MÁSTER EN COMPUTACIÓN DE ALTAS PRESTACIONES

PROYECTO FIN DE MÁSTER

Detección de bordes por umbralización basada en entropía sobre imágenes multidimensionales en GPU

Autor: Mahy Quintana Díaz.
Directores: Dora Blanco Heras.
Francisco Argüello Pedreira.

Santiago de Compostela, 20 de junio de 2016.

Resumen

En este trabajo, se presenta la extensión de un algoritmo de detección de bordes basado en entropía para imágenes en escala de grises a imágenes hiperespectrales, así como su proyección eficiente sobre GPUs basadas en CUDA.

El algoritmo detector de bordes del que se parte no utiliza las derivadas primera y segunda de la imagen para detectar bordes, a diferencia de la mayoría de los algoritmos, que emplean operadores de convolución (aproximaciones a las derivadas para valores discretos) para detectar las zonas de variación de la luminancia que indiquen la presencia de bordes, este algoritmo se basa en la umbralización de la imagen, haciendo uso de la entropía, para luego proceder a la detección de bordes mediante el cálculo de probabilidades.

Para evaluar el algoritmo detector de bordes basado en entropía, sobre imágenes en escala de grises, se hace una comparativa en calidad de detección de bordes con algoritmos como Canny y LoG que son comunes en la literatura. Además para comprobar la robustez del mismo se realizan varias pruebas frente a diferentes condiciones de ruido y contraste en las imágenes de entrada.

En cuanto a la extensión del algoritmo a imágenes hiperespectrales, se han desarrollado una serie de técnicas que permiten a este algoritmo, inicialmente pensado para imágenes bidimensionales en escala de grises, trabajar con imágenes de una mayor dimensionalidad. Los algoritmos obtenidos han sido evaluados en cuanto a la eficiencia en la detección de bordes comparándolos con otros algoritmos en la bibliografía. Los resultados mostrarán cómo el algoritmo basado en entropía es especialmente eficiente sobre imágenes hiperespectrales con un número elevado de bandas.

En lo que respecta a la implementación en GPU, se exponen diferentes estrategias de paralelización empleadas que permiten al algoritmo sacar mayor provecho de la arquitectura disponible. Con el fin de evaluar el algoritmo y su implementación, este, se ha aplicado sobre imágenes hiperespectrales reales de sensado remoto y se han analizado los resultados. Concretamente, se analiza el tiempo de ejecución en CPU y se compara con el tiempo obtenido en GPU de modo que en GPU se consigue hasta 54$\times$ de mejora respecto a la versión secuencial en CPU para una imagen de tamaño 715x1096 y 102 bandas.

Índice general

Índice de figuras **4**

Índice de tablas **5**

1. Introducción **6**
- 1.1. Teledetección . . . 6
- 1.2. Imágenes hiperespectrales . . . 6
- 1.3. Procesado de imágenes hiperespectrales . . . 7

2. Algoritmos para la detección de bordes **8**
- 2.1. Métodos de detección de bordes en 2 dimensiones . . . 9
  - 2.1.1. Técnicas basadas en el operador gradiente . . . 9
  - 2.1.2. Operadores basados en la laplaciana . . . 10
  - 2.1.3. Operador de Canny . . . 11
  - 2.1.4. Detector de bordes basado en umbralización por entropía. . . . 11
- 2.2. Extensión a imágenes hiperespectrales . . . 13
- 2.3. Método propuesto . . . 13
  - 2.3.1. Detección de bordes en 2 dimensiones . . . 13
  - 2.3.2. Extensión a imágenes hiperespectrales del método basado en entropía . . . 15

3. Detección de bordes basado en entropía en GPU **17**
- 3.1. Visión general de la arquitectura CUDA . . . 17
- 3.2. Técnicas para realizar una implementación eficiente en CUDA . . . 20
- 3.3. Algoritmo detector de bordes en GPU . . . 21
- 3.4. Niveles de paralelismo del algoritmo . . . 21
- 3.5. Implementación a nivel de kernel . . . 22

4. Análisis y Resultados **27**
- 4.1. Procedimiento y equipo utilizado . . . 27
- 4.2. Análisis gráfico de resultados de detección de bordes . . . 28
  - 4.2.1. Imágenes en escala de grises, 2D . . . 28
  - 4.2.2. Detección de bordes en imágenes hiperespectrales, comparativa de métodos de fusión . 34
  - 4.2.3. Detección de bordes en imágenes hiperespectrales, comparativa con otros métodos . . 37
- 4.3. Resultados en cuanto a rendimiento . . . 39
  - 4.3.1. Versión 1: sin reducción de ruido . . . 40
  - 4.3.2. Versión 2: con reducción de ruido . . . 41
- 4.4. Análisis de rendimiento . . . 43
  - 4.4.1. Transferencia de datos entre CPU y GPU . . . 43

5. Conclusiones y trabajo futuro **45**

Bibliografía **46**

Índice de figuras

1.1. Proceso de captación de la información. 6
1.2. Imagen hiperespectral. 7

2.1. Bordes ideales. 8
2.2. Bordes reales. 8
2.3. Imagen de entrada del algoritmo. 14
2.4. División del histograma en valores del background y del foreground. 14
2.5. Fusión de imágenes umbralizadas. 14
2.6. Mapa de bordes resultante. 14

3.1. Esquema del chip GM204 con arquitectura Maxwell 17
3.2. Esquema de un SM del chip GM204 18
3.3. Jerarquía de memoria Maxwell 19
3.4. Niveles de descomposición de los datos. 22
3.5. compactHistogram() 23
3.6. getSumVec() 24
3.7. Reutilización de datos: en la imagen, la mascara de detección de bordes se aplica a los píxeles enmarcados dentro del rectángulo de color negro. 26

4.1. Mejora de los resultados mediante el uso de un filtro de ruido. 28
4.2. Detección de bordes mediante diferentes algoritmos. 29
4.3. Detección de bordes mediante diferentes algoritmos. 30
4.4. Detector de bordes LoG vs detector de bordes basado en umbralización por entropía 31
4.5. Tolerancia al ruido. 32
4.6. Influencia del contraste de las imágenes en la detección de bordes. 33
4.7. Influencia del contraste de las imágenes en la detección de bordes. 34
4.8. Comparativa de métodos de fusión sobre la imagen PaviaU. 35
4.9. Comparativa de métodos de fusión sobre la imagen Salinas. 36
4.10. Resultados de detección de bordes sobre la imagen DC Mall, porción 1. 37
4.11. Resultados de detección de bordes sobre la imagen DC Mall, porción 2. 38
4.12. Resultados de detección de bordes sobre la imagen FLC1. 39
4.13. Imagen del profiler de Nvidia donde se refleja el acaparamiento de los recursos por parte de algunos kernels. 43

Índice de tablas

3.1. Espacios de memoria para la arquitectura CUDA. 18

4.1. Propiedades de los dataset usados en los experimentos 28
4.2. Propiedades de los dataset usados en los experimentos 35
4.3. Propiedades de los dataset usados en los experimentos 37
4.4. Tamaños de bloques óptimos. 40
4.5. Tiempos de ejecución del método detector de bordes, sobre la banda 0 de la imagen Pavia centre, para las distintas versiones CPU y GPU. 40
4.6. Tiempos de ejecución del método detector de bordes, sobre la imagen hiperespectral Pavia centre, para las distintas versiones CPU y GPU. 41
4.7. Tiempos de ejecución del método detector de bordes con reducción de ruido, sobre la banda 0 de la imagen Pavia centre, para las distintas versiones CPU y GPU. 42
4.8. Tiempos de ejecución del método detector de bordes con reducción de ruido sobre la imagen hiperespectral Pavia centre, para las distintas versiones CPU y GPU. 42
4.9. Transferencia de datos. 43
4.10. Transferencias síncronas vs asíncronas. 44
4.11. Transferencias síncronas vs asíncronas. 44

Capítulo 1

Introducción

En este capítulo se presentarán diferentes conceptos claves y se proporcionará una visión general de los problemas que presenta el procesado de imágenes de alta dimensionalidad.

1.1. Teledetección

La teledetección [1] es una disciplina científica que integra un amplio conjunto de conocimientos y tecnologías utilizadas para la observación, el análisis, la interpretación de fenómenos terrestres y planetarios. Sus principales fuentes de información son las medidas y las imágenes obtenidas con la ayuda de plataformas aéreas y espaciales.

Como su nombre indica, la teledetección supone la adquisición de información a distancia, sin contacto directo con el objeto estudiado. La adquisición de información a distancia (figura 1.1) implica la existencia de un flujo de información entre el objeto observado y el captador. El portador de esta información es la radiación electromagnética, esta puede ser emitida por el objeto o proceder de otro cuerpo y haber sido reflejada por este. Todos los cuerpos (planetas, seres vivos, objetos) emiten radiación electromagnética.

El principal emisor de radiación en el sistema solar es el propio Sol cuya radiación, reflejada por la Tierra y los objetos situados en ella, es la más utilizada en teledetección. Otra opción es que el sistema captador incorpore un emisor de radiación cuyo reflejo en la superficie del objeto de estudio lo recoge el propio captador.

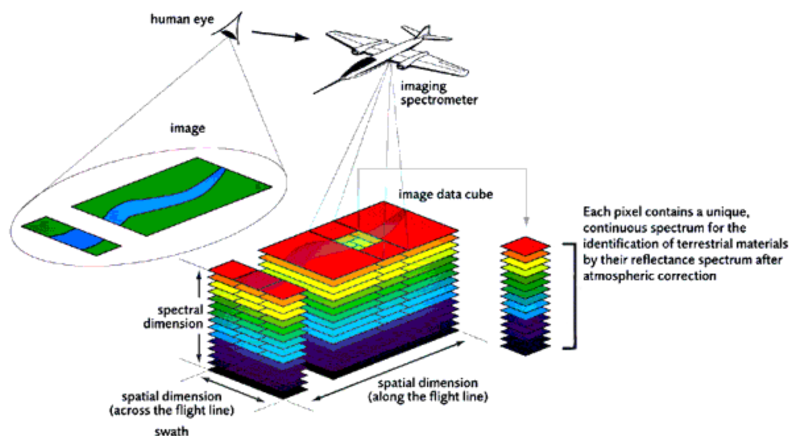


Figura 1.1: Proceso de captación de la información.

1.2. Imágenes hiperespectrales

Una imagen hiperespectral [2] es una imagen que ha sido captada con un sensor espectral capaz de medir la reflectancia en cientos de bandas del espectro electromagnético. Es decir, para cada píxel de una imagen hiperespectral se tiene un conjunto de valores, los cuales se corresponden con las componentes espectrales para distintas longitudes de onda, tal y como se representa en la figura 1.2, donde cada uno de los planos en el eje λ representan a una lámina espectral de la imagen, o lo que es lo mismo, la imagen para una determinada longitud de onda.

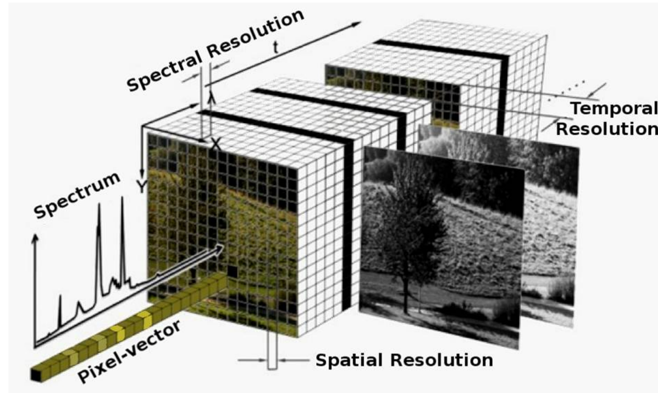


Figura 1.2: Imagen hiperespectral.

La primera de las aplicaciones que tuvieron los sensores hiperespectrales fue la observación de la tierra. Así, diversos sensores hiperespectrales han sido instalados en diferentes satélites y aviones, proporcionando una enorme cantidad de información que permite analizar cambios climáticos, la monitorización de cultivos, ayudar en estudios geológicos, etc. Con el paso de los años, estos sensores empezaron usarse en áreas muy diferentes, incluyendo la arqueología y la conservación de obras de arte [3, 4], la medicina forense [5, 6], el análisis de comida [7, 8], etc.

1.3. Procesado de imágenes hiperespectrales

Uno de los aspectos más importantes para conseguir un mayor y mejor aprovechamiento de las imágenes hiperespectrales es poder operar y procesar en tiempo real las enormes cantidades de datos que producen los sensores. Este procesamiento en tiempo real de las imágenes hiperespectrales se ha llevado a cabo mediante implementaciones paralelas en sistemas HPC [9, 10, 11, 12] con el fin de repartir el elevado coste computacional entre diferentes unidades de procesado. Sin embargo, el principal problema de los sistemas HPC es que son sistemas generalmente caros, voluminosos y difíciles de adaptar a bordo de una unidad de captura hiperespectral. En consecuencia, se hace necesaria una implementación mediante componentes de bajo peso y de bajo consumo de energía y con tiempos de ejecución muy bajos, para lograr un análisis en tiempo real viable y satisfactorio.

Hoy en día las GPUs (Graphics Processing Units) son procesadores multi-núcleo de alto rendimiento, potentes y baratos, que pueden utilizarse para acelerar no sólo las clásicas aplicaciones gráficas, sino también un conjunto muy amplio y diverso de aplicaciones [13]. La razón de esto es que en los últimos años las GPU han evolucionado de ser aceleradores específicos para gráficos a ser procesadores vectoriales programables, con una capacidad de cómputo muy superior a la de las actuales CPUs multi-núcleo [14]. Conjuntamente con esta evolución, se ha desarrollado la arquitectura CUDA (Compute Unified Device Architecture) para las GPUs de Nvidia. La ventaja de CUDA es que ofrece la posibilidad de realizar implementaciones de aplicaciones de propósito general sobre las GPUs, mediante el uso de un lenguaje de programación que extiende C/C++, gracias a lo cual se hace posible el aprovechamiento del enorme nivel de paralelismo que ofrecen las GPUs [15]. Sin embargo, las implementaciones directas de los algoritmos paralelos sobre GPUs ofrecen rendimientos muy irregulares, ya que éstos dependen fuertemente de aspectos como el patrón de acceso a memoria, o el balanceo entre comunicación y computación principalmente, lo que hace que se vuelva imprescindible llevar a cabo una adaptación de cada algoritmo a las características específicas de las GPUs, realizando las optimizaciones que sean necesarias sobre el patrón de acceso a memoria y mejorando la ratio computación/comunicación [13].

Afortunadamente, los algoritmos de procesamiento de imágenes son, en general, buenos candidatos para la implementación sobre un modelo de programación SIMD (Single Instruction, Multiple Data) como el de las GPUs. Esto es así porque todos estos algoritmos requieren, habitualmente, un número grande de computaciones repetitivas sobre conjuntos disjuntos de datos, porque los accesos presentan una elevada localidad, y porque el número de computaciones por dato transferido es alto (alta intensidad aritmética).

Capítulo 2

Algoritmos para la detección de bordes

La detección de bordes es una herramienta fundamental en el procesamiento de imágenes y en visión por computador [16], particularmente en las áreas de detección y extracción de características. Se entiende como borde aquella región donde aparece una fuerte variación del nivel de intensidad en los píxeles adyacentes. Su causa principal es originada por la intersección de varios objetos, con diferentes niveles de reflectancia, que al ser proyectados sobre la cámara generan discontinuidades de intensidad en los píxeles correspondientes. Sin embargo, estas discontinuidades también aparecen de forma no deseada por la presencia del ruido, por el efecto de sombras sobre los propios objetos o por una iluminación no uniforme dentro la escena.

La mayoría de los métodos detectores de bordes se basan en la aplicación del operador derivada en un entorno de vecindad. Si se construye una imagen sintética (figura 2.1) con zonas de alto contraste y se adquiere una
fila de la imagen, se observará una fuerte variación de la intensidad en el entorno de los bordes. Al aplicar el operador derivada, se observa que ésta toma un valor de máximo o mínimo justamente cuando en la transición se pasa de cóncavo a convexo o viceversa, esto es, en el punto de inflexión del borde. Si en vez de emplear la primera derivada se realiza con la segunda, el punto de inflexión de la primera deriva coincidirá con un paso por cero [19] . Ambos razonamientos son empleados para la detección de los bordes. Cuando se aplica el operador gradiente (primera derivada) en la imagen se localizarán valores que tengan un gran valor, normalmente, en el módulo del gradiente. Por el contrario, al emplear la laplaciana (segunda derivada) se trata de detectar píxeles en la imagen que sean pasos por cero.

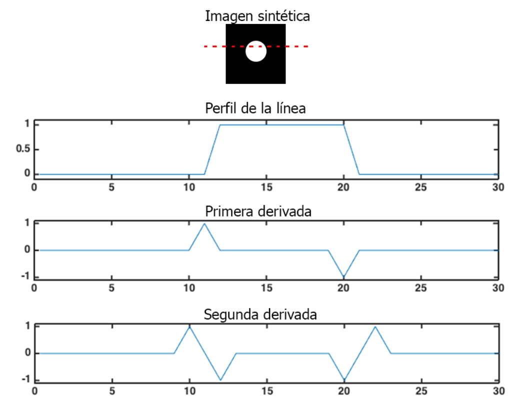


Figura 2.1: Bordes ideales.

El problema reside en los bordes de las escenas reales (figura 2.2), donde el modelo propuesto no está tan claramente definido y no se ajustan a la simplificación indicada. Las diferencias entre el modelo de los bordes y lo almacenado en el ordenador, tras un proceso de formación de la imagen, está en:

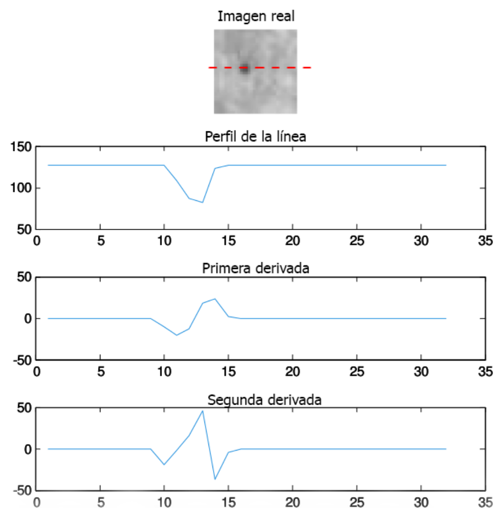


Figura 2.2: Bordes reales.

- Las imágenes digitales son de carácter discreto y no continuo.
- La presencia de ruido en la imagen, lo cual produce variaciones locales de intensidad y genera falsos bordes al aplicar los operadores derivadas. Por esta razón, la detección de bordes está ligada a etapas de eliminación del ruido.
- Los orígenes diversos de los bordes, tales como oclusiones, superficies de diferentes orientaciones, cambios de texturas o de iluminación, reflejos, sombras, etc. hace que sea difícil la detección de los bordes.

2.1. Métodos de detección de bordes en 2 dimensiones

En este apartado se expondrán diferentes métodos empleados para la detección de bordes en imágenes bidimensionales en escala de grises.

