clMeanShift:
Paralelización y parametrización del algoritmo Mean Shift utilizando OpenCL

TRABAJO FIN DE MASTER

Autor: Eduardo Martín Gaspar

Director: Doctor Mariano Rincón Zamorano

Departamento de Inteligencia Artificial

Escuela Técnica Superior de Ingenieria Informática

Universidad Nacional de Estudios a Distancia

Curso 2010-2011

clMeanShift: Paralelización y parametrización del algoritmo Mean Shift utilizando OpenCL

Trabajo Fin de Master

Máster Universitario en I.A. Avanzada: Fundamentos, Métodos y Aplicaciones

Especialidad de Sistemas Inteligentes de Diagnóstico, Planificación y Control

Autor: Eduardo Martín Gaspar

Director: Doctor Mariano Rincón Zamorano

Departamento de Inteligencia Artificial

Escuela Técnica Superior de Ingenieria Informática

Universidad Nacional de Estudios a Distancia

Curso 2010-2011

La única posibilidad de descubrir
los límites de lo posible es
aventurarse un poco más allá de ellos,
hacia lo imposible.
Sir Arthur C. Clarke

Agradecimientos

El agradecimiento
es la memoria del corazón.
J. B. Massieu

Tras cuatro años, el presente trabajo constituye el último peldaño para completar mis estudios de Postgrado. Por ello, quiero aprovechar la oportunidad para dar las gracias a todas aquellas personas que me han ayudado durante estos últimos años, y durante toda mi vida.

En primer lugar, me gustaría dar las gracias al Doctor Mariano Rincón por ofrecerme la oportunidad de realizar este Trabajo Fin de Master, cuyo desarrollo a combinado varios de los campos que más me apasionan de la Informática, además de por toda la ayuda que me ha prestado.

Del mismo modo, quisiera agradecer a mis Padres y a mi hermana, por todos estos años en los que siempre han estado ahí cuando más los he necesitado, apoyándome en los momentos más difíciles y, sobre todo, animándome a continuar estudiando día a día.

A mis amigos de toda la vida, quisiera darles las gracias por todos los grandes momentos que hemos pasado juntos, y por todos aquellos que están por venir.

Y finalmente, y no por ello menos importante, me gustaría agradecer a Marta por todo este tiempo que estamos pasando juntos, y sobre todo, por su apoyo, paciencia y comprensión durante este último año.

A todos ellos, gracias.

Índice

Agradecimientos VII

1. Introducción 1

1.1. Aportaciones del Trabajo Fin de Master . . . 3

2. Planteamiento del Problema 4

2.1. El algoritmo Mean Shift . . . 6

2.2. Objetivos . . . 10

3. Estado del Arte 12

3.1. Mean Shift. Optimización el proceso. . . . 12

3.2. Paralelización de Algoritmos. . . . 14

3.2.1. El Modelo de Programación de OpenCL . . . 18

3.3. Paralelización del Algoritmo Mean Shift . . . 21

4. clMeanShift 25

4.1. Aspectos Generales . . . 25

4.2. Proceso de Filtrado . . . 28

4.2.1. Preprocesado . . . 28

4.2.2. Creación de la Cuadrícula Uniforme Tridimensional . . . 31

4.2.3. Bucle *Mean Shift* . . . 35

4.2.4. Postprocesado . . . 39

4.3. Proceso de Etiquetado. . . . 40

5. Detalles de la Implementación 45

5.1. Ocupación del Procesador . . . 46

5.2. Gestión de Memoria . . . 47

5.3. Rendimiento de las instrucciones . . . 50

6. Resultados Experimentales **53**

6.1. Entorno de simulación 53

6.2. Evaluación del Proceso de Filtrado 56

6.2.1. Proceso de filtrado para imágenes 2D 56

6.2.2. Proceso de filtrado para imágenes 3D 69

6.3. Evaluación del Proceso de Etiquetado 73

7. Conclusiones **79**

I Apéndices **82**

A. clMeanShift. Manual de Usuario. **83**

A.1. Operaciones 83

A.2. Tipos de Datos 85

Bibliografía **87**

Índice de figuras

2.1. Ejemplos de imágenes por Resonancia Magnética 2D y 3D. 5
2.2. Imagen por Resonancia Magnética 4D. 5
2.3. Ejemplo Espacio de Características (Extraída de [8]). 6
2.4. Evolución proceso *Mean Shift* (Figura extraída de [35]). 7
2.5. Aplicación del filtrado *Mean Shift* sobre información de escala de grises (Imagen extraída de [8]). 8

3.1. GPU. Arquitectura clásica. Proceso. 15
3.2. GPU. Arquitectura con capacidad para Lenguajes de Shading. 15
3.3. GPU. Ejemplo aplicación Shaders. 16
3.4. CPU y GPU. Filosofías de diseño (Extraído de [23])) 17
3.5. CPU (▲) y GPU(■). Evolución de la capacidad de procesamiento en GFLOPS. (Extraído de [23]) 17
3.6. Arquitectura OpenCL 19
3.7. Ejemplos *NDRange* 20
3.8. OpenCL. Modelo de Memoria 20
3.9. Kd–Tree tridimensional 23

4.1. Organización interna puntos imagen. 26
4.2. Ejemplo Segmentación *Mean Shift* . Imagen 2D 27
4.3. Proceso de Filtrado. 29
4.4. Preprocesado. 29
4.5. Preprocesado. 30
4.6. Cuadrícula Uniforme Tridimensional. 32
4.7. Creación Cuadrícula Tridimensional Uniforme 32
4.8. Cuadrícula 3D. Asignación de los puntos en las celdas de la cuadrícula. 33
4.9. Cuadrícula 3D. Ordenación lista de pares. 34

4.10. Cuadrícula 3D. Obtención límites celdas. 35
4.11. Intercambio de bucles para la paralelización del algoritmo 38
4.12. Bucle *Mean Shift* 39
4.13. Elementos que participan en el proceso 39
4.14. Postproceso 40
4.15. Topología de nodos. Ejemplo en imagen 2D. 41
4.16. Etiquetado. Inicialización. 42
4.17. Etiquetado. Construcción de la Lista de Equivalencia. 42
4.18. Etiquetado. Redución Lista de Equivalencia. 43
4.19. Etiquetado. Asignación Lista de Equivalencia sobre puntos imagen. 43
4.20. Algoritmo de Etiquetado. Proceso de ejecución de los núcleos. 43

5.1. Transferencia Memoria *Anfitrión – Dispositivo OpenCL* 48
5.2. Ejemplos de patrones de acceso a información de 4 Bytes. 49
5.3. Desenrrollado de bucles. 51

6.1. Imágenes de prueba 2D. 54
6.2. Imagen de prueba 3D. MRI 3D. Tamaño 192 x 192 x 160 54
6.3. Aplicación filtrado Mean Shift sobre imagen *Mandril*. $h = (h_s, h_r)$. 57
6.4. Aplicación filtrado Mean Shift sobre imagen *Lenna*. $h = (h_s, h_r)$. 59
6.5. Aplicación filtrado Mean Shift sobre imagen *MRI 2D*. $h = (h_s, h_r)$. 61
6.6. Comparativa tiempo de proceso. CPU-GPU. 63
6.7. Tiempo por fase. 64
6.8. Porcentaje de tiempo por fase. 65
6.9. Bucle Mean Shift. Tiempo por iteración. Lena $h = (32, 32)$ 65
6.10. Comparativa de tiempos del algoritmo de ordenación 67
6.11. Comparativa coste proceso de filtrado. clMeanshift vs EDISON. 68
6.12. Speed Up conseguido respecto a EDISON. 69
6.13. Resultado proceso de filtrado. MRI 3D. 71
6.14. MRI 3D. Coste del proceso de filtrado. 72
6.15. Resultado proceso de etiquetado. Lenna. 74
6.16. Resultado proceso de etiquetado. MRI 2D. 75
6.17. Resultado proceso de etiquetado. MRI 3D. 76
6.18. EDISON. Proceso de etiquetado. 77
6.19. Coste del Proceso de Etiquetado. 78

Índice de Tablas

6.1. Entorno de pruebas. Capacidades OpenCL. 55
6.2. Filtrado 2D. Parámetros de simulación. 56
6.3. Filtrado 3D. Parámetros de simulación. 70

Capítulo 1

Introducción

La Visión Artificial, también conocida como Visión por Computador, es la rama de la Inteligencia Artificial que tiene como objetivo dotar a las máquinas de la capacidad para percibir y *entender* una escena o las características de una imagen. La Visión Artificial abarca una gran cantidad de procesos que intentan modelar ciertos atributos de la percepción humana que permitan dar esta capacidad a las máquinas, y permitan por ello, utilizarlas para la resolución de problemas específicos.

La aplicación de técnicas de Visión Artificial se extendiende a lo largo de un amplio espectro de áreas tan heterogéneas como pueden ser la Robótica, la Industria, la Medicina y los Sistemas de Vigilacia. Cada una de estas aplicaciones presenta una serie de características particulares que han dado lugar a diferentes propuestas para resolver los problemas que plantean, y cuyo objetivo fundamental es la aplicación de la Visión Artificial de manera automática o semiautomática.

Como parte de la Visión Artificial, el proceso de Segmentación constituye una de las etapas fundamentales, y a la vez, la que más dificultades implica. Su objetivo es dividir la información captada por el sistema en varias regiones o zonas disjuntas que permitan establecer una representación de la información con mayor significado y, a la vez, más fácil de analizar. Varias son las propuestas de algoritmos de Segmentación que se han ido desarrollando, muchas de ellas adaptadas a problemas concretos.

Como parte de este conjunto de algoritmos, *Mean Shift* se ha convertido en un algoritmo muy popular al presentar muy buenos resultados cuando es aplicado en procesos tan diferentes como la segmentación de imágenes, volúmenes y vídeo, el suavizado adaptativo y el seguimiento de objetos. La principal ventaja que presenta este algoritmo, frente a otros equivalentes, es que no necesita tener conocimiento previo sobre las características propias de la información a procesar, además de realizar el tratamiento de la información de manera homogénea, al trabajar en el denominado Espacio de Características, y de manera prácticamente automática, al sólo ser necesario un número reducido de parámetros para controlar el proceso.

A pesar de todos sus beneficios, *Mean Shift* presenta un alto coste computacional al presentar una complejidad $O(kdn^2)$ en su versión básica. Esto hace que su aplicación práctica en problemas, donde se trata una gran cantidad de información o donde es necesario obtener los resultados de manera rápida, esté

limitada. Por ello, existen diferentes modificaciones del algoritmo original cuyo objetivo es la reducción de su coste.

Por otro lado, a lo largo de la historia de la computación, se ha considerado la paralelización de algoritmos como un método para reducir el tiempo necesario para resolver aquellos problemas de mayor complejidad computacional. En la actualidad, con la proliferación de los microprocesadores masivamente paralelos, este paradigma ha adquirido de nuevo gran importancia. Estos dispositivos presentan, generación tras generación, una mayor capacidad de cálculo al aumentar el número de unidades de cálculo que incorporan sin mucho coste económico. Esto ha llevado a un aumento del interés en desarrollar algoritmos que exploten estas capacidades, y su aplicación en la resolución de aquellos problemas más complejos.

Esto supone la necesidad de establecer estándares que permitan definir un modelo de desarrollo de algoritmos eficientes destinados a este tipo de arquitecturas. A pesar de que los diferentes fabricantes han desarrollado sus propios modelos, se ha visto la necesidad de desarrollar un estándar para implementar soluciones que se adapten a cualquiera de estas arquitecturas. El estándar OpenCL ha sido especificado con este objetivo, estableciendo un modelo de desarrollo que permite desarrollar algoritmos paralelos que se puedan ejecutar sobre dispositivos heterogéneos.

Teniendo en cuenta todos estos aspectos, el presente Trabajo Fin de Master propone una nueva revisión de *Mean Shift* basada en la paralelización del algoritmo básico, por medio del modelo establecido por OpenCL. El principal objetivo de esta revisión es la reducción del tiempo de proceso necesario para llevar a cabo la aplicación de *Mean Shift* , estableciendo el camino para conseguir realizar el proceso prácticamente en tiempo real. Adicionalmente, se plantea la generalización del método, al realizar un diseño para su aplicación sobre imágenes 2D, 3D y 4D, permitiendo realizar el control del proceso por medio de un número reducido de parámetros establecidos por el usuario. Todo esto permite abrir una vía que plantee su aplicación en un abanico más amplio de problemas, como puede ser el seguimiento en Tiempo Real.

Este documento establece en el Capítulo 2 las bases sobre las que se ha desarrollado el presente Trabajo Fin de Master, así como el problema a resolver. Para ello, se parte del ámbito en el cual se establece la Segmentación como método de la Visión por Computador cuyo objetivo es la extracción de las características que definen una imagen, las cuales son de utilidad para su aplicación en diferentes campos. Además, se presenta el Algoritmo *Mean Shift* como una técnica que permite obtener una alta calidad en el proceso de Segmentación de Imágenes. Finalmente, se establecen el conjunto de objetivos marcados durante la realización de este Trabajo Fin de Master.

El Capítulo 3 realiza una breve descripción de las diferentes mejoras propuestas para el algoritmo *Mean Shift* , cuyo objetivo es reducir su coste computacional. Por otro lado, se describen las actuales posibilidades para crear soluciones basadas en algoritmos paralelos, que son ejecutados en las denominadas arquitecturas *many-cores*, además de destacar la necesidad de crear un modelo de programación estándar que establezca las bases para el desarrollo de este tipo de algoritmos. En concreto, se destaca el nuevo estándar OpenCL como modelo para el desarrollo de algoritmos paralelos aprovechando este tipo de arquitecturas, y que ha

sido el escogido para el diseño e implementación de la solución al problema establecido. Finalmente, como consecuencia del auge de estas tecnologías, se describen una serie de propuestas que han tenido como objetivo aplicar los algoritmos paralelos para la implementación del algoritmo *Mean Shift* .

En el Capítulo 4 se realiza la descripción detallada del algoritmo *Mean Shift* paralelo propuesto, denominado como *clMeanShift*, que aplica el modelo OpenCL para la paralelización del algoritmo. Por ello, se detallan cada una de las fases en las que queda divida la solución, así como el conjunto de labores realizadas en cada una de las mismas. La solución desarrollada tiene la principal característica de trabajar de manera homogénea sobre los diferentes tipos de imágenes para los que está diseñada, adaptándose de manera automática aquellos aspectos particulares de cada uno de los tipos de imagen procesados.

El Capítulo 5 detalla aquellos aspectos que se han tenido en cuenta durante el desarrollo de los diferentes módulos que componen la implementación. Entre estos aspectos, destacan aquellas características propias de la arquitectura donde se va a ejecutar el algoritmo, además se muestran las optimizaciones que se han realizado con el objeto de reducir el coste computacional de la implementación.

En el Capítulo 6 se describen el conjunto de simulaciones y resultados obtenidos durante la evaluación de la solución diseñada. Para ello, se realiza una batería de pruebas sobre un conjunto de imágenes modelo que, permiten comparar el tiempo necesario para la realización del proceso, respecto a otras implementaciones existentes, así como la evaluación de la calidad del resultado obtenido.

Finalmente, en el Capítulo 7 se detalla el conjunto de conclusiones obtenidas de este Trabajo Fin de Master, así como el posible conjunto de tareas y líneas de investigación que se pueden derivar de él.

1.1. Aportaciones del Trabajo Fin de Master

Este Trabajo Fin de Master presenta el siguiente conjunto de contribuciones a los actuales sistemas de Visión por Computador, y en concreto, a la aplicación del Algoritmo Mean Shift en los mismos.

1. Generalización del Algoritmo Mean Shift para su aplicación a diferentes tipos de imágenes (2D, 3D y 4D).

2. Parametrización del algoritmo para facilitar el control del proceso.

3. Paralelización del algoritmo Mean Shift para su ejecución en arquitecturas *many–cores*, aprovenchando las capacidades ofrecidas por el estándar OpenCL.

Capítulo 2

Planteamiento del Problema

En el campo de la Visión por Computador, la Segmentación de Imágenes tiene como misión dividir la información capturada en varias partes u objetos, con el objetivo de obtener una representación con mayor significado y más fácil de analizar. Este proceso es utilizado tanto para la localización de objetos como para encontrar los límites de éstos dentro de la imagen tratada, y tiene gran relevancia en diferentes campos tanto Industriales como en la Medicina. En el caso concreto de la Medicina, su aplicación facilita la separación e identificación de las partes anatómicas de interés para la realización de estudios y diagnósticos de múltiples enfermedades.

Desde hace años, la utilización de Imágenes por Resonancia Magnética (MRI) se ha extendido como una técnica no invasiva que permite la obtención de información sobre la estructura y la composición del cuerpo a analizar de una manera más segura para el paciente, al no utilizar radiaciones ionizadas (Rayos Gamma o Rayos X). Concretamente, son utilizadas ampliamente para la observación de alteraciones en los tejidos, y como método de detección de cáncer y otras patologías en el cuerpo humano. Como resultado del proceso de exploración por medio de una Resonancia Magnética se obtiene un conjunto de imágenes 2D (Figura 2.1a) o un volumen 3D (Figura 2.1b), que son tratadas por aplicaciones específicas que realizan el proceso de explotación de esta información. De manera análoga, si el proceso de captura se realiza durante un periodo de tiempo se obtiene una imagen 4D (Figura 2.2), que es una secuencia de volúmenes 3D a lo largo de dicho tiempo.

Por lo tanto, se observa que el proceso de segmentación de imágenes se debe adaptar a información de diferente naturaleza. Por ello, es preciso desarrollar algoritmos de segmentación controlados por pocos parámetros y que realicen su proceso de manera precisa y eficiente, con el objetivo de poderse aplicar a un amplio espectro de problemas.

(a) MRI 2D. (b) MRI 3D.

Figura 2.1: Ejemplos de imágenes por Resonancia Magnética 2D y 3D.

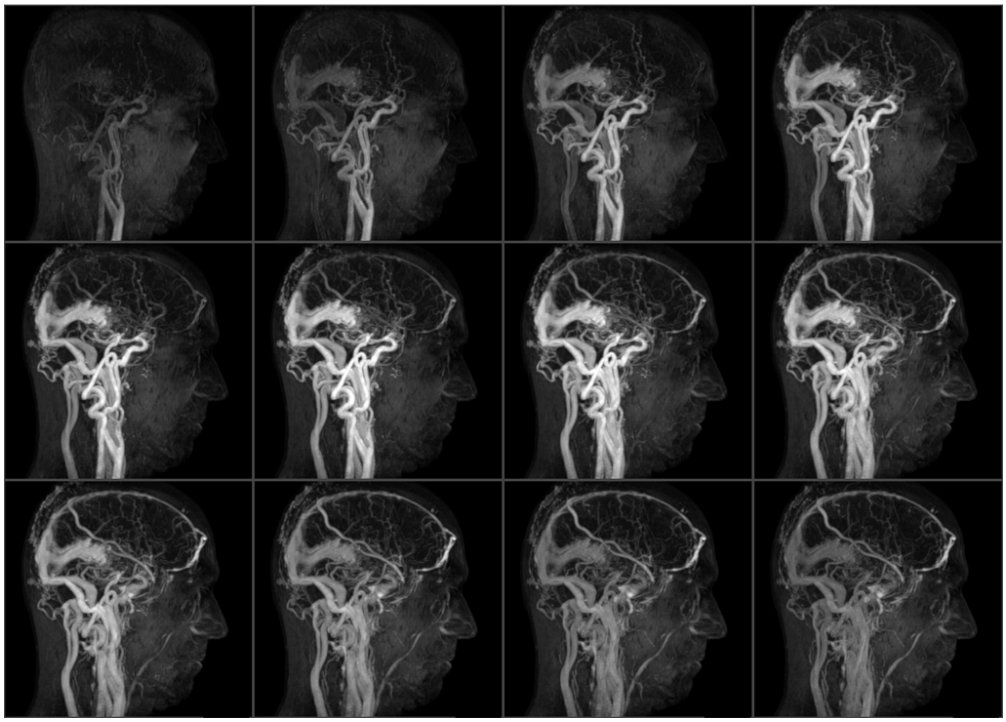


Figura 2.2: Imagen por Resonancia Magnética 4D.

A lo largo del tiempo, se han desarrollado diferentes técnicas de segmentación (ver [14]), cada una de ellas adaptadas al problema concreto donde se aplicaban y teniendo en cuenta las características propias de la información a procesar. De manera resumida, existen técnicas basadas en umbralización, técnicas de agrupación (*clustering*) como el algoritmo *K-Means* [20], técnicas basadas en detección de bordes, métodos de crecimiento de regiones, técnicas basadas en modelos, transformación divisoria (*watershed*), particionamiento

gráfico,...

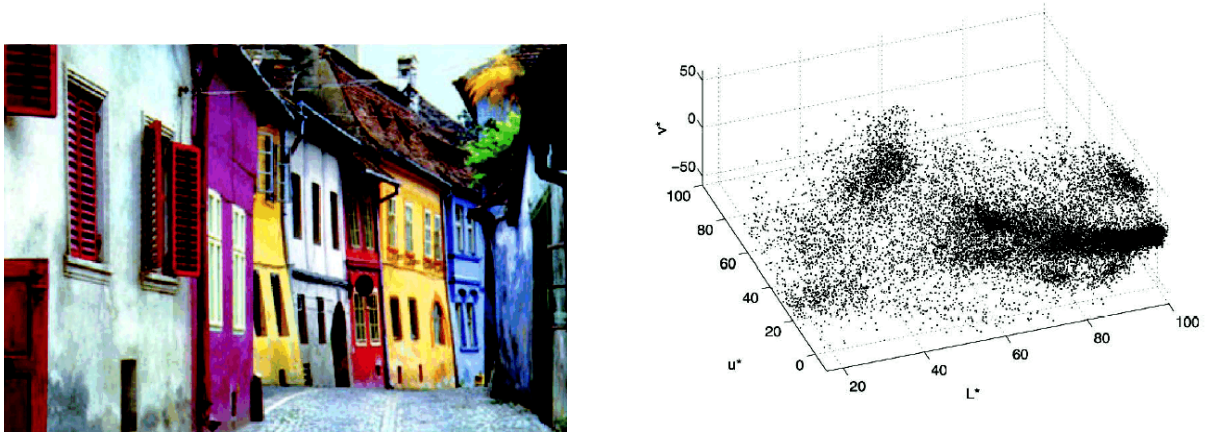
2.1. El algoritmo Mean Shift

Mean Shift es un algoritmo general de búsqueda de modas no paramétrico que ha sido aplicado de manera satisfactoria sobre un gran espectro de campos como la segmentación de imágenes 2D [7], *clustering* [5] y seguimiento [9]. La principal característica que presenta es no necesitar conocimiento acerca de la forma de la distribución o número de modas de la información, a diferencia de otros algoritmos equivalentes (por ej. K-Means).

Este algoritmo fue inicialmente propuesto por Fukunaga y Hostetler [12], y posteriormente aplicado al análisis de imágenes por Cheng [5]. Finalmente Comaniciu y Meer [8] lo refinaron para su aplicación como algoritmo de bajo nivel en Visión por Computador en diferentes aplicaciones (Segmentación, Suavizado Adaptativo, Seguimiento).

Mean Shift parte de un conjunto de puntos $S = \{x_1, x_2, \ldots, x_n\}$ que forman parte de un espacio de características d–dimensional R^d. En este espacio, cada una de las muestras que componen la información de entrada (p. ej. los píxeles de una imagen) es representada por un punto x_i en el espacio multidimensional, cuya dimensión d está determinada por las dimensiones de las características que son utilizadas para representar la muestra asociada. De esta manera, aunque la naturaleza del espacio de características es dependiente de la aplicación, al representar los puntos de una manera homogénea para todos los casos, el proceso es totalmente independiente.

A modo de ejemplo, en la figura 2.3b se muestra la representación en el espacio tridimensional de características $L * u * v$ correspondiente a los colores de los píxeles de la imagen 2.3a. Como se puede apreciar, se distinguen claramente los colores dominantes de cada sección de objeto en la imagen y los grupos que se obtienen en el espacio de características, que forman zonas densas de puntos.



(a) Imagen.

(b) Representación en el espacio de características L*u*v.

Figura 2.3: Ejemplo Espacio de Características (Extraída de [8]).

De esta manera, *Mean Shift* considera el conjunto de puntos S como una función de densidad de probabilidad empírica (p.d.f), donde las regiones densas en este espacio de características corresponden a un máximo local o *moda* en la distribución implícita. De esta manera, para cada uno de los puntos de S se realiza un proceso iterativo de ascenso de grandiente en torno a un estimador local de densidad que provoca el desplazamiento del punto hasta converger a uno de los máximos locales o *modas* (ver [10] para detalles).

Dicho desplazamiento viene definido por la expresión 2.1:

$$x_{i+1} = m(x) = \frac{\sum_{i=1}^{n} g\left(\| \frac{x-x_i}{h} \|^2\right) x_i}{\sum_{i=1}^{n} g\left(\| \frac{x-x_i}{h} \|^2\right)} \tag{2.1}$$

Donde x es el punto actual de S a procesar, $m(x)$ es la nueva posición x_{i+1} del punto tras la iteración, $g(x) = -k'(x)$ donde $k(x)$ la función de núcleo utilizada (normalmente Gausiana, Uniforme o Epanechnikov), h es el ancho de banda o radio que controla el tamaño de la función de núcleo, y x_i son el resto de puntos. En [10] se describe el desarrollo completo hasta llegar a esta expresión.

En la figura 2.4 se aprecia de manera gráfica la evolución del proceso para un punto dado.

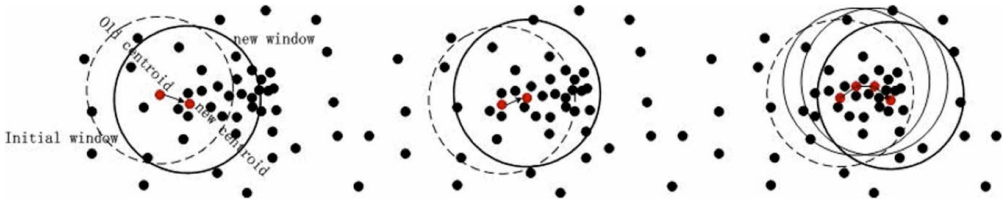
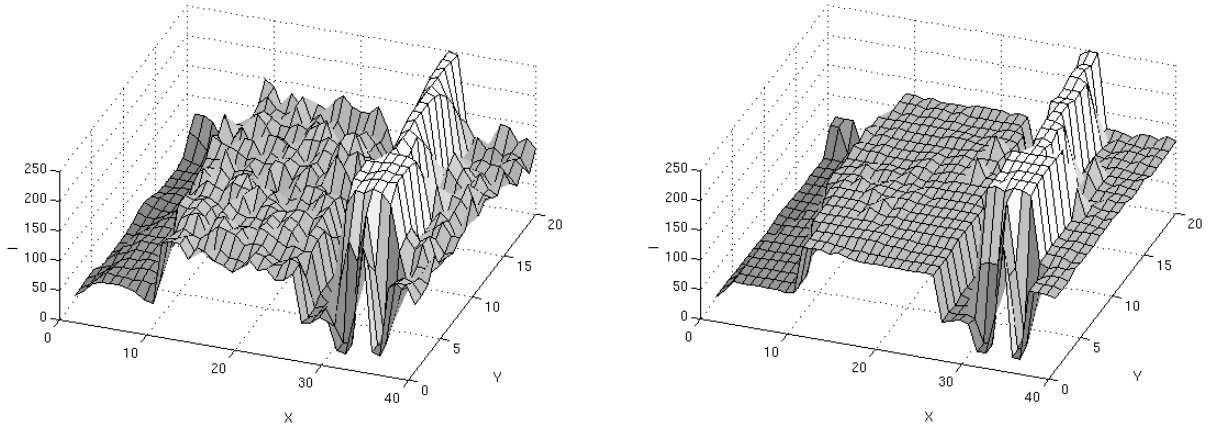


Figura 2.4: Evolución proceso *Mean Shift* (Figura extraída de [35]).

A partir de la expresión 2.1 se obtiene el vector *Mean Shift* para el punto x, que viene definido por la expresión 2.2, y apunta hacia la dirección de máximo incremento en la densidad. Además, su módulo indica el desplazamiento que sufre el punto en la interación, y es utilizado como criterio de convergencia del proceso iterativo para alcanzar el máximo local o *moda* asociado.

$$M_h(x) = m(x) - x. \tag{2.2}$$

Como resultado de este proceso, una vez aplicado a los diferentes puntos que componen S, aquellos que convergen a la misma *moda* o máximo local son clasificados en el mismo grupo. En la figura 2.5 se aprecia de manera gráfica el resultado del proceso de filtrado basado en *Mean Shift* . Como se puede observar, el filtro hace uniformes aquellas zonas en las que las características de los puntos que las componen son similares, lo que permite identificar los diferentes grupos.



(a) Información de Entrada.

(b) Resultado del proceso de filtrado.

Figura 2.5: Aplicación del filtrado *Mean Shift* sobre información de escala de grises (Imagen extraída de [8]).

El algoritmo *Mean Shift* presenta las siguientes de ventajas frente a otros algoritmos equivalentes:

- No depende de la naturaleza de la información a tratar.
- Se puede aplicar al análisis de información real.
- No asume a priori la forma de los grupos de datos resultantes.
- Puede tratar distintos tipos de espacios de características.
- Necesita pocos parámetros para su aplicación. Básicamente el tamaño de la ventana (h).

Como se ha indicado, el algoritmo *Mean Shift* no presenta restricciones respecto a la naturaleza de la información a tratar. Comaniciu, en primer lugar, propuso en [7] la segmentación aplicando esta técnica sobre imagénes 2D, aunque posteriormente extendió su aplicación también como algoritmo de Seguimiento [9]. Una imagen 2D se puede considerar como una cuadrícula bidimensional en la que cada píxel viene definido por un vector r–dimensional, que forma parte del denominado *Dominio de la Gama o Color (Range Domain)*, y que representa un valor en la escala de grises ($r = 1$), un valor de color ($r = 3$) o un valor multiespectral ($r > 3$). Dicha información debe cumplir que la diferencia entre dos puntos debe corresponder con una distancia Euclídea, por lo que es necesario representar dichas características en un formato que presente la propiedad de mapeo lineal. Por ello, para el caso de imágenes de color se considera la representación de este valor por medio del mapa de color $L * u * v$ (CIELUV) [33], donde se puede asumir esta métrica Euclídea. Para el caso de escala de grises, sólo se considera la componente L de este mapa de color. De esta manera, estos valores son utilizados para componer los puntos del espacio de características asociado a la imagen y realizar el proceso *Mean Shift* .

Al aplicar dicho método de manera directa, no se tiene en cuenta la estructura propia de la imagen, es decir, la distribución espacial de sus píxeles. Dicha posición forma parte del denominado *Dominio Espacial*

(Spatial Domain), donde se asume una métrica Euclídea. Este hecho puede provocar resultados insatisfactorios, por lo que es necesario trabajar en lo que Comaniciu denominó como *Dominio de Unión Espacio-Color*. Para ello, cada uno de los puntos que componen el Espacio de Características está compuesto por la unión de su información de color y su información de posición, y cuya dimensión total d corresponde con la suma de las dimensiones de ambos dominios. Se considera que, por construcción, este nuevo dominio cumple también la métrica Euclídea.

Al trabajar en este nuevo dominio, es necesario adaptar el núcleo multivariado al estar cada punto formado por información de distinta naturaleza. Por ello, se redefine la expresión para dicha función como el producto de dos núcleos radialmente simétricos, tal y como se muestra en 2.3.

$$K_{h_s,h_r}(x) = \frac{C}{h_s^q h_r^p} k\left(\left\|\frac{x^s}{h_s}\right\|^2\right) k\left(\left\|\frac{x^r}{h_r}\right\|^2\right). \tag{2.3}$$

Donde x^s son las coordenadas de la posición que ocupa el píxel en la imagen de dimensión $q = 2$ y x^r son las coordenadas de su valor de color de dimensión p. La unión de ambos valores forman parte el vector de características de x. Por otro lado, $k(x)$ es el núcleo común utilizado, obteniendo buenos resultados si se aplica un núcleo de Epanechnikov o un núcleo normal truncado. Finalmente, h_s y h_r son los dos únicos parámetros a establecer y que corresponden con los anchos de banda o radios que controlan el tamaño de la función de núcleo para cada dominio, y que determinan la resolución de la detección de modas.

