UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Departamento de Sistemas Informáticos y Computación

Algoritmos de Detección y Filtrado de Imágenes para Arquitecturas Multicore y Manycore

Trabajo de Tesis Doctoral

Presentado por María Guadalupe Sánchez Cervantes

Dirigido por Dr. Vicente Vidal Gimeno y
Dr. Jordi Bataller Mascarell

Valencia, Abril de 2013

Resumen

En esta tesis se aborda la eliminación de ruido impulsivo, gaussiano y speckle en imágenes a color y en escala de grises. Como caso particular se puede mencionar la eliminación de ruido en imágenes médicas.

Algunos métodos de filtrado son costosos computacionalmente y más aún, si las imágenes son de gran tamaño. Con el fin de reducir el coste computacional de dichos métodos, en esta tesis se utiliza hardware que soporta procesamiento paralelo, como lo son los cores CPU con procesadores *multicore* y GPUs con procesadores *manycore*.En las implementaciones paralelas en CUDA, se configuran algunas características con la finalidad de optimizar el procesamiento de la aplicación en las GPUs.

Esta tesis estudia por un lado, el rendimiento computacional obtenido en el proceso de eliminación de ruido impulsivo y uniforme. Por otro lado, se evalúa la calidad después de realizar el proceso de filtrado. El rendimiento computacional se ha obtenido con la paralelización de los algoritmos en CPU y/o GPU. Para obtener buena calidad en la imagen filtrada, primero se detectan los píxeles corruptos y luego se filtran únicamente los píxeles que se han detectado como corruptos. Por lo que respecta a la eliminación de ruido gaussiano y speckle, el análisis del filtro difusivo no lineal ha demostrado ser eficaz para este caso.

Los algoritmos que se utilizan para eliminar el ruido impulsivo y uniforme en las imágenes, y sus implementaciones secuenciales y paralelas se han evaluado experimentalmente en tiempo de ejecución (speedup) y eficiencia en tres equipos de cómputo de altas prestaciones. Los resultados han mostrado que las implementaciones paralelas disminuyen considerablemente los tiempos de ejecución secuenciales.

Finalmente, en esta tesis se propone un método para reducir eficientemente el ruido en las imágenes sin tener información inicial del tipo de ruido contenido en ellas.

Abstract

This thesis focuses on the removal of impulse noise, Gaussian and speckle in color images and grayscale. As a particular case we can mention the elimination of noise in medical images.

Some filtering methods are computationally expensive and even more so, if the images are large. In order to reduce the computational cost of such methods, in this thesis we use hardware that supports parallel processing such as CPU cores with *multicore* processors and GPUs with *manycore* processor, In CUDA parallel implementations, some features are set in order to optimize the application processing on GPUs.

This thesis studies on one side the computational efficiency obtained by the process of elimination of impulsive and uniform noise. On the other side the quality is evaluated after performing the filtering process. The computational performance is obtained with the parallelization of the algorithms in CPU and/or GPU. In order to obtain good quality in the filtering image, first the corrupted pixels are detected and then only the corrupted pixels that have been detected as corrupted are filtered. From which concerns to the removing of the Gaussiano and Speckle noise the analysis of the nonlinear diffusive filter has proved to be effective in this case.

The algorithms used to eliminate impulsive noise and uniform images, and their sequential and parallel implementations have been evaluated experimentally runtime (speedup) and efficiency in three computers of high performance computing. The results have shown that the parallel implementations considerably reduce the execution times regarding sequential implementations.

Finally, in this thesis we propose a method to efficiently reduce noise in images without initial information on the type of noise contained in them.

Resum

En esta tesi s'aborda l'eliminació de soroll impulsiu, gaussiano i speckle en imatges a color i en escala de grisos. Com a cas particular es pot mencionar l'eliminació de soroll en imatges mèdiques. Alguns mètodes de filtrat són costosos computacionalment i més encara, si les imatges són de grans mides. A fi de reduir el cost computacio-

nal dels dits mètodes, en esta tesi s'utilitza hardware que suporta processament paral·lel, com ho són els cores CPU amb processadors *multicore* i GPUs amb processadors *manycore*. En les implementacions paral·leles en CUDA, es configuren algunes característiques amb la finalitat d'optimitzar el processament de l'aplicació en les GPUs. Esta tesi estudia d'una banda, el rendiment computacional obtingut en el procés d'eliminació de soroll impulsiu i uniforme. D'altra banda, s'avalua la qualitat després de realitzar el procés de filtrat. El rendiment computacional s'ha obtingut amb la paralelización dels algoritmes en CPU y/o GPU. Per a obtindre bona qualitat en la imatge filtrada, primer es detecten els píxels corruptes i després són filtrats només els píxels que s'han detectat com a corruptes. Pel que fa a l'eliminació de soroll gaussiano i speckle, l'anàlisi del filtre difusiu no lineal ha demostrat ser eficaç per a este cas.

Els algoritmes que s'utilitzen per a eliminar el soroll impulsiu i uniforme en les imatges, i les seues implementacions seqüencials i paral·leles s'han avaluat experimentalment en temps d'execució (speedup) i eficiència en tres equips de còmput d'altes prestacions. Els resultats han mostrat que les implementacions paral·leles disminuïxen considerablement els temps d'execució seqüencials. Finalment, en esta tesi es proposa un mètode per a reduir eficientment el soroll en les imatges sense tindre informació inicial del tipus de soroll contingut en elles.

Dedicado con todo mi amor y gratitud a mis papás,
Aldegundo Sánchez y María de la Luz Cervantes

A mis hermanas,
Luz, Lilia y Mary.

En memoria de mis abuelitos y tíos.

Al ITCG.

Agradezco principalmente a Dios y a la virgen de Guadalupe por estar siempre a mi lado, por todo cuanto soy y tengo.

Con mi más sincero reconocimiento y agradecimiento al Dr. Vicente Vidal Gimeno por su calidad humana y profesional de la que en estos años he tenido la oportunidad de conocer. Gracias por su total apoyo e infinita paciencia durante todo este tiempo de preparación para obtener el grado.

Agradezco al Dr. Jordi Bataller Mascarell su apoyo permanente en la realización de esta tesis.

Al Dr. Gumersindo Verdú y al Dr. Josep Arnal por su amistad y por permitirme utilizar equipo informático para la conclusión de mi tesis.

A la UPV y al DSIC por darme la oportunidad de crecer a nivel profesional.

A la DGEST en México a través del programa PROMEP por los apoyos económicos recibidos durante la realización de esta tesis doctoral.

Al Dr. Guillermo de Anda, al Ing. Roberto Gudiño Venegas, así como todos los compañeros profesores por creer en mí.

Gracias a mis familiares, a todos mis amigos de México y España, que con sus palabras de aliento y apoyo me han ayudado a concluir esta etapa de mi vida.

Índice general

Índice de tablas

Índice de figuras

Índice de Algoritmos

Capítulo 1

Introducción

1.1 Motivación

Hoy en día en la era de la digitalización, son numerosas las áreas de aplicación involucradas en el procesamiento digital de imágenes, con el objetivo de mejorar la calidad de la imagen para una correcta interpretación humana o para facilitar la búsqueda de información. No son pocas las aplicaciones que necesitan trabajar con imágenes de gran tamaño, utilizando algoritmos costosos que precisan de una alta capacidad de cálculo, e incluso proporcionar los resultados en tiempo real.

Durante la formación de la imagen, adquisición, almacenamiento y/o transmisión, muchos son los factores que introducen ruido, afectando la calidad de la imagen digital (González y Woods 2008).

Los tipos más comunes de ruido son el ruido gaussiano y el ruido impulsivo. Por un lado está el ruido gaussiano que aparece en el momento de la adquisición de la imagen por un sensor de una cámara, escáner u otro dispositivo en malas condiciones y tiene como consecuencia el emborronamiento de todos los píxeles de la imagen, provocando un efecto de bruma (*blur*). Por otro lado, esta el ruido impulsivo que aparece durante la transmisión de la imagen por un canal contaminado, afectando solo a ciertos píxeles de la imagen (Camarena 2009). Hay otro tipo de ruido que se aborda en esta tesis denominado ruido speckle (Racine y col. 1999), es un ruido granular que existe en imágenes médicas tomadas con un escáner ultrasónico.

Como caso concreto podemos referirnos a la detección y eliminación de ruido en imágenes médicas obtenidas mediante rayos X o Radiografía Computarizada (CR) en condiciones adversas.

En el área de la medicina con rayos X, particularmente en mamografías, el mal funcionamiento del dispositivo del mamógrafo, el uso de un pequeño número de proyecciones en aparatos de rayos X o la transmisión de la imagen a través de un canal ruidoso, puede introducir ruido en las imágenes (impulsivo, gaussiano y/o speckle), causando dificultades en la interpretación de la imagen y afectar por ejemplo, la detección de microcalcificaciones (Mayo 2007; Gaona y col. 2012; Álvarez, Guevara y Holguín 2006).

El uso de la radiografía computarizada (CR) en la práctica clínica ha seguido por un aumento en el número de exploraciones realizadas y los casos de sobredosis en los pacientes, especialmente en niños en aplicaciones pediátricas, lo que conduce a un riesgo de sobreexposición sistemática. De acuerdo con la Comisión Internacional de Protección Radiológica (ICRP), las dosis de los pacientes en CR, especialmente en el caso de los niños, siempre debe mantenerse lo más bajo posible, de acuerdo al criterio ALARA (ICRP 2007). El informe hace énfasis en la justificación de los procedimientos médicos y en la optimización de la protección radiológica. Esos son los mecanismos adecuados para evitar la exposición a la radiación innecesaria o improductiva. Son pocos los trabajos en donde se realiza un análisis de la reducción de la exposición (mAs) en los pacientes (ICRP 1993). Es importante buscar otras estrategias para la reducción de la dosis a nivel de software.

La eliminación de ruido o filtrado de las imágenes y la correcta percepción de áreas de color es importante en aplicaciones relacionadas con la biomedicina, ciencia, comunicación de video, ciencias de la tierra, cultural (preservación del patrimonio cultural), inspección robótica y videovigilancia y es muy estudiado en el campo de procesamiento de imágenes.

Muchos filtros han sido presentados en diferentes artículos con la finalidad de reducir el ruido en imágenes. Los filtros de Media Vectorial (VMF) (Astola, Haavusto y Neuvo 1990; Plataniotis y Venetsanopoulos 2000; Lukac y col. 2005), son métodos muy usados para la reducción de ruido impulsivo en imágenes a color, porque están basados en la teoría de la estadística robusta. Sin embargo, este filtro (VMF) se aplica a cada píxel de la imagen sin considerar si el píxel es ruidoso o no, produciendo un proceso de difuminación. También debido a su naturaleza no lineal, estos métodos son bastante exigentes computacionalmente (Camarena y col. 2010b). Con la finalidad de mejorar el

inconveniente de la difuminación, se han propuesto una serie de filtros que combinan una etapa de detección con una de eliminación o filtrado (Camarena y col. 2010b; Smolka 2010; Camarena y col. 2008; Camarena y col. 2010a; Morillas, Gregori y Hervás 2009); de esta manera solo se corrigen los píxeles que son detectados como corruptos. Para la detección de los píxeles corruptos, el concepto de *peer group* (Kenney y col. 2001; Smolka 2005) es una de las técnicas recientemente utilizadas, que combinado con una métrica de distancia, proporciona buenos resultados en calidad y conservan los detalles finos de la imagen. Los filtros basados en la teoría fuzzy (González y Woods 2008; Camarena y col. 2010b) son también apropiados para la eliminación de ruido porque puede hacer frente a la naturaleza no lineal de las imágenes digitales y se puede distinguir entre ruido y estructura propia de la imagen.

Los anteriores métodos de filtrado (VMF, basados en *peer group*, fuzzy) son costosos computacionalmente y más aún, si las imágenes son de gran tamaño. Con el fin de reducir el coste computacional, en esta tesis se ha utilizado hardware que soporta procesamiento paralelo, como lo son los cores CPU (que contienen procesadores *multicore*) y las tarjetas gráficas también conocidas como *Graphics Processing Unit* (GPUs) que tienen procesadores *manycore* (Vajda 2011), que permiten la parametrización y paralelización de los algoritmos.

En los últimos años la incorporación de GPUs en tarjetas gráficas han conseguido una mejora de velocidad computacional significante, ofreciendo un alto nivel de procesamiento en paralelo, a un precio muy competitivo. Cada vez son más los desarrollos basados en este hardware, no sólo para implementaciones gráficas, sino también para aplicaciones de propósito general, tales como la medicina, astrofísica, biología o química computacional, procesamiento de señales entre muchos otros.

La plataforma de programación más utilizada para estas tarjetas gráficas es CUDA (*Compute Unified Device Architecture*) (NVIDIA 0). CUDA es relativamente sencilla de utilizar y está bien documentada, pero el problema radica en que es difícil optimizar el rendimiento de una aplicación, debido a algunas restricciones del hardware y a los múltiples tipos de memorias incluidas, debido a que están organizadas en varios niveles y con diferentes capacidades de almacenamiento, con diverso patrón de acceso a los datos y otras limitaciones. En esta tesis se realiza un estudio específico para determinar en cada tipo de problema cuál es la mejor forma de utilizar los recursos ofrecidos por CUDA para eliminar el ruido de una imagen digital.

1.2 Objetivos

El objetivo principal de esta tesis es analizar, diseñar e implementar algoritmos secuenciales y paralelos para el filtrado de imágenes utilizando arquitecturas *manycore* y *multicore*.

En concreto, se presenta un análisis, diseño e implementación de los algoritmos para eliminar el ruido impulsivo y gaussiano de manera secuencial y paralela. Para el ruido speckle solo se dispone de la implementación secuencial. Las versiones paralelas se implementan para arquitecturas *manycore* y *multicore*. En *manycore* se analiza la aplicación tomando en cuenta el tamaño del problema y el hardware disponible para hacer una versión adaptativa del problema a las características propias de hardware disponible. Además, se diseña un sistema de filtro sin tener la información inicial sobre el tipo de ruido existente en la imagen.

La lista detallada de objetivos es la siguiente:

- Analizar los métodos de filtrado a nivel de calidad con la finalidad de identificar los que proporcionen mejor rendimiento.
- Diseñar e implementar las versiones secuenciales y paralelas de los métodos para eliminar el ruido impulsivo en arquitecturas *multicore*.
- Diseñar e implementar las versiones secuenciales y paralela de los métodos para eliminar el ruido impulsivo en arquitecturas *manycore*.
- Diseñar e implementar las versiones secuenciales y paralela de los métodos para eliminar el ruido impulsivo para trabajar en combinación con arquitecturas *manycore* y *multicore* (implementación híbrida).
- Diseñar un método con la finalidad de filtrar la imagen sin disponer de información inicial a cerca del tipo de ruido introducido en la imagen, ya sea impulsivo o gaussiano.
- Diseñar un método adaptativo y parametrizado a la aplicación tomando en cuenta el tamaño del problema y el hardware disponible.
- Diseñar un método para obtener valores generales para los umbrales cuando se utiliza el concepto de *peer group*, métrica fuzzy y euclídea.

- Adaptar de los algoritmos implementados para el filtrado de imágenes RGB en aplicaciones médicas (imágenes en escala de grises). Demostrar la posibilidad de reducir la dosis de radiación en imágenes de rayos X a través de un filtro difusivo no lineal, manteniendo la integridad del diagnóstico médico.

En esta investigación se ha optado por la paralelización de los algoritmos que utilizan el concepto de *peer group* con métricas fuzzy, euclídea o coseno del ángulo (conceptos definidos en el siguiente capítulo) para detectar los píxeles erróneos, debido a que últimamente se han presentado varios estudios con resultados satisfactorios en calidad (Camarena y col. 2010b; Smolka 2010; Camarena y col. 2008; Camarena y col. 2010a; Morillas, Gregori y Hervás 2009). Se define otra etapa para eliminar los píxeles corruptos en la que se utiliza el concepto de sustitución del píxel ruidoso por el filtro Vectorial de Mediana o la Media Aritmética (González y Woods 2008), de esta manera se asegura que la imagen filtrada tenga una buena calidad final.

Se trata de un trabajo multidisciplinario porque hace uso de las áreas de matemáticas, informática y el campo de aplicación en la medicina.

Esta tesis tiene como finalidad obtener una calidad relativamente buena de la imagen filtrada con un tiempo computacional rápido, de tal manera que se pueda implementar en aplicaciones en tiempo real, por ejemplo, aplicaciones en secuencia de imágenes online.

1.3 Estructura del trabajo

El documento esta organizado de la siguiente manera: en el capítulo 2 se definen los conceptos generales de una imagen digital, los tipos de ruido más usuales que puede contener, específicamente los que se tratan en esta tesis. En este capítulo también se abordan los filtros que se utilizan para eliminar el ruido impulsivo, gaussiano y speckle. Además para la detección de los píxeles ruidosos, se define el concepto de *peer group* junto con las métricas fuzzy, euclídea y coseno del ángulo. Para medir la calidad después del proceso de detección y eliminación de los píxeles corruptos, se utilizan métricas de calidad que también se definen en este mismo capítulo.

En el capítulo 3, se presentan las características del equipo de altas prestaciones que incluyen las diferentes tarjetas utilizadas en esta tesis. En el aspecto de software, se define CUDA y OpenMP como las dos herramientas que se utilizaron para programar los algoritmos

paralelos. En el modelo de programación CUDA, se presenta las principales características a configurar para obtener mejor rendimiento de las tarjetas gráficas.

El capítulo 4, aborda los algoritmos secuenciales utilizados en esta tesis para el tratamiento del ruido impulsivo, uniforme, gaussiano y speckle. También se exponen los algoritmos que hacen uso del *peer group* para la detección de los píxeles corruptos y los filtros para eliminar el ruido impulsivo y uniforme. El algoritmo del filtro difusivo no lineal para eliminar el ruido gaussiano forma parte de este capítulo. Un nuevo algoritmo iterativo que disminuye los parámetros iniciales que utilizan los métodos del *peer group*, se presenta también en este capítulo.

Las versiones paralelas que se han propuesto para ser ejecutados en GPUs, cores y una combinación de ellos, se expone en el capitulo 5. Se muestra un estudio con los mejores valores de los parámetros de entrada cuando se utiliza el *peer group* y la métrica fuzzy.

En el capitulo 6 se aborda los detalles de las implementaciones de los algoritmos tanto secuenciales como paralelos en distintos tipos de arquitecturas: implementaciones de los algoritmos secuenciales en CPU, implementaciones de algoritmos paralelos en CPU, en GPU y en combinación CPU-GPU. En cada caso se aporta los resultados obtenidos del rendimiento computacional y calidad. Finalmente se hace una comparación entre las aproximaciones objeto de estudio.

Algunas de las implementaciones secuenciales se han utilizado para el tratamiento de imágenes médicas. En el capítulo 7, se presenta la eliminación de ruido impulsivo, gaussiano y speckle para diferentes imágenes médicas.

Por último en el capitulo 8 se presentan las conclusiones a las que se han llegado al realizar esta tesis y las perspectivas de investigación que se desprenden del mismo.

Capítulo 2

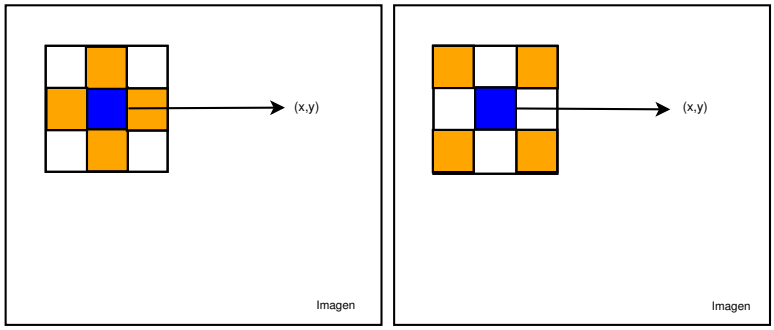
Fundamentos y estado del arte

2.1 Introducción

Una imagen digital es una representación de dos dimensiones de una matriz numérica y está compuesta de un número finito de elementos, cada uno de los cuales tiene una localización y valor. Estos elementos se llaman *píxel* (acrónimo del inglés *picture element*, elemento de imagen) o *pels*. Píxel es el término más usado para denotar el elemento de una imagen digital. La representación más usual es como una matriz tridimensional, $N \times M \times Z$, donde N y M representan las dimensiones de la imagen (alto y ancho) y Z es el número de canales o colores de la imagen. Las imágenes en un solo canal son las denominadas imágenes en escala de grises o imágenes en blanco y negro, mientras que las imágenes de tres canales son las denominadas RGB (*Red*, *Green* y *Blue*) muy frecuentemente llamada vector RGB.

Un píxel p con coordenadas (x, y) tiene cuatro vecinos de forma horizontal y vertical, cuyas coordenadas están dadas por: $(x+1, y)$, $(x-1, y)$, $(x, y+1)$, $(x, y-1)$. Este conjunto de píxeles es llamado los *4 vecinos de p* y se denota como $N_4(p)$. Los cuatro vecinos diagonales de p tienen coordenadas $(x+1, y+1)$, $(x+1, y-1)$, $(x-1, y+1)$, $(x-1, y-1)$ y son denotados por $N_D(p)$. Estos puntos, junto con los 4 vecinos, se llaman los 8-*vecinos* de p, denotado por $N_8(p)$. A lo largo de esta tesis se denominará a x_i como el valor del píxel central con coordenadas (x,y) de una ventana de filtrado W, de tamaño $n \times n$, (n=3,5,7...). Cada píxel está a una unidad de distancia desde (x, y) y algunas de las localizaciones vecinas en $N_D(p)$ y $N_4(p)$ pueden en-

contrarse fuera de la imagen digital, si (x, y) está sobre el borde de la imagen. Esta representación de vecindad se muestra en la figura 2.1.



(a) (b)

Figura 2.1: a) Vecindad horizontal y vertical de 3×3 del punto (x, y), $N_4(p)$. b) Vecindad diagonal de 3×3 del punto (x, y), $N_D(p)$.

Una vez definidos los conceptos fundamentales de una imagen digital, la sección 2.2 de este capítulo, define los tipos de ruido más usuales que puede contener una imagen digital. En la sección 2.3.1 se presentan los filtros que se utilizan en esta tesis para eliminar el ruido impulsivo, uniforme, gaussiano y speckle en una imagen digital. Además, se define el concepto del *peer group* y las métricas fuzzy, euclídea y coseno del ángulo para la detección de los píxeles corruptos. La definición de las métricas de calidad para comprobar la eficiencia de los filtros, se expone en la sección 2.3.1. El estado de arte se muestra en la sección 2.5.

2.2 Definición y tipos de ruido en una imagen digital

Durante la adquisición y/o transmisión de una imagen digital, la imagen puede contaminarse (adquirir ruido) que degrada su calidad. Una imagen está contaminada cuando algunos de sus píxeles han sido alterados o transformados, perdiendo la información del color o B/N original por otros valores. Una imagen puede verse afectada por varios factores, entre los que se encuentran, las condiciones ambientales durante la adquisición y por la calidad de los elementos propios de los sensores. Por ejemplo, en la adquisición de la imagen con una cámara CCD los niveles de luz y temperatura de los sensores, además del movimiento de la cámara o desenfocamiento, son los principales fac-

tores que afectan la cantidad de ruido en la imagen resultante. Las imágenes son corruptas durante la transmisión principalmente debido a interferencias en el canal usado para la transmisión. Por ejemplo, una imagen transmitida usando una red *wireless*, puede ser corrupta como consecuencia de relámpagos u otras perturbaciones atmosféricas (González y Woods 2008). En otras palabras el ruido es causado por ciertas deficiencias en la adquisición y/o proceso de transmisión.

Entre los ruidos más comunes de funciones de densidad probabilística (González y Woods 2008) están el ruido gaussiano, *erlang* (gamma), exponencial, uniforme e impulsivo. Las densidades exponenciales y gamma encuentran aplicación en imágenes por láser. El ruido periódico es un ruido originado típicamente de interferencias eléctricas o electromecánicas durante la adquisición de la imagen. Un ruido multiplicativo en intensidad y amplitud, es el ruido speckle. En las siguientes subsecciones se describen los ruidos que a lo largo de esta tesis se abordan.

2.2.1 Ruido Gaussiano

El ruido gaussiano es también denominado ruido *normal*. Este ruido surge en el momento de la adquisición de la imagen por un sensor de una cámara, escáner o similar en malas condiciones, a la mala iluminación y/o alta temperatura y al ruido del circuito electrónico. Este tipo de ruido tiene como consecuencia el emborronamiento de todos los píxeles de la imagen, provocando un efecto de bruma(*blur*) que los difumina.

Se dice que una variable aleatoria continua X sigue una distribución normal de parámetros $\bar{z}$ y σ. Se denota $X \sim N(\bar{z}, \sigma)$ si su función de densidad está dado por:

$$f(z) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-\bar{z})^2/2\sigma^2}, z \in \Re, \tag{2.1}$$

donde z representa la intensidad, $\bar{z}$ es el valor promedio de z y σ (sigma) es la desviación estándar (σ^2 es la varianza).

2.2.2 Ruido Uniforme

Este tipo de ruido es quizá el menos descriptivo de situaciones prácticas. Sin embargo, la densidad uniforme es bastante útil como la base para numerosas generaciones de números aleatorios que se utilizan en las simulaciones. La función de densidad probabilística esta dado por,

$$f(z) = \begin{cases} \frac{1}{b-a} & \text{si } a \leq z \leq b \\ 0 & \text{en caso contrario.} \end{cases} \tag{2.2}$$

El promedio de esta función densidad es dado por,

$$\bar{z} = \frac{a+b}{2}. \tag{2.3}$$

y su varianza por

$$\sigma^2 = \frac{(b-a)^2}{12}. \tag{2.4}$$

2.2.3 Ruido Speckle

El ruido speckle es un ruido multiplicativo en intensidad y amplitud. Se presenta en las imágenes médicas con iluminación coherente (la luz es reflejada en una superficie desigual) cuyo origen es a través de un escáner ultrasónico, sonar y radar de abertura sintética (SAR). Este ruido se desvía del ruido gaussiano, independientemente de la señal y adicionado al verdadero valor. Es generalmente mas difícil de eliminar en una imagen, porque la intensidad del ruido varía con la intensidad de la señal.

El ruido speckle se puede modelar como un ruido multiplicativo aleatorio, por ejemplo, su desviación estándar es proporcional a su media. Se puede modelar a través de,

$$I_r = I_o + n * I_o \tag{2.5}$$

donde n es una distribución uniforme aleatoria con promedio=0 y con una varianza σ^2 que por defecto es 0.04. I_o es la imagen original a la que se le agrega el ruido.

2.2.4 Ruido Impulsivo

Es el más usual y se presenta durante la transmisión de datos por un canal contaminado. Los errores sólo afectan a ciertos píxeles de la imagen a diferencia del gaussiano que afecta a todos. Se encuentra en situaciones de transito rápido, tales como defectos en los interruptores y en fenómenos tales como, las máquinas industriales en la proximidad del receptor, tránsito de comunicación en líneas eléctricas y varios interruptores sin protección. Además, las causas naturales como rayos en la atmósfera pueden afectar el proceso de transmisión. Otras causas incluyen comunicación celular, acústica submarina y tráfico en movimiento. Las imágenes en TV pueden ser corruptas por interferencias atmosféricas e imperfecciones en la recepción de la imagen (Morillas, Gregori y Hervás 2009). Este ruido también es frecuentemente llamado como "sal y pimienta". Se caracteriza por una densidad d lo cual es una proporción entre el número de píxeles corruptos y el tamaño de la imagen. La función de densidad probabilística del ruido "sal y pimienta" está dado por,

$$f(z) = \begin{cases} P_a & \text{para } z = a \\ P_b & \text{para } z = b \\ 0 & \text{en caso contrario} \end{cases} \tag{2.6}$$

Si b > a, intensidad b aparecerá como un punto luminoso en la imagen. A la inversa, el nivel a aparece como un punto negro. Si cualquiera de las dos P_a o P_b es cero, el ruido impulsivo es llamado *unipolar*. Si cualquier probabilidad es cero, y especialmente si ellos son aproximadamente igual, los valores del ruido impulsivo aparecerán granulados "sal y pimienta" uniformemente distribuidos sobre la imagen. Por esta razón el ruido bipolar también es llamado *"sal y pimienta"*. Los impulsos de ruido pueden ser negativos o positivos. Generalmente es digitalizado como un valor extremo (blanco o negro, valor máximo o valor mínimo dentro del rango de la señal) en la imagen. Como resultado, los impulsos negativos aparecen como puntos en la imagen de color negro (pimienta). Por la misma razón, los impulsos positivos aparecen como blancos (sal).

En la figura 2.2 se muestra el efecto que tiene el ruido gaussiano, uniforme, speckle e impulsivo en una imagen digital.

(a) (b)

(c) (d)

Figura 2.2: Imágenes con ruido: a) Impulsivo con densidad de 0.20, b) Uniforme con densidad de 0.20, c) Gaussiano con promedio 0 y varianza 0.01, d) Speckle con promedio 0 y varianza 0.04.

2.3 Detección y Corrección de ruido

El ruido que contiene una imagen se elimina o se reduce siguiendo un proceso de restauración, como se muestra en la figura 2.3. La imagen con ruido (I_r) es restaurada mediante filtros de eliminación de ruido proporcionando una imagen final filtrada I_f.

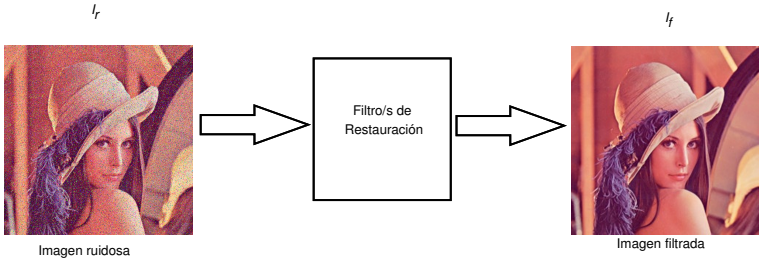


Figura 2.3: Proceso de restauración de una imagen con ruido a través de filtros.

El filtrado de imágenes es probablemente la tarea más común en el procesamiento digital de imágenes, sobre todo para prepararla para otros procesos, tales como análisis de imágenes, detección de bordes, reconocimiento de formas y/o objetos, etc. Es un paso fundamental dentro de cualquier sistema de visión por ordenador en el que las imágenes se utilizan para análisis automático o inspección humana (Camarena y col. 2008). El filtrado de ruido, es el proceso de discriminar entre la información original de la imagen y los datos ruidosos, lo que se convierte como un problema de interpretación de información.

Uno de los objetivos cuando se trabaja con filtrado de imágenes independientemente del ruido presente, es tener un buen equilibrio entre la preservación de los detalles y bordes y la supresión del ruido, tal como se pretende en (Camarena y col. 2008; Mélange, Nachtegael y Kerre 2010). Los filtros que son capaces de conservar los bordes y detalles finos, son altamente adecuados para el filtrado y mejora de imágenes. Desgraciadamente, la mejora de técnicas de procesamiento de señales tienden a difuminar los bordes y desbaratar otros detalles de la imagen (Plataniotis y Venetsanopoulos 2000). Algunos filtros al querer eliminar o reducir el ruido inevitablemente eliminan una cantidad considerable de detalles. Con el fin de no difuminar la imagen, algunos de ellos (Camarena y col. 2008; Mélange, Nachtegael y Kerre 2010), lo hacen en dos pasos, en la primera detectan los píxeles ruidosos y en la segunda corrigen solo los píxeles que previamente fueron detectados como ruidosos. De esta manera solo se filtran aquellos píxeles que

se han detectado como píxeles que contienen ruido, permaneciendo inalterables los demás.

Los filtros y los métodos de detección más comunes se exponen en las siguientes subsecciones.

2.3.1 Proceso de Corrección - Filtrado de imágenes

El ruido presente en una imagen se puede eliminar o reducir por medio de filtros basados en el dominio espacial o de frecuencia (González y Woods 2008). Los filtros basados en la frecuencia realizan operaciones sobre la transformada de *Fourier* de una imagen. Por otro lado, las operaciones de filtrado en el dominio del espacio, se llevan a cabo directamente sobre los píxeles de la imagen. El nombre *filter* es tomado del procesamiento del dominio de frecuencia, donde el filtrado del inglés *filtering* se refiere a la aceptación o al rechazo de ciertos componentes de frecuencia. La filtración se puede lograr directamente sobre la propia imagen mediante el uso de filtros espaciales (también llamados "máscaras", *kernels*, plantillas, y ventanas) (González y Woods 2008). Los filtros espaciales pueden usarse para el filtrado no lineal, a diferencia de los filtros en el dominio de frecuencia.

Los filtros en el dominio de frecuencia consisten en modificar la transformada de *Fourier* de una imagen para lograr un específico objetivo y después calcular la transformada inversa para obtener el resultado procesado. La transformada de *Fourier* se ve como una transformación de un dominio (espacial) a otro dominio (frecuencias) sin perder información de la imagen. Se pueden mencionar los filtros "de paso bajo o pasabajos" y "de paso alto o pasa altos" que se usan para atenuar las frecuencias bajas y altas respectivamente.

Los filtros en el dominio espacial consisten de una vecindad como se mencionó en 2.1 y de una operación predefinida que se realiza sobre los píxeles de la imagen abarcados por la vecindad. El filtrado crea un nuevo píxel con coordenadas iguales a las coordenadas del centro de la vecindad y cuyo valor es el resultado de la operación de filtrado (González y Woods 2008). Si la operación realizada sobre los píxeles de la imagen es lineal, entonces el filtro se refiere a un filtro espacial lineal, de lo contrario, es un filtro no lineal.

En esta tesis se han utilizado los filtros en el dominio del espacio, ya que generalmente estas técnicas son más eficientes computacionalmente y además se han presentado últimamente investigaciones que

ofrecen buenos resultados en calidad para eliminar el ruido impulsivo, gaussiano y speckle, ruidos que se tratan en esta tesis.

Los filtros más comunes en el dominio espacial (operaciones predefinidos sobre la imagen) son los filtros espaciales de suavizado. Dentro de éstos podemos encontrar:

- Filtros lineales de suavizado.
- Filtros (no lineales) de orden estadístico.

Los filtros lineales de suavizado es el promedio de los píxeles de la ventana de filtrado, que son también llamados filtros de "promedio" o filtros de "pasabajo". Su mayor uso es en la reducción de "irrelevantes" detalles en la imagen ya que los bordes son también difuminados. Los filtros de media aritmética, media geométrica, media armónica y media contra-armónica son algunos ejemplos dentro de estos filtros lineales.

Los filtros (no lineales) de orden estadístico son filtros cuya respuesta está basada en el *ranking* de los valores de los píxeles contenidos en el área de la imagen comprendida por el filtro, y después reemplazar el valor del píxel central con el valor determinado por el resultado del *ranking*. El filtro de mediana, máximos y mínimos, punto medio y media *Alpha-trimmed* (González y Woods 2008) son algunos de los ejemplos, aunque el más conocido es el filtro de mediana.

Los filtros anteriormente mencionados son muy utilizados para eliminar el ruido en imágenes en escala de grises. Las primeras soluciones para filtrar una imagen a color se trataba de aplicar las técnicas en escala de grises a cada componente de color. Sin embargo, pueden aparecer artefactos de color y otros efectos colaterales que se deben tratar con filtros más apropiados. Una de las vertientes más estudiada para eliminar el ruido en imágenes de color es el filtrado vectorial (Camarena 2009; Lukac y col. 2005). De esta manera, cada píxel se trata como un vector formado por los componentes de color y todos los canales son procesados conjuntamente. Una de las familias más populares de operadores de filtrado no lineal están los basados en la estadística de orden robusta (Lukac y col. 2005).

Dentro de los filtros de mediana vectorial, el más popular es el *Vector Median Filter*(VMF) (Lukac y col. 2005). VMF ha sido introducido como una extensión del filtro de mediana escalar. Cada píxel central de la ventana de filtrado calcula las distancias acumuladas al resto de los píxeles de la ventana. La salida VMF es la muestra $x(1) \in W$ que minimiza la distancia a las otras muestras dentro de W. Se puede

utilizar cualquier métrica para cuantificar la distancia que hay entre dos píxeles de color x_i y x_j.

La familia de Filtros de Vectores Direccionales (VDFs) (Lukac y col. 2005) operan sobre las direcciones de los vectores de la imagen con el objetivo de eliminar los vectores con las direcciones atípicas en el espacio de color. Efectúa el cálculo de la suma de los ángulos de cada píxel dentro de la ventana de filtrado al resto de los píxeles. Realiza una ordenación vectorial según los ángulos de los vectores. El más popular de esta familia es el BVDF(Lukac y col. 2005).

El VMF y DBVD están diseñados para realizar un suavizado en la imagen, debido a que no tienen en consideración si el píxel analizado es ruidoso o no y se filtran tanto los corruptos como los que no lo son. Estos filtros pueden llegar a exhibir un suavizado excesivo, afectando las texturas y los detalles finos de la imagen. Para evitar esos problemas, aparecen los filtros que se adaptan a las características locales de la imagen, los filtros adaptativos. En la literatura existen varios filtros adaptativos (González y Woods 2008), pero los que últimamente han proporcionado buena calidad en la imagen filtrada, son aquellos en los que se hace el proceso de filtrado solo de los píxeles que se detectan ruidosos. Se sustituye el valor del píxel corrupto, dejando los demás píxeles inalterables (Smolka 2010; Camarena y col. 2008). El concepto de *peer group* (Kenney y col. 2001; Smolka 2005) es uno de los métodos actuales más estudiados para la detección de los píxeles corruptos.

A continuación se amplía los conceptos de los filtros más relevantes para esta tesis.

Filtro de Media Aritmética

El Filtro de Media Aritmética (AMF) por sus siglas en inglés, (González y Woods 2008), es el filtro más simple de los de media. Sea S_{xy} que representa el conjunto de coordenadas en una ventana Wde tamaño $n \times n$, centrado en un punto x_i con coordenadas (x, y). El filtro de media aritmética calcula el valor promedio de la imagen corrupta I_r en el área definida por S_{xy}. El valor de la imagen filtrada I_f en el punto (x, y) es simplemente el cálculo de la media aritmética usando los píxeles en la región definida por S_{xy}. En otras palabras:

$$AMF_{x_i} = \frac{1}{W} \sum_{x_j \in W} x_j, \tag{2.7}$$

donde x_j es un píxel vecino de x_i con coordenadas (s, y).

Este filtro se utiliza directamente sobre la imagen ruidosa o después de un proceso de detección de píxeles corruptos. Si se utiliza directamente sobre la imagen (sin una etapa de detección) el ruido se reduce produciendo una imagen de emborronamiento, este efecto se puede mejorar si se aplica después de una etapa de detección en la que solo se modifiquen los píxeles ruidosos.

En imágenes RGB se obtiene el promedio de cada canal y se sustituye por el píxel x_i en el canal correspondiente.

$$AMF_{x_i} = \frac{1}{W} \sum_{x_j \in W} x_j(l). \tag{2.8}$$

donde (l) es R,G o B.

Sobre las bases de la detección de píxeles ruidosos, un filtro switching entre el AMF y la operación de identidad se propone en (Camarena y col. 2008). La diferencia direccional basada en la detección de ruido y un filtro de media adaptativo para eliminar el ruido impulsivo se presenta en (Pitas y Venetsanapoulos 1986).

Filtro de Mediana

Es el más conocido de los filtros de orden estadístico cuyo resultado es un ordenamiento (*ranking*). Como su nombre lo indica, reemplaza el valor de un píxel por la mediana de los niveles de intensidad en la ventana de filtrado de ese píxel:

$$x_i = mediana_{x_j \in W}\{x_j\}, \tag{2.9}$$

El valor del píxel x_i es incluido en el cálculo de la mediana. Este filtro es popular porque provee excelente capacidad en la reducción de ruido, particularmente es efectivo en imágenes con presencia de ruido de impulsivo bipolar ("sal y pimienta").

En (Kang y Wang 2009) se diseña un filtro de mediana direccional basado en el argumento fuzzy para la eliminación de ruido, así como en (Camarena y col. 2010b; Toh e Isa 2010; Imrul y Dey 2012) utilizan la mediana para eliminar el ruido impulsivo.

Filtro Difusivo no lineal

Una clase de métodos de eliminación de ruido se basan en el uso de ecuaciones de difusión no lineal (Guidotti y Lambers 2009; Reza 2011; Puvanathasan y Bizheva 2009; Mendrik 2009; Huang, Ng e Y.W. 2009; González, Hyoung y Yeol 2011; Liu y col. 2012), que generalmente aparecen asociados a un problema variacional y se puede obtener a partir de la minimización de funcionales apropiados. La elección de un determinado funcional depende del objetivo específico de interés. Por ejemplo, varios filtros de difusión, adecuados para las imágenes médicas (Keeling 2003), han sido obtenidos a partir de la minimización de funcionales apropiados. Consideremos la funcional (Weickert 2001),

$$J(u,\beta,\mu,\epsilon)=\int_{\Omega}\left(\sqrt{\beta^2+\left\|\vec{\nabla}u\right\|^2}+\frac{\mu}{2}\left(u-I_r\right)^2+\frac{\epsilon}{2}\left(\vec{\nabla}u\right)^2\right)d\vec{x}, \tag{2.10}$$

donde I_r es la imagen observada (con ruido), u es la imagen filtrada que se pretende reconstruir a partir de I_r, μ y ϵ son constantes y Ω es una región convexa de $\Re^2$ que constituyen el espacio de apoyo de la superficie $u(x,y)$, que representa la imagen. El primer término en la funcional para $\beta=1$ representa el área de la superficie de la imagen (Vogel y Oman 1996), el segundo término es la distancia entre la imagen observada y la solución deseada u y el tercer término controla la regularidad de la solución.

El proceso de eliminación de ruido corresponde al problema de minimización (Rudin, Osher y Fatemi 1992; Vogel y Oman 1996) (variación total):

$$\min_{u} J(u,\beta,\mu,\epsilon) \quad \text{sujeto a} \quad \frac{\int_{\Omega}(u-I_r)^2\,d\vec{x}}{\int_{\Omega}d\vec{x}}=\sigma^2 \tag{2.11}$$

La solución del problema es la imagen I_f que minimiza el funcional $J(u,\beta,\mu,\epsilon)$ que satisface la restricción de arriba. La condición significa que el "error" entre la imagen original y la ruidosa, debe ser igual a σ, donde σ es la desviación estándar del ruido presente en la imagen. Es importante disponer de una buena estimación de σ para minimizar la ecuación (2.10).

Para estimar el ruido presente en la imagen, se ha usado una estimación robusta propuesta por Donoho en (Donoho 1995) basada en la estimación wavelet discreta. De acuerdo a (Donoho 1995), la des-

viación estándar del ruido presente en la imagen puede ser estimada por

$$\sigma = \frac{median(\mid D_{ij} \mid)}{0{,}6745}, \tag{2.12}$$

donde D_{ij} son los coeficientes diagonales de la transformada wavelet. En esta tesis se ha utilizado la wavelet de *Dauvechies* de orden 25.

La solución del problema de minimización lleva a una discretización del tiempo y, por lo tanto, a una solución iterativa del problema. Para la discretización del tiempo se utiliza un esquema semi-implícito y para resolver las ecuaciones la alternativa de la división del operador aditivo (AOS) (Weickert, Ter y Viergever 1998; Weickert 2001). A continuación se presenta el proceso.

Discretización del problema

Partiremos de la formulación estándar de un proceso de difusión que tiene la siguiente estructura:

$$g(|\bigtriangledown u|) = \frac{\alpha}{\sqrt{\beta^2 + \|\bigtriangledown u\|^2}} + \epsilon, \tag{2.13}$$

donde α, β y ϵ son parámetros positivos.

El proceso de filtrado iterativo produce varias imágenes $u(x, y, t_k)$ $(k = 0, 1, 2, ...t_0 = 0)$ que son versiones filtradas de la imagen inicial observada $I_r(x, y)$ en valores de tiempo discretos t_k. El punto inicial del proceso difusivo es la imagen observada $u(x, y, 0) = I_r(x, y)$ y para valores mayores se obtienen representaciones aproximadas de $I_r(x, y)$. Para obtener la solución de la ecuación de difusión inicial se va a utilizar un método iterativo basado en las ecuaciones dinámicas que puede ser visto como un método de gradiente (Weickert, Ter y Viergever 1998):

$$\frac{\partial u}{\partial t} = \overrightarrow{\bigtriangledown}(\frac{\alpha \overrightarrow{\bigtriangledown} u}{\sqrt{\beta^2 + \left\| \overrightarrow{\bigtriangledown} u \right\|^2}}) + \epsilon \bigtriangledown^2 u. \tag{2.14}$$

En la discretización del problema se considera que la imagen define una malla estructurada dada por los diferentes píxeles que constituyen la imagen digital y que la longitud espacial de malla es 1.