2.1.1. Técnicas basadas en el operador gradiente

El operador gradiente [17] aplicado sobre un píxel (x, y) de la imagen, retorna un vector que indica la dirección de máxima variabilidad de la intensidad luminosa y su nivel de variación:

$$\nabla f(x,y) = \begin{bmatrix} \frac{\partial f(x,y)}{\partial x} \\ \frac{\partial f(x,y)}{\partial y} \end{bmatrix} \Rightarrow \left\{ \begin{array}{c} |\nabla f(x,y)| = \sqrt{\left(\frac{\partial f(x,y)}{\partial x}\right)^2 + \left(\frac{\partial f(x,y)}{\partial y}\right)^2} \\ \angle \nabla f(x,y) = arctan\left(\frac{\frac{\partial f(x,y)}{\partial y}}{\frac{\partial f(x,y)}{\partial x}}\right) \end{array} \right. . \tag{2.1}$$

Para saber si un píxel pertenece a un contorno se emplea una técnica denominada umbralización [18], mediante la cual una imagen en escala de grises es binarizada consiguiendo un umbral óptimo T capaz de separar los píxeles en dos regiones, una de zonas claras (foreground) y otra de zonas oscuras (background). En este caso, un píxel se considerará que pertenece a un contorno si el módulo del gradiente supera un cierto umbral T:

$$g(x,y) = \left\{ \begin{array}{l} 1\, |\nabla f(x,y)| > T \\ 0\, |\nabla f(x,y)| \leq T \end{array} \right. . \tag{2.2}$$

La discretización del operador gradiente se basa en las diferencias de los niveles de grises en el entorno de vecindad. En un entorno de 2x2 se usa el operador de Roberts [20] y en una máscara de 3x3 se aproximan por el operador de Prewitt [21], Sobel [22] y el de Frei-Chen [23]. Considerando el caso de un entorno de vecindad 3x3, las derivadas parciales serían:

$$\begin{pmatrix} f_{-1,-1} & f_{-1,0} & f_{-1,1} \\ f_{0,-1} & f_{0,0} & f_{0,1} \\ f_{1,-1} & f_{1,0} & f_{-1,1} \end{pmatrix} \rightarrow \left\{ \begin{array}{l} \left.\frac{\partial f}{\partial x}\right|_{0,0} \cong \frac{(f_{1,-1}+f_{1,0}+f_{1,1})-(f_{-1,-1}+f_{-1,0}+f_{-1,1})}{\Delta x} \\ \left.\frac{\partial f}{\partial y}\right|_{0,0} \cong \frac{(f_{-1,1}+f_{0,1}+f_{1,1})-(f_{-1,-1}+f_{0,-1}+f_{1,-1})}{\Delta y} \end{array} \right. . \tag{2.3}$$

Los incrementos de las variables independientes, Δx e Δy, no se consideran ya que son valores constantes correspondientes a la discretización espacial y en la detección de bordes sólo importa el valor del módulo relativizado. De esta expresión se desprenden las máscaras de Prewitt:

$$\underset{\frac{\partial}{\partial x}}{\begin{pmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{pmatrix}} \quad \underset{\frac{\partial}{\partial y}}{\begin{pmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{pmatrix}} . \tag{2.4}$$

El operador derivada resulta ser muy sensible con el ruido, ya que su respuesta frecuencial tiende a ser un filtro paso alto. Para reducir el realce de éste se propone una máscara que sea la convolución entre el operador derivada y un filtro binomial, obteniéndose las máscaras de Sobel:

$$S = D^T * B = \begin{Bmatrix} -1 & 0 & 1 \end{Bmatrix} * \begin{Bmatrix} 1 \\ 2 \\ 1 \end{Bmatrix} \Rightarrow \left\{ \underset{\frac{\partial}{\partial x}}{\begin{pmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{pmatrix}} \quad \underset{\frac{\partial}{\partial y}}{\begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix}} \right\} . \tag{2.5}$$

Mientras que Prewitt detecta mejor los bordes verticales, Sobel mejora su localización en los bordes diagonales. El operador isotrópico o de Frei-Chen intenta llegar a un compromiso entre ambos:

$$\underset{\frac{\partial}{\partial x}}{\begin{pmatrix} -1 & -\sqrt{2} & -1 \\ 0 & 0 & 0 \\ 1 & \sqrt{2} & 1 \end{pmatrix}} \quad \underset{\frac{\partial}{\partial y}}{\begin{pmatrix} -1 & 0 & 1 \\ -\sqrt{2} & 0 & \sqrt{2} \\ -1 & 0 & 1 \end{pmatrix}} . \tag{2.6}$$

2.1.2. Operadores basados en la laplaciana

En la detección de bordes, empleando la laplaciana [17], se trata de detectar píxeles en la imagen que sean pasos por cero. El operador laplaciana se define en el dominio continuo como:

$$\nabla^2 f(x,y) = \frac{\partial^2 f(x,y)}{\partial x^2} + \frac{\partial^2 f(x,y)}{\partial y^2}. \tag{2.7}$$

Obsérvese que a diferencia del gradiente, la laplaciana no retorna una información vectorial sino un escalar. Al aplicarla la laplaciana sobre una zona homogénea de intensidad, ésta será nula en toda la región. Por el contrario al aplicar este operador sobre un borde, aparecerán en sus alrededores valores positivos y negativos. Este comportamiento indicará la regla de actuación para la detección de los bordes.

La discretización del operador laplaciana para un entorno de 3x3 estará constituida por la suma de las derivadas parciales de segundo orden:

$$\left.\begin{array}{l} \frac{\partial^2 f(x,y)}{\partial x^2} \cong \frac{f(x+1,y)-2f(x,y)+f(x-1,y)}{\Delta x^2} \\ \frac{\partial^2 f(x,y)}{\partial y^2} \cong \frac{f(x,y+1)-2f(x,y)+f(x,y-1)}{\Delta y^2} \end{array}\right\} \Rightarrow \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix}. \tag{2.8}$$

Esta laplaciana se conoce como vecindad a 4. Obsérvese que no se ha cambiado el signo y no se ha normalizado, ya que sólo interesa conocer píxeles que transitan en un entorno de positivos a negativos o viceversa. También se emplea la máscara de vecindad a 8:

$$\begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix}. \tag{2.9}$$

Al aplicarla la laplaciana sobre una imagen real no habrá valores cero en los bordes sino que se aproximan a valores pequeños positivos o negativos, por el carácter discreto tanto de la señal como del operador. Eso sí, estos píxeles etiquetados como bordes, tendrán en su entorno valores positivos y negativos. Los bordes, con este operador, estarán formados por el ancho de un píxel a diferencia del método del gradiente. Además, a diferencia de las máscaras primera derivada, estas son simétricas rotacionalmente, lo que significa que son capaces de de detectar bordes en todas las direcciones espaciales. El mayor inconveniente del operador laplaciana es su sensibilidad al ruido.

Laplaciana de Gauss

Para disminuir el ruido y hacer que el operado laplaciana obtenga mejores resultados se suaviza la imagen con un filtro gaussiano antes de aplicar la laplaciana. Combinando el filtro gaussiano con el operador laplaciana es como surge el operador laplaciana de Gauss o LoG [17]. Desde el punto de vista formal, el operador queda definido como:

$$\nabla^2(f(x,y) * h_\sigma(x,y)) = f(x,y) * \nabla^2(h_\sigma(x,y)) = f(x,y) * LoG(x,y), \tag{2.10}$$

donde:

$$\nabla^2(h_\sigma(x,y)) = LoG(x,y) = \frac{1}{2\pi\sigma^2}\left(\frac{x^2+y^2-\sigma^2}{\sigma^4}\right) e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{2.11}$$

La representación espacial del operador muestra una forma de 'sombrero mejicano'. Notándose de la ecuación que tiene simetría radial, por tanto, sólo será necesario calcular un único cuadrante de los coeficientes. El resto son simétricos. Desde el punto de vista frecuencial, responde a un filtro paso banda bidimensional, cuyo ancho de banda depende de la varianza, σ^2.

El mayor inconveniente de este operador es su alto coste computacional. Una alternativa para la reducción de tiempos es una aproximación basada en la diferencia de la imagen suavizada por dos varianzas distintas, DoG, muy relacionado con el aspecto de diferentes escalas espaciales:

$$f(x,y) * DoG(x,y) \cong (f(x,y) * h_{o1}(x,y)) - (f(x,y) * h_{o2}(x,y)). \tag{2.12}$$

Según Marr y Hildreth [24], este proceder DoG se asemeja al operador LoG, cuando las varianzas se encuentran en una relación de:

$$\sigma_2^2 \diagup \sigma_1^2 = (1{,}6)^2. \tag{2.13}$$

2.1.3. Operador de Canny

Este algoritmo [25] es ampliamente usado en la localización de contornos. Se caracteriza por evitar la ruptura de los bordes de los objetos. Su fundamento se basa en un proceso de optimización, teniendo en cuenta los siguientes objetivos a maximizar:

1. Aumentar la relación señal-ruido de la imagen.

2. Disminuir todo lo posible la distancia entre el borde detectado y el borde real.

3. No identificar un borde por un único píxel, sino por un conjunto de píxeles que tengan una cierta conectividad.

Según Canny, el operador óptimo está en la derivada de Gauss:

$$\nabla(f(x,y) * h_\sigma(x,y)) = f(x,y) * \nabla(h_\sigma(x,y)) = f(x,y) * DroG(x,y). \tag{2.14}$$

Donde el operador DroG es una combinación de suavizado y gradiente, haciendo este operador menos vulnerable al ruido. Aquí también la varianza controlará el grado de suavizado deseado. El operador DroG quedará definido por:

$$DroG(x,y) = \begin{bmatrix} \frac{\partial h_\sigma(x,y)}{\partial x} \\ \frac{\partial h_\sigma(x,y)}{\partial y} \end{bmatrix}. \tag{2.15}$$

Pudiendo ser implementado en una sola máscara. Los pasos que sigue el algoritmo de Canny son tres:

1. Calcular el módulo y el argumento del gradiente de una imagen suavizada aplicando el operador derivada de Gauss.

2. En la dirección del gradiente, eliminar puntos que no sean máximos locales del módulo (equivalente a encontrar el paso por cero en el operador LoG). Se eliminan los píxeles que no sean máximos locales, mejorándose la localización y evitando falsas detecciones.

3. Para la detección de los bordes se emplean dos umbrales, T_1 y T_2, siendo este último el mayor. Con T_2 habría pocos píxeles de bordes y con T_1 habría muchos píxeles de bordes. T_2 es empleado para localizar las semillas de los bordes. A partir de los píxeles que superen el umbral T_2 , se seguirá construyendo el borde mediante la adición de píxeles que superen el umbral T_1 y que sean perpendiculares a la normal del borde.

2.1.4. Detector de bordes basado en umbralización por entropía.

Como ya se vio en el apartado 2.1.1, la umbralización es una técnica mediante la cual una imagen en escala de grises es binarizada consiguiendo un umbral óptimo T que permita dividir la imagen en dos regiones. En la umbralización hay dos posibles situaciones:

1. Umbral único (Global thresholding). Se da cuando solamente hay dos regiones de píxeles, para separarlos se establece un único umbral T. Este tipo de umbral se obtiene fácilmente a partir de histogramas bimodales.

2. Umbral multinivel (Local thresholding). Dada una imagen con varios objetos, para separarlos hace falta más de un umbral, de forma que los píxeles que se encuentren entre cada par de umbrales T_i y T_j representarán a un objeto. Los umbrales elegidos pueden ser de varios tipos, dependiendo de las características tenidas en cuenta para su elección.

Se llama umbralización por entropía a aquellas técnicas que se basan en la entropía [26, 27, 28, 29] para determinar cual es el umbral óptimo T capaz de separar los píxeles en dos regiones.

Método de Shannon

Shannon [30] define la entropía como una medida de incertidumbre de la información contenida en un sistema. La entropía de una variable aleatoria está definida en términos de una distribución de probabilidad, la cual puede mostrar una buena medida de incertidumbre.

Se consideran los píxeles de una imagen convertida a 256 niveles de gris como se indica en la ecuación 2.16 y se separan en dos niveles principales de gris, el primer plano o foreground (definido en la ecuación 2.17) y un fondo de base o background (definido en la ecuación 2.18). La variable g representa los valores de gris. Para imágenes de 8 bits $g = \{0, ..., 255\}$. Siendo G el valor máximo de luminancia:

$$I = \{conjunto\ de\ píxeles\ de\ la\ imagen\ de\ entrada\} \tag{2.16}$$

$$F = \{g \in I/g = 1 : T\} \tag{2.17}$$

$$B = \{g \in I/g = T + 1 : G\}. \tag{2.18}$$

En el contexto de procesamiento de imágenes, el foreground es el conjunto de píxeles con luminancia menor a un cierto valor T, mientras que el background es el conjunto de píxeles con luminancias por encima de este valor de umbral T.

Se calculan las probabilidades estimadas de cada valor de luminancia g haciendo el cociente entre n_g y N, (ecuación 2.19), siendo n_g el número de veces que se repite el valor de luminancia g en la imagen y N la cantidad total de píxeles.

$$p(g) = \frac{n_g}{N} \tag{2.19}$$

$$\sum_{g=1}^{G} p(g) = 1 \tag{2.20}$$

$$N = \sum n_g. \tag{2.21}$$

Las probabilidades del foreground y background están expresadas como se indica en las ecuaciones 2.22 y 2.23, respectivamente:

$$p_f(g),\ 0 \leq g \leq T \tag{2.22}$$

$$p_b(g),\ T + 1 \leq g \leq G. \tag{2.23}$$

Definimos la probabilidad acumulada como expresa la ecuación 2.24:

$$P(g) = \sum_{g=1}^{G} p(g). \tag{2.24}$$

Esta función de probabilidad puede ser considerada como una suma o unión de dos funciones de probabilidad, una para zonas claras (foreground) y otra para zonas oscuras (background). Ecuaciones 2.25 y 2.26.

$$P_f(T) = P_f = \sum_{g=0}^{T} p(g) \tag{2.25}$$

$$P_b(T) = P_b = \sum_{g=T+1}^{G} p(g). \tag{2.26}$$

La entropía de Shannon, paramétricamente dependiente del valor umbral T, está definida, para el foreground y background, como:

$$H_f(T) = -\sum_{g=0}^{T} p_f(g) \cdot \log p_f(g) \tag{2.27}$$

$$H_b(T) = -\sum_{g=T+1}^{G} p_b(g) \cdot \log p_b(g). \tag{2.28}$$

La suma de estas dos expresiones puede ser denotada como $H(T)$ definida mediante la ecuación 2.29:

$$H(T) = H_f(T) + H_b(T). \tag{2.29}$$

Usando las ecuaciones 2.27 y 2.28, se reemplaza obteniendo:

$$H(T) = \left(-\sum_{g=0}^{T} p_f(g) \cdot \log p_f(g) \right) + \left(-\sum_{g=T+1}^{G} p_b(g) \cdot \log p_b(g) \right). \tag{2.30}$$

El umbral óptimo será entonces aquel que maximice esta entropía global:

$$T^* = Max\{H(T)\}. \tag{2.31}$$

Una vez obtenido el umbral óptimo, se crea una imagen binaria, $g(x,y)$, a partir de la siguiente condición, donde $f(x,y)$ representa el nivel de gris del píxel en la posición (x,y) de la imagen:

$$g(x,y) = \begin{cases} 1 \; si \; f(x,y) > T \\ 0 \; si \; f(x,y) \leq T \end{cases}. \tag{2.32}$$

Sobre esta imagen binaria es donde se lleva a cabo la detección de bordes, para ello se utiliza una máscara de 3x3 píxeles que se encarga de calcular la probabilidad [31] de que el píxel central sea o no un borde. Para almacenar el mapa de bordes se crea una nueva imagen binaria donde se irán guardando los píxeles clasificados como bordes.

Además del método de Shannon también existen otros métodos basados en entropía como puede ser el método del Tsallis [26].

2.2. Extensión a imágenes hiperespectrales

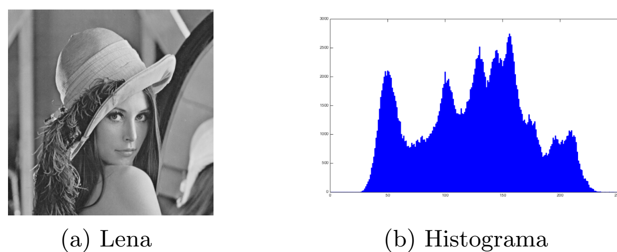
Las imágenes de entrada para los algoritmos vistos en la sección anterior, son imágenes en escala de grises, sin embargo, la aplicación de estos métodos a imágenes hiperespectrales no es una tarea trivial, previamente hay que procesar la imagen para adecuarla a las características de los algoritmos. Para el caso concreto de detección de bordes, existen diferentes algoritmos capaces de trabajar con imágenes hiperespectrales [32, 33, 34, 35, 36, 37], pero no todos trabajan con la información de la misma manera, mientras que unos dividen la imagen en el dominio espectral para procesar cada banda por separado, otros procesan la información espectral de forma conjunta.

En la literatura existen dos aproximaciones diferentes a la hora de detectar bordes en imágenes con más de una banda. El primer enfoque [32] consiste en tratar cada una de las bandas como una imagen en escala de grises y aplicar el método detector de bordes a cada una de las bandas, el resultado que se obtiene es un conjunto de mapas de bordes (tantos como bandas tiene la imagen) que posteriormente deben fusionarse. Por otra parte, el segundo enfoque [33, 34, 35, 36, 37] trata cada píxel, de una imagen hiperespectral, como un vector en el dominio espectral y realiza la detección de bordes en este dominio.

2.3. Método propuesto

2.3.1. Detección de bordes en 2 dimensiones

El método propuesto [38] para este proyecto es un método de detección de bordes basado en umbralización por entropía. Concretamente se trata de un método basado en dos algoritmos de umbralización de umbral único, donde se utiliza tanto la entropía de Shannon como la de Tsallis para detectar múltiples niveles umbrales capaces de segmentar correctamente la imagen.

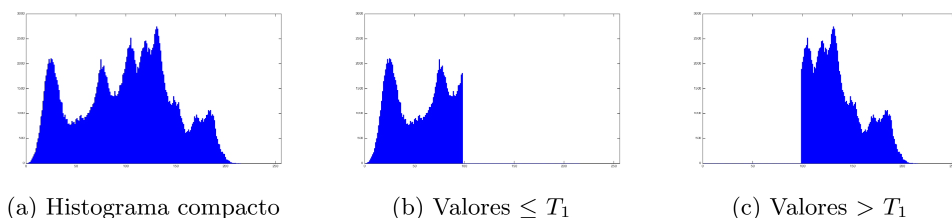


(a) Lena (b) Histograma

Figura 2.3: Imagen de entrada del algoritmo.

El primer paso del algoritmo consiste en calcular el histograma y compactarlo eliminando aquellos valores cuyo número de ocurrencias sea igual a 0. Este histograma, compactado, será la base sobre la cual trabajarán los algoritmos de Shannon y TSallis.