Extender este caso a imágenes 3D es trivial ya que la única diferencia es que el Dominio del Espacio tiene 3 dimensiones. Finalmente, para el caso de imágenes 4D, es decir, aquellas en las que interviene también el Dominio del Tiempo, se puede extender la expresión 2.3 para incorporar dicho Dominio, definiendo un nuevo valor h_t que controla su ancho de banda correspondiente. Esta nueva expresión se muestra en 2.4.

$$K_{h_t,h_s,h_r}(x) = \frac{C}{h_t h_s^q h_r^p} k\left(\left\|\frac{x^t}{h_t}\right\|^2\right) k\left(\left\|\frac{x^s}{h_s}\right\|^2\right) k\left(\left\|\frac{x^r}{h_r}\right\|^2\right). \tag{2.4}$$

Considerando todos estos aspectos, realizar la Segmentación por medio del algoritmo *Mean Shift* consiste en un proceso dividido en dos partes independientes, tal y como se indica en [8]: Una fase de filtrado y una fase de etiquetado.

En primer lugar, se realiza un proceso de filtrado de la imagen aplicando directamente el algoritmo *Mean Shift* sobre los puntos que componen el espacio de características de la información de entrada. Dicho proceso viene definido por el algoritmo básico 1.

Algorithm 1: Algoritmo básico *Mean Shift*

Entrada: $\{x_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen original en el dominio espacio-color, normalizados con los valores h_i.

Salida: $\{z_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen filtrada en el dominio espacio-color

Para cada $i = 1 \dots n$ **hacer**

Inicializar $j = 1$ e $y_{i,1} = x_i$.

Calcular $y_{i,j+1}$ a partir de 2.1 hasta converger, $y = y_{i,conv}$.

Asignar $z_i = (x_i^s, y_{i,conv}^r)$

El proceso consiste en tomar cada punto y aplicar 2.1 hasta alcanzar su máximo local o moda asociado. Entonces se le asigna el valor de color $y_{i,conv}^r$ de este máximo al punto tratado.

Una vez obtenida la imagen filtrada se procede a realizar el proceso de etiquetado, que consiste en agrupar aquellos puntos de la imagen similares para formar los diferentes grupos o *blobs*. Para ello, se tiene en cuenta tanto su posición relativa dentro de la imagen como la diferencia de valor de color.

Aunque el algoritmo *Mean Shift* , por sus características propias es un algoritmo muy eficiente y versátil, también presenta una serie de aspectos a tener en cuenta. Por un lado, a pesar de que el algoritmo sólo precisa un único parámetro para realizar el proceso, el tamaño de las ventanas h no es trivial y depende en gran medida de la información a tratar. La selección de un tamaño de ventana inapropiado puede provocar la unión de dos grupos diferentes o la definición de grupos adicionales, por lo que se pueden obtener malos resultados del proceso de segmentación.

Por otro lado, el principal handicap que presenta el algoritmo es el alto coste computacional del mismo. Si se considera que, para cada uno de los n puntos d–dimensionales de la información de entrada, es necesario realizar el proceso iterativo un número medio de k iteraciones hasta converger a su moda asociada, la complejidad propia del algoritmo es aproximadamente de $O(kdn^2)$. En dicha complejidad, se considera que en cada iteración se calcula el nuevo valor x_{i+1} a partir de x_i para todos los puntos de S.

2.2. Objetivos

Como ya se ha ido viendo a lo largo del presente capítulo, el algoritmo *Mean Shift* es un buen algoritmo para realizar el proceso de segmentación, cuyas principales bondades son la adaptación a la naturaleza de la información de entrada y, sobre todo, permitir controlar el proceso por medio unos pocos parámetros. A pesar de estos puntos fuertes, el mayor problema que plantea es su alto coste computacional. Dicho coste viene dado por el proceso iterativo hasta que el punto converge a su máximo local, y por otro lado, el coste derivado del cálculo necesario en cada una de estas iteraciones.

Estos aspectos hacen que el coste computacional del algoritmo crezca considerablemente debido a dos factores fundamentales:

- Aumento del tamaño del problema, es decir, la cantidad de información a procesar.
- Aumento del tamaño de la ventana, lo que mejora la solución obtenida, pero también aumenta el coste por iteración del proceso.

Como se describe en el próximo capítulo, varios han sido los estudios y propuestas realizadas, cuya misión ha sido reducir este coste.

Con estas premisas, en el presente Trabajo Fin de Master se propone una solución para reducir este coste computacional por medio de la paralelización del algoritmo. A diferencia de un algoritmo secuencial en el que una única unidad de proceso ejecuta el algoritmo de manera secuencial, en el caso de un algoritmo paralelo, el cálculo se divide en varias unidades de proceso que realizan el mismo cálculo en paralelo sobre diferentes datos de la información de entrada, obteniendo al final la solución al problema tratado.

Por lo tanto, teniendo en cuenta la naturaleza propia del algoritmo *Mean Shift* básico, se puede plantear la paralelización del algoritmo. Como se ha podido apreciar, durante el proceso de filtrado de la información de entrada, el algoritmo trata cada uno de los puntos a procesar de manera independiente, hasta que convergen a su máximo local. Por ello, se puede plantear un esquema en el que cada punto es asignado a una unidad de proceso, que se encarga de calcular su máximo local asociado. De esta manera, es previsible un aumento del rendimiento del algoritmo.

Adicionalmente, se plantea obtener una versión parametrizada y general del algoritmo que permita tratar imágenes de diferente naturaleza y dimensión espacial. Como se ha visto a lo largo de este capítulo, una de las ventajas que presenta el algoritmo *Mean Shift* , al mapear las muestras que componen la información de entrada en puntos del espacio de características, es que la naturaleza de la información no modifica el algoritmo. Por ello, el algoritmo propuesto trabaja indiferentemente con imágenes 2D, 3D y 4D.

Como se ha mencionado anteriormente, las imágenes 2D están compuestas por una rejilla bidimensional discreta en la que cada punto, denominado *píxel*, es considerado como un vector que representa un valor. Este vector puede representar un valor en escala de grises (dimensión 1), un color (dimensión 3) o multiespectral (dimensión mayor de 3), como sucede en las imágenes por satélite.

Por otro lado, las imágenes 3D representan un volumen, formando un rejilla tridimensional cuya unidad mínima es un cubo, denominado *voxel*, que tiene asociado un vector que representa un valor, de manera análoga al caso de las imágenes 2D.

Finalmente, las imágenes 4D son aquellas imágenes 3D que representan volúmenes pero en las que se considera también su evolución temporal, que compone la cuarta dimensión.

Capítulo 3

Estado del Arte

En el presente capítulo se va a realizar una breve descripción del conjunto de estrategias planteadas por diferentes autores para reducir el coste computacional del algoritmo *Mean Shift* . Además, se presentan los algoritmos paralelos, y más concretamente la tecnología actual disponible para el diseño e implementación de estos algoritmos en las denominadas arquitecturas *many-cores*. Finalmente, se realiza una descripción de una serie de propuestas que han surgido como aplicación de esta tecnología al caso del algoritmo *Mean Shift*.

3.1. Mean Shift. Optimización el proceso.

Como se ha indicado en el capítulo anterior, aunque el algoritmo *Mean Shift* obtiene muy buenos resultados en el proceso de segmentación, a partir de unos pocos parámetros, presenta un alto coste computacional que depende fundamentalmente del tamaño del problema y del tamaño de la ventana de búsqueda establecido. Adicionalmente, no existe una fórmula clara que defina un criterio claro para la selección del tamaño de búsqueda a establecer dependiendo del problema. Por ello, varios han sido las propuestas realizadas con el objeto de resolver todos estos aspectos.

En [4] se describen una serie de optimizaciones para el caso de imágenes 2D utilizando el proceso *Mean Shift* aplicando un núcleo Gaussiano. El objetivo de estas optimizaciones es reducir el tiempo necesario para realizar el proceso. Por otro lado, en [18] se proponen también optimizaciones considerando el proceso *Mean Shift* con un núcleo de Epanechnikov y que han sido aplicadas en el entorno EDISON [6], las cuales permiten realizar el proceso de segmentación sobre imágenes 2D.

En el proceso *Mean Shift* se distinguen claramente dos cuellos de botella. Por un lado, el número de iteraciones necesarias por punto hasta alcanzar su máximo local (Algoritmo 1), y por otro lado, el propio coste de cada iteración (Expresión 2.1), sobre todo cuando el espacio de características tiene una gran dimensionalidad. De esta manera, las estrategias propuestas en ambos artículos intentan, por un lado, organizar la información con el objetivo de reducir el coste por iteración, y por otro lado, aplicar algún tipo de heurística durante el proceso de ascensión del gradiente para reducir el número de iteraciones a realizar.

Una primera estrategia propuesta en [4], denominada *Discretización Espacial* consiste en considerar cada uno de los píxeles, que componen la imagen 2D a tratar, como un cuadrado que forma parte de una cuadrícula bidimensional. Cada uno de estos cuadrados es dividido en nxn celdas, siendo n un parámetro del proceso, repartiendo los distintos puntos a tratar entre estas celdas, a partir de sus coordenadas espaciales. De esta manera, se considera que los puntos del espacio de características, cuyas coordenadas espaciales pertenezcan a la misma celda, convergen al mismo máximo local. Por lo tanto, si durante el proceso de iteración para un punto dado, se alcanza una celda previamente visitada al realizar dicho proceso sobre otro punto, se para el proceso de iteración del punto y se le asigna la misma moda que el punto previamente calculado. Con esto, se reduce el número de iteraciones global a realizar sobre la información. El principal problema que plantea esta estrategia es que el número de máximos locales localizados es menor o igual a la versión básica, por lo que se pierde calidad en el proceso.

Por otro lado, en el mismo artículo se define la estrategia denominada como *Vecindad Espacial*. El algoritmo original *Mean Shift* considera que para cada punto hay que computar la influencia que ejercen el resto de puntos de la información sobre él, en cada una de las iteraciones. La cuestión es que el peso ejercido por los puntos más alejados es nula, por lo que sólo es necesario tener en cuenta los puntos que se encuentren dentro de la ventana establecida por el parámetro h, considerando su posición espacial. Por ello, para cada punto sólo se considera su conjunto de vecinos $N(x) = \{n \in \{1, \ldots, N\} : \|(x'_1, x'_2) - (x^n_1, x^n_2)\| \leq h$. Dicho conjunto $N(x)$ estará compuesto por $(2\lfloor r \rfloor + 1)^2)$ puntos, con lo que se reduce considerablemente el coste por iteración, al reducir el número de operaciones. El principal problema que plantea esta estrategia es que el punto puede converger a otro máximo local diferente al suyo.

De igual manera, en [18] hay que destacar las estrategias propuestas para el caso de utilización de un núcleo de Epanechnikov, que presenta la característica de converger antes que el núcleo Gaussiano. Una primera estrategia, que tiene como objetivo reducir el coste por iteración, consiste en crear una subdivisión espacial tridimensional formada por celdas de dimensión $h_s \times h_s \times h_r$, es decir, una serie de cubos cuyas dimensiones son iguales a las del parámetro h, considerando las dos coordenadas del Dominio Espacial que componen la imagen 2D y la primera coordenada del Dominio del Color (Componente L). Cada uno de los puntos del Espacio de Características es asignado a una de estas celdas a partir de sus tres primeras coordenadas que previamente son normalizadas por su h_i correspondiente. De esta manera, la búsqueda del conjunto de sus puntos vecinos $S_h(x)$ del punto a tratar se limita a los puntos asignados a su celda y a las celdas vecinas de ésta, reduciendo considerablemente el coste por iteración.

Por otro lado, se plantean otras dos estrategias que tienen como objetivo reducir el número de iteraciones, siendo una más agresiva que la otra. Básicamente, ambas se basan en la estrategia de *Discretización Espacial* propuesta en [4]. Para ello, si durante el proceso de ascensión de gradiente de un punto x_n, se alcanza un punto similar a éste cuya moda ya haya sido calculada, se para el proceso de iteración y se le asigna la moda del punto ya calculado. El criterio de similitud entre ambos puntos se calcula a partir de la diferencia de los dos vectores de características, considerando tanto la información espacial como la información de color.

Este último aspecto es la principal diferencia con la propuesta realizada en [4], en la que sólo se consideran la posición espacial, por lo que en principio se obtiene una solución de mayor calidad. A pesar de todo, como se indica en la evaluación de dichas técnicas, se obtiene un resultado con pérdida de calidad debido a la heurística de la condición de similitud en la que es necesario establecer un umbral como parámetro del proceso.

Como se describe en la solución planteada en este Trabajo Fin de Master, el aspecto fundamental a la hora de reducir el coste computacional del proceso, sin tener una pérdida considerable de la calidad de la solución, pasa por establecer estrategias que organicen la información a tratar de tal manera que faciliten la localización de los puntos que influyen en el cálculo de un punto dado, es decir, reducir el coste de cada iteración.

Adicionalmente a este conjunto de estrategias planteadas, existen otras propuestas centradas en definir un criterio que permiten establecer el valor del parámetro de ventana h que mejor se adapte a cada situación. Varias han sido las propuestas ([13], [26]) que proponen la utilización de tamaños de ventana adaptativos. Dicho aspecto no se ha tenido en cuenta a la hora de establecer la solución de este Trabajo Fin de Master al centrarse básicamente en obtener una solución que reduzca el coste computacional del mismo.

3.2. Paralelización de Algoritmos.

Durante las dos últimas décadas los fabricantes de microprocesadores, basados en una Unidad Central de Proceso (CPU), han creado dispositivos que, generación tras generación, han presentado mayor capacidad de proceso. Esta capacidad ha permitido a los desarrolladores crear aplicaciones que ejecutan cálculos cada vez más complejos, así como ofrecer una mayor versatilidad. Dicha evolución ha ido cumpliendo la denominada *Ley de Moore* que básicamente expresa que cada 18 meses se duplica el número de transistores en un circuito integrado, con el consecuente aumento de la capacidad de procesamiento del mismo.

Esto ha sido posible por dos factores primordiales. Por un lado, la mayor miniaturización de los microprocesadores, lo que ha permitido introducir una mayor cantidad de componentes en el chip. Y por otro lado, el aumento de la frecuencia del reloj del microprocesador. Todo ello se ha traducido en un aumento del número de instrucciones por segundo (IPS) y operaciones en coma flotante por segundo (FLOPS) que es capaz de ejecutar.

Este hecho se ha producido más o menos hasta 2003, año en el cual se alcanzó un cierto límite tecnológico, motivado principalmente por los problemas de consumo de energía y de disipación de calor de los últimos microprocesadores desarrollados, y que ha provocado que se frenara el aumento de la frecuencia del reloj de la CPU en los nuevos microprocesadores.

Por este motivo, los fabricantes han optado por una nueva filosofía en la que los procesadores están compuestos por varias unidades de procesamiento, denominadas *cores*, que permiten un aumento de las prestaciones del procesador sin necesidad de aumentar su velocidad o miniaturización. Esto a supuesto un

cambio en la filosofía de desarrollo de aplicaciones. Tradicionalmente, el desarrollo de software estaba basado en programas secuenciales, tal y como definió Von Neumann, donde el procesador ejecuta las instrucciones que componen el programa de manera secuencial. En cambio, los nuevos microprocesadores basados en varios *cores* permiten la concurrencia de varios hilos de ejecución en paralelo, que pueden colaborar entre sí para obtener un resultado común de manera más rápida a su equivalente secuencial, consiguiendo un aumento considerable de las prestaciones de las CPUs.

En paralelo a esta evolución, las Unidades de Procesamiento Gráfico (GPU), que integran las tarjetas gráficas, han sufrido una evolución más o menos equivalente. Dichos microprocesadores siguen una filosofía totalmente distinta a la mantenida en el diseño de las CPUs, denominada *many-cores*, en la que cada nueva generación introduce el doble de *cores* o unidades de proceso que la generación anterior.

Originalmente, las GPUs fueron diseñadas para realizar tareas de cálculo gráfico y renderizado (figura 3.1). Por ello, son arquitecturas que fueron diseñadas para el cálculo gráfico en el que se producen un gran número de transformaciones geométricas sobre vértices y polígonos, ejecutándose estos cálculos en paralelo sobre la información, con el objetivo de obtener la imagen a visualizar.

Figura 3.1: GPU. Arquitectura clásica. Proceso.

Con el paso del tiempo, estas arquitecturas han aumentado sus prestaciones (figura 3.2) permitiendo al desarrollador controlar de manera precisa sus procesos internos, por medio de los denominados *Lenguajes de Shading* (HLSL, GLSL), que ha sido ampliamente utilizado en campos como los videojuegos o herramientas de diseño gráfico para la generación de efectos, cálculos de iluminación y otras operaciones gráficas en tiempo real.

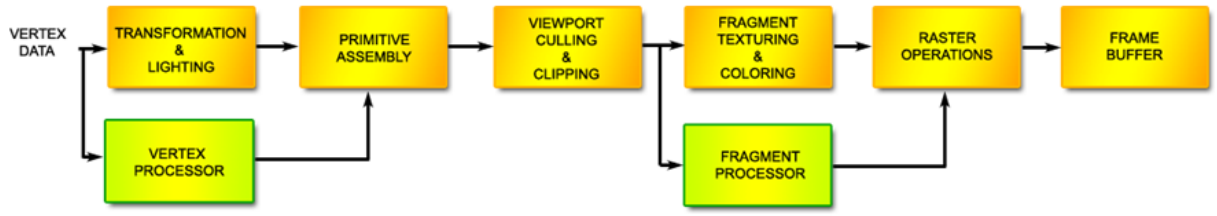


Figura 3.2: GPU. Arquitectura con capacidad para Lenguajes de Shading.

Para ello, estas arquitecturas incorporan componentes que permiten modificar las transformaciones sobre los vértices de entrada y los píxeles que componen la visualización resultante (Ver figura 3.3).

Figura 3.3: GPU. Ejemplo aplicación Shaders.

Durante los últimos años, ha nacido una nueva filosofía denomina Computación de Propósito General basada en GPU (GPGPU), en la que se ha planteado la utilización de la capacidad de cómputo de estas arquitecturas para ser aplicada a otros campos como el científico. Hay que tener en cuenta, que la arquitectura de una GPU está compuesta por unidades de proceso especializadas en cálculo en coma flotante, que son los tipos de operaciones necesarios para el tratamiento gráfico, pero que también tiene gran relevancia en áreas de cálculo científico. Estas arquitecturas están basadas en una filosofía denominada *Single Instruction – Multiple Threads* (SIMT), es decir, una misma operación es realizada a la vez sobre varios datos por distintos hilos en paralelo, que es la manera en la que se procesa la información gráfica. Por ello, su arquitectura interna está formada por múltiples *cores* que comparten el mismo control y ejecutan en orden la misma instrucción sobre diferentes datos, con lo que se alcanza un gran nivel de paralelismo en el proceso.

El ejemplo más claro de esta filosofía se puede ver en su aplicación para realizar la suma de dos vectores d–dimensionales. En el caso de un programa secuencial, éste debe iterar sobre cada una de las componentes del vector, realizando la suma y almacenando el resultado de la misma. En cambio, si consideramos un caso hipotético en el que la GPU tuviera el mismo número de núcleos que dimensiones posee el vector, cada componente se asignaría a un núcleo en concreto. Entonces, todos a la vez realizarían la operación de suma, y posteriormente, todos a la vez realizarían la operación de almacenar el resultado. De esta manera, se reduce aproximadamente d veces el tiempo necesario para llevar a cabo la operación completa.

La principal diferencia de las GPUs frente a las CPUs es que éstas últimas están especialmente diseñadas para ser de propósito general, en el cual puede haber varios ejecutables corriendo de manera independiente en cada uno de sus núcleos, o un mismo ejecutable dividido en varios hilos que son asignados a cada uno de los núcleos. Por otro lado, las GPUs son arquitecturas diseñadas especialmente para el procesamiento en paralelo, por lo que tienen gran capacidad para este tipo de operaciones. Además, debido a su proliferación y especialización, son arquitecturas más baratas en proporción, es decir, aumentar el número de unidades

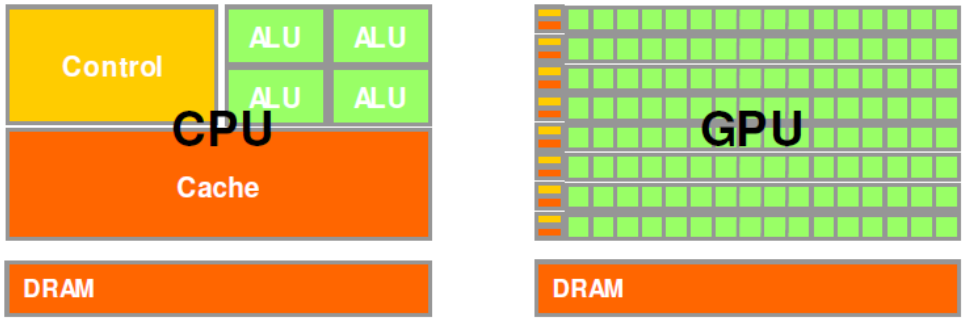


Figura 3.4: CPU y GPU. Filosofías de diseño (Extraído de [23]))

de procesamiento que incorporan se puede hacer sin un gran esfuerzo. De esta manera, en los últimos años, el aumento de potencia de cálculo de estos microprocesadores está siendo más fuerte frente a las últimas CPUs, tal y como se puede apreciar en la figura 3.5.

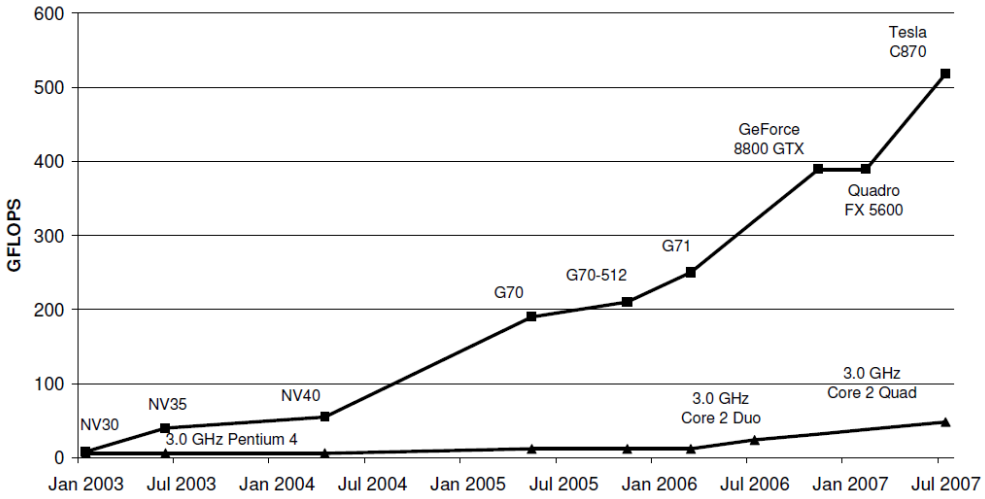


Figura 3.5: CPU (▲) y GPU(■). Evolución de la capacidad de procesamiento en GFLOPS. (Extraído de [23])

En la actualidad, ha surgido un gran interés en la paralelización de algoritmos, sobre todo aquellos de alto coste computacional, con el objetivo de aprovechar estas arquitecturas para aumentar su rendimiento y reducir su tiempo de proceso.

Esto a supuesto la necesidad, por parte de los desarrolladores, de poseer modelos y lenguajes de programación para el desarrollo de algoritmos paralelos. Varias han sido las propuestas que se han venido desarrollando a lo largo del tiempo. Los más extendidos hasta la fecha han sido *Message Passing Interface* (MPI) [27] y OpenMP [32], que son aplicados a las CPUs.

MPI está destinado a su aplicación sobre *clusters* de computadoras cada una con su propia CPU y que

no comparten memoria entre sí. En este modelo toda la iteración y compartición de memoria entre los nodos es realizada implícitamente a través de mensajes, siendo éste el aspecto fundamental y más conflictivo a la hora del desarrollo de aplicaciones basadas en esta filosofía.

Por otro lado, OpenMP está destinado a multiprocesadores que comparten memoria entre sí. Los programas desarrollados de esta manera siguen una filosofía parecida a la arquitectura *many-cores*, ya que el procesamiento de la información se paraleliza en diferentes hilos que realizan los cálculos sobre la información, pudiéndose ejecutar cada uno de ellos en los diferentes núcleos que disponga la CPU de la plataforma.

En el caso de las GPUs, y siguiendo la filosofía de OpenMP, los principales fabricantes de GPUs han desarrollado sus propias propuestas. Por un lado nVIDIA, desarrolló CUDA [30] como modelo de desarrollo de programación aplicado a sus dispositivos, y por otro lado, AMD desarrolló su modelo denominado ATI Stream, basado en el lenguaje Brook [3] para los suyos. Ambos modelos se basan en los mismos principios, permitir al desarrollador escribir programas de manera estándar y sencilla que se ejecuten en paralelo en sus dispositivos aprovechando su capacidad de proceso. En este modelo, el desarrollador programa los denominados núcleos de proceso o *kernels* que son ejecutados de manera paralela por el conjunto de unidades de procesamiento que componen el microprocesador. Igual que sucede en el caso de OpenMP, las diferentes unidades de procesamiento tienen cierta cantidad de memoria compartida que les permiten compartir información y realizar cierta sincronización en el proceso. El principal problema que plantean estos modelos es que son propietarios y sólo pueden ser aplicados a sus respectivos dispositivos.

Por ello, con el objetivo de estandarizar estos modelos de programación paralela, los principales fabricantes han definido el estándar OpenCL [22], con el objeto de unificar dichos modelos y extenderlos a otras arquitecturas con gran poder de cálculo (p. ej. Procesadores CELL). Este modelo define un conjunto de bibliotecas, extensiones de lenguaje de programación y compilador que permiten a los desarrolladores gestionar el nivel de paralelismo y tratamiento de la información en diferentes dispositivos (CPU, GPU, DSP, ...). De esta manera, el mismo programa desarrollado en OpenCL puede ser ejecutado indistintamente en todas aquellas plataformas compatibles con el estándar. Además, OpenCL aporta una manera homogénea para desarrollar programas tanto para hardware masivamente paralelo (GPU) como en hardware que permite hilos (CPU), al no estar limitado su utilización a una arquitectura específica.

3.2.1. El Modelo de Programación de OpenCL

OpenCL define un lenguaje de programación, basado en un subconjunto del estándar ISO C99 [17] y una extensión del mismo para paralelismo, que permite el desarrollo de los denominados *núcleos* (*kernels*). Un núcleo es una función especial, cuyas instrucciones pueden ser ejecutadas N veces en paralelo en un procesador compatible con OpenCL (p. ej. una tarjeta gráfica).

OpenCL define una abstracción de la arquitectura compuesta por un *Anfitrión* (*Host*) que está conectado a uno o varios *Dispositivos OpenCL* que están compuestos por una o más *Unidades de Computación*, y que a su vez están divididos en uno o más *Elementos de Proceso*. En la figura 3.6, se muestra dicha arquitectura.

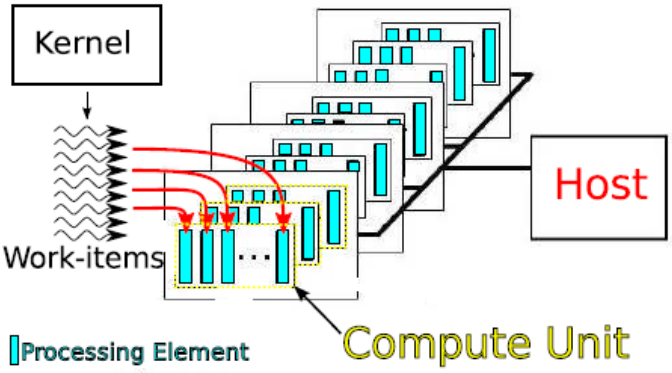
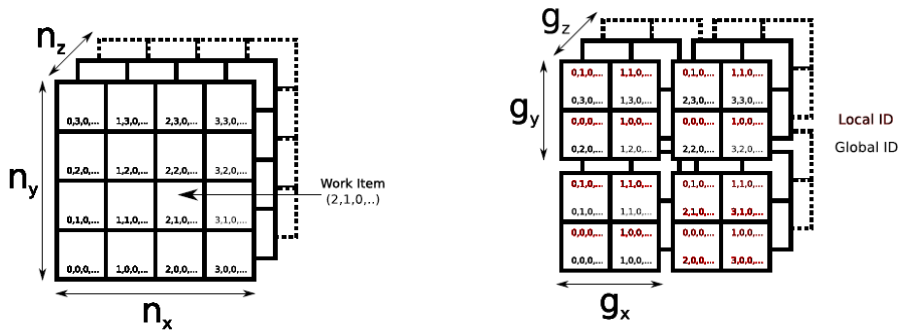


Figura 3.6: Arquitectura OpenCL

De esta manera, en el *Anfitrión* se ejecuta una aplicación nativa que está encargada de realizar la compilación del núcleo (*kernel*), la gestión de la información de entrada y salida del mismo, el número de instancias del núcleo a ejecutar y de su lanzamiento para ejecutarse en los *Dispositivos OpenCL* seleccionados.

Cuando un núcleo es lanzado por el *Anfitrión* para ejecutarse, éste define un espacio de índices, que establece el número de instancias a crear (una por índice). Cada una de estas instancias es denominada como *work–item* y es identificada por un índice que es único. Cada uno de estos *work–items* asociados al mismo núcleo ejecutan el mismo código, aunque el flujo de código seguido, debido a las estructuras condicionales y bucles, y los datos a procesar pueden variar en cada uno. Por otro lado, los *work–items* puede ser agrupados en los denominados como *work–groups* que permiten una descomposición más gruesa del espacio, asignando cada *work–item* a un único *work-group*. Cada *work–item* perteneciente a un *work–group* viene identificado por un índice local que lo identifica respecto al resto de *work–items* del mismo *work–group*, conservando su índice global que lo identifica en el conjunto de *work–items* definidos. La ejecución de los *work-items* del mismo *work–group* se realiza de manera concurrente en los *Elementos de Proceso* de una misma *Unidad de Proceso*, lo que permite la colaboración entre ellos, como se verá posteriormente. Del mismo modo que para los *work–items*, los *work–groups* vienen definidos por un identificador único.

Como se ha mencionado anteriormente, al ejecutar un núcleo OpenCL se define un espacio de índices, que se denomina como *NDRange*. Un *NDRange* es un espacio de índices N–Dimensional, donde $N = \{1, 2, 3\}$. De esta manera, el índice global y local de cada *work–item* viene establecido por un vector N–dimensional. Del mismo modo, el índice asociado a cada *work–group* viene dado por un vector de las mismas dimensiones. En la figura 3.7 se muestra un ejemplo de definición de este espacio de índices, mostrando la relación entre los diferentes índices asociados tanto a *work–items* como a *work–groups*. La definición de dicho espacio de índices es realizada de tal manera que se adapte a las características propias del algoritmo paralelo desarrollado, siendo establecida por el desarrollador.



(a) Sólo *Work–Items*. (b) Organización en *Work–Groups*.

Figura 3.7: Ejemplos *NDRange*

Por otro lado, uno de los aspectos fundamentales del modelo establecido en OpenCL se refiere al tratamiento de la memoria. Dicho modelo se muestra en la figura 3.8.

