11, pp. 1485–1500, 2002.

[79] M. Uscumlic, I. Relijin, D. Dujkovic, and B. Relijin, "Improvements in image segmentation by applying hopfield neural networks," in *2006 8th Seminar on Neural Network Applications in Electrical Engineering*, Serbia & Montenegro, Serbia, 25-27 September 2006, pp. 37–40.

[80] N. Volkmann, "A novel three-dimensional variant of the watershed transform for segmentation of electron density maps," *Journal of Structural Biology*, vol. 138, no. 1, pp. 123–129, 2002.

[81] D. Withey and Z. Koles, "Three generations of medical image segmentation: Methods and available software," *International Journal of Bioelectromagnetism*, vol. 10, no. 3, pp. 125–148, 2007.

[82] C. T. Zahn, "Graph-theoretical methods for detecting and describing gestalt clusters," *IEEE Transactions on Computers*, vol. 20, no. 1, pp. 68–86, 1971.

[83] L. G. Zeng, *Medical Image Reconstruction*. Springer, 2009.

[84] B. Zhou, X. Yang, R. Liu, and W. Wei, "Image segmentation with partial differential equations," *Information Technology Journal*, vol. 9, no. 5, pp. 1049–1052, 2010.

[85] W. Zhou and Y. Xie, "Interactive medical image segmentation using snake and multiscale curve editing," *Computational and Mathematical Methods in Medicine*, vol. 2013, pp. 1049–1052, 2013.

[86] Y. Zhuge, Y. Cao, and R. Miller, "Gpu accelerated fuzzy connected image segmentation by using cuda," in *31st Annual International Conference of the IEEE EMBS*, Minnesota, USA, 2-6 September 2009, pp. 6341–6344.

[87] Y. Zhuge, Y. Cao, J. Udupa, and R. Miller, "Parallel fuzzy connected image segmentation on gpu," *Journal of Medical Physics*, vol. 38, no. 7, pp. 4365–4371, 2011.

[88] G. Zoabli, P. Mathieu, C. Aubin, A. Tinlot, M. Beausejour, V. Feipel, and A. Malanda, "Assessment of manual segmentation of magnetic resonance iimage of skeletal muscles," in *2001 Conference Proceedings of the 23rd Annual International Conference of the IEEE Engineering in Medicine and Biology Society*, vol. 3, Istanbul, Turkey, 25-28 October 2001, pp. 2685–2687.