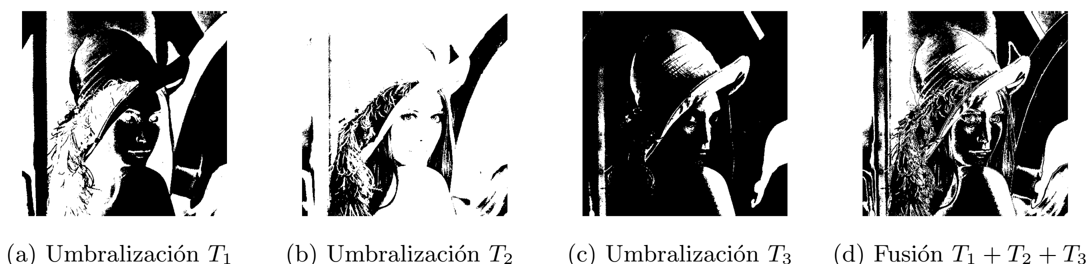
El siguiente paso consiste en calcular el umbral óptimo (T_1), que maximice la entropía de Shannon, y el lugar que ocupa este valor en el histograma (Loc). El umbral se utilizará para dividir la imagen en dos, background y foreground, y el valor Loc se utilizará para dividir el histograma en dos.



(a) Histograma compacto (b) Valores $\leq T_1$ (c) Valores $> T_1$

Figura 2.4: División del histograma en valores del background y del foreground.

A continuación, sobre el histograma de cada una de estas nuevas imágenes (background y foreground) se aplicará el método de TSallis para encontrar los umbrales ópticos (T_2 y T_3) que subdividan las imágenes. Con los umbrales calculados anteriormente, T_1, T_2 y T_3, y la imagen original, se crea una imagen binaria sobre la cual se aplicará la detección de bordes.



(a) Umbralización T_1 (b) Umbralización T_2 (c) Umbralización T_3 (d) Fusión $T_1 + T_2 + T_3$

Figura 2.5: Fusión de imágenes umbralizadas.

La detección de bordes consiste en aplicar una máscara de 3x3 píxeles que comprobará la probabilidad de que el píxel central pertenezca o no a un borde

Figura 2.6: Mapa de bordes resultante.

2.3.2. Extensión a imágenes hiperespectrales del método basado en entropía

Para la extensión del método propuesto a imágenes hiperespectrales, se han utilizado diferentes técnicas, todas ellas basadas en el primer enfoque expuesto en la sección 2.2, es decir, se procesará cada banda de forma independiente como si fuera una imagen bidimensional en escala de grises y posteriormente se aplicará un proceso de fusión con el objetivo de obtener un único mapa de bordes. En el algoritmo 1 se muestra el pseudocódigo del algoritmo para ser aplicado a una imagen en escala de grises, de modo que podría ser aplicado a una banda de la imagen hiperespectral.

Algoritmo 1 Detector de bordes basado en entropía

```
Require: El dataset debe estar previamente centrado y almacenado en memoria, a parte del dataset, en memoria deben haber otros datos,
    como el número de píxeles que contiene una banda, además de estructuras de datos necesarias para el algoritmo.
 1: procedure DETECTAR_BORDES
 2:     #pragma omp parallel for
 3:     for all 1...n_bandas do
 4:         getHistogram()                          ▷ Obtener el histograma de la imagen.
 5:         dim = 256;
 6:         for (i in 0...256) do                   ▷ Se calcula la posible nueva dimensión del histograma compactado.
 7:             if hitogram[i]==0 then
 8:                 dim- -
 9:             end if
10:         end for
11:         if dim<256 then                         ▷ Se comprueba si es necesario compactar el histograma.
12:             j=0
13:             for (i in 0...256) do
14:                 if (hitogram[i]!=0) then
15:                     histogramIndex[j]=i         ▷ Inicialización del índice del histograma.
16:                     hitogram[j]= hitogram[i]    ▷ Reubicación de los datos del histograma (compactación).
17:                     j++
18:                 end if
19:             end for
20:         else                                    ▷ En caso de que no haya que compactar, solo hay que inicializar el índice del histograma.
21:             for (i in 0...256) do
22:                 histogramIndex[i]=i             ▷ Inicialización del índice del histograma
23:             end for
24:         end if
25:         getSum()                                ▷ Obtener la suma de los valores de frecuencia del histograma.
26:         normVec()                               ▷ Normalizar el histograma.
27:         shannon()                               ▷ Calcular umbral óptico T1 que divida el histograma en background y foreground.
28:         getSumVec()                             ▷ Obtener la suma de los valores de frecuencia del histograma pertenenciente al foreground.
29:         normVec()                               ▷ Normalizar histograma del foreground.
30:         tsallis()                               ▷ Calcular umbral óptico T2 que divida el histograma del foreground.
31:         getSumVec()                             ▷ Obtener la suma de los valores de frecuencia del histograma pertenenciente al background.
32:         normVec()                               ▷ Normalizar histograma del background.
33:         tsallis()                               ▷ Calcular umbral óptico T3 que divida el histograma del background.
34:         testT()                                 ▷ Generar imagen umbralizada a partir de la imagen original y de los valores T1T2 y T3.
35:         edgeDetector():                         ▷ Detectar bordes sobre la imagen creada por el kernel anterior.
36:     end for
37: end procedure
```

Una vez conocido el algoritmo, se van a presentar las diferentes técnicas empleadas en la fusión de la información. La diferencia entre estas técnicas reside en la forma y lugar del algoritmo donde se fusiona la información.

Estas técnicas podrían dividirse en tres grupos, el primer grupo engloba aquellas técnicas que reducen la dimensionalidad de los datos para luego procesarlos, es decir tratan de reducir el número de bandas de la imagen hiperespectral antes de realizar la detección de bordes. En el segundo grupo estarían las técnicas que hacen todo lo contrario, primero procesan toda la información y luego reducen la dimensionalidad de los datos de salida, es decir, procesan todas las bandas de la imagen y a continuación fusionan los mapas de bordes obtenidos. Y por último existen otras técnicas que son un híbrido del primer y segundo grupo, estas técnicas reducen la dimensionalidad de los datos antes de ser procesados y también de los datos de salida en caso de ser necesario. Como se puede apreciar, el factor común a todas las técnicas es la etapa de reducción de la dimensionalidad, esta etapa es muy importante ya que va a permitir reducir imágenes de cientos de bandas a una imagen con la información concentrada únicamente en las primeras bandas, llegando, en el mejor de los casos a obtener una única banda útil.

- **Grupo 1:**

  Fusión de bandas a través del cálculo de medias: Este método de fusión es muy básico, consiste en calcular la imagen promedio de todas las bandas reduciendo la imagen hiperespectral a única imagen en escala de grises. Sobre esta imagen es donde se aplicará el método propuesto.

Análisis de características principales(PCA) + fusión de bandas a través del cálculo de medias: El análisis de componentes principales se aplica a la imagen hiperespectral con el objetivo de conseguir una nueva imagen hiperespectral donde las características principales se encuentren agrupadas en las primeras bandas. Una vez obtenida esta nueva imagen, se fusionan las x primeras bandas de interés mediante el cálculo de medias, tal y como hace la técnica anterior. Tras esta segunda reducción de la dimensionalidad se consigue una única imagen en escala de grises sobre la cual se aplicará el método propuesto. El resultado será un único mapa de bordes.

- **Grupo 2:**

  Fusión de mapas de bordes: Se computa cada una de las bandas por separado, como imágenes en escala de grises, y posteriormente se fusionan los mapas de bordes obtenidos. La fusión se realiza de la siguiente manera: si un píxel aparece en más del $x\,\%$ de las bandas, entonces, se considera borde. El valor x representa el porcentaje mínimo de bandas en las que tiene que aparecer el mismo píxel como borde para ser considerado borde en la fusión, el valor óptimo de este parámetro se determina por ensayo y error, ya que es altamente dependiente de la imagen.

- **Grupo 3:**

  Fusión de imágenes umbralizadas de cada banda: Este método calcula las imágenes umbralizadas de cada banda y las fusiona justo antes de la etapa de detección de bordes (línea 35 del algoritmo 1), de esta manera se obtiene un único mapa de bordes como resultado final. La fusión se realiza de la misma manera que en punto anterior: si un píxel aparece en más del $x\,\%$ de las imágenes, entonces, se considera borde. El valor x representa el porcentaje mínimo de imágenes en las que tiene que aparecer el mismo píxel para ser considerado píxel válido en la fusión, el valor óptimo de este parámetro se determina por ensayo y error, ya que es altamente dependiente de la imagen.

  Fusión de entropías a través del cálculo de medias + fusión de bandas a través del cálculo de medias: La fusión de entropías consiste en calcular la media de todos los valores de entropía para cada una de las bandas obteniendo T_1', T_2' y T_3'. Además también se hallará la imagen promedio de todas las bandas. Una vez obtenidos estos datos, la etapa de umbralización (línea 34 del algoritmo 1) se reduce a un único caso y por lo tanto la detección de bordes también, obteniendo como resultado un único mapa de bordes.

  Análisis de características principales (PCA) + fusión de mapas de bordes: Tras reducir la dimensionalidad de los datos de entrada empleando el análisis de componentes principales se aplica el método propuesto únicamente a las x primeras bandas de interés, obteniendo así un número reducido de mapas de bordes. Por ejemplo para una imagen de entrada de 103 bandas se puede obtener un total de 3 mapas de bordes. El paso siguiente sería fusionar estos mapas y se haría de la misma forma que en la técnica del grupo 2, fusión de mapas de bordes.

  Análisis de características principales (PCA) + fusión de imágenes umbralizadas: Al igual que en la técnica anterior, esta técnica utiliza PCA para reducir la dimensionalidad de los datos de entrada. Los nuevos datos son procesados hasta la etapa de umbralización (línea 35 del algoritmo) donde se fusionan para producirse a una única imagen que será sobre la cual se aplique la detección de bordes, de esta manera se obtiene un único mapa de bordes como resultado final. La fusión se realiza exactamente igual que en la primera técnica de este grupo, fusión de imágenes umbralizadas.

Capítulo 3

Detección de bordes basado en entropía en GPU

En este capítulo se explica en detalle la adaptación del algoritmo de detección de bordes basado en entropía a GPU usando CUDA. Se comienza describiendo las características básicas de las GPUs de NVidia y de la arquitectura CUDA. A continuación se explican en detalle las técnicas de mejora que se han aplicado en particular en este trabajo para aprovechar al máximo la arquitectura y, por tanto, reducir el tiempo de ejecución del algoritmo. Finalmente se explican los detalles de la implementación.

3.1. Visión general de la arquitectura CUDA

CUDA (Computer Unified Device Arquitecture) es una arquitectura para computación paralela de propósito general, que permite aprovechar la potencia de cómputo paralelo de las GPUs de Nvidia para resolver problemas computacionalmente costosos de una forma más eficiente que sobre CPU. Esta arquitectura fue introducida en noviembre de 2006 por NVIDIA y desde entonces no ha parado de evolucionar. En la actualidad, su último modelo de microarquitectura disponible en el mercado es la Pascal [39]. Para la realización de este proyecto se ha empleado una Nvidia GTX 970, de arquitectura Maxwell [40], cuyo esquema puede verse a continuación (figura 3.1) y de la cual se hablará en el resto del documento:

Figura 3.1: Esquema del chip GM204 con arquitectura Maxwell

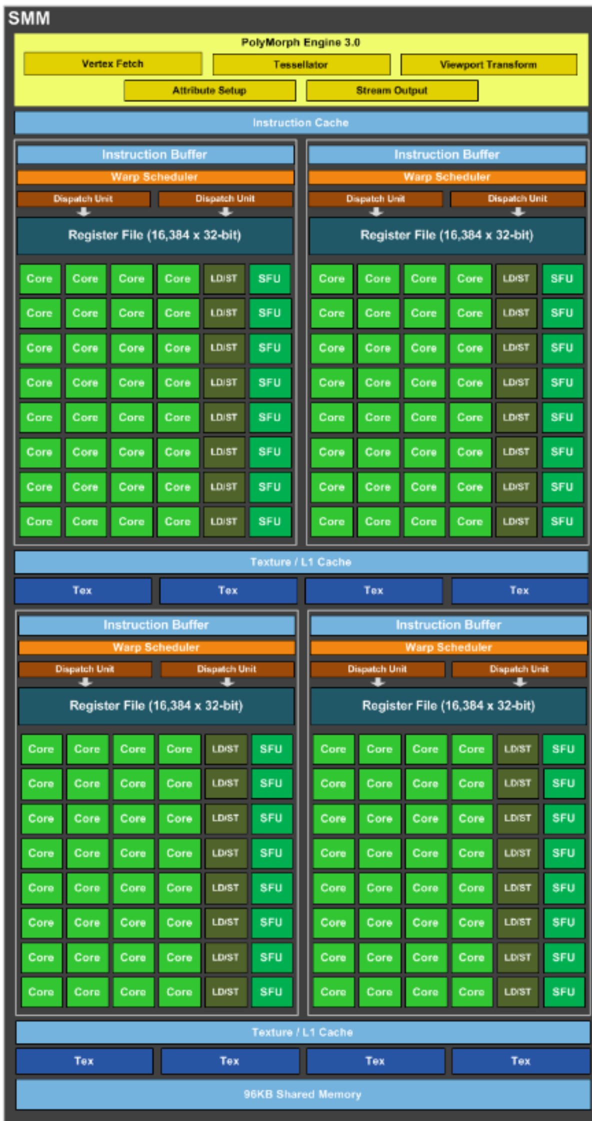


Figura 3.2: Esquema de un SM del chip GM204

La arquitectura CUDA está organizada en un conjunto de multiprocesadores o SMs (Streaming Multiprocessors), cada uno de los cuales contiene varios núcleos o SPs (Streaming Processors), siendo este número de núcleos por cada SM dependiente del modelo de tarjeta utilizado. Cada uno de estos SMs es capaz de manejar cientos de threads para ejecutar porciones de código de forma paralela, basadas en un modelo SIMD (Single Instruction, Multiple Data). Típicamente estas porciones de código, denominadas kernels, son ejecutadas de forma paralela sobre la GPU por miles de threads [15]. En la figura 3.2 puede verse una representación simplificada de los elementos que componen cada uno de los SM en la arquitectura Maxwell.

Desde el punto de vista del programador y compilador, los threads son manejados y agrupados en bloques de threads, y estos a su vez organizados en grids. Los bloques de threads son conjuntos de threads que pueden colaborar entre ellos mediante el uso de barreras de sincronización y de una memoria compartida, que tiene un tiempo de vida igual al tiempo en que el bloque de threads esté activo. Los grids son conjuntos de bloques de threads que ejecutan el mismo kernel, y que no poseen estos mecanismos de comunicación, es decir, distintos bloques no pueden sincronizarse entre ellos ni compartir datos en memoria compartida. Esta restricción resulta esencial a la hora de diseñar algoritmos que se adapten bien a la GPU, pues hay que tener en cuenta que dentro de una misma llamada de un kernel, un thread no podrá requerir datos que sean generados fuera de su propio bloque.

Por otra parte, desde el punto de vista del hardware, es importante también tener presente que los threads son gestionados y ejecutados en grupos de 32 threads, denominados warps, los cuales constituyen el tamaño mínimo de procesado SIMD (Single Instruction, Multiple Data). En la arquitectura Maxwell, además, cada SM contiene cuatro "Warp Schedulers", cada uno de ellos es capaz de enviar o despachar 2 instrucciones por warp en cada ciclo de reloj, lo que les permite ser capaces de ejecutar de manera concurrente cuatro warps diferentes [40].

Memoria	Loc	Cacheable	Acceso	Alcance	Tiempo vida
Registros	on-chip	N/A	R/W	Un thread	Thread
Local	off-chip	Si	R/W	Un thread	Thread
Compartida	on-chip	N/A	R/W	Threads de un bloque	Bloque
Global	off-chip	Si	R/W	Todos threads + host	Aplicación
Texturas	off-chip	Si	R/W	Todos threads + host	Aplicación
Constantes	off-chip	Si	R	Todos threads + host	Aplicación

Tabla 3.1: Espacios de memoria para la arquitectura CUDA.

En lo referente a la memoria de la arquitectura CUDA, físicamente está dividida en una memoria DRAM

común a todos los SMs, off-chip, y una memoria incluida dentro de cada SM, on-chip. Sin embargo, desde el punto de vista de la computación, la memoria está organizada en varios espacios de memoria diferentes con características de acceso y tiempo de vida distintos. En la tabla 3.1 pueden verse los distintos espacios de memoria existentes así como sus principales características, y en la figura 3.3 puede verse una representación de la jerarquía de niveles.

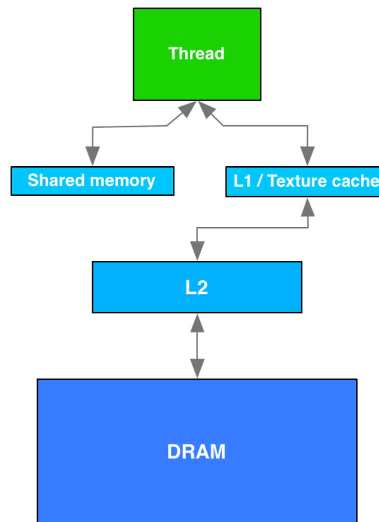


Figura 3.3: Jerarquía de memoria Maxwell

Uno de los aspectos claves a la hora de reducir la latencia en los accesos a memoria global es la coalescencia, que es la capacidad que tiene la GPU de gestionar en una única transacción de memoria las peticiones a memoria global de un half-warp bajo ciertas restricciones en el patrón de acceso [15].

Por otro lado el hecho de que la memoria compartida esté integrada on-chip dentro de cada SM hace que tenga una latencia mucho menor que los espacios de memoria local y global. Además para conseguir un ancho de banda alto la memoria compartida está dividida en módulos de memoria de igual tamaño denominados bancos, a los cuales se accede de forma simultánea. De esta forma cualquier petición de lectura o escritura realizada por un mismo warp sobre n direcciones que caigan en n bancos distintos son gestionadas de manera simultánea, mientras que si k peticiones recaen sobre el mismo banco entonces habrá conflicto en los accesos, y éstas k peticiones serán gestionadas de forma secuencial [15].

Los memorias de constantes y de texturas son memorias cacheables diseñadas para obtener un mayor rendimiento para determinados patrones de acceso. La memoria de constantes está optimizada para cuando los threads de un half-warp leen la misma dirección. La memoria de texturas está optimizada para explotar la localidad espacial 2D, y resulta más útil en patrones de acceso irregulares que la memoria global [15].

Es importante decir que el sistema de memoria en la arquitectura Maxwell cuenta con un completo sistema caché implementado a través de una memoria L1 por cada SM, y una memoria L2 de 2048 KB unificada para todos ellos [40]. En arquitecturas anteriores, como la Fermi o la Kepler, la memoria compartida y la cache L1 compartían el mismo espacio on-chip, pudiendo configurarse a media, asignado más o menos espacio a cada una de las ellas, en cambio, en la arquitectura Maxwell se proporciona un espacio dedicado a cada una de estas memorias, lo que hace que se disponga de más memoria compartida por SM, exactamente 32KB más de memoria en comparación con la arquitectura anterior [41].

Otro aspecto interesante introducido en esta arquitectura es la capacidad de realizar operaciones atómicas sobre objetos en memoria compartida; anteriormente esto solo era posible sobre la memoria global. Teniendo en cuenta las latencias de accesos de estos tipos de memorias (global vs compartida) esto supone una mejora importante ya que permite una reducción de tiempos considerable [42, 43].