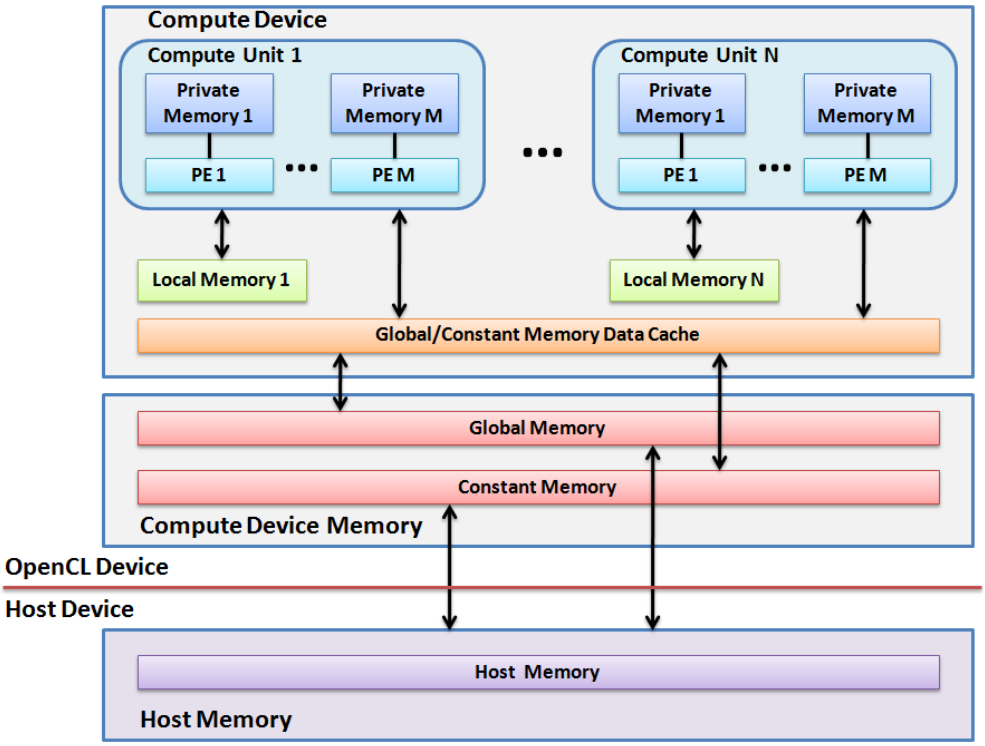


Figura 3.8: OpenCL. Modelo de Memoria

Como se aprecia, existen diferentes tipos de memoria identificados en el modelo OpenCL. Su gestión por parte de los núcleos es una de las tareas más importantes a la hora de diseñar un algoritmo basado en OpenCL, ya que una buena gestión de la misma puede mejorar considerablemente el rendimiento del algoritmo.

Cada *Elemento de Proceso* (PE) tiene algo de memoria privada que es utilizada por el *work–item* para almacenar sus variables automáticas. Dicha memoria es extremadamente rápida, pero su tamaño es muy limitado. El programa *anfitrión* no tiene acceso a dicha memoria.

Por otro lado, los *Elementos de Proceso* que pertenecen a la misma *Unidad de Proceso* comparten la denominada *Memoria Local* que permite compartir información entre los diferentes *work-items*. El acceso sobre esta memoria también es muy rápido. El programa *anfitrión* puede reservar memoria de este tipo, pero no puede hacer lecturas ni escrituras en dicha memoria. Como se verá posteriormente, dicha memoria se puede utilizar para *cachear* información de la memoria global que se utiliza de manera frecuente durante la ejecución de un núcleo concreto.

Adicionalmente, todo dispositivo OpenCL posee una memoria denominada *Memoria Global*, que se identifica con la memoria principal del dispositivo. Dicha memoria permite accesos de lectura y escritura a cualquiera de sus posiciones por parte de todos los *work–items* en ejecución. Aunque dichos accesos pueden ser *cacheados*, dependiendo de las capacidades del dispositivo, presenta una gran latencia por lo que se puede considerar como la más lenta respecto al resto de memorias.

Finalmente, dentro de esta memoria, se identifica la denominada *Memoria Constante* que permite almacenar constantes durante la ejecución de un núcleo así como los propios parámetros del núcleo. La reserva e inicialización de las posiciones de esta memoria es realizada automáticamente por el compilador, como parte del código de inicialización del núcleo, o por el anfitrión al pasar los parámetros del mismo. Esta memoria es más rápida que la *Memoria Global*, pero más lenta que el resto.

Hay que destacar que este modelo de memoria mantiene de manera independiente la memoria correspondiente al *anfitrión*, tal y como se puede apreciar en la figura 3.8. De esta manera, se mantienen aislados ambos elementos fundamentales de la arquitectura OpenCL. Esto conlleva que la reserva de memoria y la transferencia de la información desde y hacia la *Memoria Global* de los *Dispositivos OpenCL* tenga que hacerse de manera explícita a través de la aplicación nativa ejecutada en el *Anfitrión*.

Actualmente, son dos las principales limitaciones que presenta este estándar. La primera, es de reciente publicación, por lo que está en continua evolución, además de existir pocos recursos disponibles (tutoriales, literatura,...). Por otro lado, las implementaciones actuales ofrecidas por los fabricantes generan código que ofrece un menor rendimiento que la versión desarrollada en su lenguaje propietario (por ej. CUDA).

De todas maneras, OpenCL va a ser el modelo utilizado para el desarrollo de la solución al problema planteado en el presente Trabajo Fin de Master, principalmente debido a que se trata de un estándar que permite su portabilidad a diferentes sistemas.

3.3. Paralelización del Algoritmo Mean Shift

Con la aparición de los modelos de programación paralela basados en microprocesadores gráficos, como CUDA, ha habido un gran esfuerzo por obtener algoritmos paralelos que reduzcan el tiempo necesario para la

realización de labores con alto coste computacional, sobre todo en el campo científico. Entre estos algoritmos, han ido apariciendo propuestas asociadas al algoritmo *Mean Shift* destinadas a aplicaciones específicas.

En [35] se propone la aplicación del algoritmo *Mean Shift* paralelizado por medio de CUDA para su aplicación en el tratamiento de imágenes satélite. La principal característica que presenta la propuesta realizada es que trabaja con un espacio de características de gran dimensión al incorporar información de color, información de textura y de forma correspondiente a la imagen tratada. Esto es necesario debido a las características propias de las imágenes a tratar, en las que es necesario trabajar con imágenes de gran resolución. La propuesta realizada se fundamenta en el algoritmo básico *Mean Shift* sin ningún tipo de optimización, en la que cada punto del espacio de características está representado por medio de un vector de 9 dimensiones. La evaluación de la propuesta obtiene un *speed–up* de 21–23 respecto a la versión secuencial de referencia. Hay que destacar que las imágenes tratadas poseen dimensiones de potencia de dos (64x64, 128x128, 512x512) que facilitan considerablemente la gestión de la información en la memoria de las tarjetas gráficas utilizadas para realizar el cálculo, por lo que no se consideran imágenes de otras dimensiones.

Por otro lado, en [11] se propone una solución basada en CUDA y aplicada al tratamiento de volúmenes médicos 3D. En primer lugar, se apoya en la idea establecida en [18] para reducir el número de cálculos por iteración. Para ello, al tratar con información cuyo dominio espacial es tridimensional, se crea una cuadrícula tridimensional uniforme en la que cada una de las celdas tiene una dimensión de $h_s \times h_s \times h_s$. A partir de dicha rejilla, se asigna cada uno de los puntos que componen la información a su celda correspondiente a partir de sus coordenadas espaciales normalizadas por el parámetro h_s. De esta manera, el problema de búsqueda de los vecinos del punto a procesar se limita a los puntos asignados a su propia celda y a sus celdas vecinas. De igual manera que se propone en [18], se realiza una optimización del proceso para reducir el número de iteraciones globales del algoritmo.

De manera resumida, el algoritmo parte de dos conjuntos de puntos S y M, siendo S la información original y M la información en proceso que cumple $M \subseteq S$. A partir de estos conjuntos, el algoritmo realiza los siguientes pasos:

1. Cálculo en paralelo: Se realiza el cálculo de todos los puntos en paralelo, aplicando la expresión 2.1.
2. Transmisión del camino: Si durante el anterior paso, para un punto dado A se obtiene B, y a su vez ya existe ese B cuyo resultado es C en dicha iteración, considera que en la siguiente iteración de A resultará C. Por lo tanto, se asigna directamente a A el punto C. De esta manera, se reduce el número de iteraciones globales.
3. Actualización de M: Se eliminan de M todos aquellos puntos en los que se ha alcanzado su máximo local. De esta manera, se reduce el número de puntos a procesar en paralelo en la siguiente iteración.
4. Se repiten los pasos 1–3 hasta que finaliza el proceso.

Aunque el proceso reduce considerablemente el número de cálculos e iteraciones a realizar, al utilizar la subdivisión espacial de la información y la transmisión del camino, el número de grupos encontrados es

menor o igual al número de grupos encontrados en la versión básica del algoritmo. Este hecho, se puede reducir ajustando los valores $h = \{h_s, h_r\}$ a valores bajos. Por otro lado, esta propuesta sólo se considera para la aplicación especificada, aunque su extensión sea sencilla.

La idea de la subdivisión espacial por medio de una cuadrícula tridimensional uniforme se ha considerado en el presente Trabajo Fin de Master con el objetivo de realizar una organización de los puntos para aumentar el rendimiento del algoritmo.

Relacionado con esto, en [34] se propone la utilización de un Kd–Tree como estructura de datos para distribuir los puntos para reducir el coste por iteración, y está basada en la propuesta [1] que es aplicada al caso del Filtrado Bilateral. Un Kd–Tree (ver figura 3.9) es una estructura de datos de tipo árbol de particionado del espacio que organiza los puntos de un espacio euclídeo, por medio de planos perpendiculares a uno de los ejes del sistema de coordenadas, y almacenando en cada uno de sus nodos un punto. Dicha estructura presenta la principal ventaja de considerar las k dimensiones de la información de entrada.

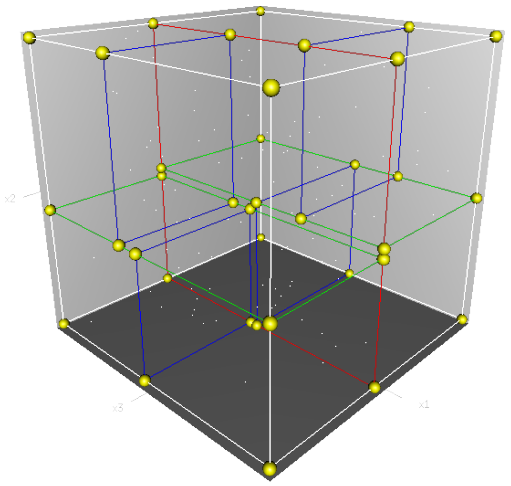


Figura 3.9: Kd–Tree tridimensional

De manera resumida, la propuesta realizada para el caso del algoritmo *Mean Shift* , considera la construcción de esta estructura a partir de los puntos a procesar, siendo la complejidad de dicho proceso $O(NlogN)$. Una vez construido el Kd–Tree, como cada uno de sus nodos sólo almacena un punto, se realiza una operación de submuestreo de los puntos a procesar con el objetivo de obtener dicho punto. De esta manera, el proceso *Mean Shift* se reduce a realizar los cálculos sobre los puntos de los nodos, con la consiguiente redución del coste del proceso. Durante el proceso realizado sobre cada punto del Kd–Tree es necesario realizar consultas sobre la estructura para obtener sus puntos vecinos. Dicho proceso, de complejidad $O(logN)$, se realiza por medio de métodos estocásticos. Una vez concluido el proceso de segmentación, se realiza un proceso de interpolación para obtener el máximo local asociado a cada uno de los puntos a procesar a partir de los puntos resultantes del proceso. La propuesta realizada aplica la paralelización del proceso de construcción y búsqueda en el Kd–Tree, aplicando la solución realizada a la segmentación de

imágenes 2D, video e incluso modelos 3D.

Al evaluar dicha solución, para su posible aplicación al presente Trabajo, se consideraron dos aspectos fundamentales. El primero es que el algoritmo de construcción y búsqueda asociado al Kd–Tree, aplicando programación paralela, no es eficiente al utilizar aspectos del modelo ofrecido por CUDA (p. ej. Operaciones Atómicas) que reducen considerablemente su rendimiento. Por otro lado, la solución planteada parte de la premisa de realizar el proceso de segmentación a partir de una versión muestreada de la información de entrada, por lo que la segmentación final puede presentar errores significantes. Por ello, se rechazó esta estrategia aunque puede ser un posible camino para mejorar las prestaciones del algoritmo *Mean Shift* , sobre todo al tratar información con gran cantidad de puntos.

Finalmente, en [24] se propone una versión paralelizada del algoritmo *Mean Shift* en CUDA para su aplicación como algoritmo de seguimiento. Dicha propuesta consigue un speed–up entre 2 y 3, dependiendo del caso tratado. Aunque, dicho estudio está fuera del objetivo marcado en este Trabajo, es importante destacar que la propuesta realizada divide el proceso en varios pasos que son asociados a distintos núcleos de CUDA. De esta manera, en cada paso, se ejecuta un núcleo que procesa la información en paralelo. Dicha división permite realizar una combinación de partes que se ejecutan secuencialmente y partes que se ejecutan en paralelo, por lo que se consigue descomponer el problema en varias partes, que son más eficientes al combinar ambos mundos. Este aspecto, se ha tenido en cuenta a la hora de abordar la solución al problema planteado.

Como se observa en la actualidad, la mayor parte de las propuestas realizadas para la paralelización del algoritmo *Mean Shift* , utilizando la capacidad de las tarjetas gráficas, se ha limitado al caso de CUDA. Esto es debido principalmente al apoyo realizado por nVIDIA a las universidades con el objetivo de extender la utilización de estos dispositivos como procesadores de cálculo masivo, además de que el código generado por el compilador de nVIDIA para OpenCL es ligeramente menos eficiente que el generado en CUDA (ver [21]).

Capítulo 4

clMeanShift

En el presente capítulo se realiza la descripción de la solución propuesta en el presente Trabajo Fin de Master, denominada como *clMeanShift*. Dicha propuesta tiene en cuenta el ámbito en el que se debe aplicar, así como el conjunto de premisas establecidas y propuestas ya realizadas en diferentes investigaciones.

4.1. Aspectos Generales

El diseño de *clMeanShift* se basa en el desarrollo de una Biblioteca portable que puede ejecutarse en diferentes entornos. Actualmente, la implementación se realiza por medio de una *Dynamic-Link Library* (DLL), destinada al Sistema Operativo Windows, que permite su incorporación a diferentes aplicaciones. Dicha implementación es altamente portable a otros Sistemas Operativos, ya que sólo depende de las Bibliotecas propias de OpenCL.

La infraestructura propuesta es capaz de procesar de manera homogénea imágenes 2D, 3D y 4D, compuestas por valores en escala de grises o por un valor de color en formato RGB, normalizados en el rango [0,0 . . . 1,0]. Estos puntos, que corresponden con los píxeles o vóxeles de la imagen, están representados por un vector r–dimensional cuya dimensión r depende del formato de color utilizado para representar la imagen a procesar.

La entrada básica al proceso de segmentación consiste de una lista de vectores r–dimensionales que representan el conjunto de puntos que componen la imagen. Por un lado, el resultado del proceso está compuesto por la lista de puntos r–dimensionales que representan la imagen ya filtrada, y por otro lado, la lista de etiquetas asociadas a cada uno de los puntos, y que forman el resultado del proceso de etiquetado. De manera adicional, el usuario establece una serie de parámetros encargados de controlar el conjunto de procesos realizados. Dichos parámetros son especificados durante la descripción de cada una de las fases del proceso. De la misma manera, en el Apéndice A se muestra el Manual de Usuario de *clMeanShift*.

La organización de esta lista de puntos depende de la dimensión de la imagen a procesar. En el caso de imágenes 2D, los puntos de la imagen se introducen en la lista por filas. En las imágenes 3D, los puntos se

introducen por corte, y para cada corte por filas (igual que para las imágenes 2D). Finalmente, para el caso de las imágenes 4D, los puntos se introducen por fotograma, entonces por corte, y finalmente por filas. De manera más gráfica, en la figura 4.1 se muestra dicha organización. Internamente *clMeanShift* establece el conjunto de puntos d–dimensionales que definen los vectores de características del conjunto de puntos de la imagen, a partir de la lista de puntos de entrada. La dimensión d de dicho vector depende de las dimensiones espaciales y de color de la imagen a procesar, así como de la dimensión temporal en el caso de las imágenes 4D.

Figura 4.1: Organización interna puntos imagen.

El proceso de segmentación por medio del algoritmo *Mean Shift* que realiza *clMeanShift*, tal y como definió Comaniciu ([7], [8]) para el caso de imágenes 2D, se divide en dos fases fundamentales: Aplicación del filtro *Mean Shift* y Etiquetado para formar los distintos grupos o *Blobs*. De manera gráfica, dicho proceso se puede ver en el ejemplo de la figura 4.2, aplicado sobre una imagen 2D. Como se vió en los capítulos anteriores, dicha aplicación se puede extender fácilmente al caso 3D y 4D.

Durante la fase de filtrado, los puntos d–dimensionales que representran la imagen de entrada son filtrados por medio de la aplicación del filtro *Mean Shift* en el Dominio de Unión (Tiempo–Espacio–Color), tal y como se describe en [7]. El resultado de dicho proceso, es una nueva imagen en la que cada punto original tiene asociado el valor de color de su máximo local o *moda*.

Al aplicar el algoritmo *Mean Shift* es necesario establecer un núcleo para realizar el proceso. En este caso concreto, se considera la aplicación del núcleo de Epanechnikov al presentar el mínimo error cuadrático

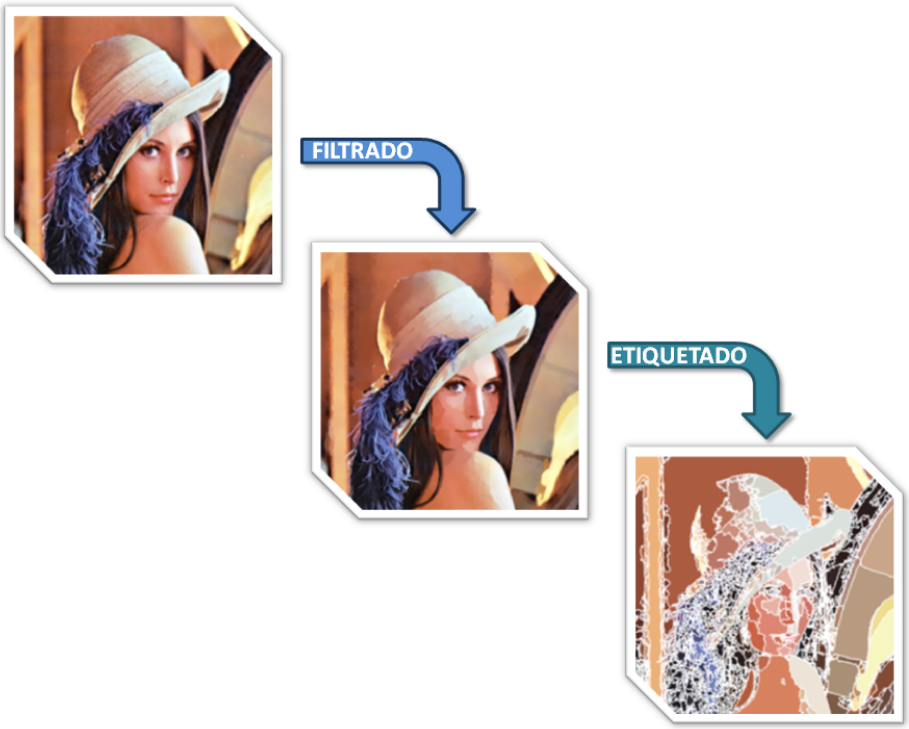


Figura 4.2: Ejemplo Segmentación *Mean Shift* . Imagen 2D

integrado medio (MISE). Este núcleo viene definido por la expresión 4.1, donde c_d es el volumen de la esfera d–dimensional unitaria.

$$K_E(x) = \begin{cases} \frac{1}{2}c_d^{-1}(d+2)(1-x^Tx) & \text{si} x^Tx < 1 \\ 0 & \text{en otro caso} \end{cases} \tag{4.1}$$

A partir de este núcleo, y obteniendo la expresión para el estimador del gradiente de densidad (ver [7] para detalles), se obtiene la expresión 4.2 que define el *Vector Mean Shift* para este núcleo.

$$M_h(x) = \frac{1}{n_x} \sum_{x_i \in S_h(x)} x_i - x \tag{4.2}$$

De esta manera, el proceso iterativo para obtener el máximo local de cualquier punto del espacio d–dimensional viene dado por la expresión 4.3, donde $x_{i,1}$ es cada uno de los puntos de la imagen original y n_k es el número de puntos que están dentro de la ventana $S_h(x_{i,k})$ y centrado en $x_{i,k}$. Adicionalmente, se considera que los puntos originales del Espacio de Características han sido normalizados con el valor de h, aplicando el correspondiente valor de h dependiendo del dominio de cada coordenada del vector.

$$y_{i,k+1} = \frac{1}{n_x} \sum_{x_i \in S_h(y_{i,k})} x_i \tag{4.3}$$

Como se puede apreciar, al aplicar este núcleo, el proceso iterativo seguido para cada punto consiste en el cálculo del valor medio del conjunto de puntos que quedan dentro de la ventana en torno al punto a tratar, operación equivalente a un proceso de convolución en el que todos los puntos tienen el mismo peso.

De esta manera, el algoritmo básico para realizar el proceso de filtrado aplicando *Mean Shift* , para el caso del núcleo de Epanechnikov, se puede reescribir tal y como se muestra en el algoritmo 4.2.3.

Algorithm 2: Algoritmo *Mean Shift* aplicando núcleo de Epanechnikov.

Entrada: $\{x_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen original en el dominio espacio-color, normalizados con los valores h_i.

Salida: $\{z_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen filtrada en el dominio espacio-color

Para cada $i = 1 \dots n$ **hacer**

Inicializar $j = 1$ e $y_{i,1} = x_i$.

Calcular $y_{i,j+1} = \frac{1}{n_x} \sum_{x_i \in S_h(y_{i,k})} x_i$, $j \leftarrow j + 1$ hasta converger.

Asignar $z_i = (x_i^s, y_{i,conv}^r)$

Por otro lado, la fase de etiquetado consiste en la agrupación de los distintos puntos de la imagen filtrada que componen cada *moda* asignándoles el mismo identificador o etiqueta. De esta manera, la imagen quedará divida en las diferentes partes.

Como resultado final del proceso completo, se obtiene por un lado la imagen filtrada y, por otro lado, la etiqueta asignada a cada punto de la imagen.

4.2. Proceso de Filtrado

El proceso de filtrado parte de la lista de puntos que componen la imagen. El conjunto de cálculos se realizan por medio de un algoritmo secuencial, combinado con procesos ejecutados en paralelo por medio de una serie de núcleos de OpenCL. De esta manera, se busca obtener el mejor rendimiento posible, explotando aquellas partes del algoritmo básico que pueden ser paralelizadas. Así, el proceso es dividido en una serie de fases, tal y como se identifican en la figura 4.3.

Básicamente las fases de *Preprocesado* y de *Creación de la Cuadrícula Tridimensional* tienen como objetivo adaptar y organizar la información de entrada para realizar el proceso *Mean Shift* propiamente dicho.

Por otro lado, la fase de *Bucle Mean Shift* es la encargada de realizar el proceso iterativo que tiene como misión el cálculo del máximo local para cada uno de los puntos de la imagen.

Finalmente, la fase de *Postproceso* tiene la misión de obtener el resultado del proceso de filtrado.

4.2.1. Preprocesado

La fase de Preprocesado tiene como objetivo la normalización de la lista de puntos del Espacio de Características que componen la imagen de entrada. Para ello, esta fase está dividida en una serie de pasos,

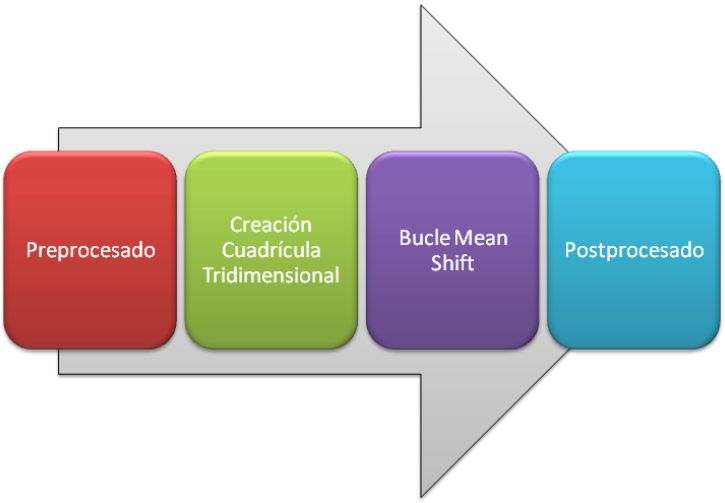


Figura 4.3: Proceso de Filtrado.

tal y como se muestra en la figura 4.4.

Figura 4.4: Preprocesado.

En primer lugar, se crea el vector de características de cada punto, a partir de su valor de color y su posición espacial, que es calculada a partir de su índice dentro de la lista de puntos a procesar. El valor de color incluido en el vector de características, que representa un color en RGB o en escala de grises, es convertido al formato de $L*u*v$ (CIELUV). Este formato de color presenta la característica de ser independiente del dispositivo de visualización, así como ser "*Perceptualmente Uniforme*", es decir, si se produce una pequeña modificación en una de las componentes de color, dicha modificación tiene el mismo efecto en todo el rango de valores posible. A diferencia de otros modelos de color, como el RGB, esta propiedad permite establecer una métrica Euclídea que define la diferencia entre dos valores de color de manera lineal en el Dominio del Color. Del mismo modo, el modelo de color denominado $L*a*b$ (CIELAB) cumple dicha propiedad, por lo que se puede utilizar de manera análoga. Hay que destacar que, para el caso de que el valor esté en escala de grises, sólo será necesario considerar el valor de L en el vector de características. El conjunto de operaciones necesarias para convertir de un mapa de color a otro se especifica en [25].

En segundo lugar, se realiza la normalización del vector de características de cada punto, ya que se va a operar en el Dominio de Unión, por lo que al trabajar en coordenadas normalizadas se reduce el número de

operaciones a realizar, además de ser necesario para distribuir los puntos en la Cuadrícula Tridimensional. Dicho proceso consiste en la división de cada una de las coordenadas que componen el vector de características por el ancho de la ventana (h) correspondiente a su Dominio. Dichos valores son establecidos por el usuario como parte del conjunto de parámetros del algoritmo y controlan la resolución del proceso en cada Dominio.

Finalmente, se realiza el cálculo del valor mínimo y máximo de los valores de las tres primeras coordenadas del vector de características de los puntos de la imagen. Dichos valores serán utilizados en la siguiente fase para el cálculo de las dimensiones de la Cuadrícula Tridimensional a crear.

Como se aprecia en la figura 4.4, el conjunto de tareas se realiza por la combinación de dos núcleos OpenCL, siendo los parámetros de entrada: la lista de puntos de la imagen, los valores h para cada dominio (h_s, h_r y h_t si fuera necesario) y el número de puntos a procesar (n).

El primer núcleo, denominado *NormalizedData*, se encarga de la transformación de los puntos de entrada en su representación equivalente en el Espacio de Características, asi como de las labores de conversión de color y normalización de este vector. Para ello, cada uno de los puntos es asignado a un único *work–item* que realiza ambas operaciones, por lo que es necesario lanzar por lo menos tantos *work–items* como puntos son necesarios tratar. Para maximizar la ocupación del procesador, tal y como se describirá en el siguiente capítulo, éstos son agrupados en *work–groups* compuestos por un número fijo de *work–items* (256) cada uno. Por ello, si el último *work-group* tiene *work–items* que no tienen puntos asignados, no harán ningún tipo de proceso. De manera gráfica, esta idea es mostrada en la figura 4.5.

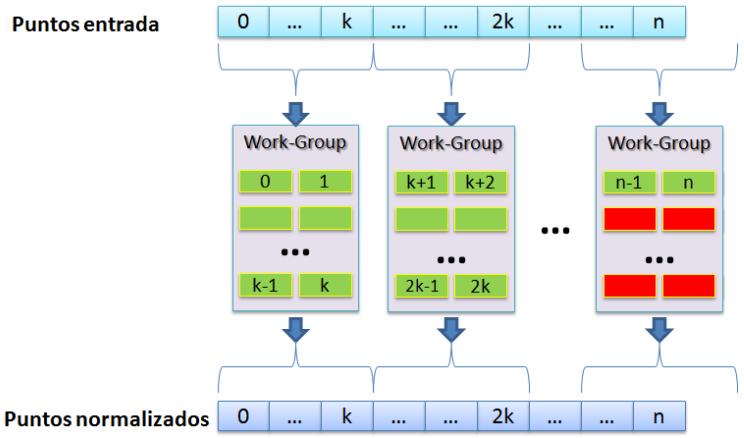


Figura 4.5: Preprocesado.

En paralelo a este proceso de conversión y normalización, cada *work–group* obtiene el valor máximo y mínimo de las tres primeras coordenadas que componen los vectores de características que han procesado sus *work-items*. Dichos valores son almacenados en un vector en el que cada *work-group* almacena su resultado en la posición que identifica al grupo.

Una vez que se concluye la ejecución del primer núcleo, se ejecuta el segundo núcleo, denominado *GetGridLimit*, cuya misión consiste en obtener un único valor mínimo y máximo para cada una de las tres

primeras coordenadas del vector de características. Dicho proceso corresponde con un problema clásico de *reducción* cuya implementación en paralelo ha sido ampliamente estudiada y aplicada.

Como resultado de la fase de Preprocesado, se obtiene la lista de vectores de características normalizados y los límites de la Cuadrícula Tridimensional a crear durante la siguiente fase.

4.2.2. Creación de la Cuadrícula Uniforme Tridimensional

Como se ha visto en el algoritmo 4.2.3, al realizar el proceso iterativo para obtener el máximo local asociado a cada uno de los puntos de la imagen a filtrar, es necesario establecer el conjunto de puntos que quedan dentro de la ventana $S_h(x)$, definida a partir de los valores h y del punto x tratado. Este proceso de *Búsqueda de Vecinos* es realizado para cada iteración y para cada punto, por lo que es necesario establecer una organización de los puntos de tal manera que su coste sea bajo.

Al trabajar en el Dominio de Unión, esto se traduce en considerar los puntos que se encuentran a una distancia menor de la ventana h, para cada uno de los dominios (tiempo, espacio, color). En concreto, al trabajar en coordenadas normalizadas dicha condición se reduce a considerar los puntos que se encuentran a una distancia menor de 1, para cada uno de los Dominios que componen el Dominio de Unión. En el caso de las imágenes 2D y 3D, teniendo en cuenta las coordenadas que componen el Dominio de Unión, dicha condición viene dada por la expresión 4.4. Para el caso de las imágenes 4D, dicha expresión viene dada por 4.5.

$$x_i \in S_h(x) \Leftrightarrow |x_i^s - x^s| < 1 \wedge |x_i^r - x^r| < 1 \tag{4.4}$$

$$x_i \in S_h(x) \Leftrightarrow |x_i^t - x^t| < 1 \wedge |x_i^s - x^s| < 1 \wedge |x_i^r - x^r| < 1 \tag{4.5}$$

Como se vió en el capítulo anterior, varias han sido las estructuras de datos utilizadas para llevar a cabo esta tarea (Cuadrícula Uniforme, Kd–Tree). En este caso concreto, se ha optado por la construcción de una Rejilla o Cuadrícula Tridimensional Uniforme, que es una estructura de subdivisión espacial regular sencilla en la cual existen métodos eficientes para realizar su construcción por medio de un algoritmo paralelo, al ser aplicado a otros problemas ([19] y [31]), además de tener un bajo coste de consulta.

Conceptualmente, una Cuadrícula Tridimensional Uniforme divide el espacio tridimensional en una serie de celdas de igual tamaño distribuidos de manera uniforme a lo largo de los tres ejes de coordenadas, tal y como se muestra en la figura 4.6. La idea básica es que si se tiene un conjunto de puntos en el espacio tridimensional que se divide espacialmente por medio de esta estructura, cada uno de éstos pertenece a una de las celdas de la Cuadrícula.