La ecuación 2.14 en geometría unidimensional, siendo g la función de difusividad dada por 2.13 se representa como:

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}(g(\overrightarrow{\bigtriangledown} u))\frac{\partial u}{\partial x}. \tag{2.15}$$

Para la discretización espacial se ha seleccionado la siguiente discretización del operador de difusión para cada nodo espacial i de la malla que define la imagen, donde g_i es la función de difusividad en el nodo i,

$$\begin{aligned}\left(\frac{\partial}{\partial x}\left(g\left(\frac{\partial u}{\partial x}\right)\right)\right)_i \approx \left(g_{i+1/2}\left(u_{i+1}-u_i\right) - g_{i-1/2}\left(u_i - u_{i-1}\right)\right) = \\ = \frac{1}{2}\left(g_i + g_{i+1}\right)\left(u_{i+1}-u_i\right) - \frac{1}{2}\left(g_i + g_{i-1}\right)\left(u_i - u_{i-1}\right)\end{aligned} \tag{2.16}$$

Para la discretización del tiempo, se usa el esquema semi-implícito (Weickert, Ter y Viergever 1998; Weickert 2001), en que la variable depende de valores anteriores:

$$u^{n+1} = u_i^n + \frac{\Delta t}{2}\left(\left(g_i^n + g_{i+1}^n\right)\left(u_{i+1}^{n+1} - u_i^{n+1}\right) - \left(g_i^n + g_{i-1}^n\right)\left(u_i^{n+1} - u_{i-1}^{n+1}\right)\right) \tag{2.17}$$

que se puede reescribir de la forma siguiente, siendo $g_i^n = g\left(\left|\overrightarrow{\bigtriangledown} u\right|_i^n\right)$ en el instante n:

$$\begin{aligned}u_i^{n+1} - \Delta t\left(\frac{1}{2}\left(g_i^n + g_{i-1}^n\right)u_{i-1}^{n+1} - \frac{1}{2}\left(g_{i-1}^n + 2g_i^n + g_{i+1}^n\right)u_i^{n+1} + \frac{1}{2}\left(g_i^n + g_{i+1}^n\right)u_{i+1}^{n+1}\right) = \\ = u_i^n\end{aligned} \tag{2.18}$$

Considerando todos los nodos de la malla y las condiciones de contorno, se puede escribir la ecuación (2.18) de la forma:

$$(I - \Delta t \cdot A_x(u^n))u^{n+1} = u^n + \Delta t \cdot I_r. \tag{2.19}$$

donde $A_x(u^n)$ tiene la siguiente estructura:

$$\begin{bmatrix} b(1) & c(1) & & & & \\ a(2) & b(2) & c(2) & & & \\ & a(2) & . & . & & \\ & & . & . & . & \\ & & & . & . & c(m-1) \\ & & & & a(m) & b(m) \end{bmatrix}$$

donde,

$$\begin{aligned} &a(1) = 0, a(i) = \frac{1}{2}\left(g_i^n + g_{i-1}^n\right), i = 2, \ldots m \\ &b(i) = -\frac{1}{2}\left(g_{i-1}^n + 2g_i^n + g_{i+1}^n\right), i = 1, \ldots m \\ &c(i) = \frac{1}{2}\left(g_i^n + g_{i+1}^n\right), i = 1, \ldots m-1, c(m) = 0. \end{aligned} \tag{2.20}$$

donde m es el número de píxeles de la imagen. Se trata de una matriz tridiagonal y el sistema de ecuaciones asociado puede resolverse de forma muy eficiente (Teukolsky y Betterling 1992; Golub y Van Loan 1996).

Discretizando la ecuación 2.14 en geometría 2D, siguiendo el mismo razonamiento del caso 1D, se llega a la siguiente ecuación matricial:

$$(I - \Delta t \cdot A_x(u^n) - \Delta t \cdot A_y(u^n)) \cdot u^{n+1} = u^n + \Delta t \cdot I_r. \tag{2.21}$$

El hecho de invertir este sistema de ecuaciones desde el punto de vista computacional es costoso, por lo que se usa el esquema AOS (Weickert, Ter y Viergever 1998; Weickert 2001). Este esquema de discretización es más rápido que los esquemas convencionales y es estable para todos los valores de paso de tiempo. Una vez que el esquema se aplica, se obtiene la siguiente ecuación:

$$u^{n+1} = \frac{1}{2}((I - 2\cdot\Delta t\cdot A_x(u^n))^{-1}\cdot(u^n + \Delta t\cdot I_r) + (I - 2\cdot\Delta t\cdot A_y(u^n))^{-1}\cdot(u^n + \Delta t\cdot I_r)) \tag{2.22}$$

denotando: $\hat{A}_x = (I - 2\cdot\Delta t\cdot A_x(u^n))$, $\hat{A}_y = (I - 2\cdot\Delta t\cdot A_y(u^n))$ y $\hat{f} = (u^n + \Delta t \ \cdot I_r)$ la ecuación 2.22 se redefine como:

$$u^{n+1} = \frac{1}{2}\left(\hat{A}_x^{\ -1}\cdot\hat{f} + \hat{A}_y^{\ -1}\cdot\hat{f}\right). \tag{2.23}$$

En el proceso iterativo dado por la ecuación 2.23, en cada paso de tiempo hay que resolver dos sistemas de ecuaciones tridiagonales, representados por las matrices $\hat{A}_x$ y $\hat{A}_x$. Mediante dicho esquema en cada paso de tiempo, se preserva el valor medio del nivel de gris y converge a un estado constante (Weickert, Ter y Viergever 1998).

Para la aplicación del esquema AOS cabe considerar el valor del parámetro Δt de paso de tiempo, pues un valor grande puede reducir el número de iteraciones del proceso. En la práctica el valor del parámetro Δt debe ser elegido como compromiso entre la exactitud del cálculo y la eficiencia.

Los valores de los parámetros de la ecuación de difusión que se han tomado para llevar a cabo el proceso de restauración en la presente tesis son: el valor de paso de tiempo Δt=0.05 y el paso de espacio 1. Por ello, el conjunto de imágenes filtradas que se van obteniendo $u(x, y, t_k)$ son las imágenes filtradas en los instantes de tiempo discretos t_k=$k\Delta t$ (k=0,1,2, ...). Para el resto de parámetros en la ecuación de difusión (2.10), son los siguientes: $\epsilon = \beta = 1$, μ=1 como valor de partida y varía en cada iteración para mantener la varianza. La elección de estos valores para el conjunto de parámetros de la ecuación de difusión asegura una buena actuación del filtro difusivo para evaluar el criterio de parada del filtro basado en la función de neguentropía (Mayo 2007).

Criterio de parada

Sea $f(x, y)$ la función que representa los niveles de gris de la imagen observada, se considera Ω una región convexa de R^2 entonces la imagen observada $f(x, y)$ es una representación y puede considerarse la suma de una imagen limpia de ruido y una señal de ruido, tal como:

$$f(x, y) = f'(x, y) + n(x, y) \tag{2.24}$$

siendo f' la imagen limpia de ruido que se pretende reconstruir a partir de f y n un ruido aditivo gaussiano, de media cero $n \sim N(0, \sigma^2)$.

En esta tesis se ha utilizado el tiempo de parada del proceso iterativo descrito por la ecuación 2.23 propuesto por Mrázek y Navara (Mrázek y Navara 2003). La idea se basa en asumir que la imagen ideal libre de ruido f' y el ruido n no están correlacionados, de tal manera que la imagen $u(t)$ será cercana a f' si la covarianza de la estimación del ruido

$u(0) - u(t)$ y de la imagen $u(t)$ es lo más pequeña posible. Su criterio de tiempo de parada está basado en la minimización del coeficiente de correlación siguiente:

$$corr(u(0) - u(t), u(t)) = \frac{cov(u(0) - u(t), u(t))}{\sqrt{var(u(0) - u(t)) \cdot var(u(t))}} \tag{2.25}$$

La varianza relativa de una imagen $u(x, y, t)$ de un proceso difusivo (para un tiempo fijo t, las variables son (x, y)), se define como:

$$r(u(t)) = \frac{var(u(t))}{var(u(0))} \; t > 0 \tag{2.26}$$

Así, la ecuación del tiempo de parada de Mrázek y Navara T_{MN} es denotado como:

$$T_{MN} = argmin_t \, |corr(u(0) - u(t), u(t))| \tag{2.27}$$

2.3.2 Proceso de detección

La mayoría de métodos para identificar los píxeles corruptos de una imagen se basan en el concepto de los k-vecinos (k-neighbors) de p ($N_k(p)$) (González y Woods 2008) consiste en calcular la distancia entre un píxel y sus vecinos, y etiquetarlo como erróneo cuando esa distancia supere o sea inferior a un cierto umbral para la mayoría de vecinos con que se ha comparado.

Las medidas de distancia más comunes entre dos píxeles p y q, son: la distancia Euclídea, *city-block* y *chessboard*, (González y Woods 2008). Estas medidas pueden aplicarse para conocer la distancia en intensidad que hay entre el píxel (x_i) y el píxel (x_j). En esta tesis se va a referir a la distancia en términos de intensidad de dos píxeles. Otras métricas se basan en el ángulo que pueden formar dos píxeles, la que se denomina ángulo del coseno. Una de las distancias muy utilizadas actualmente, es la métrica o distancia fuzzy. A continuación se explican las métricas que a lo largo de esta tesis se utilizan.

Distancia euclídea

La distancia euclídea entre dos píxeles x_i y x_j para imágenes a color, se define como

$$L(x_i, x_j) = \|(x_i - x_j)\|_2 . \tag{2.28}$$

Y para imágenes cuyo píxel contiene un solo canal, se define con la siguiente ecuación:

$$L(x_i, x_j) = |x_i - x_j| . \tag{2.29}$$

Métricas fuzzy

Una métrica fuzzy estacionaria M sobre X (Gregori y S. 2004; George y Veeramani 1994; George y Veeramani 1997; Gregori y Romaguera 2000) es un conjunto fuzzy de $X \times X$ satisfaciendo las siguientes condiciones para todo $x, y, z \in X$:

(FM1) $M(x, y)>0$

(FM2) $M(x, y)=1$ si y únicamente si $x = y$

(FM3) $M(x, y) = M(y, x)$

(FM4) $M(x, z)\geq M(x, y) * M(y, z)$

donde $*$ es una continuidad de $t-norma$. $M(x, y)$ representa el grado de cercanía de x y y y de acuerdo a (FM2), $M(x, y)$ es cercano a 0 cuando x esta lejos de y.

En (López 2010; Camarena y col. 2010b) se presenta una gran variedad de métricas fuzzy. En esta tesis se han utilizado dos de ellas que se presentan a continuación.

La distancia fuzzy entre los vectores RGB de dos píxeles x_i y x_j, está dada por la función:

$$M(x_i, x_j) = \prod_{l=1}^{3} \frac{min\{x_i(l), x_j(l)\} + k}{max\{x_i(l), x_j(l)\} + k}, \tag{2.30}$$

donde (x_i (1), x_i (2), x_i (3)) es el vector de color para el píxel x_i en RGB respectivamente, x_j son los píxeles vecinos de x_i y $k > 0$.

La métrica fuzzy M entre el píxel x_i y x_j para imágenes en escala de grises es dado por la siguiente función:

$$M(x_i, x_j) = \frac{min\{x_i, x_j\} + k}{max\{x_i, x_j\} + k}, \tag{2.31}$$

Otra de las métricas fuzzy es la que se obtiene por la división del valor de k sobre la suma de k y la norma euclídea (de la diferencia de los valores de los píxeles x_i y x_j). La función es,

$$G(x_i, x_j) = \frac{k}{k + \|x_i - x_j\|_2}, \tag{2.32}$$

donde k >0. Esta ecuación es el patrón de la métrica fuzzy cuando se trabaja en imágenes con formato RGB. Si se aplica esta métrica fuzzy a imágenes en escala de grises, entonces la ecuación 2.32 se transforma en:

$$G(x_i, x_j) = \frac{k}{k + |x_i - x_j|}. \tag{2.33}$$

Métrica del coseno del ángulo

La distancia del ángulo del coseno entre dos píxeles x_i y x_j, es denotado como:

$$C(x_i, x_j) = \frac{x_i^t \cdot x_j}{\left(\|x_i\|_2 \cdot \|x_j\|_2\right)}. \tag{2.34}$$

donde $x_i \neq 0$ y $x_j \neq 0$. En la práctica, se considera el rango de los píxeles de [1,256] en vez de [0,255] para evitar la división por cero.

Peer group

El concepto de *peer group*(Kenney y col. 2001; Smolka 2005), se ha usado para la detección y supresión de ruido impulsivo en imágenes en escala de grises y a color con buenos resultados en calidad. Para una imagen I, el *peer group* $\mathcal{P}(x_i, d)$ asociado con un píxel x_i, consiste de los x_j píxeles en una ventana (W) de tamaño $n \times n$, centrado en x_i que están más cercanos en intensidad a $I(x_i)$ medida a través de una distancia d. En otras palabras, los k vecinos son todos los vecinos de un píxel x_i en W, y los x_j son los vecinos que están más cercanos en intensidad al píxel x_i.

A partir de los x_j píxeles del *peer group*, se puede determinar si x_i es corrupto o no corrupto (Camarena y col. 2010a).

Para calcular la distancia (cercanía en intensidad) entre los píxeles x_i y x_j, se puede utilizar cualquiera de las medidas mencionadas anteriormente.

La representación del $\mathcal{P}(x_i, d)$ con la métrica fuzzy M y G, se denota por el conjunto:

$$\mathcal{P}(x_i, d) = \{x_j \in W : M(x_i, x_j) \geq d\}. \tag{2.35}$$

$$\mathcal{P}(x_i, d) = \{x_j \in W : G(x_i, x_j) \geq d\}. \tag{2.36}$$

donde d es un umbral de distancia ($0 < d \leq 1$) para decidir si los píxeles están cerca en intensidad y por lo tanto forman parte del *Peer group.*

Los píxeles que pertenecen al conjunto *peer group* y la métrica euclídea, son todos los píxeles que satisfacen la ecuación:

$$\mathcal{P}(x_i, d) = \{x_j \in W : L(x_i, x_j) \leq d\}, \tag{2.37}$$

donde $d > 0$. Conjunto de píxeles cuya norma euclidiana no excede a d.

El *peer group* con la distancia del coseno del ángulo se define como

$$\mathcal{P}(x_i, d) = \{x_j \in W : C(x_i, x_j) \geq d\} \tag{2.38}$$

donde $0 < d \leq 1$. En este caso, el píxel vecino x_j pertenece al conjunto del *peer group* si la distancia de él hacia el píxel central x_i es mayor o igual al valor d.

A lo largo del documento se nombrará como PGMF, PGME y PGMC para denotar al *peer group* con la métrica fuzzy, *peer group* con la métrica euclídea y el *peer group* con el coseno del ángulo, respectivamente.

2.4 Métricas de calidad

Con la finalidad de comprobar la eficiencia de los filtros, el punto de partida del proceso es una imagen original (I_o) sin ruido a la que se le añade la cantidad y tipo de ruido (modelo de ruido, explicado en la sección anterior). Se sigue el proceso de filtrado presentado en la

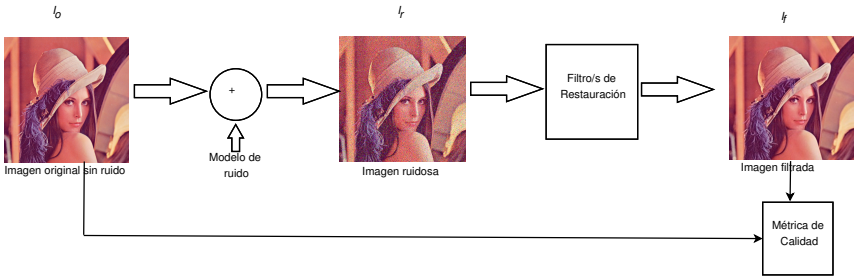


Figura 2.4: Comparación de las imágenes I_o y I_f a través de las métricas de calidad.

figura 2.3 y después se compara la imagen I_o con la I_f a través de métricas de calidad (figura 2.4).

El objetivo de la restauración es obtener una imagen estimada que sea lo más cercana posible a la imagen original de entrada sin ruido.

Se compara la similitud entre la imagen de salida (imagen filtrada) y la imagen original libre de ruido.

Las medidas de calidad comúnmente usadas son: PSNR (*Peak Signal-to-Noise Ratio*), MSE (*Mean Square Error*), MAE (*Mean Absolute Error*) y NCD (*Normalized Color Difference*) (Smolka 2005). Estas medidas se utilizan en esta tesis para evaluar la cantidad de ruido eliminado y la conservación de los detalles de la imagen. Para medir la capacidad del filtro para reducir el ruido, se utilizó PSNR; MAE es la función utilizada para evaluar la preservación de los detalles y NCD se utilizó para medir la percepción humana. El valor *psnr* es mejor cuanto mayor es. El valor *mse* y *mae* es mejor cuando es menor.

Para definir PSNR se requiere calcular el MSE(*Mean Square Error*) , el cual para dos imágenes: Imagen filtrada I_f e Imagen de referencia (Imagen original I_o) se define como:

$$MSE = \frac{1}{MN} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (|I_o(i,j) - I_f(i,j)|)^2, \tag{2.39}$$

donde MN es el número de píxeles de la imagen, $I_o(i,j)$ y $I_f(i,j)$ denota el (i,j) i-ésimo componente del canal del píxel de la imagen original y ruidosa.

Así, el PSNR se define como:

$$PSNR = 20log_{10}\left(\frac{MAX_{I_o}}{\sqrt{MSE}}\right), \tag{2.40}$$

donde MAX_{I_o} es el máximo valor posible del píxel en la imagen (255).

El error absoluto medio está dado por,

$$MAE = \frac{1}{MN}\sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\left|I_o\left(i,j\right) - I_f\left(i,j\right)\right|. \tag{2.41}$$

Las diferencias entre los colores del espacio se perciben uniforme y no se corresponden con las diferencias de color percibido por los seres humanos (Smolka 2005), por eso se analizan los errores de restauración usando espacios de color uniforme. Para ello se utiliza el NCD que se define como:

$$NCD = \frac{NM \triangle E}{\sum_{i=0}^{NM-1}\sqrt{(L^*_{I_{o_i}})^2 + (U^*_{I_{o_i}})^2 + (V^*_{I_{o_i}})^2}}, \tag{2.42}$$

donde L^* representa el valor de luminiscencia (brillo) y U^*,V^* los valores de crominiscencia (color) correspondientes a la imagen original I_{0_i}. El espacio de color CIE LUV, $\triangle E$ se define como:

$$= \frac{1}{N}\sum_{i=0}^{NM-1}\sqrt{(L^*_{I_{o_i}} - L^*_{I_{f_i}})^2 + (U^*_{I_{o_i}} - U^*_{I_{f_i}})^2 + (V^*_{I_{o_i}} - V^*_{I_{f_i}})^2}, \tag{2.43}$$

donde I_{f_i} es la imagen filtrada.

2.5 Estado del arte de eliminación de diferentes tipos de ruido

Los trabajos más recientes en el campo de procesamiento de imágenes podríamos catalogarlos en innovaciones en el proceso de detección (utilización de *peer group*, métricas fuzzy, distancia euclídea), en el proceso de filtrado (métodos no lineales, anisotrópicos, de variación total) y en la paralelización y utilización de GPUs de cualquiera de los métodos.

Se han presentado varios enfoques basados en el concepto de *peer group* con la finalidad de eliminar el ruido impulsivo con una detección fiable de los impulsos. La detección de los píxeles dañados propuesto en (Smolka 2010) se realiza usando el discriminante lineal de Fisher, dividiendo los píxeles en dos clases: un *peer group* de píxeles similares a la mediana vectorial y un conjunto de píxeles que constan de valores atípicos inyectados en la imagen por el proceso de ruido. El concepto de *peer group* se redefine en (Camarena y col. 2008) por medio de ciertas métricas fuzzy para la detección rápida de los píxeles ruidosos tomando las ventajas de las propiedades de la métrica fuzzy, dando origen a un filtro *swithing* entre el AMF y la operación de identidad.

El detector fuzzy (Camarena y col. 2010b) para detectar los impulsos, se basa en las diferencias de rangos ordenados (ROD) y estadísticas. El concepto de *peer group fuzzy* (Morillas, Gregori y Hervás 2009) elimina el ruido impulsivo y gaussiano que consiste en extender el concepto de *peer group* en el campo *fuzzy*. Un filtro fuzzy propuesto en (Mélange, Nachtegael y Kerre 2010) elimina el ruido impulsivo aleatorio en video y trabaja con diferentes pasos sucesivos de filtrado en el que los píxeles ruidosos se detectan a través reglas fuzzy. Se obtiene un equilibrio entre la preservación de los detalle y la eliminación de ruido.

El algoritmo presentado en (Camarena y col. 2010a) tiene un paso de refinamiento para los píxeles que fueron detectados como no diagnosticados respecto a (Camarena y col. 2008; Morillas, Gregori y Peris-Fajarnés 2008).

El ruido gaussiano se puede tratar por medio del uso de ecuaciones de difusión no lineal (Guidotti y Lambers 2009; Reza 2011; Liu y col. 2012). El filtro de Difusión no lineal se aplica en imágenes obtenidas mediante Rayos X (González, Hyoung y Yeol 2011) que elimina el ruido y a la vez preserva los bordes que son necesarios para el diagnóstico médico. La reducción de ruido en CT con difusión híbrida anisotrópica se presenta en (Mendrik 2009).

El ruido speckle ha sido tratado recientemente mediante un filtrado de difusión anisotrópica (Puvanathasan y Bizheva 2009; Jin y Yang 2010; Bioucas y Figueiredo 2010). Alguna recopilación y análisis de rendimiento de las técnica más utilizadas de suavizado o supresión del ruido speckle en imágenes de ultrasonido se presenta en (Mateoa y Fernández 2009; Sudha, Suresh y Sukanesh 2009). El método basado en *Undecimated Doble Densidad Wavelet Transform* (UDDWT) (Gnanadurai y col. 2009) se utiliza para eliminar el ruido speckle en imágenes SAR.

Se han propuesto diversas estrategias con el fin de reducir la dosis en radiografía y actualmente son pocos los trabajos en donde se realiza un análisis de la reducción de la exposición (mAs) en los pacientes (ICRP 1993). Trabajos donde el objetivo es la reducción de ruido en imágenes médicas son (Nair y Reji 2011; Padma y Sukanesh 2010) . En (Silva y col. 2010) se presenta un nuevo algoritmo de reconstrucción de imágenes de CT con adaptación estadística de reconstrucción iterativa (ASIR) para reducir la dosis de la radiación en las CT de cuerpo. ASIR también se utiliza para estimar la radiación en la angiografía coronaria CT (Leipsic y col. 2010; Singh y col. 2009) donde se presenta la reducción de la dosis con protocolos pediátricos de CT a la medida de las indicaciones clínicas, el peso del paciente, y el número de estudios previos. En (Juste y col. 2008) se presenta un análisis basado en la reducción de ruido simulado por computadora.

Las GPUs ya no sólo son utilizadas con el propósito inicial que se tenía (videojuegos), si no que se utilizan en aplicaciones en donde se consume mucho tiempo de procesamiento en forma normal y por lo tanto se requiere acelerar los procesos.

En (Harding 2008) usan *Cartesian Genetic Programming* para generar programas de sombreado que implementan las operaciones del filtrado de imágenes. Usando la GPU pueden aplicar rápidamente estos programas a cada píxel en una imagen y evaluar el rendimiento de un filtro dado. El filtro elimina con éxito el ruido de las imágenes produciendo mejor calidad que con un filtro estándar de media.

El ruido impulsivo en secuencias de video de imagen multicanal, se puede suprimir a través del método presentado en (Kravchenko, Ponomaryov y Pustovoit 2010) para reconocer los movimientos en los *frames* vecinos de una secuencia y para reconstruir los contornos y detalles finos. La propuesta desarrollada está basada en la teoría del conjunto fuzzy, en la teoría de la desviación angular de los píxeles de la imagen en los *frames* vecinos pertenecientes a un multicanal de secuencia de video cuando se forman los *frames* de la imagen filtrada.

Para mejorar el rendimiento y eliminar la presencia de ruido en los sistemas de comprensión de video en (Guo y col. 2010) adoptan un filtro de eliminación de ruido como un módulo de pre-procesamiento para la codificación de video, o como un módulo de post-procesamiento para la decodificación de video, pero la complejidad introducida por la eliminación de ruido puede ser muy grande. Se presenta un filtro temporal recursivo LMMSE (*Linear Minimum Mean Squared Error*) para eliminar el ruido en video.

No hay muchos trabajos que reporten aceleración de algoritmos en GPUs para eliminar el ruido en una imagen. Algunos de ellos presentados en (Sánchez y col. 2011c; Sánchez y col. 2011a; Sánchez, Vidal y Bataller 2012; Sánchez, Vidal y Bataller 2010; Sánchez, Vidal y Bataller 2011) son algunos ejemplos de filtrado para eliminar el ruido impulsivo mediante el uso de GPU que utilizan el *peer group* y métricas fuzzy y/o euclídea. En estos trabajos se reporta un equilibrio entre calidad y reducción del tiempo computacional.

En esta tesis se ha optado por la paralelización de los algoritmos que utilizan el concepto de *peer group* con métricas fuzzy o euclídea (conceptos definidos en el siguiente capítulo) para detectar los píxeles erróneos, debido a que últimamente se han presentado varios estudios con resultados satisfactorios en calidad (Camarena y col. 2010b; Smolka 2010; Camarena y col. 2008; Camarena y col. 2010a; Morillas, Gregori y Hervás 2009). Se define otra etapa para eliminar los píxeles corruptos en la que se utiliza el concepto de sustitución del píxel ruidoso por los filtros Vectoriales de Mediana o la Media Aritmética (González y Woods 2008), de esta manera se asegura que la imagen filtrada tenga una buena calidad final. Para reducir el coste computacional se analizan, diseñan e implementan algoritmos secuenciales y paralelos para el filtrado de imágenes utilizando arquitecturas *many-core* y *multicore*.

Capítulo 3

Hardware y software de computación de altas prestaciones

3.1 Introducción

En los últimos años las GPUs incluidas en tarjetas gráficas han conseguido una mejora de velocidad computacional muy importante, ofreciendo un alto nivel de procesamiento en paralelo a un precio muy competitivo. Las empresas que lideran actualmente el mercado de GPUs son NVIDIA y ATI/AMD. De ellas, NVIDIA es la líder actual en el campo de la GPGPU (*General Purpose Graphics Processing Unit*) debido a su bajo coste y al uso de la plataforma CUDA.

La CPU se caracteriza por tener memoria caché muy rápida, obtener un alto rendimiento sobre un único hilo de ejecución y es óptimo en paralelismo de tareas. La GPU tiene una DRAM muy rápida, el hardware está dedicado para cálculos matemáticos, se obtiene alto rendimiento ejecutando tareas paralelas y especialmente es óptimo en paralelismo de datos. En esta tesis se utilizó CPU (arquitecturas *multicore*) y GPUs (*many-core*), con la finalidad de observar el rendimiento de los algoritmos en diferente hardware. Para obtener mejor rendimiento se realizaron diferentes optimizaciones en CUDA con la finalidad de tomar ventajas del hardware disponible.

Este capítulo esta dividido en tres partes. La sección 3.2 presenta el equipo de altas prestaciones que se empleó en esta tesis para realizar las pruebas. La sección 3.2 presenta CUDA y OpenMP como las dos

herramientas que se han utilizado para programar los algoritmos paralelos. Las herramientas para medir los recursos hardware utilizados se exponen en la sección 3.2.

3.2 Características del hardware utilizado

Los tres equipos de altas prestaciones que se utilizaron en esta tesis, se describen a continuación.

- DSICMAC1 (UPV). Apple Mac Pro Xeon/2.66 GHz, Quad-Core 3500 y con una memoria RAM de 8GB. Contiene 16 cores CPU intel y cuatro GPUs NVIDIA GeForce GT 120. Cada tarjeta gráfica tiene 32 cores CUDA (arquitectura *many-core* dotada de 32 cores). La versión del driver instalado en las tarjetas es 4.0.5.
- Cluster de Computación- IUII- (EULER, UA). El cluster HPC está formado por 26 nodos de cálculo: 2 procesadores Intel Xeon X5660 hexacore a 2.8 GHz y con 48 GB de memoria RAM. Uno de los nodos está equipado adicionalmente con 3 módulos GPU HP Tesla M2050 con 3GB de memoria cada uno y adicionalmente uno de los nodos cuenta con un módulo GPU NVidia Tesla 2070 con 6GB de memoria. Cada tarjeta M2050 tiene 448 cores CUDA (arquitectura *many-core* dotada de 448 cores, procesadores streaming).
- Equipo de altas prestaciones GPU9800 (UPV). Este equipo de cómputo cuenta con dos tarjetas gráficas GeForce 9800 GX2. Cada tarjeta tiene 2x128 cores CUDA (arquitectura *many-core*, 128 cores en cada tarjeta).

En la tabla 3.1 se hace una comparativa de las características principales de las GPUs.

La arquitectura Tesla tiene una memoria de video superior a las otras dos. En general los Tesla están diseñados particularmente para computación científica por lo que no cuenta con salida de video.

El ancho de banda es uno de los factores importantes a tomar en cuenta para un buen rendimiento en las aplicaciones. La figura 3.1 muestra el ancho de banda y el tamaño de la memoria de las GPUs incorporadas en los equipos DSICMAC1, GPU9800, EULER.

Las GPUs no comparten memoria de video, por lo que el programador debe encargarse de transferir los datos entre las GPUs (en caso de que

Tabla 3.1: Características principales de las GPUs utilizadas

	GeForce GT 120	GeForce 9800 GX2	Tesla M2050
Total de memoria Global	512MB	512MB	3GB.
Memoria Compartida disponible por bloque(KB)	16	16	48
Número de registro por bloque de hilos	8192	8192	32768
Tamaño Warp en hilos	32	32	32
Memoria Pitch(MB)	256	256	2048
Máximo hilo por bloque	512	512	1024
Dimensión Máxima de hilos (bloques)	512 512 64	512 512 64	1024 1024 64
Dimensión Máxima de hilos (*grid*)	65535 65535 1	65535 65535 1	65535 65535 65535
Total de memoria constante(bytes)	65536	65536	65536
Capacidad de Cómputo	1.1	1.1	2.0
Velocidad de reloj(MHz)	1400	1512	1150
Alineación de Textura(bytes)	256	256	512
Número de Multiprocesadores	4	16	14
Tipo de operaciones	simple precisión	simple precisión	doble precisión

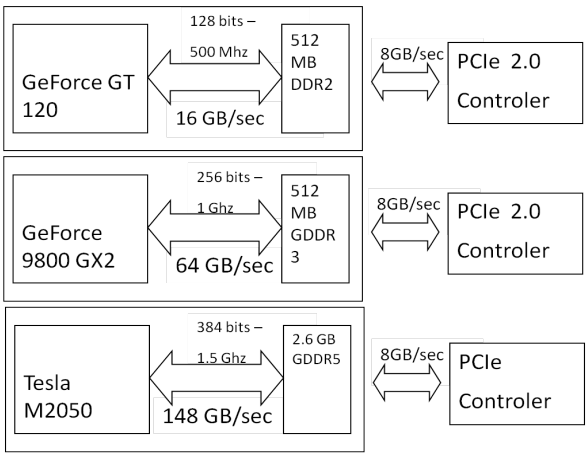


Figura 3.1: Ancho de banda y tamaño de la memoria de la GPU de tres tarjetas gráficas.

se requiera trabajar con más de una GPU) a través del bus PCI-express.

3.3 Herramientas software de entornos secuenciales y paralelos

Existen dos entornos de programación muy utilizados actualmente en las tarjetas gráficas que explotan las características y permiten obtener el máximo rendimiento de ellas, CUDA y OpenCL(Khronos Group 2012). Aunque OpenCL es el lenguaje de programación en GPGPU

para todo tipo de tarjetas gráficas, CUDA es la plataforma de programación más utilizada para las tarjetas NVIDIA.

Esta tesis se inclinó por la programación en el lenguaje CUDA debido a la sencillez de programación (tiene pocas extensiones al lenguaje C convencional) y con la que se obtiene un buen rendimiento.

El entorno de programación MATLAB se empleó para verificar los algoritmos de manera secuencial y obtener los parámetros óptimos, después estos algoritmos fueron ejecutados sobre entornos paralelos.

Para utilizar los cores y las GPUs en la implementación híbrida, se hizo uso de OpenMP.

3.3.1 CUDA

Modelo de ejecución CUDA

Las tarjetas de procesamiento gráfico, han experimentado en los últimos años un crecimiento espectacular en capacidad de cálculo paralelo debido a la gran demanda de aplicaciones de videojuegos. Aunque inicialmente ése fuera el principal destino de este hardware, en 2007 NVIDIA introdujo CUDA, una tecnología que permite utilizar estas unidades para realizar programas con cualquier otra finalidad de cálculo (GPGPU). Esta plataforma está diseñada a nivel de software y hardware para aprovechar la potencia de una GPU en aplicaciones de propósito general. A nivel software permite programar la GPU en lenguaje C con pocas extensiones SIMD(*Single Instruction, Multiple Data*) para lograr una ejecución eficiente y escalable. A nivel hardware habilita múltiples niveles de paralelismo (para múltiples cores y para múltiples hilos por core). Por ello, se ha popularizado su uso en la comunidad científica debido a la muy buena relación entre precio y potencia de cálculo que ofrece.

La GPU, denominada dispositivo (*device*) según la terminología de CUDA, contiene físicamente un conjunto de multiprocesadores N, cada uno dotado de M procesadores y su propia memoria de video DRAM (figura 3.2). Los multiprocesadores procesan los programas según el modelo SIMD, es decir, en cada ciclo de reloj un procesador del multiprocesador ejecuta la misma instrucción aplicada sobre datos diferentes, entendiendo que cada aplicación de esa instrucción la realiza un hilo (de ejecución) distinto.

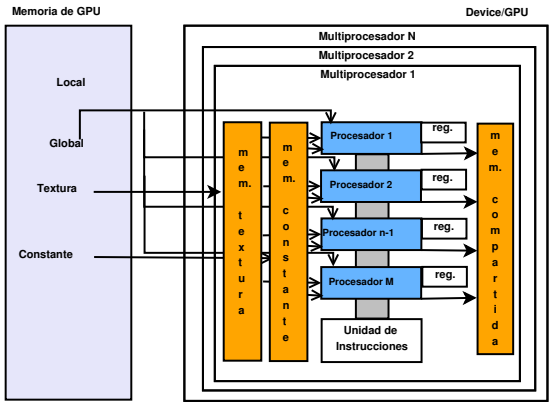


Figura 3.2: Modelo de hardware de CUDA

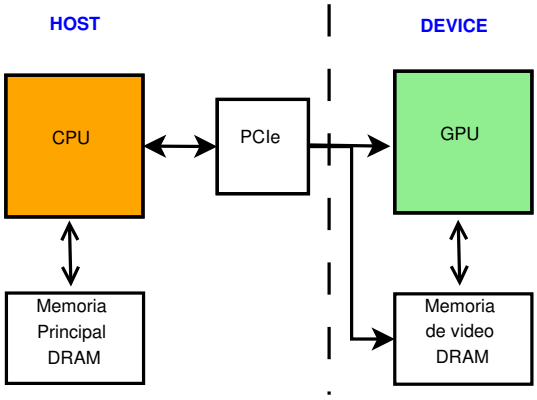


Figura 3.3: *Host* y *device* en CUDA

CUDA ejecuta un programa en un *device* que actúa como coprocesador de un *host* (CPU) como se muestra en la figura 3.3.

Espacios de memoria en CUDA

El *device* tiene varios tipos de memoria, incluyendo caché de diferentes niveles. La memoria física puede ser utilizada de diferentes formas, su uso principal es como memoria global compartida entre los multiprocesadores de la GPU. El acceso a los datos almacenados en la memoria global tiene mayor latencia que otros modos de acceso a los datos, debido a que no pasan por caché. La memoria global es el único visible a

todos los entes definidos por el programador (hilos, bloques, *kernels*, *grids*). Esta memoria permite que distintas áreas suyas puedan ser utilizadas de otras formas:

- Como memoria local. Cada hilo puede utilizar 16KB de memoria de la GPU de forma privativa.
- Como textura. Se trata de la posibilidad de bloquear una zona de la memoria de la GPU para ser utilizada sólo de lectura, de forma compartida, y optimizada para albergar estructuras (arreglos) de 1, 2 o 3 dimensiones. Una textura puede ser cualquier región de la memoria lineal o un arreglo CUDA.
- Como zona de lectura constante, compartida entre todos los hilos.

Internamente, cada multiprocesador tiene cuatro tipos de memorias (NVIDIA 2009):

- Un conjunto de registros de 32 bits por procesador, su acceso puede ser de lectura o escritura. Tiene baja latencia por estar cerca del procesador.
- Una caché de lectura y escritura de 16KB (memoria compartida), para optimizar el acceso a la memoria global compartida por todos los procesadores del multiprocesador. Tiene baja latencia y un ancho de banda elevado pero su tamaño es reducido; actúa como una caché y es gestionada por el usuario.
- Una caché constante de sólo lectura de 64KB, compartida por todos los procesadores y acelera la lectura de la memoria constante.
- Una caché de sólo lectura llamada caché de textura, compartida por todos los procesadores y acelera la lectura de la memoria de textura.

El ancho de banda de la memoria global se utiliza más eficientemente cuando los accesos a memoria simultáneo de hilos se realizan en la mitad de un *warp* (NVIDIA 2009), ya que pueden ser coalescentes en una simple transacción de memoria de 32, 64, o 128 bytes. Inicialmente, los datos están disponibles en la memoria global de GPU pero si existen problemas de coalescencia, se hace necesario considerar si sería recomendable copiar los datos de la memoria global a la memoria compartida de cada multiprocesador con el fin de procesarlos. Otra opción es el uso de texturas, si parte de la información es de sólo lectura y está organizada dimensionalmente en arreglos.

La función *cudaMallochost* permite asignación de espacios de memoria de gran tamaño con asignación de memoria lineal *page-locked*, esto facilita las transferencias de los datos entre CPU y GPU debido a que es accesible al *device*. Cuando se utiliza memoria *page-locked*, no hay necesidad de asignar cualquier *device* de memoria y explícitamente copiar datos entre el dispositivo y la memoria principal. Las transferencias de datos son implícitamente y se realiza cada vez que el *kernel* accede a la memoria asignada. Para un rendimiento máximo, estos accesos de memoria deben estar coalescentes como con accesos a la memoria global (véase la Sección 5.3.2.1). Si se asume que son y que la memoria asignada se lee o se escribe sólo una vez, utilizando asignación de memoria *page-locked* en lugar de copias explícitas entre el *device* y la memoria *host* se puede lograr buen rendimiento.

Con la función *cudaMallocPitch()* se reserva espacio en la memoria del dispositivo linealmente en 2D.

Programación de kernels

Según el modelo de programación CUDA, hay que distinguir entre el código que se ejecuta en la CPU ("*host* code", según su nomenclatura) del que se ejecuta en cada núcleo de la GPU ("*device* code"). En particular, la función (código de programa) que se ejecuta en la GPU recibe el nombre de "*kernel*". Un *kernel* se procesa en paralelo por un conjunto de hilos que ejecutará las instrucciones de ese *kernel* en una parte diferente de los datos en memoria.

El código de la CPU debe realizar los siguientes pasos para lanzar la ejecución de *kernels* en la GPU:

- Copiar los datos de la memoria del *host* a la memoria del *device*.
- Lanzar la ejecución del/los *kernel*(s), decidiendo la cantidad de hilos y su organización en bloques necesarios para el procesamiento.
- Esperar a que termine la ejecución del/los *kernel*(s) para transferir los datos de la memoria de la GPU a la memoria de la CPU.

Los *kernel*s se lanzan en *grid*s y solo un *kernel* se ejecuta en un momento dado en una GPU (no hay multiprocesamiento de *kernels*). Cuando el *kernel* finaliza, todos los recursos de la GPU se liberan y quedan disponibles íntegramente para el *kernel* siguiente. Un *kernel* se ejecuta como una malla o *grid* 1D o 2D de bloques de hilos 1D, 2D o 3D.

Bloques, hilos y grids

Dado que la GPU tiene varios multiprocesadores, es necesario agrupar los hilos en "bloques"para dejar clara la asignación de la ejecución de un bloque a un multiprocesador concreto de la GPU (cada bloque es mapeado en un multiprocesador). Esta característica permite sincronizar los hilos sólo si pertenecen al mismo bloque. Los bloques no pueden sincronizarse, esto hace que el propio hardware tenga la libertad para acomodar la ejecución de un bloque de hilos en cualquier multiprocesador en cualquier instante. El conjunto de bloques se denomina *grid* (matriz de bloques de hilos que ejecutan un *kernel*). Cuando un bloque de hilos termina y aún hay datos por analizar, se asigna un bloque nuevo al multiprocesador que está disponible.

Con la finalidad de obtener un buen rendimiento en las aplicaciones, uno de los factores a considerar es el número de hilos y bloques en cada uno de los *kernel* lanzados. Este factor depende del número máximo de hilos soportados por las tarjetas gráficas y el tamaño de los datos que se consideran.

Un hilo puede acceder a cualquier posición de memoria, puede leer y/o escribir en tantas posiciones de memoria como sea necesario y puede cargar datos en memoria compartida de forma cooperativa dentro de un bloque, es decir, los hilos pueden comunicarse entre ellos a través de la memoria compartida si pertenecen al mismo bloque (NVIDIA 2012). Los hilos se ejecutan en paralelo sobre los núcleos (cores o *stream processors*) de un multiprocesador.

Cada hilo puede: leer/escribir en registros, leer/escribir en memoria compartida, leer/escribir en memoria global, leer/escribir en memoria local, leer en memoria de constantes y leer en memoria de texturas.

Problemas en la programación CUDA

La plataforma CUDA es relativamente sencilla de utilizar y está bien documentada, pero el problema radica en que es difícil optimizar el rendimiento de una aplicación, debido a algunas restricciones del hardware y a los múltiples tipos de memorias que incluyen las tarjetas, debido a que están organizadas en varios niveles y con diferentes capacidades, con diverso patrón de acceso a los datos y otras limitaciones. Por ello, es necesario un estudio específico con la finalidad de identificar las optimizaciones de los recursos ofrecidos por CUDA y obtener el máximo rendimiento posible, decidiendo la cantidad de hilos y su organización en bloques, la mejor ubicación entre las diferentes memo-

rias disponibles, considerando los datos de entrada, los datos de salida y la forma de acceder a ellos y realizando oportunamente las copias necesarias.

Una de las principales cuestiones que se deben considerar para obtener programas eficientes es la coalescencia de los accesos a la memoria global. La memoria global se direcciona en desplazamientos de 16 o 32 bytes. Además es posible leer 4, 8 ó 16 bytes de la memoria global en una simple instrucción. Pero si una variable no se guarda justo a partir de una dirección de memoria múltiplo de 16, o su tamaño no es múltiplo de 4, hay que hacer más de un acceso para leerla, lo cual penaliza el rendimiento. Además, como en las GPUs hay una gran cantidad de hilos, este problema se agudiza, ya que incluso dos hilos pueden competir al acceder a una zona de memoria. Cuando esto ocurre se dice que los accesos no tienen coalescencia (NVIDIA 2009) (no están bien alineados) y hay que evitarlo a toda costa. La forma más elemental (aunque hay otras) es que el hilo número i acceda a la variable (o índice de un arreglo) número i, y que la dirección de acceso a ese punto sea múltiplo de 16.

3.3.2 OpenMP

El OpenMP *Application Program Interface* (API) soporta multiplataforma paralela de memoria compartida, utilizando C/C++ y Fortran para todas las arquitecturas. OpenMP es un modelo portable y escalable que ofrece programación paralela en memoria compartida, una simple y flexible interfaz para el desarrollo de aplicaciones paralelas en equipos de cómputo que van desde un simple desktop hasta supercomputadoras.