3.2. Técnicas para realizar una implementación eficiente en CUDA

Un aspecto básico para llevar a cabo una implementación eficiente es saber cómo trabaja el algoritmo con los datos, es decir, saber la forma en que se realizan las lecturas y las escrituras, ya que deben realizarse los accesos de la manera más eficiente posible para reducir tiempos de espera. Normalmente, las imágenes hiperespectrales se almacenan en formato “raw”, pudiendo tener dos codificaciones posibles que dan lugar a un orden de almacenamiento de datos en memoria diferentes:

- **Píxel-vector:** Con esta codificación, la información espectral es almacenada píxel a píxel, es decir, primero se almacenan todos los valores de reflectancia del primer píxel, a continuación todos los valores del segundo píxel, etc.
- **Banda-vector:** Esta codificación almacena la información banda a banda, entendiendo que una banda está formada por todos los valores de reflectancia para una longitud de onda determinada.

Como se vio en el apartado 2.3, el método propuesto procesa banda a banda la imagen hiperespectral, por lo que la codificación adecuada será “banda-vector”. Esta codificación va a permitir aprovechar mejor la jerarquía de memoria y por tanto, explotar mucho mejor la localidad de los datos, repercutiendo de forma positiva en la eficiencia del algoritmo.

A continuación se exponen una serie de técnicas [15] que se han utilizado en la implementación con el objetivo de crear unos kernels lo más eficientes posible:

1. **Utilización de memoria compartida siempre que sea posible**: Los threads de un bloque disponen de dos mecanismos para compartir datos, o bien usar la memoria global, o bien usar la memoria compartida. Ante estas dos posibilidades siempre va a ser mejor utilizar la memoria que ofrezca latencias más bajas, siendo la memoria compartida la opción adecuada por su mayor rapidez.
2. **Uso de operaciones atómicas:** Una operación atómica consta de una serie de instrucciones que son ejecutadas de forma indivisible, previniendo que cualquier otro thread pueda inferir en dicho proceso. Este tipo de operaciones garantizan que, cuando varios hilos escriban sobre la misma posición de memoria, se acceda a ella a través de una sección crítica, actualizando de manera secuencial su valor. Como se puede deducir, las operaciones atómicas son clave en el uso de recursos compartidos y cobran especial interés cuando el recurso es compartido por cientos de threads. CUDA proporciona operaciones atómicas en diferentes niveles de memoria, siendo las operaciones atómicas sobre memoria compartida las más rápidas.
3. **Streams**: Un stream es una secuencia de instrucciones o kernels que se ejecutarán en orden. Si se dispone de varios streams, estos podrán ejecutarse concurrentemente unos con otros, permitiendo así ejecuciones paralelas de diferentes kernels en un mismo dispositivo.
4. **Transferencias asíncronas:** Este tipo de transferencias permiten solapar la computación con la transferencia de datos, no teniendo que esperar a que lleguen todos los datos al dispositivo para comenzar con la computación.
5. **Page-Locked Host Memory:** Esta forma de reservar memoria en el host permite que las transferencias entre el host y el dispositivo sean mucho más rápidas ya que se aprovecha mejor el ancho de banda disponible.
6. **Dynamic parallelism:** Esta función permite que los subprocesos de la GPU produzcan dinámicamente nuevos subprocesos, por lo que la GPU puede adaptarse a los datos de forma dinámica.
7. **Shuffle instructions:** Las shuffle instructions permiten el intercambio de datos entre los hilos de un mismo warp sin tener que utilizar la memoria compartida para ello, permitiendo así evitar la sincronización entre hilos y acelerar el proceso. Son muy útiles en procesos de reducción.

8. **Evitar conflictos de bancos mediante el correcto alineamiento los datos en memoria**: La memoria compartida se encuentra dividida en módulos del mismo tamaño denominados bancos, a los cuales se puede acceder de forma simultanea. Por tanto, cualquier operación de carga o almacenamiento, realizada por múltiples threads, que accedan a bancos diferentes, pueden ser servidas de forma simultanea, empleándose una única transacción. En cambio, si estas cargas o almacenamientos acceden a un único banco pero a posiciones de memoria diferentes dentro del banco, se producirán conflictos, teniendo que ser serializados estos accesos. Para evitar estos conflictos lo que se hace es alinear los datos en memoria de tal forma que al acceder a ellos de forma paralela no se produzcan conflictos.

3.3. Algoritmo detector de bordes en GPU

En esta sección se describe la implementación en GPU del algoritmo detector de bordes presentado en el apartado 2.3. El pseudocódigo indicado en el algoritmo 2 muestra el algoritmo implementado, estando los kernels delimitados por los símbolos <>. El pseudocódigo incluye también los acrónimos GM y SM para indicar que los kernels trabajan con datos en memoria global (GM) o en memoria compartida (SM).

En primer lugar, el algoritmo necesita que el dataset se encuentre codificado en formato banda-vector y que esté centrado [0:255]. Antes de comenzar, es necesario que todos los datos con los que va a trabajar el algoritmo estén en la memoria global del dispositivo.

Al igual que la implementación en CPU presentada en la sección 2.3.2, algoritmo 1, esta versión también computa de forma paralela las bandas de la imagen, con la diferencia de que en la versión CPU se trabajaba con los píxeles de forma secuencial y en esta versión se explota también el paralelismo a nivel de píxeles. En el siguiente apartado se explicará exactamente los diferentes niveles de paralelismo que presenta este método. De forma implícita, los kernels trabajan paralelamente con píxeles y para poder explotar el paralelismo a nivel de bandas es necesario el uso de streams (técnica 3, sección 3.2). Esta opción consiste en utilizar tantos streams como bandas posee la imagen, donde cada stream se encargará de realizar toda la computación correspondiente a una banda; pudiéndose solapar, en teoría, toda la computación de todas las bandas. En la práctica, esto no ocurre así debido a que los recursos son limitados, no pudiéndose solapar todo el cómputo sino una pequeña parte del mismo.

Algoritmo 2 Detector de bordes basado en entropía en CUDA

Require: El dataset debe estar previamente centrado y almacenado en memoria global, a parte del dataset, en memoria global deben haber otros datos, como el número de píxeles que contiene una banda, además de estructuras de datos necesarias para el algoritmo.

1: **procedure** DETECTAR_BORDES
2: **for all** 1..n_bandas ***in parallel*** **do**
3: <getHistogram(): Obtener el histograma de la imagen> ▷ SM+GM
4: <compactHistogram(): Compactar el histograma (eliminar entradas a cero)> ▷ SM+GM
5: <getSum(): Obtener la suma de los valores de frecuencia del histograma> ▷ SM+GM
6: <normVec(): Normalizar el histograma> ▷ GM
7: <shannon(): Calcular umbral óptico T_1 y su posición, *Loc*, que divida el histograma en background y foreground> ▷ GM
8: <getSumVec(): Obtener la suma de los valores de frecuencia del histograma pertenenciente al foreground> ▷ SM+GM
9: <normVec(): Normalizar histograma del foreground> ▷ GM
10: <tsallis(): Calcular umbral óptico T_2 que divida el foreground en dos nuevas regiones> ▷ SM
11: <getSumVec(): Obtener la suma de los valores de frecuencia del histograma pertenenciente al background> ▷ SM+GM
12: <normVec(): Normalizar histograma del background> ▷ GM
13: <tsallis(): Calcular umbral óptico T_3 que divida el background en dos nuevas regiones> ▷ SM
14: <testT(): Generar imagen umbralizada a partir de la imagen original y de los valores T_1, T_2 y T_3 > ▷ GM
15: <edgeDetector(): Detectar bordes sobre la imagen creada por el kernel anterior> ▷ SM+GM
16: **end for**
17: **end procedure**

▷ GM: Global Memory, SM: Shared Memory

3.4. Niveles de paralelismo del algoritmo

Como se vio en el apartado anterior, el algoritmo detector de bordes presenta diferentes niveles de paralelismo que pueden ser explotados individualmente o de forma conjunta para conseguir un mayor nivel de eficiencia. Concretamente se puede diferenciar entre tres niveles de paralelismo:

El primer nivel de paralelismo se encuentra a nivel de píxeles, una imagen se divide en cientos o miles de píxeles que pueden ser procesados de forma paralela. Podría decirse que este es el nivel de paralelismo

mas fino, ya que el píxel es la unidad mínima de trabajo para el método en general. Paralelismo explotado de forma implícita por los kernels CUDA.

El segundo nivel de paralelismo se encuentra a nivel de kernels, el método detector de bordes consta de varios kernels, alguno de los cuales puede ejecutarse de forma paralela dado que no trabajan con los mismo datos a la vez. Este nivel de paralelismo no se explota en la implementación GPU ya que aumenta la bastante la complejidad en la programación si se quiere combinar con el siguiente nivel de paralelismo.

Por último, el tercer nivel de paralelismo se encuentra presente a nivel de imágenes bidimensionales. Como se mencionó en el apartado 1.2, una imagen hiperespectral consta de una serie de bandas que almacenan la información correspondiente a unas determinadas longitudes de onda. Cada una de estas bandas pueden considerarse imágenes, 2D, independientes y que por tanto pueden ser procesadas en paralelo. Paralelismo explotado gracias al uso de streams.

A continuación, en la figura 3.4, se identifica claramente el paralelismo presente en los datos y el cual se explota tal y como se explicó en los párrafos anteriores.

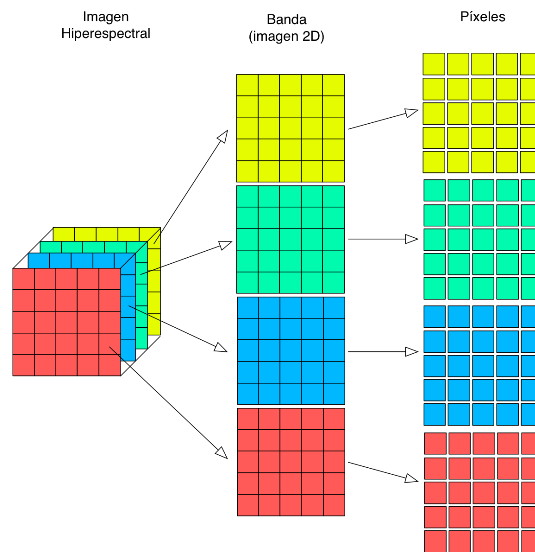


Figura 3.4: Niveles de descomposición de los datos.

3.5. Implementación a nivel de kernel

El método detector de bordes consta de una serie de kernels que se han ido paralelizando acorde a las características que iban presentando. En general, se ha intentado seguir los patrones básicos a la hora de desarrollar aplicaciones en CUDA, como puede ser la coalescencia en los accesos a memoria, evitar conflictos de banco en memoria compartida, evitar el uso de memoria local (spilling), etc. A continuación se presentan los diferentes kernels que conforman el método y se explica brevemente su utilidad y las características de su implementación.

getHistogram(): Como su nombre indica, este kernel se encarga de obtener el histograma de una imagen en escala de grises. Calcular el histograma consiste en contabilizar, para cada nivel de gris (0,...,255), el número de apariciones en toda la imagen. El resultado será un vector de 256 elementos con las frecuencias de aparición de cada valor de gris.

El cálculo del histograma es un problema ampliamente estudiado y que se caracteriza por existir un cuello de botella a la hora de realizar las sumas de las apariciones. Este cuello de botella se ve acentuado cuando las latencias de escrituras son altas, causando un deterioro en el rendimiento considerable. Para solventar este problema se han usado una serie de técnicas que se describen a continuación:

- Cálculo en memoria compartida de histogramas de manera individual para cada subimagen (ver técnica 1 en la sección 3.2). Con el objetivo de evitar, en la medida de lo posible, el cuello de botella, la

imagen se divide en varias subimágenes, para las cuales se calculará el histograma por separado. Estos histogramas locales se almacenarán en memoria compartida y se fusionarán cuando cada subimagen haya sido procesada. Al particionar la imagen, lo que se pretende es evitar colisiones en los accesos a las mismas posiciones de memoria cuando se quiere incrementar un valor.

- Uso de operaciones atómicas en memoria compartida (ver técnica 2 en la sección 3.2). Para las escrituras en memoria compartida se utilizan las operaciones atómicas, de esta manera aunque varios threads intenten incrementar la misma posición de memoria no se producirán errores. En arquitecturas anteriores no era posible realizar operaciones atómicas sobre memoria compartida por lo que había que emplear la memoria global, una memoria mucho más lenta.

compactHistogram(): Este kernel se encarga de compactar el histograma, su función es suprimir los valores con 0 apariciones y desplazar el resto de valores del histograma tantas posiciones como entradas se hayan eliminado.

A simple vista, esta función puede parecer difícil de paralelizar; se podrían lanzar tantos hilos como entradas tuviera el vector y comprobar si esas posiciones se encuentran a cero o no. Una vez localizadas esas posiciones habría que desplazar la parte del vector que queda a la derecha del la posición a cero. Si todos los hilos realizan el desplazamiento a la vez, el resultado sería imprevisible. Para garantizar el correcto funcionamiento habría que hacer los desplazamientos de forma secuencial. La clave para paralelizar esta etapa de reordenamiento está en buscar un enfoque alternativo. Al lanzar los hilos para hacer las comprobaciones, se puede crear un vector binario (paso 1 de la figura 3.5) donde todos sus valores sean 1, excepto en las posiciones donde se encuentre un valor a cero en el histograma. A partir de este vector binario se pueden calcular las posiciones finales de las entradas a 1 realizando una suma de prefijos [44] (paso 2 de la figura 3.5). Este tipo de sumas se pueden realizar perfectamente de forma paralela. Una vez realizada esta suma ya se sabe cual será la posición final de cada elemento del histograma en el vector resultado así como la dimensión de este nuevo vector.

Todo el proceso se realiza en memoria compartida, intentando evitar siempre los conflictos de bancos mediante el correcto alineamiento de los datos en memoria (ver técnica 8 en la sección 3.2).

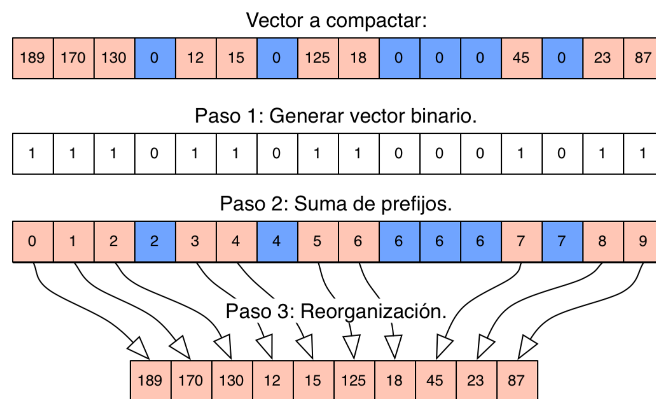


Figura 3.5: compactHistogram()

getSumVec(): La función de este kernel consiste el calcular el sumatorio de todos los elementos del vector de entrada.

Para poder realizar esta tarea de forma paralela se recurre a la suma por reducción [45]. Una reducción es una combinación de todos los elementos de un vector en un valor único, utilizando para ello algún tipo de operador asociativo. Las implementaciones paralelas aprovechan esta asociatividad para calcular operaciones en paralelo, calculando el resultado en O(logN) pasos sin incrementar el número de operaciones realizadas. La clave para explotar de forma adecuada la arquitectura es optimizar los accesos a memoria para evitar conflictos. También hay que tener en cuenta la longitud del vector a reducir, si el número de elementos es

mayor mayor que el número máximo de threads por bloque, se va a necesitar la colaboración entre bloques para realizar todas las reducciones, esta comunicación solo será posible mediante la memoria global, lo que penalizaría un poco la ejecución.

En este caso concreto, el tamaño máximo del vector nunca será superior a 256, por lo que todo el proceso de reducción se podrá realizar en un único bloque, pudiendo utilizar la memoria compartida para todas las operaciones. Con el objetivo de reducir aún más los tiempos, se pueden evitar las sincronización a nivel de warp, los hilos pertenecientes a un mismo warp se ejecutan de forma sincronizada no siendo necesario utilizar la rutina __syncthreads() para sincronizar estos hilos. Para aprovechar esta característica de los warps se han usado las instrucciones shuffle (técnica 7, sección 3.2). Este tipo de instrucciones permite a un hilo acceder a los registros de cualquier otro hilo dentro de su mismo warp, pudiendo, de esta forma, reducir subvectores de 32 elementos de forma muy rápida. En la siguiente figura (figura 3.6) se puede observar como la primera de las reducciones se realiza directamente al traer los datos desde memoria global, esto se hace así para evitar escrituras y lecturas innecesarias en memoria compartida y para reducir el tamaño de la memoria compartida necesaria para la reducción.

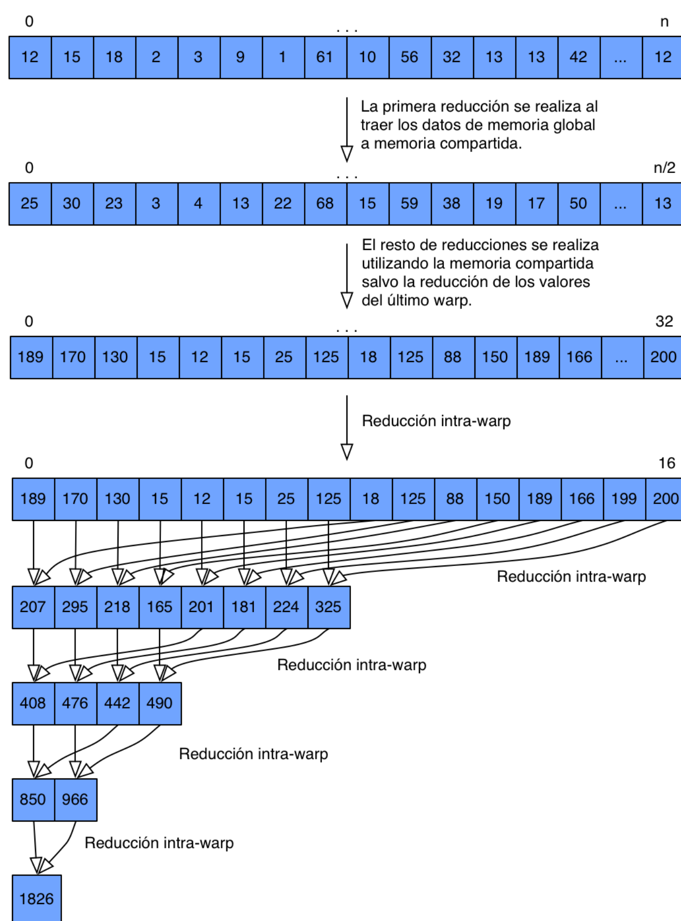


Figura 3.6: getSumVec()

normVec(): Este es un kernel muy simple que se utiliza para normalizar un vector. Cada hilo lee una entrada del vector a normalizar (almacenado en memoria global), divide su valor entre el sumatorio de los elementos del vector, previamente calculado con el kernel getSumVec(), y almacena el resultado en otro vector diferente ubicado en memoria global.

shannon(): Este es un kernel complejo que trabaja sobre el histograma de la imagen para calcular el umbral óptimo (T_1), que maximice la entropía de Shannon, y el lugar que ocupa este valor en el histograma (Loc) (método de Shannon 2.1.4).