Aplicando esta idea al Espacio de Características, se puede realizar una distribución de dicha información considerando el espacio tridimensional que definen las coordenadas $< x^1, x^2, x^3 >$ del vector de características, y de esta manera asignar una celda a cada uno de los puntos de la imagen. La dimensión de cada

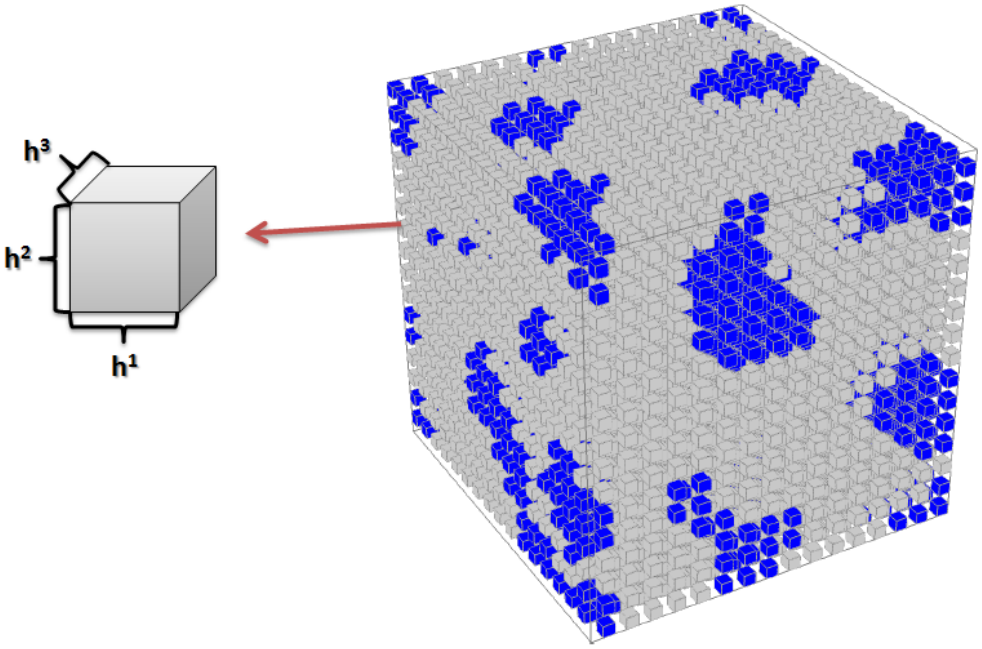


Figura 4.6: Cuadrícula Uniforme Tridimensional.

celda viene definida por el valor de h asociado a cada una de estas coordenadas, y de esta manera, el espacio tridimensional definido queda divido espacialmente por esta cuadrícula.

Por lo tanto, durante esta fase del proceso de filtrado se identifican una serie de pasos, tal y como se muestra en la figura 4.7.

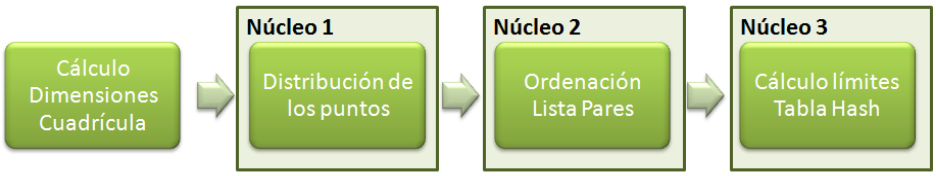


Figura 4.7: Creación Cuadrícula Tridimensional Uniforme

En primer lugar, se calcula el número total de celdas de la cuadrícula por dimensión, a partir de los límites calculados durante la fase de preprocesado. Dichos valores vienen definidos por la expresión 4.6, donde $\lceil \bullet \rceil$ indica que el resultado se redondea al entero superior.

$$< N_z, N_y, N_x >=< \lceil max(x^1) - min(x^1) \rceil, \lceil max(x^2) - min(x^2) \rceil, \lceil max(x^3) - min(x^3) \rceil > \quad (4.6)$$

Cada una de las celdas que componen la cuadrícula, está identificada inequivocamente por una posición relativa y un valor *hash*. Se considera como posición relativa la definida por un vector tridimensional que

representa la posición de la celda, dentro de la cuadrícula, respecto a los tres ejes de coordenadas. Por otro lado, el valor de *hash* se considera como un identificador único de la celda, cuyo valor depende de su posición relativa y cuyo significado es similar al valor de las *Tablas Hash*. Si se considera una celda cuya posición relativa es $< C_z, C_y, C_x >$, en la expresión 4.7 se muestra el cálculo de su valor Hash asociado.

$$C_{rel} =< C_z, C_y, C_x > \Rightarrow Hash(C_{rel}) = (C_z * N_y * N_x) + (C_y * N_x) + C_x \tag{4.7}$$

Una vez definidas las dimensiones de la cuadrícula, el proceso de distribución de los puntos se realiza por medio de un núcleo que realiza esta labor en paralelo. Dicho núcleo ejecuta un *work-item* por cada uno de los puntos de la información de entrada, siguiendo la misma estrategia que la indicada para el caso del primer núcleo de la fase de Preprocesado. Como ya se ha mencionado, la celda a la que corresponde cada punto se obtiene a partir de las coordenadas normalizadas $< x^1, x^2, x^3 >$ de su vector de características, aplicando el cálculo indicado en la expresión 4.8, donde $INT(x)$ indica la parte entera de x.

$$C(x) =< INT(x^1 - min(x^1)), INT(x^2 - min(x^2)), INT(x^3 - min(x^3)) > \tag{4.8}$$

De esta manera, al trabajar con coordenadas normalizadas, la búsqueda de los puntos pertenecientes a $S_h(x)$ para un punto x se limitará a los puntos de su propia celda junto con las 26 celdas adyacentes de ésta, es decir, un número fijo de celdas. Teniendo en cuenta la manera en que se construye la cuadrícula, así como las dimensiones de cada una de sus celdas, se garantiza que los puntos pertenecientes al resto de celdas no influyen en el cálculo del máximo local del punto x, al estar a una distancia Euclídea mayor de uno.

Una vez obtenida la posición relativa de la celda, se calcula el valor Hash asociado, que es almacenado junto con el índice que identifica al punto asignado, en una Lista de Pares, tal y como se muestra en la figura 4.8. El objetivo es establecer un formato compacto para almacenar la configuración de la cuadrícula, teniendo en cuenta que cada celda puede tener un número variable de puntos asignados (≥ 0).

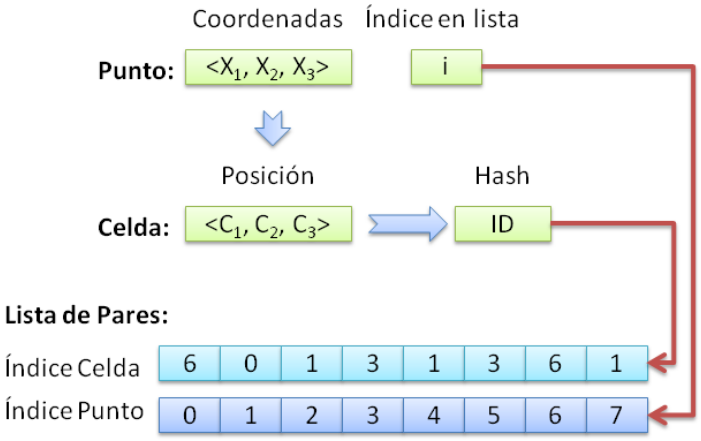


Figura 4.8: Cuadrícula 3D. Asignación de los puntos en las celdas de la cuadrícula.

Una vez concluido el proceso de distribución, es necesario ordenar la Lista de Pares por identificador de celda. De esta manera, se agrupan aquellos puntos que corresponden a la misma celda de la cuadrícula, tal y como se muestra en la figura 4.9.

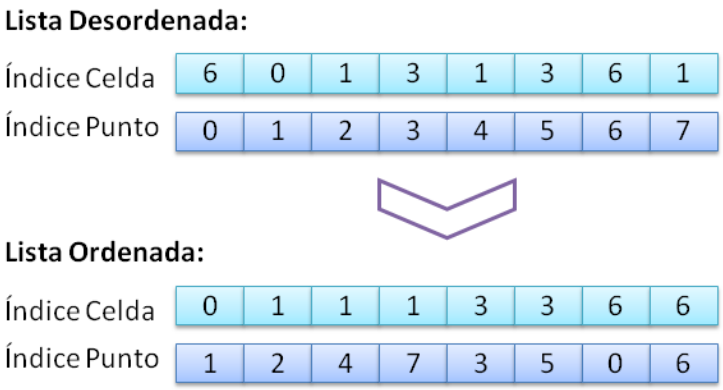


Figura 4.9: Cuadrícula 3D. Ordenación lista de pares.

Este proceso se realiza por medio del algoritmo de *Ordenación Bitónica* [2] que presenta la principal característica de haber sido diseñado especialmente para su ejecución en paralelo, teniendo una complejidad de $O(log^2(n))$. En este caso, se utilizan dos implementaciones distintas dependiendo de la cantidad de datos a ordenar.

Por un lado, se utiliza la implementación propuesta por AMD. Dicha implementación forma parte del conjunto de ejemplos del SDK que proporciona para el desarrollo de aplicaciones basadas en OpenCL, pero presenta la limitación de sólo ser válida para un número de datos potencia de dos. Esto es debido a que este algoritmo de ordenación realiza particiones de los datos a ordenar en potencias de dos, además de ser más natural de implementar en OpenCL para arquitecturas *many–cores*.

Por ello, se ha desarrollado una segunda implementación del algoritmo que es ejecutada en el *anfitrión*. Dicha implementación no presenta las limitaciones mostradas en el anterior caso, siendo fácil de implementar como un proceso en un único hilo de ejecución. La principal desventaja que presenta esta segunda opción es que es necesario transferir la lista de pares de la memoria del *Dispositivo OpenCL* a la memoria del *Anfitrión*, realizar el proceso de ordenación y, finalmente, volver a escribir la lista ordenada en la memoria del *Dispositivo OpenCL*. Por ello, esta segunda opción presenta un mayor coste derivado de las operaciones de transferencia de la información, así como, de su ejecución en un único hilo.

La selección de un algoritmo u otro se realiza en tiempo de ejecución a partir del tamaño de la Lista de Pares a ordenar.

Finalmente, una vez que la lista queda ordenada, se realiza un último proceso cuyo objetivo es facilitar la consulta de la lista de pares durante la siguiente fase del proceso de filtrado. Para ello, se crea una lista de tamaño igual al número total de celdas, denominada *Límites celdas*, que describe la configuración completa

de la cuadrícula. Cada índice de la lista corresponde con el valor Hash de una celda de la cuadrícula, y se almacena para cada celda el índice inicial y final de la Lista de Pares que corresponden con el tramo de la lista donde están sus puntos asociados, tal y como se ve en la figura 4.10.

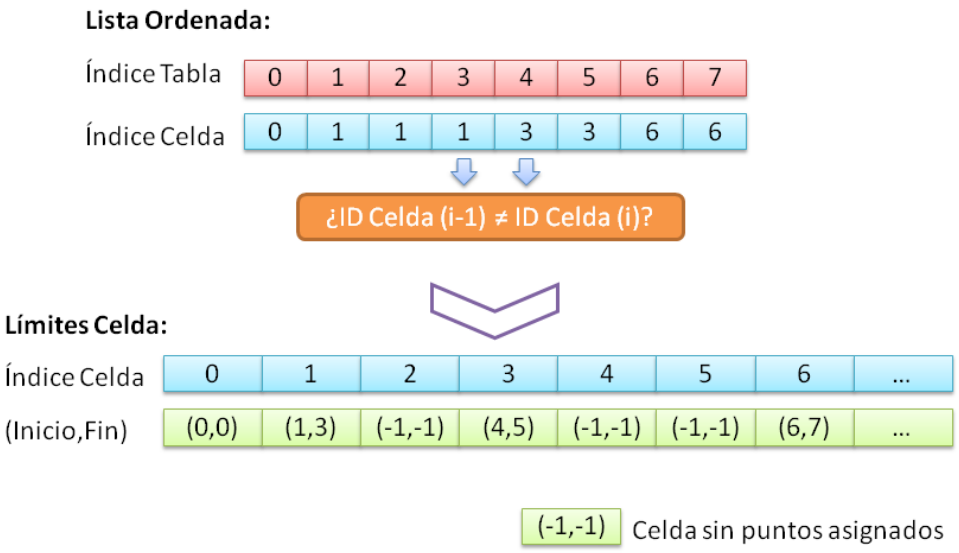


Figura 4.10: Cuadrícula 3D. Obtención límites celdas.

Este proceso es realizado por medio de un nuevo núcleo cuyos *work-items* recorren la Lista de Pares obteniendo los índices indicados. Para ello, al estar la Lista ordenada por celda, tan sólo es necesario detectar los índices consecutivos en los que hay cambio de celda. Por otro lado, como se aprecia en la figura 4.10, en el caso de que alguna de las celdas de la cuadrícula no tenga asociado ningún punto, en su posición correspondiente se le asigna un valor especial indicando este hecho.

El resultado final de esta fase es la Lista de Pares que relaciona cada celda con sus puntos, y la lista que que representa los límites de cada celda dentro de la anterior lista.

4.2.3. Bucle *Mean Shift*

Esta fase es el núcleo del proceso de Filtrado por medio del algoritmo *Mean Shift* . Dicho proceso consiste básicamente en la aplicación del algoritmo 4.2.3 para localizar el máximo local asociado a cada uno de los puntos del Espacio de Características. Como ya se ha visto, en las anteriores fases, se ha realizado la adaptación y organización de la información de tal manera que se facilita este proceso iterativo, y a la vez, de alto coste computacional.

La paralelización de este proceso es fundamental para obtener una solución que mejore las prestaciones de los algoritmos desarrollados hasta la fecha. Varias son las estrategias de paralelización posibles, teniendo en cuenta las características propias de la arquitectura disponible para la ejecución del proceso.

Completando el algoritmo visto al principio de este capítulo, y considerando la utilización de la Cuadrícula

Tridimensional Uniforme, este algoritmo se puede reescribir tal y como se muestra en el algoritmo 3.

Algorithm 3: Algoritmo *Mean Shift* con Cuadrícula Tridimensional

Entrada: $X = \{x_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen original en el dominio espacio-color, normalizados con los valores h_i.

Entrada: G =Cuadrícula tridimensional uniforme

Salida: $Z = \{z_i\}_{i=1..n}$ son los puntos d-dimensionales de la imagen filtrada en el dominio espacio-color

Para $i = 1 \dots n$ **hacer**
- $k \leftarrow 1$
- $y_{i,k} \leftarrow x_i$
- **Repetir**
  - $n_x \leftarrow 0$
  - *% Se itera por cada punto perteneciente a la celda a la que pertenece $y_{i,k}$ y sus celdas adyacentes*
  - **Para cada** x_j *en* $G(y_{i,k}) \cup Adj(G(y_{i,k}))$ **hacer**
    - **Si** $|x_j - y_{i,k}| < 1$ **entonces**
      - $y_{i,k+1} \leftarrow y_{i,k+1} + x_j$
      - $n_x \leftarrow n_x + 1$
  - $y_{i,k+1} \leftarrow \frac{y_{i,k+1}}{n_x}$
  - $|Mh| \leftarrow |y_{i,k+1} - y_{i,k}|$
  - $k \leftarrow k + 1$
- **Hasta que** $|Mh| < \epsilon$;
- $z_i = (x_i^s, y_{i,k}^r)$;

Como se puede apreciar, dicho algoritmo consta de tres bucles anidados. El bucle exterior itera sobre el conjunto de vectores normalizados que forman la imagen a segmentar. Por otro lado, el bucle intermedio es el que se encarga de realizar el proceso iterativo de ascenso de gradiente para cada punto hasta converger a su máximo local asociado. Finalmente, el bucle interno es el encargado de calcular la nueva posición para dicho punto. Por lo tanto, se pueden considerar varias estrategias posibles para paralelizar este proceso, teniendo en cuenta estos bucles y las características propias del proceso.

Una primera estrategia podría consistir en paralelizar el primer bucle creando un *work–item* por punto, de tal manera que realice el proceso de ascensión del gradiente de manera independiente. De esta manera, cada *work–item* procesa un punto y obtiene su máximo local asociado. Esta estrategia es posible ya que el proceso a realizar para cada punto de entrada no depende del resto. La cuestión es que el proceso hasta alcanzar el máximo local no conlleva el mismo número de iteraciones para todos los puntos, por lo que cada *work–item* termina de manera independiente. Realizar este tipo de ejecución sobre una arquitectura SIMT (*Single Instruction – Multiple Threads*) supone una fuerte penalización en el coste del proceso, ya que se crean divergencias en el flujo del código, acentuadas por el resto de bucles que debería ejecutar cada

work–item, lo que hace que dicha estrategia no sea viable.

Por otro lado, se puede considerar la paralelización del bucle interno, de tal manera que el resto de bucles se mantenga de manera secuencial, con lo que sólo se procesa un punto cada vez. Al utilizar la Cuadrícula Tridimensional para acelerar el proceso de búsqueda de los puntos que participan en el cálculo, sólo es necesario considerar los puntos correspondientes a la propia celda del punto procesado, junto con el resto de celdas adyacentes. Por ello, la paralelización de este bucle consiste en la creación de un *work–item* por cada una de las celdas a visitar, en concreto 27 celdas. De esta manera, cada *work–item* sólo se encarga de acumular los puntos de su celda correspondiente que se encuentran a la distancia válida respecto al punto de referencia. Una vez que todos los *work–items* completan el proceso, es necesario realizar un proceso de reducción para obtener el valor medio final y así concluir una iteración de ascenso de gradiente. Dicho proceso se repite hasta alcanzar el máximo local del punto, con lo que se comienza el proceso para un nuevo punto.

Esta estrategia se puede extender considerando la posibilidad de procesar varios puntos a la vez, siguiendo la misma filosofía. Para ello, se crean *work–groups* de 27 *work–item*, de tal manera que cada *work–groups* se encargue del cálculo del máximo local de un punto en concreto. De esta manera, se aumenta la paralelización del algoritmo general.

Ambas estrategias, presentan el problema de que el número de puntos a tratar por cada celda no tiene por qué ser el mismo, por lo que los diferentes *work–items* de ejecución terminarían en diferentes momentos. Adicionalmente, por limitación de la propia arquitectura, el número de *work–groups* que se pueden ejecutar a la vez dentro del procesador está limitado (por ejemplo, actualmente sólo 8 grupos en el caso de las tarjetas gráficas de nVIDIA), tal y como se verá en el capítulo siguiente. Teniendo en cuenta este aspecto, y que cada grupo necesita un número bajo de hilos, la ocupación del procesador sería baja ($< 25\,\%$).

Por todo ello, se ha optado por realizar una estrategia híbrida que consiste en una pequeña modificación sobre el algoritmo 3. Para conseguir la mayor ocupación del procesador, se necesita poder ejecutar el mayor número de *work-items* en paralelo, teniendo en cuenta la manera en que se ejecutan los mismos, es decir, SIMT. Por ello, se plantea intercambiar los dos primeros bucles que componen el algoritmo, tal y como se muestra en el esquema de la figura 4.11. De esta manera, se puede paralelizar, por medio de un núcleo, cada iteración del proceso de ascenso del gradiente para realizar dicho cálculo en paralelo para todos los puntos. Este núcleo crea, para cada iteración, tantos *work-items* como puntos haya en la imagen a filtrar.

Al adoptar una estrategia en la que se realiza una iteración del proceso de ascenso del gradiente para todos los puntos a la vez, es necesario almacenar los distintos puntos que va alcanzando el proceso para cada uno de los puntos procesados. Para ello, se utiliza una lista que va almacenando los distintos vectores de características obtenidos durante el proceso, y que es la utilizada durante las iteraciones del proceso. De esta manera, al alcanzar el máximo local de cada punto, dicho valor queda almacenado en esta lista. Por otro lado, la lista de los vectores de características a procesar es utilizada para realizar el cálculo del valor medio asociado a cada punto, aplicando la cuadrícula generada para realizar el proceso de búsqueda de los puntos vecinos.

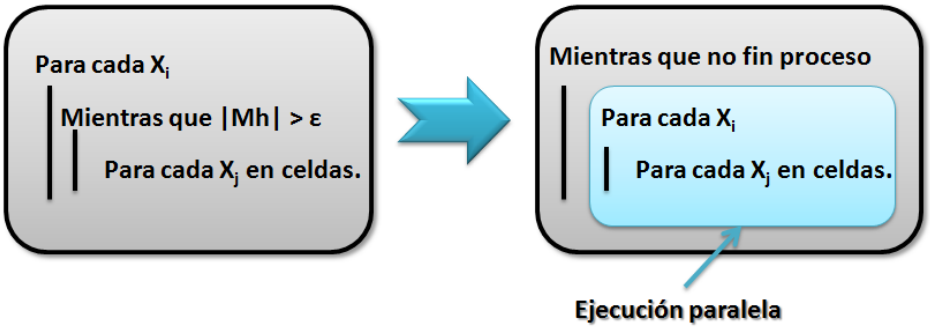


Figura 4.11: Intercambio de bucles para la paralelización del algoritmo

Como se aprecia en la figura, el intercambio de los dos bucles modifica el significado del nuevo bucle exterior. Como se ha indicado anteriormente, el proceso de ascenso de gradiente no necesita el mismo número de iteraciones para cada punto a procesar. Es necesario, por lo tanto, establecer un criterio que indique cuando hay que concluir dicho proceso. Por ello, la condición del primer bucle se modifica de tal manera que el proceso iterativo continúa mientras haya puntos para los que no se haya alcanzado su máximo local, o bien, se alcance un número de iteraciones máximas establecidas como parámetro del proceso (protección).

Por ello, se establece lo que se denomina como *Cola de Proceso*, estructura que mantiene en cada iteración el conjunto de puntos a procesar. Esta estructura almacena los índices de aquellos puntos de la imagen para los que aún no se ha alcanzado su máximo local.

En la primera iteración del bucle Mean Shift, todos los puntos de la imagen son insertados en la *Cola de Proceso*, de tal manera que durante dicha interación son procesados todos los puntos. Al realizar el paso de la iteración para cada punto de la cola, además de obtener el valor del vector *Mean Shift* a partir del punto obtenido en el cálculo, se obtiene su módulo. Si éste valor es inferior a un umbral ϵ, establecido por el usuario como parámetro, se considera que se ha alcanzado el máximo local asociado a dicho punto, por lo que se marca en la cola como concluido.

De esta manera, el bucle Mean Shift es realizado por la combinación de dos núcleos OpenCL (ver figura 4.12). El primer de ellos es el encargado de realizar la iteración para cada uno de los puntos de la *Cola de Proceso*. Una vez que concluye este núcleo, se lanza otro encargado de actualizar la *Cola* eliminando de ésta todos aquellos puntos cuyo máximo local ha sido alcanzado, de tal manera, que no se realice el cálculo sobre éstos durante la siguiente iteración del bucle. Una vez concluido dicho proceso, se comienza una nueva iteración del bucle, siempre y cuando queden puntos por procesar o se haya alcanzado el número máximo de iteraciones.

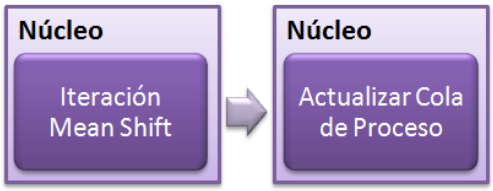


Figura 4.12: Bucle *Mean Shift* .

Así, al utilizar la *Cola de Proceso*, va disminuyendo progresivamente el número de *work–items* a lanzar en cada una de las iteraciones del cálculo *Mean Shift*, ya que tan sólo serán necesarios tantos como puntos queden en la cola, y por lo tanto, se reduce progresivamente el tiempo necesario para realizar una iteración del Bucle Mean Shift. La configuración del número de *work–items* a lanzar y su distribución en diferentes *work–groups* se realiza de tal manera que se maximize la ocupación del procesador.

De manera gráfica, en la figura 4.13 se muestra de manera resumida el conjunto de elementos que participan en el proceso completo.

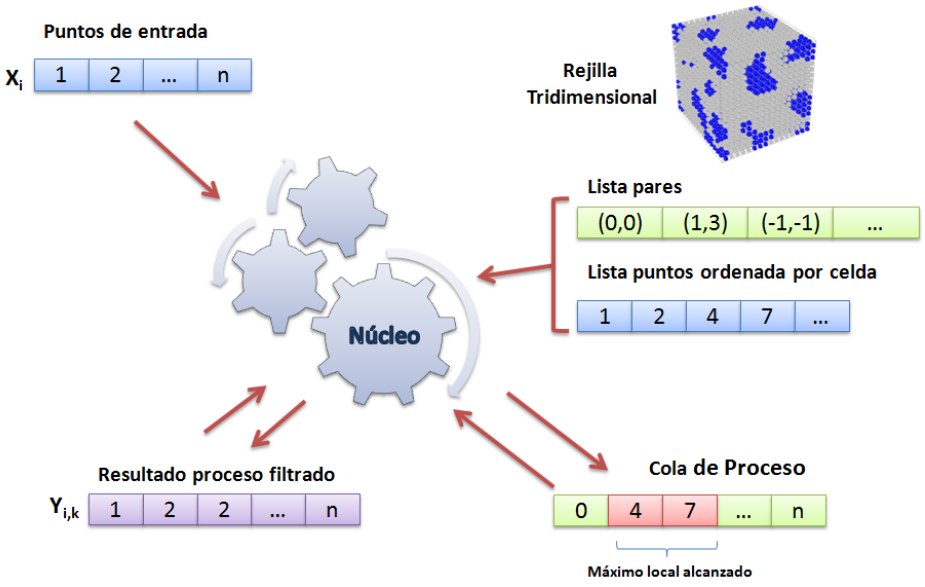


Figura 4.13: Elementos que participan en el proceso

Al terminar de procesar todos los puntos que componen la imagen a filtrar, en el vector de resultados intermedios quedan almacenados los máximos locales asignados a cada punto, que serán recuperados durante la siguiente fase del proceso.

4.2.4. Postprocesado

Una vez obtenidos todos los máximos locales de los distintos puntos de la imagen se realiza la última fase de filtrado cuyo objetivo es obtener la imagen filtrada. Durante este proceso se realizan una serie de labores,

tal y como se muestra en la figura 4.14.

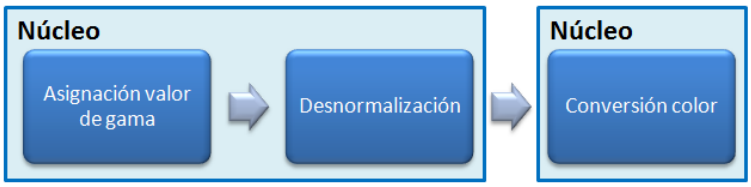


Figura 4.14: Postproceso

Por un lado, a cada uno de los puntos de la imagen a filtrar se le asigna la información de color ($y^r_{i,conv}$) de su máximo local obtenida durante la anterior fase. Por otro lado, se realiza la desnormalización y conversiones necesarias para obtener la imagen resultante en el formato correcto.

Dicho proceso es realizado por medio de dos núcleos que se encargan de realizar todas estas labores a partir de la información resultante del proceso de filtrado. En ambos casos se lanza un *work-item* por punto de la imagen, siguiendo la misma filosofía indicada en casos anteriores para maximizar la ocupación del procesador.

Para el caso del primer núcleo, cada *work–item* selecciona el máximo local correspondiente a su punto y extrae la información de color del mismo. Como dicho valor está normalizado por el valor h_r, para obtener el valor de color, multiplica el vector extraído por este valor.

Por otro lado, el segundo núcleo, convierte dicho valor al formato de color o escala de grises correspondiente invirtiendo la transformación realizada al principio del proceso de filtrado.

Como se puede observar, todo el proceso se podría haber realizado por medio de un único núcleo. Se ha decidido separar el proceso de conversión del color ya que durante el proceso de etiquetado es conveniente utilizar la información en el mapa de color $L * u * v$ al permitir establecer una métrica Euclídea.

En el caso de que sólo se desee realizar el proceso de filtrado de la imagen, todo el Postprocesado es realizado por un único núcleo para reducir el coste computacional del proceso.

Como resultado del Postprocesado, se obtiene la imagen filtrada por medio del algoritmo *Mean Shift* , que sirve como base para realizar el proceso de etiquetado.

4.3. Proceso de Etiquetado.

El proceso de etiquetado se identifica como la tarea de determinar y etiquetar grupos de nodos conectados en un gráfico. Esta definición general puede ser particularizada para este caso, identificando la tarea de etiquetado como aquella en la que, a partir de la imagen filtrada por medio de la algoritmo *Mean Shift* , determina y etiqueta el conjunto de puntos de la imagen que tienen relación entre sí, y conlleva la identificación de las diferentes partes que componen dicha imagen (*Blobs*).

Dicho proceso ha sido profundamente estudiado como problema resoluble por medio de un algoritmo

paralelo, y actualmente se ha llevado al campo de su resolución por medio de microprocesadores *many-cores*. En el artículo de Hawick y otros [15] se realiza un extenso estudio de diferentes propuestas de etiquetado, aplicando dichos métodos tanto a grafos arbitrarios como a mallas hipercúbicas (como es este caso concreto), y utilizando CUDA para llevar a cabo su implementación.

En concreto, para este caso, se ha considerado la utilización del método basado en una *Lista de Equivalencia* (Algoritmo 8 de [15]), algoritmo de multipasada que presenta las mejores prestaciones respecto a sus algoritmos equivalentes.

A continuación, se va a realizar la descripción de dicho algoritmo, ilustrándolo para el caso concreto de imágenes 2D. Dicho caso se puede extender fácilmente a imágenes 3D y 4D.

La idea parte de considerar la imagen como un grafo compuesto por una serie de nodos, es decir, los píxeles que componen la imagen 2D (voxeles en el caso 3D). Cada nodo de este grafo tiene asignada una propiedad, en este caso un valor de color. Esta propiedad permite verificar si dos nodos son similares, por lo que se pueden considerar que están conectados. En este caso concreto el criterio establecido para dicha similitud es la distancia Euclídea entre ambos valores de color, tal y como se muestra en la expresión 4.9, donde ϵ es un umbral establecido como parámetro.

$$|x_1^r - x_2^r| \leq \epsilon \tag{4.9}$$

Del mismo modo, la imagen define una topología que establece la conexión espacial entre los diferentes nodos, en este caso se considera para cada píxel de la imagen los 8 píxeles vecinos alrededor suyo (26 voxeles en el caso 3D), tal y como se muestra en la figura 4.15.

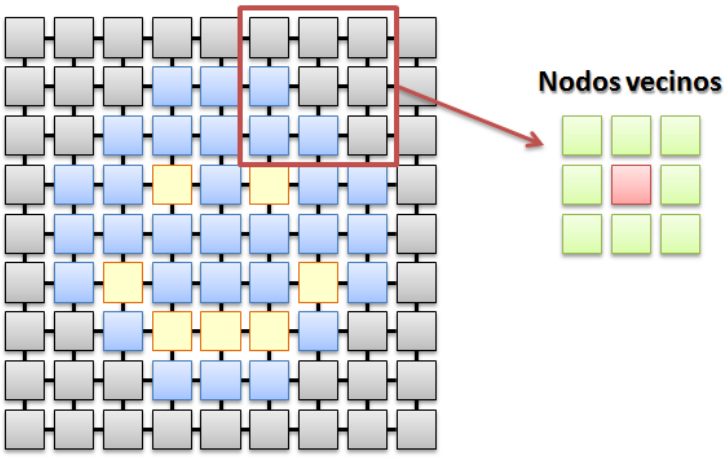


Figura 4.15: Topología de nodos. Ejemplo en imagen 2D.

El algoritmo de etiquetado comienza asignando a cada uno de los puntos que componen la imagen filtrada un índice correlativo, tal y como se muestra en el ejemplo de la figura 4.16, de tal manera que cada punto tenga un índice distinto.

Figura 4.16: Etiquetado. Inicialización.

A continuación, se construye una Lista de Equivalencia (E) en la que para cada píxel o vóxel se almacena el índice más bajo que tiene a su alrededor, es decir, cada píxel almacena el índice del vecino que es similar (utilizando el criterio indicado previamente) que tiene el índice más bajo, incluyendose a sí mismo. Dicho proceso se puede ver en el ejemplo de la figura 4.17.