3.4 Herramientas para medir los recursos hardware utilizados

En esta sección se verán algunas herramientas utilizadas para evaluar la efectividad con la que un algoritmo usa los recursos de un sistema paralelo para eliminar el ruido.

Aceleración de ejecución (Speedup). Mide la ganancia de velocidad que se consigue con un algoritmo paralelo al resolver un problema con respecto a un algoritmo secuencial para el mismo problema (Al-

meida y col. 2008). La fórmula del speedup es:

$$S(n,p) = \frac{t(n)}{t(n,p)} \tag{3.1}$$

donde $t(n)$ es el tiempo correspondiente a la mejor implementación secuencial que resuelve el problema, y $t(n,p)$ es el tiempo correspondiente al algoritmo paralelo.

Eficiencia. Es la porción del tiempo que los procesadores se dedican a trabajo útil. Normaliza el valor del speedup dividiendo por el número de procesadores (Almeida y col. 2008):

$$E(n,p) = \frac{S(n,p)}{p} = \frac{t(n)}{pt(n,p)} \tag{3.2}$$

Tendrá un valor entre cero y uno, siendo lo ideal que sea igual a uno.

Escalabilidad. En un sistema paralelo interesa que las prestaciones se sigan manteniendo en cierta medida al aumentar el tamaño del sistema. En general, no se pueden seguir manteniendo las prestaciones para un tamaño de problema fijo y aumentando el número de procesadores, pero sí interesa que cuando aumenta el número de procesadores se sigan manteniendo las prestaciones al aumentar el tamaño del problema. En realidad, interesa mantener cierta eficiencia cuando se pretende resolver problemas de mayor tamaño en sistemas con más procesadores. Los sistemas físicos(máquinas) o lógicos(algoritmos) que cumplen esta propiedad se dice que son *escalables*.

Interesa que la arquitectura de un sistema físico mantenga sus propiedades aunque aumente el número de procesadores. Los sistemas de memoria compartida son en general menos escalables que los de memoria distribuida, pues para asegurar un acceso (uniforme o no uniforme) a los bloques de memoria, vistos como una única memoria común, se necesita una red de interconexión que se satura al aumentar el número de procesadores. Esto hace que el número máximo de procesadores de un sistema de memoria compartida sea de unas pocas centenas, mientras que en memoria distribuida se llega a tener varios miles. Es por esto que en los últimos años no aparecen multiprocesadores simétricos en la lista TOP500, siendo todos máquinas de memoria distribuida o clúster (Almeida y col. 2008).

Coste. El coste de un algoritmo paralelo representa el tiempo (el trabajo) realizado por todo el sistema en la resolución del problema. Es el tiempo de ejecución multiplicado por el número de procesadores

usados:

$$C(n,p) = pt(n,p) \tag{3.3}$$

En un algoritmo óptimo coincidiría con el tiempo secuencial, pero en general es mayor. A la diferencia entre el coste y el tiempo secuencial se le llama *función overhead*:

$$t_0(n,p) = C(n,p) - t(n) = pt(n,p) - t(n) \tag{3.4}$$

Cuanto mayor es el orden de su función overhead peor es el comportamiento de un algoritmo paralelo.

Tiempo de ejecución. En el caso de programas y algoritmos paralelos la ejecución se lleva a cabo en varios procesadores y puede que no todos ellos empiecen o acaben la ejecución en el mismo momento. Por tanto, el tiempo de ejecución es el que transcurre desde que empieza la ejecución el primero de los procesadores que se pone en marcha en el sistema hasta que acaba el último de los procesadores. En un sistema paralelo homogéneo donde se dispone de p procesadores con la misma potencia computacional, el tiempo dependerá del tamaño de la entrada n, y del tamaño del sistema, con lo que es función de dos parámetros, $t(n,p)$. El tiempo además dependerá de la forma en que los procesadores estén conectados entre sí (*topología*) y con los distintos módulos de memoria(memoria compartida, virtual compartida, distribuida, etc.) Se puede hacer también por conteo de instrucciones. El factor de que en el caso paralelo la ejecución no sea determinista, hacen que el modelado del tiempo de ejecución se complique en el caso paralelo mucho más que en el secuencial. Por lo tanto, es necesario simplificar el modelo del tiempo paralelo de forma que sea válido para amplios sistemas, y a la vez que el modelo refleje suficientemente bien el tiempo de ejecución real de manera que las decisiones tomadas al utilizar el modelo sean satisfactorias. Para simplificar el modelado se puede suponer que existen sincronizaciones (reales o ideales) en determinados puntos del algoritmo.

Causas de reducción de las prestaciones. Hay una serie de factores que producen reducción en las prestaciones que se podrían esperar de un programa paralelo. Algunas causas son:

- Competencia por la memoria.
- Código secuencial.
- Tiempo de creación de procesos.
- Computación extra.

- Comunicaciones
- Tiempo de sincronización
- Desequilibrios de carga

Capítulo 4

Descripción de los métodos y algoritmos secuenciales

4.1 Introducción

En este capítulo se explican en detalle los algoritmos secuenciales utilizados en esta tesis para el tratamiento del ruido impulsivo, uniforme, gaussiano y speckle. En la sección 4.2 se exponen los algoritmos para eliminar el ruido impulsivo y uniforme que hacen uso del *peer group* con diferentes métricas. Un nuevo algoritmo iterativo que no depende de los parámetros iniciales que utilizan los métodos del *peer group* que utiliza el coseno del ángulo, se expone también en este apartado. En la sección 4.3 se presenta el algoritmo del filtro difusivo no lineal para eliminar el ruido gaussiano y speckle. En la sección 4.4 se expone un método de parametrización cuando se utiliza el *peer group* y la métrica fuzzy para imágenes de color. Por último, en la sección 4.5 se formula un método para eliminar el ruido presente en la imagen desconociendo inicialmente el tipo y la cantidad de ruido. Las implementaciones y los resultados arrojados por los experimentos numéricos se presentan en el capítulo 6 para estos algoritmos.

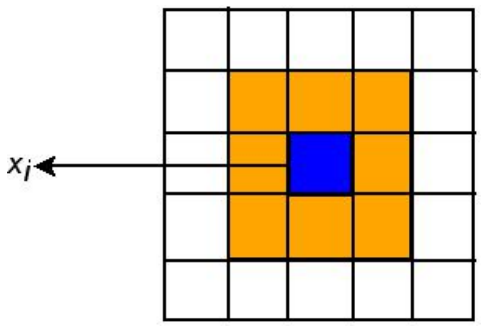


Figura 4.1: Ventana W de filtrado.

4.2 Algoritmos para eliminar el ruido impulsivo.

A partir de los trabajos (Camarena y col. 2008; Morillas, Gregori y Peris-Fajarnés 2008; Camarena y col. 2010a) en los que se presentan métodos de filtrado con buenos resultados en calidad, se deducen los algoritmos que se plantean en esta tesis. Se proponen tomando en cuenta que se puedan abordar desde el punto de vista de las tarjetas gráficas para su ejecución de forma paralela.

Los algoritmos presentados en (Camarena y col. 2008; Morillas, Gregori y Peris-Fajarnés 2008; Camarena y col. 2010a) para eliminar el ruido en una imagen, constan de dos pasos: la detección y el filtrado. El paso de detección se divide en dos fases. En la primera fase, los píxeles se analizan en ventanas disjuntas y se etiquetan como corruptos, no corruptos y no diagnosticados. Los no diagnosticados son analizados en una segunda fase. En el paso de filtrado se corrigen los píxeles clasificados como corruptos. Se usan ventanas disjuntas en el paso de detección con la finalidad de acelerar el procesamiento del algoritmo (Camarena y col. 2008). En esta tesis estos dos pasos son el punto de partida.

Para calcular el *peer group* y la métrica fuzzy, euclídea o coseno del ángulo en el paso de detección, cada píxel x_i forma una ventana de $n \times n$ píxeles (ver sección 2), como se muestra en la figura 4.1. La métrica considera la distancia en intensidad que existe entre el píxel central y sus ocho vecinos en la ventana W.

Se han clasificado los algoritmos para eliminar el ruido impulsivo en:

- Detección del ruido a través del concepto del *peer group*,

- Filtros,
- Detección y filtrado (Iterativo del coseno del ángulo).

A continuación se explican cada uno de ellos.

4.2.1 Algoritmos de detección de ruido a través del *peer group*

En los algoritmos que utilizan el *peer group*, el tamaño de la ventana de filtrado es de $n \times n, n = 3$, porque remueve mayor cantidad de ruido y preserva mejor los contornos y detalles finos de la imagen.

El *peer group* utiliza una métrica para medir la distancia en intensidad entre dos píxeles x_i, x_j, por ejemplo, la métrica fuzzy, euclídea o coseno del ángulo. En las siguientes apartados se presentan los algoritmos para cada una de las métricas utilizadas.

Peer group con métrica fuzzy

La primera aproximación que se realizó en esta tesis para eliminar el ruido impulsivo, se expone en los algoritmos 1, 2 y 6.

En el algoritmo 1 denominado "$PGMF_{f1d}secuencial$" (*Peer Group* Métrica Fuzzy fase 1 detección) se calcula el *peer group* del píxel central x_i en W con la métrica fuzzy. El píxel x_i y los x_j son declarados como no corrupto si la cardinalidad (#) del $\mathcal{P}(x_i, d)$ es mayor o igual que $(m + 1)$, donde m es un umbral que decide si el píxel es corrupto o no; de lo contrario todos los vecinos de x_i que forman parte de la ventana, pero no del *peer group*, son etiquetados como no diagnosticados. Cada píxel es analizado formando su propia ventana, proceso que se realiza hasta completar todos los píxeles de la imagen.

En el algoritmo 2 denominado "$PGMF_{f2d}secuencial$" (*Peer Group* Métrica Fuzzy fase 2 detección) los píxeles previamente etiquetados como no diagnosticados son analizados en esta fase. Se calcula nuevamente el *peer group* con la métrica fuzzy de los píxeles que sean "no diagnosticados". Los píxeles que forman parte del *peer group*, son etiquetados como no corrupto si la cardinalidad del $\mathcal{P}(x_i, d)$ es mayor o igual que $(m + 1)$, de lo contrario el píxel x_i es marcado como píxel corrupto.

Después del paso de detección (algoritmos 1 y 2), los píxeles están etiquetados como corruptos o no corruptos de acuerdo con el número de píxeles que pertenecen al *peer group*.

Una vez detectado los píxeles con ruido se procesa un algoritmo de filtrado expuestos en el siguiente apartado (ver 4.2.2) por ejemplo, el AMF.

Algoritmo 1 *$PGMF_{f1d}$ secuencial.*

Entrada: Imagen ruidosa (píxeles de la imagen), m, k, d, n.
Resultado: Imagen ruidosa con píxeles clasificados como corruptos, no corruptos y no diagnosticados.

1: //La imagen se particiona en ventanas disjuntas (W) de tamaño $n \times n$ centrado en el píxel x_i;
2: **for** *cada píxel* x_i **do**
3: //Utiliza su ventana W de píxeles;
4: **for** *todo píxel* x_j *de* W, $j \neq i$ **do**
5: //Se calcula la métrica fuzzy (M ó G) entre los píxeles x_i, x_j, (ver sección 2.3.2);
6: **if** (M (x_i,x_j)$\geq d$) **then**
7: $x_j \in \mathcal{P}(x_i, d)$;
8: **end if**
9: **end for**
10: **if** ($\# \mathcal{P}(x_i, d) \geq (m+1)$) **then**
11: $\forall x_j \in \mathcal{P}(x_i, d)$, x_j se etiqueta como no corrupto;
12: $\forall x_k \in W, x_k \notin P(x_i, d)$, x_k se etiqueta como no diagnosticado;
13: **else**
14: x_i se etiqueta como corrupto;
15: $\forall x_j \in W, j \neq i, x_j$ se etiqueta como no diagnosticado;
16: **end if**
17: **end for**

Algoritmo 2 *$PGMF_{f2d}$ secuencial.*

Entrada: Imagen ruidosa con píxeles clasificados como corruptos, no corruptos y no diagnosticados, m, k, d, n.

Resultado: Imagen ruidosa con píxeles clasificados como corruptos y no corruptos.

```
1:  for cada píxel x_i etiquetado como no diagnosticado do
2:    //Utiliza su ventana W de tamaño n × n centrado en el píxel x_i;
3:    for todo píxel x_j de W, j ≠ i do
4:      //Se calcula la métrica fuzzy (M ó G) entre los píxeles x_i, x_j, (ver sección 2.3.2);
5:      if (M (x_i,x_j) ≥ d) then
6:        x_j ∈ P(x_i, d);
7:      end if
8:    end for
9:    if (# P(x_i, d) ≥ (m + 1)) then
10:     ∀x_j ∈ P(x_i, d), x_j se etiqueta como no corrupto;
11:   else
12:     x_i se declara como píxel corrupto;
13:   end if
14: end for
```

El hecho de dividirlo en detección y filtrado no es únicamente para seguir la separación de los puntos principales, sino que, en el paso de filtrado, los píxeles corruptos toman cuenta el estado de los píxeles vecinos (previamente evaluados como corruptos o no) para definir su nuevo valor. La fase de filtrado no puede comenzar hasta que la fase de detección ha finalizado.

Uno de los hardware paralelos con el que se trabaja en esta tesis es con GPUs, lo que permite tratar en paralelo cada píxel, realizando los cálculos completos para cada uno de ellos, de esta manera, se puede eliminar las ventanas disjuntas y por lo tanto la fase 2 del paso de detección desaparece ($PGMF_{f2d}$). Además, optar la estrategia de lanzar tantos hilo como píxeles tiene la imagen, se gana coalescencia (NVIDIA 2012) de acceso a los datos en memoria de la GPU. Cada hilo detecta si es corrupto o no el píxel que esta analizando, no clasifica a sus vecinos en caso de ser no diagnosticados, de esta manera se evita escribir en posiciones de memoria vecinos (se evita conflictos de acceso a memoria), y de esta manera sólo se clasifica como corrupto o no corrupto el píxel que se está analizando. En (Sánchez y col. 2011c) se comparan los resultados en calidad del algoritmo utilizando una

sola fase en el paso de detección con otros algoritmos propuestos en otros trabajos y se presentan resultados que están dentro de la misma magnitud con respecto a las métricas PSNR, MAE y NCD (descritos en la sección 2.4).

El algoritmo 3 denominado "*PGMFsecuencial*" (*Peer Group* Métrica Fuzzy) muestra la detección como una sola fase del proceso de detección.

Algoritmo 3 *PGMF secuencial.*

Entrada: Imagen ruidosa (píxeles de la imagen), m, k, d, n.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos.

1: **for** *cada píxel* x_i **do**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en el píxel x_i;
3: **for** *todo píxel* x_j *de* W, $j \neq i$ **do**
4: //Se calcula la métrica fuzzy (M) entre los píxeles x_i, x_j, (ver sección 2.3.2);
5: **if** (M (x_i,x_j)$\geq d$) **then**
6: $x_j \in \mathcal{P}(x_i, d)$;
7: **end if**
8: **end for**
9: **if** ($\# \mathcal{P}(x_i, d) \geq (m+1)$) **then**
10: x_i se declara como píxel no corrupto;
11: **else**
12: x_i se declara como píxel corrupto;
13: **end if**
14: **end for**

En el anterior caso, se definió la métrica fuzzy M, pero puede ser cualquiera de las que se definieron en el capítulo 2. Cuando se utiliza la métrica fuzzy G, la decisión de que un píxel pertenece o no al *peer group* es el mismo que con la métrica fuzzy M, "**if**(G (x_i,x_j)$\geq d$) **then**.."; en este caso se denominará al algoritmo "*PGMG secuencial*".

Peer group con la distancia euclídea

Cuando se trata de la métrica euclídea en el *peer group*, la decisión que se toma en cuenta para ver si un píxel es corrupto o no, es: "**if** (L $(x_i,x_j) \leq d$) **then**.." como se muestra en el algoritmo 4 denominado *PGME secuencial* (*Peer Group* Métrica Euclídea).

Algoritmo 4 *PGME secuencial.*

Entrada: Imagen ruidosa (píxeles de la imagen), m, d, n.
Resultado: Imagen ruidosa con píxeles clasificados como corruptos y no corruptos.

1: **for** *cada píxel* x_i **do**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en el píxel x_i;
3: **for** *todo píxel* x_j *de* W, $j \neq i$ **do**
4: //Se calcula la distancia euclídea (L) entre los píxeles x_i, x_j, (ver sección 2.3.2);
5: **if** (L $(x_i,x_j) \leq d$) **then**
6: $x_j \in \mathcal{P}(x_i, d)$;
7: **end if**
8: **end for**
9: **if** ($\# \mathcal{P}(x_i, d) \geq (m+1)$) **then**
10: x_i se etiqueta como píxel no corrupto;
11: **else**
12: x_i se etiqueta como píxel corrupto;
13: **end if**
14: **end for**

Peer group con el coseno del ángulo

El uso de la distancia entre dos píxeles a través del coseno del ángulo en el *peer group*, se expone en el algoritmo 5 denominado *PGMC secuencial* (*Peer Group* métrica Coseno). El método consiste en lo siguiente. Inicialmente se obtiene la norma 2 de todos los píxeles de la imagen. Se construye la ventana para cada píxel x_i analizado. Para cada uno de los píxeles vecinos x_j de x_i se emplea la norma euclídea y se realiza el cálculo de la distancia dividiendo el resultado de multiplicar los valores RGB de los píxeles x_i y x_j entre sus normas. Si la distancia obtenida es mayor o igual que el umbral de distancia d, entonces x_i pertenece al *peer group*. La división por cero se controla

tomando en cuenta el valor de los píxeles en el rango de [1,256] en vez de [0,255].

Algoritmo 5 *PGMC secuencial.*

Entrada: Imagen ruidosa (píxeles de la imagen), d, n.
Resultado: Imagen ruidosa con píxeles clasificados como corruptos y no corruptos.

1: **for** *cada píxel* x_i **do**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en el píxel x_i;
3: **for** *todo píxel* x_j *de* W, $j \neq i$ **do**
4: $aux = \sum x_i \cdot x_j$;
5: $distancia = aux/(\|x_i\|_2 \cdot \|x_j\|_2)$;
6: **if** $distancia \geq d$ **then**
7: $x_j \in \mathcal{P}(x_i, d)$;
8: **end if**
9: **end for**
10: **if** $(\# \mathcal{P}(x_i, d) \geq (m+1))$ **then**
11: x_i se declara como píxel no corrupto;
12: **else**
13: x_i se declara como píxel corrupto;
14: **end if**
15: **end for**

Para hacer eficiente los accesos a la norma 2 de todos los píxeles en el algoritmo 5, dicho cálculo se realiza antes de iniciar la primer sentencia y se almacena en un arreglo bidimensional. De tal manera que cuando se utilizan esos valores (sentencia cinco), ese valor ya está calculado y se accede directamente al valor correspondiente.

4.2.2 Algoritmos de filtrado

Los algoritmos de filtrado se ejecutan después de un proceso de corrección o bien directamente sobre la imagen. Dado un píxel x_i previamente marcado como corrupto, se reemplaza por el resultado del cálculo de AMF/VMF, únicamente de los píxeles vecinos etiquetados como no corruptos en W como se muestra en el algoritmo 6. Para el proceso de filtrado directamente sobre la imagen sin un paso previo de detección, se considera todos los píxeles como corruptos inicialmente. Al finalizar esta etapa la imagen está libre de ruido o filtrada.

Algoritmo 6 Filtros *secuenciales*

Entrada: Imagen ruidosa con píxeles clasificados como corruptos y no corruptos, n.
Resultado: Imagen Filtrada.
1: **for** *cada píxel* x_i *corrupto* **do**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en el píxel x_i;
3: Calcular el filtro AMF/VMF de los píxeles no corruptos de W;
4: Reemplazar x_i con la salida del filtro AMF/VMF;
5: **end for**

4.2.3 Algoritmo iterativo del coseno del ángulo

Los algoritmos que utilizan el *peer group* con cualquier métrica, son dependientes de parámetros de entrada, por ejemplo, m, d, k, de los cuales el cálculo se debe realizar previamente para poder obtener buena calidad en la imagen filtrada. Se propuso un nuevo algoritmo iterativo (algoritmo 7) que utiliza la distancia del coseno del ángulo cuya finalidad es disminuir la cantidad de parámetros de entrada. El método se explica a continuación.

Se define un umbral ($umb1$) para la cantidad de iteraciones que se ejecutan y otro umbral ($umb2$) para la toma de decisión del píxel corrupto o no. Se calcula la norma 2 de todos los píxeles de la imagen, que se empleará en la sentencia seis. Inicialmente todos los píxeles se tratan como ruidosos.

En cada iteración se identifican todos los píxeles ruidosos y para todos ellos se realiza lo siguiente:

Para todos los píxeles de la ventana y para todos los canales del píxel se obtiene la suma de la diferencia de la distancia entre los píxeles x_i, $x_j, \forall x_j \in W$. Si el valor obtenido es mayor que un $umb2$ dado, entonces se calcula el AMF y se hace el reemplazo del valor del píxel analizado. Como se obtiene un nuevo valor del píxel, se vuelve a calcular su norma. Si la diferencia es menor que el $umb1$ dado, entonces el píxel se clasifica como no corrupto, en caso contrario se sigue conservando como píxel corrupto.

En la primera iteración todos los píxeles están etiquetados como ruidosos y por lo tanto todos son evaluados. Las iteraciones finalizan cuando el $umb1$ se ha cumplido.

Algoritmo 7 *Iterativo$_{coseno}$ secuencial*

Entrada: Imagen ruidosa (píxeles de la imagen), $umb1$, $umb2$, tol, n.
Resultado: Imagen Filtrada.

while ($umb1 \geq$ tol) **do**
  $difdireccion$=0;
  for *cada píxel etiquetado como corrupto* **do**
    for *todo píxel* x_j *de* W, $i \neq j$ **do**
      aux=$\sum x_i \cdot x_j$
      $distancia$= $aux/(\|x_i\|_2 \cdot \|x_j\|_2)$;
      $diferenciadireccion$+=(1-$distancia$);
    end for
    if ($diferenciadireccion >umb2$) **then**
      Reemplazar x_i con la salida del filtro (AMF);
      Calcular $\|x_i\|_2$;
      x_i se etiqueta como no corrupto;
    end if
    if ($diferenciadireccion \leq umb1$) **then**
      x_i se etiqueta como no corrupto;
    end if
  end for
  $umb1$=$umb1$-0.1;
end while
Para todo píxel etiquetado como ruidoso aplicar el filtro (AMF) de la ventana;

Nota: $0{,}1 \leq umb1 \leq 0{,}3$; $tol = 0{,}1$ y $umb2 = 1$, son los valores de los umbrales que se han utilizado y con los que se proporcionan mejor calidad en la imagen filtrada.

4.3 Algoritmo para eliminar el ruido gaussiano y speckle.

4.3.1 Filtro difusivo no lineal

El Filtro Difusivo No Lineal (FDNL) que se explicó en el apartado 2.3.1, se representa mediante el algoritmo 8. La solución es iterativa y se requiere de un tiempo de parada. En esta tesis se hace uso del propuesto por Mrázek y Navara (Mrázek y Navara 2003). En caso de que no se llegue a cumplir el tiempo de parada después de 10 iteraciones, se obliga a que el proceso finalice.

Algoritmo 8 *FDNL.*

Entrada: Imagen ruidosa (píxeles de la imagen), u^0=I_r, t=0, Δt=0.1, ϵ=1, β=1, μ=1.
Resultado: Imagen Filtrada

1: **while** *No se cumple el tiempo de parada de la ecuación 2.27 ó t* $\leq$*10* **do**
2: Construir las matrices $\hat{A}_x$ y $\hat{A}_y$ dadas en las ecuaciones 2.18 y 2.20;
3: Resolver los dos sistemas tridiagonales dadas en la ecuación 2.23;
4: Calcular u^{t+1} dado en la ecuación 2.23;
5: Calcular el valor de μ para mantener la varianza (ecuación 2.12);
6: $t = t + 1$;
7: **end while**

Este algoritmo sólo se dispone de su implementación en secuencial. A simple vista se observa que el proceso permite la paralelización tanto en la construcción de las matrices como en la resolución de los sistemas tridiagonales. Un análisis más detallado de la sentencia cinco mostraría también el paralelismo implícito.

4.4 Método de parametrización

El uso del *peer group* con cualquiera de las métricas, requiere un estudio previo heurístico para ajustar los parámetros d y m en las métricas euclídea, fuzzy y coseno, además del valor k en la métrica fuzzy, con el fin de obtener la mejor configuración que proporcione buena calidad en la imagen filtrada.

El parámetro k incluido en la definición de la métrica fuzzy M y G, tiene una influencia importante en el rendimiento del filtro (Morillas y col. 2005). La métrica es no-uniforme en el sentido de que la medida para dos diferentes pares de números consecutivos (o vectores) puede no ser la misma. El valor de k debe ser lo suficientemente alto tal que se reduzca la no uniformidad, pero no tan alto a fin de evitar la reducción de la importancia de los datos. En los trabajos (Camarena y col. 2008; Morillas y col. 2005) se hizo un estudio para encontrar el mejor valor del parámetro k para la métrica fuzzy M en imágenes RGB, donde se menciona que los valores adecuados están entre

[512,2048], con 2^{10} (1024) como el mejor valor para una variedad de tipos de ruido. Aunque por otro lado se menciona que un valor óptimo de los parámetros d y k dependen de la imagen e intensidad de ruido y no existe un valor genérico.

En el artículo de (Smolka 2005) se menciona que dado un número entero m, se denominará como *un peer group de m vecinos* (asociado a x_i) un subconjunto del *peer group*, constituido por x_i y ($m \leq n^2 - 1$) píxeles más, en otras palabras, si $\# P(x_i, d) = c{+}1$ entonces el $P(x_i, d)$ es un *peer group* de c vecinos. En la tesis doctoral de (Camarena 2009) se analiza el comportamiento del filtro del *peer group* con la métrica fuzzy para ajustar los parámetros m y d en algunas imágenes y porcentaje de ruido. Se menciona que el valor adecuado de m= n-1, para tamaño de n=3, m= 2. El análisis de los valores óptimos para los parámetros d y m se hicieron para imágenes RGB.

El análisis para el mejor valor k expuesto en (Camarena y col. 2008; Morillas y col. 2005), no se puede generalizar a imágenes en escala de grises, tampoco para los valores d y m. Por lo anterior, en esta tesis se propone un algoritmo de parametrización (para k, m y d) cuando se desconoce el porcentaje de ruido y el tipo de imagen RGB a filtrar.

El método propuesto en esta tesis es aplicado a los algoritmos que usan el concepto de *peer group* y la métrica fuzzy para eliminar el ruido impulsivo en la imagen.

Se utilizaron 50 imágenes de la base de datos (Kodak 2012) y de (PEIPA 2003). Todas son imágenes RGB, unas son de caras con objetos en la parte de atrás y otras imágenes son las comunes en el procesamiento digital de imágenes como se muestra en la sección 6.

El método propuesto en esta tesis para obtener los mejores valores de los parámetros es el siguiente:

1. Obtener de forma heurística el mejor valor PSNR de d, m y k para cada imagen y cada porcentaje de ruido.
2. Obtener la moda en el valor m para cada porcentaje de ruido considerando todas las imágenes.
3. Obtener la moda en el valor d para cada porcentaje de ruido considerando todas las imágenes.
4. Obtener la moda en el valor k para cada porcentaje de ruido considerando todas las imágenes.

Al finalizar este proceso, se tiene el valor de los mejores parámetros para una gran variedad de imágenes.

4.5 Método PGFDNL

Se diseñó un método que lo denominamos *Peer Group Fuzzy -Difusivo No Lineal* (PGFDNL), para hacer referencia a la combinación del *peer group* (algoritmos 1, 2 y 6) que utiliza la métrica fuzzy (PGMF) con el filtro de difusión no lineal FDNL (algoritmo 8). El objetivo es eliminar de forma eficiente el ruido impulsivo, gaussiano y speckle y la combinación de los tres, sin tener información inicial del tipo de ruido contenido en la imagen. En el algoritmo 9 se presenta dicho método de detección y filtrado.

Algoritmo 9 *PGFDNL.*

Entrada: Imagen ruidosa (píxeles de la imagen), m, k, d, n,u^0=I_r, $t = 0$, Δt=0.1,ϵ=1, β=1, μ=1.
Resultado: Imagen filtrada
1: Procesar el algoritmo 1 ($PGMF_{f1d}$ *secuencial*);
2: Procesar el algoritmo 2 ($PGMF_{f2d}$ *secuencial*);
3: Procesar el algoritmo 6 (Filtros *secuenciales*);
4: Procesar el algoritmo 8 ($FDNL$);

Nota: La imagen resultante después de la sentencia uno, es la entrada para la sentencia dos, la salida de la sentencia dos es la entrada para la tres y así sucesivamente.

En la sección 7.2 se analizarán los resultados obtenidos después de aplicar este método en las imágenes con ruido.

4.6 Conclusiones

En este capítulo se exponen los algoritmos secuenciales para eliminar el ruido impulsivo, uniforme, gaussiano y speckle.

Los algoritmos de detección y filtrado del ruido impulsivo se clasifican en los que utilizan el concepto de *peer group* y los iterativos.

Los algoritmos de detección con el *peer group* , muestran la condición de formar parte del conjunto *peer group* de acuerdo a la proximidad

en intensidad de dos píxeles, basándose en la métrica fuzzy, euclídea o coseno del ángulo. Se utiliza un umbral de clasificación para decidir si el píxel analizado es corrupto o no, dependiendo de la cantidad de elementos que forman parte del *peer group*.

Una versión que se ha desarrollado, utiliza dos fases para la detección y trabaja sobre ventanas de filtrado disjuntas. Se ha analizado esta versión y se ha diseñado un nuevo algoritmo con una sola fase en el paso de detección, de tal manera que pueda ser paralizado en GPUs.

El algoritmo de filtrado que se presenta (AMF o VMF), se aplica para eliminar el ruido de los píxeles de la imagen. El proceso se realiza después de una etapa de detección o directamente en la imagen sin previa clasificación.

El algoritmo presentado que elimina el ruido gaussiano y speckle, se basa en el filtro difusivo no lineal y se trata de un proceso iterativo y el criterio de parada utilizado, es el propuesto por Mrázek y Navara o bien hasta que se alcance un número máximo de iteraciones.

El uso del *peer group* con cualquiera de las métricas, requiere un estudio previo heurístico para ajustar los parámetros d y m en las métricas euclídea, fuzzy y coseno, además del valor k en las métricas fuzzy, con la finalidad de obtener la mejor configuración que proporcione buena calidad en la imagen filtrada. Por un lado, con el objetivo de obtener valores generales para dichos parámetros independientemente de la cantidad de ruido introducido en la imagen, se ha desarrollado un método al que se ha denominado "método de parametrización". Este método se aplica para eliminar el ruido impulsivo en las imágenes a color cuando se utiliza la métrica fuzzy. Por otro lado, con el fin de disminuir la cantidad de los parámetros y evitar el estudio previo heurístico, se ha diseñado un nuevo algoritmo iterativo denominado "$iterativo_{coseno}$", utiliza el coseno del ángulo para medir la diferencia en intensidad de dos píxeles. Este algoritmo combina la detección y filtrado en una misma etapa para eliminar el ruido impulsivo independientemente de la densidad de ruido introducido.

Cuando se filtra una imagen real se desconoce el tipo y la cantidad de ruido presente en la imagen a procesar. El método PGNLD se propone para reducir eficientemente el ruido en las imágenes sin tener información inicial del tipo de ruido contenido en la imagen. Este filtro es una combinación de los algoritmos que utilizan el concepto de *peer group* con la métrica fuzzy y el filtro de difusión no lineal.

Capítulo 5

Descripción de los métodos y algoritmos paralelos

5.1 Introducción

En el capítulo anterior se explicaron los métodos secuenciales que se implementaron en esta tesis para eliminar el ruido en una imagen. En este capítulo se presentan las versiones paralelas que se han propuesto para ser ejecutadas en GPUs, multicores y una combinación de ellas. Primeramente se analizó los algoritmos secuenciales que utilizan el *peer group* en la etapa de detección y AMF o VMF en la etapa de corrección para identificar las partes más apropiadas y después ser paralelizadas.

La primera implementación se efectúa en multicore con OpenMP, el segundo con CUDA en NVIDIA y el tercero es una combinación de multicore y GPUs.

Para la distribución de los píxeles de una imagen en la primera implementación, se divide el número de píxeles por el número de cores (menor o igual a los disponibles). En la segunda aplicación, la imagen se divide por un número menor o igual a las GPUs disponibles. En la tercera aplicación, los píxeles se distribuyen en multicores y GPUs.

El diagrama de la figura 5.1, muestra el proceso que se sigue para eliminar el ruido con estas tres implementaciones. Como datos de entrada tenemos la imagen, el tamaño de la imagen, el número de cores y

GPUs que intervienen, la distribución de los píxeles que será asignado a los cores $(1/2, 1/4, \ldots)$ y por último, los valores de las variables m, d y k necesarios para el proceso de detección de los píxeles en caso de que se utilice el *peer group* y la métrica fuzzy. Se tienen dos decisiones principales: cuando se trabaja en cores (si el número de cores es mayor a cero) y/o cuando se trabaja en GPUs (si el número de GPUs es mayor a cero).

- Si el número de cores es cero, quiere decir que el algoritmo se procesa en GPUs, y entonces el tamaño de la imagen se divide entre el número de GPUs disponibles.
- Si el número de GPUs es cero, significa que se trabaja sólo en arquitectura de CPU y el tamaño de la imagen se divide entre el número de cores que intervienen.
- Si el número de cores y GPUs es mayor que cero, implica que la ejecución se lleva acabo en ambas arquitecturas y por lo tanto, se distribuyen los píxeles que corresponden procesarse en cada arquitectura.

Además, si se trabaja con GPU hay un paso de transferencia de los datos de la memoria RAM a la memoria de la GPU, en caso contrario, los cores directamente procesan los datos. Después se procesan los pasos de detección y filtrado obteniendo una imagen filtrada. Los algoritmos paralelos en *multicore* no se especifican en este capítulo, pues coinciden con las versiones secuenciales trabajando con la parte de la imagen que le corresponda en la distribución de la carga.

5.2 Algoritmos paralelos para GPU

Programar en CUDA implica que los datos a ser procesados deben ser transferidos explícitamente de la memoria principal a la memoria de la GPU y una vez que los cálculos finalizaron, los resultados deben ser copiados de vuelta a la memoria principal. Antes de comenzar el procesamiento en la GPU, el control del programa en la CPU debe seleccionar la GPU a utilizar, de esta manera se copian los datos y se lanzan los *kernels* para consecuentemente devolver los resultados. Las tareas a ejecutarse por una GPU son codificados en funciones llamadas *kernels*.

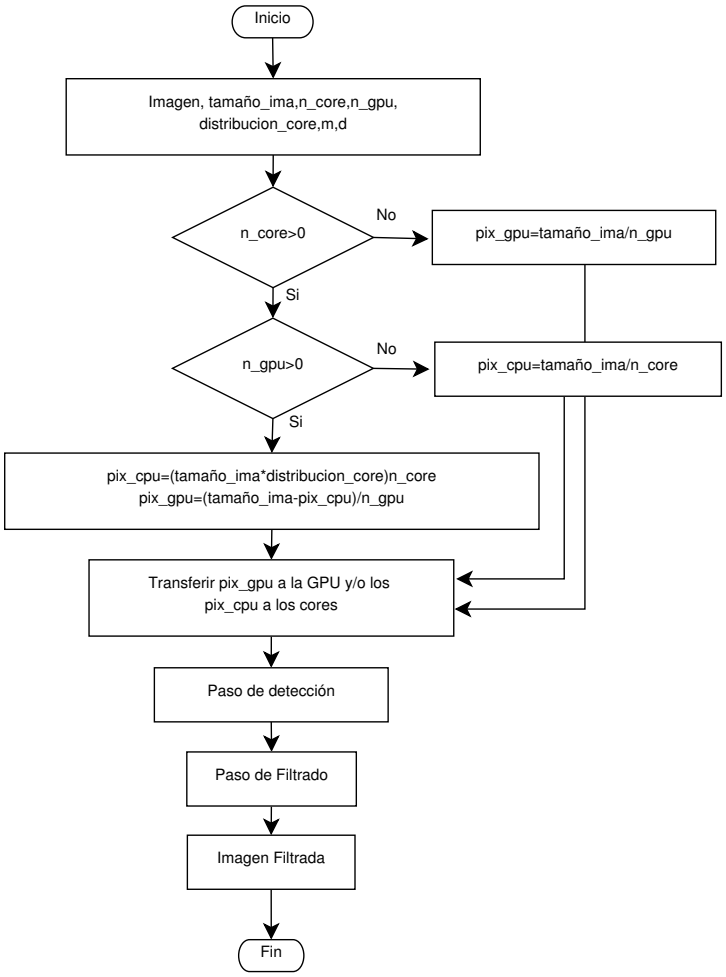


Figura 5.1: Diagrama de flujo para eliminar el ruido a través de arquitecturas CPU, GPU e híbridas.

Algoritmos de peer group con métrica fuzzy

Los algoritmos 1 y 2 se modificaron para adaptarlos a las arquitecturas paralelas, dando lugar a los algoritmos 10 y 11 . El algoritmo 10 trabaja con ventanas disjuntas. El símbolo # representa la cardinalidad del conjunto.

En todas las versiones paralelas, cada hilo analiza un píxel de la imagen. El tamaño de la ventana W al igual que en los algoritmos secuenciales fue $n \times n$, n=3. La implementación paralela en GPU se desarrolla en dos *kernel*. Un *kernel* de detección y un *kernel* de filtrado. El *kernel* de filtrado no se lanza hasta que el *kernel* de detección ha finalizado.

Algoritmo 10 *$PGMF_{f1d}$ paralelo*

Entrada: Imagen ruidosa (píxeles de la imagen),m, d, k, n.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos, no corruptos y no diagnosticados.

El hilo *i que se corresponde con el píxel x_i* **hace:**
  //Utiliza su ventana W de píxeles de tamaño $n \times n$;
  for *todo píxel x_j de W, $j \neq i$* **do**
    //Se calcula la métrica fuzzy (M) entre los píxeles x_i, x_j;
    if ($M(x_i, x_j) \geq d$) **then**
      $x_j \in \mathcal{P}(xi, d)$;
    end if
  end for
  if ($\#\mathcal{P}(x_i, d) \geq m + 1$) **then**
    etiquetar:
    $\forall x_j \in \mathcal{P}(x_i, d)$, x_j como no corrupto;
    $\forall x_k \in W, x_k \notin \mathcal{P}(x_i, d)$, x_k como no diagnosticado;
  else
    etiquetar:
    x_i como corrupto;
    $\forall x_j \in W, j \neq i, x_j$ como no diagnosticado;
  end if
end

Algoritmo 11 *$PGMF_{f2d}$ paralelo*

Entrada: Imagen ruidosa con píxeles clasificados como corruptos, no corruptos y no diagnosticados, m, d, k, n.
Resultado: Imagen ruidosa con píxeles clasificados como corruptos y no corruptos.

```
1:  El hilo i que se corresponde con el píxel x_i etiquetado como no diagnosticado hace:
2:    //Utiliza su ventana W de tamaño n × n centrado en el píxel x_i;
3:    for todo píxel x_j de W, j ≠ i do
4:      //Se calcula la métrica fuzzy (M) entre los píxeles x_i, x_j;
5:      if (M(x_i, x_j) ≥ d) then
6:        x_j ∈ P(xi, d);
7:      end if
8:    end for
9:    if (#P(x_i, d) ≥ m + 1) then
10:     etiquetar:
11:     x_i como no corrupto;
12:   else
13:     etiquetar:
14:     x_i como corrupto;
15:   end if
16: end
```

Los algoritmos 3, 4 y 5 pueden ser paralelizados fácilmente porque el trabajo realizado por cada píxel se puede realizar independientemente y se corresponden a los algoritmos paralelos 12, 13 y 14 respectivamente. Cada hilo analiza un píxel de la imagen y construye su ventana W de píxeles.

Por la misma razón que se expuso en el algoritmo secuencial (la mayoría de los píxeles que se detectan como no diagnosticados en el algoritmo 1 pasan a ser corruptos en el algoritmo 2) se fusionaron estas dos fases para obtener un nuevo algoritmo. El algoritmo 12 muestra esta versión. En este algoritmo, cada píxel de la imagen construye su ventana, obtiene el valor del píxel y calcula la distancia de acuerdo a una métrica usada. Si el tamaño del *peer group* es mayor que $m + 1$, el píxel se etiqueta como no corrupto.

Algoritmo 12 *PGMF paralelo*

Entrada: Imagen ruidosa (píxeles de la imagen), m, d, k, n.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos.

1: **El hilo** *i que se corresponde con el píxel* x_i **hace:**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en x_i;
3: **for** *todo píxel* x_j *de* W, $j \neq i$ **do**
4: //Se calcula la métrica fuzzy (M) entre los píxeles x_i, x_j (ver sección 2.3.2);
5: **if** $(M(x_i, x_j) \geq d)$ **then**
6: $x_j \in \mathcal{P}(xi, d)$;
7: **end if**
8: **end for**
9: **if** $(\#\mathcal{P}(x_i, d) \geq m + 1)$ **then**
10: etiquetar:
11: x_i como no corrupto;
12: **else**
13: etiquetar:
14: x_i como corrupto;
15: **end if**
16: **end**

Algoritmo 13 *PGME paralelo*

Entrada: Imagen ruidosa (píxeles de la imagen), m, d, n.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos.

1: **El hilo** *i que se corresponde con el píxel x_i* **hace:**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en x_i;
3: **for** *todo píxel x_j de W, $j \neq i$* **do**
4: //Se calcula la distancia euclídea (L) entre los píxeles x_i, x_j (ver sección 2.3.2);
5: **if** $(L(x_i, x_j) \leq d)$ **then**
6: $x_j \in \mathcal{P}(xi, d)$;
7: **end if**
8: **end for**
9: **if** $(\#\mathcal{P}(x_i, d) \geq m + 1)$ **then**
10: etiquetar:
11: x_i como no corrupto;
12: **else**
13: etiquetar:
14: x_i como corrupto;
15: **end if**
16: **end**

Después de aplicar el algoritmo 12, 13 o 14 los píxeles están etiquetados como píxeles corruptos y no corruptos. Para filtrar los píxeles etiquetados como no corruptos, se ejecuta el algoritmo 15. Cada hilo lee el estado del píxel, si se trata de un píxel corrupto, entonces aplicamos el método AMF, VMF o cualquier otro filtro sobre los píxeles vecinos no corruptos. El hilo i analiza el píxel i de la imagen. Se transfiere la imagen del *host* a la GPU y después de realizar el algoritmo los datos son devueltos a la memoria de la CPU.

Como se mencionó anteriormente, la imagen no se particiona en ventanas disjuntas en la implementación en GPU ya que un hilo lanzado trabaja en cada píxel de la imagen. Así, cada hilo realiza la clasificación (*corrupto* o *no corrupto*), por esta razón se realiza una sola fase en el paso de detección.

Construir la ventana W significa la lectura de los valores RGB de sus píxeles vecinos. Puede calcularse la media o mediana de los píxeles no corruptos, o en su defecto de todos los píxeles cuando no se utiliza el paso de detección.

El algoritmo 14 es la representación del algoritmo de detección de los píxeles erróneos con el coseno del ángulo.

Algoritmo 14 *PGMC paralelo*

Entrada: Imagen ruidosa (píxeles de la imagen), m, d, n.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos.

1: **El hilo** *i que se corresponde con el píxel* x_i **hace:**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en x_i;
3: **for** *todo píxel* x_j *de* W $j \neq i$ **do**
4: $aux = \sum x_i \cdot x_j$;
5: $distancia = aux/(\|x_i\|_2 \cdot \|x_j\|_2)$;
6: **if** $(distancia \geq d)$ **then**
7: $x_j \in \mathcal{P}(xi, d)$;
8: **end if**
9: **end for**
10: **if** $(\#\mathcal{P}(x_i, d) \geq m + 1)$ **then**
11: etiquetar:
12: x_i como no corrupto;
13: **else**
14: etiquetar:
15: x_i como corrupto;
16: **end if**
17: **end**

Algoritmos de filtrado

El algoritmo secuencial 6 se paralelizó dando lugar al algoritmo 15.