Cada thread de este kernel tiene que calcular la entropía para un valor de gris concreto. Una vez que cada thread calcula su entropía, se hace una puesta en común, a través de la memoria compartida, para proceder a realizar una reducción y encontrar el valor, cuya entropía, sea la máxima. Este valor del histograma determinará el umbral (T_1) y su posición dentro del histograma (*Loc*) que determinará el punto de división de la imagen. La reducción para encontrar el máximo se realiza exactamente igual que en kernel getSumVec() pero en vez de usar el operador "+" se utiliza el operador "máximo".

La complejidad del kernel se encuentra en el cálculo de la entropía, esto se debe a que cada thread tiene que realizar un número diferente de operaciones provocando divergencia en las ejecuciones.

El problema, tal como se puede ver en el algoritmo 3 que describe los pasos necesarios para calcular la entropía de Shannon, se localiza en tres bucles "for", cuyos rangos de iteración varían de acuerdo al índice del thread que los ejecuta:

Algoritmo 3 Cálculo de la entropía de Shannon en CUDA

```
 1: procedure SHANNON_GPU(norm_histogram,n,T,Loc)
 2:     t=threadIdx.x;
 3:     if t<n then
 4:         s_norm_histogram[t]=norm_histogram[t];          ▷ Se carga el histograma en SM
 5:         for i in 0..t do
 6:             PA+=s_norm_histogram[i];                    ▷ Lectura desde la SM
 7:         end for
 8:         for i in 0..t do
 9:             pp1 = s_norm_histogram[i]/PA;               ▷ Lectura desde la SM
10:             Sa += pp1*log2f(pp1);
11:         end for
12:         PB=1-PA;
13:         for j in t+1..n do
14:             pp2 = s_norm_histogram[i]/PB;               ▷ Lectura desde la SM
15:             Sb += pp2*log2f(pp2);
16:         end for
17:         entropia_sm = -1 * (Sa + Sb);                   ▷ Se almacenan el valor de entropía en la SM
18:         __syncthreads();
19:     end if
20:     ...                                                 ▷ Reducción del vector entropia_sm para encontrar val. máx. SM
21: end procedure
                                                            ▷ GM: Global Memory, SM: Shared Memory
```

Como se puede ver, existe un comportamiento irregular que provocará divergencia en los threads de un mismo warp.

tsallis(): Este kernel trabaja con una porción del histograma para determinar el umbral óptico que maximice la entropía de TSallis (T_2, T_3), este valor permitirá segmentar la imagen de forma adecuada para posteriormente aplicar la detección de bordes. En lo referente a la complejidad, este kernel es exactamente igual de complejo que el kernel anterior. Su estructura es prácticamente igual salvo porque varía el tipo de instrucciones que se ejecutan. Al igual que en el caso anterior, el problema principal es el mismo, que el número de operaciones a realizar dependerá del thread.

Algoritmo 4 Cálculo de la entropía de TSallis en CUDA

```
 1: procedure TSALLIS_GPU(norm_histogram,n,T)
 2:     t=threadIdx.x;
 3:     if t<n then
 4:         s_norm_histogram[t]=norm_histogram[t];          ▷ Se carga el histograma en SM
 5:         for i in 0..t do
 6:             PA+=s_norm_histogram[i];                    ▷ Lectura desde la SM
 7:         end for
 8:         for i in 0..t do
 9:             Sa += sqrt(s_norm_histogram[i]/PA);         ▷ Lectura desde la SM
10:         end for
11:         PB=1-PA;
12:         for j in t+1..n do
13:             Sb += sqrt(s_norm_histogram[i]/PB);         ▷ Lectura desde la SM
14:         end for
15:         entropia_sm = (Sa * Sb)-1;                      ▷ Se almacenan el valor de entropía en la SM
16:         __syncthreads();
17:     end if
18:     ...                                                 ▷ Reducción del vector entropia_sm para encontrar val. máx. SM
19: end procedure
```

testT(): Este kernel se encarga de generar la fusión de las imágenes segmentadas. Para evitar crear tres imágenes diferentes con cada uno de los umbrales, hallados con los kernels anteriores, y luego fusionarlas, se crea una única imagen teniendo en cuenta estos tres valores (T_1,T_2 y T_3). Este kernel divide la imagen en bloques de 16*16 píxeles, donde por cada píxel existirá un thread que comprobará si dicho píxel cumple con la condición impuesta por los valores T_1, T_2 y T_3. Si cumple dicha condición, ese píxel se guardará como píxel válido, en caso contrario no se hace nada quedando a cero su posición en la imagen de salida.

edgeDetector(): El detector de bordes es un código stencil que calcula la probabilidad de que un píxel sea o no un borde dentro de la imagen. Este detector utiliza una ventana de 3x3 píxeles que se va desplazando por cada uno de los píxeles de la imagen, excepto por los bordes, comprobando si los píxeles de la ventana cumplen una determinada condición. En caso afirmativo, el píxel detectado como borde se guarda en una nueva imagen que será el mapa de bordes final.

Una característica que hace que este método sea adecuado para la paralelización mediante CUDA es que sigue un patrón regular. Este patrón se replicará en cada uno de los threads para así computar de forma totalmente paralela cada uno de los píxeles. El principal problema que presenta es el acceso a los datos, acceder a la información de 9 píxeles de forma eficiente no es una tarea trivial cuando hay cientos de hilos intentando acceder a la misma información a través de un mismo canal de datos. Los hilos contiguos acceden a datos contiguos y además estos datos son compartidos por ambos hilos. Que varios hilos accedan al mismo dato indica que el dato se está reutilizando, lo que hace propicio el uso de memoria compartida. Utilizando memoria compartida para almacenar los datos que se reutilizan se consigue disminuir las latencias de accesos ya que se evita tener que ir a buscar de nuevo un dato a memoria global, que previamente había sido accedido por otro thread. Como se puede ver en la imagen 3.7, donde cada número indica las veces que es utilizado un píxel, existe una alta reutilización de los datos en el cómputo de la detección de bordes.

Reutilización de datos

1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1
2	4	6	6	6	6	6	6	6	6	6	6	6	6	6	6	4	2
3	6	9	9	9	9	9	9	9	9	9	9	9	9	9	9	6	3
3	6	9	9	9	9	9	9	9	9	9	9	9	9	9	9	6	3
2	4	6	6	6	6	6	6	6	6	6	6	6	6	6	6	4	2
1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	2	1

Figura 3.7: Reutilización de datos: en la imagen, la mascara de detección de bordes se aplica a los píxeles enmarcados dentro del rectángulo de color negro.

Otro aspecto importante a tener en cuenta es la forma de acceder a los datos en memoria global. Las imágenes se encuentran almacenadas en forma de vector unidimensional, por filas, es decir, lo primero que se encuentra en memoria es la primera fila, a continuación, la segunda fila etc. Y los píxeles de las imágenes se encuentran representados por datos de tipo uint8_t (1 byte). Este kernel trabaja con bloques bidimensionales de 16x16 threads, por lo que cada fila de 16 threads va a poder traer desde memoria global 16 datos en una sola transferencia en el mejor de los casos. 16 datos equivalen a 16 bytes, lo que implica que no se esta aprovechando todo el ancho de banda disponible (256 bytes). Para poder leer hasta 256 bytes en una única lectura, se necesita que los datos se encuentren contiguos en memoria y alineados. En este caso, cuando las dimensiones de la imagen lo permiten, las lecturas se realizan como si los datos fueran de tipo double, uint o uint16_t, consiguiendo de esta forma aprovechar mejor el ancho de banda disponible. Al traer más de un dato a la vez, cada hilo puede computar varios píxeles sin necesidad de acudir a memoria global, lo que reduce los tiempos y evita las esperas por falta de datos.

Capítulo 4

Análisis y Resultados

En este capítulo se expondrán las pruebas realizadas y los resultados obtenidos así como un análisis de los mismos. Se comenzará con una descripción de los procedimientos seguidos para la realización de pruebas y toma de resultados. A continuación se mostrarán los resultados, los cuales se han dividido en los secciones bien diferenciadas, una donde se abordan los resultados de aplicar el algoritmo detector de bordes sobre diferentes tipos de imágenes y otra donde se presentan los resultados en términos de rendimiento.

4.1. Procedimiento y equipo utilizado

Las implementaciones propuestas han sido evaluadas sobre una GPU Nvidia de arquitectura Maxwell, la GTX 970, la cual cuenta con 1664 cores a 1,25MHz, agrupados en 13 SMs, con 128 cores por cada uno de ellos, alcanzando un rendimiento pico en simple precisión de 3494 GFLOPS. En cada instancia de tiempo, cada uno de estos SM es capaz de tener activos hasta 2048 threads simultáneamente (64 warps).

La GTX 970 está equipada sobre un interfaz PCI Express 3.0 x 16 que alcanza un pico de 32GB/s de ancho de banda en comunicación CPU-GPU. La GTX 970 tiene una memoria off-chip de 4GB, GDDR5, a 3505Mhz, con un ancho de banda de 224GB/s, y una memoria on-chip de 96KB por cada SM, dedicada exclusivamente para la memoria compartida. Además, cuenta con un sistema de memoria caché, con un nivel L1 por cada SM y un nivel unificado L2 de 1792KB.

Para contrastar resultados los mismos algoritmos fueron adaptados e implementados de forma secuencial y paralela para su ejecución sobre CPU. Para la paralelización de los códigos en CPU se utilizó la API OpenMP y para la optimización se aprovecharon los flags de optimización del compilador. En este trabajo dichas ejecuciones fueron realizadas con un equipo dotado de un procesador AMD Athlon(tm) 64 X2 5600+ a 2.8GHz, que cuenta con dos cachés L2 (una por cada core) de 1MB, con 64B de tamaño de línea y de asociatividad 16, y con cachés L1 de instrucciones y datos por cada core de 64KB, 64B de tamaño de línea y asociatividad 2. En lo referente a la memoria, el equipo cuenta con 4GB de memoria DDR2 a 800MHz. La interfaz de conexión con la tarjeta gráfica es un puerto PCI Express 2.0 x16.

Por otra parte, todas las medidas de tiempo mostradas en este trabajo fueron obtenidas como promedio de 20 ejecuciones independientes, con el sistema completo dedicado de forma exclusiva, y haciendo uso de la función gettimeofday() y de los eventos CUDA [54] para medir tiempos. En la medida de tiempos de ejecución de los diferentes kernels se ha tenido en cuenta la sincronización de los threads necesaria antes y después de cada llamada a la función de toma de tiempos. Además, las medidas del tiempo consumido por cada algoritmo realizadas se inician una vez la imagen hiperespectral se encuentra almacenada en la memoria global de la GPU, y terminan una vez los resultados de la detección son obtenidos, por lo tanto no se tienen en cuenta los tiempos de transferencia.

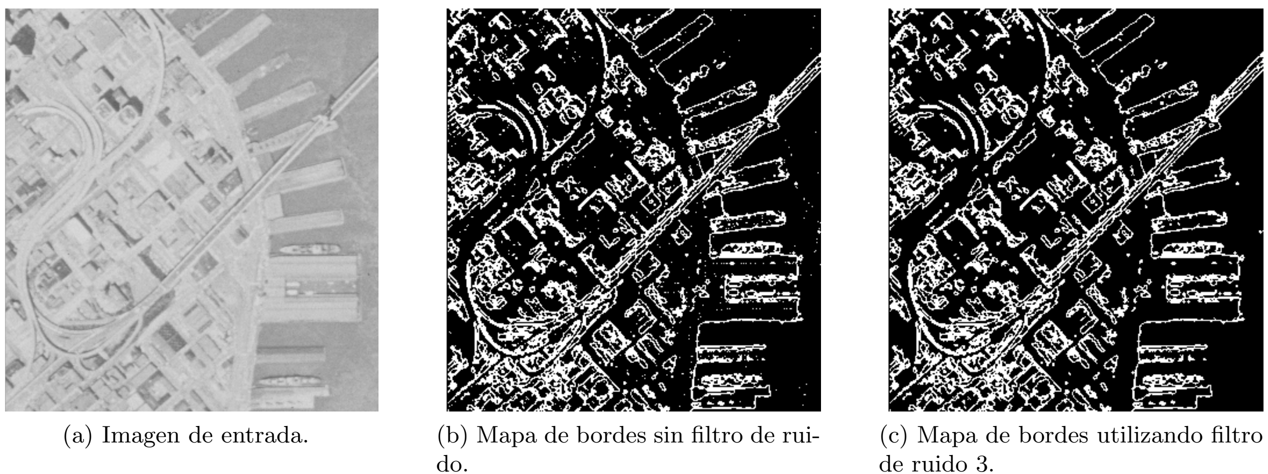
Respecto a la precisión es importante señalar que ambas implementaciones, GPU y CPU, proporcionan exactamente los mismos resultados ante los mismos parámetros de entrada.

Por último, hay que decir que se han realizado dos tipos de análisis, uno a nivel gráfico, donde se realizan comprobaciones de calidad basadas en la calidad visual de la detección de bordes en comparación con otros algoritmos. Y otro en cuanto a eficiencia computacional medida como tiempo de ejecución y como speedups.

4.2. Análisis gráfico de resultados de detección de bordes

En esta sección se mostrarán los resultados gráficos del método detector de bordes tanto para imágenes en escala de grises como para imágenes hiperespectrales. Para poder medir la calidad de los resultados se realizarán comparaciones con otros algoritmos de detección de bordes. Cabe destacar que la comparación de los resultados es algo subjetivo ya que no se puede cuantificar la calidad de los resultados más que a simple vista, tal como se realiza en papers de la bibliografía [37, 46]. Por último, también se analizará el comportamiento del algoritmo frente a posibles variaciones en la imagen de entrada.

Durante el proceso de obtención de resultados se observó que por regla general toda las imágenes de salida del algoritmo detector de bordes basado en umbralización por entropía contienen ruido en mayor o menor medida (figura 4.1). Con el objetivo de mejorar estos resultados se aplicó un filtro reductor de ruido [47], basado en el etiquetado de píxeles mediante la técnica union-find, capaz de eliminar agrupaciones de píxeles de un tamaño inferior a un valor indicado. En la figura 4.1 se puede ver el resultado de aplicar el detector de bordes a una imagen y su equivalente pero aplicando también el filtro reductor de ruido.



(a) Imagen de entrada. (b) Mapa de bordes sin filtro de ruido. (c) Mapa de bordes utilizando filtro de ruido 3.

Figura 4.1: Mejora de los resultados mediante el uso de un filtro de ruido.

A partir de este punto, todos los resultado del detector de bordes basado en umbralización por entropía que se muestren en el documento cuentan con la aplicación del filtrado de ruido. Se indicará entre paréntesis el límite a partir del cual se considera ruido una agrupación de píxeles. Este límite se elegirá mediante la realización de pruebas y comparación de resultados.

4.2.1. Imágenes en escala de grises, 2D

En este apartado se va a analizar el comportamiento del detector de bordes basado en umbralización por entropía sobre imágenes bidimensionales en escala de grises. Las imágenes empleadas han sido obtenidas de la base se datos de imágenes [55] de la Universidad del Sur de California. Algunas de ellas han sido recortadas para poder mostrar mejor los detalles de determinadas zonas.

Nombre de la imagen	Dimensiones espaciales
Lena	512*512
Cameraman	512*512
Airport	1024*1024
Stockton	1024*1024

Tabla 4.1: Propiedades de los dataset usados en los experimentos

Lena

(a) Img. original.

(b) Canny.

(c) Laplaciana de Gauss (LoG).

(d) Sobel.

(e) Prewitt.

(f) Roberts.

(g) Detector basado en umbralización por entropía (8).

Figura 4.2: Detección de bordes mediante diferentes algoritmos.

En la figura 4.2 se muestran la imagen original llamada Lena así como los resultados de detección de bordes ofrecidos por el algoritmo de detección de bordes basado en entropía y otros algoritmos clásicos descritos en el capítulo 2. Para la imagen Lena, el mejor resultado lo ofrece el algoritmo de Canny, en su mapa de bordes se pueden identificar fácilmente la gran mayoría de bordes y no aparecen falsos bordes. El segundo puesto no está tan claro. LoG ofrece buenos resultados pero faltan bordes, en cambio el detector de bordes basado en umbralización por entropía sí que identifica la gran mayoría de los bordes pero también introduce falsos bordes. Por último, estarían Prewitt, Sobel y Roberts, siendo este últimos el que peores resultados ofrece.

Cameraman

(a) Img. original. (b) Canny. (c) Laplaciana de Gauss (LoG).

(d) Sobel. (e) Prewitt. (f) Roberts.

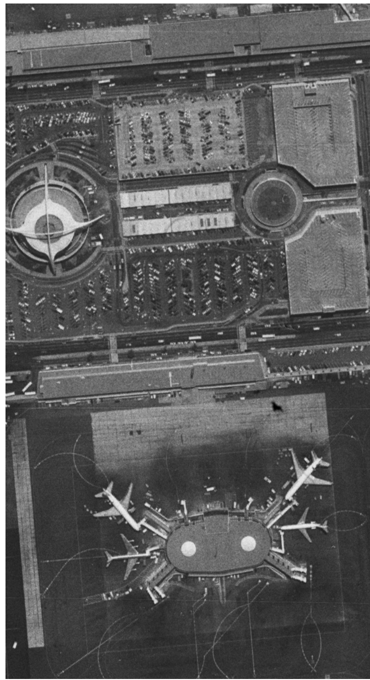
(g) Detector basado en umbralización por entropía (3).

Figura 4.3: Detección de bordes mediante diferentes algoritmos.

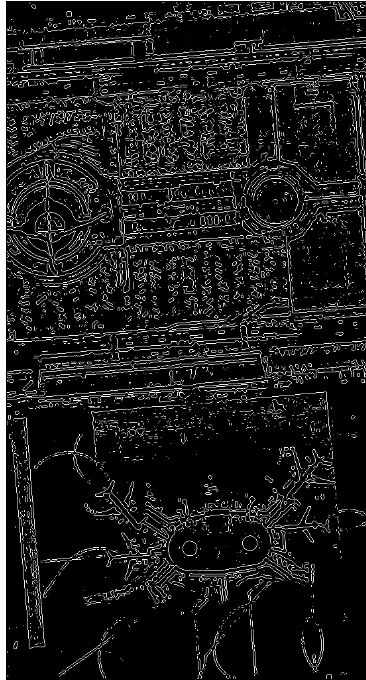
Al igual que sucedía con la imagen de Lena (figura 4.2), para la imagen llamada Cameraman (figura 4.3)los mejores resultados se obtienen empleando el detector de bordes Cany. De todos los métodos empleados es el único capaz de detectar bien las dos edificaciones altas del fondo de la imagen y otros detalles casi imperceptibles como los botones de la chaqueta del cámara. En segundo lugar estaría LoG, junto con Cany este método es capaz de detectar los bordes pertenecientes a los dedos del cámara, no detecta demasiado bien los bordes de los objetos del fondo pero en general ofrece unos resultados aceptables. En tercer lugar se encontraría el detector de bordes basado en umbralización por entropía, este es capaz de detectar mejor algunos objetos del fondo que no detectan otros detectores como Prewitt, Sobel o Roberts.

En líneas generales puede decirse que el método de Canny es el que mejores resultados ofrece, siendo el único capaz de detectar la mayor parte de los bordes presentes en las imágenes. Por el contrario, el método que peores resultados proporciona es el Roberts. Sobel y Prewitt ofrecen resultados muy similares. En lo referente al método LoG y al método detector de bordes basado en umbralización por entropía, no podría decirse cual de los dos es mejor. A continuación se realizará una prueba más para intentar encontrar alguna evidencia que nos haga decantarnos por uno o por otro.