Figura 4.17: Etiquetado. Construcción de la Lista de Equivalencia.

Una vez construida la Lista de Equivalencia (E) se realiza el proceso de reducción, en el que cada píxel (i) verifica su entrada en la Lista de Equivalencia (E), dando las siguientes dos posibles situaciones:

1. Si $E(i) = i$ entonces el píxel es el comienzo de una nueva región, por lo que no se hace nada.

2. Si $E(i) \neq i$ entonces el píxel intenta determinar el píxel inicial de su región de manera iterativa. Para ello, se traslada al índice de la Lista de Equivalencia que indica su entrada $E(i)$. Si en dicha entrada, el índice en la lista corresponde con el valor que indica ($E(E_{current}) = E_{current}$) se asigna dicho valor a $E(i)$ y se concluye el proceso. Si no, se salta al índice que indica dicha entrada y se vuelve a realizar la verificación.

Este proceso se puede ver en el ejemplo de la figura 4.18. Dicho proceso concluye cuando se haya procesado todas las entradas de la Lista de Equivalencia.

Entonces, a cada píxel de la imagen se le asigna su valor resultante en la Tabla de Equivalencia, tal y como se muestra en el ejemplo de la figura 4.19. El proceso se repite mientras algún píxel cambie su valor en alguna iteración.

Para llevar a cabo la implementación de este método para el problema a resolver, se considera la solución

Figura 4.18: Etiquetado. Redución Lista de Equivalencia.

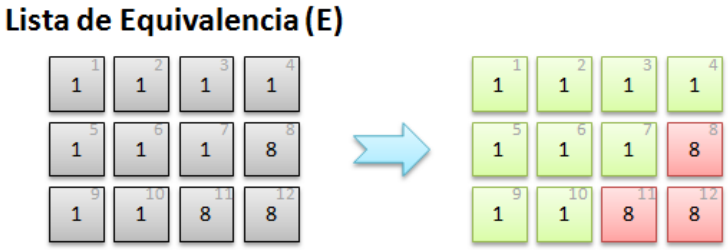


Figura 4.19: Etiquetado. Asignación Lista de Equivalencia sobre puntos imagen.

indicada en [15], estableciendo cuatro núcleos que se encargan de realizar las labores del proceso. En la figura 4.20 se muestra gráficamente el diagrama que representa la manera en que se desarrolla la ejecución de los diferentes núcleos.

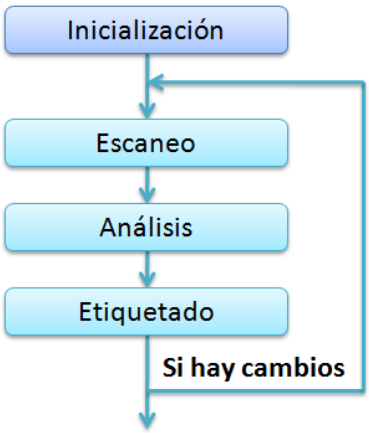


Figura 4.20: Algoritmo de Etiquetado. Proceso de ejecución de los núcleos.

El núcleo denominado *Inicialización* está encargado simplemente de realizar inicialización de la lista de

etiquetas de la imagen.

A continuación, comienza un proceso iterativo que se realiza mientras haya cambios en el etiquetado de la imagen. El primer núcleo que se ejecuta en él, denominado *Escaneo*, está encargado de la creación de la *Lista de Equivalencia* en cada iteración, es decir, para cada punto mira a sus puntos vecinos para obtener el mínimo índice a asignar en la lista. Para realizar la comparación de cada punto con sus vecinos, al establecer una métrica Euclídea, el color de dicho punto viene definido por el mapa de color $L * u * v$, si la imagen es en color, y por la componente L del mismo, si la imagen es en escala de grises.

Una vez concluido la ejecución de este núcleo, se ejecuta el núcleo de *Análisis* que se encarga de realizar la reducción de la *Lista de Equivalencia*, tal y como se ha indicado anteriormente.

Finalmente, el núcleo de *Etiquetado* se encarga de asignar la *Tabla de Equivalencia* resultante a los índices de la imagen.

La ejecución de todos los núcleos se realiza asignando a cada *work–item* un punto de la imagen para que realice las operaciones necesarias sobre él.

El resultado de este proceso es una lista con la etiqueta asignada a cada punto de la imagen.

Capítulo 5

Detalles de la Implementación

A lo largo del capítulo anterior se ha presentado *clMeanShift* como propuesta de solución para la adaptación del algoritmo *Mean Shift* por medio de un conjunto de núcleos OpenCL, que realizan ciertas partes del proceso en paralelo. Como se ha visto, dicha propuesta difiere del algoritmo básico debido principalmente a la naturaleza de la información a tratar (imágenes 2D, 3D y 4D), pero también de manera especial a las características propias de la combinación de código secuencial con un conjunto de núcleos OpenCL que realizan partes del proceso en paralelo por medio de una serie de *work–items*.

La Ley de Amdahl es considerada en muchas ocasiones como un método para cuantizar la mejora de un sistema cuando se optimizan partes de éste. En el caso de la paralelización de un algoritmo, la Ley de Amdahl especifica el máximo *speed-up* que puede ser alcanzado por la paralelización de porciones del código secuencial. Dicha ley establece que el *speed–up* máximo (S) de un programa viene dado por la expresión 5.1, donde P es la proporción del tiempo de ejecución total del código secuencial que puede ser paralelizado, y por otro lado, N es el número de procesadores, hilos de ejecución o *work–items*, en los que el código paralelo es ejecutado.

$$S = \frac{1}{(1-P) + \frac{P}{N}} \tag{5.1}$$

De manera teórica, la idea básica extraída de la expresión 5.1 es que cuanto mayor es el número de hilos de ejecución en paralelo, mayor es el *speed–up* conseguido. Este resultado se obtiene en un escenario ideal, pero la arquitectura donde se ejecuta el algoritmo presenta en muchas ocasiones una serie de factores que limitan esta ganancia. Por ello, es fundamental conseguir identificar estos factores a la hora de diseñar el algoritmo paralelo, y en consecuencia, realizar una serie de optimizaciones sobre el código desarrollado, con el objeto de minimizar el impacto que provocan sobre la ejecución del algoritmo. En el caso concreto de la solución planteada, dichos factores se producen principalmente en el conjunto de núcleos OpenCL ejecutados.

En la mayoría de las ocasiones, el conjunto de limitaciones que presenta la arquitectura es establecida por el propio fabricante del dispositivo compatible con OpenCL, que dedica capítulos de sus manuales a

describir el conjunto de limitaciones y consideraciones a tener en cuenta al ejecutar un núcleo OpenCL sobre su dispositivo([29], [28], [16]). Su objetivo es guiar al desarrollador en la implementación de las soluciones diseñas, de tal manera que se consideren las particularidades propias de la arquitectura utilizada en cada caso, y así obtener un mayor rendimiento del algoritmo desarrollado. Tres son las estrategias básicas a tener en cuenta para optimizar el algoritmo desarrollado en OpenCL:

- Maximimizar la ocupación del procesador, maximizando la ejecución en paralelo.
- Optimizar la gestión de la memoria para lograr maximizar el ancho de banda de transferencia de información.
- Optimizar el uso de operaciones en el núcleo para lograr maximizar el rendimiento de las instrucciones.

En este caso concreto se han utilizado dos dispositivos compatibles con OpenCL con el objetivo de verificar la implementación realizada del algoritmo *Mean Shift* , así como para realizar la evaluación del mismo. Por un lado, se ha utilizado la propia CPU del PC gracias a los controladores desarrollados por AMD para arquitecturas X86, que permiten ejecutar núcleos OpenCL aprovechando los diferentes núcleos del procesador. Además, habilitan ciertas extensiones propias que facilitan la labor de depuración de la ejecución de los núcleos OpenCL, aspecto que ha sido fundamental durante el desarrollo de éstos. Por otro lado, se ha utilizado una tarjeta gráfica nVIDIA basada en una arquitectura *many–cores* pura que permite establecer una mayor paralelización del proceso al disponer de mayor capacidad para ejecutar *work–items* en paralelo.

Con todo lo indicado, en el presente capítulo se realiza una breve descripción del conjunto de optimizaciones y consideraciones que se han tenido en cuenta a la hora de realizar la solución planteada, así como destacar aquellos detalles propios de la implementación.

5.1. Ocupación del Procesador

La ocupación del procesador tiene relación directa con la expresión de la Ley de Ahmdal vista en 5.1. En el caso de un núcleo OpenCL, el valor de N de la expresión se refiere al número de *work–items* que se pueden ejecutar en paralelo cuando se ejecuta el núcleo.

En un principio, dicho valor viene limitado por la capacidad de la propia arquitectura en la que se ejecuta el código. Por ejemplo, en el caso de una tarjeta gráfica nVIDIA, ésta puede ejecutar hasta 1024 *work–items* en paralelo, aunque este valor puede ser inferior dependiendo de las características propias del núcleo a ejecutar.

Uno de los factores básicos que reduce este número es la utilización de registros hardware por parte de los *work–items* en ejecución. Estos registros son utilizados durante la ejecución del código asociado al *work–item* para almacenar resultados parciales, variables,... Los *Dispositivos de Cálculo* poseen un número limitado de registros que son repartidos por todos los *work–items* que se encuentran en ejecución dentro de él. Por ello, si

los *work–items* necesitan gran cantidad de registros para llevar a cabo su ejecución, al tratarse de un recurso compartido, puede limitar el número de *work–items* que se pueden ejecutar en paralelo.

Por otro lado, como se ha visto en el Capítulo 3 (3.2.1), los *work–items* pueden ser agrupados en *work–groups*. Dicha estrategia es utilizada si los *work–items* de un mismo *work–group* necesitan compartir información entre ellos para llevar a cabo sus cálculos, por medio de la utilización de *Memoria Local*. Del mismo modo, se puede utilizar esta estrategia para agrupar *work–items* totalmente independientes entre sí, cuya ejecución se realiza en paralelo dentro del procesador. En este caso, se busca aumentar la ocupación del procesador, estableciendo el tamaño del *work–group*, de tal manera que se maximize el número de *work–items* en ejecución, teniendo en cuenta el resto de aspectos propios de la arquitectura. Esta última estrategia ha sido intensivamente utilizada en los diferentes núcleos encargados de realizar operaciones y transformaciones sobre los puntos de la imagen, tal y como se ha indicado a lo largo del Capítulo 4 (4).

Finalmente, cada *work–group* puede tener asociado un número máximo de *work–items* que viene definido en la propia arquitectura (por ej. 512 para el caso de dispositivo nVIDIA). Adicionalmente, en el caso particular de los procesadores gráficos, se aconseja que dicho número sea múltiplo de un número N específico (por ej. 32 para nVIDIA), con el objetivo de favorecer el *scheduler* de ejecución, y sobre todo las operaciones de acceso a memoria.

5.2. Gestión de Memoria

La gestión de la memoria es uno de los aspectos fundamentales a la hora de diseñar un algoritmo basado en OpenCL. Una mala gestión de la misma puede suponer una pérdida considerable del rendimiento del algoritmo, y por lo tanto, una reducción del *Speed–Up* conseguido respecto a otras implementaciones posibles. Como se vió en el Capítulo 3 (3.2.1), el modelo de memoria establecido por OpenCL identifica una serie de tipos de memoria asociadas tanto al *Dispositivo Anfitrión* como al *Dispositivo OpenCL*.

OpenCL establece una separación de la memoria del *Dispositivo Anfitrión* y la memoria propia del *Dispositivo OpenCL*. Esto supone que la transferencia de información entre ambos dispositivos debe realizarse de manera explícita desde el código ejecutado en el *Anfitrión*.

Como se muestra en la figura 5.1, dicha transferencia se realiza entre la memoria principal del *Dispositivo Anfitrión* (Memoria RAM) y la *Memoria Global* o *Memoria Constante* del Dispositivo OpenCL. El ancho de banda del interfaz a través del que se realiza la transferencia entre ambos dispositivos (por ej. Bus PCI–Express) está limitado, presentando una capacidad de entre $2\frac{GB}{s}$ y $6\frac{GB}{s}$. Por ello, es necesario limitar el número de transferencias entre ambos Dispositivos.

En el caso concreto de la solución planteada, su diseño ha permitido minimizar dichas transferencias. En concreto, se realizan dos transferencias fundamentales, una para transferir los puntos a procesar al Dispositivo OpenCL, y por otro lado, para recuperar los resultados del proceso. El resto del algoritmo trabaja con la información almacenada en la memoria del Dispositivo OpenCL y procesada por los diferentes núcleos, a

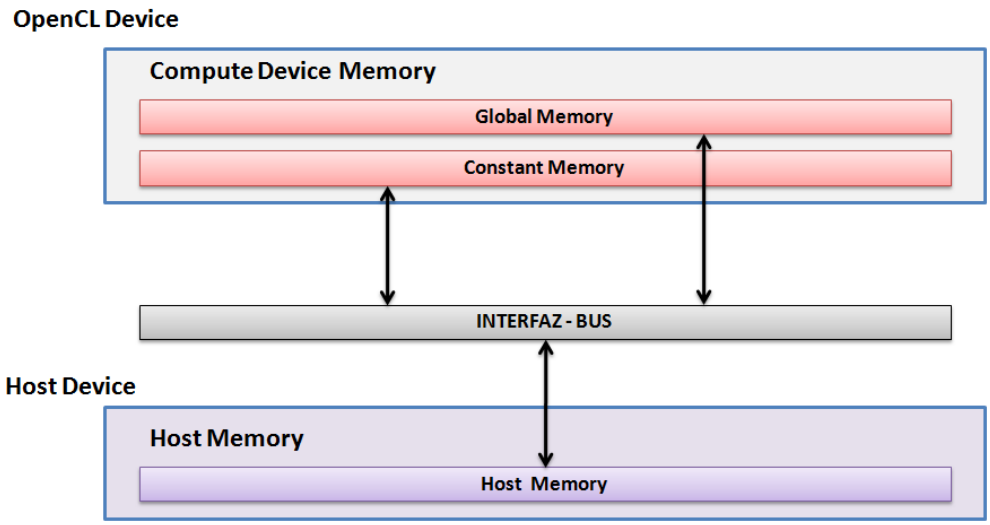


Figura 5.1: Transferencia Memoria *Anfitrión – Dispositivo OpenCL*

excepción de la condición de salida del bucle *Mean Shift* , cuyo valor es establecido por el núcleo asociado y leído por el *Programa Anfitrión* para verificar la condición del bucle.

Como se indicó en el capítulo 3, la *Memoria Global* es la que presenta la mayor capacidad para almacenar información dentro del Dispositivo. Por contra, es una memoria no cacheada que, aunque es más rápida que la memoria propia del *Dispositivo Anfitrión*, presenta una alta latencia por lo que el coste de acceso es elevado. Por ello, su hardware está diseñado para proporcionar información en ráfagas con el objetivo de enmascarar dicha latencia.

Por un lado, el Dispositivo es capaz de leer información 4, 8 ó 16 Bytes con una única instrucción, por lo que es aconsejable que las estructuras de datos utilizadas estén alineados a uno de estos tamaños para reducir el número de instrucciones necesarias para realizar la transferencia. En el caso de que la estructura de datos tenga un tamaño superior a 16 Bytes son necesarias realizar varias instrucciones de acceso, cuyo número depende de su alineación posible respecto a los tres tamaños indicados.

Por otro lado, el aspecto fundamental a la hora de acceder a la información contenida en la *Memoria Global* es el patrón de acceso utilizado. El ancho de banda de transferencia de la *Memoria Global* es utilizado de manera más eficiente cuando el conjunto de accesos simultáneos realizados por los *work–items* (operaciones de lectura o escritura de una posición de memoria) se pueden *fusionar* (*coalesced*) en una única transacción de memoria, consiguiendo enmascarar la latencia propia de la transferencia. En concreto, para el caso de los Procesadores Gráficos nVIDIA (ver [28] para detalles) dicho acceso está optimizado a nivel de hardware para que 16 *work–items* accedan a la vez a direcciones consecutivas de memoria de 32, 64 ó 128 Bytes. Dependiendo del patrón utilizado dicha operación puede suponer una única transacción en el mejor de los casos ó 16 en el peor. En la figura 5.2 se muestran dos ejemplos en los que se muestra un patrón cuyo coste es una única transacción y otra que provoca 16 transacciones.

(a) Patrón de acceso a memoria *coalesced*. Una sola transacción de memoria.

(b) Patrón de acceso a memoria no *coalesced*. Necesarias 16 transacciones de memoria.

Figura 5.2: Ejemplos de patrones de acceso a información de 4 Bytes.

En el caso de la solución planteada, varios de los procesos realizados explotan esta características, ya que muchos de los cálculos se realizan sobre listas de puntos en los que cada *work–item* se encarga de procesar un punto de dicha lista. De esta manera, el patrón de acceso es similar al mostrado en el ejemplo, por lo que dichos accesos se puede *fusionar*. Por contra, en el caso del proceso iterativo de cálculo del máximo local para cada punto, que supone el mayor porcentaje de proceso a realizar, dicho patrón de acceso a la *Memoria Global* no es el ideal. Como se explica en el capítulo 4, durante el proceso iterativo cada *work–item* realiza el cálculo sobre un punto en concreto de la lista de puntos a procesar. A partir de dicho punto, es necesario realizar el cálculo de la nueva posición a partir del conjunto de puntos *vecinos* de éste, que forman parte de la celda de la rejilla tridimensional a la que pertenece el punto, así como al conjunto de celdas adyacentes a ésta. Teniendo en cuenta que todos los *work–items* realizan dicho proceso en paralelo, es necesario que éstos realicen accesos a la *Memoria Global* para leer la información de cada punto siguiendo un patrón que se puede considerar *aleatorio*, lo que provoca que no se cumplan las condiciones para fusionar las transacciones

de memoria, por lo que aumenta el número de transacciones necesarias, y por lo tanto la penalización en el proceso.

Por otro lado, se identifica la denominada *Memoria Local* que es una zona de memoria dentro de los Dispositivos OpenCL, que es más rápida que la *Memoria Global*, además de permitir que los diferentes *work–items* de un mismo *work–group* compartan información. Esta memoria puede ser utilizada para *cachear* información de la *Memoria Global* que vaya a ser utilizada por los *work–items* del *work–group* durante su proceso, y de esta manera, disminuir el tiempo requerido para acceder a la información. Este aspecto es de gran interés si la misma información es utilizada por varios *work–items*. Para ello, cada *work–item* lee en paralelo una porción de la información a *cachear* que es almacena en la *Memoria Local*. Tras la operación de transferencia, se establece una barrera para asegurar que todos los *work–items* han completado dicho proceso. De esta manera, cada *work–item* puede trabajar directamente con la información almacenada en la *Memoria Local*, acelerando considerablemente el acceso a la misma.

A pesar de la gran ventaja que presenta esta técnica, hay que tener en cuenta que la *Memoria Local* es de un tamaño muy reducido (normalmente $16KB$) comparado con el tamaño *Memoria Global*. Además está compartida por los diferentes *work–groups* que se ejecutan en paralelo dentro del Dispositivo, aunque no puedan compartir información entre ellos, lo que limita el número de *work–groups* que se puede ejecutar en paralelo.

En el caso particular de la solución propuesta, esta técnica ha sido utilizado en aquellas partes del código que por su naturaleza ha permitido su aplicación. Un claro ejemplo es el conjunto de celdas de la Rejilla Tridimensional a visitar durante cada iteración del bucle *Mean Shift* . Las posiciones relativas de éstas respecto a la celda a la que pertenece el punto que se está procesando es siempre la misma, por lo que son precalculadas y almacenadas en *Memoria Local* para que sean utilizadas por todos los *work–items* para el cálculo de la posición absoluta de la celdas adyacentes.

5.3. Rendimiento de las instrucciones

Como se ha visto en el apartado anterior, la Gestión de la Memoria es el factor fundamental a la hora de desarrollar algoritmos paralelos basados en OpenCL. Aunque este factor tiene un gran peso, existen otras estrategias que permiten reducir el coste computacional del algoritmo desarrollado. Dentro de este conjunto de optimizaciones se identifican aquellas que afectan directamente a la ejecución del conjunto de instrucciones que forman el núcleo OpenCL. Para el desarrollo de la solución propuesta se han considerado una serie de optimizaciones básicas para reducir el tiempo de ejecución de cada uno de los núcleos OpenCL desarrollados.

En primer lugar, tal y como sucede en las actuales arquitecturas cualquier instrucción de control de flujo (bucles, condiciones) puede impactar significativamente en el rendimiento de la ejecución debido principalmente a que los *work–items* que se están ejecutan en paralelo diverjan, es decir, que tomen diferentes caminos de ejecución. Dicho aspecto se ve claramente en una estructura básica *IF-THEN-ELSE* en la que los

work–items toman los dos caminos de ejecución posibles. En dicha situación, al ejecutarse el código siguiendo un modelo SIMT, esto provoca que se serializen ambos caminos de ejecución, lo que aumenta el número de instrucciones a ejecutar, con la consiguiente penalización de tiempo. Lógicamente, dicho aspecto no sucede si todos los *work–items* siguen el mismo camino. Del mismo modo sucede en aquellos casos en los que los *work–items* que están ejecutando un bucle salen de éste en diferentes iteraciones.

Por otro lado, el salto realizado en cada iteración de un bucle supone en las arquitecturas actuales una fuerte penalización en ciclos de procesador. Por ello, una estrategia clásica que se suele realizar consiste en el *desenrrollado* del bucle. Dicha estrategia se ha utilizado intensivamente en los cálculos realizados sobre los vectores d–dimensionales que representa cada punto de la imagen en el Espacion de Características. Básicamente, sobre dicha información siempre se realiza un proceso iterativo para realizar la operación correspondiente sobre cada una de las componentes de dicho vector. Tal y como se muestra en la figura 5.3 a), la manera básica de realizar dicho proceso es por medio de un bucle que itera sobre sus componentes realizando las operaciones correspondientes. Al conocer de manera determinista el número de iteraciones necesarias, se puede realizar un desenrrollado completo del bucle, obteniendo el código mostrado en la figura 5.3 b). De esta manera, al eliminar el salto del bucle se consigue un aumento del rendimiento del proceso. En el caso de las componentes del vector que representa cada punto de la imagen hay que tener en cuenta que su dimensión varía dependiendo del tipo de imagen a procesar. Por ello, el núcleo OpenCL es compilado especialmente para cada caso teniendo en cuenta la dimensión de éstos.

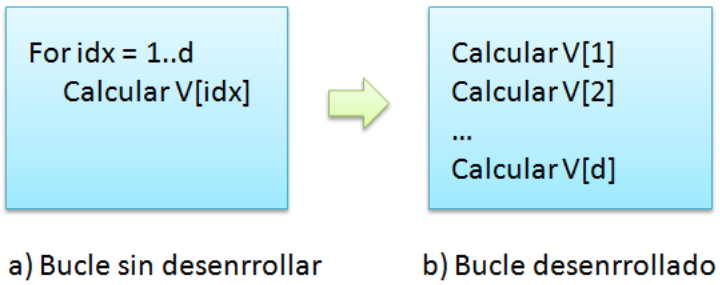


Figura 5.3: Desenrrollado de bucles.

Por otro lado, a la hora de desarrollar el conjunto de núcleos OpenCL se han tenido en cuenta otra serie de optimizaciones menores cuyo objetivo ha sido reducir el coste computacional de partes concretas del código. Por un lado, se han precalculado aquellos valores utilizados por todos los *work–items* del núcleo, por medio de la definición de constantes (por ej. conversión $RGB \rightarrow L * u * v$ y viceversa) . Por otro lado, se han sustituido operaciones aritméticas de alto coste, que se realizan de igual manera sobre varios datos consecutivos, por operaciones equivalentes de menor coste (por ej. Sustituir varias divisiones con el mismo denominador por varias multiplicaciones por el inverso del denominador). También se han utilizado tipos de datos vectoriales estándar OpenCL en aquellos casos factibles, lo que permite realizar al compilador optimizaciones automáticas del código. Finalmente, se ha utilizado intensivamente la compilación condicional

para adaptar el código del núcleo dependiendo del tipo de imagen a procesar.

Capítulo 6

Resultados Experimentales

A lo largo de este capítulo se muestran los resultados del proceso de evaluación de *clMeanShift*. Esta evaluación se realiza mediante una serie de simulaciones en las que se aplica el proceso de segmentación sobre un conjunto de imágenes de prueba. El objetivo principal, es establecer la ganancia de proceso que se alcanza con *clMeanShift* respecto a otras implementaciones disponibles, así como verificar la calidad de los resultados obtenidos

Dicha evaluación se divide en dos bloques fundamentales. Por un lado, se estudia el comportamiento del proceso de filtrado Mean Shift, y por otro lado, se evalúa del proceso de etiquetado a partir de la imagen previamente filtrada.

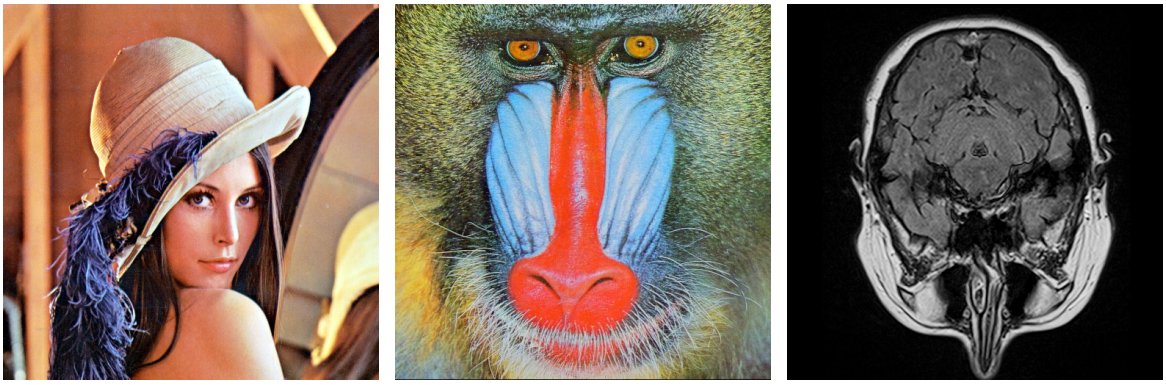
6.1. Entorno de simulación

La evaluación de *clMeanShift* se realiza por medio de un conjunto de imágenes 2D y 3D. El conjunto de imágenes 2D seleccionadas son mostradas en la figura 6.1, y se identifican como *Lenna* 6.1a, *Mandril* 6.1b y *MRI 2D* 6.1c.

Las imágenes *Lenna* y *Mandril* se han utilizado en muchas de las investigaciones de Tratamiento de Imagen y Visión por Computador para realizar la evaluación de diferentes algoritmos y métodos. La principal diferencia entre ambas, y que afectan al proceso a evaluar, es la uniformidad de la imagen. La imagen *Lenna* es una imagen que presenta grandes zonas uniformes de color, a diferencia del caso de la imagen *Mandril*, en la que hay una gran variación de detalles que provocan variaciones de color, principalmente en las zonas del pelaje del primate. Para ambos casos, la representación del color de cada píxel viene definido por un mapa de color RGB (3 dimensiones).

Por otro lado, la imagen denominada *MRI 2D* es un claro ejemplo de posible aplicación de *clMeanShift* para el tratamiento de imágenes médicas 2D. A diferencia de las otras dos imágenes mostradas, en este caso el color de cada píxel viene representado por un valor en la escala de grises (una dimensión).

Por otro lado, la evaluación del proceso para imágenes 3D se realiza por medio de un ejemplo de MRI 3D.



(a) Lenna. Tamaño 512 x 512 (b) Mandril. Tamaño 512 x 512 (c) MRI 2D. Tamaño 424 x 512

Figura 6.1: Imágenes de prueba 2D.

En la figura 6.13 se muestra una serie de cortes representativos de los 160 cortes que componen la imagen completa. Como sucede en el caso de la MRI 2D, su valor de color es representado por medio de un mapa en escala de grises.

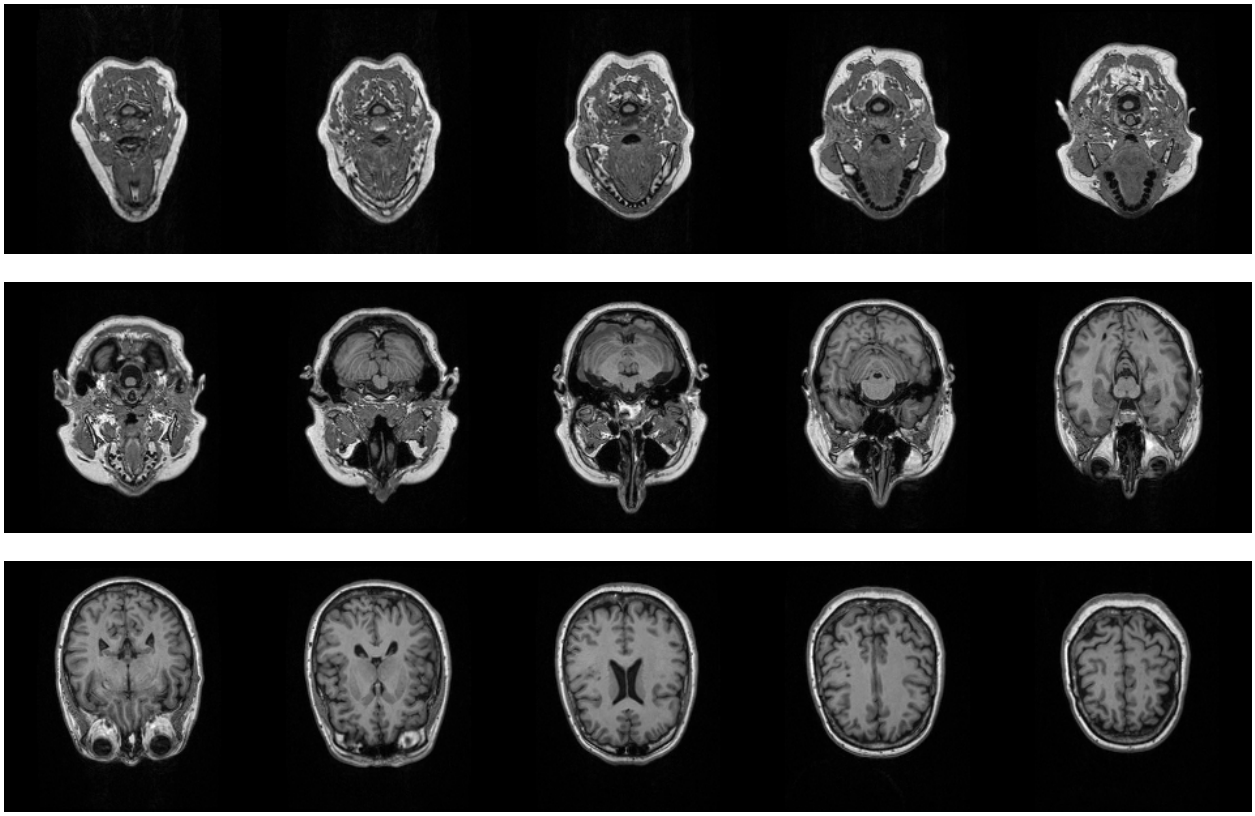


Figura 6.2: Imagen de prueba 3D. MRI 3D. Tamaño 192 x 192 x 160

El conjunto de simulaciones se realizan por medio de un PC portátil que incorpora un procesador de gama media Intel i5-430M compuesto por 2 núcleos a una velocidad de 2.26 GHz y 4 GB de memoria RAM. Adicionalmente, incorpora una tarjeta gráfica de gama baja nVIDIA GeForce GT 325M con 1 GB de memoria dedicada. Esta máquina dispone de dos *Dispositivos OpenCL* compatibles con la revisión 1.1 de OpenCL. Por un lado, la GPU de la tarjeta gráfica nVIDIA por medio del controlador suministrado por el fabricante, y por otro lado, la propia CPU del computador gracias al controlador desarrollado por AMD, que permite la ejecución de núcleos OpenCL en arquitecturas X86. Las prestaciones de *clMeanShift* se realizan a partir de ambos *Dispositivos OpenCL*.

