U

P

T

Fusión de imágenes multifoco usando unidades de procesamiento gráfico

por

Jorge Alberto Hernández Tapia

Tesis sometida como requisito parcial para obtener el grado de

MAESTRO EN COMPUTACIÓN ÓPTICA

en la

UNIVERSIDAD POLITÉCNICA DE TULANCINGO

Septiembre 2014

Tulancingo de Bravo, Hidalgo.

Supervisada por:

Dr. Juan Carlos Valdiviezo Navarro
Dra. Carina Toxqui Quitl

Dedicatoria

A mi madre por brindarme todo el amor, la compresión, su apoyo incondicional y su plena confianza en mi día con día.

Agradecimientos

De forma particular quiero agradecer a mis asesores el Dr. Juan Carlos Valdiviezo Navarro y la Dra. Carina Toxqui Quitl por su apoyo, y guía en el desarrollo de este trabajo de tesis.

A mis sinodales el Dr. Sergio Vázquez y Montiel, el Dr. Alfonso Padilla Vivanco y al MCC. Uriel Edgardo Escobar Franco por sus invaluables consejos y recomendaciones.

De igual forma me permito agradecer al Laboratorio de Óptica y Sistemas de Visión de la Universidad Politécnica de Tulancingo por las facilidades otorgadas para la realización de este trabajo de tesis.

Al rector el Maestro Gerardo Téllez Reyes por su invaluable apoyo al inicio de este ciclo de estudio mediante una beca de posgrado.

Abstract

For different applications ranging from biology to electronics, it is important to increase the visibility of objects whose dimensions are in microscopic scales. For this purpose, microscopy systems are used to acquire images of the object under study at different amplifications. However, as the magnification increases, the depth of field of the corresponding optical system decreases. This last causes that several image planes of the same object with different focused regions can be observed.

In order to obtain a completely focused high quality image, digital image processing algorithms, such as image fusion, can be effectively used. This research work introduces a new pixel-by-pixel technique for image fusion, based on the calculus of the Euclidean distance between adjacent pixels. In order to decrease the high computational time required to perform the fusion of several images, the technique is implemented on a Graphic Processing Unit (GPU). The technique here presented has allowed us to obtain highly focused images from several image planes, at low computational times. In fact, our proposal is suitable for real time applications.

Resumen

Para diferentes aplicaciones que van desde la biología a la electrónica, es importante aumentar la visibilidad de los objetos cuyas dimensiones se encuentran en escalas microscópicas. Con este propósito, se emplean sistemas de microscopía para adquirir imágenes del objeto de estudio a diferentes amplificaciones. Sin embargo, al incrementar la amplificación, la profundidad de campo del sistema óptico disminuye. Esto último genera diversos planos objeto donde se observan diferentes regiones enfocadas.

Con el objeto de obtener una imagen fusionada de calidad, se emplean algoritmos como la fusión de imágenes digitales. Este trabajo propone una nueva fusión de imágenes pixel a pixel basada en el cálculo de la distancia euclidiana entre pixeles vecinos. Para disminuir los elevados tiempos de cómputo requeridos para realizar una fusión con diferentes imágenes, la técnica mencionada será implementada en una unidad de procesamiento gráfico (GPU). La técnica aquí presentada ha proporcionado una imagen fusionada de alta calidad, con tiempos de cómputo que son factibles para aplicaciones cercanas al tiempo real.

Índice general

Índice de figuras

Capítulo 1

Introducción

La fusión de imágenes es el proceso de integrar la información adquirida por diferentes sensores de un sistema óptico, o en nuestro caso mediante un mismo sensor, con diferentes planos de enfoque, en una única imagen. La resolución de la cámara, las distancias entre la lente y el objeto, entre otros, son factores que modifican el número de elementos en la escena, limitando a la lente a centrase en sólo un objeto, o en una zona limitada del mismo. Por ejemplo en el caso de la figura 1.1 se puede observar una misma escena con diferentes planos de enfoque, generados por la utilización de diversas aperturas numéricas en una cámara fotográfica.

Figura 1.1: Fotografía digital con diversas profundidades de foco causadas por diferentes N.A.

Cuando se presentan diversos elementos en una escena, los objetos enfocados son perfectamente visibles mientras que el fondo u otros objetos son indistingibles o borrosos [1]. Sin embargo, la fusión de imágenes permite unir las imágenes originales enfocadas en distintos planos para producir una imagen donde todos los objetos estén enfocados [2]. La fusión de imágenes más simple consiste en tomar los niveles de gris de las imágenes originales y promediarlos para obtener una imagen "fusionada"; sin embargo, este método a menudo lleva a resultados no deseados como la pérdida de contraste [3]. Así pues, es necesario el desarrollo de técnicas más complejas para la fusión de imágenes que además sean eficientes en tiempo de computó permitiendo acercarse más a aplicaciones en tiempo real.

1.1. Trabajos Relacionados

La fusión de imágenes multifoco se ha convertido en un área de estudio importante en el campo del procesamiento de imágenes. En el ámbito de la fusión de imágenes existen numerosos artículos que aportan diversas soluciones a este problema, para lo cual se emplean diversas técnicas basadas en imágenes digitales para determinar el grado de enfoque en la escena, el cual a su vez está relacionado con el grado de contraste en la imagen. Por ejemplo, *Naidu* [4], propone una fusión basada en bloques de tamaño $N \times N$ empleando como medida de enfoque la Transformada Coseno Discreta, posteriormente se realizan pruebas con diversos tamaños de bloque que van desde 2×2 hasta 512×512 pixeles, con tiempos de cómputo que recaen dentro de los 40 a 50 segundos; siendo el tamaño máximo de las imágenes de prueba de 512×512 pixeles. *Li et al.* proponen un método basado de igual forma en la subdivisión por bloques y el uso de la frecuencia espacial como métrica de enfoque [5]; los autores afirman que el método propuesto es computacionalmente simple y proponen su uso en aplicaciones en tiempo real, asimismo aseguran que el método supera visual y cuantitativamente al basado en la transformada Wavelet, propuesto en [6].

Los trabajos antes citados se basan en determinar de forma empírica el mejor tamaño del segmento o bloque para realizar la fusión. Para solucionar este problema Jun Kong *et. al* proponen el uso de algoritmos genéticos para determinar la dimensión más óptima para la

subdivisión por bloques y el uso de la frecuencia espacial como métrica de enfoque [7]. Una limitante que presentan dichos trabajos es que la fusión se realiza sobre imágenes en escala de grises.

Dentro del contexto del color, los procedimientos para la fusión se han considerado como una extensión de las técnicas en escala de grises, de tal forma que el procedimiento que se realiza a un solo canal se replica a los otros dos. Asimismo, dichos trabajos sólo se enfocan en elementos macroscópicos y los autores generan imágenes "artificiales", donde las zonas de defoco son generadas por computadora.

Finalmente, entre los trabajos estudiados con enfoque en microscopía se encuentra el trabajo de Padilla *et al*, que proponen un método de fusión de imágenes de microscopía mediante el uso de Wavelets [6]. En este trabajo se puede observar una pérdida de contraste en la imagen fusionada ya que la imagen resultante pierde en gran medida los colores originales de las muestras de referencia. El trabajo de Chen Y. *et. al* propone una técnica basada en la transformación de las imágenes al espacio de color YIQ, extrayendo el componente de luminancia (Y) de las imágenes originales y aplicando como criterio de fusión para este canal la BEMD*(Bidimensional Empirical Mode Decomposition)*, mientras que para los canales (I,Q) se emplea un criterio basado en el PCA*(Análisis de Componentes Principales)* . Las imágenes que se emplean en dichos trabajos son imágenes en color y acorde a los autores fueron adquiridas por microscopio [8]. Este método propuesto preserva mayor información enfocada de cada plano comparado con el método basado en las transformada wavelet, además de una marcada mejora en el contraste.

1.2. Planteamiento del Problema

En microscopia se utilizan amplificaciones que van desde los 10× y pueden ser superiores a los 100×, por tanto, al aumentar la amplificación se ve disminuida la profundidad de campo. Debido a esto una sola imagen no es suficiente para obtener enfocado todo el volumen del objeto, generando diversas imágenes que poseen las diferentes zonas enfocadas del objeto de interés. Una solución a este problema en el campo de la visión por computadora es la fusión

de imágenes, que permite aumentar de forma directa la *profundidad de campo*. Los métodos actuales presentan la aparición de conjuntos o bloques de pixeles con forma rectangular ocasionados principalmente por la división de las imágenes originales en bloques rectangulares. Dichas regiones a las que denominaremos como artefactos, presentan tonalidades que no corresponden con el resto de los pixeles vecinos. Por otra parte los tiempos de cómputo que se han documentado en trabajos previos son elevados puesto que el procesador principal tiene que soportar una gran carga de trabajo, lo que lleva a que las técnicas antes descritas no sean viables para aplicaciones que requieran un tiempo bajo de procesamiento.