Algoritmo 15 *Filtros paralelos*

Entrada: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos,n.
Resultado: Imagen Filtrada.

1: **El hilo** *i que se corresponde con un píxel* x_i *corrupto* **hace:**
2: //Utiliza su ventana W de tamaño $n \times n$ centrado en el píxel x_i;
3: Calcular el filtro AMF/VMF de los píxeles no corruptos de W;
4: Reemplazar x_i con la salida del filtro AMF/VMF.
5: **end**

En el algoritmo 16 se presenta el proceso de filtrado de los píxeles cuando la ventana se expande hasta encontrar un píxel vecino que no sea corrupto para realizar los cálculos de AMF o VMF.

Algoritmo 16 *$Filtrado_{expW}$ paralelo*

Entrada: Imagen ruidosa con píxeles etiquetados como corruptos y no corruptos, n.
Resultado: Imagen Filtrada

```
1: El hilo i que se corresponde con un píxel x_i corrupto hace:
2:   //Utiliza su ventana W de tamaño n × n centrado en el píxel x_i;
3:   if todos los píxeles de la ventana son corruptos then
4:     Expandir la ventana W;
5:     Calcular el filtro AMF/VMF de los píxeles no corruptos de W;
6:     Reemplazar x_i con la salida del filtro AMF/VMF;
7:   else
8:     Calcular el filtro AMF/VMF de los píxeles no corruptos de W;
9:     Reemplazar x_i con la salida del filtro AMF/VMF;
10:  end if
11: end
```

5.3 Algoritmos híbridos para GPUs y *multicores*

En esta sección se presenta las versiones de los métodos para trabajar en arquitecturas heterogéneas, es decir, en *manycore* (GPUs) y *multicore* (CPU).

En esta implementación los hilos trabajan concurrentemente, dependiendo de la arquitectura asignada, es decir, en un determinado tiempo la función de detección se está ejecutando con los hilos trabajando en cores y simultáneamente el *kernel* de detección está siendo procesado por otros hilos asignados a la GPU. El algoritmo 17 muestra los pasos a realizar por el proceso maestro en (P_0) donde se reparte el trabajo tanto en GPUs y *multicore*. En esta implementación se lanzan tantos hilos como píxeles tiene la imagen.

Algoritmo 17 *MCGPUs* (*MultiCore GPUs*).

Entrada: Número de GPUs, número de cores CPU, imagen (píxeles de la imagen), tamaño de la imagen.
Resultado: Imagen Filtrada

1: Distribuir los píxeles de la imagen entre las GPUs y cores;
2: **if** *el hilo i corresponde a la ejecución en GPUs* **then**
3: Asignación de memoria en el *device* para los píxeles;
4: Transferir los datos de CPU(P_0) a GPU; //de la memoria del *host* a la memoria del *device*;
5: Los hilos ejecutan en paralelo en GPU;
6: Ejecutar el *kernel* de detección;
7: Sincronizar los hilos;
8: Ejectuar el *kernel* de filtrado;
9: Transferir los datos de GPU a CPU(P_0); //de la memoria del *device* a la memoria del *host*;
10: Sincronización de hilos;
11: Liberar memoria;
12: **end if**
13: **if** *el hilo i corresponde a la ejecución en cores* **then**
14: Distribuir los píxeles de la imagen entre los cores a ser usados ($P_0, P_1, P_2 \ldots$);
15: Ejecutar la función de detección;
16: Ejecutar la función de filtrado;
17: **end if**
18: Agrupar la información en P_0 para formar la imagen resultado;

5.4 Conclusiones

En este capítulo se exponen los algoritmos paralelos para implementarse en *multicore* y/o en *GPUs*. Estos algoritmos eliminan el ruido impulsivo o uniforme de la imagen.

En las versiones paralelas ejecutadas en GPU, cada hilo analiza un píxel de la imagen. En las versiones paralelas procesadas en *multicore*, un conjunto de filas de píxeles son examinados por un core.

Los algoritmos paralelos para detectar los píxeles ruidosos utilizan el concepto de *peer group*. Cada hilo que corresponde con un píxel de la imagen para su análisis, calcula el conjunto del *peer group* de acuerdo a la proximidad en intensidad de su píxel con cada vecino dentro de la ventana usando la métrica fuzzy, euclídea o coseno del ángulo.

Se utiliza un umbral de clasificación para decidir si el píxel analizado es corrupto o no, dependiendo de la cantidad de elementos que forman parte del conjunto *peer group*. Cada hilo realiza este proceso del píxel que le corresponde analizar.

Una versión del método utiliza dos fases para la detección y ventanas disjuntas. Se ha analizado esta versión y se ha diseñado un nuevo algoritmo con una sola fase en el paso de detección. Esta fusión se realizó pensando en que cada píxel se puede tratar en paralelo en las tarjetas gráficas.

Cada hilo aplica el algoritmo de filtrado para eliminar el ruido de los píxeles de la imagen. El filtrado se realiza después de una etapa de detección o directamente en la imagen sin previa clasificación.

En el algoritmo híbrido que se presenta, los hilos trabajan concurrentemente dependiendo de la arquitectura asignada, es decir, en un determinado tiempo la función de detección se está ejecutando en cores con los hilos que le corresponden y simultáneamente el *kernel* de detección está siendo procesado por otros hilos asignados a la GPU.

Capítulo 6

Implementación y evaluación de los algoritmos de filtrado en imágenes

6.1 Introducción

En este capítulo abordamos los detalles de las implementaciones de los algoritmos tanto secuenciales como paralelos en distintos tipos de arquitecturas: implementaciones de algoritmos secuenciales en CPU, implementaciones de algoritmos paralelos en CPU, en GPU y en combinación CPU-GPU. Se hará referencia indistintamente a los conceptos "paralelo CPU" y *multicore*; "GPU" y *manycore*. El problema de referencia estudiado es la eliminación de ruido impulsivo y uniforme. En cada caso aportamos los datos obtenidos en cuanto a rendimiento computacional y calidad obtenidos. Finalmente hacemos una comparación entre las aproximaciones objeto de estudio.

En concreto, en la sección 6.2, se analiza el rendimiento computacional obtenido al ejecutar los algoritmos secuencialmente. Además, se presenta una nueva versión optimizada en los accesos a memoria principal y de la cantidad de cálculos realizados.

En la sección 6.3, se analiza el rendimiento computacional obtenido al ejecutar los algoritmos en arquitecturas *multicore* y de la versión optimizada con este tipo de arquitecturas.

En la sección 6.4, se configuran las características de hardware para optimizar el procesamiento de los algoritmos en GPU. Se presenta la

definición de los *kernels* que se lanzan en la GPU con diferente patrón de acceso y se realiza un análisis computacional de las operaciones aritméticas y de los accesos a memoria. Se analiza la calidad de las implementaciones con tres filtros de la literatura. Para finalizar este apartado, se comparan los costes computacionales entre la versión secuencial y paralela.

En la sección 6.5, se presentan los resultados obtenidos de los algoritmos paralelos implementados en *multicore* y *manycore* en forma complementaria (combinado). Se evalúa la carga computacional que cada arquitectura puede procesar con la finalidad de obtener mejor rendimiento aprovechando ambas arquitecturas.

Se comparan los costes computacionales entre la versión secuencial y paralela y se analiza el speedup alcanzado en la sección 6.6.

El análisis de calidad de las implementaciones y su comparación con otros filtros de la literatura, se presentan en la sección 6.7.

Para finalizar este apartado, en el sección 6.8 se exponen las conclusiones a las que se ha llegado después de los resultados obtenidos en los experimentos.

Se utilizó OpenMP para manejar el paralelismo en *multicore* y controlar los hilos asociados a las GPUs. Los artículos (Sánchez, Vidal y Bataller 2010; Sánchez, Vidal y Bataller 2011; Sánchez y col. 2011a; Sánchez y col. 2011c; Sánchez, Vidal y Bataller 2012; Sánchez y col. 2011b; Sánchez y col. 2012c) son algunos ejemplos de trabajos donde se refleja el rendimiento obtenido de los algoritmos al ser implementados en forma paralela en estas arquitecturas.

Imágenes utilizadas

En el área de procesamiento digital de imágenes, hay una gran variedad de imágenes que se utilizan para ser analizados. Las imágenes a color que se utilizaron en esta tesis, fueron obtenidas de la base de datos de Kodak (Kodak 2012) (figura 6.1) y de otras bases de datos (figura 6.2). En el caso de las implementaciones médicas, las imágenes mamográficas están en escala de grises y fueron obtenidas de la base de datos de minimias (PEIPA 2003) (figura 6.3).

En algunas pruebas, las imágenes fueron redimensionadas a dimensiones cuadradas de 128, 256, 512, 1024, 2048 y 4096 píxeles; en otras su redimensión fue de 96×64,192×128, 384×256, 768×512, 1536×1024,

3072×2048 y 6144×4096 con la finalidad de analizar el coste computacional con tamaños de datos diferentes.

El ruido impulsivo añadido en las imágenes, fue generado utilizando el entorno MATLAB a través de la instrucción *imnoise*. Las densidades utilizadas fueron de 0.1, 0.2, 0.3, 0.4 o 0.5 que representan un 5, 10, 20, 30, 40 y 50 % de ruido impulsivo respectivamente.

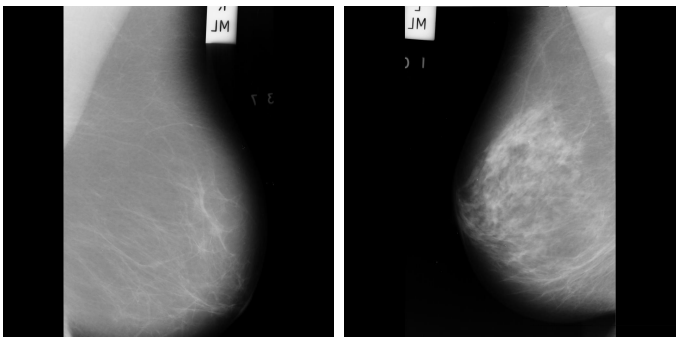
(a) (b)

(c) (d)

Figura 6.1: Imágenes a color de la base de datos de Kodak. a) *Caps* 768x512, b)*Building* 768x512, c)*Girl* 512x768, d)Estatua 512x768.

(a) (b)

Figura 6.2: Imágenes a color. a) *World* 2400x1200, b) *Lenna* 512x512.



(a) (b)

Figura 6.3: Imágenes de mamografías. Tamaño 1024×1024.

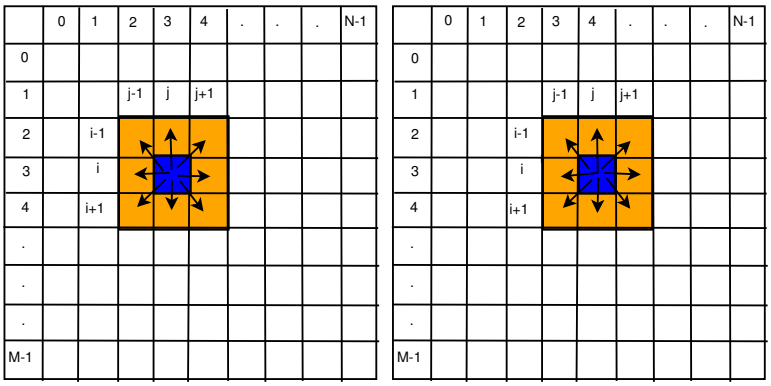


Figura 6.4: Ejemplo de acceso y cálculo a los datos en una ventana de filtrado W. a) píxel i , b) píxel $i + 1$.

6.2 Implementaciones secuenciales

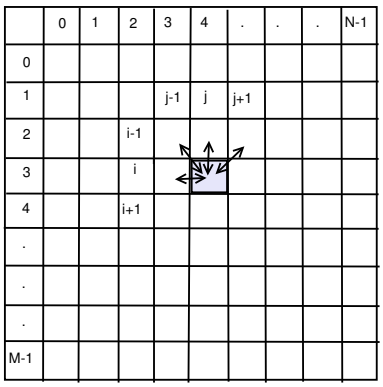
Los algoritmos 4 (*PGME secuencial*), y 6 (*AMF secuencial*) presentados en el capítulo 4, se implementaron secuencialmente en lenguaje C en dos funciones separadas para realizar el proceso de detección y corrección, de esta forma se asegura que los datos en la etapa de corrección estén actualizados para realizar el AMF sobre los píxeles etiquetados previamente como no corruptos. Se cuenta con una estructura de datos para almacenar el estado del píxel que se corresponde con los píxeles de la imagen.

El proceso de detección requiere calcular las distancias a los 8 vecinos de su ventana W (ver figura 6.4). En esta figura podemos visualizar que se producen cuatro accesos y cálculos repetidos al tratar píxeles contiguos (figura 6.5). El acceso a los datos para la lectura en memoria por cada píxel, se realiza hacia posiciones de memoria anteriores (atrás) y posteriores (adelante), como se muestra en la figura 6.6.

Con la finalidad de ahorrar tiempo de procesamiento y cantidad de accesos de lectura a los datos (hacia atrás y hacia adelante) se propone una mejora que consiste en:

Por cada píxel (i, j) hacer:

- calcular la distancia solo de los vecinos (i-1,j-1), (i-1,j), (i-1,j+1), (i,j-1),



(a)

Figura 6.5: Acceso a memoria y cálculos repetidos en el píxel $i+1$ dentro de la ventana de filtrado W.

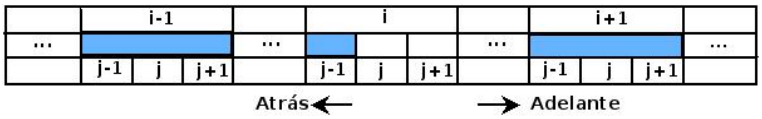


Figura 6.6: Lectura a posiciones de memoria hacia atrás y adelante.

- acumular la cardinalidad en cada uno de estos píxeles,
- tomar la decisión de "corrupto o no corrupto"del píxel (i-1,j-1).

El proceso se puede visualizar en la figura 6.7. Por ejemplo, el píxel (3,3) tiene todas la distancias calculadas hacia sus ocho vecinos en el paso (6.7e) y por lo tanto la cardinalidad está completa. En este punto, el píxel (4,4) toma la decisión de corrupto o no del píxel (i-1,j-1) o (3,3) (figura 6.8).

Con esta optimización se accede a memoria para la lectura de datos solo hacia atrás, como se muestra en la figura 6.9. Además, en lugar de realizar ocho accesos de lectura y calcular ocho veces la distancia, se accede a leer los datos solo cuatro veces y se calcula cuatro veces la distancia. Sin embargo, incrementan los accesos de escritura para aumentar la cardinalidad en los píxeles vecinos en caso de que se satisfaga la condición de pertenencia en el *peer group*.

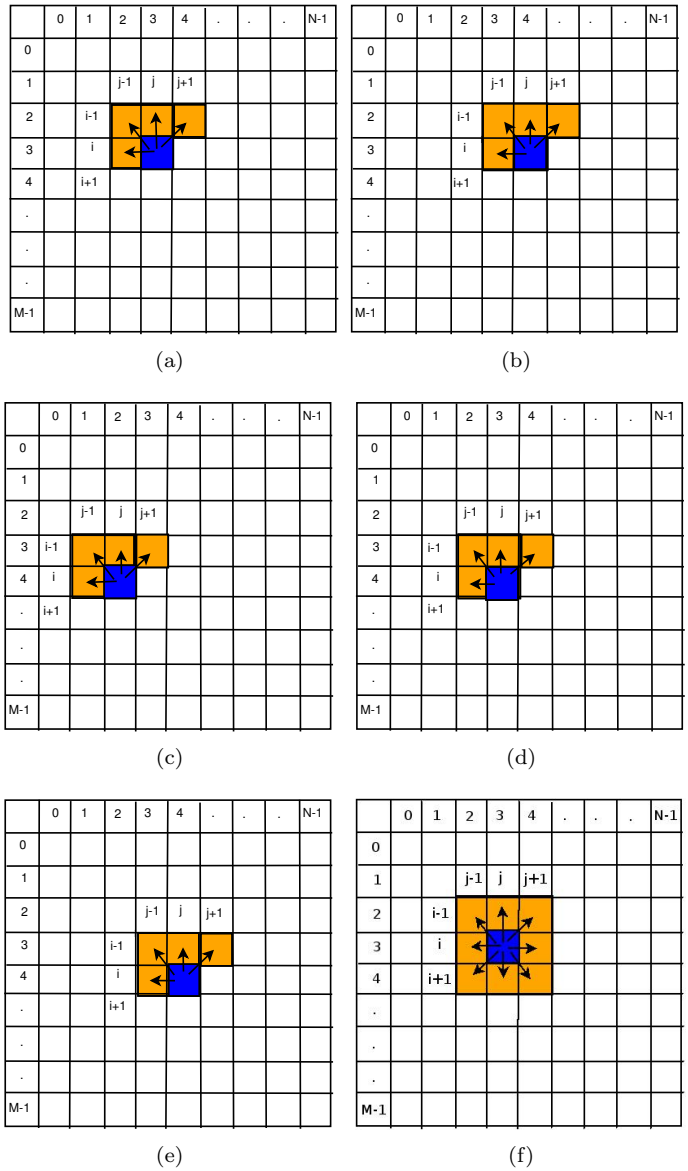


Figura 6.7: Optimización del proceso secuencial. a) Ejecución (i), b) Ejecución ($i+1$), contribución del píxel (i,j+1) a la cardinalidad del píxel (3,3), c) Contribución del píxel (i+1,j-1) a la cardinalidad del píxel (3,3), d) Contribución del píxel (i+1,j) a la cardinalidad del píxel (3,3), e) Contribución del píxel (i+1,j+1) a la cardinalidad del píxel (3,3), f) Distancias y cardinalidad completas para el píxel (3,3).

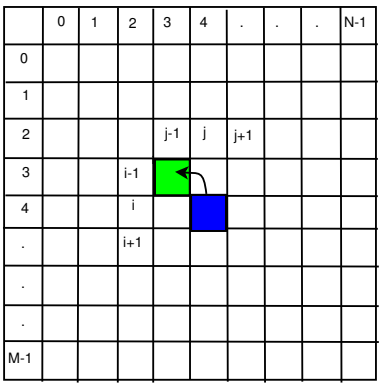


Figura 6.8: Clasificación del píxel (i-1, j-1) por el píxel (i,j) como corrupto o no corrupto.

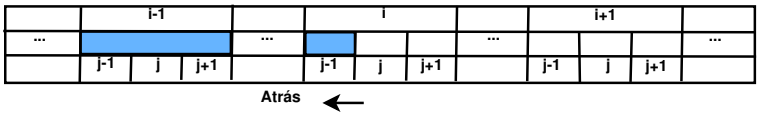
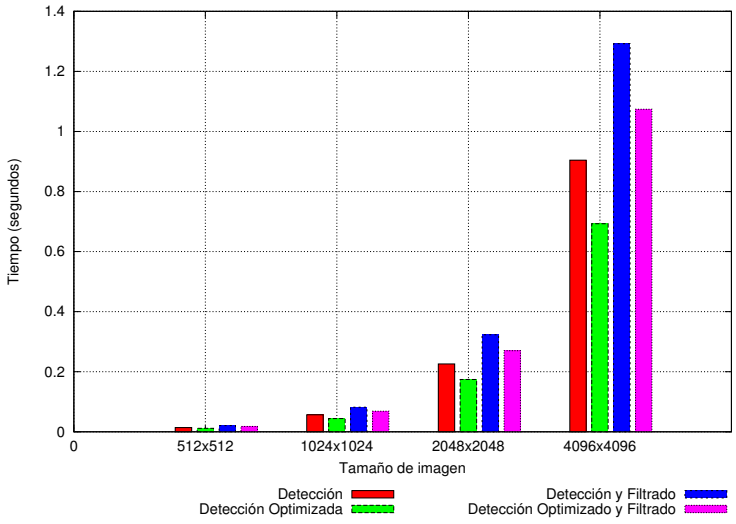


Figura 6.9: Lectura a posiciones de memoria hacia atrás.

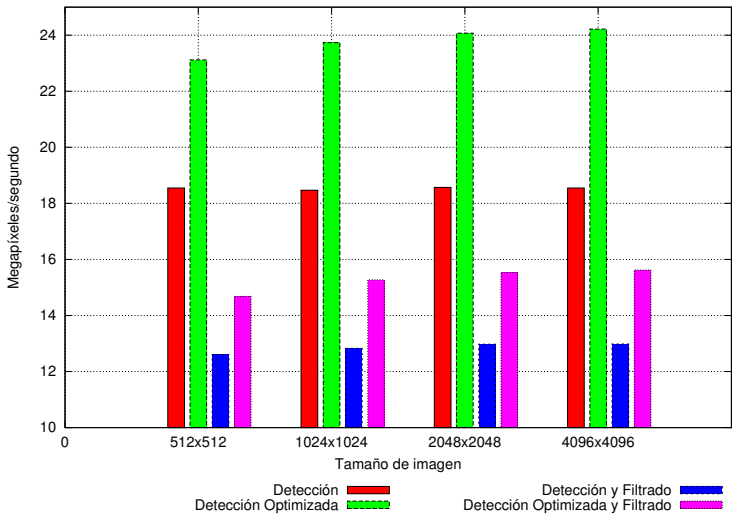
Resultados

La figura 6.10*a* presenta el tiempo obtenido en segundos de la implementación secuencial en el proceso de detección (algoritmo 4) para dos tamaños de imágenes relativamente grandes, comparado con la versión optimizada. La imagen utilizada fue *Lenna* con 10 % de ruido impulsivo. En esta figura podemos observar que el proceso de detección es mejorada computacionalmente por la detección optimizada; además, esta mejora también se refleja en el tiempo final de detección y filtrado. La figura 6.10*b* muestra la comparativa en megapíxeles por segundo procesados. Para la imagen de tamaño 4096×4096 el proceso de detección con la versión optimizada es 6 megapíxeles por segundo más, que la versión no optimizada.

En la figura 6.11 se analizan los GFlops para la etapa de detección. Como podemos ver, para el tamaño de imagen 4096×4096, los GFlops disminuyen de 1,501 a 1,08876 GFlops (27 % de reducción) cuando se compara el proceso de detección optimizada y no optimizada.



(a)



(b)

Figura 6.10: Paso de detección secuencial con y sin optimización. a) Tiempo computacional (segundos), b) Megapíxeles por segundo.

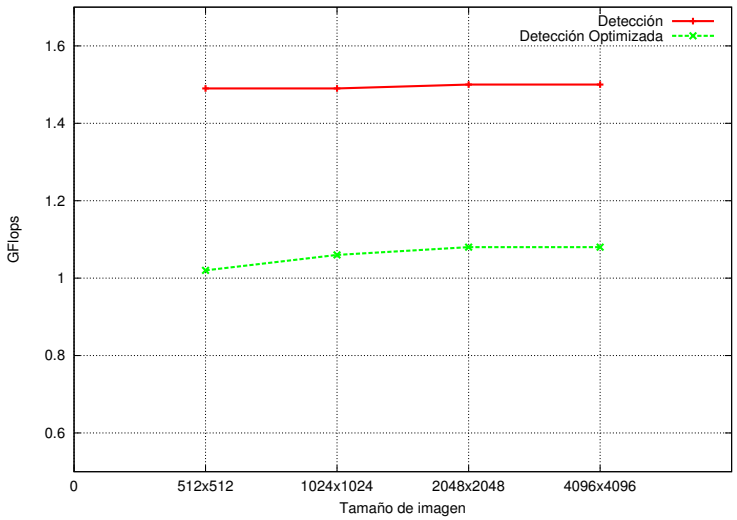
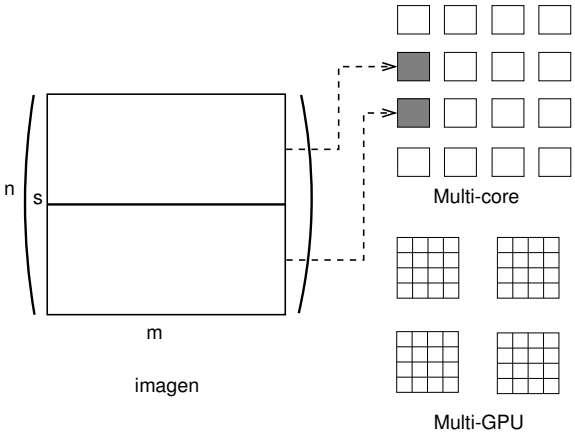


Figura 6.11: Comparativa secuencial de la etapa de detección con y sin optimización.

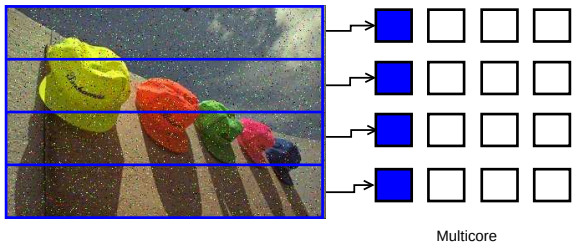
6.3 Implementaciones paralelas en CPU

Los algoritmos secuenciales 4, y 6 presentados en el capítulo 4 y la optimización propuesta en 6.2, fueron programados en arquitecturas *multicore*. La paralelización se llevó a cabo en el equipo de cómputo de altas prestaciones DSICMAC1. La implementación se realizó en OpenMP, así que los datos están actualizados y visibles para cualquier hilo.

La distribución de los píxeles entre los cores disponibles que se ha optado en esta tesis, se muestran en la figura 6.12. Se dividió el número de píxeles entre el número de cores (menor o igual a los disponibles). En el ejemplo mostrado en la figura, la división se hizo de tal manera que los renglones (píxeles) arriba de la línea s son asignados a un core y el resto a un segundo core, dejando el resto de los cores y GPUs inactivos.



(a)



(b)

Figura 6.12: Distribución de la imagen a) en 2 cores. b) en 4 cores. Ninguna GPU utilizada

Tabla 6.1: Paralelización de la imagen *Caps* en *multicore*. Tiempo en segundos.

	# cores				
Tamaño de imagen	1	2	4	8	16
96x64	0.0037	0.0020	0.0012	0.0009	0.0010
192x128	0.0148	0.0076	0.0040	0.0024	0.0024
384x256	0.0601	0.0306	0.0156	0.0087	0.0083
768x512	0.2381	0.1185	0.0604	0.0318	0.0274
1536x1024	0.9367	0.4687	0.2348	0.1256	0.0986
3072x2048	3.7023	1.8814	0.9554	0.5049	0.3817
6144x4096	14.7825	7.4480	3.7615	2.0435	1.4790

Resultados

El rendimiento obtenido al distribuir la imagen en *multicores* para diferentes tamaños de la imagen *Caps* y con ruido impulsivo del 10 % se muestra en la tabla 6.1. Se utilizó la métrica fuzzy en el *peer group* para la etapa de detección y el ajuste del parámetro para el valor de k fue de 1024 (Camarena y col. 2008), el mejor valor de d después de varios test heurísticos fue de 0.96 y $m = 2$. Este estudio fue presentado en (Sánchez y col. 2011b). Como puede observarse, el mejor tiempo se presenta cuando la carga computacional se reparte entre los 16 cores disponibles, excepto para tamaños de imagen de 96x64 píxeles; en este caso la mejor distribución píxeles-cores se presenta con ocho cores debido a que no es necesario tanto poder de cómputo para procesar imágenes relativamente pequeñas.

En la figura 6.13 se reflejan los resultados en megapíxeles por segundo procesados en *multicores* para la imagen *building* con 15 % de ruido impulsivo. Para tamaños más grandes que 384x256, se obtienen mejores resultados cuando todos los cores se utilizan. Por el contrario, con igual comportamiento que el caso anterior, para tamaños pequeños es mejor usar ocho cores. La tabla 6.2 muestra el mejor procesamiento en megapíxeles por segundo que se presenta para cada tamaño de imagen. Este análisis fue presentado en (Sánchez, Vidal y Bataller 2011).

En la figura 6.14 se presenta por un lado, los resultados obtenidos del tiempo en segundos del proceso de detección con y sin optimización paralelizada en *multicores*, y por otro, los megapíxeles procesados por segundo para la imagen *Lenna* con 10 % de ruido impulsivo, ambos resultados obtenidos al procesar los algoritmos 4 y 6. Se pude observar en la figura 6.15, que la versión paralela al utilizar ocho cores,

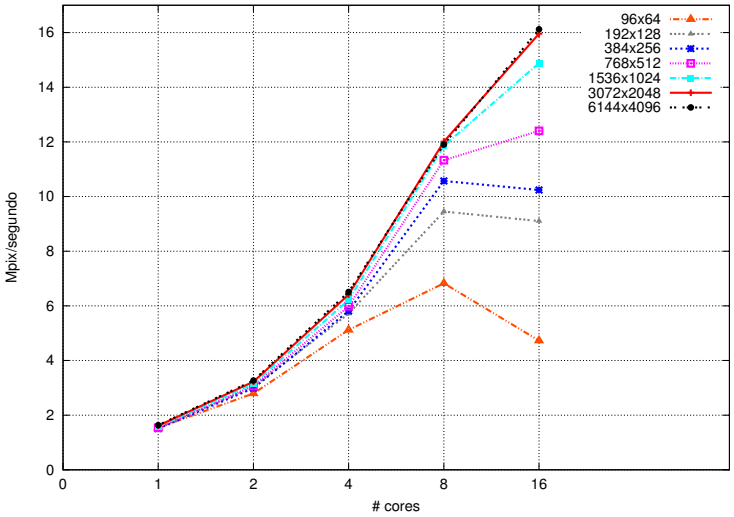


Figura 6.13: Paralelización de la imagen *building* en *multicore*

Tabla 6.2: Rendimiento en megapíxeles por segundo procesados en *multicore*. Mejores valores obtenidos para cada tamaño de imagen.

Tamaño de la imagen	*multicore*
96x64	6.8267
192x128	9.4523
384x256	10.5703
768x512	12.4043
1536x1024	14.8805
3072x2048	15.9601
3072x2048	15.9601
6144x4096	16.1206

el rendimiento en GFlops disminuye en un 41 % utilizando la versión optimizada.

6.4 Implementaciones paralelas en GPU

6.4.1 Características a configurar para optimizar las ejecuciones en GPU

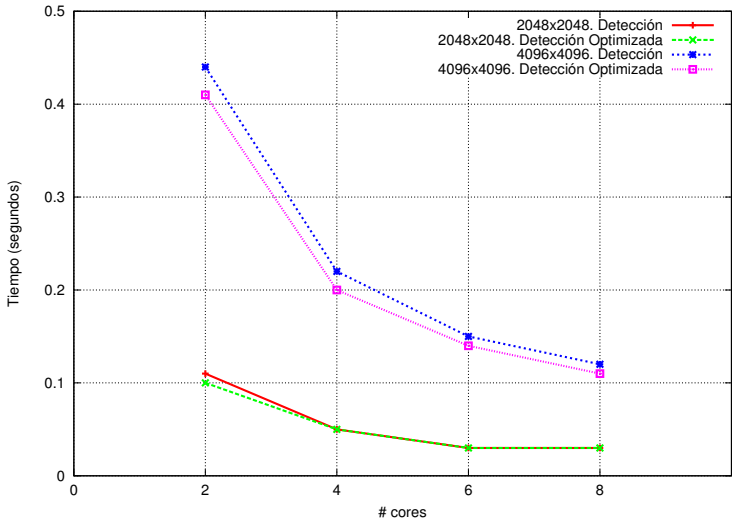
Las características principales a configurar para optimizar el procesamiento en CUDA de los algoritmos paralelos expuestos en el capítulo 5 son los siguientes:

1. Configuración de hilos y bloques.
2. Cantidad de bytes por píxel a reservar en la memoria de la GPU.
3. Modo de reserva de memoria.
4. Acceso a los datos a través de la memoria de texturas.
5. Acceso a los datos a través de la memoria compartida.

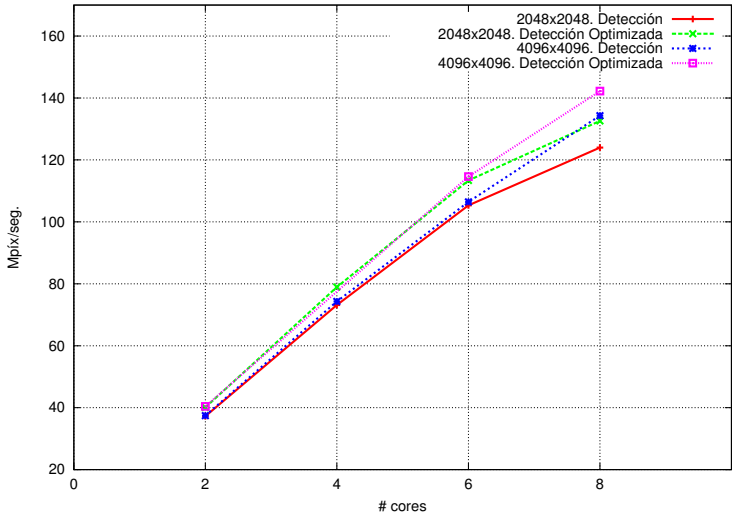
Configuración de hilos y bloques

El número de hilos usados por bloque, está limitado por los recursos de memoria de un procesador core (NVIDIA 2012). En esta tesis se trabajó solo con bloques de dos dimensiones por la simplicidad de acceder a los datos y por el uso de texturas en las implementaciones que lo utilizan. El número total de hilos por bloque en 2D debe estar lo más cercano posible al número máximo de hilos por bloque que soporta cada arquitectura. El número total de hilos es equivalente al número de hilos por bloque, por el número de bloques(NVIDIA 2012). El número de bloques en un *grid* está dictada por el tamaño de los datos procesados, en lugar de por el número de procesadores en el sistema. Este número puede exceder al número de bloques de hilos. El trabajo realizado por un procesador en cada GPU depende de los bloques lanzados por el *kernel*, teniendo en cuenta el tamaño de la imagen.

Se realizó un análisis del número de bloques que le corresponden a cada core, con la finalidad de obtener el mejor rendimiento computacional considerando el tamaño de los datos. Este análisis se llevó a cabo para



(a)



(b)

Figura 6.14: Comparativa de rendimiento del paso de detección con y sin optimización. a) Tiempo computacional, b) Megapíxeles por segundo.

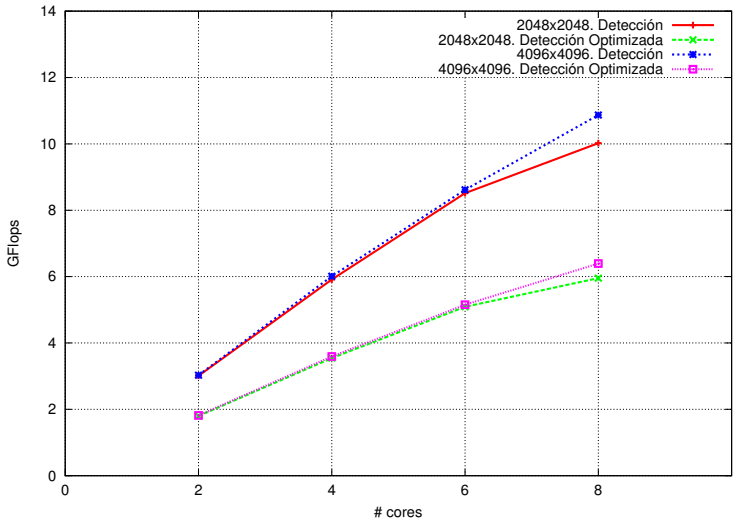


Figura 6.15: Rendimiento en GFlops de la versión paralela en *multicore* con y sin optimización de accesos a los datos y cálculos.

tres arquitecturas de GPU utilizadas. La cantidad máxima de hilos por bloque en el modelo GeForce es 512, de esta manera se pueden generar bloques de 16x16 hilos, para un total de 256 hilos activos, por lo que habrá dos bloques ejecutándose simultáneamente en cada procesador. En la arquitectura Tesla, el número máximo de hilos por bloque es de 1024, por lo que se puede generar bloques de 32x32 hilos para un total de 1024 hilos, o de 16x16 para un total de 512 hilos en cada bloque.

La tabla 6.3 muestra la configuración de bloques, hilos y la ocupación de los cores en cada una de las arquitecturas de GPU utilizadas para diferente tamaño de imagen. El número de bloques por core se obtiene al dividir el número total de bloques con los cores CUDA disponibles para cada arquitectura (ver el capítulo 3). Los tamaños de las imágenes son: (I_1)= 256x384, (I_2)= 512x768, (I_3) = 1536x1024, (I_4) = 3072x2048 y (I_5)=6144x4096. Como podemos observar, para la arquitectura Tesla el número de bloques creados para el primer tamaño de imagen y 32x32 hilos por bloque es de 96 bloques por core y ésta arquitectura tiene 448 cores CUDA, por lo que 352 cores quedan inactivos. Para el segundo tamaño de imagen 64 cores son los inactivos, a partir del tercer tamaño de imagen, todos los cores tienen trabajo por realizar. Este análisis fue presentado en (Sánchez y col. 2012c).

Tabla 6.3: Configuración de bloques e hilos.

	Tamaño de imagen	Max. hilos por bloque	Hilos por bloque	Número de bloques	Número de bloques/core
GeForce GT 120	I_1	512	16x16	16x24=384	12
	I_2			32x48=768	24
	I_3			64x96=1536	48
	I_4			128x192=3092	96
	I_5			256x384=6144	192
GeForce 9800 GX2	I_1	512		16x24=384	3
	I_2			32x48=768	6
	I_3			64x96=1536	12
	I_4			128x192=3092	24
	I_5			256x384=6144	48
Tesla M2050	I_1	1024		16x24=384	1
	I_2			32x48=768	1
	I_3			64x96=1536	3
	I_4			128x192=3092	6
	I_5			256x384=6144	13
	I_1	1024	32x32	8x12=96	1
	I_2			16x24=384	1
	I_3			32x48=1536	3
	I_4			64x96=6144	13
	I_5			128x192=24576	54

En la implementación del algoritmo 10, la imagen se dividió en $(N1 \times N2)/n$ ventanas, donde $N1$ y $N2$ es el ancho y alto de la imagen respectivamente y n=3. Así, en el primer *kernel* (detección) se generaron bloques de tamaño $(N1 \times N2)/n$ hilos. Cada hilo con coordenadas generales (f,c) dentro del grid, analiza sus píxeles de W para realizar los cálculos del *peer group*. El hilo etiqueta el píxel central x_i como no corrupto, corrupto y no diagnosticado en la primera fase del paso de detección. La segunda fase del paso de detección, descrito en el algoritmo 11 y la etapa de corrección descrito en el algoritmo 15, se generaron la misma cantidad de hilos como píxeles tiene la imagen. El hilo correspondiente al píxel x_i que está etiquetado como no diagnosticado calcula el *peer group* y si se cumple la cardinalidad de $m+1$, el hilo clasifica el píxel central como no corrupto, de lo contrario como corrupto.

Para que todos los píxeles de la imagen sean examinados, ganar coalescencia y evitar conflictos de acceso a los datos en la memoria de la GPU al implementarse los algoritmos 12 y 15, la estrategia en GPU que se utilizó fue, que el i-ésimo hilo del bloque le corresponde el i-ésimo píxel dentro de la imagen, en este caso la cantidad de hilos que se generan para los *kernels* es el mismo número de píxeles que contiene la imagen.

El estudio del rendimiento de la GPU con tres posibles combinaciones de tamaño de bloque (8x8, 16x16 y 32x32) en la arquitectura Tesla M2050 se muestra en la figura 6.16. Se puede observar que cuando se utiliza la memoria global la mejor configuración de bloques es de 16x16 y la inferior es de 8x8. Se procesan mayor cantidad de píxeles por segundo cuando se accede a los datos a través de la memoria de textura. Para el tamaño más grande y con bloques de 32x32 hilos por bloque, se puede observar que los resultados del uso de la memoria global o la memoria de textura obtienen buen rendimiento.

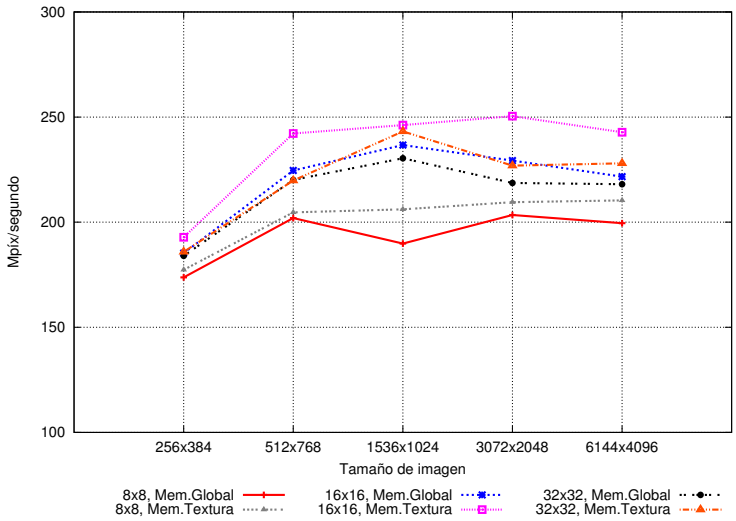


Figura 6.16: Rendimiento en megapíxeles por segundo ejecutados usando diferente tamaño de bloques para distintos tamaños de imagen en la arquitectura Tesla M2050.

Cantidad de bytes por píxel a reservar en la memoria de la GPU

La cantidad de bytes por píxel a reservar en la memoria de la GPU se refiere al almacenamiento de la imagen con tres canales por píxel o cuatro canales por píxel. La intensidad de un canal del píxel (0-255) se puede almacenar en el tipo de datos *char*, así que se reservó un byte por cada canal RGB. Se añadió un byte extra por cada píxel, al que le llamamos *padding* (p) o relleno para almacenar el estado del píxel como corrupto o no, de esta manera un píxel se compone de cuatro bytes ($RGBp$). El alineamiento en el código del *device* fue en bloques de cuatro bytes.

Tabla 6.4: Rendimiento computacional en milisegundos comparando la imagen almacenada en formato RGB y RGB*p*. Tamaños de imágenes en dimensiones cuadradas.

	256	512	1024	2048
GPU con imagen RGB	3.84	23.26	94.68	412.92
GPU con imagen RGB*p*	3.21	12.28	51.36	263.16

La tabla 6.4 presenta los resultados obtenidos al ejecutar el algoritmo 12 utilizando tres o cuatro bytes para almacenar el valor del píxel. Se empleó la imagen *Lenna* con 5 % de ruido impulsivo. Se utilizó la norma euclídea en el *peer group* y se sustituyó el primer píxel no corrupto por el píxel analizado y con 64x64 hilos por bloque. El tiempo presentado considera el tiempo de transferencia de datos entre CPU y GPU. Estudio presentado en (Sánchez y col. 2010). Se puede observar que, conforme incrementa el tamaño de la imagen se obtiene mayores prestaciones con RGB*p* frente a RGB, partiendo de una mejora del 16 % para la imagen de tamaño 256×256 hasta una mejora del 36 % para un tamaño de 2048×2048.

La tabla 6.5 muestra los resultados obtenidos del tiempo total en milisegundos, considerando la trasferencia entre CPU y GPU; el cálculo realizado por los *kernels* y los Mpix/seg procesados al utilizar el almacenamiento en RGB y en RGB*p* para la imagen *Lenna* con 10 % de ruido impulsivo. Como podemos observar, el trabajar con cuatro bytes reduce el tiempo computacional del *kernel* al momento de acceder a los datos en la GPU.

Tabla 6.5: Comparación de la versión RGB frente a RGB*p* para la imagen *Lenna* de tamaño 2048×2048 con 10 % de ruido impulsivo. Rendimiento en segundos y Mpix./seg.

	RGB	RGB*p*
Tiempo del *kernel*	1163.41	976.30
Tiempo total	1379.02	1226.30
Megapíxels/segundo	3.04	3.42

La imagen en RGB*p* (tres canales + *padding*) obtiene mejores prestaciones frente a la imagen almacenada en RGB (tres canales) debido a que se consigue que los accesos tengan coalescencia. Es decir, RGB no es múltiplo de 32, 64 o 128 bits que hacen un patrón de acceso correcto para escribir o leer de la memoria. Sin embargo, si se añade un nuevo byte, los accesos completan bloques de cuatro bytes y el rendimiento mejora, además de que el cuarto byte se ha utilizado para indicar el estado del píxel y de esta manera se tiene la ventaja de que no se

requiere reservar memoria aparte con otra estructura de datos para almacenar el estado del píxel.