En la tabla 6.1 se muestra las capacidades básicas de ambas arquitecturas para la ejecución de núcleos OpenCL. Como se observa la GPU presenta un mayor número de Unidades de Computación respecto a la CPU. Por contra, la CPU presenta una mayor capacidad de memoria y frecuencia de reloj. Este último aspecto no es significativo debido a la naturaleza de ambos procesadores. Por un lado, la CPU es un procesador de propósito general cuya arquitectura está pensada para ejecutar varios procesos diferentes de manera concurrente. Por otro lado, la arquitectura de la GPU está diseñada para realizar operaciones de tipo *SIMT* por lo que tiene una mayor capacidad de paralelismo sin necesidad de procesar a una frecuencia de reloj comparable a la de una CPU.

Parámetro	**i5-430M (CPU)**	**nVIDIA GT325M (GPU)**
Unidades de Computación	4	6
Frecuencia del Reloj (MHz)	2261	990
Dimensiones máximas *Work–Item*	1024 x 1024 x 1024	512 x 512 x 64
Tamaño máximo *Work–Group*	1024	512
Tamaño Memoria Global (MB)	2048	978
Tamaño Memoria Constante (KB)	64	64
Tamaño Memoria Local (KB)	32	16

Tabla 6.1: Entorno de pruebas. Capacidades OpenCL.

Finalmente, en cada simulación, la medición del tiempo de proceso se realiza por medio de un paquete de medición propio desarrollado para registrar el tiempo consumido por las diferentes fases del proceso. Dicho paquete combina dos métodos de medición diferentes. Para el caso del código ejecutado en el *anfitrión*, la medición se realiza registrando el número de ciclos de reloj transcurridos entre dos puntos del ejecutable. Por otro lado, el tiempo de ejecución de cada núcleo OpenCL y las operaciones de transferencia de memoria entre la memoria del *anfitrión* y la del *Dispositivo OpenCL*, se realiza por medio del registro de *eventos* que proporciona la API de OpenCL (ver capítulo 5.9 de [22] para detalles). Dichos eventos registran el momento temporal de comienzo y finalización de la ejecución de la operación a la que se asocia el evento, por lo que se obtiene su coste.

6.2. Evaluación del Proceso de Filtrado

6.2.1. Proceso de filtrado para imágenes 2D

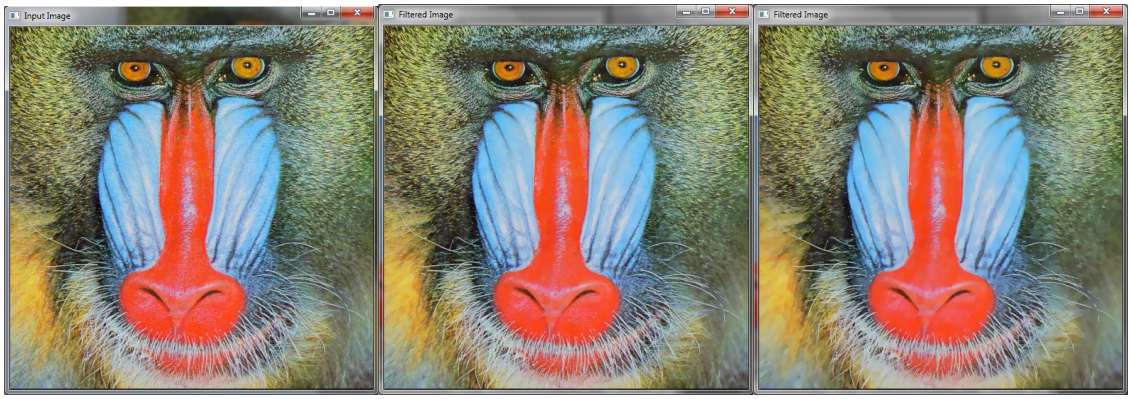
La evaluación del proceso de filtrado, para el caso de imágenes 2D, se realiza aplicando distintos parámetros de ventana para los valores de h_s y h_r. Respectivamente, estos valores establecen la resolución del proceso en el Dominio Espacial y el Dominio del Color, y establecen los parámetros básicos necesarios para realizar el proceso de filtrado.

Para el conjunto de evaluaciones realizadas, se llevan a cabo un total de ocho simulaciones por imagen de prueba, en la que se establecen los valores para los parámetros indicados en la tabla 6.2. En estas simulaciones se establece $\epsilon = 0{,}01$, como umbral para establecer el criterio de finalización del proceso de ascenso de gradiente para un punto dado, y por otro lado, se limita el número máximo de iteraciones del bucle *Mean Shift* a 100.

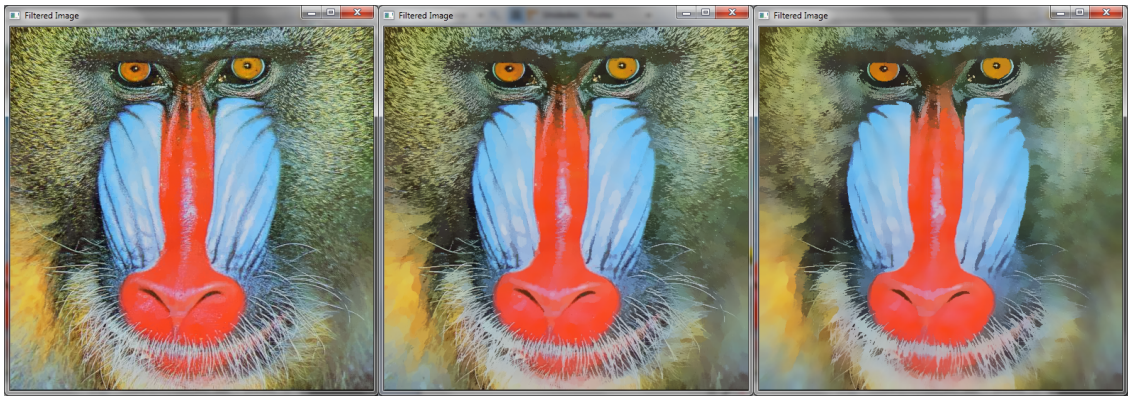
Simulación	h_s	h_r	**Simulación**	h_s	h_r
Simulación 1	4	4	Simulación 5	8	16
Simulación 2	8	4	Simulación 6	16	8
Simulación 3	4	8	Simulación 7	16	16
Simulación 4	8	8	Simulación 8	32	32

Tabla 6.2: Filtrado 2D. Parámetros de simulación.

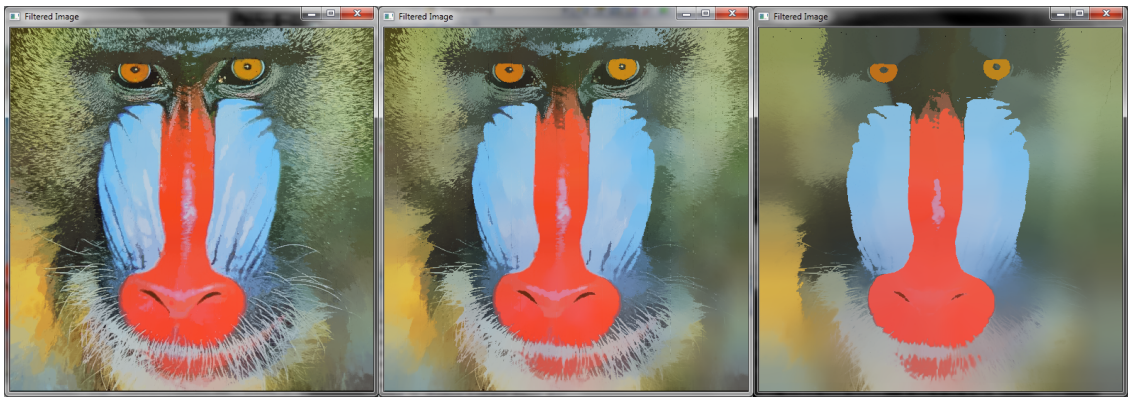
En la figura 6.3, se muestra los resultados de las diferentes simulaciones realizadas sobre la imagen *Mandril*.



(a) Original (b) $h = (4, 4)$ (c) $h = (4, 8)$



(d) $h = (8, 4)$ (e) $h = (8, 8)$ (f) $h = (8, 16)$



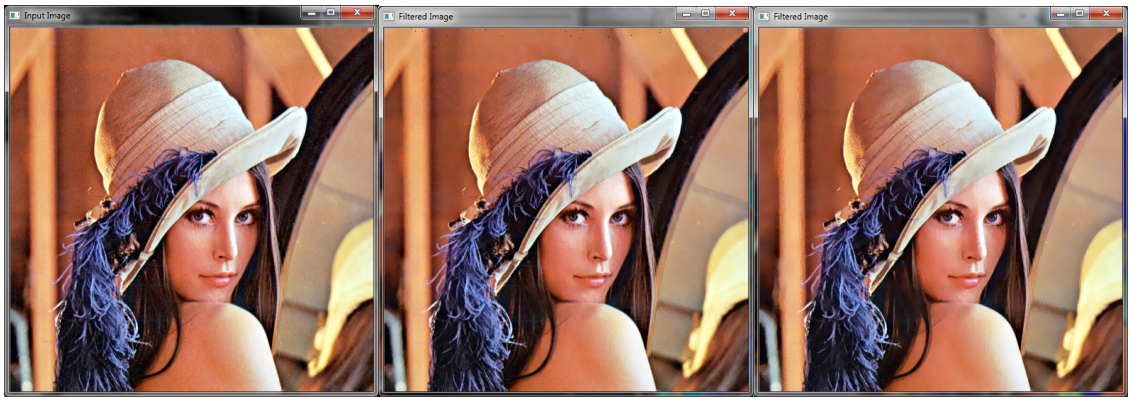
(g) $h = (16, 8)$ (h) $h = (16, 16)$ (i) $h = (32, 32)$

Figura 6.3: Aplicación filtrado Mean Shift sobre imagen *Mandril*. $h = (h_s, h_r)$.

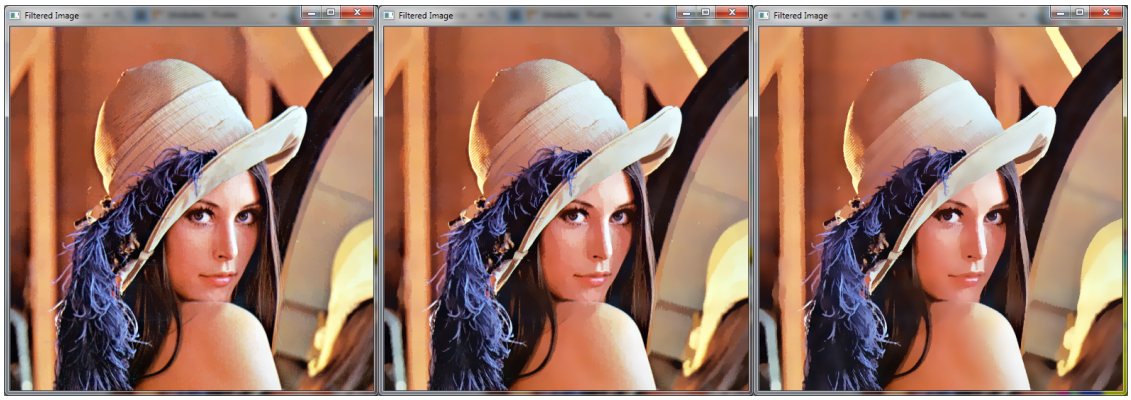
Como se puede observar, la aplicación del filtro *Mean Shift* sobre la imagen provoca la uniformidad de las distintas partes que componen la imagen. En concreto, la textura que compone el pelaje del animal queda elimina, mientras que los ojos y la zona de la barba permanecen definidas. Como se observa, la variación del parámetro h_s no presenta los mismos efectos que la variación del parámetro h_r. El aumento del valor de h_s provoca que sólo aquellas características que se repiten espacialmente a lo largo de la imagen permanecen,

como es el caso de los ojos. Por el contrario, el aumento del valor de h_r provoca que sólo las características con alto contraste de color permanezcan, como sucede con la zona de la nariz. Esto provoca que al aumentar el tamaño de ambos valores, la imagen cada vez sea más uniforme permaneciendo sólo aquellos detalles que predominan en la misma.

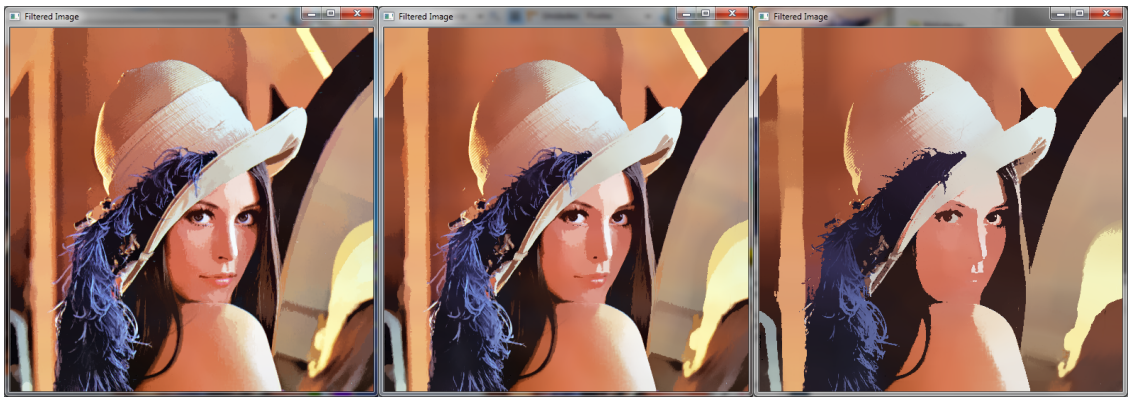
Repitiendo el mismo conjunto de simulaciones sobre la imagen *Lenna*, se obtienen los resultados mostrados en 6.4.



(a) Original (b) $h = (4, 4)$ (c) $h = (4, 8)$



(d) $h = (8, 4)$ (e) $h = (8, 8)$ (f) $h = (8, 16)$



(g) $h = (16, 8)$ (h) $h = (16, 16)$ (i) $h = (32, 32)$

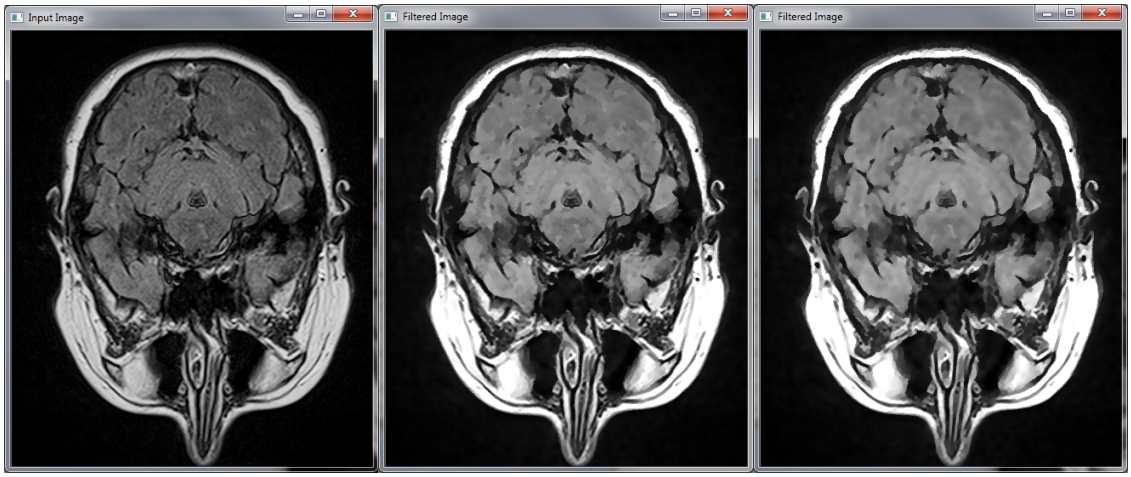
Figura 6.4: Aplicación filtrado Mean Shift sobre imagen *Lenna*. $h = (h_s, h_r)$.

En este caso, el conjunto de zonas más o menos uniformes de color es mayor, si no tenemos en cuenta la zona formada por las plumas que componen el sombrero de la mujer. En este caso, se observa claramente como al ir aumentando los valores de ambas ventanas, cada vez las diferentes partes que componen la imagen son uniformes. Más concretamente, para el caso del espejo se observa claramente como poco a poco se van definiendo claramente las distintas partes que forman dicha zona de la imagen. A su vez, el conjunto de

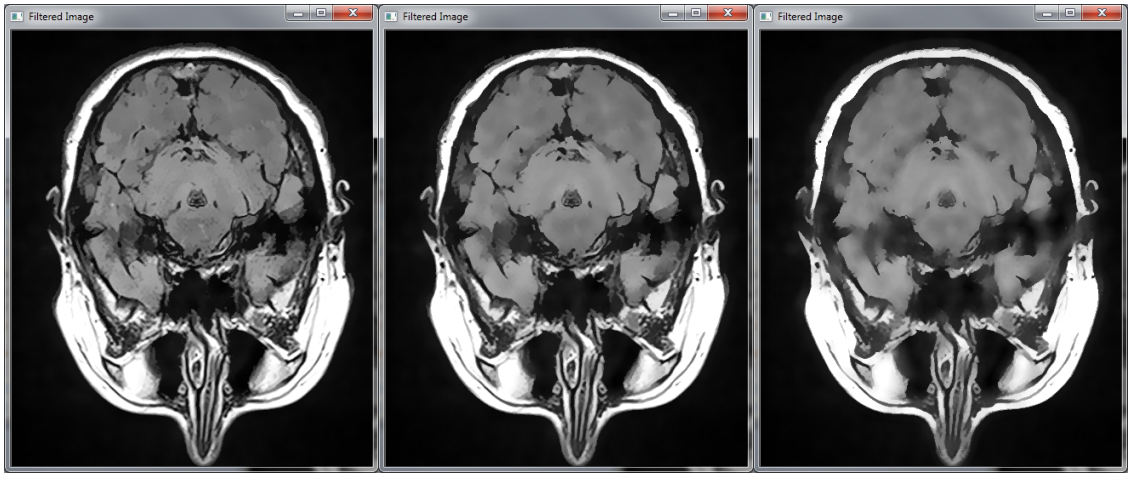
plumas que componen el sombrero poco a poco se vuelve más uniforme, perdiendo los detalles innecesarios para el posterior proceso de etiquetado.

Finalmente, se aplica el proceso de filtrado sobre la imagen denominada *MRI 2D* que presenta la particularidad de representar su color por medio de un mapa de escala de grises. De esta manera, dicho valor en el Dominio del Color viene definido por medio de un único valor unidimensional. El resultado de dicho proceso para cada uno de los parámetros de ventana se muestra en 6.5.

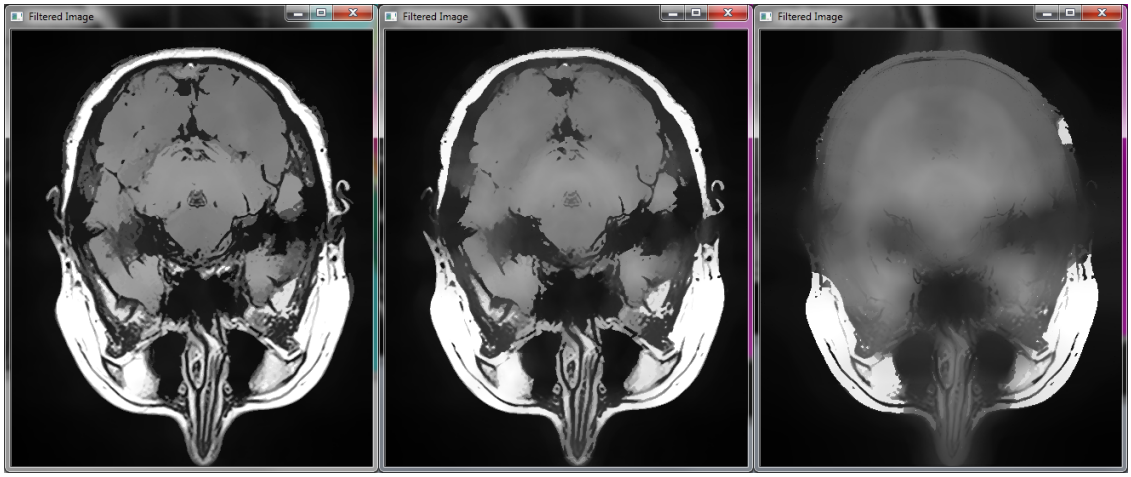
Para este caso, se observa claramente como al aumentar el tamaño de las ventanas, quedan mejor definidos las diferentes zonas que representan partes identificables de la imagen. Se observa claramente, como la zona correspondiente al cerebro se va unificando, definiéndose claramente la zona que agrupa. Del mismo modo sucede con los diferentes huesos que componen el cráneo. Por otro lado, para el caso de $h = (32, 32)$ se muestra el resultado de establecer un valor de parámetros excesivo, que provoca la pérdida de la definición de las distintas partes al homogeneizar excesivamente zonas de la imagen similares. Hay que tener en cuenta que al trabajar en escala de grises el rango de valores para representar cada valor de color es menor que al trabajar con un mapa de color, por lo que es más sensible al valor del parámetro h_r.



(a) Original (b) $h = (4, 4)$ (c) $h = (4, 8)$



(d) $h = (8, 4)$ (e) $h = (8, 8)$ (f) $h = (8, 16)$



(g) $h = (16, 8)$ (h) $h = (16, 16)$ (i) $h = (32, 32)$

Figura 6.5: Aplicación filtrado Mean Shift sobre imagen *MRI 2D*. $h = (h_s, h_r)$.

Como se puede concluir del conjunto de simulaciones realizadas, la elección de los valores de h_s y h_r influyen notablemente en el resultado obtenido. Los valores a establecer dependen básicamente de las características propias de la imagen a filtrar, al no existir un criterio claro para su elección. Como se ha mencionado en el capítulo 3.1, existen distintas propuestas, cuyo objetivo es establecer criterios para la selección de dicho valores, así como otras que proponen la utilización de ventanas dinámicas que se vayan adaptando durante el proceso de filtrado. Hay que tener en cuenta que el resultado del proceso de filtración es fundamental para la correcta realización del proceso de etiquetado, por lo que es fundamental para facilitar la identificación de las distintas partes que componen la imagen segmentada, y de esta manera agrupar aquellos píxeles que las componen.

6.2.1.1. Coste del Proceso de Filtrado

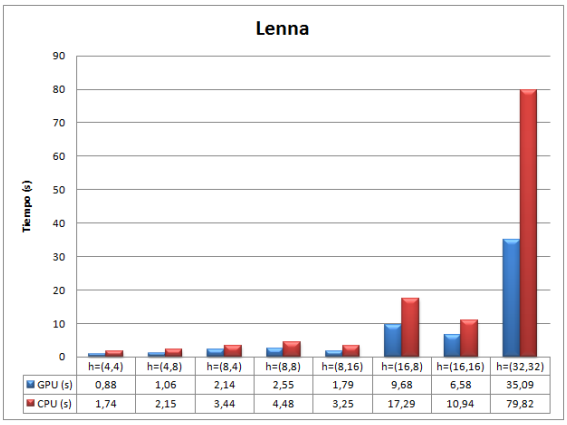
Uno de los objetivos fundamentales buscados en el diseño de *clMeanShift* es reducir el tiempo global necesario para realizar el proceso de filtrado aplicando el algoritmo *Mean Shift* . Como se ha mencionado anteriormente, este algoritmo presenta muy buenos resultados para su aplicación en procesos de Segmentación destinados a diferentes aplicaciones de Visión por Computador, pero tiene el incoveniente de ser muy costoso computacionalmente. Como se ha indicado en el capítulo 2.1, el coste del algoritmo básico *Mean Shift* es $O(kdn^2)$, siendo k el número medio de iteraciones necesarias para alcanzar el máximo local de cada punto, d la dimensión del Espacio de Características, y n el número de puntos de la imagen.

Al plantear un nuevo diseño basado en OpenCL, se intenta explotar la paralelización del algoritmo para reducir el coste computacional general, de tal manera que facilite su implantación en aquellas aplicaciones de Visión por Computador en las que sea fundamental disponer de procesos de segmentación de alta calidad y, a la vez, de bajo coste computacional que permitan realizar los cálculos en poco tiempo, siendo el objetivo final su aplicación en procesos de tiempo real. Con esta premisa, el siguiente conjunto de análisis tienen como objetivo evaluar detalladamente el coste computacional que presenta *clMeanShift*.

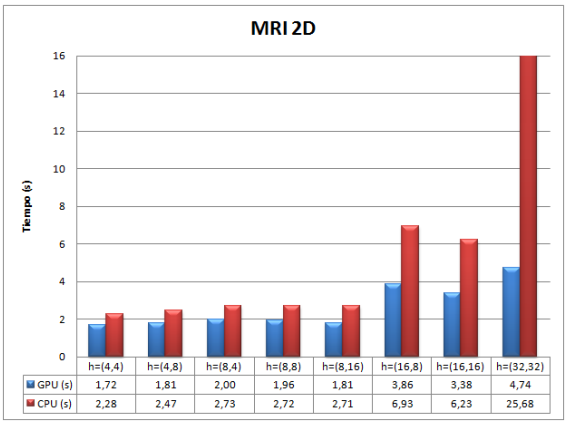
Como se ha indicado al principio de este capítulo, la máquina utilizada para realizar el conjunto de simulaciones dispone de dos *Dispositivos OpenCL*, basados en arquitecturas de distinta filosofía. Por un lado, la CPU de la computadora se basa en un procesador de propósito general *multi–core*, y por otro lado, la GPU está basada en una arquitectura *many–cores*. De esta manera, se establece como primer objetivo, evaluar las prestaciones del algoritmo desarrollado para ambas arquitecturas, a fin de establecer cual es la que mejor se adapta a las características propias del mismo. Hay que destacar que ambas arquitecturas pueden ejecutar el proceso sin ningún tipo de modificación en la implementación realizada, pudiéndose utilizar indistintamente.

La evaluación para los diferentes casos se realiza registrando el tiempo necesario para llevar a cabo el proceso de filtrado sobre las imágenes de test *Lenna* y *MRI 2D*, de tal manera, que se establece la comparativa al tratar una imagen en escala de grises y otra en color. Del mismo modo, para ambos escenario se establecen los valores de ventana h_s y h_r mostrados en la tabla 6.2.

En la figura 6.6 se muestra el resultado de dichas simulaciones. Como se aprecia, la ejecución del conjunto de núcleos OpenCL en la GPU presenta unas mejores prestaciones respecto a la ejecución en la CPU. Este fenómeno aumenta considerablemente al aumentar el tamaño de las ventanas, en el cual es fundamental una mayor paralelización del proceso. La diferencia fundamental entre ambos Dispositivos es que la GPU ejecuta de manera *puramente* paralela el número de *work–items* establecidos en cada núcleo, al disponer cada *Unidad de Computación* del número de *Elementos de Proceso* necesarios para realizar el proceso en paralelo. Por contra, la CPU dispone de un número de *Unidades de Computación* que al lanzar sus *Elementos de Proceso*, éstos deben compartir los recursos de la CPU para su ejecución, similar a como se ejecutan los hilos dentro de un procesador. Por ello, su grado de paralelismo es menor.



(a) Lenna



(b) MRI 2D

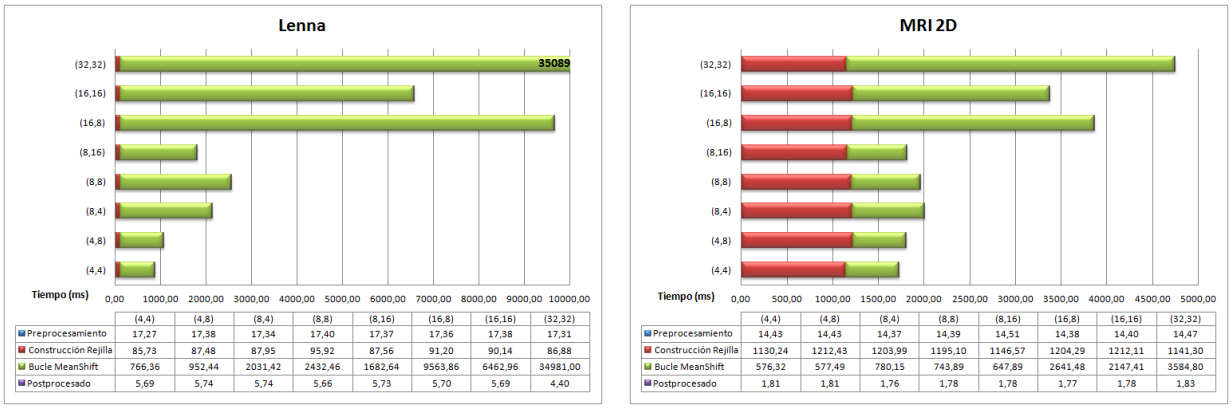
Figura 6.6: Comparativa tiempo de proceso. CPU-GPU.

Este resultado confirma el conjunto de aspectos indicados en el capítulo 3.2, donde se mostraba como la capacidad de cálculo de los Procesadores Gráficos aumenta de manera más considerable respecto a los últimos procesadores *multi–core*, al permitir un aumento de sus unidades de cálculo, generación a generación, de manera sencilla y barata. Adicionalmente, la arquitectura en que se basan los Procesadores Gráficos está especialmente diseñada para realizar cálculos en paralelo. Por lo tanto, para la realización del resto de simulaciones se va a considerar únicamente la ejecución basada en el Procesador Gráfico.

Un aspecto a destacar es la diferencia entre los casos $h = (16, 8)$ y $h = (16, 16)$, en los que al aumentar la ventana h_r el tiempo necesario para completar el proceso se reduce. Esto es debido a que al aumentar dicha ventana, el número de iteraciones necesarias para calcular el máximo local de todos los puntos se reduce ya que convergen antes a sus máximos locales al ampliar al ampliar la resolución del proceso.

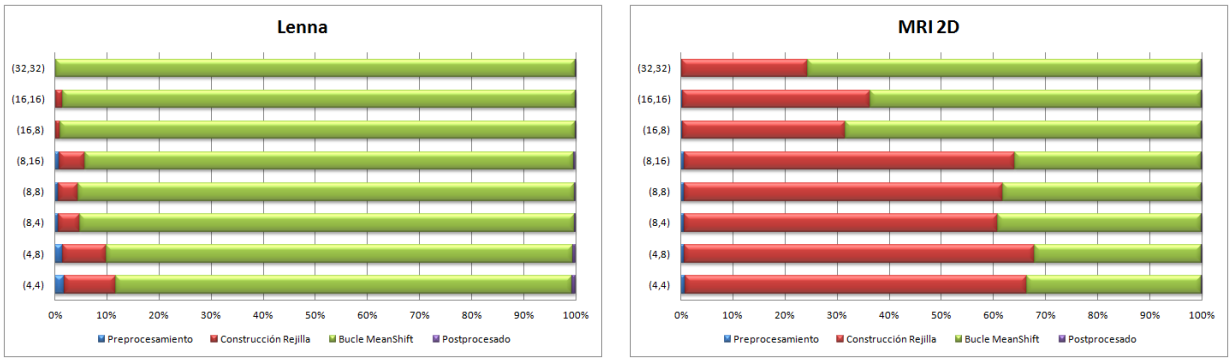
Por otro lado, como se ha mencionado durante la descripción del proceso de filtrado de *clMeanShift*, éste se divide en cuatro fases fundamentales: *Preprocesado*, *Creación de la Cuadrícula Uniforme Tridimensional*, *Bucle Mean Shift* y *Postprocesado*. De estas cuatro fases, la denominada como *Bucle Mean Shift* es la que realiza el proceso de filtrado *Mean Shift* , siendo el resto de fases las encargadas de la adaptación y organización de los puntos que componen la imagen para realizar el proceso de filtrado de la manera más eficiente. Por ello, es importante evaluar el coste computacional de cada una de estas fases para identificar posibles *cuellos de botella* y vías de mejora del algoritmo de filtrado implementado en *clMeanShift*.