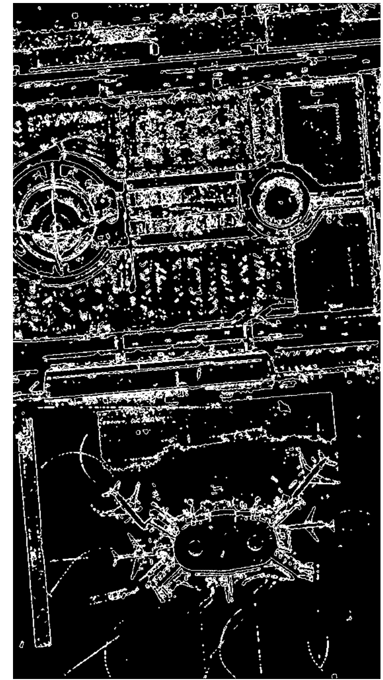
Airport



(a) Imagen original



(b) Laplaciana de Gauss (LoG).



(c) Detector basado en umbralización por entropía (3).

Figura 4.4: Detector de bordes LoG vs detector de bordes basado en umbralización por entropía .

Como se puede observar en la parte superior izquierda, existe una especie de cúpula con unos bordes bien definidos, los cuales quedan mejor reflejados por el método LoG. Además también se puede ver como este método ofrece unas líneas más finas y unos contornos mejor definidos. En la parte central inferior de la imagen se pueden ver dos círculos de color blanco cuyos bordes se encuentran bien definidos, sin embargo el método detector de bordes basado en umbralización por entropía no es capaz de detectarlos del todo bien. Como conclusión final podría decirse que el método LoG es ligeramente superior al método detector de bordes basado en umbralización por entropía.

Tolerancia al ruido

Para probar el comportamiento del método detector de bordes basado en entropía ante la presencia de ruido en las imágenes, se ha utilizado la imagen Lena añadiéndole diferentes niveles de ruido. En este caso no se comparará el método detector de bordes basado en entropía con otros métodos, sino que se generarán varias imágenes con ruido y se irá analizando el comportamiento del mismo ante estas imágenes. Para añadir ruido a las imágenes se ha utilizado la rutina "imnoise"de matlab con el tipo de ruido: blanco gaussiano. Para variar el nivel de ruido se ha modificado únicamente la varianza (v), dejando constante el valor de la media ($m = 0$).

En la siguiente figura (figura 4.5) se muestra la imagen de Lena original así como modificaciones de la misma donde se le ha añadido ruido. Para cada una de estas imágenes se muestra el correspondiente mapa de bordes producido por el detector de bordes basado en entropía. Además también se muestran los mapas de bordes a los cuales se les ha aplicado el filtro de ruido.

Lena con ruido

(a) Img. original.

(b) Mapa de bordes generado por el algoritmo detector basado en entropía sin filtro de ruido.

(c) Mapa de bordes generado por el algoritmo detector basado en entropía con filtro de ruido 8

(d) Imagen con ruido $v = 0{,}001$

(e) Mapa de bordes generado por el algoritmo detector basado en entropía sin filtro de ruido.

(f) Mapa de bordes generado por el algoritmo detector basado en entropía con filtro de ruido 10.

(g) Imagen con ruido $v = 0{,}005$

(h) Mapa de bordes generado por el algoritmo detector basado en entropía sin filtro de ruido.

(i) Mapa de bordes generado por el algoritmo detector basado en entropía con filtro de ruido 10.

(j) Imagen con ruido $v = 0{,}010$

(k) Mapa de bordes generado por el algoritmo detector basado en entropía sin filtro de ruido.

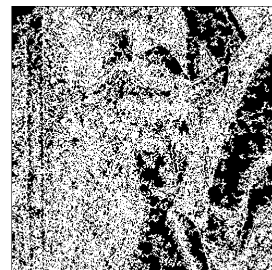

(l) Mapa de bordes generado por el algoritmo detector basado en entropía con filtro de ruido 50.

Figura 4.5: Tolerancia al ruido.

Como se puede observar, a medida que aumenta el nivel de ruido en la imagen los resultados empeoran de forma considerable. Aunque para el ojo humano sigue siendo fácil distinguir los bordes en la imagen con

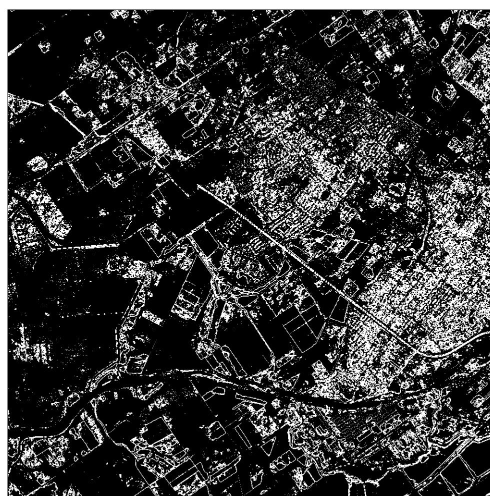
más ruido, para el método detector de bordes se vuelve una tarea prácticamente imposible. En el capítulo 2 se mencionó en varias ocasiones la sensibilidad de los detectores de bordes, basados en la primera y segunda derivada, al ruido, hecho que queda corroborado también para este método.

Influencia del contraste en la detección de bordes

En este apartado se estudiará la influencia del contraste en las imágenes sobre la detección de bordes mediante el método detector de bordes basado en umbralización por entropía.

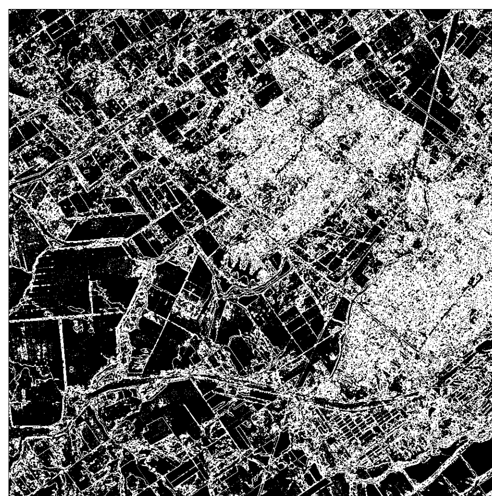
Stockton

(a) Imgagen original.



(b) Mapa de bordes generado por el algoritmo detector basado en entropía.

(c) Imagen con realce de contraste.



(d) Mapa de bordes generado por el algoritmo detector basado en entropía.

Figura 4.6: Influencia del contraste de las imágenes en la detección de bordes.

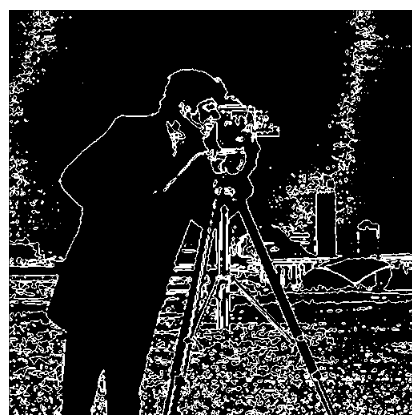
Como se puede ver, realzar el contraste en la imagen de entrada puede favorecer la detección de bordes, sin embargo, también hace que se pierdan algunos bordes o que aparezcan falsos bordes. En esta imagen podemos ver cómo la carretera en posición diagonal, que va desde la parte inferior derecha hacia la parte superior izquierda, desaparece al aplicar el realce del contraste.

Cameraman

(a) Imgagen original.

(b) Mapa de bordes generado por el algoritmo detector basado en entropía (0)

(c) Imagen con realce de contraste.



(d) Mapa de bordes generado por el algoritmo detector basado en entropía (0).

Figura 4.7: Influencia del contraste de las imágenes en la detección de bordes.

Al igual que en el caso anterior, en este caso se detectan más bordes en la imagen con mayor contraste que en la imagen original, pudiéndose detectar mucho mejor las edificaciones que se encuentran al fondo de la imagen, sin embargo también aparecen falsos bordes y ruido en la zona del cielo.

En general, se puede decir que el contraste es un aspecto muy importante a la hora de detectar bordes. Una imagen con un contraste bajo dificultará la tarea de detección de bordes, en cambio una imagen con demasiado contraste puede provocar la aparición de falsos bordes.

4.2.2. Detección de bordes en imágenes hiperespectrales, comparativa de métodos de fusión

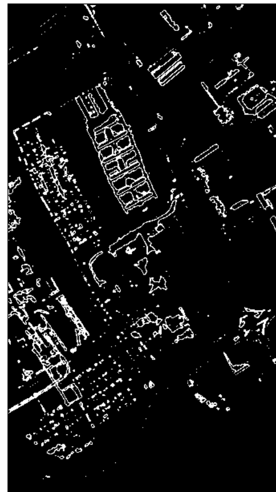
Como se vio en el apartado 2.3.2, se utilizaron diferentes técnicas para la fusión de la información de salida del algoritmo. En este apartado se estudiará cuales de estas técnicas ofrece mejores resultados. Para la realización de las pruebas se escogieron los parámetros que mejores resultados ofrecían de acuerdo con la imagen. En lo referente a los conjuntos de datos, para estos experimentos se utilizarán dos imágenes hiperespectrales. La primera de ellas es una imagen de la universidad de Pavia [48] y la segunda es una imagen de unas plantaciones en Salinas Valley, California [49]. Las características de estos conjuntos de datos son las siguientes:

Nombre del conjunto de datos	Sensor	Nº de bandas espectrales	Dimensiones espaciales
PaviaU	ROSIS	103	340*610
Salinas corrected	AVIRIS	224	217*512

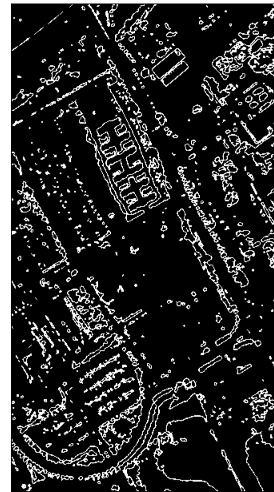
Tabla 4.2: Propiedades de los dataset usados en los experimentos

PaviaU

(a) Composición de las bandas RGB.



(b) F. de bandas a través del cálculo de medias (0).

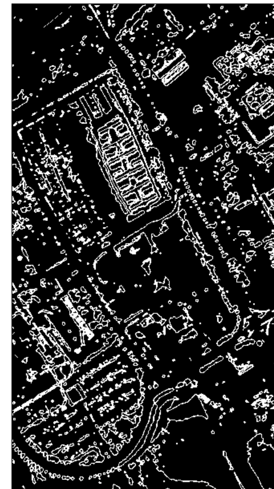


(c) PCA + f. de las 3 primeras bandas a través del cálculo de medias (3).

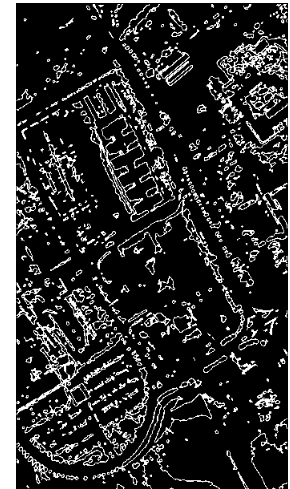
(d) Fusión de mapas de bordes, umbral 3, (5).

(e) F. de imágenes segmentadas de cada banda, umbral 14, (5).

(f) F. de entropías a través del cálculo de medias + f. de bandas a través del cálculo de medias, (0).



(g) PCA + f. de los 3 primeros mapas de bordes, umbral 35, (5).



(h) PCA + f. de las 3 primeras imágenes segmentadas, umbral 66, (5).

Figura 4.8: Comparativa de métodos de fusión sobre la imagen PaviaU.

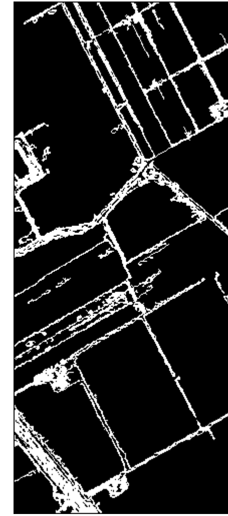
A simple vista se puede observar que las dos técnicas de fusión que mejores resultados ofrecen son la técnica de fusión de mapas de bordes (figura 4.8(d)) y la técnica de fusión de imágenes segmentadas (figura 4.8(e)). Estas dos técnicas son capaces de captar una gran variedad de bordes, siendo la técnica de fusión de mapas de bordes la que parece captar un mayor número. Si nos fijamos en la parte superior derecha de la composición RGB podemos ver una construcción en forma de semicírculo que únicamente aparece reflejado cuando se hace uso de la técnica de fusión de mapas de bordes.

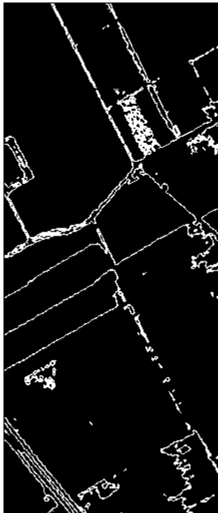
Salinas corrected

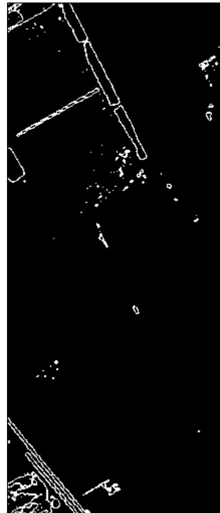
(a) Composición RGB. [50]

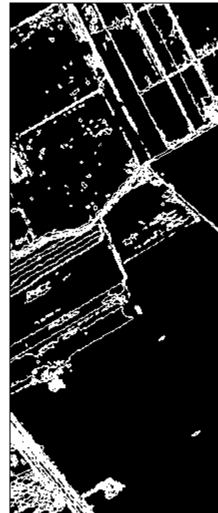
(b) F. de bandas a través del cálculo de medias (0).

(c) PCA + f. de las 8 primeras bandas a través del cálculo de medias, (3).


(d) Fusión de mapas de bordes, umbral 11, (30).


(e) F. de imágenes segmentadas de cada banda, umbral 70, (3).


(f) F. de entropías a través del cálculo de medias + f. de bandas a través del cálculo de medias, (0).


(g) PCA + f. de los 10 primeros mapas de bordes, umbral 10, (5).

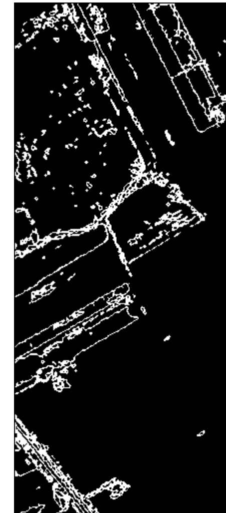

(h) PCA + f. de las 10 primeras imágenes segmentadas,umbral 10, (3).

Figura 4.9: Comparativa de métodos de fusión sobre la imagen Salinas.

Para esta otra imagen los mejores resultados los vuelve a proporcionar la técnica de fusión de mapas de bordes (imagen d). Esta técnica produce un mapa de bordes donde se pueden detectar de una forma relativamente clara la mayor parte de las divisiones entre los campos de cultivo. La técnica de fusión de imágenes segmentadas (figura 4.9(e)) junto con la técnica de PCA + fusión de bandas a través del cálculo de medias (figura 4.9(c)) también ofrecen unos resultados aceptables aunque no son capaces de producir un mapa con tantos bordes como la técnica anterior.

Como se ha visto en esta pequeña comparativa las dos técnicas que mejores resultados proporcionan son la técnica de fusión de imágenes segmentadas y la técnica de fusión de mapas de bordes. Para las dos imágenes anteriores parece ser mejor la técnica de fusión de mapas pero los resultados podrían variar con imágenes diferentes. Las técnicas que emplean el análisis de componentes principales podría decirse que

ocuparían el segundo lugar en cuanto a calidad de resultados. Por último estaría la técnica de fusión de entropías a través del cálculo de medias + fusión de bandas a través del cálculo de medias y la técnica de fusión de bandas a través del cálculo de medias, las cuales ofrecen malos resultados.

4.2.3. Detección de bordes en imágenes hiperespectrales, comparativa con otros métodos

Tras la comparativa sobre imágenes 2D en niveles de grises que se realizó en la sección 4.2.1, en esta sección se realizará una comparativa con algoritmos en la bibliografía que han sido diseñados para su aplicación a imágenes hiperespectrales. Concretamente se comparará con tres métodos: el método de Huntsberger y Descalzi [33], el método de Di-Zenzo [34] y el método propuesto por Dinh, Leitner, Paclik y Duin [37]. Los dos primero métodos se han seleccionado porque son habituales en la bibliografía de detección de bordes sobre imágenes hiperespectrales y el último método, el propuesto de por Dinh y otros, se ha seleccionado porque se presenta como una alternativa a los anteriores.

Los datos de mapas de bordes así como las representaciones de las imágenes hiperespectrales que se utilizan en esta comparativa han sido tomadas del artículo "A Clustering Based Method for Edge Detection in Hyperspectral Images" [37].

Las pruebas se realizarán sobre diferentes porciones de dos conjuntos de datos. El primer conjunto de datos es una imagen hiperespectral de Washington DC Mall [51] y el segundo conjunto es "Flightline (FLC1)-[52] capturado en la parte sur de Tippecanoe County en Indiana. En el artículo de donde se extrajo la información consideraron oportuno dividir las imágenes en porciones más pequeñas para hacerlas manejables y así poder mostrarlas mejor en el documento.

Nombre del conjunto de datos	Sensor	Nº de bandas	Dimensiones espaciales
DC Mall	HYDICE	191	307*1280
FLC1	-	12	220*949

Tabla 4.3: Propiedades de los dataset usados en los experimentos

DC Mall 1

(a) Representación RGB usando PCA.

(b) Método de Di-Zenzo.

(c) Método de Huntsberge y Descalzi.

(d) Método de Dinh, Leitner y otros.

(e) Fusión de mapas de bordes, umbral 15, (3).