1.3. Objetivo General

Implementar una técnica novedosa para realizar la fusión de imágenes provenientes de muestras microscópicas, usando unidades de procesamiento gráfico para su aplicación en tiempo real.

1.3.1. Objetivos Particulares

1. Realizar una estudio de las métricas frecuentemente utilizadas en la literatura para medir el enfoque en imágenes digitales.

2. Implementar algoritmos de fusión en una unidad de procesamiento gráfico (GPU) para su ejecución mediante cómputo paralelo.

3. Implementar un algoritmo de fusión de imágenes multifoco que sea computacionalmente eficiente.

4. Realizar una comparativa en tiempos de computó de los algoritmos tanto en CPU como en GPU.

1.4. Justificación

Una extensa variedad de algoritmos de fusión de imágenes han sido presentados en los últimos años, que en su mayoría se utilizan para fusión de imágenes en escala de grises, donde la idea principal es "quitar" aquellos segmentos desenfocados de una imagen para ser reemplazados por las zonas enfocadas que se presentan en el resto de las imágenes origen. Lo anterior lleva a una subdivisión por bloques cuadrados, que no coinciden con la morfología de la mayoría de los objetos en una escena, lo que produce artefactos en la imagen fusionada. Adicionalmente, puesto que la gran mayoría de técnicas se basan en dicha subdivisión por bloques, encontrar el tamaño ideal suele requerir un tiempo considerable de cómputo.

Finalmente, los pocos trabajos orientados a microscopía nos llevan a buscar una solución que mejore los puntos previamente descritos y que además solucione el problema de fusión de imágenes en tiempos de cómputo bajos, cercanos al tiempo real.

1.5. Metodología

Se realizará la adquisición de las muestras sobre las que se probarán los métodos presentados a lo largo de este trabajo de tesis, con el objeto de no recurrir a ninguna base de datos existente. Una vez que dichas imágenes sean adquiridas, los algoritmos se programarán de forma secuencial para su ejecución en una *Unidad Central de Procesamiento* (CPU), con el objetivo de analizar de forma detallada el comportamiento de los mismos. Habiendo probado los algoritmos de forma secuencial, se procederá a realizar su implementación en una *Unidad de Procesamiento Gráfico* (GPU), usando programación en paralelo. En base a la cuantificación de los resultados obtenidos, en cuanto a calidad de la imagen fusionada y tiempo de ejecución, se habrá de determinar el algoritmo más óptimo para imágenes de microscopía.

Para poder realizar las pruebas sobre las muestras adquiridas, se medirá el tiempo de cómputo necesario para la ejecución de las mismas tanto en la CPU como en GPU y finalmente, se medirá de forma cuantitativa la calidad de imagen obtenida para determinar cuál es la mejor técnica en términos de tiempo de cómputo y calidad de imagen.

Este trabajo de tesis se conforma de la siguiente manera: el capítulo 2 presenta una breve

introducción a los sistemas formadores de imágenes, en particular de un microscopio. Además, se describen los parámetros de adquisición de imágenes empleados en este trabajo para la toma de muestras. Por otra parte, el capítulo 3 describe las métricas comúnmente utilizadas para medir el nivel de contraste o las altas frecuencias de una imagen que está asociado al nivel de "enfoque" de la misma. De igual forma en el mismo capítulo se discuten las modificaciones a nivel lógico que fueron necesarias realizar a algunos algoritmos para su correcto funcionamiento en forma paralela, necesaria para trabajar en la GPU. El capítulo 4 presenta una introducción al lenguaje de programación requerido para trabajar en una GPU, así como una revisión a los tiempos de cómputo entre los lenguajes de programación más empleados con CUDA, lo que permitió definir el lenguaje de programación empleado en este trabajo de tesis. En el capítulo 5 se presentan los resultados de implementar dos métricas de fusión: una basada en la distancia euclidiana y otra basada en el ángulo espectral; en este mismo capítulo se hace un análisis cuantitativo de los resultados de fusión obtenidos en cuanto a calidad de la imagen y tiempo de cómputo. Finalmente las conclusiones y el trabajo a futuro son presentados en el capítulo 6.

1.6. Aportaciones

El desarrollo de este trabajo de tesis contribuye con las siguientes aportaciones

- Un análisis sobre la factilibidad de la ejecución en paralelo de algoritmos que miden el contraste sobre la GPU [9].
- La implementación en paralelo de algoritmos de autoenfoque.
- Un algoritmo de fusión pixel a pixel eficiente en tiempos de cómputo y con una buena calidad de imagen.
- La técnica desarrollada incrementa, de forma digital, la profundidad de campo en los sistemas ópticos.
- La técnica de fusión puede ayudar a mejorar el diagnóstico y análisis de muestras microscópicas.

Bibliografía

[1] Q. Li, J. Du, L. Xu. "Multi-focus image fusion using the local fractal dimension". *Int J Adv Robot Syst*, Vol. 10, pp. 1–11, (2013).

[2] S. Li, and B. Yang. "Multifocus image fusion using region segmentation and spatial frequency", *Image Vision Comput*, Vol.26, No. 7, pp. 971–979, (2008).

[3] E. Ahmet-M., P.S Fisher, "Image quality measures and their performance," *IEEE Transactions on Communications* , Vol. 43, No. 12, pp. 2959–2965, (1995).

[4] VPS Naidu, "Discrete cosine transform based image fusion techniques", *Journal of Communication, Navigation and Signal Processing*, Vol. 1, No. 1, pp. 35–45, (2012).

[5] S. Li, J.T Kwok, Y. Wang, "Combination of images with diverse focuses using the spatial frequency", *Information Fusion*, Vol. 2, No. 3, pp. 169–176, (2001).

[6] A. Padilla-Vivanco, I. Tellez-Arriaga, C. Toxqui-Quitl, C. Santiago-Tepantlan, "Multifocus microscope color image fusion based on Daub(2) and Daub(4) kernels of the Daubechies wavelet family", *Proc. SPIE: Applications of Digital Image Processing XXXII*, Vol. 7443, pp. (2009).

[7] J. Kong, K. Zheng, J. Zhang, X. Feng., "Multi-focus Image Fusion Using Spatial Frequency and Genetic Algorithm," *IJCSNS International Journal of Computer Science and Network Security*, Vol. 8 No. 2, pp.1–9, (2008).

[8] Y. Chen, L. Wang, Z. Sun, Y. Jiang, G. Zhai, "Fusion of color microscopic images based on bidimensional empirical mode decomposition.", *Opt Express*, Vol. 18, No. 21, pp. 1–12, (2010).

[9] J. C. Valdiviezo-N., J. Hernández-Tapia ; L. Mera-González, C. Toxqui-Quitl, A. Padilla-Vivanco; "Autofocusing in microscopy systems using graphics processing units", *Proc. SPIE: Applications of Digital Image Processing XXXVI*, Vol. 8856, pp.1-9 (2013).

Capítulo 2

Introducción a los sistemas formadores de imágenes

2.1. Formación de imágenes

La formación de imágenes en un sistema óptico puede depender de la difracción, reflexión o refracción. El enfoque de este trabajo estará orientado a un sistema óptico basado en refracción. La refracción puede ser definida como el efecto en el que se curvan los rayos de luz al pasar de un medio a otro más denso, donde su velocidad de propagación es diferente; por ejemplo, el cambio al propagarse de aire a vidrio. Puesto que la velocidad de la luz es menor en un medio como el vidrio en comparación al aire, un rayo de luz se curvará al entrar y salir de una lente en una dirección que depende tanto de la forma como de la curvatura de la lente. En el caso de una lente convergente, como la que se muestra en la figura 2.1, los rayos de luz paralelos al eje óptico serán reunidos en un solo punto; el punto donde dichos rayos convergen se conoce como foco f, y la distancia entre la lente y el foco principal de la misma se conoce como *distancia focal*.