Modo de reserva de memoria

El modo de reserva de memoria se refiere a transferir los datos entre las memorias CPU-GPU utilizando *cudaMallocHost* o sin él.

Se analizó el hecho de considerar si se obtiene mejor rendimiento cuando se realiza la asignación de los datos a través la función *cudaMallocHost*, tomando en cuenta los algoritmos 10, 11 y 15 . La tabla 6.6 muestra los resultados obtenidos. Se puede observar que el trabajar con *cudaMallocHost* se obtiene mejor tiempo en la transferencia de datos entre CPU-GPU y viceversa, ya que el cálculo en la GPU es el mismo. Además, se utiliza la función *cudaMallocPitch* y memoria *page-locked* para acelerar la transferencia.

Tabla 6.6: Comparación del modo de reserva de datos: RGB*p* (malloc) y RGB*p* (cudaMallocHost). Imagen *Lenna* de tamaño 2048×2048 con 10 % de ruido impulsivo. Rendimiento en segundos.

	RGB*p* (malloc)	RGB*p* (cudaMallocHost y cudaMalloc)
Tiempo del *kernel*	976.30	976.30
Tiempo total	1226.30	998.66

Acceso a los datos a través de la memoria de texturas

Se realizó un análisis para acceder a los datos de la memoria de la GPU usando dos optimizaciones: memoria compartida y memoria de texturas con la finalidad de encontrar el mejor acceso a los datos en memoria del *device* con la que se proporciona mejor rendimiento. La figura 6.17 muestra estas dos opciones: usando la memoria compartida (a) o usando la memoria de texturas (b). En este apartado se analiza la memoria de texturas y en el siguiente, la memoria compartida. Análisis presentado en (Sánchez, Vidal y Bataller 2011).

Se evaluaron las mejoras derivadas de acceder a los datos en memoria global a través de la memoria de texturas. Se usó asignación de memoria lineal en la memoria del *device* para almacenar los datos, además, con la finalidad de incrementar la velocidad de transferencia entre la memoria del *host* y la memoria del *device* se usa *page-locked*.

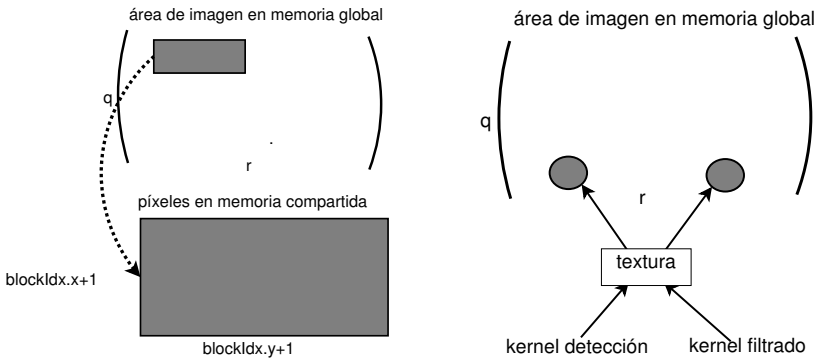


Figura 6.17: Acceso a la memoria Global de la GPU con a) Memoria compartida b) Memoria de texturas.

La función *cudaMallocPitch* se utilizó para garantizar una alineación óptima de los píxeles en la memoria global cuando se utiliza la memoria de texturas.

Debido a que la textura es de sólo lectura, cuando se requiere escribir, se realiza directamente en la memoria global. Los siguientes pasos se realizaron para procesar la imagen con texturas a través de la utilización de dos texturas:

1. Declarar la referencia de textura, una de entrada y otra de salida.
2. Asignar un arreglo de entrada 2D en memoria del *device* para almacenar los píxeles de entrada.
3. Copiar los píxeles al arreglo de entrada 2D de la textura.
4. Asociar el arreglo de entrada a la textura de entrada.
5. Asociar la textura de salida con los píxeles de salida en la memoria del *device*.
6. Ejecutar los *kernels*.
7. Copiar a la memoria del *host* los píxel de salida que se encuentran en memoria global del dispositivo.
8. Liberar memoria del dispositivo

Se realizó un análisis comparativo del almacenamiento de los datos en tres y cuatro bytes y acceso a la memoria de texturas. El uso de la memoria de texturas se encuentra almacenada en cuatro bytes. La tabla 6.7 muestra los resultados obtenidos al ejecutar los algoritmos 10, 11 y 15 para 5 % y 10 % de ruido impulsivo. Los valores de los parámetros con los que se obtuvieron dichos resultados son: $d = 0{,}95$, $k = 1024$ y $m = 2$. Estudio presentado en (Sánchez, Vidal y Bataller 2010).

Tabla 6.7: Comparativa del procesamiento en milisegundos para los modos RGB, RGB*p* y RGB*p* con texturas para 5 % y 10 % de ruido impulsivo.

	5 % de ruido			10 % de ruido		
Tamaño	RGB	RGB*p*	Textura	RGB	RGB*p*	Textura
256×256	4.25	3.09	1.79	4.77	3.47	1.95
512×512	20.56	12.59	7.11	24.38	14.41	7.88
1024 ×1024	87.14	51.01	29.33	105.80	59.21	32.50
2048×2048	408.67	250.96	136.91	503.79	278.29	141.24

Se puede observar que en el caso de utilizar el formato RGB*p* en la GPU con 5 % de ruido, la mejora en el peor de los casos es de un 27 % para el tamaño de imagen 256×256 y en el mejor de los casos de un 41 % para el tamaño de imagen 1024×1024, en comparación con la aplicación RGB. Por otro lado, con un 10 % de ruido, el rendimiento es peor con un 27 % para el tamaño de imagen de 256×256 y en el mejor de los casos de un 44 % para el tamaño de imagen 2048×2048. Los mejores resultados se obtuvieron cuando se utiliza la GPU con textura, ya que la mejora es de aproximadamente de un 45 % con 5 % de ruido y de un 43 % a 49 % con un 10 % de ruido en comparación con la versión RGB*p* (sin textura).

Definición de texturas

En algunas implementaciones se utilizaron dos texturas y en otras solo una. Cuando se definen dos texturas, una es para el *kernel* de detección y otra para el *kernel* de filtrado. Una vez que los píxeles se encuentran en la memoria del *device*, cada hilo del bloque en el proceso de detección, lee el píxel que le corresponde analizar y sus vecinos a través de la textura. Al finalizar esta etapa, el byte *padding* de RGB contiene el estado del píxel: corrupto o no. Para la fase de corrección, cada hilo del bloque lee el píxel que le corresponde analizar y los de sus vecinos a través de la segunda textura y se escriben los nuevos valores en el píxel que se está analizando.

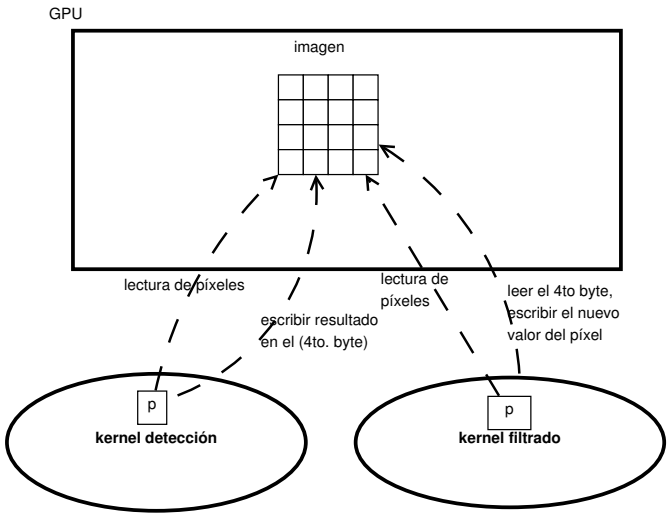


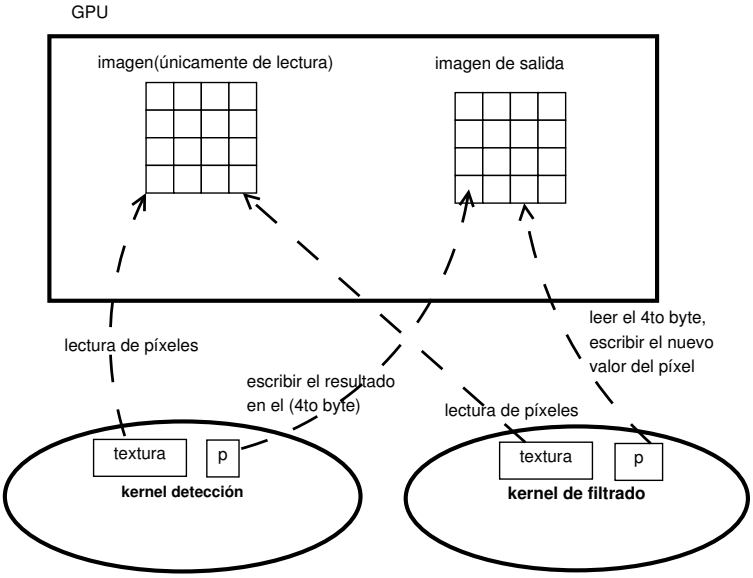
Figura 6.18: Diagrama de acceso a los datos que se encuentran en la memoria de la GPU a través de texturas.

En otra de las implementaciones solo se declaró una textura y fue usada en ambos *kernels*.

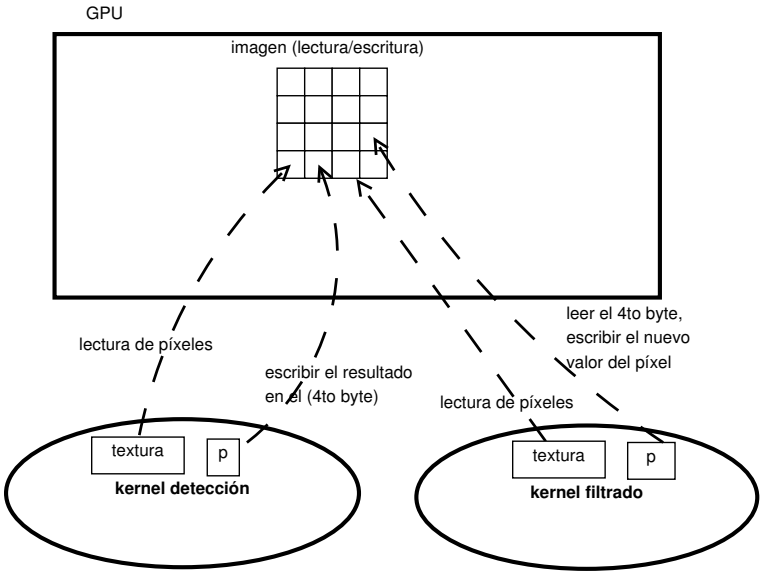
En una de las aproximaciones, se utilizó una sola copia de la imagen almacenada en la memoria global, como se muestra en la figura 6.18.

Utilizando esta configuración, mientras unos hilos leen los datos concurrentemente otros hilos pueden estar escribiendo. Por ejemplo, en el *kernel* detección, un hilo puede estar leyendo los datos de los píxeles vecinos mientras otro escribe el estado del píxel como corrupto o no corrupto. Además, en el *kernel* filtrado, un hilo debe leer los píxeles vecinos para el cálculo del AMF, mientras que otro hilo escribe el nuevo valor para un píxel ruidoso. Una segunda versión se desarrolló con dos copias de la imagen para evitar conflictos de acceso a memoria y agilizar el proceso.Una copia de los datos se utiliza para el acceso de solo lectura, y se accede a través de una textura. La segunda copia se realiza a partir de la primera y los hilos del *kernel* escriben para clasificar el píxel como corrupto o no corrupto; es a través de ésta copia por el cual el kernel de filtrado lee el estado del píxel. Esto se representa en la figura 6.19.

La tabla 6.8 muestra el tiempo computacional obtenido en GPU para cada uno de los *kernels* con un 5 % de ruido impulsivo utilizando la métrica fuzzy en el *peer group* para el filtrado de la imagen *Lenna*. Se



(a)



(b)

Figura 6.19: Diagrama de acceso a los datos que se encuentran en la memoria de la GPU a través de texturas, a) Dos copias de los datos, b) Una copia de los datos.

hace una comparativa del almacenamiento de los datos en tres bytes y cuatro bytes en memoria global frente al almacenamiento en cuatro bytes con memoria de texturas. En este caso sólo se contabiliza el tiempo de los *kernels*. Los resultados fueron obtenidos con la tarjeta gráfica GeForce GT 120. Análisis presentado en (Sánchez y col. 2011a).

Tabla 6.8: Tiempo de procesamiento en segundos de la ejecución en GPU para los *kernels* de detección y filtrado con 5 % de ruido impulsivo.

Tamaño de	Estrategia					
	3 bytes Mem. Global		4 bytes Mem. Global		4 bytes Mem. Textura	
imagen	Detección	Filtrado	Detección	Filtrado	Detección	Filtrado
128	0.92	0.52	0.73	0.37	0.28	0.19
256	3.70	1.92	2.78	1.40	0.92	0.70
512	23.40	8.38	11.20	5.47	3.60	2.69
1024	96.80	33.65	46.21	21.88	14.07	10.84
2048	427.66	141.69	215.91	90.29	57.82	46.66
4096	1937.16	599.00	886.58	364.31	232.93	186.99

Las tablas 6.9 y 6.10 muestran el tiempo de procesamiento en GPU y el tiempo total, que incluye las transferencias CPU-GPU y preparación de la imagen, para un 5 % y 10 % de ruido impulsivo. Estos resultados fueron obtenidos con la tarjeta gráfica GeForce GT 120.

Se puede observar en las tablas (6.8, 6.9 y 6.10) que el almacenamiento en cuatro bytes obtiene menor tiempo de ejecución que la opción de almacenamiento en tres bytes, además, la versión de cuatro bytes con texturas refleja el menor coste computacional comparado con las otras dos opciones para todos los tamaños de imágenes. Análisis presentado en (Sánchez y col. 2011a).

Acceso a los datos a través de la memoria compartida

La memoria compartida teóricamente debe ser más rápida que el acceso a la memoria de texturas o global, pero esto va a depender de la aplicación. En las implementaciones desarrolladas en esta tesis no se toman ventajas del uso de la memoria compartida, ya que hay dependencia de píxeles en la segunda etapa (estado del píxel - corrupto o no corrupto). Este valor debe estar almacenado en una zona de memoria que sea de lectura y escritura, además que sea visible para los demás hilos (específicamente visible para los ocho vecinos de un píxel dado), pues el píxel x_i accede al estado de todos sus píxeles vecinos para calcular el AMF de los píxeles no corruptos.

Tabla 6.9: Tiempo de procesamiento en milisegundos con diferente modo de almacenamiento de los datos. 5 %de ruido impulsivo.

Tamaño	5 %noise					
	3 bytes por píxel. Acceso a los datos directamente a memoria global		4 bytes por píxel. Acceso a los datos directamente a memoria global		4 bytes por píxel. Acceso a los datos a través de a memoria de texturas	
	GPU	Total(GPU y CPU)	GPU	Total(GPU y CPU)	GPU	Total(GPU y CPU)
128	1.44	2.03	1.11	1.34	0.47	0.87
256	5.62	6.85	4.18	4.65	1.63	2.39
512	31.78	35.79	16.67	18.11	6.29	8.54
1024	130.45	145.95	68.09	73.48	24.91	33.08
2048	569.35	630.91	306.20	327.09	104.48	136.14
4096	2536.16	2779.93	1250.89	1333.77	419.92	545.59

Tabla 6.10: Tiempo de procesamiento en milisegundos con diferente modo de almacenamiento de los datos. 10 % de ruido impulsivo.

Tamaño	10 %noise					
	3 bytes por píxel. Acceso a los datos directamente a memoria global		4 bytes por píxel. Acceso a los datos directamente a memoria global		4 bytes por píxel. Acceso a los datos a través de a memoria de texturas	
	GPU	Total(GPU y CPU)	GPU	Total(GPU y CPU)	GPU	Total(GPU y CPU)
128	1.48	2.07	1.15	1.38	0.51	0.90
256	5.82	7.07	4.34	4.82	1.78	2.56
512	32.69	36.70	17.25	18.69	6.68	9.10
1024	134.85	150.30	70.61	75.99	27.54	35.72
2048	593.74	655.02	316.54	337.47	115.35	147.00
4096	2653.68	2897.51	1300.98	1383.85	464.69	590.29

Si se realiza con memoria compartida, los bordes de cada bloque de hilos no tienen actualizado el estado de los píxeles vecinos y por lo tanto el AMF no se calcula con valores correctos, esto repercute en la calidad de la imagen final. Realizar los cálculos y escribir directamente en memoria global el coste computacional de esta solución es igual o un poco mayor comparado con al coste cuando se utiliza la memoria de texturas. El análisis de los resultados con memoria compartida y de texturas para diferente arquitectura de tarjetas gráficas fue presentada en (Sánchez y col. 2012c).

El uso de la memoria compartida implica que los datos deben ser copiados de la memoria global de la GPU a un bloque de memoria compartida, disponible únicamente por el conjunto de hilos ejecutándose.

El estudio del rendimiento de la GPU con dos posibles combinaciones de tamaño de bloque (8x8 y 16x16) en GeForce GT 120 se muestra en la figura 6.20. Se puede observar que el mejor tamaño de bloque es de 8x8 con memoria de textura y 16x16 para los otros dos accesos. La memoria compartida con bloques de 16x16 es mejor que la memoria de textura con bloques de 16x16 y la memoria global con bloques de 8x8. La razón radica en que el procesador está inactivo a la espera de más datos; en otras palabras, la relación entre la velocidad de reloj del procesador y el ancho de banda no son apropiados. El comportamiento en el modelo GeForce 9800 son similares.

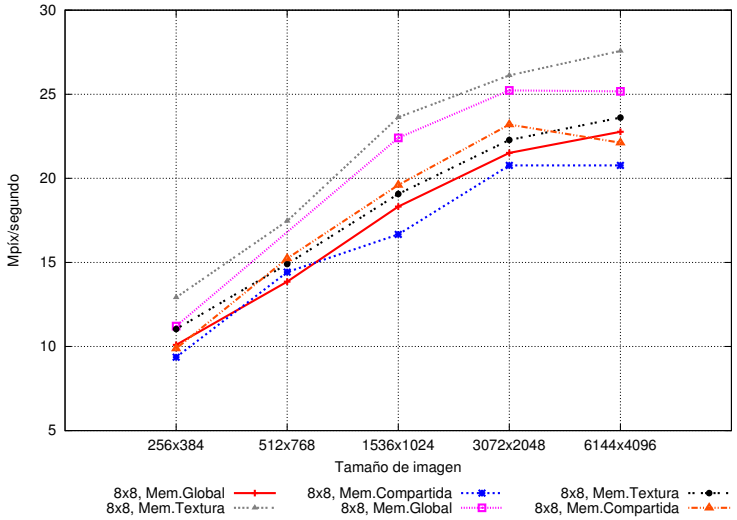


Figura 6.20: Rendimiento en megapíxeles por segundo procesados usando diferente tamaño de bloques para distintos tamaños de imagen en la arquitectura Geforce GT 120.

6.4.2 Definición de los *kernels*

Definir uno, dos o tres *kernels* implica siempre la separación de la primera etapa de detección con la segunda de filtrado a través de una sincronización, debido a que el AMF se calcula con los píxeles no corruptos de la ventana, de esta manera se leen los valores actualizados de los vecinos en el paso de filtrado. Cuando se declara un solo *kernel* (donde se encuentran definas ambas etapas del proceso) se tiene que especificar explícitamente la sincronización de las dos etapas a través de la instrucción *__synchilos*(). Cuando se declara dos o tres *kernels*, el *kernel* de filtrado no inicia hasta que todos los hilos del *kernel* de detección han finalizado.

La utilización de un solo *kernel* donde se especifica todo el proceso (detección y corrección) sin algún tipo de sincronización no es lo recomendado, pues la calidad varía en cada ejecución debido a que en cierto momento el hilo lee un píxel que ya está etiquetado por otro hilo, y en otros casos no se ha dado la actualización. En otras palabras, la calidad baja y el tiempo computacional no disminuye si se compara con la utilización de dos o tres *kernels*. Los resultados los podemos visualizar en la figura 6.21. Como podemos observar, el rendimiento obtenido utilizando un *kernel* sin sincronización en los dos pasos es peor que cuando se utiliza dos kernels con sincronización entre los dos pasos. Esto es debido a que existen muchos conflictos de accesos a memoria entre los hilos, es decir, mientras unos hilos están intentando leer a cierta posición de memoria, hay otros hilos que quizá aún están en la primera etapa y están intentando también escribir para etiquetar su píxel, esto genera conflictos de memoria.

Recordemos que los algoritmos 10 y 11 explicados en el capítulo 5 conforman la etapa de detección en dos fases. En concordancia con ello, se ha implementado respectivamente cada fase en dos *kernels*. El algoritmo 12 que fusiona las dos fases de la detección en una sola, se ha implementado en el *kernel* 1. La definición de este *kernel* es con la métrica fuzzy, pero puede adaptarse para la distancia euclídea o coseno del ángulo, que corresponden a los algoritmos 13 o 14 respectivamente. El algoritmo 15 corresponde a la etapa de filtrado y se ha implementado en el *kernel* 2.

El *kernel* de corrección no puede comenzar hasta que el/los de detección han finalizado. Los hilos se sincronizan antes y después del *kernel* de filtrado. El *kernel* de detección que corresponde al algoritmo 12 se describe en el *kernel* 1 con acceso a los datos a memoria global. En este algoritmo la llamada $FuzzyMetricFunction\ (a_{pix}, b_{pix})$ es la función para calcular la métrica fuzzy entre el píxel central x_i y píxeles

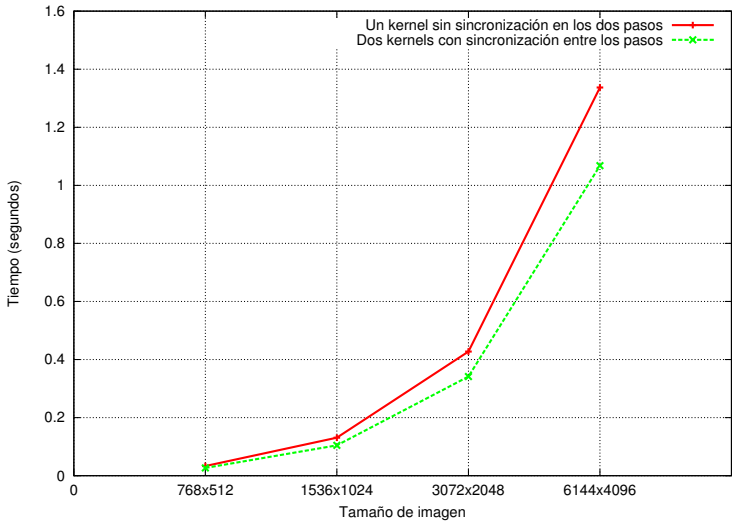


Figura 6.21: Rendimiento computacional (segundos) con uno o dos *kernels*.

vecinos de la ventana. Todos los píxeles después de ejecutar el *kernel* de detección están etiquetados como corruptos o no corruptos. Inicialmente el cuarto byte está inicializado con el valor 0, que indica que esta libre de ruido.

***Kernel* 1** KernelDetecciónMemGlobal

Entrada: Imagen ruidosa (píxeles de la imagen), m, k, d, $n = 3$.
Resultado: Imagen ruidosa con píxeles etiquetados como corruptos o no corruptos.

1: **El hilo** *i que se corresponde con el píxel* x_i **hace:**
2: *col*← ID del hilo global en columna;
3: *row*← ID del hilo global en renglón;
4: byte *padding* (p) se inicializa en cero;
5: $a_{pix} \leftarrow$ valor de RGB en x_i desde *row* y *col*; //lectura a memoria global.
6: **for** $i \leftarrow -1$ to 1 **do**
7: **for** $j \leftarrow -1$ to 1 **do**
8: $b_{pix} \leftarrow$ valor RGB del vecino (i,j) de *row* y *col*; //lectura a memoria global.
9: $dist \leftarrow FuzzyMetricFunction$ (a_{pix}, b_{pix});
10: **if** $dist \geq d$ **then**
11: píxel (b_{pix}) $\in$ P(a_{pix},d);
12: $cardinalidad \leftarrow cardinalidad + 1$;
13: **end if**
14: **end for**
15: **end for**
16: **if** $cardinalidad < (m{+}1)$ **then**
17: a_{pix} en $p \leftarrow$ valor 1; //píxel corrupto.
18: **end if**
19: **end**

El *kernel* de filtrado con los accesos a memoria global se describe en el *kernel* 2 que corresponde al algoritmo 12. En este algoritmo la llamada $MeanFunction$ ($bpixOK$), es la función para calcular el AMF, $bpixOK$ es un arreglo de píxeles no corruptos de x_i.

***Kernel* 2** KernelFiltradoMemGlobal

Entrada: Imagen ruidosa con los píxeles etiquetados como corruptos y no corruptos, $n = 3$.
Resultado: Imagen Filtrada.

```
1:  col← ID del hilo global en columna;
2:  row← ID del hilo global en renglón;
3:  a_pix ← valor RGB de x_i desde row y col; //lectura a memoria global.
4:  if a_pix en p ==1 then
5:    for i ← −1 to 1 do
6:      for j ← −1 to 1 do
7:        b_pix ← valor RGB del vecino (i,j) de row y col; //lectura a memoria global.
8:        if b_pix en byte p == 0 then
9:          b_pixOK ← b_pix;
10:       end if
11:     end for
12:   end for
13:   mean ← MeanFunction (b_pixOK);
14:   a_pix ← mean;
15: end if
```

Los hilos correspondientes al píxel corrupto en el "KernelFiltrado", aplican el proceso de filtrado por sustitución. El valor que se reemplaza es el resultado del cálculo del AMF de los píxeles no corruptos de la ventana W. Los píxeles no corruptos continúan su valor original, y aunque estos hilos no tienen actividad en esta etapa, esperan a que los demás terminen su ejecución.

Cuando se accede a los datos a través de la memoria de texturas, se realizan los siguientes cambios a los dos *kernels* anteriores:

La sentencia cinco del *kernel* 1 y la sentencia tres del *kernel* 2 que corresponde a la lectura del píxel central de la ventana, se reemplaza por:

$a_{pix} \leftarrow$ valor de RGB en x_i desde row y col; //lectura a memoria del *device* a través de la memoria de textura.

La sentencia ocho del *kernel* 1 y la sentencia siete del *kernel* 2 que corresponde a los accesos de cada uno de los vecinos de x_i en W, se reemplaza por:

$b_{pix} \leftarrow$ valor RGB del vecino (i,j) de row y col; //lectura a memoria del *device* a través de la memoria de textura.

Cuando se accede a los datos a través de la memoria compartida, se realizan los cambios a los *kernels* anteriores en las sentencias siguientes:

La sentencia cinco del *kernel* 1 y la sentencia tres del *kernel* 2 que corresponde a la lectura del píxel x_i, se reemplaza por:

$a_{pix} \leftarrow$ cargar de la memoria del *device* a la memoria compartida el valor de RGB en x_i de row y col;

La sentencia ocho del *kernel* 1 y la sentencia siete del *kernel* 2 que corresponde a los accesos de cada uno de los vecinos de x_i en W, se reemplaza por:

$b_{pix} \leftarrow$ cargar de la memoria del *device* a la memoria compartida el valor RGB del vecino (i,j) de row y col.

6.4.3 Análisis Computacional

En este apartado se presenta el análisis computacional con y sin optimización de los accesos a la memoria de la GPU que se presentó en el trabajo (Sánchez y col. 2011c).

Denominamos p_1 al paso de detección y p_2 al paso de filtrado. Dos de los enfoques para el acceso a los datos son: acceso a la memoria global sin texturas y el acceso a la memoria global con texturas. El análisis computacional de ambos enfoques se presenta a continuación:

a) Acceso a la memoria global sin texturas (sT):

- p_1: 3 x $n \times n$ acceso de sólo lectura (sTr) para obtener los valores RGB de los píxeles en W (sentencias 1-4 del *kernel* 1) y un acceso a escritura (sTw) para escribir el valor del cuarto byte de RGB para indicar si el píxel es o no corrupto (sentencia 17 en el *kernel* 1). La expresión del coste computacional de acceso a los datos sin texturas para el paso de detección está dada por,

$$sTp_1 = (3 \times n^2)sTr + sTw \tag{6.1}$$

  En este paso se realiza el cálculo de la distancia euclídea (dE), distancia fuzzy (dF) o distancia del ángulo del coseno (dC) para

determinar los píxeles que forman parte del *peer group* (sentencias 6-15 del *kernel* 1).

El coste computacional de la distancia euclídea es: dE =$\eta\times$(tres sustracciones, dos adiciones, tres productos y una raíz cuadrada), donde $\eta = n \times n - 1$ correspondiente a los píxeles vecinos de x_i.

Para el caso de la métrica fuzzy el coste es: dF= $\eta\times$(tres comparaciones, seis sumas, tres divisiones y dos productos).

El coste computacional de la distancia del ángulo del coseno es: dC= $n \times n \times$ (tres productos y dos sumas) + $\eta\times$((tres productos y dos sumas) + una división + un producto).

- p_2: $n \times n$ accesos de lectura para obtener el valor del cuarto byte del conjunto RGB para todo W etiquetado como corrupto o no (sentencia tres y cuatro del *kernel* 2); 3 x (ηp_{nc})[1] accesos de lectura para obtener los valores RGB de todos sus vecinos que no son corruptos para el cálculo del AMF, y tres accesos de escritura para escribir el nuevo valor RGB. El número total de accesos sería,

$$sTp_2 = (n^2)sTr + (3 \times \eta p_n c)sTr + 3sTw \tag{6.2}$$

En este paso se calcula la media aritmética de ηp_{nc} y su coste es: AM_c = (tres adiciones y tres divisiones) $\times p_{nc}$

El coste total de accesos a memoria global para cada distancia es:

$$sT_E = sTp_1 + dE + sTp_2 + AM_c \tag{6.3}$$

$$sT_F = sTp_1 + dF + sTp_2 + AM_c \tag{6.4}$$

$$sT_C = sTp_1 + dC + sTp_2 + AM_c \tag{6.5}$$

b) Acceso a la memoria global con texturas (T):

- p_1: 3 x n^2 accesos de sólo lectura a la memoria de textura (Tr) para obtener los valores RGB de los píxeles W, un acceso de escritura a la memoria global (sTw) para obtener el valor del cuarto byte del conjunto RGB para indicar si el píxel es o no corrupto. Por lo tanto, el acceso a los datos con memoria de texturas para el paso de detección es,

$$Tp_1 = (3 \times n^2)Tr + 1sTw \tag{6.6}$$

[1] p_{nc} son los píxeles no corruptos

El coste computacional de la distancia euclídea, fuzzy o coseno del ángulo es la misma que se presentó en el paso 1 del acceso a memoria global sin textura.

- p_2: $n \times n$ accesos de lectura a la memoria de textura para obtener el valor de corrupto o no corrupto, que contiene el cuarto byte del conjunto RGB de W, $3 \times (\eta p_n c)$ accesos a la memoria de textura para obtener los valores RGB de todos sus vecinos que no son corruptos para el cálculo de AMF, y tres accesos de escritura para escribir el nuevo valor RGB en la memoria global, es decir,

$$Tp_2 = (n^2)Tr + (3 \times p_n c)Tr + 3sTw \tag{6.7}$$

  El coste computacional de la media aritmética es la misma que se presentó en el paso uno de la memoria global.

El costo total del uso de memoria de acceso global con textura es,

$$T = Tp1 + dE + Tp2 + AM_c \tag{6.8}$$

Los costes presentados, son válidos cuando el número de píxeles que contiene la imagen (N_{pic}) es menor o igual al número de hilos (τ) que se liberan; en caso contrario, algunos hilos que se lanzan tienen que trabajar más de una vez y el coste computacional vendrá dado por,

$$(sT|T) \times \beta \tag{6.9}$$

donde β es la parte entera por exceso del cociente $(N_{pic} \div \tau) + 1$.

Los costes totales presentados corresponden con los pasos que se realizan cuando los datos están en la memoria de la GPU. Otro coste computacional es la preparación y transferencia de los datos de la CPU a la GPU, pero dependerá de la arquitectura de la GPU/CPU que se esté utilizando en su momento.

6.4.4 Optimización de las implementaciones

El tiempo computacional en GPU dedicado por cada *kernel*, al ejecutarse los algoritmos 10, 11 y 15, se muestran en la tabla 6.11. Como puede observarse, el coste computacional de la fase uno del paso de detección (P1f1) es menor que el de la fase dos (P1f2) para todos los casos, además el coste computacional se reduce considerablemente al acceder a los datos con la memoria de texturas. Análisis presentado en (Sánchez, Vidal y Bataller 2010).

Tabla 6.11: Tiempo de procesamiento en GPU para los *kernels* de detección(2 fases) y filtrado.

	Tamaño de imagen	Detección Fase 1 (P1f1)	Detección Fase 2 (P1f2)	Filtrado (P2)
GPU RGB	256	0.59	1.77	1.90
	512	3.53	8.48	8.55
	1024	17.41	34.61	35.12
	2048	77.56	175.03	156.08
GPU RGBp	256	0.44	1.29	1.36
	512	2.33	4.61	5.64
	1024	11.03	17.40	22.58
	2048	55.08	93.97	101.91
GPU Textura	256	0.43	0.63	0.73
	512	1.69	2.56	2.88
	1024	7.34	10.22	11.76
	2048	40.36	44.05	52.51

Tabla 6.12: Tiempo de procesamiento en milisegundos de la ejecución de la imagen *world*, usando acceso a la memoria global.

	Detección	Filtrado	Total GPU	Proceso Total
5 %	53.09	36.27	89.36	111.21
10 %	64.81	57.26	122.08	143.93
20 %	64.79	75.65	140.45	162.29
30 %	64.80	82.25	147.06	168.91

Se realizó el proceso de detección y filtrado a la imagen *world*, con 20 % de ruido uniforme. Los mejores valores utilizados después de un estudio heurístico para los parámetros m y d son $m = 2$ y $d = 49$ con la métrica euclídea en el *peer group*.

En la tabla 6.12 se muestra el tiempo en milisegundos de la ejecución de ambos procesos: detección y filtrado; además se muestra el tiempo total obtenido (incluye la transferencia entre CPU y GPU) para los algoritmos 12 y 15 utilizando el acceso a la memoria global.

En la tabla 6.13 se muestran los resultados de la ejecución de las dos etapas utilizando la memoria de texturas. Como puede observarse, la paralelización es más notoria. Estudio presentado en (Sánchez y col. 2011c).

Tabla 6.13: Tiempo del procesamiento en milisegundos de la ejecución de la imagen *world* con acceso a la memoria global a través de texturas.

	Detección	Filtrado	Total GPU	Proceso Total
5 %	23.15	31.64	54.80	76.63
10 %	23.15	38.40	61.55	83.48
20 %	23.13	46.65	69.79	91.62
30 %	23.13	51.58	74.72	96.57

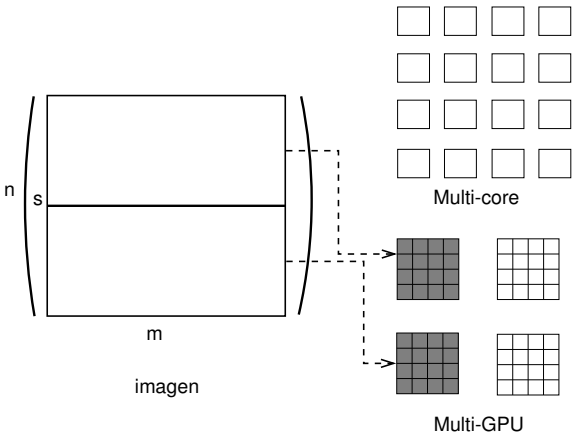
En GPU, para ganar coalescencia de acceso a los datos en memoria, la estrategia que se ha implementado en esta tesis es lanzar tantos hilos como píxeles tiene la imagen. Sin embargo, esta estrategia con la opción optimizada del algoritmo 12 (descrito en la sección 6.2), genera conflictos de acceso de lectura y escritura y además puede darse el caso de una clasificación del píxel con datos donde aún no están todas sus distancias acumuladas y por lo tanto generar un filtrado erróneo.

Por lo anterior, se ha considerado que no es recomendable su implementación de la versión optimizada en accesos y cálculos en GPU. Las GPUs son particularmente útiles realizando cálculos y no son recomendables para muchos accesos a memoria y más aún si no se cuenta con una buena estrategia de acceso.

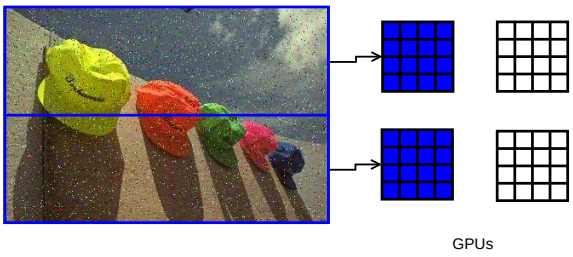
Procesamiento en multi-GPU

La distribución de los píxeles entre las GPUs disponibles se muestran en la figura 6.22. La imagen se dividió entre el número de GPUs menor o igual a las disponibles, dependiendo de las GPUs con las que se trabaja. Por ejemplo, la imagen se divide en dos partes si se desea procesar los datos en dos GPUs, donde cada parte se procesa por una GPU diferente.

Los resultados de dividir la carga computacional entre las GPUs disponibles se muestra en la tabla 6.14 para diferentes tamaños de la imagen *Caps* con ruido impulsivo del 10 %. Paralelizar una imagen con tamaño menor a 1536x1024 es mejor cuando se utiliza una GPU. Distribuir una imagen de tamaño menor que 3072x2048 en cuatro GPUs, lleva a un tiempo mayor que distribuir la imagen en dos GPUs. Con esto se puede deducir que, con imágenes de gran tamaño la distribución óptima se obtiene usando cuatro GPUs, pero no para tamaños de imágenes más pequeñas que 3072x2048. Análisis presentado en (Sánchez y col. 2011b).



(a)



(b)

Figura 6.22: Distribución de la imagen en 2 GPUs, ningún core utilizado.

Tabla 6.14: Paralelización de la imagen *Caps* en GPUs. Tiempo en segundos

	# GPUs		
Tamaño de la imagen	1	2	4
96x64	0.0009	0.0357	0.1058
192x128	0.0022	0.0363	0.1061
384x256	0.0071	0.0391	0.1074
768x512	0.0267	0.0483	0.1117
1536x1024	0.1047	0.0858	0.1306
3072x2048	0.3415	0.2357	0.2019
6144x4096	1.0681	0.8144	0.4734

La figura 6.23 muestra el rendimiento en megapíxeles por segundo cuando se procesa la imagen *building* con 15 % de ruido impulsivo en más de una GPU. Se consideró únicamente los cálculos realizados dentro de la GPU, es decir, sin tener en cuenta la transferencia CPU-GPU. Los mejores resultados se presentan cuando todas las GPUs se utilizan (cuatro en ese caso para el equipo de altas prestaciones DSICMAC1). Estudio presentado en (Sánchez, Vidal y Bataller 2011).

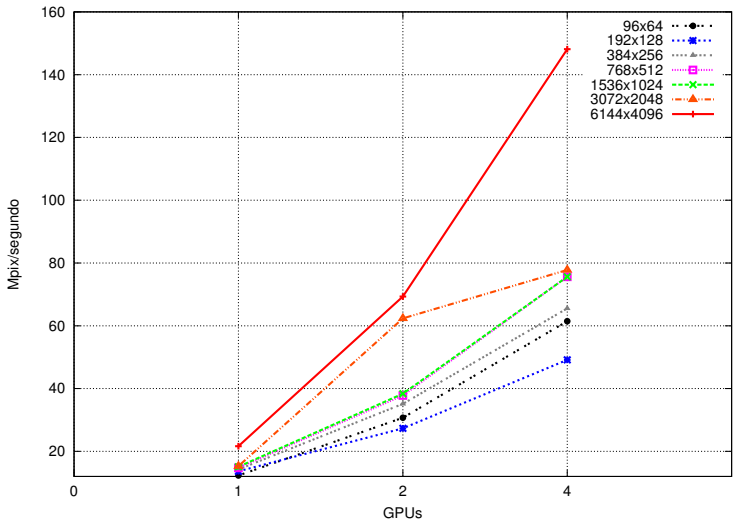


Figura 6.23: Rendimiento en megapíxeles por segundo procesados por los *kernels* de detección y filtrado (sin tomar en cuenta el tiempo de transferencia).

La figura 6.24 muestra los resultados de la ejecución en megapíxeles por segundo, incluyendo la transferencia de datos entre RAM-GPU. Se puede observar que para imágenes más pequeñas que 1536x1024 píxeles, es mejor usar una sola GPU porque el tiempo empleado en las transferencias no se compensa usando más GPUs. De lo contrario, para tamaños de imágenes más grandes que 1536x1024, usando más GPUs mejora el rendimiento independiente de las transferencias de la memoria. En otras palabras:

- Para tamaño de imágenes menores que 768x512, es mejor usar una GPU.
- Para tamaño de imágenes entre 768x512 y 3072x2048, es mejor usar dos GPUs.
- Para tamaño de imágenes mas grande que 6144x4096, es mejor usar cuatro GPUs.

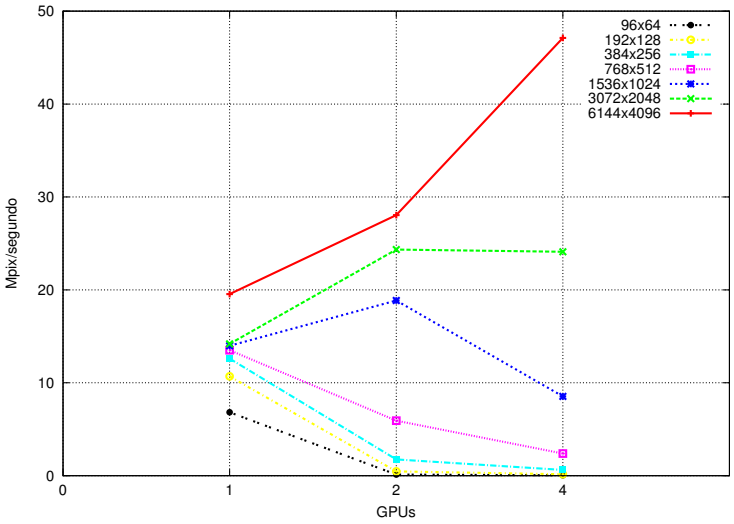


Figura 6.24: Megapíxeles/segundo procesados por los *kernels* de detección y filtrado (total del procesamiento que incluye transferencia).

La tabla 6.15 muestra el mejor procesamiento en megapíxeles por segundo que se presenta para cada tamaño de imagen en GPUs. Análisis presentado en (Sánchez, Vidal y Bataller 2011).

Tabla 6.15: Rendimiento en megapíxeles por segundo para el procesamiento de la imagen en Multi-GPU. Mejores valores obtenidos para cada tamaño de imagen.

Tamaño de la imagen	Multi-GPU
96x64	6.83
192x128	10.69
384x256	12.60
768x512	13.51
1536x1024	18.86
3072x2048	24.34
3072x2048	24.10
6144x4096	47.12

Se comparó el acceso a los datos utilizando la memoria de textura y la memoria compartida usando 1, 2 y 4 GPUs. La tabla 6.16 resume los resultados obtenidos. Como puede observarse, la memoria de texturas supera aproximadamente en un 12 % de mejora cuando se utiliza 4 GPUs. Estudio presentado en (Sánchez, Vidal y Bataller 2011).

Tabla 6.16: Rendimiento en megapíxeles por segundo al acceder a la memoria de la GPU a través de la memoria de texturas compartida.