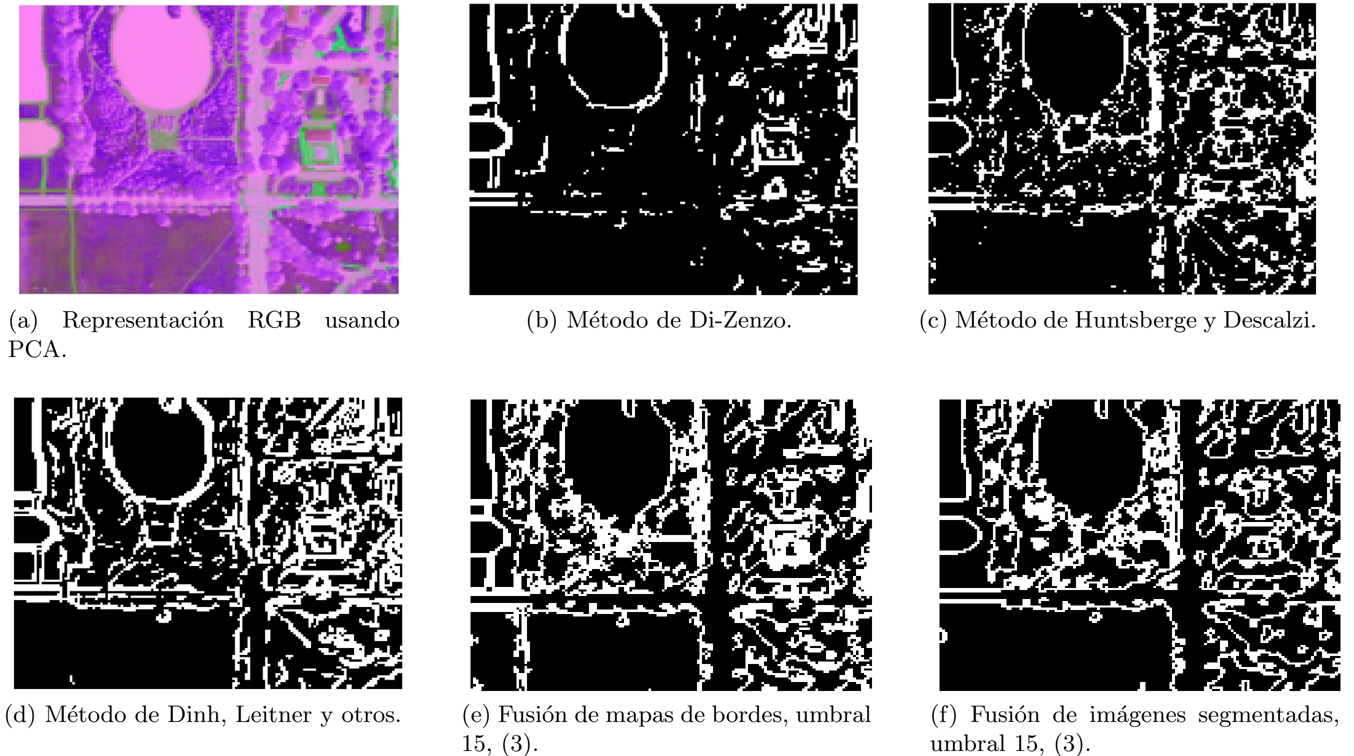
(f) Fusión de imágenes segmentadas, umbral 15, (3).

Figura 4.10: Resultados de detección de bordes sobre la imagen DC Mall, porción 1.

Para proporcionar una representación intuitiva de las imágenes se usó PCA, utilizando las tres primeras bandas, pertenecientes a las tres primeras componentes principales, se creó una composición RGB. La representación en color de cada una de las imágenes son la figura 4.10(a), la figura 4.11(a) y la figura 4.12(a).

En lo referente a la configuración del detector de bordes basado en umbralización por entropía, se ha usado junto con las técnicas de fusión que mejores mapas de bordes proporcionaban (apartado 4.2.2), estas eran la fusión de mapas de bordes y la fusión de imágenes segmentadas. La configuración de las técnicas de fusión se indican debajo de cada mapa de bordes obtenido con estas técnicas.

DC Mall 2



(a) Representación RGB usando PCA. (b) Método de Di-Zenzo. (c) Método de Huntsberge y Descalzi.

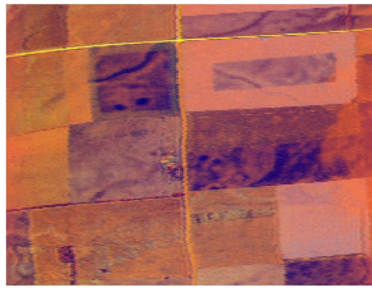
(d) Método de Dinh, Leitner y otros. (e) Fusión de mapas de bordes, umbral 15, (3). (f) Fusión de imágenes segmentadas, umbral 15, (3).

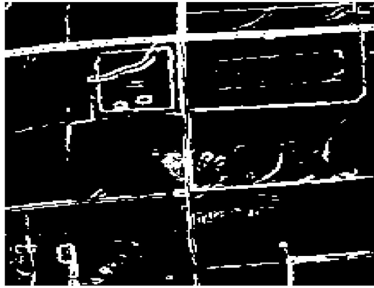
Figura 4.11: Resultados de detección de bordes sobre la imagen DC Mall, porción 2.

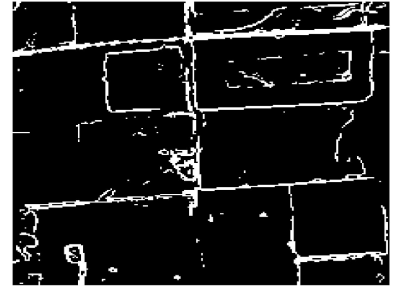
Para la primera de las porciones de la imagen DC Mall, figura 4.10, el método detector de bordes basado en umbralización por entropía con la fusión de mapas de bordes presenta unos resultados bastante buenos, siendo este método superior al resto salvo al método de Di-Zenzo, que ofrece también un buen resultado en comparación con los demás. En esta situación sería difícil decantarse por uno u otro método.

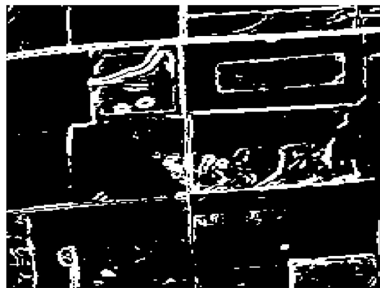
Atendiendo ahora a la segunda de las porciones de la imagen DC Mall (figura 4.11), se puede ver como los mejores resultados los ofrece el método propuesto por Dinh, Leitner, Paclik y Duin [37]. A continuación podría decirse que el método detector de bordes basado en umbralización por entropía con la fusión de mapas de bordes presenta unos resultados aceptables, detectando un camino que no es capaz de detectar el resto de detectores, por contra, la detección de bordes en la parte donde aparece una especie de círculo en la imagen no es del todo buena. El detector de bordes basado en umbralización por entropía con la fusión de imágenes segmentadas no presenta tan buenos resultados, al igual que ocurre con el método de Di-Zenzo y el método de Huntsberge y Descalzi.

FLC1


(a) Representación usando PCA.


(b) Método de Di-Zenzo.


(c) Método de Huntsberge y Descalzi.


(d) Método de Dinh, Leitner y otros.

(e) Fusión de mapas de bordes, umbral 20, (3).

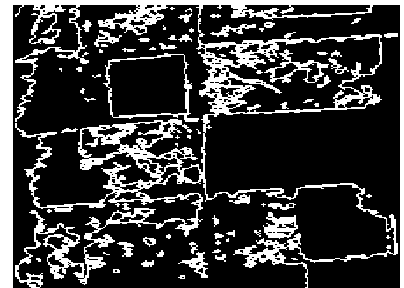

(f) Fusión de imágenes segmentadas, umbral 15, (3).

Figura 4.12: Resultados de detección de bordes sobre la imagen FLC1.

Por último se va a analizar una porción de la imagen FLC1. Para estos datos de entrada, el mejor resultado lo ofrece el método propuesto por Dinh, Leitner, Paclik y Duin [37]. Este método es capaz de detectar más bordes y de forma más clara que el resto de métodos. El siguiente mejor mapa de bordes lo ofrece el método de Di-Zenzo, seguido por el mapa de bordes del método de Huntsberge y Descalzi. Para esta imagen, los peores resultados los ofrece el método detector de bordes basado en umbralización por entropía, no pudiéndose distinguir cual de sus dos variantes es mejor.

Tras el análisis de los resultados podría decirse que el método detector de bordes basado en umbralización por entropía ofrece mejores resultados cuanta más información espectral tiene a su disposición. Para la imagen DC Mall, imagen con 191 bandas, este método ofreció unos resultados buenos a nivel global. En cambio para la imagen FLC1, imagen con tan solo 12 bandas, los resultados fueron realmente malos.

4.3. Resultados en cuanto a rendimiento

Las ejecuciones de prueba se realizaron con dos versiones diferentes del algoritmo, una versión del algoritmo de detección de bordes basado en entropía tal como fue definido en el algoritmo 2, sección 3.3 y una versión en que además se realiza una reducción de ruido mediante el algoritmo comentado en la sección 4.2. Además, para estudiar más en detalle el rendimiento del algoritmo se han realizado medidas de tiempo de ejecución no solamente para la imagen completa (versión "multibanda"del algoritmo) sino también cuando se aplica a una única banda (versión "monobanda"del algoritmo). Esta versión "monobanda"también nos permitirá observar el speedup a nivel de kernels.

En lo referente a la configuración de cada kernel, los que trabajan con el histograma han sido diseñados para trabajar siempre con tamaño de bloque igual a 256 ó 128. En cambio los que trabajan con la imagen si permiten modificar este parámetro. Para las ejecuciones se ha utilizado el tamaño de bloque considerado como óptimo (tabla 4.4) en base a medidas de rendimiento realizadas.

Kernel	Tamaño de bloque
getHistogram()	512
compactHistogram()	128
getSum()	128
normVec()	32
shannon()	256
tsallis()	256
testT()	16x16
edgeDetector()	16x16

Tabla 4.4: Tamaños de bloques óptimos.

Como imagen de entrada para la toma de resultados se ha empleado la imagen hiperespectral Pavia centre [53]. Esta imagen cuenta con 102 bandas espectrales y sus dimensiones espaciales son 715*1096 píxeles.

4.3.1. Versión 1: sin reducción de ruido

En la tabla 4.5 se muestran los tiempos de ejecución del algoritmo cuando se aplican a una sola banda de la imagen. En este caso de aplica a la primera banda de la imagen Pavia centre, cuyas dimensiones espaciales son 715x1096 píxeles. La segunda columna muestra los tiempos en CPU para la versión secuencial del algoritmo. La tercera columna los tiempos de la versión OpenMP, empleando los dos threads de los que dispone la CPU. En la cuarta columna los tiempos de ejecución en GPU donde solo se explota el paralelismo a nivel de píxeles y finalmente en la última columna los tiempos de ejecución en GPU para la versión que aprovecha tanto el paralelismo a nivel de píxeles como el paralelismo a nivel de kernels (ver sección 3.4).

Versión monobanda				
	CPU-Sec. (s)	**CPU-OMP.(s)**	**GPU (s)**	**GPU-Par. a nivel de kernels (s)**
Tiempo de cómputo	0,011332	0,007326	0,000445	0,000268
getHistogram()	0,001433	0,001422	0,000049	
compactHistogram()	0,000002	0,000002	0,000011	
getSum()	0,000001	0,000003	0,000030	
normVec()	0,000004	0,000004	0,000047	
shannon()	0,001450	0,000853	0,000100	
tsallis()	0,000367	0,000188	0,000112	
testT()	0,001921	0,001219	0,000041	
edgeDetector()	0,005613	0,003368	0,000057	

Tabla 4.5: Tiempos de ejecución del método detector de bordes, sobre la banda 0 de la imagen Pavia centre, para las distintas versiones CPU y GPU.

Como se puede apreciar en la tabla 4.5 el mejor tiempo se obtiene usando la versión GPU que explota el paralelismo a nivel de píxeles junto al paralelismo a nivel de kernels. Esta versión obtiene un speedup de 42,28× frente a la versión secuencial en CPU y 27,33× frente a la versión paralela en CPU. Comparando las dos versiones GPU se puede ver que explotar el paralelismo a nivel de kernels supone que se consiga un speedup de 1,66× frente a la versión que solo explota el paralelismo a nivel de píxeles.

En lo referente a los kernels, hay varios, como getSum(), que ofrecen peores resultados después de haber sido paralelizados para GPU. Lo lógico sería utilizar las versiones CPU pero el problema que se presenta, al hacer esto, es la necesidad de realizar transferencias de datos entre CPU y GPU, lo cual penaliza mucho más la ejecución que el tiempo empleado por estos kernels en GPU. El kernel que mejor speedup consigue frente a las versiones CPU es el kernel encargado de la detección de bordes, edgeDetector(), que obtiene un speedup de 98,47× frente a la versión secuencial en CPU. Si nos fijamos bien en la tabla de tiempos, vemos como los kernels que mejores ganancias consiguen son aquellos kernels trabajan directamente con la imagen y no con su histograma. Estos kernels son getHistogram(), testT() y edgeDetector(), los cuales explotan muy bien el paralelismo a nivel de píxeles. Al contrario que la CPU, que solo puede procesar los

píxeles de forma secuencial, o en el mejor de los casos de dos en dos, gracias a la implementación paralela con OpenMP, la arquitectura paralela de la GPU permite procesar cientos de píxeles a la vez, haciendo que los tiempos mejoren sustancialmente.

Ahora pasaremos a ver los resultados obtenidos en la versión "multibanda"donde el método va a trabajar con todas las bandas de la imagen hiperespectral. En la tabla 4.6, la segunda columna muestra los tiempos en CPU para la versión secuencial del algoritmo. La tercera columna los tiempos de la versión OpenMP, empleando los dos threads de los que dispone la CPU. En la cuarta columna los tiempos de ejecución en GPU donde solo se explota el paralelismo a nivel de píxeles y finalmente, en la última columna, los tiempos de ejecución en GPU para la versión que aprovecha tanto el paralelismo a nivel de píxeles como el paralelismo de imágenes (ver sección 3.4).

Versión multibanda				
	CPU-Sec. (s)	**CPU-OMP.(s)**	**GPU (s)**	**GPU-Par. a nivel de imág. (s)**
Tiempo de cómputo	1,210668	0,619099	0,021446	0,011368

Tabla 4.6: Tiempos de ejecución del método detector de bordes, sobre la imagen hiperespectral Pavia centre, para las distintas versiones CPU y GPU.

Para esta versión del algoritmo donde se trabaja con todas las bandas de la imagen hiperespectral, la paralelización cambia tanto para el código CPU como para el código en GPU. Mientras que para la versión monobanda la paralelización en CPU se realizaba a nivel de píxeles, ahora la paralelización pasa a realizarse a nivel de imágenes (bandas). Algo parecido ocurre con el código en GPU, se sigue explotando el paralelismo a nivel de píxeles pero en vez de explotar el paralelismo a nivel de kernels ahora se explota el paralelismo a nivel de imágenes (bandas). Esta nueva forma de explotar el paralelismo nos permitirá aprovechar mejor el hardware y reducir los tiempos de cómputo.

Atendiendo a los resultados de la tabla 4.6,si se compara la versión paralela en CPU con la versión GPU en la que se procesa cada banda de forma secuencial podemos ver que se obtiene un speedup de 28,86× a favor de la versión GPU; en cambio si se compara con la versión GPU que aprovecha el paralelismo a nivel de imágenes (bandas) vemos como el speedup aumenta hasta 54,45×. Para calcular exactamente la mejora que proporciona una versión frente a otra se comparan los tiempos de ambas versiones en GPU y se obtiene un speedup de 1,88× a favor de la versión que aprovecha el paralelismo a nivel de imágenes. En teoría el speedup debería ser mucho mayor ya que tal y como está programado el algoritmo debería ser posible solapar la computación de todas las bandas. El problema está en que los recursos disponibles no son infinitos por lo que va a ser imposible solapar toda la computación.

4.3.2. Versión 2: con reducción de ruido

En este apartado se mostrarán las medidas de tiempo así como un análisis de las mismas para la versión del método detector de bordes que ha sido mejorada añadiéndole un filtro de ruido. Este filtro de ruido viene representado en la tabla 4.7 con el nombre RemoveNoise. RemoveNoise no es un único kernel, es un grupo de kernels que se encarga de todo el proceso de detección y eliminación de aquellas agrupaciones de píxeles consideradas ruido. Se ha presentado así ya que se trata de un código externo en el que no se ha trabajado y lo único que se ha hecho es configurar los parámetros y comprobar que los tamaños de bloques eran los adecuados para maximizar la ocupación.

Versión monobanda				
	CPU-Sec. (s)	**CPU-OMP(s)**	**GPU (s)**	**GPU-Par. a nivel de kernels (s)**
Tiempo de cómputo	0,025875	0,020453	0,001061	0,000996
getHistogram()	0,001192	0,001194	0,000050	
compactHistogram()	0,000002	0,000002	0,000010	
getSum()	0,000002	0,000002	0,000029	
normVec()	0,000004	0,000005	0,000046	
shannon()	0,001447	0,000852	0,000100	
tsallis()	0,000368	0,000187	0,000112	
testT()	0,001878	0,001248	0,000042	
edgeDetector()	0,005446	0,003394	0,000055	
*RemoveNoise	0,013576	0,012676	0,000615	

Tabla 4.7: Tiempos de ejecución del método detector de bordes con reducción de ruido, sobre la banda 0 de la imagen Pavia centre, para las distintas versiones CPU y GPU.

Como se puede ver en la tabla 4.7, la eliminación de ruido es la tarea más costosa en todas las implementaciones, empleando más tiempo que el resto de kernels juntos. Esto implica que las mejoras en el cómputo global van a depender mucho del rendimiento de esta tarea al ser paralelizada. Si calculamos el speedup de la versión GPU frente a la versión secuencial en CPU, obtenemos un resultado de 22×, por lo que el speedup global va a estar también en torno a este valor. Concretamente, si se comparan los tiempos globales de la mejor versión GPU con la versión secuencial en CPU, se obtiene un speedup de 25× y si se compara con la versión OpenMP el speedup disminuye ligeramente hasta 20×.

Por último, si se comparan los tiempo de ambas implementaciones GPU, el speedup conseguido por la versión que explota el paralelismo a nivel de kernels es mínimo, de tan solo 1,065×, esto se debe a que la ganancia conseguida con el solapamiento de la computación de los kernels representa una ínfima parte del tiempo global, marcado este, como ya vimos anteriormente, por el tiempo consumido por la tarea de detección y eliminación de ruido.

A continuación se muestra la tabla de tiempos obtenidos para la ejecución multibanda con reducción de ruido.

Versión multibanda				
	CPU-Sec. (s)	**CPU-OMP.(s)**	**GPU (s)**	**GPU-Par. a nivel de imág. (s)**
Tiempo de cómputo	3,323860	1,911469	0,120929	0,084990

Tabla 4.8: Tiempos de ejecución del método detector de bordes con reducción de ruido sobre la imagen hiperespectral Pavia centre, para las distintas versiones CPU y GPU.

Para la versión multibanda (tabla 4.8), el tiempo de la implementación GPU que aprovecha el paralelismo a nivel de imágenes (bandas) mejora considerablemente, obteniendo un speedup de 39× frente a la versión secuencial en CPU y de 22× frente a la versión OpenMP. Hay que destacar que la versión OpenMP también mejoró bastante los resultados en comparación con la versión monobanda, de ahí que el speedup con la versión GPU no sea mayor.

Al igual que ocurría en la versión multibanda sin reducción de ruido, al haber una mayor cantidad de kernels que pueden solapar su computación se produce una mejora en los tiempos. Para cuantificar esta mejora comparamos los tiempos para las dos versiones GPU obteniendo un speedup de 1.42×, si comparamos este resultado con el mismo resultado pero para la versión sin reducción de ruido vemos cómo en la versión sin reducción de ruido los resultados son mejores, 1,88×. Esto se debe a que los kernels encargados de la detección y eliminación del ruido utilizan un elevado número de hilos y acaparan la gran mayoría de recursos, teniendo el resto de kernels que esperar a que estén disponibles para poder ejecutarse, en la imagen 4.13 se puede ver claramente lo que ocurre.