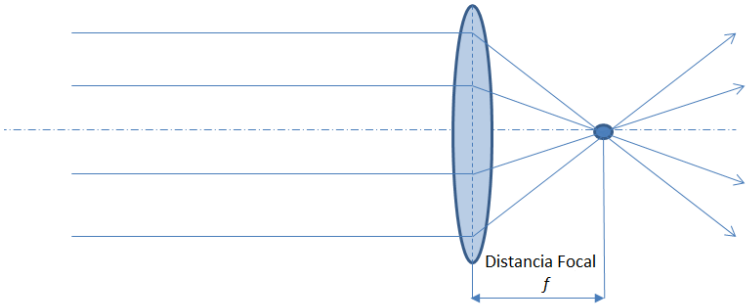


Figura 2.1: Los rayos de luz provenientes de un objeto se concentran en un solo punto, a la distancia focal (f) de la lente.

Para una lente delgada convergente se dice que la imagen de un punto sobre el objeto se puede obtener trazando tres rayos como sigue: 1) un rayo que viaja paralelo al eje óptico de la lente, tras refractarse la prolongación de dicho rayo pasa por el foco del lado de la imagen de esta lente, 2) un rayo central o rayo principal que atraviesa el centro de la lente, el cual no modifica su trayectoria, y 3) un rayo que pasa por el foco objeto que atraviesa la lente y después de atravesar la lente es paralelo al eje óptico. La Figura 2.2 muestra la localización del plano objeto y del plano imagen para una lente delgada así como el trazo de rayos de un objeto. Las distancias objeto (s_o), e imagen (s_i) y la distancia focal (f) están relacionados por la *fórmula de Gauss*, la cual está dada por,

$$\frac{1}{s_i} - \frac{1}{s_o} = \frac{1}{f} \tag{2.1}$$

Variaciones en la distancia s_o producen una imagen en posiciones s_i diferentes, modificando su amplificación así como el tipo de imagen que una lente positiva produce. En la Figura 2.3 (a) se observa que si el objeto se localiza a dos veces la distancia focal, la imagen será **real**, **invertida** y **del mismo tamaño**. En otro caso, si el objeto se sitúa a una distancia $f \leq |s_o| \leq 2f$, la imagen será **real**, **invertida** y **amplificada**. Si el objeto se sitúa a una distancia $f \leq |s_o| \leq 2f$, la imagen será **real invertida** y **amplificada**. Por otro lado si el objeto se sitúa a una distancia $|s_o| = f$ se dice que la imagen se forma en el **infinito** Fig. 2.3 (c).

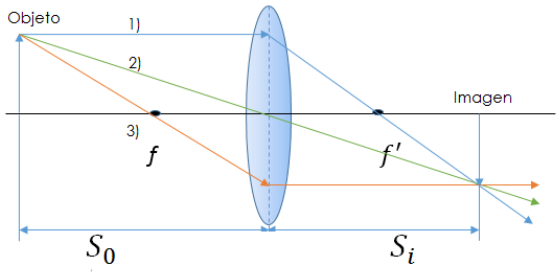


Figura 2.2: Localización de los planos objeto e imagen para una lente delgada

Si el objeto se sitúa entre la lente y el foco, se obtiene una **imagen virtual**, **derecha** y **de mayor tamaño**. Finalmente cuando la distancia $|s_o|$ es mayor que $2f$, la imagen producida será **real**, **invertida** y **de menor tamaño**. Recordemos que una imagen real es aquella que se forma cuando al pasar por el sistema óptico los rayos son convergentes, es decir, esta imagen la podemos percibir con el ojo y registrarla usando una pantalla en el punto donde convergen los rayos. Caso contrario, una imagen virtual se forma cuando al pasar por el sistema óptico, los rayos divergen. Las imágenes virtuales no se pueden registrar sobre una pantalla pero es posible percibirlas con el ojo [1].

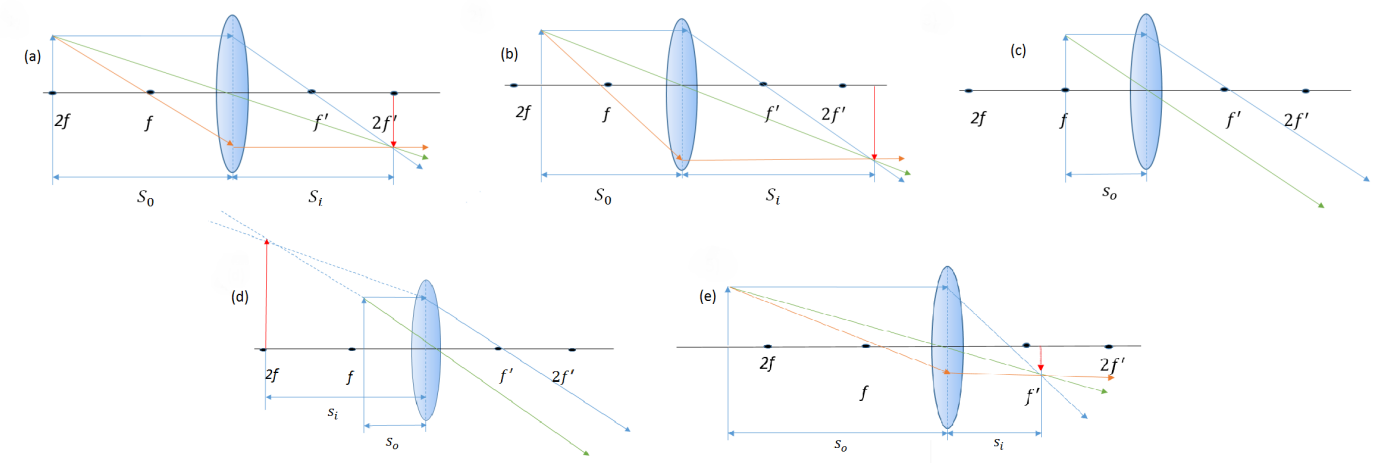


Figura 2.3: Formación de imágenes a partir de un objeto en (a) $\mid s_o \mid = 2f$, (b) $2f < \mid s_o \mid < f$, (c) $\mid s_o \mid = f$, (d) $\mid s_o \mid < f$, (e) $\mid s_o \mid > 2f$

2.2. Enfoque

En cualquier sistema óptico se dice que una imagen está enfocada cuando el objeto a capturar aparece nítido en la imagen obtenida, es decir, los rayos procedentes del objeto se concentran en el plano de la imagen. La calidad de una imagen digital depende de varios factores algunos puramente subjetivos como el movimiento de la cámara o sensor al momento de adquirir la imagen. Algunos parámetros como la nitidez dependen de otros factores como son: la lente, la apertura numérica de la cámara, la distancia entre el plano imagen y el punto donde se coloca sensor. Es posible decir entonces que enfocar es lograr que los rayos de luz provenientes de cada uno de los puntos del objeto se concentren de forma puntual en la superficie del sensor como se observa en la figura 2.4.

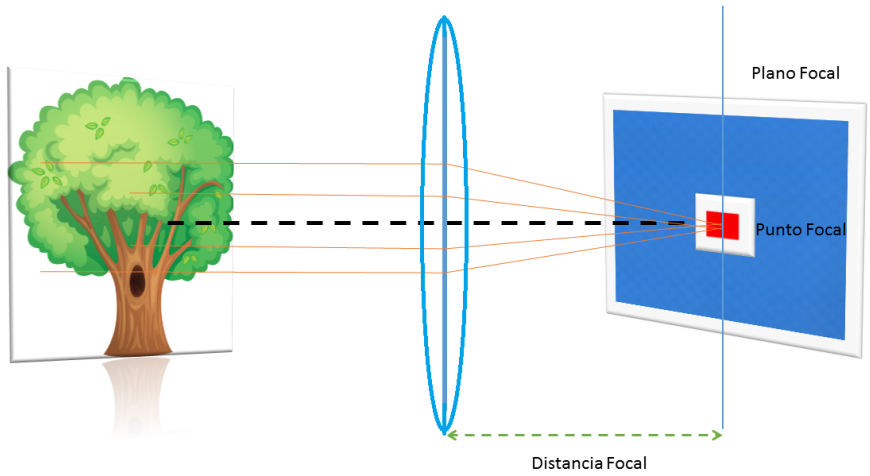


Figura 2.4: Concepto de imagen nítida: los rayos de luz se concentran de forma uniforme sobre la superficie del sensor.

2.2.1. Profundidad de Campo

Se denomina *profundidad de campo* (h) a la distancia comprendida entre el punto más proximo y el más lejano del objeto que está siendo enfocado, donde los detalles del mismo

pueden ser capturados nítidamente. Si parte del objeto que se observa cae antes o después de dicha zona, entonces la imagen correspondiente aparecerá desenfocada [2].