Dicha evaluación se realiza a partir del conjunto de simulaciones, realizadas anteriormente, para los diferentes casos ejecutados sobre el *Dispositivo OpenCL* basado en la GPU del Procesador Gráfico de la máquina de pruebas. En la figuras mostradas en 6.7 se muestran el tiempo consumido por cada fase para ambas imágenes de prueba y los diferentes parámetros de simulación. Adicionalmente, en los gráficos de la figura 6.8 se muestra la misma información destacando la proporción de tiempo que consume cada fase.



(a) Lenna (b) MRI 2D

Figura 6.7: Tiempo por fase.



(a) Lenna (b) MRI 2D

Figura 6.8: Porcentaje de tiempo por fase.

Como se aprecia en el caso de *Lenna* el mayor tiempo de proceso se consume durante la fase del *Bucle Mean Shift*, al tratarse de un proceso iterativo en el que se van procesando los puntos hasta obtener sus máximos locales. También hay que tener en cuenta que, durante dicha fase, hay un coste adicional debido al proceso de actualización de la *Cola de Proceso*. Para el caso de la imagen *MRI 2D*, dicho aspecto se cumple para valores altos de las ventanas. Para valores bajos, la mayor parte del proceso es consumida por la Construcción de la Cuadrícula Tridimensional, justificándose este aspecto en los próximos párrafos.

Por un lado, hay que destacar que el coste por iteración del bucle disminuye, iteración a iteración, conforme se van alcanzando los máximos locales de los diferentes puntos, lo que supone una reducción del número de elementos activos en la *Cola de Proceso*, y por lo tanto, menor cantidad de puntos a procesar. Dicho aspecto se muestra para una simulación concreta en la figura 6.9.

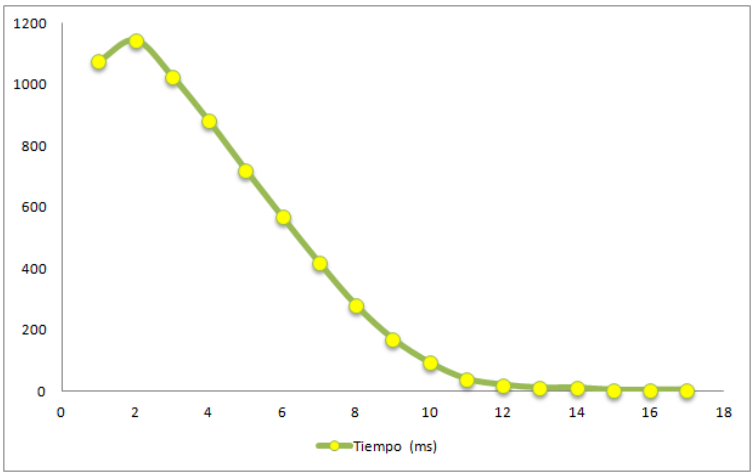


Figura 6.9: Bucle Mean Shift. Tiempo por iteración. Lena $h = (32, 32)$

Como se aprecia, el coste inicial por iteración es alto debido a que es necesario realizar el cálculo sobre todos los puntos de la imagen. Dicho coste va disminuyendo de manera brusca, iteración a iteración, conforme

se van alcanzando los máximos locales de los distintos puntos, hasta alcanzar una zona de coste prácticamente constante hasta concluir el proceso para todos los puntos. El tiempo constante de las últimas iteraciones es producida por aquellos puntos que se encuentran próximos a su máximo local pero cuya velocidad de convergencia es lenta.

Por otro lado, el coste de las fases de *Preprocesado* y *Postprocesado* es prácticamente insignificante debido principalmente a su perfecta paralelización. Hay que tener en cuenta que durante dichas fases básicamente consisten en la transformación de los puntos para adaptarlos para su procesamiento durante las siguientes fases (*Preprocesado*) o para obtener la imagen filtrada (*Postprocesado*), por lo que cada *work–item* lanzado únicamente tiene que realizar el proceso sobre un punto en concreto.

Finalmente, como se ha indicado anteriormente, hay que destacar el tiempo necesario para realizar la fase de *Construcción de la Cuadrícula Tridimensional*. Como se aprecia en las figuras 6.7 y 6.8, a pesar de que dicha fase sólo se ejecuta una vez durante el proceso completo, su coste puede influir negativamente en el resto del proceso, como sucede para el caso de establecer valores bajos de ventana en el caso de la imagen *MRI 2D*.

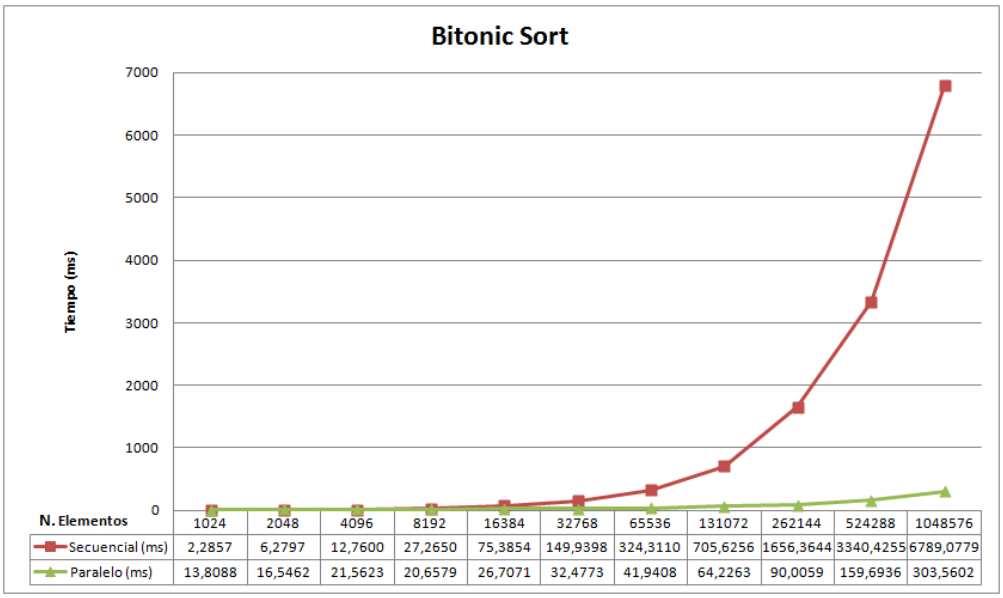
Como se ha indicado en el capítulo 4, durante esta fase se realizan tres labores básicas. Por un lado, los diferentes puntos de la imagen se distribuyen a lo largo de las diferentes celdas que componen la Cuadrícula Tridimensional, dando lugar a la Lista de Pares que relaciona cada punto con su celda correspondiente. Por otro lado, la Lista de Pares se ordena, por medio de una ordenación Bitónica, para agrupar los puntos que pertenecen a la misma celda. Finalmente, se completa el proceso con el cálculo de los límites que establecen los puntos que pertenecen a la misma celda.

Los núcleos OpenCL encargados de realizar el proceso de distribución y cálculo de límites explotan eficientemente el paralelismo que permiten los correspondientes algoritmos desarrollados. Por contra, en el caso de la ordenación de la Lista de Pares, *clMeanShift* selecciona entre dos implementaciones del algoritmo, dependiendo del número de elementos de la lista a ordenar. La diferencia entre ambas implementaciones es que una de ellas está basada en OpenCL, sólo pudiendo trabajar con listas cuyo número de elementos sea potencia de dos, y por otro lado, otra basada en código secuencial que puede trabajar con listas con cualquier número de elementos. Por ello, *clMeanShift* realiza el proceso de ordenación aplicando el algoritmo basado en OpenCL o secuencial dependiendo del número de elementos.

Como se observa en el caso del filtrado de *Lenna* el tiempo necesario para realizar la construcción de la Cuadrícula Tridimensional no afecta considerablemente al tiempo necesario para completar el proceso de filtrado. En este caso, al tratarse de una imagen de 512 x 512 píxeles, se puede aplicar la versión de ordenación basada en OpenCL. Por contra, para el caso de la *MRI 2D*, el tiempo necesario para la construcción de la Cuadrícula influye negativamente, habiendo situaciones en las que dicho coste es superior al necesario para realizar el Bucle Mean Shift. En este caso, al tratarse de una imagen de 424 x 512, es necesario aplicar la versión secuencial del algoritmo de ordenación. Por ello, se realiza la evaluación del impacto que supone dicha ordenación en el proceso general de filtrado, teniendo en cuenta que sólo se realiza una vez durante

este proceso.

La evaluación se realiza midiendo el tiempo que implica la ordenación de diferentes lista, compuestas por elementos aleatorios. Como la versión basada en OpenCL sólo puede trabajar con listas cuyo número de elementos sea potencia de dos, el conjunto de simulaciones realizadas se lleva a cabo sobre listas con dicha característica (1024 a 1048576 elementos). En la figura 6.10 se muestran los resultados obtenidos en las diferentes simulaciones realizadas.



N. Elementos	1024	2048	4096	8192	16384	32768	65536	131072	262144	524288	1048576
Secuencial (ms)	2,2857	6,2797	12,7600	27,2650	75,3854	149,9398	324,3110	705,6256	1656,3644	3340,4255	6789,0779
Paralelo (ms)	13,8088	16,5462	21,5623	20,6579	26,7071	32,4773	41,9408	64,2263	90,0059	159,6936	303,5602

Figura 6.10: Comparativa de tiempos del algoritmo de ordenación

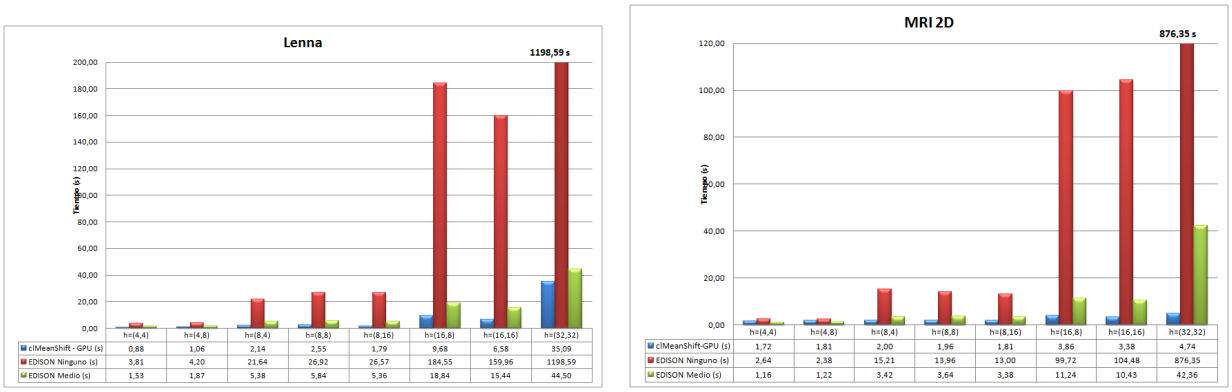
Como se observa, claramente el algoritmo desarrollado en OpenCL ofrece unas mejores prestaciones conforme va aumentando el número de elementos a ordenar, dando constancia del alto grado de paralelización que ofrece el algoritmo de Ordenación Bitónica. Hay que destacar que, para un número bajo de elementos a ordenar, la diferencia entre ambas implementaciones se reduce, e incluso el algoritmo secuencial presenta mejores prestaciones cuando el número de elementos es muy bajo (< 4096). Esto es debido al coste que supone el lanzamiento de la ejecución del núcleo OpenCL en el procesador, el cual afecta notablemente cuando el número de elementos es bajo, quedando enmascarado para un número de elementos grande.

En consecuencia, obtener un algoritmo de ordenación Bitónica basado en OpenCL que, sea capaz de trabajar de manera eficiente con cualquier número de elementos, es fundamental para reducir el impacto del proceso de construcción de la rejilla tridimensional en el proceso total de filtrado. Por lo tanto, es una futura vía de mejora de *clMeanShift*.

Otro de los aspectos para llevar a cabo la evaluación de *clMeanShift* es comparar su coste computacional respecto a otra solución existente. En concreto, se escoge el entorno EDISON (*Edge Detection and Image SegmentatiON*), que permite realizar el proceso de filtrado y segmentación de imágenes 2D. EDISON es un entorno, presentado en [6], que implementa tres niveles de optimización (*Ninguno*, *Medio* y *Alto*) del

algoritmo *Mean Shift* , los cuales son detallados en [18]. Como se ha indicado, *clMeanShift* parte del algoritmo *Mean Shift* básico. Para realizar la evaluación correspondiente, se establece la comparativa entre *clMeanShift* respecto al algoritmo sin optimización y de optimización media ofrecido por EDISON.

El proceso seguido es análogo al mostrado en el anterior caso. Se realizan diferentes simulaciones sobre las dos imágenes de test, estableciendo diferentes valores de ventana. En las figura 6.11 se muestran los resultados comparativos de la aplicación de los tres métodos sobre las dos imágenes.



(a) Imagen Lenna (b) Imagen MRI

Figura 6.11: Comparativa coste proceso de filtrado. clMeanshift vs EDISON.

Se observa que el tiempo necesario para realizar el filtrado de la imagen es inferior para el caso de *clMeanShift*, respecto al nivel sin optimizaciones de EDISON. En esta situación, se pone de manifiesto que, al realizar el proceso en paralelo por medio de una serie de núcleos OpenCL, se consigue una reducción considerable del tiempo necesario para llevar a cabo los cálculos necesarios. Comparando dichos resultados con la optimización media de EDISON se aprecia que, al seleccionar valores bajos para las ventanas, EDISON es superior a la solución propuesta. Pero en cambio, para valores altos, los resultados obtenidos para la solución propuesta son mucho mejores. Más detalladamente, en las tablas de las figuras 6.12a y 6.12b se muestran los valores de *speed-up* conseguidos para ambas imágenes por *clMeanShift* respecto a los dos tipos de simulaciones realizadas en EDISON.

Lenna		clMeanShift - GPU (s)	EDISON Ninguno (s)	EDISON Medio (s)
h=(4,4)	Tiempo	0,88	3,81	1,53
	Speed-Up	1,00	4,35	1,75
h=(4,8)	Tiempo	1,06	4,20	1,87
	Speed-Up	1,00	3,95	1,76
h=(8,4)	Tiempo	2,14	21,64	5,38
	Speed-Up	1,00	10,10	2,51
h=(8,8)	Tiempo	2,55	26,92	5,84
	Speed-Up	1,00	10,55	2,29
h=(8,16)	Tiempo	1,79	26,57	5,36
	Speed-Up	1,00	14,82	2,99
h=(16,8)	Tiempo	9,68	184,55	18,84
	Speed-Up	1,00	19,07	1,95
h=(16,16)	Tiempo	6,58	159,96	15,44
	Speed-Up	1,00	24,32	2,35
h=(32,32)	Tiempo	35,09	1198,59	44,50
	Speed-Up	1,00	34,16	1,27

(a) Speed–Up Imagen Lenna

MRI 2D		clMeanShift - GPU (s)	EDISON Ninguno (s)	EDISON Medio (s)
h=(4,4)	Tiempo	1,72	2,64	1,16
	Speed-Up	**1,00**	**1,53**	**0,67**
h=(4,8)	Tiempo	1,81	2,38	1,22
	Speed-Up	**1,00**	**1,32**	**0,68**
h=(8,4)	Tiempo	2,00	15,21	3,42
	Speed-Up	**1,00**	**7,60**	**1,71**
h=(8,8)	Tiempo	1,96	13,96	3,64
	Speed-Up	**1,00**	**7,14**	**1,86**
h=(8,16)	Tiempo	1,81	13,00	3,38
	Speed-Up	**1,00**	**7,18**	**1,87**
h=(16,8)	Tiempo	3,86	99,72	11,24
	Speed-Up	**1,00**	**25,82**	**2,91**
h=(16,16)	Tiempo	3,38	104,48	10,43
	Speed-Up	**1,00**	**30,95**	**3,09**
h=(32,32)	Tiempo	4,74	876,35	42,36
	Speed-Up	**1,00**	**184,79**	**8,93**

(b) Speed–Up Imagen MRI 2D

Figura 6.12: Speed Up conseguido respecto a EDISON.

Como conclusión general del conjunto de pruebas realizada sobre imágenes 2D se puede indicar que *clMeanShift* presenta mejores prestaciones frente a otros algoritmos equivalentes basados en código secuencial. Su principal ventaja se encuentra en el procesamiento en paralelo de las distintas fases que componen el algoritmo de filtrado. Adicionalmente, existen una serie de puntos en el algoritmo propuesto que pueden ser mejorados en el futuro.

6.2.2. Proceso de filtrado para imágenes 3D

Una vez validado el proceso para imágenes 2D, se realiza una serie de simulaciones aplicando el algoritmo de filtrado de *clMeanShift* sobre la imagen de prueba 3D denominada *MRI 3D*.

El proceso de filtrado para este tipo de imágenes es el mismo que el seguido para el caso de las imágenes 2D, igual que sucede para el caso de imágenes 4D. La única diferencia es el número de componentes que

forma el vector de características que representa cada punto. Por ello, el conjunto de simulaciones realizadas están destinadas a evaluar la escalabilidad de la solución al aumentar las dimensiones de la imagen (3D y 4D), lo que se traduce en un aumento del número de puntos a tratar. Por lo tanto, el objetivo es establecer el impacto que supone en el proceso de filtrado el aumento del número de puntos a tratar, así como el aumento de la dimensión del vector de características que representa cada punto.

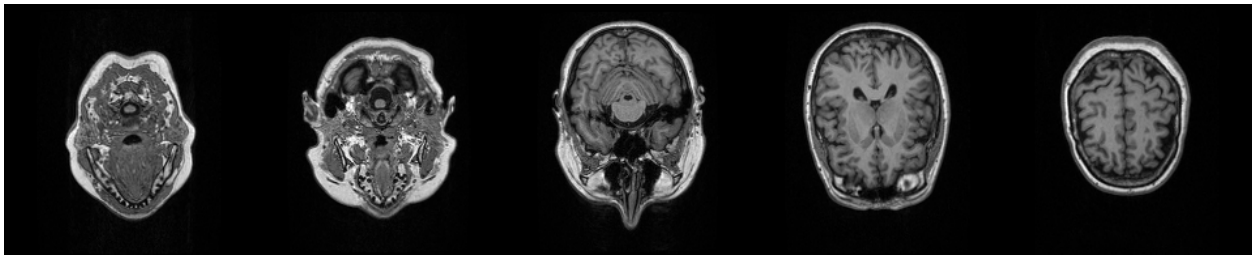
En este caso concreto, no se puede realizar la evaluación del beneficio de tiempo de proceso que ofrece *clMeanShift* frente a otras herramientas equivalentes, al no disponer de ninguna herramienta que realice el proceso de segmentación sobre este tipo de imágenes. De igual manera, la influencia de los parámetros de ventana establecidos, para este caso, presentan las mismas propiedades que para el caso de imágenes 2D por lo que se limita el número de casos a estudiar a dos representativos. Este conjunto de casos establecen los valores de ventana mostrados en la tabla 6.3. El resto de parámetros se mantienen respecto al caso anterior, es decir, $\epsilon = 0{,}01$ y número máximo de iteraciones a 100.

Simulación	h_s	h_r
Simulación 1	4	4
Simulación 2	8	8

Tabla 6.3: Filtrado 3D. Parámetros de simulación.

En la figura 6.13, se muestran diferentes cortes del resultado del proceso para cada uno de las simulaciones, comparando dicho resultado respecto a los cortes originales.

Igual que sucede para el caso de las imágenes 2D, al aplicar el filtro *Mean Shift* sobre el volumen 3D que representa la image tratada, se consigue la uniformidad de las distintas partes que componen el volumen. Los valores establecidos como parámetros de ventana para este caso tienen los mismos efectos indicados anteriormente, tal y como se aprecia al comparar los resultados de ambas simulaciones. De esta manera, tal y como se ha fundamentado durante la descripción del algoritmo *Mean Shift* , al establecer el conjunto de puntos que componen la imagen por medio de su representación en el Espacio de Características, no importa la naturaleza de la información procesada para llevar a cabo el proceso de filtrado. De esta manera, queda probado que el proceso de filtrado para imágenes 2D es también válido para imágenes 3D, y por extensión, a imágenes 4D.



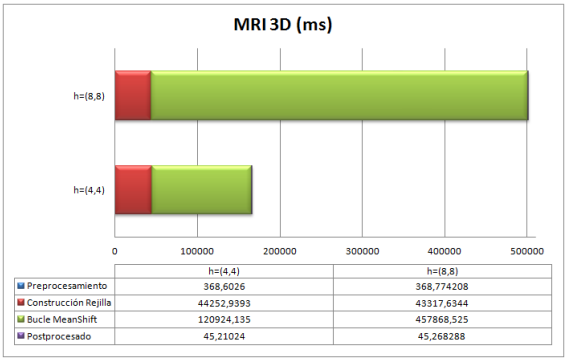
(a) Original. Corte 30. (b) Corte 60. (c) Corte 90. (d) Corte 120. (e) Corte 150.

(f) $h = (4, 4)$. Corte 30. (g) Corte 60. (h) Corte 90. (i) Corte 120. (j) Corte 150.

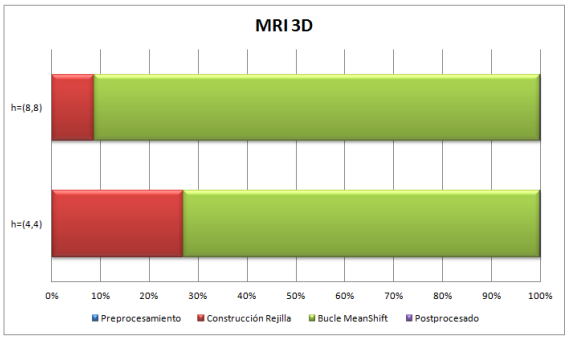
(k) $h = (8, 8)$. Corte 30. (l) Corte 60. (m) Corte 90. (n) Corte 120. (ñ) Corte 150.

Figura 6.13: Resultado proceso de filtrado. MRI 3D.

Como se ha indicado, el aspecto fundamental a evaluar para este caso es el tiempo necesario para llevar a cabo el proceso de filtrado. De esta manera, se puede estudiar el impacto que supone el aumento del número de puntos a procesar. En la figura 6.14 se muestran el tiempo de proceso consumido por cada una de las fases en que se divide el proceso de filtrado para cada una de las simulaciones realizadas. Como se observa, las dos fases que consumen más tiempo corresponden con la *Construcción de la Rejilla Tridimensional* y el *Bucle Mean Shift*.



(a) Tiempo por fase



(b) Porcentaje de tiempo por fase

Figura 6.14: MRI 3D. Coste del proceso de filtrado.

Tal y como sucede en el caso de las imágenes 2D, la mayor parte del tiempo necesario para realizar el filtrado de la imagen es consumido por el *Bucle Mean Shift*. Para este tipo de imágenes, hay que tener en cuenta que el número de puntos de la imagen es muy superior al de las imágenes 2D, pero además, hay que tener en cuenta que el número de puntos por celdas es superior, por lo que el tiempo necesario para realizar cada iteración es superior.

La principal diferencia entre la Cuadrícula establecida para una imagen 2D y una 3D es su tercera dimensión. Para el caso de una imagen 2D, la tercera dimensión de la Cuadrícula es establecida a partir del rango de valores de la componente de color L del Espacio de Características de los puntos que componen la imagen. Por el contrario, para el caso de una imagen 3D, la Cuadrícula queda definida por las tres componentes espaciales que definen su Espacio de Características. Al considerarse la imagen 3D como un volumen cúbico, su conjunto de vóxeles quedan repartidos uniformemente por las diferentes celdas de la rejilla, lo que supone que todas las celdas estén ocupadas con el mismo número de puntos de la imagen. De esta manera, el número de puntos vecinos (N) del punto procesado, que pertenecen al conjunto de 27 celdas vecinas (incluyendo la propia celda del punto), viene definido por la expresión 6.1, donde n_i indica el número de vóxeles en la dimensión i de la imagen.

$$N = 27 * \frac{n_z * n_y * n_x}{h_s^3} \tag{6.1}$$

En el caso de las imágenes 2D, al utilizar la componente L del Espacio de Características como tercera dimensión, la Rejilla Tridimensional posee celdas que no tienen ningún punto asociado, con lo que el número de puntos vecinos que participan en cada cálculo es muy inferior, reduciendo el tiempo de proceso.

Por otro lado, como se ve en las gráficas de la figura, como la imagen utilizada para la simulación no tiene un número de vóxeles que sea potencia de dos, el tiempo empleado en la construcción de la rejilla es importante, debido principalmente al algoritmo de ordenación. En este caso, y tal como se ha mostrado en la figura 6.10, trabajar con una lista de pares con un gran número de elementos provoca que el impacto del tiempo necesario para realizar su ordenación sea mayor.

Finalmente, se el tiempo necesario para realizar las fases de *Preprocesado* y *Postprocesado* es insignificante respecto al resto de fases, ya que como se ha dicho anteriormente, en estos casos se explota perfectamente el paralelismo del proceso.

6.3. Evaluación del Proceso de Etiquetado

Una vez que la imagen ha sido filtrada por medio del algoritmo *Mean Shift* , se realiza el proceso de etiquetado en el que se agrupan aquellos puntos de la misma que componen las diferentes partes que componen la imagen. El resultado de dicho proceso es una lista identificadores o etiquetas que están asignadas a cada uno de los puntos de la imagen, de tal manera, que cada *blob* quedará definido por el conjunto de puntos que poseen la misma etiqueta.

Como se ha mencionado anteriormente, el resultado del proceso de filtrado influye notablemente en el resultado del este proceso de etiquetado, y en concreto en el número de regiones identificadas.

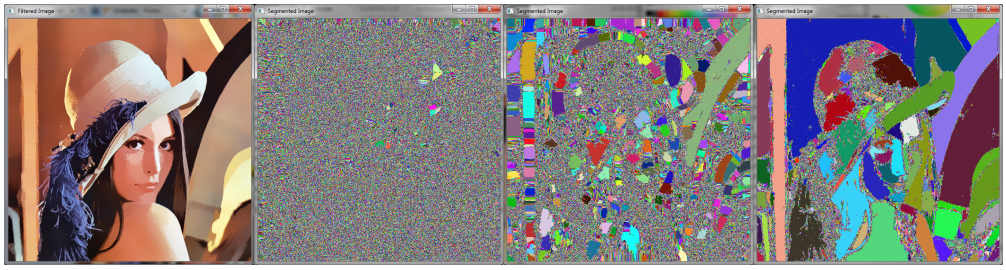
En el capítulo 4.3 se establece el valor de umbral ϵ como único parámetro del proceso de etiquetado. Este valor de umbral establece el criterio de conectividad entre los distintos puntos (píxeles o vóxeles) que componen la imagen a etiquetar, y está basado en la distancia Euclídea existente entre los valores de color de dos puntos vecinos.

Para realizar el proceso de evaluación de este algoritmo se parte de una imagen previamente filtrada, y se establecen un conjunto de simulaciones en las que se varía el valor de ϵ. Las imágenes de prueba utilizadas son las denominadas como *Lenna*, *MRI 2D* y *MRI 3D*. Para el caso de las imágenes *Lenna* y *MRI 2D* se utiliza la imagen filtrada al aplicar los valores de ventana $h = (16, 16)$, al presentar los mejores resultados. Por otro lado, para el caso de la imagen *MRI 3D*, se parte del resultado del proceso de filtrado con valores de ventana $h = (8, 8)$

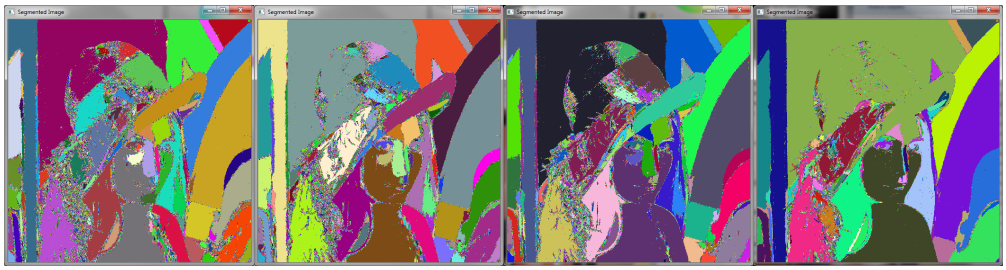
Para el caso de las dos imágenes 2D, se establecen el conjunto de valores $\epsilon = \{0{,}01; 0{,}1; 0{,}5; 1{,}0; 1{,}5; 2{,}0; 5{,}0\}$. Por el contrario, para el caso de la imagen 3D se realiza únicamente la validación del proceso al aumentar la dimensión espacial, por lo que se selecciona los valores de $\epsilon = \{0{,}5; 1{,}0; 1{,}5\}$, que como se muestra, ofrecen los mejores resultados para el caso de la imagen *MRI 2D*.

Hay que destacar, tal y como se ha indicado en el capítulo 4.3, que la única diferencia del algoritmo en el tratamiento de imágenes 2D y 3D es durante la verificación del conjunto de puntos vecinos de cada punto de la imagen para comprobar su conectividad. En el caso de la imagen 2D, el conjunto de puntos vecinos corresponden con los 8 puntos que se encuentran alrededor del punto tratado. En cambio, para el caso de una imagen 3D, hay que tener en cuenta 26 puntos vecinos, al aumentar la dimensión espacial.

Para el caso de la imagen *Lenna*, el resultado de las diferentes simulaciones se muestra en la figura 6.15. Hay que indicar, que el color asignado a cada etiqueta es realizado de manera aleatoria por lo que cada simulación presentará distinta gama de colores para identificar los diferentes *blobs*.



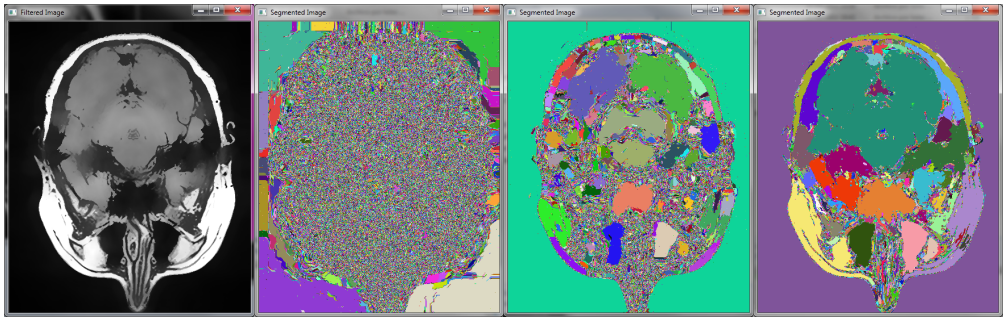
(a) Original (b) $\epsilon = 0{,}01$ (c) $\epsilon = 0{,}1$ (d) $\epsilon = 0{,}5$.



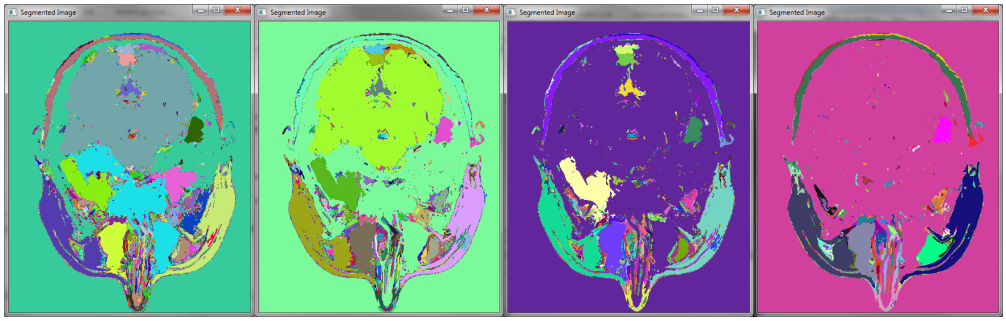
(e) $\epsilon = 1{,}0$ (f) $\epsilon = 1{,}5$ (g) $\epsilon = 2{,}0$ (h) $\epsilon = 5{,}0$

Figura 6.15: Resultado proceso de etiquetado. Lenna.