Tamaño de la imagen	Memoria de Texturas	Memoria Compartida	GPU
6144x4096	20.6599	17.3378	1
6144x4096	30.5744	29.4889	2
6144x4096	47.1182	41.3707	4

Comparativa en diferentes arquitecturas GPU

Se analizaron por un lado los algoritmos 12 y 15 en tres arquitecturas paralelas diferentes, respecto al tiempo de acceso a la memoria global y el uso de la memoria de texturas. Se propuso una estrategia para maximizar el ancho de banda de la memoria de la GPU. El rendimiento que se obtiene al ejecutar el algoritmo en la arquitectura Tesla M2050 se compara con los resultados obtenidos de las arquitecturas GeForce GT 120 y GeForce 9800 GX2. La imagen que se utilizó fue *Girl* con 10 % de ruido impulsivo. Estudio realizado en (Sánchez y col. 2012c).

La figura 6.25 compara el rendimiento en megapíxeles por segundo obtenido en las tres arquitecturas de GPU utilizadas en esta tesis y para los tres tipos de acceso a los datos en memoria de la GPU: directamente a memoria global, texturas y compartida.

El rendimiento de la GPU GeForce GT 120 es mejor cuando se utiliza la memoria de textura en comparación con las otras dos formas de acceso. En la imagen con tamaño 6144x4096 hay una pequeña disminución del rendimiento cuando se accede a través de la memoria compartida, debido a la sobrecarga asociada en las transferencias. Todos los hilos obtienen los datos de la memoria del *device* a la memoria compartida.

El rendimiento obtenido al paralelizar con la GPU GeForce9800 GX2 con la memoria de textura es la mejor opción para acceder a los datos seguidos por el acceso a través de la memoria compartida. En tamaños 6144x4096 con memoria global, la memoria de textura tiene una disminución del rendimiento, debido a la sobrecarga generada por las transferencias. En este caso la velocidad de reloj del procesador con el ancho de banda está fuertemente acoplado, lo que se refleja en la diferencia entre la memoria global y textura.

La memoria de textura proporciona mejor rendimiento cuando se utiliza la tarjeta gráfica Tesla M2050. El uso del acceso a la memoria compartida y el acceso a la memoria global, obtienen resultados muy

similares. Se puede observar que que en la GPU Tesla M2050, independientemente del tipo de acceso, el rendimiento del tamaño de la imagen 512x768 es menor comparada con los otros tamaños de imagen. Esto se debe a que no hay datos suficientes para que todos los cores trabajen.

Se puede observar que obviamente la arquitectura Fermi-Tesla supera el rendimiento computacional comparado con las otras dos tarjetas para todos los tamaños de imagen. Si hay más ancho de banda, los accesos son más rápidos.

Para las tres arquitecturas el rendimiento es mejor con la memoria de textura para cualquier tamaño de imagen. Con estos resultados, podemos decir que el uso de la memoria de textura es la mejor opción para todas las formas de acceso a los datos.

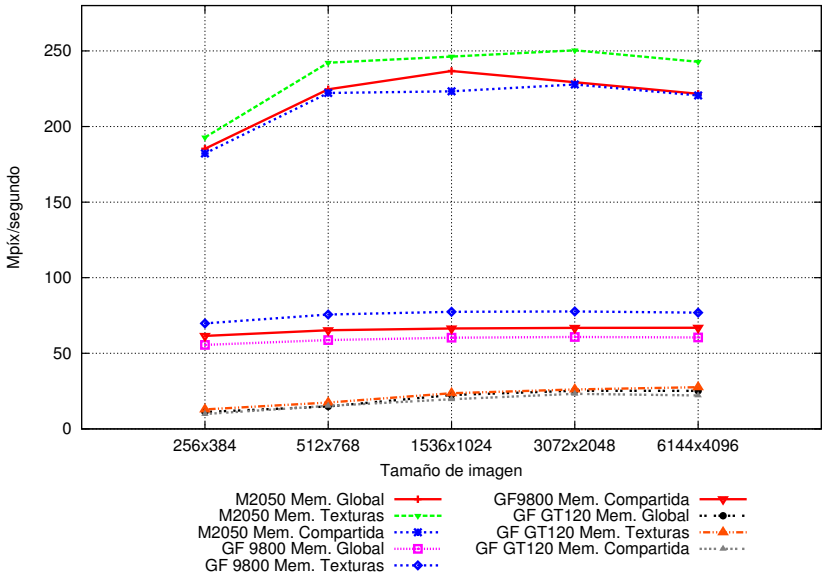
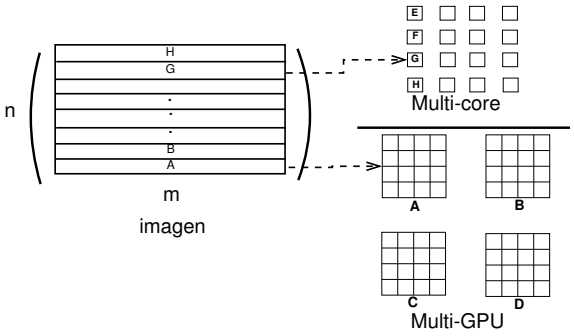


Figura 6.25: Rendimiento en megapíxeles por segundo procesados para las etapas de detección y filtrado usando diferente tamaño imagen con tres modelos de tarjetas gráficas.

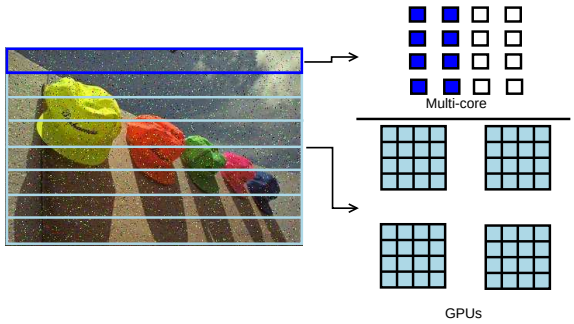
6.5 Implementaciones híbridas CPU-GPU

Se analizó la forma más conveniente de distribuir los píxeles en *multicores* y multi-GPU cuando se utilizan ambas arquitecturas en combinación, con la finalidad de obtener buen rendimiento computacional y aprovechar el hardware con el que se cuenta.

La distribución de los píxeles entre los *multicores* y *manycore* disponibles cuando se manejan ambas arquitecturas, se muestran en la figura 6.26. Los píxeles fueron repartidos entre los cores y GPUs disponibles por grupos de filas de píxeles. Por ejemplo, la figura 6.26*b* muestra la partición de la imagen en ocho bloques de filas, que se reparten en cuatro GPUs y ocho cores, siete y un bloque respectivamente.



(a)



(b)

Figura 6.26: Distribución de la imagen en 4 GPUs y 8 cores.

Tabla 6.17: Rendimiento en megapíxeles por segundo para el procesamiento en multi-GPU, y en implementación híbrida.

Tamaño de la imagen	Multi-GPU	*multicore* y Multi-GPU
96x64	6.83	8.3706
192x128	10.69	15.5446
384x256	12.60	20.0130
768x512	13.51	37.1379
1536x1024	18.86	42.3953
3072x2048	24.34	43.7819
3072x2048	24.10	69.9105
6144x4096	47.12	68.4600

6.5.1 Resultados de la implementación CPU-GPU

La tabla 6.17 muestra los mejores resultados paralizando la imagen *building* en multi-GPU (ver tabla 6.15) comparado con la combinación de *multicore* y multi-GPU. Se puede observar que para cualquier tamaño de imagen, es mejor usar la combinación de ambos, superando al procesamiento en multi-GPU. Análisis presentado en (Sánchez, Vidal y Bataller 2011).

6.5.2 Carga computacional en la implementación CPU-GPU

La tabla 6.18 muestra la mejor combinación de GPU y cores CPU para tamaño de imagen *building* con 15 % de ruido impulsivo, en la que se proporciona mayor rendimiento diferente. Se presenta la distribución óptima de la carga computacional en la GPU y por consecuencia el resto se procesa en *multicore*. Se puede observar que, al incrementar el tamaño de imagen, incrementa la carga computacional en la GPU y se utilizan más GPUs. El mejor tiempo se muestra cuando más carga se asigna a la GPU dejando menos a los cores. Estudio presentado en (Sánchez, Vidal y Bataller 2011).

La tabla 6.19 muestra el rendimiento de la imagen *Caps* con 10 % de ruido impulsivo, usando los cores disponibles en el equipo de altas prestaciones DSICMAC1 y 4 GPUs para la distribución de 1/8 de la imagen en cores y 7/8 en GPU. El uso de cero cores implica que toda la carga esta en 4GPUs. Como puede observarse en los resultados, la paralelización realizada usando el hardware de cores y GPUs, proporciona mejores resultados que la paralelización realizada únicamente en cuatro GPUs. Análisis presentado en (Sánchez y col. 2011b).

Tabla 6.18: Carga computacional de la imagen asignada para el procesamiento en GPU y cores.

Tamaño de la imagen	GPUs	*core*s	Proporción de imagen en GPU
96x64	1	16	1/4
192x128	1	11	3/8
384x256	1	7	1/2
768x512	1	9	3/4
1536x1024	2	9	3/4
3072x2048	2	9	3/4
3072x2048	4	11	7/8
6144x4096	4	7	7/8

Tabla 6.19: Rendimiento computacional (segundos) para 1/8 de imagen paralelizada en *multicore* y 7/8 en 4 GPUs.

core	3072x2048	6144x4096
0	0.2019	0.4734
1	0.4625	1.8448
2	0.2338	0.9331
3	0.1568	0.6251
4	0.1201	0.4747
5	0.1053	0.3877
6	0.0911	0.3479
7	0.0926	0.3560
8	0.0851	0.3241
9	0.1942	0.3423
10	0.0888	0.3605
11	0.0973	0.3396
12	0.0900	0.3566
13	0.1023	0.3726
14	0.0979	0.3877
15	0.1035	0.3692
16	0.0980	0.3916

El uso de más de tres cores con las cuatro GPUs proporciona mejor rendimiento computacional comparado con cuatro GPUs para el tamaño de imagen 3072x2048.

Para el tamaño de imagen 6144x4096 la mejor combinación de arquitectura para obtener el menor tiempo computacional ha sido con cuatro GPUs y a partir de cinco cores con 1/8 de carga computacional asignado a la GPU y 7/8 a la GPU, obteniendo el óptimo resultado para ocho cores, a partir de ahí el rendimiento disminuye, pero sigue siendo mejor que con cuatro GPUs.

6.6 Comparación de las implementaciones a nivel computacional

En esta sección vamos a presentar los resultados de las comparaciones referente al rendimiento computacional de las distintas implementaciones disponibles. En concreto en la sección 6.6.1 comparamos implementaciones secuenciales con implementaciones paralelas. Y en el apartado 6.6.2 comparamos las distintas implementaciones paralelas de las que disponemos (CPU, GPU, híbridas).

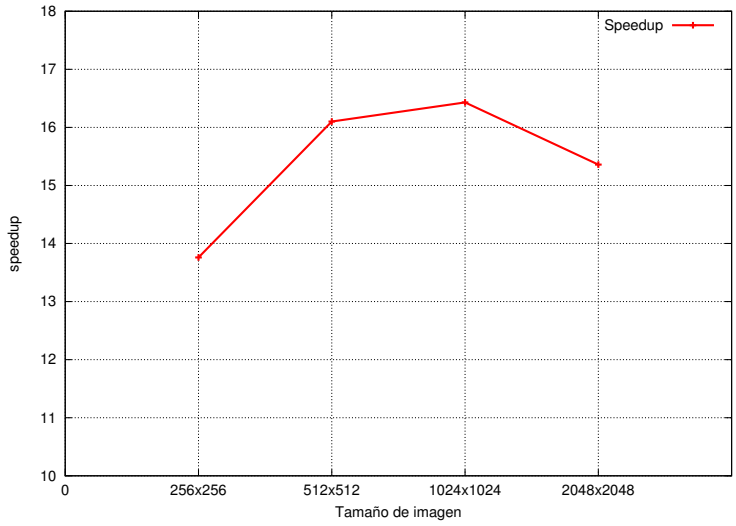
6.6.1 Secuencial versus paralela

En la tabla 6.20 y figura 6.27(a), se presenta la comparativa de la versión secuencial y la versión paralela en la GPU utilizando texturas y el formato de imagen RGBA para la imagen *Lenna* , con la métrica euclídea en el *peer group* y la sustitución del primer píxel no corrupto por el píxel que se está analizando (corrupto). Análisis presentado en (Sánchez y col. 2010).

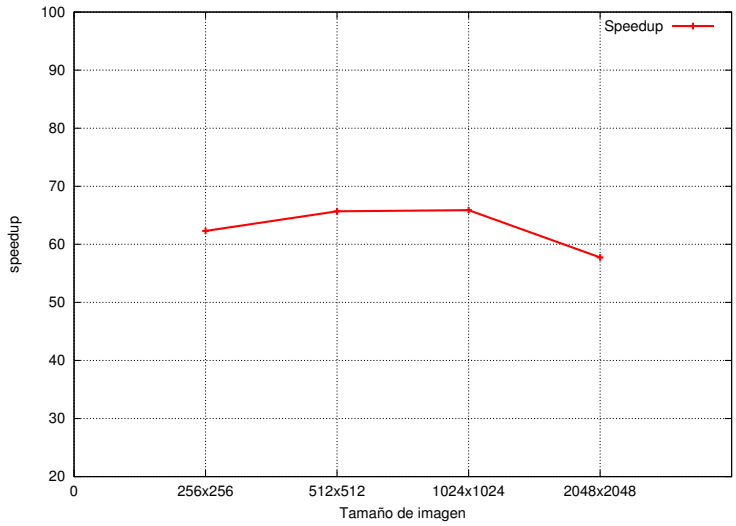
Tabla 6.20: *Speedup* alcanzado para la imagen *Lenna* con 5 % de ruido impulsivo para distintos tamaños de imagen

Imagen de entrada	CPU(ms)	GPU(ms) RGBA con texturas	Speedup
256	19.29	1.40	13.76
512	70.75	4.39	16.10
1024	277.06	16.86	16.43
2048	1096.94	71.42	15.36

Como se observa, en el peor de los casos la versión en GPU (paralela) es 13 veces más rápida que la secuencial, lo cual es un excelente resultado.



(a)



(b)

Figura 6.27: *Speedup* para distintos tamaños de imagen. a) CPU-GPU(RGBp con texturas), b) *Speedup* para la imagen *Lenna* con 10 % de ruido impulsivo.

La tabla 6.21 se muestra por un lado, la comparativa en milisegundos de los mejores resultados del procesamiento en GPU y CPU para diferente tamaño de imágenes y por otro lado, el *speedup* generado con estos resultados. El resultado del *speedup* también se muestra en la figura 6.27(b). Los resultados utilizan la métrica fuzzy en el *peer group* y el AMF como método de filtrado con la imagen de *Lenna* y 10 % de ruido impulsivo. El acceso a los datos en GPU se realiza a través de texturas.

Tabla 6.21: Rendimiento computacional en milisegundos para la ejecución en GPU y CPU. Resultado de *speedup*

Tamaño	GPU	CPU	Speedup
256	1.79	111.88	62.30
512	7.11	467.63	65.69
1024	29.33	1932.72	65.89
2048	136.92	7907.12	57.75

Como puede observarse, incluso el peor resultado para la versión de GPU es 57 veces más rápido que el secuencial, esto indica que es un excelente resultado. Estudio presentado en (Sánchez, Vidal y Bataller 2010).

El speedup generado al procesar la imagen *world* con y sin texturas, se presenta en la tabla 6.22. Esta misma tabla muestra una comparativa del tiempo computacional de la ejecución sin texturas (sT) y con texturas (T) para el acceso a los datos en GPU. Estos tiempos incluyen el tiempo de transferencia de los datos entre CPU-GPU. Como puede observarse, se llega a obtener una mejora de hasta un 12x-13x. Análisis presentado en (Sánchez y col. 2011c).

Tabla 6.22: *Speedup* alcanzado con la imagen *world* y 20 % de ruido impulsivo.

%Ruido	GPU sT	GPU T	Secuencial	Speedup sT	Speedup T
5 %	111.21	76.63	939.21	8.44	12.25
10 %	143.93	83.48	1035.61	7.19	12.40
20 %	162.29	91.62	1182.71	7.28	12.90
30 %	168.91	96.57	1284.46	7.60	13.30

Para un 5 % de ruido se obtiene una mejora del 91.84 %, en 10 % la ganancia es del 91.93 %, para el 20 % se obtiene una mejora del 92.25 %

y para un 30 % del 92.48 %. Se compara las versiones, secuencial y paralelo de la imagen *world*. La figura 6.28 muestra los resultados.

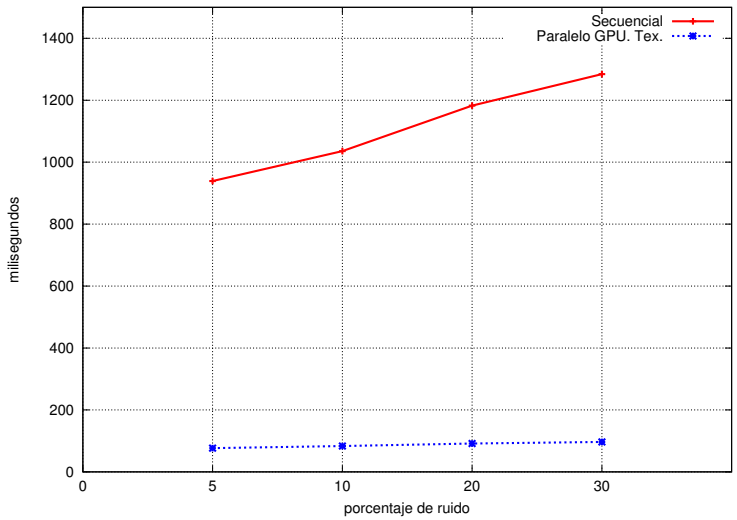


Figura 6.28: Comparativa del rendimiento computacional GPU vs CPU

Continuando con las comparativas, la tabla 6.23 muestra los tiempos entre la versión secuencial y la versión paralela en GPU con texturas para la imagen de *Lenna* con 5 % de ruido impulsivo. Como puede observarse, el *speedup* generado alcanza aproximadamente un 15x, conforme se incrementa el tamaño se mantiene el *speedup*, con lo que se concluye que el proceso es escalable. Estos resultados fueron analizados y presentados en (Sánchez y col. 2011a).

Tabla 6.23: *Speedup* alcanzado con la imagen *Lenna* para diferente tamaño de imagen con 5 % de ruido impulsivo.

Tamaño	Texturas con GPU	CPU	Speedup
128	0.87	8.17	9.39
256	2.39	31.84	13.32
512	8.54	126.96	14.87
1024	33.08	491.20	14.85
2048	136.14	1961.51	14.41
4096	545.59	7895.66	14.47

En la tabla 6.24 se muestra una comparativa de los resultados obtenidos al procesar la imagen de *Lenna* con 10 % de ruido impulsivo en

Tabla 6.24: Rendimiento computacional en milisegundos para imagen *Lenna* procesada en CPU y GPU con *cudaMallocHost*

	256	512	1024	2048
CPU	19.29	70.75	277.06	1096.94
GPU	3.84	23.26	94.68	412.92

CPU y GPU. En GPU se usa la función *cudaMallocHost* y texturas. Se utiliza la métrica fuzzy en el *peer group* y para la corrección el primer píxel de la ventana que no es corrupto se reemplaza por el central que se está analizando. Análisis presentado en (Sánchez y col. 2010). También se puede deducir de la tabla 6.24, que en todos los casos la ganancia obtenida de utilizar GPU frente a CPU es mayor del 62 %.

En la figura 6.29 se muestra la comparativa del coste computacional obtenido entre la versión secuencial y paralela para la implementación de los algoritmos 12, 13 y 15 en una imagen de mamografía. Estos algoritmos utilizan la métrica fuzzy (M) y la distancia euclídea en el *peer group* y en el paso de filtrado se utiliza el AMF de los no corruptos, análisis presentado en (Sánchez, Vidal y Bataller 2012). Como puede observarse, la mayor cantidad de píxeles por segundo se

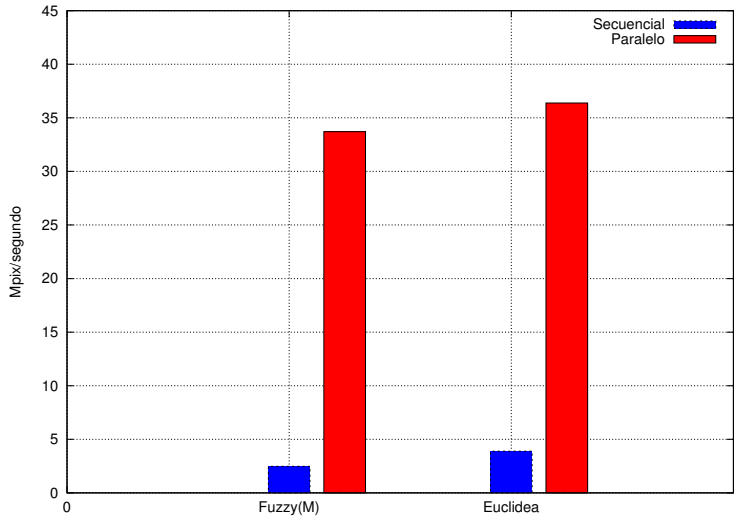


Figura 6.29: Comparación de los filtros paralelos y secuenciales de una imagen mamográfica con ruido impulsivo del 20 %.

procesan cuando se utiliza la GPU independientemente de la métrica utilizada.

6.6.2 Comparaciones entre las implementaciones: secuenciales, paralelas en CPU, GPU e híbridas.

La comparación de la imagen *Caps* con los mejores resultados obtenidos de forma secuencial, paralela en *multicore*, paralela en GPU y paralela utilizando cuatro GPUs se muestra en la tabla 6.25.

Tabla 6.25: Mejores tiempos computacionales en segundos de la ejecución en diferentes arquitecturas para la imagen *Caps*.

Tamaño de imagen	CPU	*multicore*	GPU	Multi-GPU(4GPUs)
96x64	0.0037	0.0009	0.0009	0.1058
192x128	0.0148	0.0024	0.0022	0.1061
384x256	0.0601	0.0083	0.0071	0.1074
768x512	0.2381	0.0274	0.0267	0.1117
1536x1024	0.9367	0.0986	0.1047	0.1306
3072x2048	3.7023	0.3817	0.3415	0.2019
6144x4096	14.7825	1.4790	1.0681	0.4734

Estos resultados son los mejores obtenidos para cada tamaño y arquitectura empleada (ver tablas 6.1 y 6.14). El tiempo obtenido por una GPU es similar al obtenido al paralelizar a través de 16 cores, pero si el tamaño de la imagen incrementa, entonces la diferencia de tiempo incrementa. Como puede observarse, para el menor tamaño testeado es preferible utilizar *multicore*. A partir de la imagen 192x128 la ejecución en GPU empieza a obtener mejor rendimiento que de manera secuencial o con *multicores*. La utilización de multi-GPU obtiene mejores resultados para tamaños de imágenes mayores o iguales a 3072x2048. En la tabla 6.26 se resume los mejores resultados en cada arquitectura y se compara con la implementación híbrida para los dos tamaños de imagen mayores testados. La mejor carga computacional para cada tamaño se obtiene de la tabla 6.19. Como puede observarse, la implementación híbrida reduce en un 57 % a la implementación en cuatro GPUs para un tamaño de 3072x2048 y de un 31.54 % para el tamaño de imagen 6144x4096.

El mejor tiempo para la imagen 3072x2048 fue de 0.3817 segundos con 16 cores y de 0.2019 segundos utilizando cuatro GPUs. Estos tiempos han sido reducidos distribuyendo la carga computacional entre las dos arquitecturas, obteniendo por ello 0.0851 seg.

Tabla 6.26: Mejores tiempos computacionales en segundos de la ejecución en core de la imagen *Caps*.

Tamaño de imagen	CPU	Multi-core	GPU	Multi-GPU(4GPUs)	CPU-GPU
3072x2048	3.7023	0.3817	0.3415	0.2019	0.0851
6144x4096	14.7825	1.4790	1.0681	0.4734	0.3241

Para el tamaño de imagen 6144x4096 el mejor tiempo en *multicore*s se presentó utilizando 16 cores con 1.4790 segundos y 0.4734 segundos con cuatro GPUs.

El menor coste computacional de GPUs (tabla 6.1) y 16 cores-CPU (6.14), se presenta cuando se utilizan las GPUs, no necesariamente con el tiempo de ejecución de las cuatro GPUs, como es el caso de las imágenes cuyo tamaño es menor que 3072x2049. Se obtiene mayor rendimiento computacional para los dos últimos tamaños de imágenes utilizando cuatro GPUs frente a los 16 cores.

Para tamaños más grandes que 384x256, se obtienen mejores resultados cuando todos los cores se utilizan. Por el contrario, para tamaños pequeños es mejor usar ocho cores.

Para tamaños de imágenes menores que 768x512, es mejor usar una GPU. Para tamaños de imágenes entre 768x512 y 3072x2048, es mejor utilizar dos GPUs. Para tamaños de imágenes mas grande que 6144x4096, es mejor emplear cuatro GPUs.

La paralelización realizada usando cores CPU y GPUs proporciona mejores resultados que la paralelización realizada únicamente en 4 GPUs o con cores CPU, dicha combinación se lleva a cabo para disminuir los costes computacionales. En esta implementación, el mejor tiempo se muestra cuando más carga se asigna a la GPU dejando con menos carga a los cores. La mejor combinación de arquitectura para obtener el menor tiempo computacional fue con 4 GPUs y a partir de 5 cores con 1/8 de carga computacional asignado a la GPU y 7/8 a la GPU.

6.7 Comparación de los algoritmos a nivel de calidad

En esta sección comparamos la calidad obtenida por los distintos algoritmos estudiados en esta tesis. Hay que destacar que ateniéndose a la calidad, no es relevante la distinción entre sí la implementación en secuencial o paralela.

6.7.1 Evaluación de la calidad en imágenes en escala de grises

Se hizo un análisis de los algoritmos que utilizan el *peer group*, comparando tres métricas: dos fuzzy (M y G) y una euclídea, con la finalidad de lograr un buen balance entre la calidad y la velocidad. La imagen utilizada para este análisis es una mamografía con 5 %, 10 % y 20 % de ruido impulsivo. Estudio presentado en (Sánchez, Vidal y Bataller 2012).

Los resultados obtenidos de calidad usando dos filtros (mediana y media) se muestran en la tabla 6.27, sin tomar en cuenta la etapa de detección. La mejor calidad se presenta con el filtro de mediana para cualquier porcentaje de ruido.

La calidad obtenida del filtro de mediana y media utilizando el *peer group* para detectar el ruido impulsivo con dos métricas fuzzy y una euclídea, se muestra en la tabla 6.28. En este caso se considera todos los píxeles no corruptos de la ventana para el paso de filtrado. El filtro de media es mejor que el de mediana para cualquier métrica usada, excepto para 5 % de ruido. La calidad obtenida para las métricas son similares, siendo un poco mejor la métrica fuzzy (M).

La calidad obtenida de la media y mediana utilizando todos los píxeles de la ventana de filtrado (corruptos o no corruptos) comparada con las tres métricas para el *peer group*, se muestra en la tabla 6.29. La mediana es mejor que la media y los resultados de las tres métricas tienen resultados similares.

Tabla 6.27: Calidad obtenida considerando todos los píxeles de la ventana W para el paso de filtrado.

	5 %			10 %			20 %		
	PSNR	MAE	MSE	PSNR	MAE	MSE	PSNR	MAE	MSE
Media	26.97	6.26	130.62	23.84	10.60	268.76	20.23	18.01	617.20
Mediana	46.65	0.50	1.41	42.14	0.57	3.97	31.50	0.96	46.00

Comparando las tablas 6.28 y 6.29 para un 5 % y 10 % de ruido, se puede observar que los mejores resultados se obtienen cuando se usa la mediana y considerando todos los píxeles de la ventana para el paso del filtrado, además, el usar el paso de detección y después el filtrado es la mejor opción de eliminación del ruido. Con el mayor porcentaje de ruido presentado, el filtro media es óptimo al considerar únicamente los píxeles no corruptos en el paso de filtrado. Con las tres métricas que se usaron se puede obtener similares resultados, aunque un poco mejor con la métrica fuzzy (M).

Con la finalidad de determinar el coste computacional en cada etapa, se comparó el procesamiento en Megapíxeles/segundo para un 10 % y 20 % de ruido, la tabla 6.30 muestra los resultados. Como se puede observar, el coste computacional es más costoso cuando se usa la métrica fuzzy comparada con la métrica euclídea, el paso de filtrado es más costoso que el paso de detección y por último, el cálculo del promedio es más rápido que la media. La métrica fuzzy G es más rápida que la métrica M, porque G únicamente realiza cálculos aritméticos mientras que con M hay una comparación para calcular el valor máximo. Algunas de las imágenes obtenidas tras realizar el proceso de filtrado se muestran en la figura 6.30 para 20 % de ruido. Estudio presentado en (Sánchez, Vidal y Bataller 2012)

Si se quiere tener calidad y coste computacional al mismo tiempo, se debe ejecutar los dos pasos: detección y filtrado, y la métrica euclídea en el paso de detección proporciona mejores tiempos computacionales aunque la calidad es un poco inferior a la métrica fuzzy.

Para un porcentaje menor que el 10 % de ruido, la mejor elección es emplear la media de todos los píxeles (incluyendo los píxeles corruptos) en el paso de filtrado.

Tabla 6.28: Calidad obtenida considerando únicamente los píxeles no corruptos para el paso de filtrado.

		Fuzzy (M)			Fuzzy (G)			Euclídea		
		PSNR	MAE	MSE	PSNR	MAE	MSE	PSNR	MAE	MSE
5 %	Media_{nc}	47.64	0.06	1.12	47.42	0.07	1.18	47.42	0.07	1.18
	Mediana_{nc}	48.33	0.05	0.96	48.11	0.06	1.00	48.06	0.06	1.02
10 %	Media_{nc}	42.81	0.13	3.41	42.59	0.14	3.58	42.59	0.14	3.58
	Mediana_{nc}	42.59	0.12	3.58	42.36	0.13	3.78	42.36	0.13	3.78
20 %	Media_{nc}	35.52	0.46	18.24	35.48	0.48	18.43	35.48	0.48	18.43
	Mediana_{nc}	34.02	0.47	25.77	34.00	0.49	25.88	34.00	0.49	25.88

Tabla 6.29: Calidad obtenida considerando todos los píxeles de la ventana W para el paso de filtrado.

		Fuzzy (M)			Fuzzy (G)			Euclídea		
		PSNR	MAE	MSE	PSNR	MAE	MSE	PSNR	MAE	MSE
5 %	Media_a	35.09	0.82	20.15	35.09	0.83	20.16	35.08	0.83	20.97
	Mediana_a	50.91	0.05	0.53	50.48	0.06	0.58	50.56	0.06	0.57
10 %	Media_a	30.67	1.84	55.77	30.69	1.83	55.44	30.69	1.83	55.44
	Mediana_a	43.33	0.12	3.02	43.18	0.13	3.13	43.18	0.13	3.13
20 %	Media_a	24.29	5.80	241.90	24.38	5.72	237.37	24.38	5.72	237.37
	Mediana_a	31.98	0.52	41.20	31.93	0.54	41.74	31.93	0.54	41.74

Tabla 6.30: Rendimiento computacional en megapíxeles por segundo para un 10 % y 20 % de ruido impulsivo.

		Fuzzy (M)		Fuzzy (G)		Euclídea	
		Detección	Filtrado	Detección	Filtrado	Detección	Filtrado
10 %	Media_{nc}	170.90	84.04	196.37	84.37	258.26	84.21
	Mediana_{nc}	170.99	36.36	196.12	36.57	258.26	36.51
	Media_a	170.98	80.11	196.32	80.29	258.26	80.28
	Mediana_a	171.16	25.21	195.91	25.28	258.42	25.27
20 %	Media_{nc}	170.72	54.16	196.15	54.77	258.42	54.72
	Mediana_{nc}	170.73	25.16	196.29	25.34	258.50	25.42
	Media_a	170.62	47.61	196.12	48.18	258.26	48.17
	Mediana_a	170.82	6.73	196.18	6.93	258.20	6.93

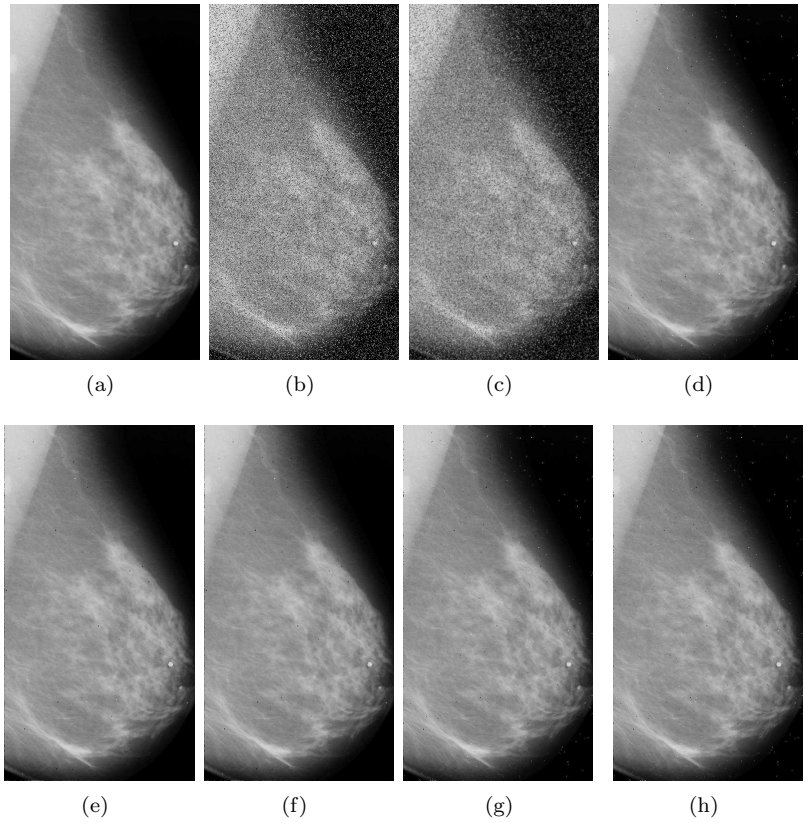


Figura 6.30: Imagen de mamografía, a) Original de 512x960 píxeles, b) 20 % de ruido impulsivo, c) Después del filtro de media, d) Después del filtro mediana, e) Después del filtro media$_{nc}$-fuzzy(M), f) Después del filtro media$_{nc}$-euclídea, g) Después del filtro mediana$_a$-fuzzy(M), h) Después del filtro mediana$_a$-euclídea.

Tabla 6.31: Valores óptimos para el parámetro d con diferentes densidades de ruido impulsivo.

Imagen	Mα 5 %	Mγ 5 %	Mα 10 %	Mγ 10 %	Mα 20 %	Mγ 20 %	Mα 30 %	Mγ 30 %
Caps	45	63	45	51	57	53	51	52
Estatua	70	70	70	70	70	70	70	65

Cuando el ruido es mayor que el 10 %, la mejor opción es el uso de la mediana de los píxeles no corruptos en el paso de filtrado.

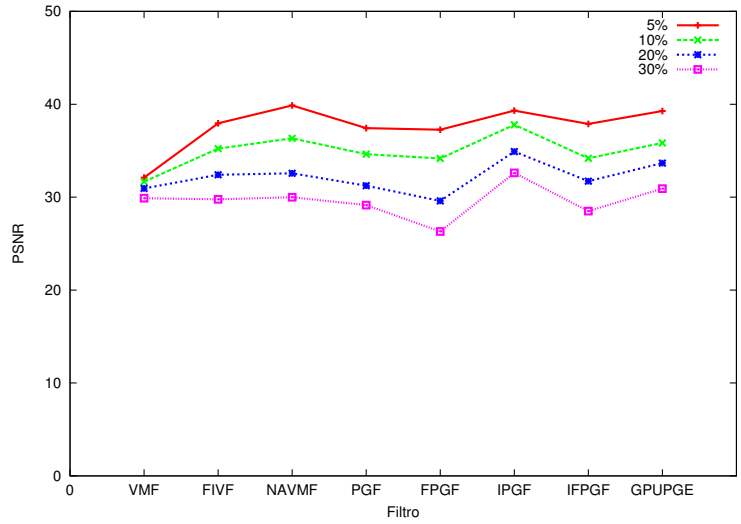
6.7.2 Evaluación de la calidad en imágenes a color

Se analizó la calidad obtenida de las imágenes cuando se añade ruido impulsivo y uniforme con la métrica euclídea en el *peer group*. Se denotará Mγ para referirnos al ruido uniforme y Mα para el ruido "sal y pimienta". Se calculó de manera heurística los valores para m y d. En la tabla 6.31 se muestra los mejores resultados para el parámetro d con diferente porcentaje y tipo de ruido utilizado para dos imágenes con distinta intensidad de color. Como puede observarse, el valor d es menor en imágenes con variedad de colores y es mayor con poca variación del color o colores oscuros, como es el caso de "Estatua".

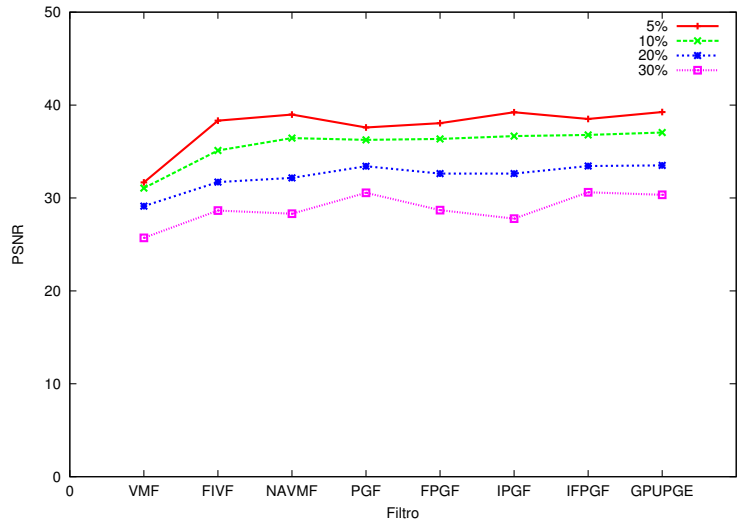
Los valores de m y d dependerá de la intensidad y el tipo de ruido que se está controlando. El valor óptimo de d para imágenes a color y métrica euclídea está en el intervalo [60,70]. Para estas dos imágenes y ruido uniforme, los mejores resultados se obtienen con $m = 2$, en contraste con el ruido impulsivo que tiene dos variantes: $m = 2$, cuando las imágenes tienen una menor intensidad del ruido o igual a 10 % y $m = 3$ cuando es mayor que 10 %. Esto se ilustra en la figura 6.31.

Para la imagen *Goldhill* con 10 % de ruido impulsivo es 49 y para 10 % de ruido uniforme es 46. Estudio realizado en (Sánchez y col. 2011c).

La calidad obtenida con los métodos 13 (que utiliza la métrica euclídea) y 15 han sido comparados con otros filtros: VMF, FIVF, NAVMF, PGF, FPGF, IPGF y IFPGF cuyos resultados están publicados en el trabajo (Camarena y col. 2010b) que refieren a la ejecución de forma secuencial. Las tablas 6.32, 6.33, 6.34 y 6.35 muestran dicha comparativa. La tabla 6.32 refleja los resultados obtenidos en calidad con la imagen *Caps* y ruido uniforme. La columna *PGME paralelo** es una versión de la *PGME paralelo* si se conoce de antemano que el ruido de la imagen es de tipo "sal y pimienta", trabajando sólo con

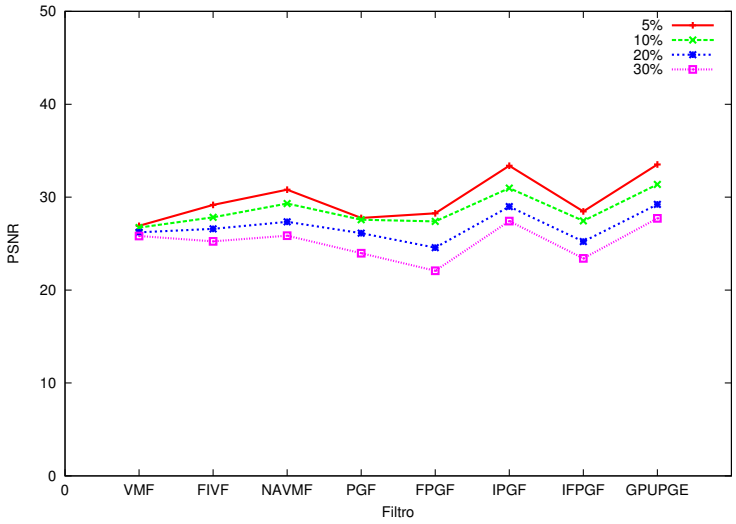


(a)

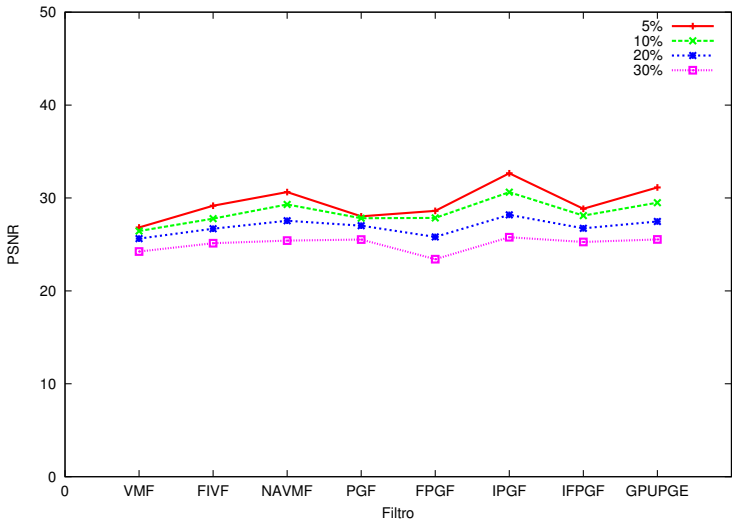


(b)

Figura 6.31: Calidad en términos PSNR para la imagen *Caps*. a) Ruido impulsivo, b) Ruido uniforme.



(a)



(b)

Figura 6.32: Calidad en términos PSNR para la imagen "Estatua". a) Ruido impulsivo, d) Ruido uniforme.

píxeles que tienen un valor extremo en uno de sus canales. La tabla 6.33 presenta la calidad obtenida de la imagen *Caps* con ruido impulsivo valor uniforme. Como puede observarse, para la mayoría de los casos el algoritmo paralelo elimina mayor cantidad de ruido que con los otros filtros.

Tabla 6.32: Resultados del calidad para la imagen *Caps* corrupta con diferentes densidades de ruido impulsivo.