En otras palabras la profundidad de campo se define como el conjunto de posiciones o desplazamientos donde el ojo es incapaz de detectar cambios en la nitidez de la imagen cuando distancia objeto s_o se modifica.

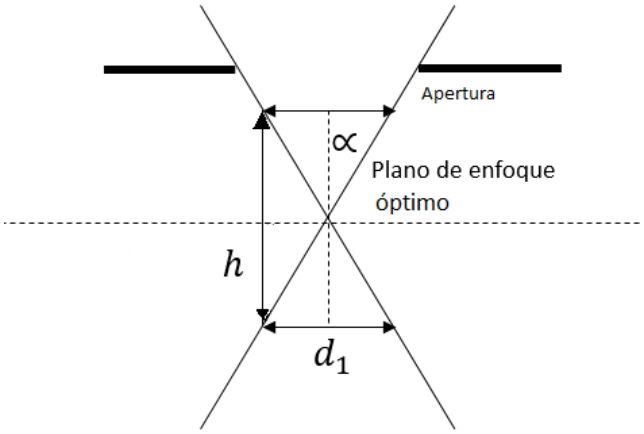


Figura 2.5: Esquema utilizado para explicar el concepto de profundidad de campo

El valor de h puede obtenerse a partir de la Figura 2.5, donde α corresponde a la mitad del ángulo que se substiende respecto de una apertura numérica(ver Figura 2.6). Por otra parte el límite de resolución r_1 está definido por

$$r_1 = \frac{d_1}{2} = \frac{C\lambda}{n \sin \alpha} = \frac{C\lambda}{\text{N.A.}}, \tag{2.2}$$

donde C corresponde a una constante de proporcionalidad relacionada con el grado de superposición de dos puntos que puedan ser resueltos por el ojo humano, cuyo valor es de 0.61; λ corresponde a la longitud de onda de la fuente de iluminación; n representa el índice de refracción entre el objeto y la lente objetivo; el producto $n \sin \alpha$ es llamado comúnmente como la *apertura numérica* (N.A.).

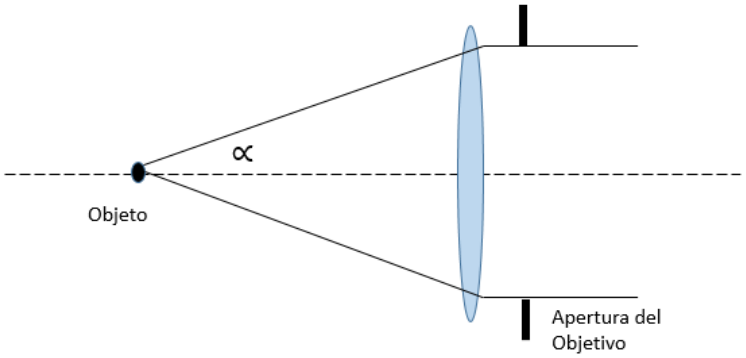


Figura 2.6: Ilustración de la mitad de ángulo que se substiende, en este caso respecto de la apertura del objetivo.

De acuerdo con la figura 2.5 y en base a la ecuación anterior es posible definir la profundidad de campo como:

$$h = \frac{0{,}61\lambda}{n \sin\alpha \tan\alpha} = \frac{0{,}61\lambda}{\text{N.A.} \tan\alpha}. \tag{2.3}$$

La figura 2.7 muestra una imagen macroscópica con diferentes profundidades de campo.

Figura 2.7: De izquierda a derecha: imagen con poca profundidad de campo, imagen con profundidad de campo moderada

2.2.2. Profundidad de Foco

La *profundidad de foco* es la distancia total axial a partir del plano de la imagen hasta el punto donde se coloca el sensor de la cámara y la imagen registrada por éste sigue estando enfocada [3]. Usualmente la profundidad de foco, que es del orden de los micrómetros, es aproximadamente equivalente a la profundidad de campo. Este margen de tolerancia varía considerablemente en función de la apertura numérica del objetivo y del aumento del mismo; de esta manera, con un apertura numérica pequeña se tiene una mayor profundidad de foco, pero también implica una baja amplificación. Si se considera el criterio de resolución de Rayleigh[1], una tolerancia aceptada comúnmente para obtener el nivel de defoco de una imagen Δz, puede resolverse geométricamente por la siguiente ecuación [4],

$$\Delta z = \pm \frac{\lambda}{2(\text{N.A.})^2} \tag{2.4}$$

donde λ es la longitud de onda y N.A. corresponde a la apertura numérica del objetivo. Es importante destacar que para un sistema real, tanto la profundidad de campo, como la profundidad de foco, son fuertemente dependientes de la aplicación que se busca y del hardware que se posee.

2.3. Formación de imágenes en un microscopio

El microscopio es un instrumento que permite observar objetos que son demasiado pequeños para ser vistos a simple vista, fue inventado por Zacharias Janssen en 1590. En su forma más simple, la óptica del microscopio se compone de dos lentes, el primero con una distancia focal muy corta, conocido como el *objetivo* y el segundo es conocido como *ocular*. A pesar de que en la actualidad dichas ópticas incluyen diversos elementos que ayudan a reducir las aberraciones ópticas, su funcionamiento puede ser ilustrado con lentes simples como se muestra en la figura 2.8 [5]. De acuerdo a dicha figura el objeto, marcado con (1), se sitúa dentro de la distancia focal del objetivo de tal forma que generará una imagen real y amplificada de la imagen en (2).

[1]Es el criterio generalmente aceptado para la resolución del mínimo detalle. Precisa en que condiciones se pueden distinguir dos fuentes puntuales

Esta imagen se convierte entonces en el objeto para la segunda lente, el ocular. Funcionando como una lente de aumento, el ocular forma una imagen virtual en (3), la cual se convierte en el objeto que verá el ojo, formando la imagen final en (4).

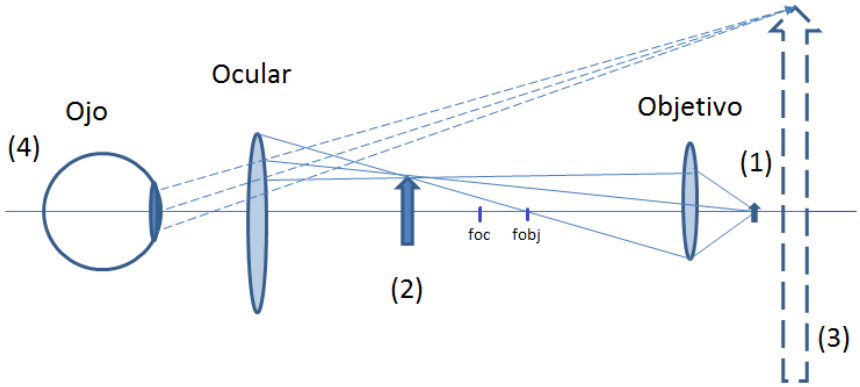


Figura 2.8: Principio de formación de imágenes en el microscopio

2.3.1. Apertura Numérica (N.A.)

La apertura numérica brinda una indicación de la capacidad de colección de luz y poder de resolución (a una distancia fija) de un objetivo. A mayor N.A. mayor resolución. Los objetivos con mayores N.A. son generalmente preferidos en la mayoría de las aplicaciones de imágenes. Un objetivo con una gran N.A. colecta más luz, brinda imágenes más brillantes, es capaz de definir detalles más finos y generalmente tienen distancias de trabajos más cortas. La apertura numérica se define mediante la siguiente fórmula:

$$\text{N.A.} = n \sin \alpha, \tag{2.5}$$

donde n es el índice de refracción del medio de trabajo de la lente y α es la mitad de la apertura angular de la lente [6]. Así pues para un ángulo $\alpha = 64°$ la mitad de este es 32° y entonces su apertura numérica será de 0.6, como muestra la figura 2.9.

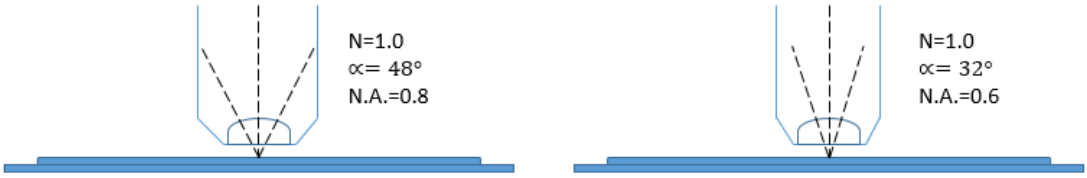


Figura 2.9: N.A. para diferentes objetivos

2.3.2. Número f (#f)

El *número f*, escrito usualmente como $\#f$, es una medida que indica la irradiancia de luz que incide en el medio de grabado (película fotográfica, sensor CCD, etc). Está relacionado tanto con la apertura numérica como con la distancia focal de la lente y en general provee una idea de la capacidad para colectar luz de un sistema óptico [4].