Del mismo modo, para el caso de la imagen *MRI 2D*, los resultados de las diferentes simulaciones realizadas se muestran en la figura 6.16.



(a) Original (b) $\epsilon = 0{,}01$ (c) $\epsilon = 0{,}1$ (d) $\epsilon = 0{,}5$.



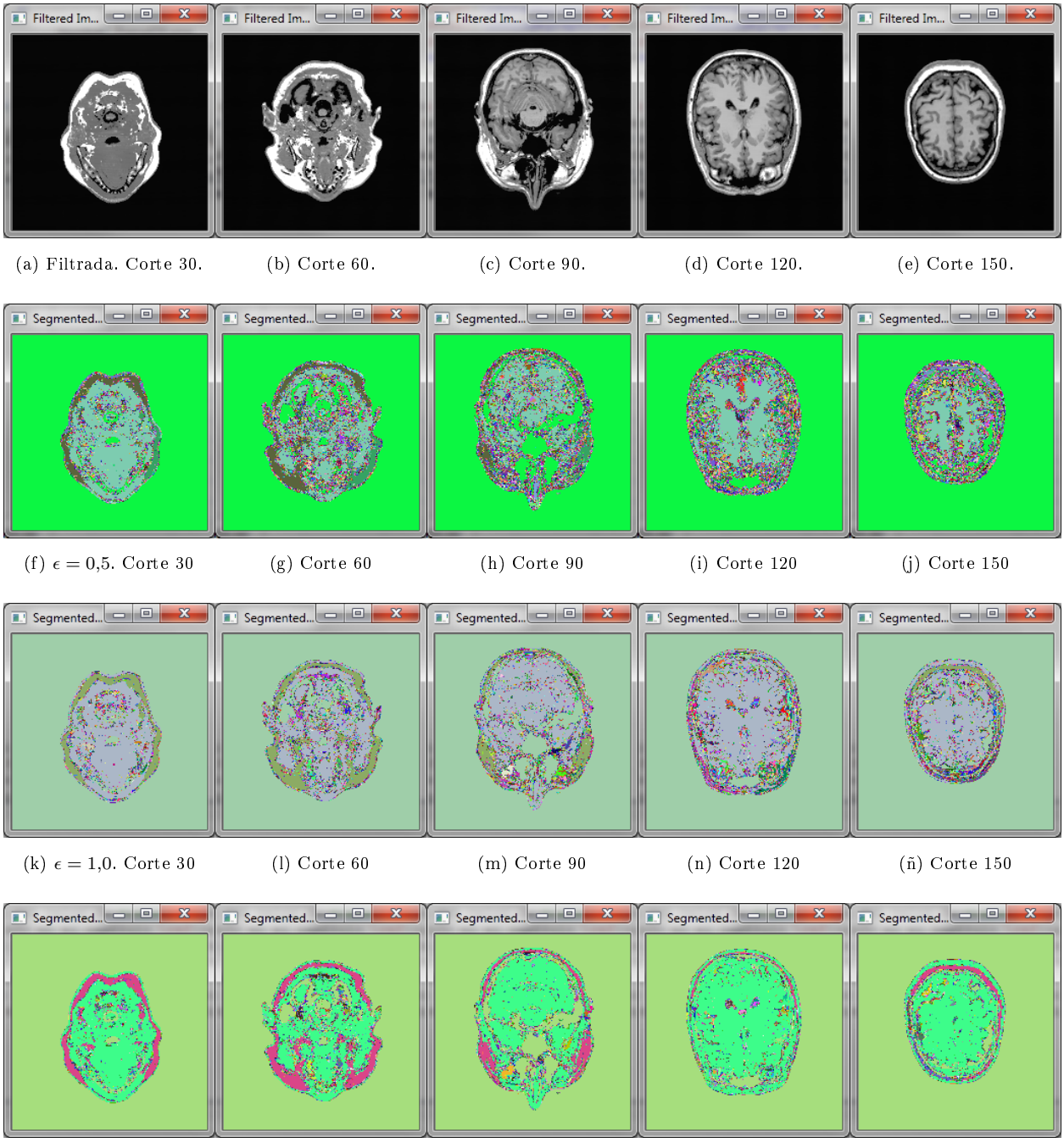
(e) $\epsilon = 1{,}0$ (f) $\epsilon = 1{,}5$ (g) $\epsilon = 2{,}0$ (h) $\epsilon = 5{,}0$

Figura 6.16: Resultado proceso de etiquetado. MRI 2D.

Finalmente, para el caso de la imagen *MRI 3D*, los resultados obtenidos se muestran en la figura 6.17

Como se observa claramente en los distintos casos de prueba, establecer un valor bajo de ϵ implica que las pequeñas variaciones de color que existan entre los diferentes puntos de la imagen no permitan etiquetar zonas claras de la imagen, tal y como se ve para el caso de $\epsilon = 0{,}01$, por lo que el número de *blobs* que se obtienen es muy elevado. Conforme dicho valor va a aumentando, el número de *blobs* obtenidos es mucho menor, al establecer la conectividad de diferentes puntos de la imagen que presentan valores de color *similares*. Adicionalmente, un exceso de umbral implica la excesiva conectividad de los puntos, tal y como se aprecia en el caso de $\epsilon = 5{,}0$ de la imagen *MRI 2D*. Este hecho se traduce en la identificación de pocos *blobs*, lo que conlleva la pérdida de la información de las diferentes partes identificables en la imagen.

Como se ha indicado anteriormente, el algoritmo de etiquetado depende considerablemente del resultado del proceso de filtrado, y en concreto, de la uniformidad de la imagen filtrada. A este factor, hay que añadir el valor de umbral ϵ que se establece. Dicho valor depende considerablemente del contraste de color o escala de grises presentado por las diferentes partes de la imagen filtrada. Cuanto mayor sea este contraste, más sencillo es el proceso de etiquetado, y por lo tanto, mejores resultados se obtienen.



(a) Filtrada. Corte 30. (b) Corte 60. (c) Corte 90. (d) Corte 120. (e) Corte 150.

(f) $\epsilon = 0{,}5$. Corte 30 (g) Corte 60 (h) Corte 90 (i) Corte 120 (j) Corte 150

(k) $\epsilon = 1{,}0$. Corte 30 (l) Corte 60 (m) Corte 90 (n) Corte 120 (ñ) Corte 150

(o) $\epsilon = 1{,}5$. Corte 30 (p) Corte 60 (q) Corte 90 (r) Corte 120 (s) Corte 150

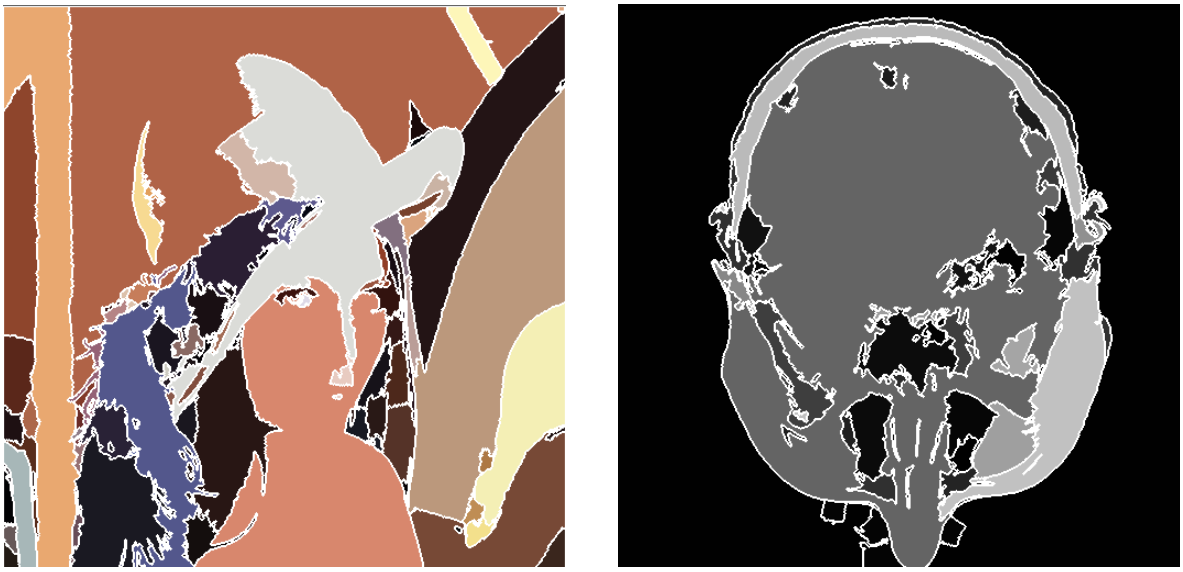
Figura 6.17: Resultado proceso de etiquetado. MRI 3D.

Finalmente, en los diferentes resultados obtenidos, se observa la existencia de *blobs* de pequeña área o volumen, sobre todo en los límites de cada *blob* principal, que pueden ser unificados con éstos. Dicho aspecto es una de las futuras mejoras que plantea el algoritmo presentado. La idea fundamental es establecer un criterio de área o volumen mínimo de *blob* que permita, por medio de un proceso adicional, unificar aquellos

blobs que no cumplan el criterio establecido con otros *blobs* vecinos.

En el caso de las imágenes 2D, el resultado del proceso de etiquetado puede ser comparado con el entorno EDISON. En este entorno, el proceso de etiquetado se realiza sin establecer ningún tipo de parámetro de similitud, por parte del usuario. En cambio, incorpora un mecanismo de fusión para eliminar los *blobs* de pequeña área por medio de un parámetro de umbral que establece el área mínima permitida.

Para el caso de las imágenes de *Lenna* y *MRI 2D*, estableciendo los mismo parámetros para realizar el proceso de filtrado, se obtienen los resultados mostrados en 6.18.



(a) Lenna (b) MRI 2D

Figura 6.18: EDISON. Proceso de etiquetado.

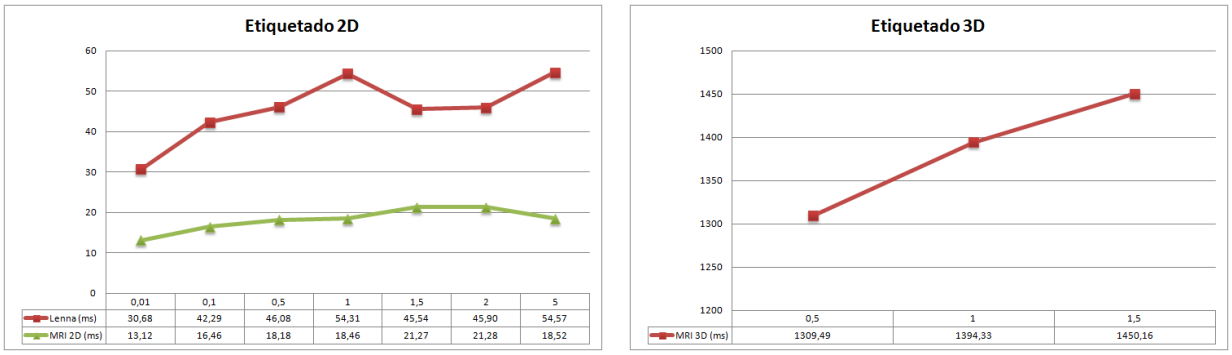
Como se observa, los resultados obtenidos en el entorno EDISON presentan una mejor calidad que los obtenidos para el caso de *clMeanShift*, principalmente, porque incorpora el mecanismo de fusión de *blobs* que unifica diferentes zonas de la imagen. Por ello, se deja como posible vía de trabajo la mejora del algoritmo propuesto.

6.3.0.1. Coste del Proceso de Etiquetado

Otro de los aspectos a evaluar del algoritmo propuesto es su coste computacional, y como afecta al proceso de segmentación. Como se ha incluido en el

En la figura 6.19a se muestran los resultados obtenidos para las diferentes simulaciones realizadas sobre las dos imágenes 2D. Del mismo modo, en la figura 6.19b se muestran los resultados obtenidos para el caso de la imagen *MRI 3D*.

Como se observa el tiempo necesario para realizar el proceso de etiquetado es ínfimo respecto al tiempo necesario para realizar el proceso de filtrado, tanto para una imagen 2D como para una imagen 3D. El tiempo



(a) Imagen Lenna

(b) Imagen MRI

Figura 6.19: Coste del Proceso de Etiquetado.

consumido para completar el proceso depende de una serie de factores fundamentales.

En primer lugar, el número de píxeles o vóxeles aumenta el tiempo necesario para realizar el proceso de manera lineal, tal y como se aprecia al comparar el tiempo necesario para etiquetar ambos tipos de imágenes.

Por otro lado, el valor de ϵ afecta en cierta manera al tiempo necesario para completar el proceso. Como se ha indicado en el capítulo 4.3, una vez construída la *Lista de Equivalencia*, se realiza un proceso de reducción, en el que a cada punto se le asigna la etiqueta correspondiente al punto de su grupo o *blob* con menor valor de etiqueta. Dicho proceso es iterativo por lo que al aumentar el valor del umbral ϵ se aumentan la superficie o volumen que forman los distintos *blobs*, lo que supone un mayor número de iteraciones en este proceso. A pesar de este aspecto, como se comprueba dicho coste no es apreciable al variar dicho umbral para la misma imagen.

Finalmente, como se ha indicado anteriormente, el resultado del proceso de filtrado influye notablemente en el número de *blobs identificables*, por lo que cuanto más uniforme sea la imagen, es decir, menos partes sean identificables, más superficie o volumen tendrán los *blobs* identificados. Esto supone, como sucede para el factor anterior, un aumento del número de iteraciones necesarias para completar el proceso.

Capítulo 7

Conclusiones

Este Trabajo Fin de Master ha presentado *clMeanShift* como una revisión del algoritmo de *Mean Shift* basada en la paralelización del conjunto de procesos realizados. Dicha paralelización se basa en el modelo especificado por el estándar OpenCL, que permite el desarrollo de algoritmos paralelos destinados a arquitecturas masivamente paralelas.

La utilización de este modelo permite explotar la paralelización del algoritmo *Mean Shift* con la consiguiente reducción del tiempo necesario para realizar su proceso, aspecto fundamental cuando se quiere utilizar este algoritmo para problemas en los que el tiempo es fundamental o cuando el tamaño de la información a tratar es elevado. Adicionalmente, al realizar el proceso en el denominado *Espacio de Características*, permite tratar la información de distinta naturaleza de manera homogénea. Por ello, el diseño de *clMeanShift* establece ambos aspectos como sus pilares fundamentales.

A pesar de la existencia de otros modelos equivalentes, OpenCL establece las bases que permiten explotar la capacidad de cálculo de los procesadores masivamente paralelos, con el objetivo de reducir el tiempo de cómputo de algoritmos de alto coste computacional. Aunque existen otros modelos equivalentes específicos para ciertas arquitecturas, como CUDA, la utilización del modelo establecido por OpenCL permite ampliar el abanico de dispositivos compatibles con las implementaciones desarrolladas, pudiendo aumentar considerablemente su aplicación en diferentes ámbitos. Entre los principales problemas existentes actualmente al desarrollar una solución basada en OpenCL, está el hecho de que aun se trata de un estándar que está en vías de maduración. Esto hace que los fabricantes aun estén desarrollando sus controladores compatibles con OpenCL, además de existir poca bibliografía al respecto en la que se presenten casos de uso equivalentes, que permitan establecer una serie de criterios claros para el diseño de algoritmos. Además, tal y como se ha indicado en el capítulo 5, aunque OpenCL es un estándar que puede ser aplicado sobre una gran gama de dispositivos, a la hora de desarrollar el conjunto de núcleos que componen el algoritmo diseñado, es necesario tener muy en cuenta el conjunto de restricciones que presenta la arquitectura sobre la que se van a ejecutar los mismos, ya que pueden influir muy negativamente en sus prestaciones.

clMeanShift parte del algoritmo básico *Mean Shift* , y divide el proceso de segmentación en dos fases

fundamentales. En primer lugar, se realiza el filtrado de la información, por medio de la aplicación del algoritmo *Mean Shift* . Posteriormente, se realiza un proceso de etiquetado cuyo objetivo es la identificación de los diferentes grupos o *blobs* que forman parte de la información procesada.

Como se ha visto durante la evaluación de *clMeanShift*, la paralelización del proceso supone una reducción considerable del tiempo necesario para llevar a cabo el conjunto de cálculos, respecto a la implementación secuencial básica. La utilización de núcleos desarrollados en OpenCL permiten explotar las capacidades de cálculo en paralelo de los Dispositivos OpenCL, lo que permite acelerar aquellos puntos del algoritmo en los que se realiza el mismo conjunto de cálculos sobre una gran cantidad de datos.

De los dos procesos básicos realizados por *clMeanShift*, el proceso de filtrado es el que más tiempo consume. Con la utilización de una Cuadrícula Tridimensional Uniforme para distribuir los puntos que componen la información a filtrar, se consigue acelerar el proceso de búsqueda de puntos vecinos, con la consecuente reducción del tiempo por iteración del algoritmo de filtrado.

Del mismo modo, el resultado del proceso de filtrado es controlado por un pequeño número de parámetros, lo que da la capacidad de establecer una serie de criterios para su elección, que permiten configurar la resolución del mismo dependiendo de las características propias de la información a procesar.

A pesar de que la mayor parte de las simulaciones realizadas se han centrado en imágenes 2D, la escalabilidad del algoritmo a imágenes 3D y 4D es inmediata, siendo sólo necesario definir el vector de características que define cada punto de la imagen, sin tener que realizar modificaciones en el núcleo del algoritmo. La principal diferencia en el tratamiento de cada uno de los tipos de imágenes es la cantidad de puntos a procesar, lo que supone un aumento del tiempo necesario para completar el proceso.

Por otro lado, el algoritmo de Etiquetado propuesto, está basado en la denominada *Lista de Equivalencia*, que permite realizar el proceso de manera iterativa y paralela de manera eficiente por medio del modelo OpenCL. El coste de dicho proceso es insignificante respecto al proceso de filtrado *Mean Shift* . Como se ha visto, durante el conjunto de pruebas realizadas, la identificación del conjunto de *blobs* que compone la imagen depende considerablemente de la calidad de la imagen filtrada, y del único parámetro que controla el proceso de etiquetado. Dichos aspectos influyen notablemente en el número de *blobs* identificados, así como en el error cometido durante el proceso. Del conjunto de pruebas realizadas, se observa como el algoritmo actual presenta ciertos errores en el etiquetado, especialmente en los bordes de los *blobs* principales, lo que supone la generación de *blobs* de pequeño tamaño. Dicho aspecto es mejorable desarrollando algún mecanismo secundario que elimine este conjunto de *blobs* de pequeño tamaño, fusionándolos con los principales.

Aunque originalmente, *clMeanShift* está diseñado para realizar la segmentación de imágenes 2D, 3D y 4D en el ámbito médico, se consideran una serie de vías de investigación cuyo objetivo es la aplicación del algoritmo desarrollado en otros problemas de Visión Artificial, y una mayor reducción del coste computacional, que permita su aplicación en sistemas en los que el tiempo de proceso es fundamental, como son los sistemas de tiempo real. Del mismo modo, se plantea enviar los resultados obtenidos a alguna de las revistas especializadas para su difusión.

De manera resumida, se pueden considerar los siguientes temas a desarrollar:

- Aplicación de otras funciones de núcleo (p. ej. Gaussiana) para realizar el proceso de filtrado *Mean Shift* .
- Establecimiento de algún tipo de criterio, automático o semiautomático, para la selección de los parámetros de ventana (p. ej. ventana dinámica) que se adapten a la información a tratar.
- Incorporación de alguna de las propuestas realizadas para acelerar la convergencia del proceso, con el objetivo de reducir el número de iteraciones necesarias.
- Organización de la información en algún tipo de estructura de datos (p. ej. Kd–Tree) que permita realizar el conjunto de cálculos de una manera más rápida y eficiente.
- Mejora del algoritmo de etiquetado para mejorar la calidad del resultado.

Parte I

Apéndices

Apéndice A

clMeanShift. Manual de Usuario.

clMeanShift ha sido desarrollado por medio de una biblioteca independiente que pueda ser integrada en otros programas en los que sea necesaria la aplicación del algoritmo propuesto. Su implementación original está basada en una DLL específica para el Sistema Operativo Windows, pero fácilmente portable a otros Sistemas Operativos (MacOS, Linux) al sólo depender de la biblioteca propia de OpenCL.

Para poder ejecutar el conjunto de núcleos OpenCL que integra, es necesario que la máquina disponga de dispositivos compatibles con dicho estándar, así como los controladores necesarios que se habilite dicha capacidad. Normalmente, el propio fabricante del dispositivo los proporciona en las últimas versiones de los mismos.

clMeanShift define una serie de operaciones que permiten configurar y ejecutar el proceso de filtrado o segmentación. La configuración del proceso consiste básicamente en la compilación de los diferentes núcleos OpenCL para el dispositivo específico de la máquina, así como el tipo de imágen a procesar. Esto último es debido a que, al compilar el conjunto de núcleos OpenCL de *clMeanShift*, se realizan una serie de optimizaciones que dependen del tipo de información a procesar. Esta compilación se realiza estableciendo internamente variables de precompilación en los núcleos que se consideran durante su proceso de compilación.

A continuación, se detalla el conjunto de operaciones establecidas para el interfaz de *clMeanShift*, así como los tipos de datos asociados.

A.1. Operaciones

bool Init(eOpenCLDeviceType deviceType, eMeanShiftDataType datatype)

Operación encargada de inicializar el módulo y compilar los diferentes núcleos OpenCL dependiendo del tipo de imagen a procesar.

- **Parámetros:**

- **deviceType** (IN): Valor que establece el dispositivo OpenCL a utilizar. Actualmente, CPU o GPU (Ver tipo de datos asociado).
- **datatype** (IN): Valor que establece el tipo de imagen a procesar (ver tipo de datos asociado).

▪ **Devuelve:**

- TRUE si la operación se ha realizado correctamente.

void Release(void)

Operación que libera toda la memoria dinámica reservada por el módulo.

void OpenCLInfo(void)

Operación que imprime por la salida estándar las características de los diferentes Dispositivos OpenCL instalados en la computadora.

bool Filter(float* vDataIn, float* vDataOut, eMeanShiftDataType MeanShiftDataType, const int* vDataDims, float* vSigmas, float MSThreshold, unsigned int MSMaxIterations)

Operación que realiza únicamente el proceso de filtrado de la imagen aplicando *Mean Shift* .

▪ **Parámetros:**

- **vDataIn** (IN): Lista con los puntos (píxeles o vóxeles) que componen la imagen a procesar. Dichos puntos vienen representados por su valor de color en el rango [0.0,1.0].
- **vDataOut** (OUT): Resultado del proceso de Filtrado consistente en la lista con los puntos (píxeles o vóxeles) resultantes del proceso de filtrado.
- **MeanShiftDataType** (IN): Tipo de imagen a procesar. Utilizado para verificar que los núcleos OpenCL están compilados para el tipo de imagen a procesar.
- **vDataDims** (IN): Vector que indica las dimensiones de la imagen, incluyendo las dimensiones del valor de color.
- **vSigmas** (IN): Vector con los valores de ventana a aplicar a cada una de las componentes del vector de características que representa cada punto de la imagen.
- **MSThreshold** (IN): Umbral que el valor máximo del vector Mean Shift para considerar que se ha alcanzado el máximo local de un punto.
- **MSMaxIterations** (IN): Número máximo de iteraciones del bucle Mean Shift.

▪ **Devuelve:**

- TRUE si la operación se ha realizado correctamente.

bool Segmentation(float* vDataIn, float* vDataOut, unsigned int* vLabels, eMeanShiftDataType MeanShiftDataType, const int* vDataDims, float* vSigmas, float MSThreshold, unsigned int MSMaxIterations, float LblThreshold)

Operación que realiza el proceso completo de segmentación.

- **Parámetros:**
  - **vDataIn** (IN): Lista con los puntos (píxeles o vóxeles) que componen la imagen a procesar. Dichos puntos vienen representados por su valor de color en el rango [0.0,1.0].
  - **vDataOut** (OUT): Resultado del proceso de Filtrado consistente en la lista con los puntos (píxeles o vóxeles) resultantes del proceso de filtrado.
  - **vLabels** (OUT): Resultado del proceso de Etiquetado consistente en la lista etiquetas asociadas a los puntos de la imagen.
  - **MeanShiftDataType** (IN): Tipo de imagen a procesar. Utilizado para verificar que los núcleos OpenCL están compilados para el tipo de imagen a procesar.
  - **vDataDims** (IN): Vector que indica las dimensiones de la imagen, incluyendo las dimensiones del valor de color.
  - **vSigmas** (IN): Vector con los valores de ventana a aplicar a cada una de las componentes del vector de características que representa cada punto de la imagen.
  - **MSThreshold** (IN): Umbral que el valor máximo del vector Mean Shift para considerar que se ha alcanzado el máximo local de un punto.
  - **MSMaxIterations** (IN): Número máximo de iteraciones del bucle Mean Shift.
  - **LblThreshold** (IN): Criterio de similitud utilizado por el algoritmo de etiquetado
- **Devuelve:**
  - TRUE si la operación se ha realizado correctamente.

A.2. Tipos de Datos

eMeanShiftDataType:

Enumerado que representa los diferentes tipos de imagen soportados.

- **Valores:**

- MS_2D_SPATIAL_1D_RANGE: Imagen 2D en escala de grises.
- MS_2D_SPATIAL_3D_RANGE: Imagen 2D en color RGB.
- MS_3D_SPATIAL_1D_RANGE: Imagen 3D en escala de grises.
- MS_3D_SPATIAL_3D_RANGE: Imagen 3D en color RGB.
- MS_4D_SPATIAL_1D_RANGE: Imagen 4D (Tiempo + Volumen) en escala de grises.
- MS_4D_SPATIAL_3D_RANGE: Imagen 4D (Tiempo + Volumen) en color RGB.
- MS_UNKNOWN_DATATYPE: Tipo desconocido.

eOpenCLDeviceType

Enumerado que representa los diferentes tipos de Dispositivos OpenCL soportados.

- **Valores:**
  - MS_DEVICE_TYPE_CPU: Procesador de la computadora.
  - MS_DEVICE_TYPE_GPU: Procesador gráfico.

Bibliografía

[1] Adams, A., Gelfand, N., Dolson, J. y Levoy, M. Gaussian Kd-Trees for fast high-dimensional filtering. En *ACM SIGGRAPH 2009 papers*, SIGGRAPH '09, páginas 21:1–21:12. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-726-4.

[2] Batcher, K. E. Sorting networks and their applications. páginas 307–314. 1968.

[3] Buck, I. Brook Specification, versión 0.2. Specification, 2003.

[4] Carreira-Perpiñan, M. A. Acceleration strategies for gaussian mean-shift image segmentation. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, vol. 1, páginas 1160–1167, 2006. ISSN 1063-6919.

[5] Cheng, Y. Mean shift, mode seeking, and clustering. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 17(8), páginas 790–799, 1995.

[6] Christoudias, C., Georgescu, B. y Meer, P. Synergism in low level vision. En *Pattern Recognition, 2002. Proceedings. 16th International Conference on*, vol. 4, páginas 150–155. IEEE, 2002.

[7] Comaniciu, D. y Meer, P. Mean shift analysis and applications. En *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, vol. 2, páginas 1197–1203. Ieee, 1999.

[8] Comaniciu, D. y Meer, P. Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, páginas 603–619, 2002.

[9] Comaniciu, D., Ramesh, V. y Meer, P. Real-time tracking of non-rigid objects using mean shift. En *cvpr*, página 2142. Published by the IEEE Computer Society, 2000.

[10] Comaniciu, D. I. *Nonparametric robust methods for computer vision.* Tesis Doctoral, New Brunswick, NJ, USA, 2000. AAI9958400.

[11] Fangfang, F. y Ma, K.-L. Parallel mean shift for interactive volume segmentation. En *In Proceedings of MICCAI Workshop on Machine Learning in Medical Imaging*, páginas 67–75. 2010.

[12] Fukunaga, K. y Hostetler, L. The estimation of the gradient of a density function, with applications in pattern recognition. *Information Theory, IEEE Transactions on*, vol. 21(1), páginas 32–40, 1975.

[13] GEORGESCU, B., SHIMSHONI, I. y MEER, P. Mean shift based clustering in high dimensions: A texture classification example. En *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, páginas 456–463. IEEE, 2003.

[14] GONZALEZ, R. C. y WOODS, R. E. *Digital Image Processing (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013168728X.

[15] HAWICK, K. A., LEIST, A. y PLAYNE, D. P. Parallel graph component labelling with GPUs and CUDA. *Parallel Comput.*, vol. 36, páginas 655–678, 2010. ISSN 0167-8191.

[16] INTEL CORPORATION. *Writing Optimal OpenCL Code with Intel OpenCL SDK*, 2011.

[17] ISO/IEC. ISO C Standard 1999. Informe técnico, 1999. ISO/IEC 9899:1999 draft.

[18] KAFTAN, J. N., BELL, A. A. y AACH, T. Mean Shift Segmentation Evaluation of Optimization Techniques. En *Proceedings of the Third International Conference on Computer Vision Theory and Applications, VISAPP 2008*, páginas 365–374. INSTICC - Institute for Systems and Technologies of Information, Control and Communication, Funchal, Madeira - Portugal, 2008.

[19] KALOJANOV, J. y SLUSALLEK, P. A parallel algorithm for construction of uniform grids. En *Proceedings of the Conference on High Performance Graphics 2009*, HPG '09, páginas 23–28. ACM, New York, NY, USA, 2009. ISBN 978-1-60558-603-8.

[20] KANUNGO, T., MOUNT, D., NETANYAHU, N., PIATKO, C., SILVERMAN, R. y WU, A. An efficient-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, páginas 881–892, 2002.

[21] KARIMI, K., DICKSON, N. G. y HAMZE, F. A performance comparison of cuda and opencl. *Read*, vol. cs.PF(1), página 12, 2010.

[22] KHRONOS OPENCL WORKING GROUP. *The OpenCL Specification, versión 1.1*, 2011.

[23] KIRK, D. B. y HWU, W.-M. W. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edición, 2010. ISBN 0123814723, 9780123814722.

[24] LI, P. y XIAO, L. Mean shift parallel tracking on GPU. *Pattern Recognition and Image Analysis*, páginas 120–127, 2009.

[25] LINDBLOOM, B. Bruce Lindbloom Website: Color model transformations", note = `http://www.brucelindbloom.com/index.html?equations.html`. ????

[26] MAYER, A. y GREENSPAN, H. An adaptive mean-shift framework for mri brain segmentation. *Medical Imaging, IEEE Transactions on*, vol. 28(8), páginas 1238–1250, 2009.

[27] MESSAGE PASSING INTERFACE FORUM. MPI: A Message-Passing Interface Standard, versión 2.2. Specification, 2009.

[28] NVIDIA CORPORATION. *OpenCL Best Practices Guide*, 2.3 edición, 2009.

[29] NVIDIA CORPORATION. *OpenCL Programming Guide for the CUDA Architecture*, 2.3 edición, 2010.

[30] NVIDIA CORPORATION. NVIDIA CUDA C: Programming Guide, versión 4.0. Specification, 2011.

[31] NYLAND, L., HARRIS, M. y PRINS, J. Fast N-Body simulation with CUDA. En *GPU Gems 3* (editado por H. Nguyen), capítulo 31. Addison Wesley Professional, 2007.

[32] OPENMP ARCHITECTURE REVIEW BOARD. OpenMP Application Program Interface, versión 3.1. Specification, 2011.

[33] SCHANDA, J. y ON ILLUMINATION, I. C. *Colorimetry: understanding the CIE system*. CIE/Commission internationale de léclairage, 2007. ISBN 9780470049044.

[34] XIAO, C. y LIU, M. Efficient Mean-shift Clustering Using Gaussian KD-Tree. *Comput. Graph. Forum*, vol. 29(7), páginas 2065–2073, 2010.

[35] XIAO-GU, S., MAN-CHUN, L., YONG-XUE, L., LU, T. y WEI, L. Accelerated segmentation approach with CUDA for high spatial resolution remotely sensed imagery based on improved mean shift. En *Urban Remote Sensing Event, 2009 Joint*, páginas 1–6. IEEE, 2009.