	5 %			10 %			20 %			30 %		
Filtro	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD
Ruido	2.54	21.76	2.96	4.84	18.97	5.57	9.97	15.81	11.42	14.67	14.15	17.02
VMF	2.65	32.11	1.83	2.80	31.66	1.92	3.05	30.94	2.08	3.38	29.89	2.30
FIVF	0.31	37.95	0.30	0.63	35.22	0.54	1.21	32.41	1.02	1.85	29.77	1.62
NAVMF	0.25	39.87	0.25	0.51	36.33	0.49	1.16	32.57	0.99	1.85	29.99	1.58
PGF	0.41	37.43	0.33	0.66	34.63	0.59	1.28	31.24	1.23	2.17	29.14	1.96
FPGF	0.39	37.26	0.34	0.66	34.16	0.62	1.54	29.59	1.40	3.31	26.31	2.33
IPGF	0.23	39.32	0.25	0.41	37.79	0.43	0.86	34.91	0.85	1.39	32.61	1.31
IFPGF	0.35	37.88	0.29	0.64	34.18	0.57	1.34	31.72	1.11	2.12	28.50	1.83
PGME paralelo	0.30	38.43	0.32	0.58	35.16	0.59	1.22	32.60	1.26	1.98	29.53	2.11
*PGME paralelo**	**0.27**	**39.27**	**0.26**	**0.51**	**35.84**	**0.53**	**1.46**	**33.67**	**1.09**	**1.71**	**30.92**	**1.72**

Tabla 6.33: Calidad obtenida para la imagen *Caps* corrupta con diferentes densidades de ruido uniforme

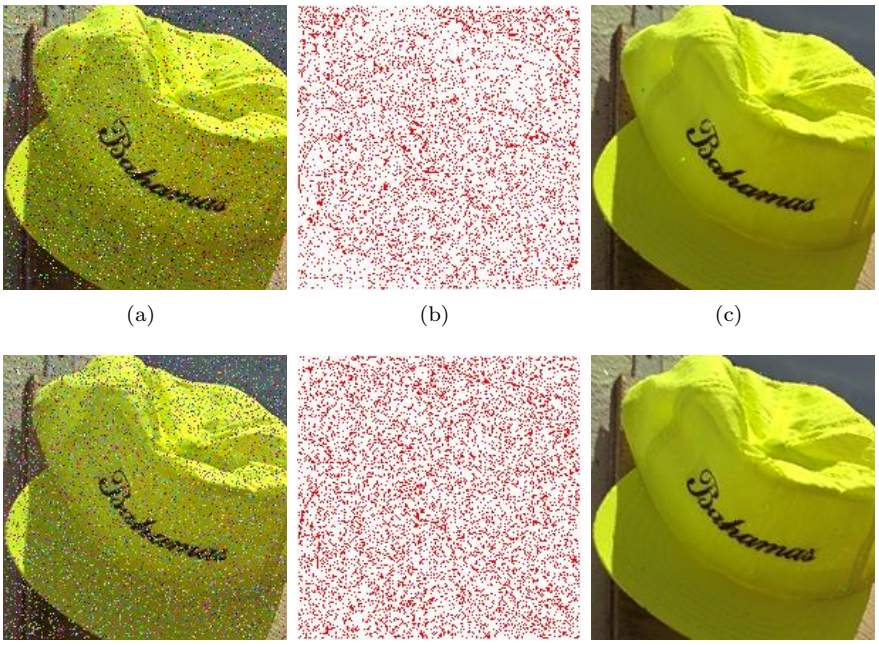
	5 %			10 %			20 %			30 %		
Filtro	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD
Ruido	4.29	20.75	3.72	8.31	17.86	7.22	16.77	14.79	14.55	25.28	13.00	21.97
VMF	2.74	31.68	1.88	2.96	31.07	2.00	3.54	29.12	2.37	4.76	25.71	3.38
FIVF	0.32	38.32	0.21	0.66	35.11	0.40	1.34	31.71	0.82	2.16	28.65	1.37
NAVMF	0.30	38.98	0.19	0.55	36.45	0.36	1.28	32.16	0.81	2.27	28.31	1.49
PGF	0.44	37.59	0.26	0.63	36.25	0.40	1.20	33.42	0.77	1.92	30.56	1.23
FPGF	0.41	38.05	0.25	0.63	36.35	0.39	1.35	32.62	0.83	2.49	28.69	1.41
IPGF	0.27	39.23	0.18	0.52	36.66	0.34	1.15	32.62	0.78	2.31	27.78	1.67
IFPGF	0.38	38.50	0.23	0.59	36.78	0.36	1.16	33.44	0.72	1.88	30.61	1.14
PGME paralelo	**0.33**	**39.25**	**0.23**	**0.57**	**37.04**	**0.37**	**1.19**	**33.51**	**0.77**	**1.93**	**30.34**	**1.27**

La figura 6.33 muestra la imagen *Caps* con 20 % de ruido impulsivo y uniforme, el mapa de detección con los píxeles corruptos encontrados, y las imágenes resultantes después del paso de filtrado.

La tabla 6.34 muestra los resultados obtenidos para la imagen estatua y ruido impulsivo. Como puede observarse, el método *PGME paralelo* y *AMF*, suprime mayor cantidad de ruido comparándolo con los otros filtros, para todas las densidades.

La tabla 6.35 muestra los resultados obtenidos para la imagen "Estatua" con ruido impulsivo y ruido uniforme. Se puede ver el método utilizado es muy competitivo con este tipo de ruido.

La figura 6.34, muestra la imagen de "estatua" con 20 % de ruido uniforme, mapa de detección de píxeles ruidosos, y la imagen resultante después del proceso de filtrado.



(a) (b) (c)

(d) (e) (f)

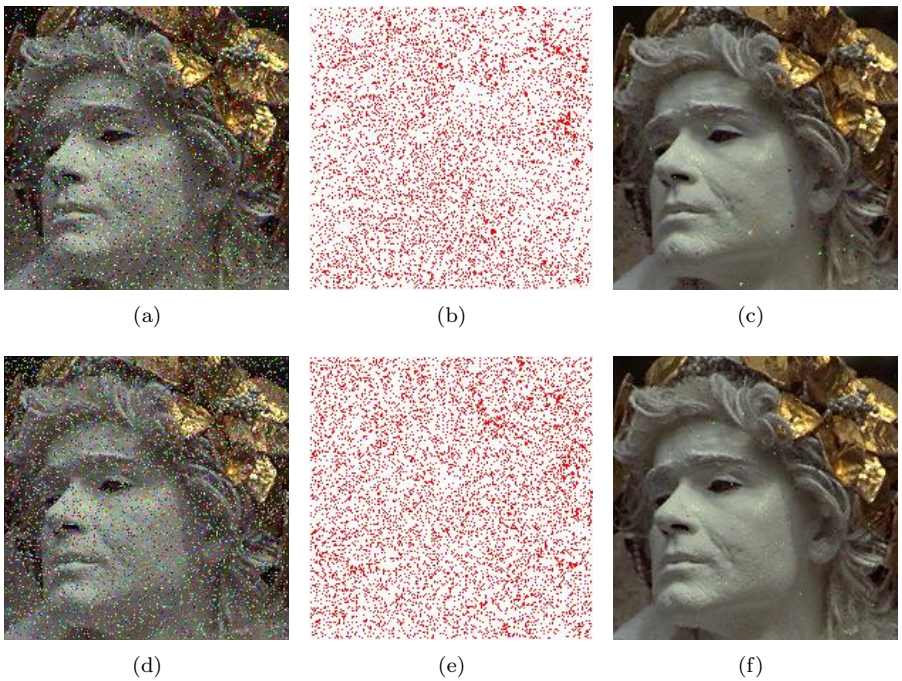
Figura 6.33: Comparativa de la imagen *Caps*. a) Imagen con un 20 % de ruido impulsivo, b) Detección de los píxeles ruidosos, c) Imagen filtrada, d) Imagen con 20 % de ruido uniforme, e) Detección de los píxeles ruidosos, f) Imagen filtrada.

Tabla 6.34: Calidad obtenida con la imagen "Estatua" corrupta con diferentes densidades ruido impulsivo.

	5 %			10 %			20 %			30 %		
Filtro	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD
Ruido	2.53	22.30	4.45	5.03	19.21	8.67	10.04	16.24	17.49	14.82	14.54	25.89
VMF	5.39	26.92	3.72	5.58	26.72	3.84	6.04	26.22	4.14	6.45	25.82	4.46
FIVF	1.22	29.17	0.99	1.94	27.83	1.49	2.97	26.59	2.38	4.02	25.24	3.51
NAVMF	0.90	30.80	0.75	1.45	29.31	1.27	2.65	27.35	2.18	3.84	25.85	3.26
PGF	1.58	27.76	0.98	1.91	27.57	1.48	2.87	26.12	2.92	4.80	23.96	4.34
FPGF	1.46	28.26	0.99	1.98	27.39	1.62	3.70	24.57	3.17	6.10	22.08	5.40
IPGF	0.57	33.38	0.42	1.10	30.97	0.79	2.05	28.98	1.50	3.15	27.43	2.38
IFPGF	1.36	28.47	0.86	1.86	27.45	1.36	3.07	25.22	2.87	4.70	23.41	3.77
PGME paralelo	0.91	30.62	1.11	1.49	28.73	2.14	2.64	26.42	4.23	4.28	24.57	5.99
*PGME paralelo**	**0.58**	**33.52**	**0.80**	**1.06**	**31.37**	**1.21**	**1.98**	**29.22**	**2.17**	**2.87**	**27.72**	**3.23**

Tabla 6.35: Comparación de la calidad de la imagen "Estatua" con ruido uniforme

	5 %			10 %			20 %			30 %		
Filtro	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD
Ruido	3.59	21.99	3.71	7.39	18.82	7.53	15.10	15.75	15.52	22.67	13.99	23.29
VMF	5.49	26.83	3.80	5.82	26.44	4.01	6.63	25.63	4.54	7.81	24.24	5.35
FIVF	1.22	29.17	0.70	1.97	27.78	1.19	3.04	26.69	1.93	4.31	25.13	2.86
NAVMF	0.92	30.64	0.55	1.49	29.30	0.94	2.70	27.55	1.74	4.26	25.41	2.82
PGF	1.52	28.02	0.81	1.88	27.82	1.09	2.73	27.02	1.71	3.88	25.53	2.57
FPGF	1.40	28.61	0.77	1.89	27.85	1.09	3.35	25.81	1.99	5.55	23.41	3.38
IPGF	0.67	32.66	0.43	1.25	30.62	0.83	2.43	28.18	1.62	4.00	25.77	2.84
IFPGF	1.29	28.84	0.69	1.75	28.10	1.00	2.73	26.74	1.64	3.92	25.27	2.43
PGME paralelo	**0.86**	**31.13**	**0.57**	**1.45**	**29.49**	**1.08**	**2.62**	**27.47**	**2.11**	**3.95**	**25.54**	**3.06**



(a) (b) (c)

(d) (e) (f)

Figura 6.34: Imagen "Estatua". a) 20 % de ruido impulso, b) Detección de píxeles ruidosos, c) Filtrada, d) 20 % de ruido uniforme, e) Detección de píxeles ruidosos, f) Imagen filtrada.

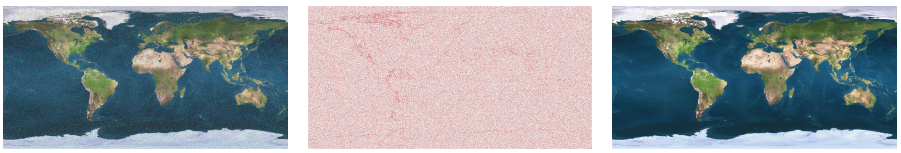
La tabla 6.36 muestra una comparativa de los resultados para el procesamiento de imágenes *Goldhill* (con el método *PGME paralelo* y *AMF paralelo*) con los resultados presentados en (Smolka 2010). Se puede observar que para los dos modelos de ruido utilizados, el método empleado supera en PSNR y NCD a los filtros con los que se compara: PGSR, SANRF, ACWVMF, VLUMS, FANRF, PGF, FPGF, FMVMF, MICM , AVMF, RODSVMF, AVLUMS, FFNRF y VMF.

La imagen resultante y el mapa de ruido después del proceso de detección y filtrado de la imagen *world* de tamaño 2400×1200 píxeles, se muestra en la figura 6.35, con un 20 % de ruido uniforme.

Se realiza una comparativa de calidad del método *PGME paralelo* con el método $Iterativo_{coseno}$. En la tabla 6.37 se muestran los resultados

Tabla 6.36: Calidad obtenida de la imagen *Goldhill* corrupta con 10 % de ruido impulsivo.

	Sal y pimienta			Ruido Uniforme		
Filtro	MAE	PSNR	NCD	MAE	PSNR	NCD
PGSF	0.75	36.14	6.71	0.81	35.86	6.81
SANRF	0.01	35.94	8.01	0.78	36.05	6.9
ACWVMF	0.04	35.9	6.2	0.84	35.41	7.33
VLUMS	0.04	35.85	6.25	0.85	35.41	7.39
FANRF	0.04	35.66	6.83	0.81	34.73	7.34
PGF	0.07	35.28	5.85	0.83	35.10	6.76
FPGF	0.07	35.37	6.79	0.82	34.95	7.19
FMVMF	0.07	35.48	6.17	0.84	34.50	6.99
MICM	0.03	35.64	6.67	0.94	33.72	9.09
RODSVMF	0.1	35.37	7.08	0.85	34.37	9.34
AVLUMS	0.66	34.5	9.5	1.7	33.24	11.32
AVMF	0.19	34.22	9.46	0.96	33.50	8.32
FFNRF	0.28	34.21	6.88	1.03	33.24	7.87
VMF	4.13	30.01	41.87	4.98	28.89	43.9
PGME paralelo	0.83	34.91	7.9	0.77	36.26	5.3
*PGME paralelo**	**0.61**	**38.79**	**0.41**	**0.77**	**36.26**	**5.3**



(a) (b) (c)

Figura 6.35: Imagen *world*. a) 20 % de ruido impulso, b) Detección de píxeles ruidosos, c) Imagen filtrada.

para dos imágenes, *Caps* y "Estatua" para diferente porcentaje de ruido impulsivo.

Como puede verse, el método $Iterativo_{coseno}$ obtiene una calidad competitiva al compararse con el otro filtro. Este método tiene la ventaja de que se desconoce inicialmente la cantidad de ruido presente en la imagen y sin dependencia de varios parámetros que se calculan previamente para obtener buenos resultados en calidad.

6.7.3 Evaluación del método de parametrización

Como puede observarse en el apartado anterior, cuando se utiliza el *peer group* con cualquier métrica, se requiere de un estudio previo heurístico para conocer los mejores valores de los parámetros (d, m para la distancia euclídea y coseno del ángulo y además del valor de k para la métrica fuzzy) con los que se obtiene mayor calidad. Se ha propuesto un método de parametrización que se explicó en el apartado 4.4, con la finalidad de evitar este tipo de cálculos previos.

Los resultados obtenidos al aplicar el método de parametrización para valores d, m y k, se muestran en la tabla 6.38 para 5, 10, 20, 30, 40 y 50 % de ruido impulsivo.

Se puede observar que el valor de k es 1024 para todas las imágenes y con cualquier porcentaje de ruido. El valor d es el mismo para imágenes con ruido mayor o igual al 20 %, de esta forma solo se tienen tres valores para d, 0.9, 0.88 y 0.85. El valor m tiene tres valores, 2, 4 y 5, donde éste último se presenta en imágenes con ruido mayores al 20 %. Con la anterior deducción podemos reducir la tabla 6.38 a la tabla 6.39 en la que se consideran todos los porcentajes de ruido.

Si se conoce de antemano el porcentaje de ruido que contiene la imagen a color, se aplican los valores de la tabla 6.38, esto solo se presenta en pruebas para verificar la validez o eficiencia de un método o procedimiento. En imágenes reales no se conoce inicialmente la cantidad de ruido que contiene la imagen, por lo que se desconoce los mejores

Tabla 6.37: Comparación de calidad con cuatro algoritmos para la imagen *Caps* y "Estatua" con ruido impulsivo.

	Filtro	5 %			10 %			20 %			30 %		
		MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD	MAE	PSNR	NCD
Caps	*PGME paralelo*	0.30	38.43	0.32	0.58	35.16	0.59	1.22	32.60	1.26	1.98	29.53	2.11
	$Iterativo_{coseno}$	0.83	34.08	0.01	1.18	32.41	0.01	2.10	30.31	0.02	3.39	27.96	0.03
"Estatua"	*PGME paralelo*	0.91	30.62	1.11	1.49	28.73	2.14	2.64	26.42	4.23	4.28	24.57	5.99
	$Iterativo_{coseno}$	1.15	30.19	0.01	1.77	28.54	0.01	3.34	25.79	0.03	5.28	23.73	0.05

Tabla 6.38: Mejores valores después del proceso de parametrización para los parámetros m, d y k con distinto porcentaje de ruido impulsivo.

%	k	m	d
5	1024	2	0.9
10	1024	2	0.88
20	1024	4	0.85
30	1024	5	0.85
40	1024	5	0.85
50	1024	5	0.85

Tabla 6.39: Mejores valores después del proceso de parametrización para los parámetros m y d de forma ascendente.

% ruido	k	m	d
Cualquiera	1024	2	0.9
		2	0.88
		4	0.85
		5	0.85

parámetros con los que se elimina mayor cantidad de ruido. En estos casos, el método propuesto es:

1. Ejecutar el filtro con los parámetros d y m de la tabla 6.39. Estos valores incluyen todos los mejores resultados en todos los porcentajes de ruido, comenzando con los parámetros del 5 %, o bien,
2. Ejecutar el filtro con los parámetros d y m en el orden inverso como se muestra en la tabla 6.40. Este orden comienza con los valores donde se supone que el ruido es mayor o igual al 20 %.

Tabla 6.40: Mejores valores después del proceso de parametrización para los parámetros m y d de forma descendente.

% ruido	k	m	d
Cualquiera	1024	5	0.85
		4	0.85
		2	0.88
		2	0.9

Para verificar la calidad obtenida después de aplicar la primera y segunda opción del método, se eligió una imagen al azar y los resultados son los que se muestran en la figura 6.36. Como puede observarse,

cuando la imagen contiene un porcentaje de ruido menor o igual al 20 %, son similares los resultados al ejecutar la primera o segunda solución del método, sin embargo, para un ruido mayor o igual al 30 % es preferible la segunda solución. Aplicar la segunda solución cuando se desconoce la cantidad de ruido que contiene la imagen es la mejor opción.

La calidad que se obtiene al filtrar la imagen *Lenna* de tamaño 512x512 con 10 % de ruido impulsivo, con cuatro métodos distintos para eliminar el ruido impulsivo se muestra en la tabla 6.41. Como puede observarse el método $Iterativo_{coseno}$ elimina mayor cantidad de ruido en la imagen y el método que elimina menos ruido es el "$PGMC$ *secuencial* y AMF". El que preserva mejor los detalles de la imagen es el "$PGME$ *secuencial* y AMF" y el que preserva menos detalles una vez que se filtra la imagen es el método "$PGMC$ *secuencial* y AMF".

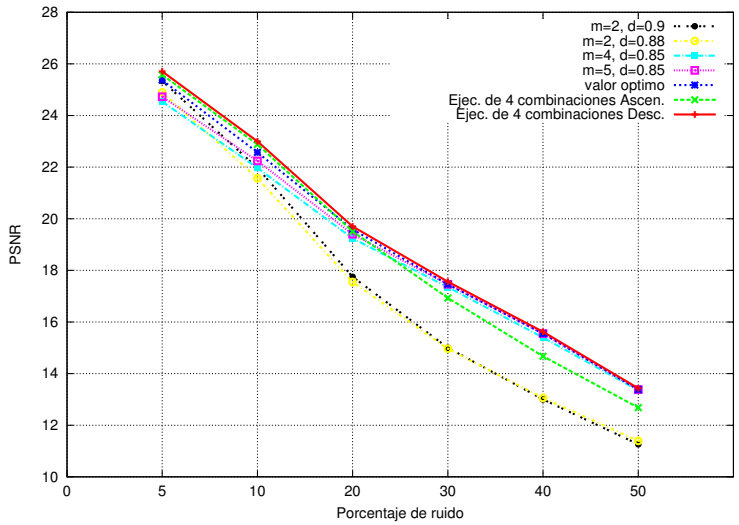


Figura 6.36: Calidad en términos PSNR para diferentes valores de m y d para cada porcentaje de ruido.

Tabla 6.41: Comparación de la calidad con cuatro métodos y filtros usando imagen *Lenna* y 10 % de ruido impulsivo

Método	PSNR	MSE	MAE
PGME secuencial y *AMF*	29.44	73.89	2.20
PGMF secuencial y *AMF*	29.32	76.13	2.24
PGMC secuencial y *AMF*	27.25	122.38	2.62
Filtro_Iterativo_Coseno y *AMF*	30.16	62.59	2.48

6.8 Conclusiones

En este capítulo hemos presentado las implementaciones de los algoritmos tanto secuenciales como paralelos en arquitecturas CPU y GPU y la combinación de ambos para eliminar el ruido impulsivo y uniforme. Por un lado, se ha presentado el rendimiento computacional obtenido, y por otro, la calidad obtenida después de realizar el proceso de filtrado.

Para obtener una buena calidad en la imagen filtrada, se ha utilizado una primera etapa de detección de los píxeles ruidosos basándose en el concepto del *peer group* con métricas fuzzy, euclídea o coseno del ángulo. En una segunda etapa se filtran aquellos píxeles cuya clasificación es corrupta, en la que se sustituye el píxel ruidoso por el cálculo de la mediana o media aritmética de los píxeles de la ventana de filtrado. Filtrar la imagen utilizando en la etapa de detección la métrica fuzzy o euclídea, proporciona prácticamente la misma calidad (no llega al 1 % PSNR de diferencia). Sin embargo, el coste computacional al utilizar la métrica fuzzy es mayor que con la distancia euclídea (26 %-28 % Megapíxeles/segundo). En general, la calidad de los filtros implementados en esta tesis comparados con otros, es competitiva.

Como se ha mencionado en capítulos anteriores, el uso del *peer group* con cualquiera de las métricas, requiere un estudio previo heurístico para ajustar algunos umbrales de entrada cuando se utiliza las métricas euclídea, fuzzy y coseno, con la finalidad de obtener la mejor configuración que proporcione buena calidad en la imagen filtrada. El "método de parametrización" obtiene valores generales para los umbrales que se puede aplicar independientemente de la cantidad de ruido introducido y proporciona una calidad aceptable para diferentes tipos de imágenes. Por otro lado, el algoritmo iterativo propuesto con la finalidad de disminuir la cantidad de los parámetros y evitar el estudio previo heurístico, proporciona buena calidad en PSNR, MAE y NCD independientemente de la densidad de ruido introducido. En algunos

casos, los resultados se encuentran dentro de la misma magnitud con respecto al filtro $PGME_{paralelo}$.

El rendimiento computacional se ha mejorado con la paralelización de los algoritmos en CPU y GPU. En las versiones secuenciales el algoritmo optimizado mejora en un 27 %-30 % GFlops a la versión no optimizada. Estos resultados dependen de la cantidad de ruido introducido. La versión paralela optimizada con ocho cores, disminuye en un 41 % a la no optimizada paralela.

Al utilizar *multicores* se ha comprobado que para tamaños pequeños de imágenes (96×64 o 192×128) el rendimiento computacional es óptimo al utilizar ocho cores. Por el contrario, para tamaños mayores es preferible repartir la carga computacional entre los cores disponibles en el sistema. La versión en *multicores*, reduce el tiempo computacional en un 76 %- 90 % en segundos con respecto a la versión secuencial, dependiendo del tamaño de la imagen.

Cuando se trabaja con GPU, es necesario configurar algunas características para optimizar el procesamiento. Entre las que se podemos mencionar las siguientes: la configuración de bloques e hilos, cantidad de bytes por píxel a reservar en la memoria de la GPU, modo de reserva de memoria, acceso a los datos a través de la memoria de texturas y acceso a los datos a través de la memoria compartida. En general, el tamaño de bloques es mejor cuando se agrupan en 16×16 hilos, excepto para tarjetas gráficas que soportan 1024 hilos por bloque y el tamaño de la imagen es pequeño (96×64 o 192×128), ya que hay cores inactivos y por lo tanto el rendimiento no es el óptimo. Los mejores resultados se han obtenido cuando se ha reservado espacio en la GPU con 4 bytes porque se gana coalescencia en los accesos a memoria del *device*. En las implementaciones desarrolladas cuando se accede a los datos en memoria de la GPU a través de la memoria de texturas, se ha obtenido menor coste computacional comparado con las otras opciones. Hemos realizado diferentes pruebas con diferente tamaño de imagen y distinta cantidad de ruido, y en todas ellas se observa una mejora considerablemente cuando se utiliza el acceso a memoria del *device* con texturas.

En la práctica, hemos observado que el tamaño de la imagen a partir del cual es conveniente utilizar una GPU es 768×512. Si se dispone de más GPUs, convendrá por tanto, asignar a cada GPU una porción de imagen en ese rango.

Hemos probado implementaciones híbridas GPU y *multicores* en un co-procesamiento heterogéneo, encontrando que no necesariamente la

carga computacional en cada arquitectura es del 50 % y 50 %, si no que va a depender del tamaño de la imagen y del hardware disponible. Se ha comparado los resultados de una GPU con la versión híbrida y se obtiene una mejora con ésta última arquitectura entre el 70 % y 75 % en segundos. Cuando se utiliza la versión híbrida, la mejora es de un 32 %- 58 % en segundos con respecto a la utilización de varias GPUs. Por último, la paralelización en *multicore* mejora en un 78 % con la versión híbrida. Las mejoras anteriores se refieren a tamaños de imagen mayores o iguales que 3072x2048 píxeles, pues a partir de este tamaño el co-procesamiento entre CPU-GPU se puede utilizar con buenos resultados.

En todas las comparaciones entre implementaciones GPU frente a CPU secuencial, la diferencia fue favorable a la GPU, con lo que se concluye que siempre conviene utilizar versiones paralelas.

Capítulo 7

Aplicaciones en imágenes médicas

7.1 Introducción

Las imágenes obtenidas por Rayos X o Tomografía Computarizada en condiciones adversas pueden ser contaminadas con diversos ruidos que pueden afectar la detección de enfermedades, por ejemplo, dificultar la detección de microcalcificaciones. El uso de radiografías computarizadas (CT por sus siglas en inglés) en la práctica clínica ha seguido por un alto incremento en el número de exámenes realizados y casos de sobredosis en pacientes, principalmente en niños en aplicaciones pediátricas.

Este capítulo tiene como objetivo presentar las aplicaciones de los métodos de filtrado de ruido presentados anteriormente en esta tesis, en imágenes médicas de mamografía y tórax para eliminar el ruido impulsivo, gaussiano y/o speckle, y su relación con la reducción de la dosis de radiación.

El presente capítulo se ha organizado de la siguiente manera. La sección 7.2 presenta un método diseñado para reducir el ruido gaussiano, impulsivo y speckle y la combinación de ambos, denominado PGFDNL. Este método combina el filtro difusivo no lineal con una técnica de *peer group* y la métrica fuzzy, ambos filtros introducidos en la sección 4. Se propone para eliminar el ruido en la imagen sin tener información inicial acerca del tipo de presente en la imagen. El estudio de este análisis ha sido presentado en (Sánchez y col. 2012b).

La sección 7.3, presenta una técnica para la reducción de dosis de radiación en CT absorbidas por los pacientes, especialmente en niños en aplicaciones pediátricas a través del filtro de difusión no lineal, presentado en la sección 4, manteniendo la integridad del diagnóstico. Se ha estudiado el impacto del ajuste de varios *milliAmpere-second* (mAs) en la calidad de la imagen obtenida de un fantoma RANDO. Esta sección trata la viabilidad de evaluar la precisión de la metodología empleada como una función de reducción de la dosis. El análisis y la evaluación se ha presentado en (Sánchez y col. 2012a).

7.2 Reducción del ruido mediante el método PGFDNL

Las técnicas de eliminación de ruido para restaurar las imágenes ruidosas es un importante objeto de estudio hoy en día, por ejemplo, las imágenes obtenidas por rayos X o CT en condiciones adversas. En el caso de la adquisición de imágenes mamográficas, éstas pueden estar contaminadas con ruido que puede afectar la detección de microcalcificaciones.

En esta sección se presenta la evaluación del método PGFDNL (*Peer Group* Fuzzy- Difusivo No Lineal) presentado en el apartado 4.5 para eliminar el ruido impulsivo, gaussiano y speckle de las imágenes de mamografía. Las imágenes fueron filtradas con los métodos PGMF, FDNL y PGFDNL. El concepto de *peer group* con la métrica fuzzy es buen método para eliminar el ruido impulsivo en la imagen, el método FDNL para eliminar el ruido gaussiano, el FDNL y ambos métodos pueden eliminar el ruido speckle. A continuación se analiza las ventajas o prestaciones que se obtienen del método PGFDNL con respecto a los métodos por separado.

7.2.1 Análisis de Resultados

Una de las imágenes originales que se han utilizado para las pruebas es la que se muestra en la figura 7.1. Para mejor apreciación de la imagen después del proceso de filtrado, se ha trabajado con un fragmento de imagen de tamaño 512×960. Las imágenes originales para estas pruebas, fueron obtenidas de la base de datos (PEIPA 2003) a las cuales se les añadió ruido a través de la función de MATLAB *imnoise* con diferentes magnitudes, asumiendo que estaban libres de ruido. Los niveles de ruido introducidos en las imágenes fueron: ruido impulsivo (densidad 0.10) , ruido gaussiano (media cero y varianza de 0.01) y

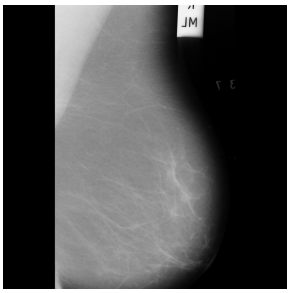


Figura 7.1: Imagen de mamografía. Tamaño 1024×1024.

ruido speckle (varianza 0.04). Estos ruidos se introdujeron de forma separada o combinada en la imagen.

El valor de los parámetros m y d cuando se utilizó la métrica fuzzy fueron obtenidos a través de un estudio heurístico. Estos valores dependen del tipo y cantidad de ruido introducido. Los resultados son los que se muestran en la tabla 7.1. En los casos en los que se considera la varianza, es el mismo valor para el parámetro m, $m = 8$, lo que significa que se requiere involucrar a todos los píxeles vecinos de la ventana de filtrado (en una ventana de tamaño 3x3) para decidir si el píxel es corrupto o no, de lo contrario, el valor es 4. En los casos con varianza de 0.01, el valor de d es 0.92. A través de un proceso similar al descrito en el artículo (Camarena y col. 2008) se obtiene el valor de k, cuyo valor resultante fue de k=1024.

Aplicando los filtros PGMF, FDNL y PGFDNL a la imagen con 10 % de ruido impulsivo, los resultados obtenidos son los que se muestran en la tabla 7.2. Esta tabla muestra la calidad en términos MAE, PSNR y MSE después del proceso de filtrado con los filtros. Como se puede observar, cuando la imagen contiene únicamente ruido impulsivo, el mejor método de filtrado es el PGMF. También se puede usar el método PGFDNL con poca diferencia en calidad con respecto al método anterior. El método FDNL no proporciona buena calidad en la imagen

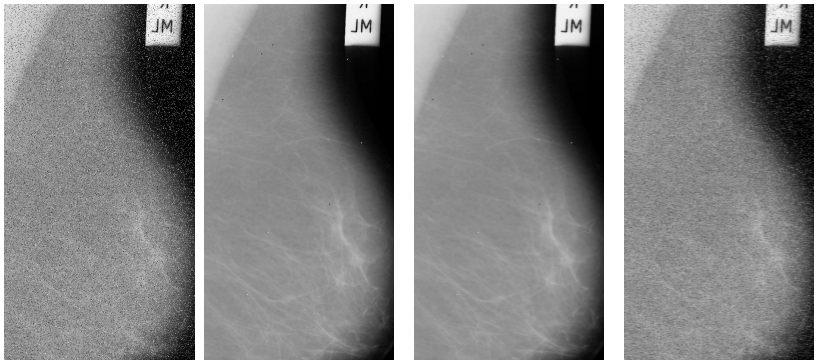
Tabla 7.1: Mejores valores para los parámetros d y m

	m	d
$D = 0.10$ para ruido impulsivo valor fijo	5	0.85
$\sigma^2 = 0.01$ para ruido gaussiano	8	0.92
$D = 0.10$ para ruido impulsivo y $\sigma^2 = 0.01$ ruido gaussiano	8	0.92
$\sigma^2 = 0.04$ para ruido speckle	8	0.88

Tabla 7.2: Calidad obtenida para la imagen de mamografía con 10 % de ruido impulsivo.

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	5.5365	40.6985	0.1793
Imagen filtrada por el método FDNL	36.4523	24.3934	9.4899
Imagen filtrada por el método PGFDNL	7.1813	39.5688	0.759
Imagen ruidosa	1.90E+02	15.3459	12.8218

resultante. La figura7.2*a* muestra la imagen con ruido impulsivo. La figura 7.2*b* es la imagen filtrada obtenida después de aplicar el método PGMF, 7.2c y 7.2*d* son las imágenes obtenidas después de filtrarse con los métodos PGFDNL y FDNL respectivamente. Se puede observar que los métodos utilizados para obtener las imágenes 7.2*b* y 7.2*c* son los mejores para este tipo de ruido.



(a) (b) (c) (d)

Figura 7.2: Imagen con a) Densidad (D) = 0.10 de ruido "sal y pimienta", b) Imagen filtrada por el PGMF, c) Imagen filtrada por el PGFDNL, d) Imagen filtrada por el FDNL.

.

Cuando la imagen contiene únicamente ruido gaussiano, la calidad de los métodos FDNL y PGFDNL obtienen resultados muy similares y son mejores que el método PGMF. La tabla 7.3 muestra los resultados. El método FDNL y PGFDNL están aproximadamente ocho unidades de PSNR mejor, respecto a la imagen ruidosa.

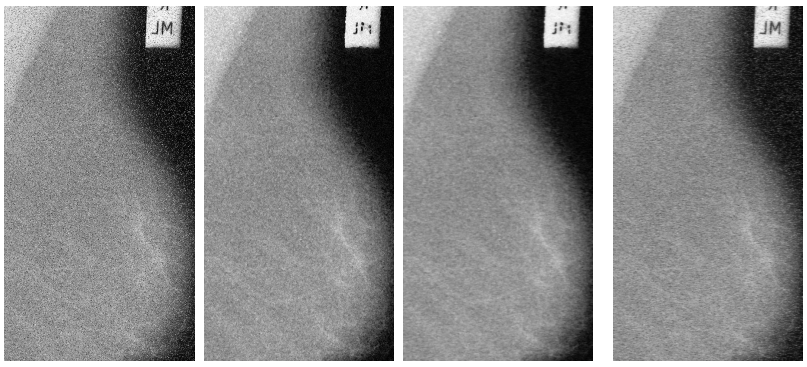
La calidad obtenida después de filtrar las imágenes contaminadas con los dos tipos de ruido, se muestran en la tabla 7.4 y las imágenes obtenidas después de los filtros se encuentran en la figura 7.3.

Tabla 7.3: Calidad obtenida para la imagen de mamografía con varianza = 0.01 (ruido gaussiano).

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	305.8954	23.2751	13.2662
Imagen filtrada por el método FDNL	88.5022	28.6613	7.3277
Imagen filtrada por el método PGFDNL	99.7893	28.14	6.9826
Imagen ruidosa	621.5723	20.1959	19.6725

Tabla 7.4: Calidad obtenida de la imagen con $(D) = 0.10$ y varianza = 0.01 (Ruido impulsivo y gaussiano).

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	322.9214	23.0398	13.6645
Imagen filtrada por el método FDNL	299.4993	23.3668	12.5675
Imagen filtrada por el método PGFDNL	103.1628	27.9956	7.3411
Imagen ruidosa	2.23E+03	14.6383	29.6629



(a) (b) (c) (d)

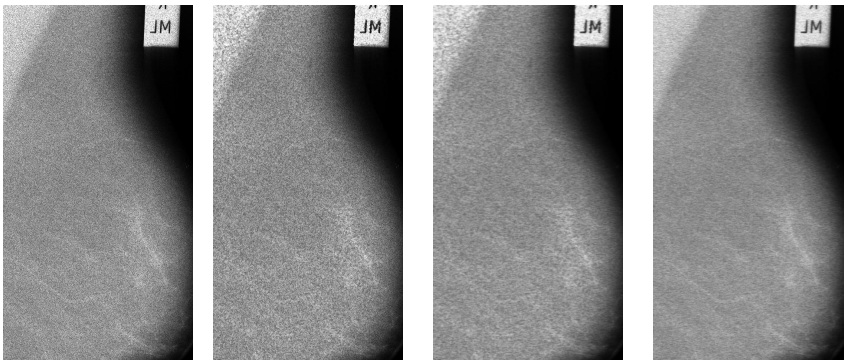
Figura 7.3: a) Densidad $(D) = 0.10$ para ruido "sal y pimienta", y con varianza = 0.01 para ruido gaussiano, b) Imagen filtrada por el PGMF, c) Imagen filtrada por el PGFDNL, d) Imagen filtrada por el FDNL.

El método PGFDNL es aproximadamente cuatro unidades de PSNR mejor que los otros métodos y con respecto a la imagen original ruidosa es aproximadamente 13 unidades de PSNR. La figura 7.3*c* es la que se visualiza mejor en calidad, que se corresponde a la imagen filtrada con el método PGFDNL.

Los resultados obtenidos después de aplicar los filtros con la imagen que contiene ruido speckle se muestra en la tabla 7.5 y figura 7.4.

Tabla 7.5: Calidad obtenida para la imagen de mamografía con ruido speckle.

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	1.25E+02	20.3004	19.0856
Imagen filtrada por el método FDNL	1.02E+02	28.0572	7.412
Imagen filtrada por el método PGFDNL	7.37E+03	27.1649	8.1496
Imagen ruidosa	8.47E+02	18.8536	23.3195



(a) (b) (c) (d)

Figura 7.4: a) Ruido speckle, b) Imagen filtrada por el PGMF, c) Imagen filtrada por el PGFDNL, d) Imagen filtrada por el FDNL.

El método PGFDNL y FDNL obtienen un comportamiento similar. El método PGMF con este tipo de ruido no lo elimina. Se puede apreciar que en las figuras 7.4*c* y 7.4*d* son las mejores imágenes filtradas, correspondientes a los método PGFDNL y FDNL.

Otras pruebas.

Se filtraron imágenes de mamografías con ruido uniforme correspondiente al 10 %, la tabla 7.6 muestra los resultados. El ajuste de parámetros para d y m, fueron 4 y 0.96 respectivamente, para el filtro que utiliza la métrica fuzzy y el *peer group*. Se puede observar que es mejor usar el método PGMF para eliminar este tipo y cantidad de ruido, aunque también con el método PGFDNL que proporciona similares resultados. El método FDNL por sí mismo no provee mejor calidad en la imagen.

Tabla 7.6: Calidad de la imagen obtenida con $(D) = 0.10$ (Ruido impulsivo uniforme).

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	4.7986	41.3196	0.2324
Imagen filtrada por el método FDNL	118.427	27.3963	6.2254
Imagen filtrada por el método PGFDNL	6.6632	39.894	0.7643
Imagen ruidosa	7.93E+02	19.1365	7.3114

La tabla 7.7 muestra la calidad resultante después de aplicar el filtro a la imagen con 20 % de ruido impulsivo. La figura 7.5 muestra la imagen filtrada. El ajuste de parámetros para d y m en este caso es 6 y 0.85 respectivamente cuando se utiliza la métrica fuzzy y el *peer group*.

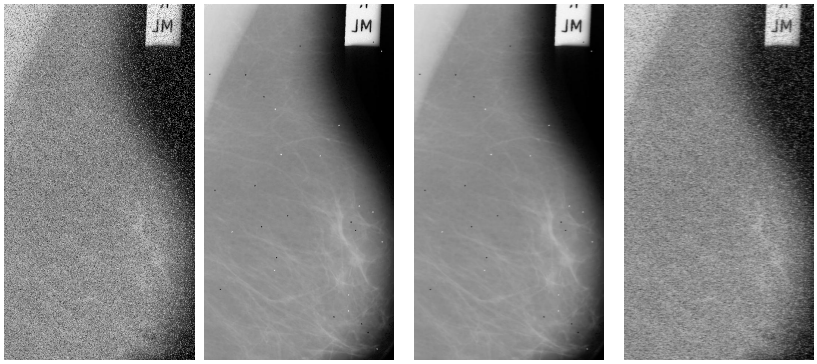
Tabla 7.7: Calidad obtenida para la imagen de mamografía con 20 % de ruido impulsivo.

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	16.194	36.0373	0.4109
Imagen filtrada por el método FDNL	500.667	21.1353	15.75
Imagen filtrada por el método PGFDNL	11.5312	37.5121	0.9083
Imagen ruidosa	3.77E+03	12.3731	25.5063

La tabla 7.8 muestra los resultados cuando la imagen contiene 20 % de ruido uniforme. En este caso el ajuste de parámetros de m y d es 4 y 0.97 respectivamente cuando se utiliza la métrica fuzzy y el *peer group*.

Con esta densidad de ruido impulsivo, el método PGFDNL provee una mejor calidad que si se aplica únicamente el método PGMF.

Se puede concluir después de los resultados presentados que cuando la imagen contiene únicamente ruido impulsivo (valor fijo) la mejor técnica de filtrado es el PGMF, aunque el método PGFDNL proporciona similares resultados. Si la imagen contiene únicamente ruido gaussiano



(a) (b) (c) (d)

Figura 7.5: a) Densidad (D) = 0.20 de ruido "sal y pimienta", b) Imagen filtrada por el PGMF, c) Imagen filtrada por el PGFDNL, d) Imagen filtrada por el FDNL.

Tabla 7.8: Calidad obtenida para la imagen de mamografía con 20 % de ruido uniforme.

	MSE	PSNR	MAE
Imagen filtrada por el método PGMF	8.6848	38.7432	0.3683
Imagen filtrada por el métodoFDNL	273.37	23.7633	10.9101
Imagen filtrada por el método PGFDNL	8.1951	38.9953	0.8187
Imagen ruidosa	1.60E+03	16.0896	14.7629

o speckle, la mejor técnica para eliminar el ruido fue el método FDNL seguido por muy cerca por el método PGFDNL.

Cuando la imagen contiene ambos ruidos: impulsivo y gaussiano, la mejor técnica es el PGFDNL (PGMF+ no lineal Difussion). La combinación del método de Difusión después del método PGMF mejora la calidad de la imagen obtenida, pero debe ejecutarse en este orden. Esta sucesión se debe a que, el uso de *peer group* y métrica fuzzy elimina eficientemente los valores extremos del ruido después el FDNL hace una difusión de los valores de los píxeles. Si el método difusivo fuera el primero, la eliminación de los valores extremos (ruido impulsivo) fallaría.

Cuando se desconoce la información del tipo de ruido contenido en la imagen, el mejor método de filtrado es PGFDNL.

7.3 Reducción de la dosis de radiación mediante el uso del FDNL.

El uso de radiografías computarizadas (CT por sus siglas en inglés) en la práctica clínica ha seguido por un aumento en el número de exposiciones realizadas y casos de sobredosis en pacientes, principalmente en niños en aplicaciones pediátricas.

En la técnicas radiográfica, la cantidad de la radiación necesaria para producir una imagen adecuada, es específica para los sistemas *Screen-Film* y las condiciones de procesamiento químico. De lo contrario, el proceso de adquisición en CR es independiente del proceso de proyección, y permite producir imágenes aceptables durante un gran rango de exposiciones. Desafortunadamente, este hecho introduce el riesgo de sobre-exposición sistemática.

Según ICRP (Comisión Internacional sobre la Protección Radiológica), la dosis de los pacientes en CR, especialmente en el caso de los niños, siempre deben mantenerse según el criterio ALARA: tan bajo como sea razonablemente posible.

Dado que menos radiación significa más ruido cuántico en la imagen y desde la aceptabilidad de imágenes digitales depende sobre el contenido del ruido, se analizó una técnica para garantizar la calidad de la imagen y hacer un diagnóstico preciso en la aplicación clínica. Se investigó la viabilidad de evaluar la precisión diagnóstica como una función de reducción de dosis, bajar los mAs, a través de la aplicación de un filtro difusivo no lineal (FDNL).

7.3.1 Metodología

El filtro denominado FDNL presentado en el apartado 4.3, se utilizó para filtrar el ruido inherente en una CR para reducir la dosis absorbida por los pacientes. El método se implementó con el fin de disminuir la dosis CR, basada en la selección de exposición baja de rayos X mediante el uso del FDNL.

La metodología a seguir consiste en lo siguiente: inicialmente se toman dos imágenes, I_A e I_B con $AmAs$ y $BmAs$ respectivamente. Supondremos que $B > A$. Para reducir la dosis a la imagen I_A se le aplica el FDNL obteniéndose la imagen filtrada I_A*. Para validar el proceso, a la imagen I_B se le añade un ruido gaussiano con varianza (σ^2) como se explica en los trabajos (Donoho 1995), obteniendo la imagen I_B*. Para comprobar la reducción de dosis, se compara el ruido SD corres-

pondiente a la imagen I_A y B*, y también el correspondiente a $A*$ y I_B. La figura 7.6 muestra este proceso.