El $\#f$ se calcula como la razón de la distancia focal de la lente (f) al diámetro de la pupila de entrada (D). En el caso de imágenes de microscopia esta relación queda expresada por la apertura numérica.

Si bien el $\#f$ y la N.A. pueden parecer diferentes, ambos expresan la capacidad de colección de luz de una lente fotográfica o de un objetivo de microscopio. El $\#f$ puede ser convertido a su valor en apertura numérica y viceversa, empleando las siguientes ecuaciones:

$$\#f = \frac{1}{2\text{N.A.}} \quad ; \quad \text{N.A.} = \frac{1}{2 \cdot \#f} \tag{2.6}$$

2.4. Adquisición de Imágenes de Microscopía

Las imágenes presentadas en este trabajo fueron tomadas empleando un microscopio *Axio Imager M1*, (Carl Zeiss, Jena, Alemania), mediante iluminación por reflexión que se encuentra en el Laboratio de Óptica y Sistemas de Visión de la UPT. Objetivos con diversos aumentos que van desde 2× hasta los 100×fueron empleados para tomar las imágenes. El microscopio cuenta además con una plataforma capaz de desplazarse en los planos x, y y z. Las imágenes

que se obtienen poseen una resolución de 2584×1936 pixeles, y un tamaño de pixel de 8,7μm $\times$ 6,6μm.

Cada plano de imagen fue determinado de forma manual por un observador y posteriormente se relizaron N numero de pasos Δz, a lo largo del eje z. Los detalles como la apertura numérica (N.A.), la amplificación empleada y el tamaño de los pasos Δz se explicarán a detalle en cada una de las imágenes de prueba.

La figura 2.10, corresponde al microscopio utilizado, las principales partes se describen a continuación:

1. Cámara CCD o sensor CCD: las imágenes vistas por el ocular son llevadas para su grabado a este sensor con resolución aproximada de 5 Mpx en formato 4:3.

2. Oculares: pueden ser uno (monoculares), o como en este caso binoculares, es el área donde se posan los ojos y su función es ampliar la imagen de los objetivos.

3. Revólver: es donde se alojan los objetivos, puede tener espacio para tres, cuatro o hasta 6 objetivos

4. Platina: zona donde se coloca la muestra, su posición es controlable.

5. Objetivos: son los "lentes" más cercanos a la muestra y son los encargados de la amplificación que varía desde $4\times, 10\times, 40\times, 45\times$, hasta $100\times$; sin embargo, este último corresponde a una objetivo de inmersión.

6. Pantalla Táctil TFT: presente en microscopios modernos, básicamente se usa para intercambiar los objetivos y controlar parcialmente por software el microscopio.

7. Control para el Eje z: Como su nombre lo indica se controla la profundidad o el acercamiento al eje z.

8. Macrométrico: tornillo de enfoque para acercamiento rápido. Se recomienda para amplificaciones pequeñas.

9. Micrométrico: tornillo de enfoque para acercamiento más sutil.

Figura 2.10: Microscopio Axio Imager M1 utilizado para registrar las imágenes de esta investigación

2.4.1. Imágenes adquiridas

A continuación se presentan algunas de las imágenes que se emplearán en las pruebas de este trabajo de tesis. Las imágenes fueron registradas con la misma cámara para no afectar las pruebas. El sensor posee una resolución correspondiente a 5 Mpx , que es el equivalente a una resolución de 2584×1936 pixeles. Cada imagen presenta diferentes amplificaciones y fueron tomadas a distintos desplazamientos en el eje z del microscopio (Δz). La apertura numérica se describirá de forma particular en cada muestra. Adicionalmente es importante recalcar que las muestras empleadas corresponden tanto a metales, como minerales, y muestras biológicas.

Moneda Mexicana 1

Imagen correspondiente a una pequeña región de una moneda Mexicana. La imagen fue tomada con un aumento de 10×, que corresponde a una apertura numérica de 0,30, en iluminación de campo brillante, empleando luz reflejada. La muestra consiste de 3 planos con diferente enfoque, las cuales se muestran en la Figura 2.11.

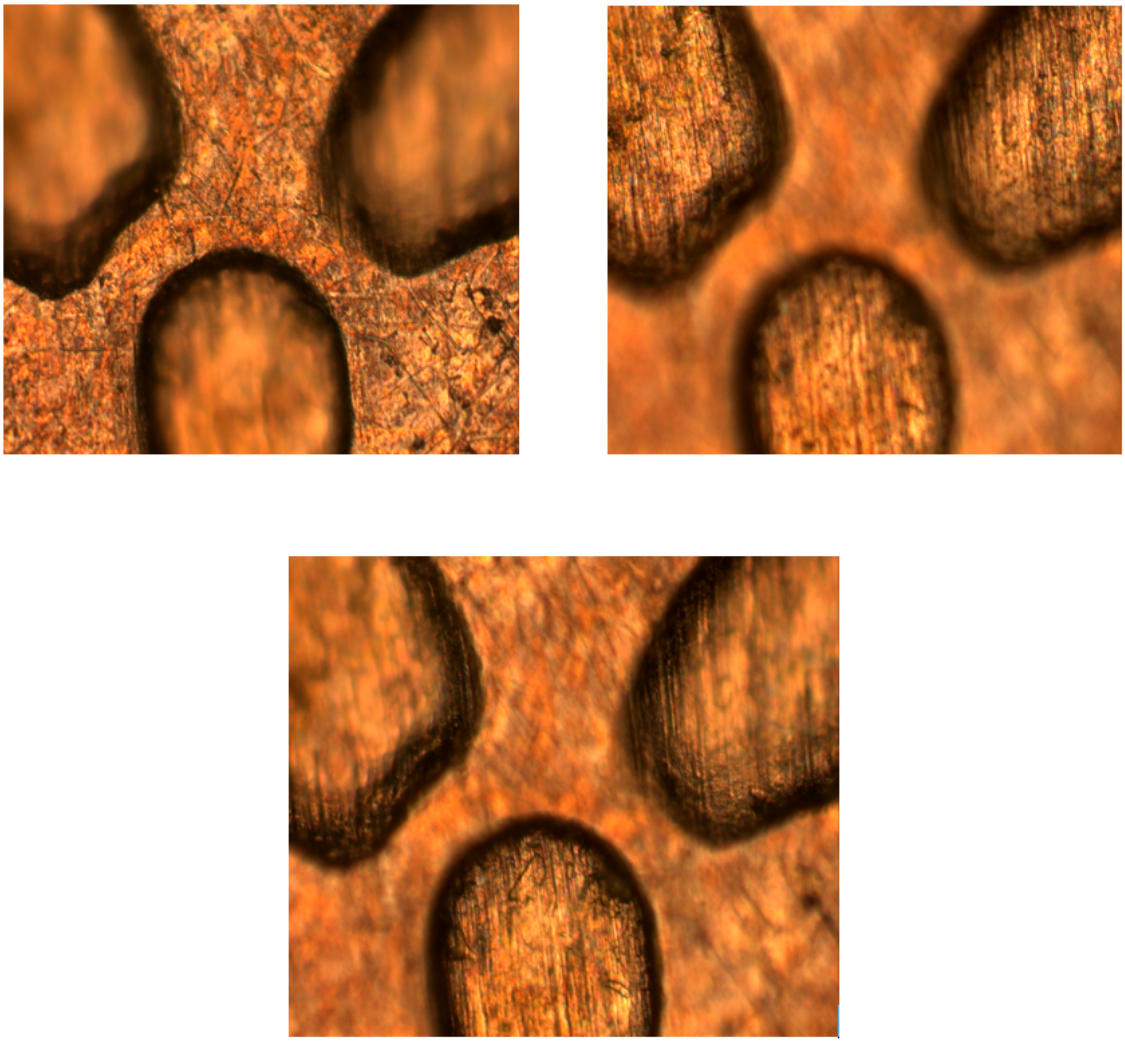


Figura 2.11: Imagen de una moneda mexicana con tres diferentes planos de enfoque. Imagen tomada con un objetivo de 10×, N.A.= 0,30, en iluminación de campo brillante.