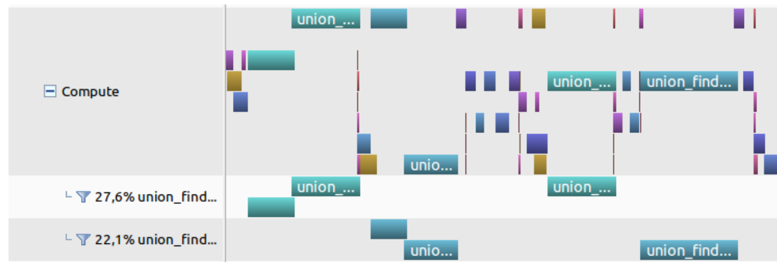


Figura 4.13: Imagen del profiler de Nvidia donde se refleja el acaparamiento de los recursos por parte de algunos kernels.

En esta imagen (figura 4.13) se puede ver como los kernels union_find_shared_gpu() y union_find_global_gpu(), que son kernels pertenecientes al filtro de ruido, se ejecutan de forma casi independiente debido a que no hay más recursos disponibles que permitan a otro kernel ejecutarse de forma concurrente junto a ellos. Para la imagen Pavia centre una ejecución de estos kernels puede lanzar más de 700000 threads. Para entender mejor por qué consume tantos recursos es necesario conocer su funcionamiento: el filtro de ruido se basa en el etiquetado de píxeles mediante la técnica union-find; se realizan varios barridos de la imagen para detectar agrupaciones de píxeles y etiquetar dichos grupos según la cantidad de píxeles que contengan. Una vez identificados estos grupos, se eliminan aquellos cuyo tamaño sea inferior a un valor especificado. Por tanto, los diferentes barridos unidos a la gran cantidad de comprobaciones que hay que realizar hacen que este filtro sea una tarea costosa.

4.4. Análisis de rendimiento

4.4.1. Transferencia de datos entre CPU y GPU

Como se explicó en la sección 4.1, en este trabajo las medidas del tiempo consumido por cada algoritmo se inician una vez la imagen hiperespectral se encuentra almacenada en la memoria global de la GPU, y terminan una vez los resultados de la detección son obtenidos, por lo que los tiempos de transferencia no se están teniendo en cuenta. Con el objetivo de comprobar si se consigue una mejora en cuanto al tiempo global de la aplicación, se van a sumar dichos tiempos de transferencia a los tiempos de cómputo para ver si se consigue, o no, una mejora al utilizar la GPU para realizar la detección de bordes.

Para obtener un buen ancho de banda en las transferencias se ha utilizado memoria page-locked que permite que las transferencias entre el host y el dispositivo sean mucho más rápidas ya que se aprovecha mejor el ancho de banda disponible. Los datos relativos a las transferencias se pueden ver en la siguiente tabla:

Datos	Tamaño (MB)	Transferencia (s)	Ancho de banda (GB/s)	Tipo de transf.
Pavia_centre.raw	79,9	0,02586	3,091	H. a D.
Otros datos	0.0012	0,000284	0,004	H. a D.
Mapas de bordes	79,9	0,03699	2,161	D. a H.

Tabla 4.9: Transferencia de datos.

A continuación se van a realizar las sumas de estos tiempos a los mejores tiempos para las dos versiones GPU, con reducción de ruido y sin reducción de ruido, para comprobar si los tiempos finales son inferiores a los tiempos de la versión OpenMP en CPU. Adicionalmente, también se desarrollaron dos nuevas versiones que trabajan con transferencias asíncronas con el objetivo de comparar y saber que ventajas ofrece el solapamiento de las transferencias con la computación.

La comparativa se realizará con los tiempos de las implementaciones que trabajan con la imagen entera, es decir con las implementaciones denominadas “multibanda”.

Solo detección de bordes: 0.011368s. (t. comp. GPU) + 0.063134s. (t. transf.) = 0,074502s.

Detección de bordes			
CPU-OMP.(s)	**GPU(s)**	**Speedup**	**Transferencias asíncronas**
0,619099	0,074502	8,31x	No
0,619099	0,051786	11,95x	Si

Tabla 4.10: Transferencias síncronas vs asíncronas.

Detección de bordes + reducción de ruido: 0.084990s. (t. comp. GPU) + 0.063134s. (t. transf.) = 0,148124s.

Detección de bordes + reducción de ruido			
CPU-OMP.(s)	**GPU(s)**	**Speedup**	**Transferencias asíncronas**
1,911469	0,148124	12,90x	No
1,911469	0,087125	21,93x	Si

Tabla 4.11: Transferencias síncronas vs asíncronas.

De acuerdo a los resultados obtenidos, tablas 4.10 y 4.11, se puede afirmar que la paralelización en GPU mejora considerablemente los tiempos de ejecución, sobre todo si se utilizan transferencias asíncronas que puedan solaparse con el proceso de computación. Cabe destacar los buenos resultados obtenidos para la versión donde se utiliza la reducción de ruido, consiguiéndose un speedup global de 21,93× sobre la misma versión pero implementada en CPU y paralelizada mediante OpenMP.

Capítulo 5

Conclusiones y trabajo futuro

La extensión del algoritmo de detección de bordes basado en entropía para imágenes en escala de grises a imágenes hiperespectrales se ha llevado a cabo con éxito. Para esta extensión se han aplicado diferentes técnicas de reducción de la información obteniendo mejores resultados con la técnica basada en la fusión de mapas de bordes. Además de su extensión, también se ha mejorado añadiéndole un filtro de ruido, lo cual hace que mejoren los resultados obtenidos.

Para comprobar su eficacia se ha comparado con otros métodos de detección de bordes en imágenes hiperespectrales, obteniendo que los resultados que ofrece este detector de bordes son mejores cuanta más información espectral se posee, es decir cuantas más bandas tiene la imagen. También se ha probado la robustez del algoritmo ante variaciones en los datos de entrada, comprobando que es un método bastante sensible al ruido en las imágenes y a los niveles de contraste de las mismas.

Por otra parte también se han llevado a cabo diferentes implementaciones del método detector de bordes, dos implementaciones para CPU y una para GPU. De las implementaciones en CPU, una es totalmente secuencial y la otra saca partido de los procesadores multi-núcleo haciendo uso de la API para programación paralela OpenMP. En lo referente a la implementación GPU, esta aprovecha varios de los niveles de paralelismo que presentan los datos, explotando implícitamente el paralelismo a nivel de píxeles y aplicando el método de forma paralela a las bandas de la imagen hiperespectral.

Para cada una de las implementaciones anteriores se han tomado medidas de tiempo y se han comparado con el objetivo de demostrar cual de ellas ofrecía mejores resultados. Como era de esperar, la implementación en GPU ofrece aceleraciones superiores a las versiones en CPU, alcanzándose por ejemplo un speedup máximo de 22$\times$ para la versión con reducción de ruido y de 54$\times$ para la versión sin reducción de ruido (comparaciones realizadas con la versión paralela en CPU y para la imagen hiperespectral Pavia centre con 102 bandas y dimensiones 715*1096). También se han analizado los tiempos de transferencia y su influencia en las aceleraciones a nivel global del algoritmo, obteniendo como resultado que la implementación en GPU sigue siendo más eficiente que las implementaciones en CPU aún sumando los tiempos de transferencia a los tiempos de cómputo.

La eficiencia de la implementación sobre GPU del método detector de bordes es consecuencia de las diferentes estrategias de optimización llevadas a cabo para los distintos kernels que conforman el método. En todos ellos se ha cuidado, en la medida de lo posible, el acceso coalescente a memoria global, la minimización de divergencias de hilos de un mismo warp, la ausencia de conflictos en el acceso a los bancos de memoria compartida, la maximización de la ocupancia y del aprovechamiento de los recursos de la GPU, buscando el tamaño de bloque de threads óptimo, maximizando el reuso de la memoria compartida y aprovechando diferentes características que ofrece CUDA como es el paralelismo dinámico, la ejecución concurrente de kernels mediante el uso de streams o los intercambios de información asíncronos para permitir el solape de la computación con las transferencias.

Por último, cabe señalar como trabajo futuro la implementación de la mejor técnica de fusión de información en GPU, actualmente el código GPU procesa toda la imagen hiperespectral y devuelve los resultados de cada una de las bandas, faltaría llevar a cabo la fusión de estos resultados en GPU. También como posible trabajo futuro podría plantearse la realización de pruebas menos subjetivas para valorar la eficacia del método detector de bordes sobre imágenes hiperespectrales. Estas pruebas podrían consistir en introducir este método en una cadena de procesamiento junto, por ejemplo, a clasificadores u otros métodos y medir su eficacia de manera indirecta, es decir, ver con qué método de detección de bordes clasifica mejor la cadena de procesamiento, si con este o con otro diferente.

Bibliografía

[1] Teledetección. http://concurso.cnice.mec.es/cnice2006/material121/unidad1/defin_td.htm *Consultado el 15 de junio de 2016.*

[2] J.M. Bioucas-Dias, A. Plaza, G. Camps-Valls, P. Scheunders, NM. Nasrabadi and J. Chanussot (2013), "Hyperspectral Remote Sensing Data Analysis and Future Challenges". *IEEE Geoscience and Remote Sensing Magazine,vol. 1, no. 2, pp. 6-36.*

[3] Fischer C. and Kakoulli I. (2006), "Multispectral and hyperspectral imaging technologies in conservation: current research and potential applications". *Reviews in Conservation 7, pp.3-16.*

[4] Liang H. (2012), "Advances in multispectral and hyperspectral imaging for archaeology and art conservation". *Applied Physics A., vol. 106, no. 2, pp. 309–323.*

[5] Edelman G. J. et al. (2012), "Hyperspectral imaging for non-contact analysis of forensic traces". *Forensic Science International, vol. 223(1-3), pp. 28-39.*

[6] Malkoff D. B., Oliver W. R. (2000), "Hyperspectral imaging applied to forensic medicine". *Proc. SPIE. 3920, pp. 108-116.*

[7] Gowen A. A. et al. (2007), "Hyperspectral imaging an emerging process analytical tool for food quality and safety control". *Trends Food Sci. Technol., vol. 18, no. 12, pp. 590-598.*

[8] Feng Y. Z., Sun D. W. (2012), "Application of hyperspectral imaging in food safety inspection and control: a review". *Crit. Rev. Food Sci. Nutr., vol. 52, no. 11, pp. 1039-1058.*

[9] Bernabé, S., Plaza, A., Marpu, P. R., Benediktsson, J. A. (2012)."A new parallel tool for classification of remotely sensed imagery". *Computers & Geosciences, vol. 46, pp. 208-218.*

[10] Lee, C. A., Gasster, S. D., Plaza, A., Chang, C. I., Huang, B. (2011). "Recent developments in high performance computing for remote sensing: A review". *Selected Topics in Applied Earth Observations and Remote Sensing, IEEE Journal of, vol. 4(3), pp. 508-527.*

[11] A. Plaza, J. Plaza, and H. Vegas (2010) , "Improving the performance of hyperspectral image and signal processing algorithms using parallel, distributed and specialized hardware based systems". *J. Signal Process. Syst., vol. 61, pp. 293–315.*

[12] S. Sánchez, G. Martín, A. Plaza, and C.-I. Chang (2010). "GPU implementation of fully constrained linear spectral unmixing for remotely sensed hyperspectral data exploitation". *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series, vol. 7810.*

[13] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell (2010). "A survey of general-purpose computation on graphics hardware". *Computer Graphics Forum, vol. 26, no. 1, pp. 80–113.*

[14] Erik Lindholm, John Nickolls, Stuart Oberman, and John Montrym (2008). "Nvidia tesla: A unified graphics and computing architecture". *IEEE Micro, vol. 28, pp.39–55.*

[15] NVIDIA Corporation. NVIDIA CUDA C Programming Guide. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf 2015, ver. 7.5 *Consultado el 15 de junio de 2016.*

[16] Richard Szeliski. (2010). Computer vision: Algorithms and Applications.

[17] Carlos Platero Dueñas (2009), "Técnicas de preprocesado". Apuntes de robótica y visión artificial. http://www.elai.upm.es/webantigua/spain/Asignaturas/Robotica/ApuntesVA/cap4Procesadov1.pdf *Consultado el 15 de junio de 2016.*

[18] Eduardo Dvorkin, Marcela Goldschmit, Mario Storti (2010), "Segmentación de imágenes digitales mediante umbralizado adaptativo en imágenes de color". *Mecánica Computacional, Vol. 29, p. 6177-6193, Buenos Aires, Argentina.*

[19] R. K. Haralick (1982), "Zero-crossings of second directional derivative operator". *SPIE Proc. On Robot Vision.*

[20] L. G. Roberts (1965), "Machine perception of three-dimensional solids". *Optical and Electro-Optical Information Processing, MIT Press*, pp. 159-197.

[21] J. M. S. Prewitt (1970), "Object enhancement and extraction". *Picture Processing and Psychopictorics, B. Lipkin and A. Rosenfeld, Eds. New York: Academic,* pp. 75-149.

[22] I. Sobel (1990), "An Isotropic 3x3 Gradient Operator, Machine Vision for Three Dimensional Scenes". *Freeman, H., Academic Press,* pp. 376–379.

[23] W. Frei and C. Chen (1977), "Fast Boundary Detection: A Generalization and New Algorithm". *IEEE Trans. Computers*, vol. C-26, no. 10, pp. 988-998.

[24] D. Marr and E. Hildreth (1980), "Theory of edge detection". *Proc. of the Royal Society of London, Series B*, vol.207, pp. 187-217.

[25] J. Canny (1986), "A computational approach to edge detection". *IEEE Trans. on PAMI*, vol. 8, no. 6, pp: 344-371.

[26] M. P. de Albuquerque, I. A. Esquef , A.R. Gesualdi Mello (2004), "Image Thresholding Using Tsallis Entropy". *Pattern Recognition Letters* vol. 25, pp. 1059–1065.

[27] Prasanna K. Sahoo, Gurdial Arora (2004), "A thresholding method based on two-dimensional Renyi's entropy". *Pattern Recognition,* no. 37, pp. 1149-1161.

[28] T. Pun (1980), "A new method for gray-level picture thresholding using the entropy of the histogram". *Signal Processing*, vol. 2, pp. 223-237.

[29] J. N. Kapur, P. K. Sahoo, and A. K. C. Wong (1985), "A new method for gray-level picture thresholding using the entropy of the histogram". *Computer Vision Graphics Image Processing*, vol. 29, pp. 273-285.

[30] Shannon, Claude E. (1948), .A Mathematical Theory of Communication". *Bell System Technical Journal*, vol. 27, pp. 379-423, 623-656.

[31] Mohamed A. El-Sayed et al. (2011), "New Edge Detection Technique based on the Shannon Entropy in Gray Level Images". *International Journal on Computer Science and Engineering (IJCSE),* vol. 3, no. 6, pp. 2224-2232.

[32] G.S. Robinson (1977), "Color edge detection". *Optical Engineering*, pp. 479–484.

[33] T. Huntsberger, M. Descalzi (1985), "Color edge detection". *Pattern Recognition Letter*, pp. 205–209.

[34] S. Di Zenzo (1986), "A note on the gradient of a multi-image". *Computer Vision, Graphics, and Image Processing,* pp. 116–125.

[35] P.W. Trahanias and A.N. Venetsanopoulos (1993), "Color edge detection using vector statistics". *IEEE Transactions on Image Processing,* vol. 2, pp. 259–264.

[36] S.Verzakov, P. Paclík, R. P. W. Duin (2006), "Edge Detection in Hyperspectral Imaging: Multivariate Statistical Approaches". *Structural, Syntactic, and Statistical Pattern Recognition* ,vol. 4109, pp. 551-559.

[37] V.C. Dinh, R. Leitner, P. Paclik and R. P. W. Duin (2009), "A Clustering Based Method for Edge Detection in Hyperspectral Images". *Lecture Notes in Computer Science, Springer*, vol. 5575, pp. 580-587.

[38] Mohamed A. El-Sayed. (2011), "A New Algorithm Based Entropic Threshold for Edge Detection in Images". *IJCSI International Journal of Computer Science Issues,* Vol. 8, Issue 5, No. 1, pp. 71-78.

[39] NVIDIA Corporation. Nvidia Tesla P100. https://images.nvidia.com/content/pdf/tesla/whitepaper/pascal-architecture-whitepaper.pdf *Consultado el 17 de junio de 2016.*

[40] NVIDIA Corporation. GeForce GTX 980 Whitepaper. http://international.download.nvidia.com/geforce-com/international/pdfs/GeForce_GTX_980_Whitepaper_FINAL.PDF *Consultado el 17 de junio de 2016.*

[41] NVIDIA Corporation. Maxwell tuning guide. http://docs.nvidia.com/cuda/maxwell-tuning-guide/#axzz47CYYO86X *Consultado el 16 de junio de 2016.*

[42] NVIDIA Corporation. GPU Pro tip: Fast histogram. https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-fast-histograms-using-shared-atomics-maxwell/ *Consultado el 16 de junio de 2016.*

[43] NVIDIA Corporation. GPU Pro tip: Fast histogram. http://www.fz-juelich.de/SharedDocs/Downloads/IAS/JSC/EN/slides/nvidia-ws-2014/02-adinets-maxwell.pdf?__blob=publicationFile *Consultado el 16 de junio de 2016.*

[44] NVIDIA Corporation. GPU Gems 3. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html *Consultado el 16 de junio de 2016.*

[45] NVIDIA Corporation. Optimizing Parallel Reduction in CUDA. http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf *Consultado el 16 de junio de 2016.*

[46] Raman Maini and Dr. Himanshu Aggarwal (2000), "Study and Comparison of Various Image Edge Detection Techniques". *International Journal of Image Processing (IJIP), vol. 3(1), pp. 1 – 12.*

[47] F. Arguello, D.L. Vilarino, D.B. Heras and A. Nieto (2014),"GPU-Based Segmentation of Retinal Blood Vessels". *Journal of Real-Time Image Processing, SPRINGER HEIDELBERG, pp. 1-10.* https://citius.usc.es/investigacion/publicacions/listado/gpu-based-segmentation-retinal-blood-vessels *Consultado el 17 de junio de 2016.*

[48] ROSIS. "University of Pavia dataset". http://www.ehu.eus/ccwintco/uploads/e/ee/PaviaU.mat *Consultado el 15 de junio de 2016.*

[49] AVIRIS."Salinas Valley dataset". http://www.ehu.eus/ccwintco/uploads/a/a3/Salinas_corrected.mat *Consultado el 15 de junio de 2016.*

[50] AVIRIS."Salinas RGB". http://zhaohuixue.weebly.com/uploads/4/0/0/6/40060385/1731198.jpg *Consultado el 15 de junio de 2016.*

[51] HYDICE. "DC Mall dataset". http://cobweb.ecn.purdue.edu/~biehl/Hyperspectral_Project.zip *Consultado el 15 de junio de 2016.*

[52] "Flightline C1 dataset". ftp://ftp.ecn.purdue.edu/biehl/MultiSpec/ModDimensionDataSet.zip *Consultado el 15 de junio de 2016.*

[53] ROSIS. "Pavia Centre dataset". http://www.ehu.eus/ccwintco/uploads/e/e3/Pavia.mat *Consultado el 15 de junio de 2016.*

[54] NVIDIA Corporation. How to Implement Performance Metrics in CUDA C/C++. https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc/ *Consultado el 15 de junio de 2016.*

[55] University of Southern California. The USC-SIPI Image Database. http://sipi.usc.edu/database/database.php *Consultado el 15 de junio de 2016.*