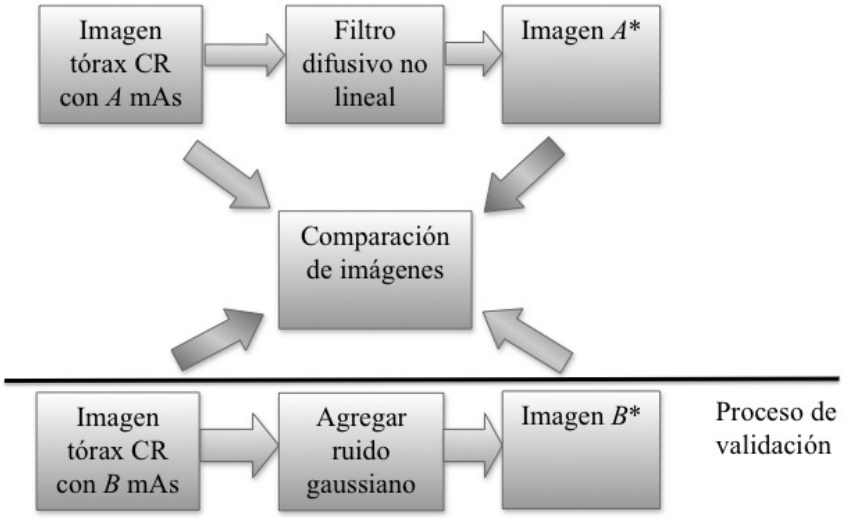


Figura 7.6: Metodología para la reducción de dosis.

7.3.2 Evaluación del método

El impacto del establecimiento de varios miliAmpere-segundos (mAs) en la calidad de la imagen, ha sido estudiada usando el fantoma femenino de tórax de RANDO® (Alderson). Este fantoma está marcado como un esqueleto real de un humano, que simula los tejidos. Los tejidos en el fantoma están diseñados para tener la misma absorción que se tiene en el tejido humano a niveles de exposición normales de radioterapia. La figura 7.7 muestra este fantoma.

Las imágenes con las que se realizaron las pruebas fueron adquiridas con un dispositivo de imagen digital AGFA a 1, 0.8, 0.6, 0.5, 0.4mAs con 70 kV y 80kV. La figura 7.8 muestra el CR de tórax del fantoma.

Para una mejor apreciación de la imagen obtenida con el método, se analizó sólo un fragmento de la imagen de tamaño 512x512 píxeles.

Los resultados de ruido SD para el fragmento de las imágenes dependiendo de los mAs, se muestran en la figura 7.9. En esta figura podemos ver que si la exposición incrementa, el ruido SD de la imagen disminuye.

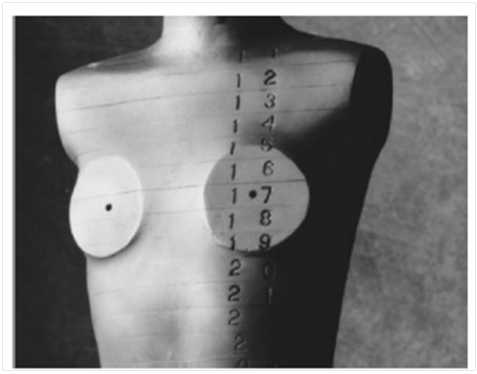


Figura 7.7: Imagen de tórax del fantoma RANDO.

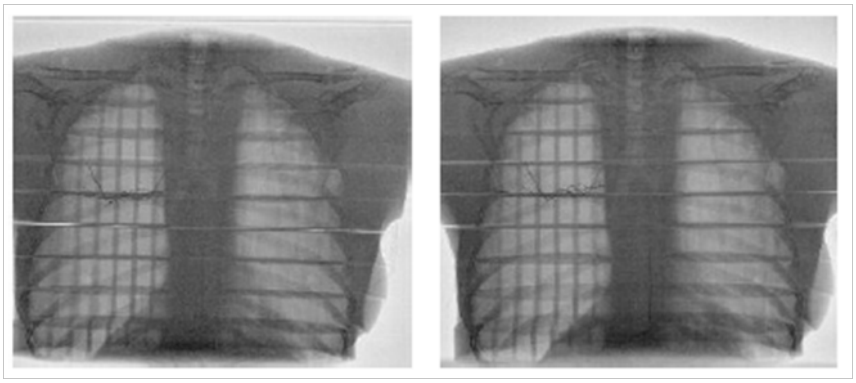


Figura 7.8: CR de tórax del fantoma RANDO para 70V y 80V.

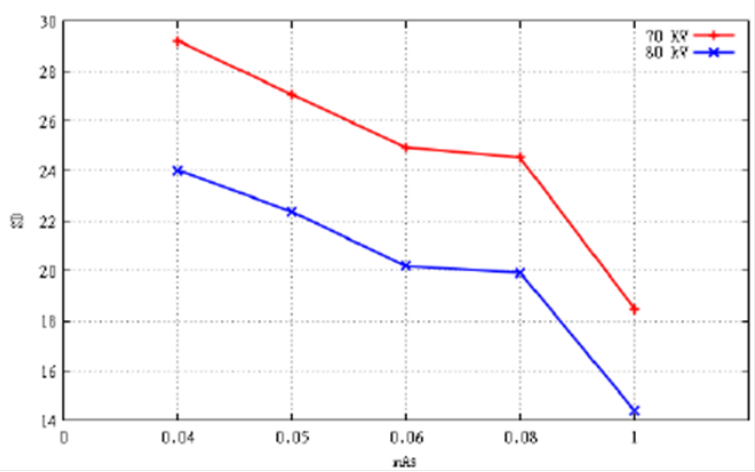


Figura 7.9: Resultados del ruido SD con diferentes mAs 70V y 80V

El proceso para evaluar el método fue el siguiente: se seleccionó A con $0{,}4mAs$ y B con $1mAs$. La imagen con $0{,}4mAs$ se le aplicó el filtro de difusión no lineal. Para determinar la cantidad de mAs que se reducen después de aplicarse el filtro, se estimó el ruido SD del fragmento de la imagen resultante. La tabla 7.9 muestra los resultados. Se puede observar que la imagen filtrada corresponde a una imagen real entre 0.8 y $1mAs$ de exposición.

Tabla 7.9: Comparación del ruido SD con $0{,}4mAs$, $0{,}8mAs$ y $1mAs$ por el FDNL.

	$0{,}4mAs$	$0{,}8mAs$	$1mAs$	Después de FDNL con $0{,}4mAs$
70V	29.2156	24.5312	18.4855	21.0745
80V	24.0432	19.9276	14.4075	17.2756

Con el fin de verificar la consistencia del método, tomamos la imagen con $1mAs$ y se añadió ruido aditivo gaussiano ($\sigma = 0{,}0481$). Los resultados se muestran en la tabla 7.10. Se puede observar que el ruido SD de la imagen resultante es similar al ruido SD correspondiente a una imagen real con $0{,}4mAs$. Además, después de aplicar el filtro de difusión no lineal para la imagen contaminada, se obtienen similares resultados que los presentados en la tabla 7.9. Con esto se puede observar que el ruido SD de la imagen filtrada es similar al ruido SD correspondiente a la imagen real con $0{,}9mAs$ aproximadamente. La comparación se realizó utilizando el ruido SD de la imagen total, que

se calcula tomando la derivación absoluta media del wavelet Daubechy de orden 25, como se describió en el apartado 2.3.1.

Tabla 7.10: Comparación del ruido SD de $1mAs$, $0{,}8mAs$ y $0{,}4mAs$ después de añadir ruido gaussiano.

	$0{,}4mAs$	$0{,}8mAs$	$1mAs$	Después del ruido gaussiano con $1mAs$
70V	29.2156	24.5312	18.4855	29.8719
80V	24.0432	19.9276	14.4075	24.5901

Las figuras 7.10 y 7.11 muestran los fragmentos de tórax con 70kV y 80kV respectivamente con diferente mAs e imagen filtrada por el filtro difusivo no lineal.

7.4 Conclusiones

En este capítulo se ha presentado por un lado, la eliminación de ruido impulsivo, gaussiano y speckle a través del método PGFDNL, y por otro, la reducción de dosis de radiación en CT a través del método FDNL. De la aplicación de estos algoritmos en las imágenes de mamografías y de tórax se derivan las siguientes conclusiones:

- La aplicación de los filtros de *peer group* con la métrica fuzzy y difusivo no lineal en imágenes médicas, proporcionan una buena solución para eliminar el ruido en las imágenes de mamografía. Cuando se desconoce la información del tipo de ruido contenido en la imagen, el mejor método de filtrado es el de PGFDNL. La combinación del método PGMF seguido por el FDNL, mejora la calidad de la imagen obtenida, pero debe ejecutarse en este orden para que el *peer group* y la métrica fuzzy detecte y elimine primero eficientemente los valores extremos del ruido (ruido impulsivo) y después el FDNL para que elimine el ruido gaussiano o speckle presente en la imagen.
- El filtro difusivo no lineal es una buena solución para reducir la dosis de los pacientes, especialmente para niños en aplicaciones pediátricas de Rayos X. Los resultados muestran que la imagen filtrada con el FDNL, el ruido SD corresponde a una imagen real con $0{,}9mAs$ aproximadamente. Se verificó la consistencia del método añadiendo ruido aditivo gaussiano a la imagen, lo que corresponde a una imagen real con $0{,}4mAs$ aproximadamente.

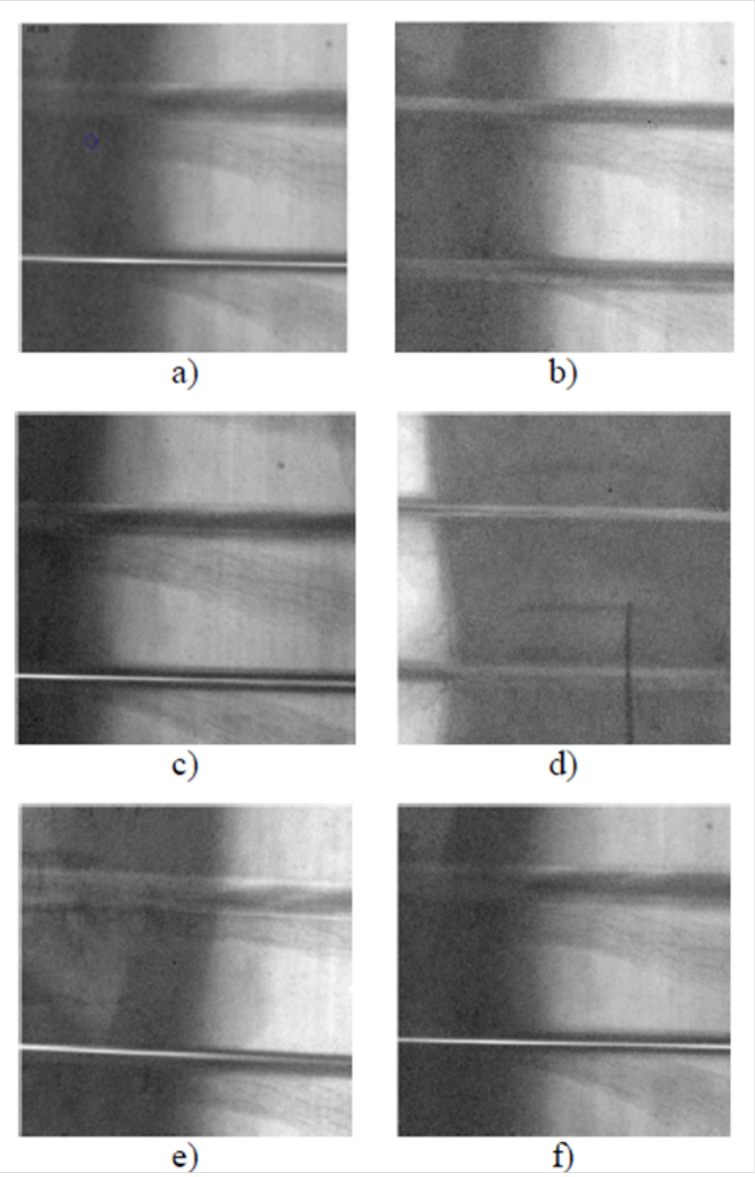


Figura 7.10: Fragmento de tórax del fantoma RANDO con 70kV y a) 0.4mAs b) 0.5 mAs c) 0.6 mAs d) 0.8 mAs e) 1mAs. f)Imagen filtrada por el filtro difusivo no lineal.

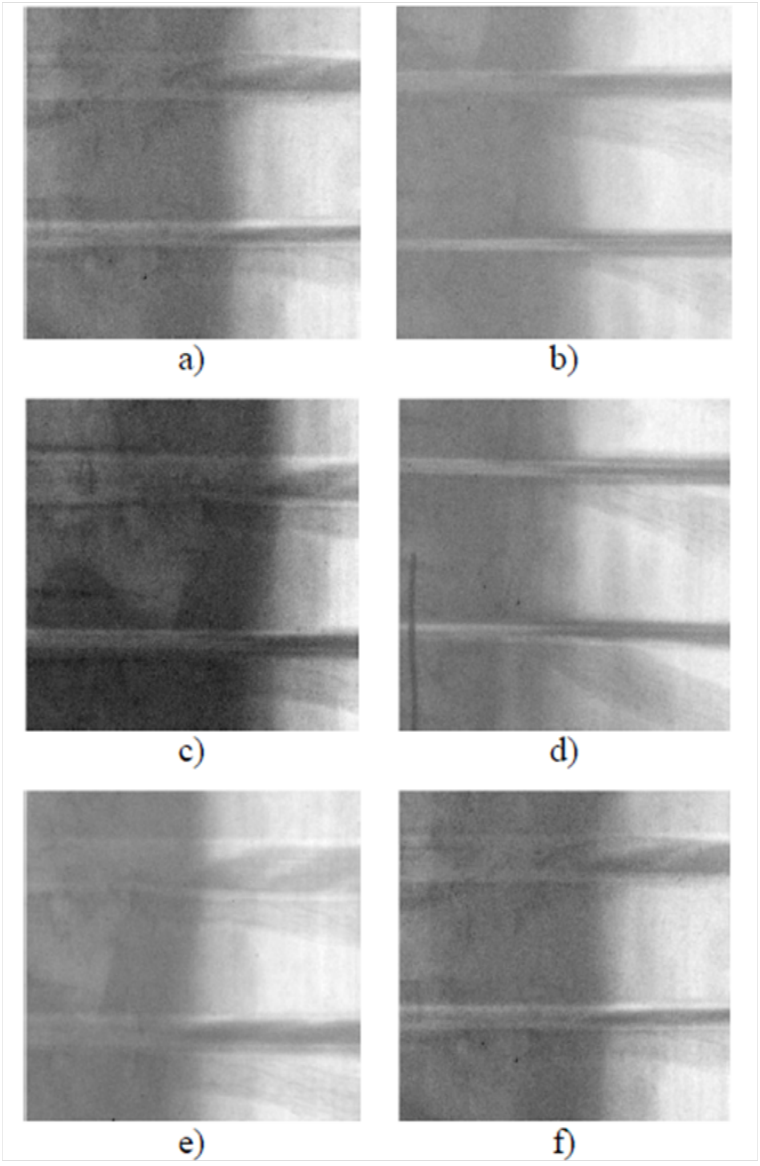


Figura 7.11: Fragmento de tórax del fantoma RANDO con 80kV y a) 0,4mAs b) 0,5mAs c) 0,6mAs d) 0,8mAs e) 1mAs. f)Imagen filtrada por el filtro difusivo no lineal.

Se puede concluir que, el FDNL aplicado en las imágenes con dosis baja, puede producir una imagen con una calidad de dosis alta, de tal manera que el diagnóstico médico sea fiable sin utilizar dosis altas que pueden perjudicar al paciente.

Capítulo 8

Conclusiones finales y trabajo futuro

La calidad de la imagen digital se ve afectada durante la adquisición, almacenamiento y/o transmisión a través de un canal contaminado. El filtrado de imágenes y la correcta percepción de áreas de color es probablemente la tarea más común en el procesamiento digital de imágenes, sobre todo para prepararla para otros procesos.

En esta tesis se ha utilizado hardware que soporta procesamiento paralelo (cores CPU y GPUs) que permiten la paralelización de los algoritmos, con el fin de reducir el coste computacional de los métodos de filtrado para eliminar el ruido impulsivo y gaussiano.

Por un lado, se ha presentado el rendimiento computacional obtenido con el proceso de eliminación de ruido impulsivo y uniforme, y por otro, la calidad resultante después de realizar el proceso de filtrado. El rendimiento computacional se ha mejorado con la paralelización de los algoritmos en CPU y GPU. Para obtener una buena calidad en la imagen filtrada, se ha utilizado una primera etapa de detección de los píxeles ruidosos basándose en el concepto del *peer group* con métricas fuzzy, euclídea o coseno del ángulo y una segunda etapa donde son filtrados aquellos píxeles cuya clasificación es corrupta usando la sustitución del píxel ruidoso por los filtros de mediana o media aritmética. Se proporciona mejor calidad en la imagen filtrada si hay proceso de detección y filtrado, que si se realiza únicamente el proceso de filtrado directamente en la imagen.

Una aproximación de los algoritmos que se han desarrollado para detectar los píxeles ruidosos con el concepto de *peer group*, utiliza dos fases para la detección y trabaja con ventanas de filtrado disjuntas. Se ha analizado esta versión y se ha diseñado un nuevo método con una sola fase en el paso de detección, de tal manera que pueda ser paralizado en GPUs. Los resultados experimentales muestran que este método proporciona buena calidad en la imagen filtrada y es competitivo en calidad si se compara con otros filtros presentados en otros trabajos de investigación.

Concretamente, los resultados de las investigaciones realizadas en esta tesis para eliminar el ruido impulsivo, gaussiano y speckle en las imágenes digitales dan lugar a las siguientes conclusiones:

- **Calidad y métricas**. La calidad obtenida en las imágenes cuando se emplea la métrica fuzzy, es prácticamente la misma (no llega al 1 % PSNR de diferencia) que el de la distancia euclídea al utilizar el *peer group*. Sin embargo, el coste computacional de la métrica fuzzy es mayor (26 %-28 % Megapíxeles por segundo) que con la distancia euclídea. Si el objetivo es conseguir aplicaciones lo más rápido posibles con una calidad aceptable, la detección con la métrica euclídea es una buena opción. En general, la calidad de los filtros implementados comparados con otros filtros presentados en otras investigaciones es competitiva.

  El diseño del método de parametrización para los valores cuando se utiliza el *peer group* y la métrica fuzzy, obtiene valores generales para los umbrales y proporciona una calidad aceptable para diferente tipo de imagen.

- **Algoritmos secuenciales.** Se ha diseñado e implementado algoritmos secuenciales que eliminan el ruido impulsivo, uniforme, gaussiano y speckle. El rendimiento computacional en la versión secuencial, se ve reducido de 27 %-30 % Gflops si se utiliza la versión optimizada (en accesos y cálculos para el *peer group* con cualquiera de las métricas utilizadas) para eliminar el ruido impulsivo. Estos resultados dependen de la cantidad de ruido introducido. Esta optimización ahorra tiempo de procesamiento y cantidad de accesos de lectura a los datos.

  Los algoritmos que utilizan el *peer group* con cualquier métrica, dependen de los parámetros de entrada. Estos parámetros deben evaluarse previamente para poder obtener buena calidad en la imagen filtrada. Como mejora, se ha propuesto un nuevo algoritmo iterativo que utiliza la distancia del coseno del ángulo sin dependencia de los parámetros de entrada que requiere el *peer*

group para eliminar el ruido impulsivo. Además, en este algoritmo la detección y el filtrado están combinados en una misma fase.

En cuanto a la eliminación del ruido gaussiano y speckle, los resultados obtenidos al aplicar el filtro difusivo no lineal demuestran que es un buen filtro para eliminar dichos ruidos.

- **Algoritmos paralelos.**

  Implementados en CPU. Hemos diseñado e implementado algoritmos paralelos en CPU para eliminar el ruido impulsivo en una imagen digital.

  Los resultados experimentales han mostrado, que para tamaños pequeños de imágenes (96×64 o 192×128) la distribución píxeles-cores, presenta mejor rendimiento computacional cuando se utilizan ocho cores. Por el contrario, para tamaños mayores es preferible repartir la carga computacional entre los cores disponibles en el sistema. Esto indica que no es necesario tanto poder de cómputo para procesar el filtrado de imágenes con tamaños pequeños.

  La versión paralela optimizada en accesos y cálculos para el *peer group* utilizando ocho cores disminuye en un 41 % a la no optimizada.

  La versión paralela en *multicores*, reduce el tiempo computacional en un 76 %- 90 % segundos con respecto a la versión secuencial, dependiendo del tamaño de la imagen.

  Implementados en GPU. Se ha diseñado e implementado algoritmos paralelos en GPU para eliminar el ruido impulsivo y uniforme de una imagen.

  En la práctica, hemos observado que el tamaño de la imagen a partir del cual es conveniente utilizar una GPU es 768×512. Si se dispone de más GPUs, convendrá por tanto, asignar a cada GPU una porción de imagen en ese rango.

  La utilización de una GPU con respecto a la versión secuencial, ha obtenido una mejora del 76 %-93 % segundos. El rendimiento obtenido utilizando varias GPUs mejora del 41 % al 56 % segundos con respecto a utilizar una sola GPU, para tamaños mayores o iguales a 3072x2048 píxeles. El uso de una GPU obtiene una mejora del 8 % al 28 % segundos con respecto a la versión paralela en *multicores*, a partir del tamaño de imagen 192x128 píxeles.

  Los beneficios obtenidos del estudio utilizando la GPU para resolver el problema de la reducción de ruido de forma paralela, han mostrado disminución considerable con respecto a

la versión secuencial; con lo que se concluye que el problema de la reducción de ruido impulsivo con el uso del *peer group* y las distintas métricas es completamente paralelizable.

Los parámetros que se han ajustado en CUDA y los resultados obtenidos son los siguientes:

Bloques e hilos. La configuración de bloques de hilos de acuerdo a los resultados obtenidos es mejor si se utilizan tamaños de bloques de 16×16 hilos, excepto para tarjetas gráficas que soportan 1024 hilos por bloque y el tamaño de la imagen es pequeño (96×64 o 192×128), ya que hay cores inactivos y por lo tanto el rendimiento no es el óptimo.

Bytes por píxel a reservar en la memoria de la GPU. Los mejores resultados se han presentado cuando se ha reservado espacio en la GPU con 4 bytes, ya que con esta configuración se gana coalescencia en los accesos a memoria. El cuarto byte se ha aprovechado para guardar el estado del píxel como corrupto o no.

Modo de reserva de memoria. Se han utilizado funciones específicas para acelerar la transferencia de datos entre las memorias CPU-GPU y los resultados han mostrado buen rendimiento final del proceso de filtrado.

Modo de acceso a los datos. En las implementaciones en las que se accedió a los datos en memoria de la GPU a través de la memoria de texturas, han demostrado mejorar considerablemente en el rendimiento computacional. Este acceso obtiene menor coste computacional (31 %-42 % segundos) comparado con el acceso a través de la memoria compartida o directamente a memoria global. Este comportamiento es similar para todos los tamaños de imágenes y cantidad de ruido considerados en esta tesis.

Implementados en CPU-GPU. Se ha diseñado e implementado algoritmos para procesarse en arquitecturas híbridas (CPU-GPU) para eliminar el ruido impulsivo.

En el hardware que se ha probado en esta tesis, se ha visto que sí es conveniente utilizar conjuntamente CPU y GPU como un sistema de coprocesamiento heterogéneo, aunque no necesariamente sea la carga computacional del 50 % y 50 % a cada arquitectura paralela empleada. Esto da lugar a un algoritmo "heurístico", el cual es adaptativo y parametrizado para realizar el proceso de detección y filtrado de ruido, tomando en cuenta el tamaño de la imagen de entrada y el hardware disponible para que se obtenga un buen rendi-

miento computacional y se utilice eficientemente el hardware disponible.

Los mejores resultados obtenidos entre una versión híbrida y la versión con una GPU es del 70 % al 75 % segundos, con varias GPUs su mejora como mínimo es del 32 % y máximo del 58 % segundos, y por último, de un 78 % respecto a la versión paralela en *multicore*. Las mejoras anteriores refieren a tamaños de imagen mayores o iguales que 3072x2048 píxeles, pues a partir de este tamaño de imagen, el coprocesamiento entre CPU-GPU se puede utilizar con buenos resultados.

- **Eliminación del ruido en una imagen, desconociendo el tipo y cantidad de ruido**. Se ha diseñado un método denominado PGFDNL para eliminar el ruido impulsivo, gaussiano y speckle. Este método proporcionan una buena solución cuando se desconoce la información del tipo de ruido contenido en la imagen, especialmente en imágenes médicas. La combinación del método PGMF seguido por el FDNL, mejora la calidad de la imagen obtenida, pero debe ejecutarse en este orden para que el *peer group* y la métrica fuzzy eliminen primero eficientemente los valores extremos del ruido (ruido impulsivo) y después el FDNL para que elimine el ruido gaussiano o speckle presente en la imagen.

- **Aplicación en imágenes médicas.** La aplicación de filtro FDNL en las imágenes CT, es una buena solución para reducir la dosis de radiación en aplicaciones pediátricas de Rayos X, manteniendo la integridad del diagnóstico médico.

  Los resultados muestran que la imagen filtrada con el FDNL, el ruido SD corresponde a una imagen real con alto mAs. Se puede concluir que, el FDNL aplicado en las imágenes con dosis bajas, puede producir una imagen con una calidad de dosis alta, de tal manera que el diagnóstico médico sea fiable sin utilizar dosis altas que pueden perjudicar al paciente.

Esta tesis se ve claramente que es multidisciplinar, porque involucra aspectos matemáticos, informáticos y de aplicación médica, consiguiendo explicar cómo mejorar notablemente el rendimiento de algoritmos de corrección de ruido en imágenes digitales utilizando todos los recursos hardware disponibles, de forma que puedan ser utilizados para aplicaciones que requieran respuesta en tiempo real o con un gran volumen de datos a procesar.

Trabajos Futuros

Los trabajos futuros que se desprenden de la presente memoria son:

- Por el momento, solo se dispone de la implementación secuencial del algoritmo del coseno del ángulo. Se desea analizar el rendimiento en forma paralela en CPU y GPU ya que proporciona una calidad competitiva.
- Paralelizar y analizar el rendimiento computacional del método $FDNL$ (Filtro Difusivo no Lineal) para eliminar el ruido gaussiano y speckle, tanto en CPU como en GPU.
- Mejorar el método de parametrización en los algoritmos que utilizan el *peer group* y la métrica fuzzy, para que se pueda aplicar a una mayor cantidad de imágenes de diferente tipo.
- Analizar y comparar los entornos de programación de las tarjetas gráficas (OpenCL, etc.) para que las implementaciones sean válidas en cualquier modelo de tarjetas gráficas.
- Analizar del rendimiento computacional al aplicar la versión optimizada de accesos y cálculos para el *peer group* en GPUs. Algunos resultados solo han sido presentados en arquitecturas que soporta simple precisión. Como consecuencia, realizar un análisis de la mejora que se puede obtener al utilizar arquitecturas GPUs de doble precisión.
- Buscar problemas reales donde sea conveniente aplicar las técnicas de eliminación de ruido desarrolladas en esta tesis, específicamente en problemas que manejen gran cantidad de datos o necesiten una respuesta en tiempo real.

Publicaciones

Parte de los resultados experimentales obtenidos de las investigaciones realizadas en esta tesis, han dado lugar a las siguientes publicaciones:

Internacionales:

- M. Guadalupe Sánchez, Vicente Vidal and Jordi Bataller. "Comparative Analysis on Metrics and Filters to Reduce Impulsive Noise in Medical Images using GPU". Proceedings en BIOSTEC 2012. Vilamoura, Portugal.

- M. Guadalupe Sánchez, V.Vidal, G. Verdú, P. Mayo. "A Fuzzy Non Lineal Diffusion Filter to Restore Noisy Medical Images". Proceedings en iCBBE 2012. Shangai, China.
- M. G. Sánchez, V. Vidal, J. Bataller and G. Verdú. "Performance Analysis on Several GPU Architectures of an Algorithm for Noise Removal". Proceedings en PDPTA 2012. Las Vegas, Nevada, USA.
- Ma. Guadalupe Sánchez, Vicente Vidal, Gumersindo Verdú, Patricia Mayo and Francisco Rodenas. "Medical Image Restoration with Different Types of Noise". EMBC 2012. Los Angeles, California, USA.
- Ma. Guadalupe Sánchez, Belén Juste, Vicente Vidal, Gumersindo Verdú, Patricia Mayo, Francisco Rodenas. "Estimated Radiation Dose Reduction Using Non-Linear Diffusion Method in Computed Radiography". EMBC 2012. Los Angeles, California, USA.
- J. Arnal, M. Sánchez, V. Vidal, E. Quintana. "An Efficient Image Noise Removal Method on Heterogeneous CPU-GPU Configurations". PMAA 2012. Londres, Inglaterra.
- M. Guadalupe Sánchez, Vicente Vidal, Jordi Bataller and Josep Arnal. "A Fuzzy Filter in GPUs: Fast and Efficient Method for the Impulsive Image Noise Correction". Proceedings de ISCIS 2011. Londres, Inglaterra.
- M. Guadalupe Sánchez, Vicente Vidal, and Jordi Bataller. "Peer group and Fuzzy Metric to Remove Noise in Images using Heterogeneous Computing". HeteroPar2011. Burdeux, Francia.
- María. G. Sánchez, Vicente Vidal, Jordi Bataller, Josep Arnal y Juan Seguí. "Performance Evaluation of using Multi-core and GPU to Remove Noise in Images". Proceedings de CMSSE 2011. Benidorm, España.
- M. Guadalupe Sánchez, Vicente Vidal, Jordi Bataller y Josep Arnal. "Implementing a GPU Fuzzy Filter for Impulsive Image Noise Correction". Proceedings de CMSSE 2010. 2010 Almería, España.

Nacionales:

- Ma. Guadalupe Sánchez, Belén Juste, Vicente Vidal, Gumersindo Verdú, Patricia Mayo y Francisco Rodenas. "Estimación de la

Reducción de la Radiación de Dosis usando el método de Difusión No-Lineal en Radiografías Computarizadas". Sociedad Nuclear Española 2012.

- M. Guadalupe Sánchez, Vicente Vidal, Jordi Bataller, Alejandro Rivera. "Reducción de Ruido Impulsivo Fijo y Uniforme en Imágenes Digitales usando las GPUs". Proceedings de Jornadas de Paralelismo 2011. La Laguna, Tenerife, España.
- Ma. Gpe. Sánchez, Vicente Vidal y Jordi Bataller. "Comparison of Metrics and Filters to Reduce Impulsive Noise in Medical Images on GPU". Mathematical Modelling in Engineering & Human Behaviour 2011. Valencia, España.
- M. Gpe. Sánchez, V. Vidal, G. Verdú, P. Mayo, J. Bataller, F. Rodenas, D. Ginestar. "An Alternating method (Fuzzy + Non-linear Diffusion) to Restore Noisy Medical Images". Sociedad Nuclear Española 2011. Burgos, España.
- M. Guadalupe Sánchez, Jordi Bataller, Vicente Vidal y Josep Arnal. "Estudio sobre la utilización de CUDA para Programas de Corrección de Ruido Impulsivo en Imágenes". Proceedings de CEDI 2010. 2010. Valencia, España.

Publicaciones en proceso de revisión en revista :

- María G. Sánchez, Vicente Vidal, Jordi Bataller y Josep Arnal. "A Fuzzy Metric in GPUs and Multi-cores: Fast and Efficient Method for Impulsive Image Noise Removal". Computer Journal.
- María G. Sánchez, Vicente Vidal, Enrique Quintana y Josep Arnal. " Image noise removal on heterogeneous CPU-GPU configurations". Parallel Computing.

Bibliografía

Almeida, F. y col. (2008). *Introducción a la Programación Paralela*. Paraninfo Cengage Learning (vid. págs. 41, 42).

Álvarez, G., M. Guevara y G. Holguín (2006). «Preprocesamiento de Imágenes Aplicadas a Mamografías Digitales». En: (vid. pág. 2).

Astola, J., P. Haavusto e Y. Neuvo (1990). «Vector Median Filters». En: (vid. pág. 2).

Bioucas, J. M. y M. A. T. Figueiredo (2010). «A Variational Model to Remove the Multiplicative Noise in Ultrasound Images». En: *Journal of Mathematical Imaging and Vision* (vid. pág. 29).

Camarena, J. G. y col. (2008). «Fast Detection and Removal of Impulsive Noise using Peer Group and Fuzzy Metrics». En: (vid. págs. 3, 5, 13, 16, 17, 29, 31, 46, 55, 56, 82, 141).

Camarena, J.G (2009). *Aplicació de Mètriques Fuzzy en la Millora Computacional de Algorismes de Filtratge de Imatges en Color*. Tesis Doctoral (vid. págs. 1, 15, 56).

Camarena, J.G. y col. (2010a). «Some Improvements for Image Filtering using Peer Group Techniques». En: (vid. págs. 3, 5, 26, 29, 31, 46).

— (2010b). «Two step Fuzzy Logic based Method for Impulse noise Detection in Color Images». En: (vid. págs. 2, 3, 5, 17, 24, 29, 31, 126).

Donoho, D.L. (1995). «De-Noising by Soft-Thresholding». En: (vid. págs. 18, 147).

Gaona, E. y col. (2012). «Optimización de la Calidad de Imagen en la Mamografía Analógica y su Comparación con la Mamografía Digital». En: (vid. pág. 2).

George, A. y P.V. Veeramani (1994). «One Some Results in Fuzzy Metric Spaces». En: (vid. pág. 24).

George, A. y P.V Veeramani (1997). «One Some Results of Analysis for Fuzzy Metric Spaces». En: (vid. pág. 24).

Gnanadurai, D. y col. (2009). «Undecimated Double Density Wavelet Transform based Speckle Reduction in SAR Images». En: (vid. pág. 29).

Golub, G. H. y C.F Van Loan (1996). *Matrix Computations*. The Johns Hopkins University Press. 3rd. ed. (vid. pág. 21).

González, R. y R. Woods (2008). *Digital Image Processing*. Person Education. 3rd. ed. (vid. págs. 1, 3, 5, 9, 14-16, 23, 31).

González, E.M, C. M. Hyoung y L. S. Yeol (2011). «Geometric Nonlinear Diffusion Filter and its Application to X-Ray Imaging». En: (vid. págs. 18, 29).

Gregori, V. y S. Romaguera (2000). «Some Properties of Fuzzy Metric Spaces.» En: (vid. pág. 24).

Gregori, V. y Romaguera S. (2004). «Characterizing Completable Fuzzy Metric Spaces». En: (vid. pág. 24).

Guidotti, P. y J. V. Lambers (2009). «Two New Nonlinear Nonlocal Diffusions for Noise Reduction». En: *Journal of Mathematical Imaging and Vision. Volumen 33, Number 1 (), 25-37* (vid. págs. 18, 29).

Guo, L. y col. (2010). «Integration of Recursive Temporal LMMSE Denoising Filter into Video Codec». En: (vid. pág. 30).

Harding, S. (2008). «Evolution of Image Filters on Graphics Processor Units Using Cartesian Genetic Programming». En: (vid. pág. 30).

Huang, Y. M., K. M. Ng y Wen Y.W. (2009). «A New Total Variation Method for Multiplicative Noise Removal». En: (vid. pág. 18).

ICRP (1993). «Managing Patient Dose in Digital Radiology». En: (vid. págs. 2, 30).

ICRP, R (2007). «Radiological Protection in Medicine». En: (vid. pág. 2).

Imrul, J. M. y M. Dey (2012). «An Enhanced Adaptive Vector Median Filtering Technique to Remove High Density Salt-And-Pepper Noise from Microarray Image». En: *International Journal of Computer Applications. Volumen 45, 13* (vid. pág. 17).

Jin, Z. y X Yang (2010). «A Variational Model to Remove the Multiplicative Noise in Ultrasound Images». En: *In Journal of Mathematical Imaging and Vision* (vid. pág. 29).

Juste, B. y col. (2008). «Analysis of CR Dose Reduction in Pediatric Patients, based on Computer simulated Noise Addition». En: (vid. pág. 30).

Kang, C. y J Wang (2009). «Fuzzy Reasoning-based Directional Median Filter Design». En: (vid. pág. 17).

Keeling, S.L. (2003). «Total Variation based Convex Filters for Medical Imaging». En: (vid. pág. 18).

Kenney, C. y col. (2001). «Peer Group Image Enhancement». En: (vid. págs. 3, 16, 25).

Khronos Group, Conecting Software to Silicon.
(2012). «URL: http://www.khronos.org/opencl/». En: (vid. pág. 35).

Kodak (2012). «URL: http://r0k.us/graphics/kodak/index.html». En: (vid. págs. 56, 72).

Kravchenko, V.F, V.I. Ponomaryov y V.I. Pustovoit (2010). «Three-Dimensional Filtration of Multichannel Video Sequences on the Basis of Fuzzy-Set Theory». En: (vid. pág. 30).

Leipsic, J. y col. (2010). «Estimated Radiation Dose Reduction Using Adaptive Statistiscal Iterative Reconstruction in Coronary CT Angiography: The ERASIR Study». En: (vid. pág. 30).

Liu, X. Y. y col. (2012). «On a Modified Diffusion Model for Noise Removal». En: *Journal of Algorithms & Computational Technology. Volume 6, 1, 35-58* (vid. págs. 18, 29).

López, A. (2010). «Métricas Fuzzy. Aplicaciones al Filtrado de Imágenes en Color». En: (vid. pág. 24).

Lukac, R. y col. (2005). «Vector Filtering for Color Imaging». En: (vid. págs. 2, 15, 16).

Mateoa, J.L. y C. A. Fernández (2009). «Finding out General Tendencies in Speckle Noise Reduction in Ultrasound Images». En: (vid. pág. 29).

Mayo, P. (2007). *Evaluación Automatizada de la Calidad de Imagen obtenida de equipos de Mamografía Analógica y Digital. Técnicas de restauración para Imágenes Mamográficas*. Tesis Doctoral (vid. págs. 2, 22).

Mélange, T., M. Nachtegael y E. E. Kerre (2010). «Fuzzy Random Impulse Noise Removal from Colour Image Sequences». En: (vid. págs. 13, 29).

Mendrik, A.M. (2009). «Noise Reduction in Computed Tomography Scans Using 3-D Anisotropic Hybrid Diffusion with Continuous Switch». En: (vid. págs. 18, 29).

Morillas, S., V. Gregori y A. Hervás (2009). «Fuzzy Peer Groups for Reducing Mixed Gaussian-Impulse Noise from Color Images». En: (vid. págs. 3, 5, 11, 29, 31).

Morillas, S., V. Gregori y G. Peris-Fajarnés (2008). «Isolating Impulsive Noise Pixels in Color Images by Peer Group Techniques». En: (vid. págs. 29, 46).

Morillas, S. y col. (2005). «A Fast Impulsive Noise Color Image Filter using Fuzzy Metric.» En: (vid. págs. 55, 56).

Mrázek, P. y M. Navara (2003). «Selection of Optimal Stopping Time for Nonlinear Diffusion Filtering». En: *International Journal of Computing Vision, vol. 52 pp. 189* (vid. págs. 22, 54).

Nair, M. y J. Reji (2011). «An Efficient Directional Weighted Median Switching Filter for Impulse Noise Removal in Medical Images». En: (vid. pág. 30).

NVIDIA (0). «NvidiaHome». En: (vid. pág. 3).

NVIDIA, Corporation (2012). «CUDA Programming Guide Version 4.2, 2012.» En: (vid. págs. 40, 49, 84).

NVIDIA, Corporation Programing (2009). «NVIDIA Programming Guide Version 2.3.1». En: (vid. págs. 38, 41).

Padma, A. y S. Sukanesh R. y Vijayan (2010). «An Efficient Directional Weighted Median Switching Filter for Impulse Noise Removal in Medical Images». En: *International Journal of Computer Applications* (vid. pág. 30).

PEIPA, the Pilot European Image Processing Archive (2003). «URL: http://peipa.essex.ac.uk/info/mias.html». En: (vid. págs. 56, 72, 140).

Pitas, I. y A. N. Venetsanapoulos (1986). «Nonlinear Mean Filters in Image Processing». En: (vid. pág. 17).

Plataniotis, K.N. y A.N. Venetsanopoulos (2000). «Color Image Processing and Applications». En: (vid. págs. 2, 13).

Puvanathasan, P. y K. Bizheva (2009). «Interval Type-II Fuzzy Anisotropic Diffusion Algorithm for Speckle Noise Reduction in Optical Coherence Tomography Images». En: (vid. págs. 18, 29).

Racine, R. y col. (1999). «Speckle Noise and the Detection of Faint Companions». En: (vid. pág. 1).

Reza, H. M. (2011). «An Anisotropic Fourth-Order Diffusion Filter For Image Noise Removal». En: *International Journal of Computer Vision. Volume 92, Number 2 (), 177-191* (vid. págs. 18, 29).

Rudin, L.I., S. Osher y E Fatemi (1992). «Nonlinear Total Variation based Noise Removal Algorithm». En: (vid. pág. 18).

Sánchez, M. G., V. Vidal y J. Bataller (2012). «Comparative Analysis on Metrics and Filters to Reduce Impulsive Noise in Medical Images Using Gpu». En: (vid. págs. 31, 72, 119, 122, 123).

Sánchez, M. G., V. Vidal y J. Bataller J. y Arnal (2010). «Implementig a GPU Fuzzy Filter for Impulsive Image Noise Correction». En: (vid. págs. 31, 72, 92, 104, 117).

Sánchez, M. G. y col. (2010). «Estudio sobre la utilización de CUDA para Programas de Corrección de Ruido Impulsivo en Imágenes». En: (vid. págs. 89, 115, 119).

Sánchez, M. G. y col. (2011a). «A Fuzzy Metric in GPUs: Fast and Efficient Method for the Impulsive Image Noise Correction». En: (vid. págs. 31, 72, 95, 118).

Sánchez, M. G. y col. (2011b). «Performance Evaluation of using Multi-Core and GPU to Remove Noise in Images». En: (vid. págs. 72, 82, 106, 113).

Sánchez, M. G. y col. (2011c). «Reducción de Ruido Impulsivo Fijo y Uniforme en Imágenes Digitales usando las GPUs.» En: (vid. págs. 31, 49, 72, 102, 105, 117, 126).

Sánchez, M. G. y col. (2012a). «Estimated Radiation Dose Reduction Using Non-Linear Diffusion Method in Computed Radiography». En: (vid. pág. 140).

Sánchez, M. G. y col. (2012b). «Medical Image Restoration with Different Types of Noise». En: (vid. pág. 139).

Sánchez, M. G. y col. (2012c). «Performance Analysis on Several GPU Architectures of an Algorithm for Noise Removal». En: (vid. págs. 72, 86, 97, 110).

Sánchez, M.G., V. Vidal y J. Bataller (2011). «Peer Group and Fuzzy Metric to Remove Noise in Images using Heterogeneous Computing». En: (vid. págs. 31, 72, 82, 90, 108, 109, 113).

Silva, A.C. y col. (2010). «Innovations in CT Dose Reduction Strategy: Application of the Adaptive Statiscal Iterative Reconstruction Algorithm». En: (vid. pág. 30).

Singh, S. y col. (2009). «Dose Reduction and Compliance with Pediatric CT Protocols Adapted to Patiente Size, Clinical Indication, and Number of Prior Studies». En: (vid. pág. 30).

Smolka, B. (2005). «Fast Detection and Impulsive Noise Removal in Color Images». En: (vid. págs. 3, 16, 25, 27, 28, 56).

— (2010). «Peer Group Switching Filter for Impulse Noise Reduction in Color Images». En: (vid. págs. 3, 5, 16, 29, 31, 132).

Sudha, S., G.R Suresh y R. Sukanesh (2009). «Comparative Study on Speckle Noise Suppression Techniques for Ultrasound Images». En: *International Journal of Engineering and Technology Vol. 1, 1, 1793-8236* (vid. pág. 29).

Teukolsky, W.T y B.P Betterling (1992). «Numerical Recipes in C». En: (vid. pág. 21).

Toh, K. y N Isa (2010). «Noise Adaptive Fuzzy Switching Median Filter for Salt-and-Pepper Noise Reduction». En: (vid. pág. 17).

Vajda, A (2011). «Multi-core and Many-core Processor Architectures». En: (vid. pág. 3).

Vogel, C.R. y M.E. Oman (1996). «Iterative Methods for Total Variation Denoising». En: (vid. pág. 18).

Weickert, J. (2001). «Efficient Image Segmentation using Partial Differential Equations and Morphology». En: (vid. págs. 18-21).

Weickert, J., B.M. Ter H. R. y M.A. Viergever (1998). «Efficient and Reliable Schemes for Nonlinear Diffusion Filtering». En: (vid. págs. 19-22).