Muestra de Grafito

Imagen correspondiente a un trozo de grafito. La imagen fue tomada con un aumento de 10× , que corresponde a una apertura numérica de 0,30 en iluminación de campo brillante, empleando luz reflejada. La muestra consiste de 3 planos con diferente enfoque como se muestra en la Figura: 2.12.

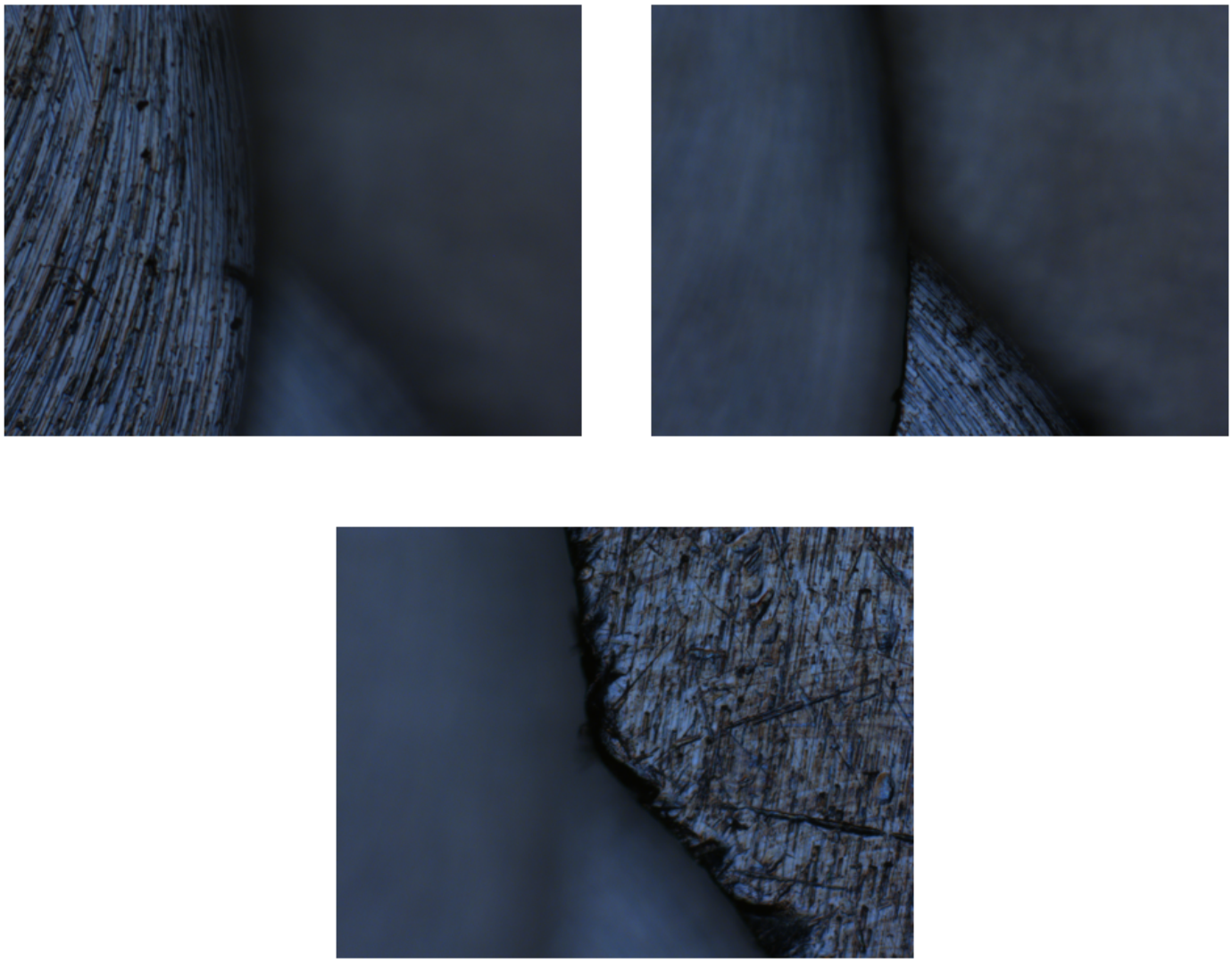


Figura 2.12: Muestra de grafito con tres planos de enfoque, amplificación de 10×, N.A. de 0,30 en iluminación de campo brillante.

2.4.2. Muestra de metal desvastado

Muestra correspondiente a un metal que fue desvastado aplicandole ácido de forma gradual. Dicha muestra fue tomada con un objetivo de 2,5×, correspondiente a una N.A. de 0,16, los desplazamientos en el eje Z corresponde a $\Delta z = 50\mu$m, mediante iluminación de campo brillante y luz reflejada. La muestra se conforma por 23 planos, la figura 2.13 muestra los planos 1, 6, 12, y 18 respectivamente.

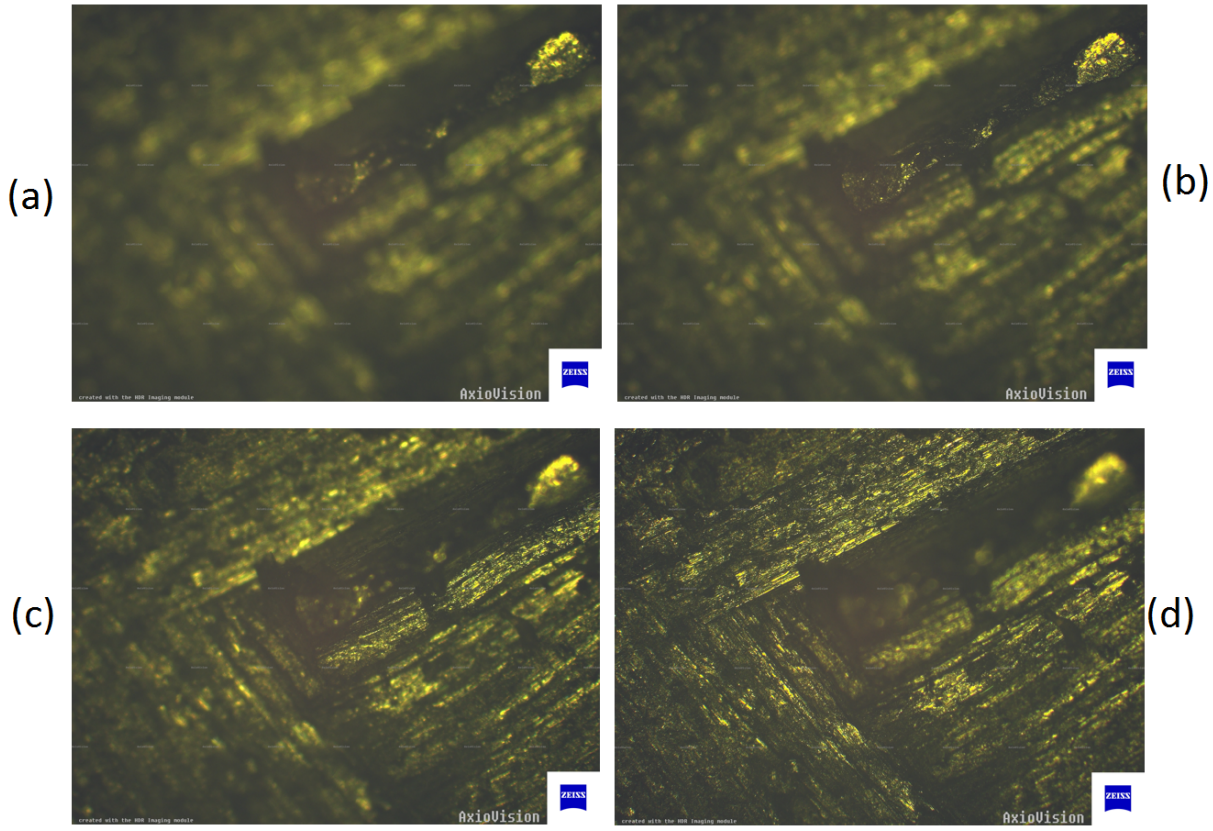


Figura 2.13: Muestra de metal desvastado con 23 planos con diferente enfoque, amplificación de 2,5×, N.A.=0.16. (a) plano 1, (b) plano 6, (c) plano 12, y (d) plano 18.

Llave de aluminio

La muestra siguiente corresponde a una zona de una llave metálica. La imagen fue tomada a 10×, lo que conlleva una N.A. de 0.3, los desplazamientos corresponden a una $\Delta z = 10\mu$m, mediante iluminación de campo brillante y luz reflejada. La muestra se conforma por 19 planos, la figura 2.14 muestra los planos 1, 5, 10, y 15 respectivamente.

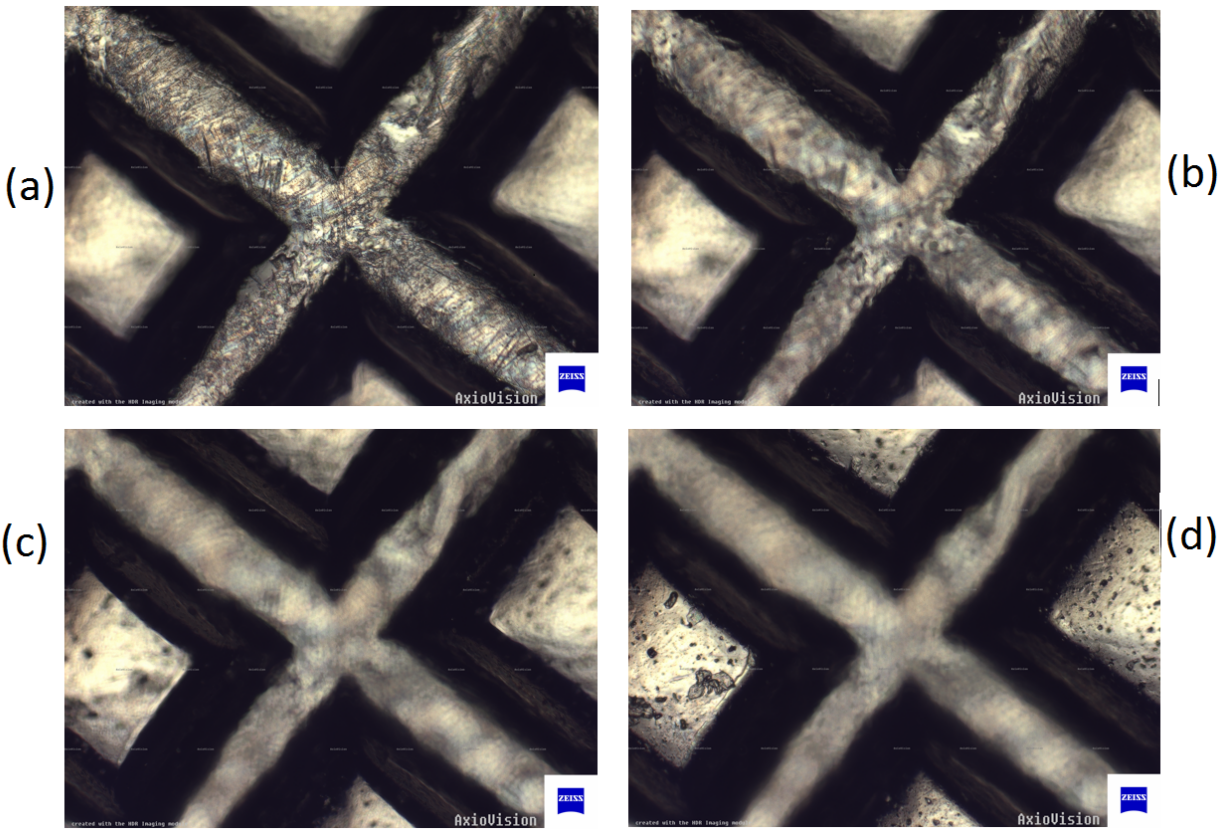


Figura 2.14: Muestra de una zona de una llave de aluminio, la muestra cuenta con 19 planos en diferente foco, amplificación de 10×, N.A. de 0.30 en iluminación de campo brillante.

Llave de aluminio (Otra Región)

La muestra siguiente corresponde a una zona (diferente a la anterior) de una llave metálica. La imagen fue tomada a 20×, lo que conlleva una N.A. de 0.75, los desplazamientos corresponden a una $\Delta z = 2\mu$m, mediante iluminación de campo brillante y luz reflejada. La muestra se conforma por 18 planos, la figura 2.15 muestra los planos 1, 5, 10, y 15 respectivamente.

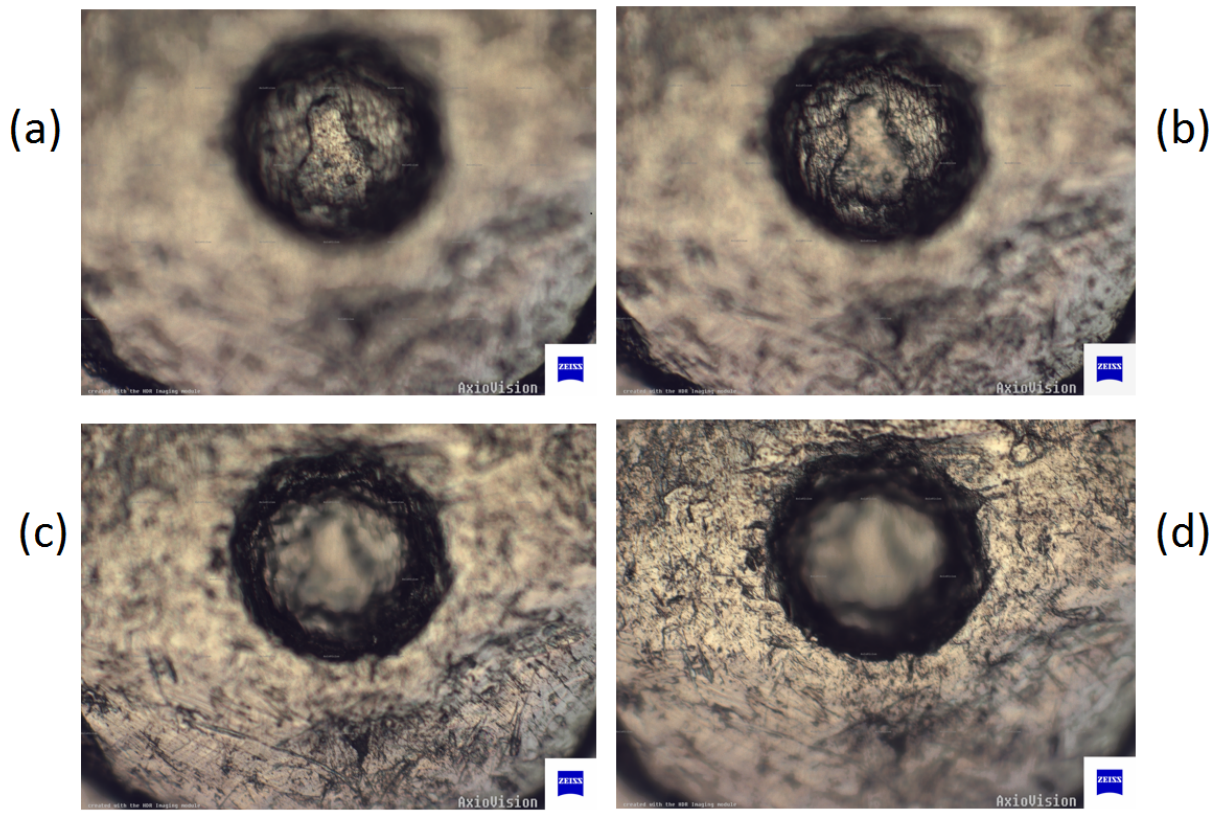


Figura 2.15: Muestra de una zona de una llave de aluminio, la muestra cuenta con 18 planos en diferente foco, amplificación de 20×, N.A. de 0.75. (a) plano 1, (b) plano 5, (c) plano 10, (d) y plano 15

Moneda mexicana (Cobre)

La muestra siguiente corresponde a una moneda mexicana de cobre. La imagen fue tomada a 10×, lo que conlleva una N.A. de 0.3, los desplazamientos corresponden a una $\Delta z = 5\mu$m, mediante iluminación de campo brillante y luz reflejada. La muestra se conforma por 11 planos, la figura 2.16 muestra los planos 1, 3, 6, y 9 respectivamente.

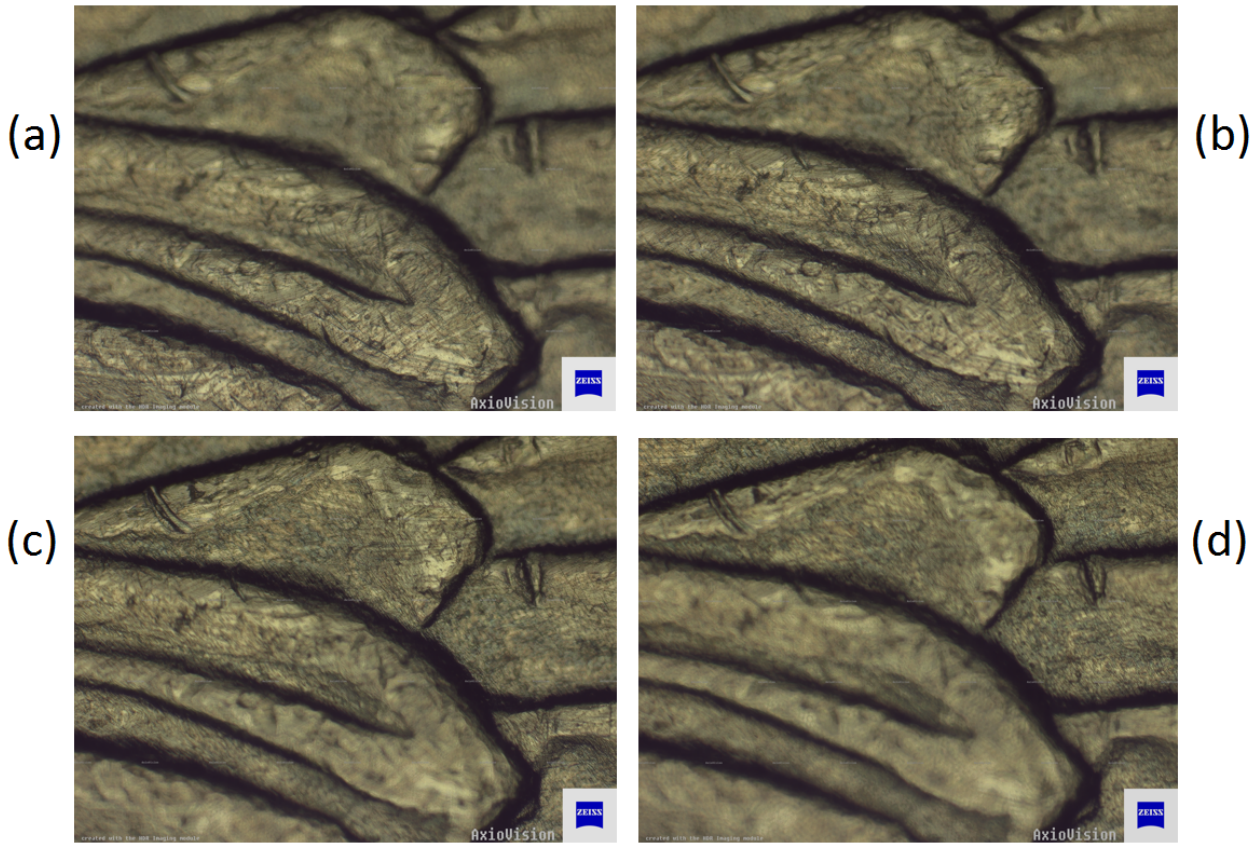


Figura 2.16: Muestra de una zona de una moneda de cobre, la muestra cuenta con 11 planos en diferente foco, amplificación de 10×, N.A. = 0.3. (a) plano 1, (b) plano 3, (c) plano 6 y (d) plano 9.

Espina dorsal de un pez

Imagen correspondiente a un corte de una espina dorsal de un pez. La muestra fue tomada con un objetivo de 2,5× lo que implica una N.A. de 0.16, los desplazamientos en el eje z corresponden a $\Delta z = 10\mu$m, mediante iluminación de campo brillante y luz reflejada. La muestra se conforma de 3 planos con diferente nivel de enfoque la figura 2.17 muestra las imágenes que componen la muestra.

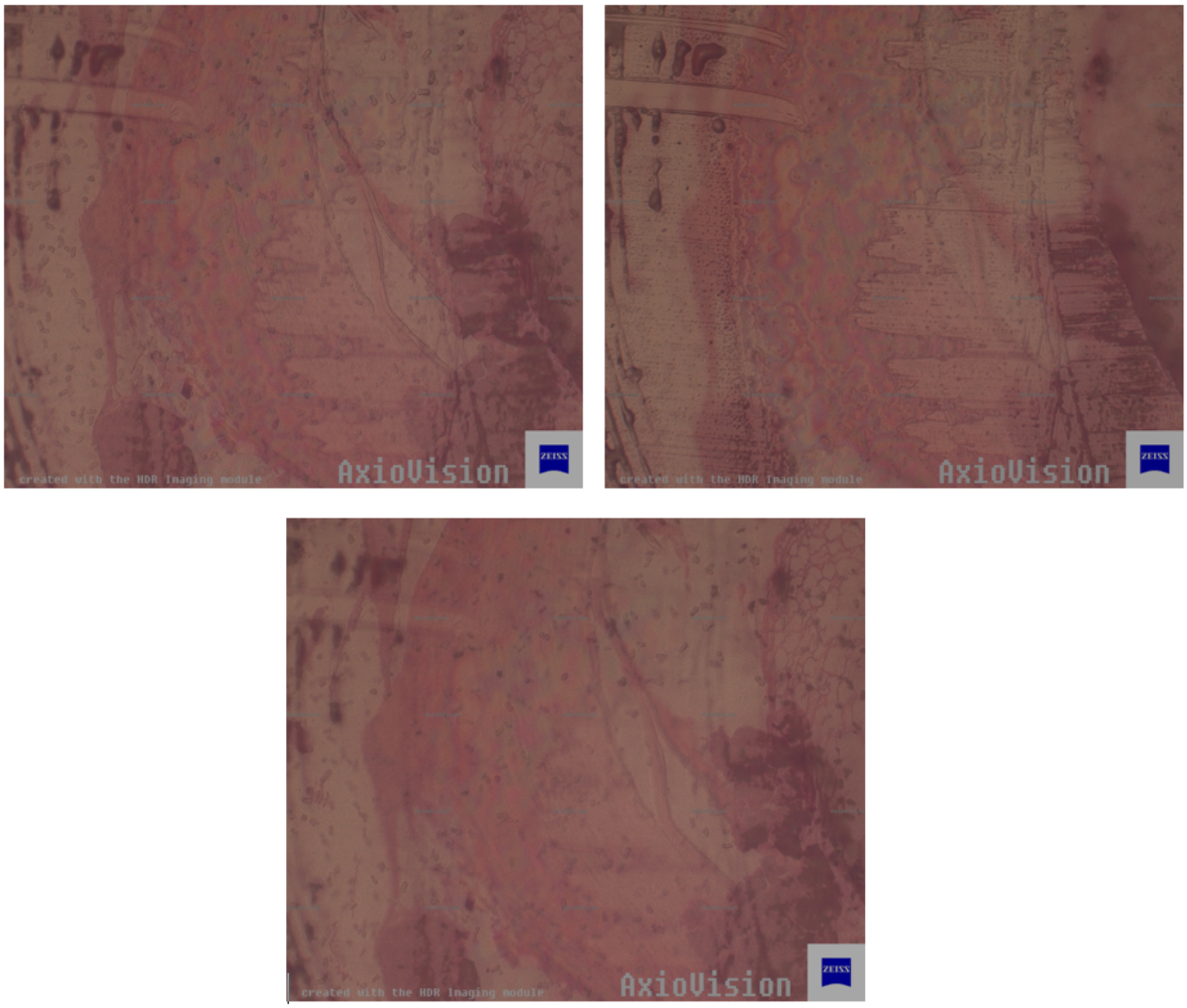


Figura 2.17: Muestra biológica de un pez, la muestra consta de 3 planos, amplificación de 2.5 ×, N.A.= 0.16.

Bibliografía

[1] E. Hecht, *Optics*, Addison-Wesley Longman, Cuarta Edición, Cap. 5, (2002).

[2] P. J. Goodhew *Electron Microscopy and Analysis*, Taylor & Francis Group, Tercera Edición, pp. 12–14, (2000).

[3] R. O. Wayne *Light and Video Microscopy*, Academic Press, Primera Edición, pp. 98, (2014).

[4] R. G. Driggers *Encyclopedia of Optical Engineering*,CRC Press, Vol. 1, pp. 557-558, (2003).

[5] F.A. Jenkisn, H.E. White, *Fundamentals of Optics*, Shirley Grall, Cuarta Edición, pp. 198–200, (2001) .

[6] P. Gray, *Hand book of basic microtechnique*, McGraw-Hill, Segunda Edición, pp. 7–10, (1956).

Capítulo 3

Medidas de contraste en imágenes digitales

3.1. Introducción

El presente capítulo tiene como objetivo primordial describir matemáticamente los algoritmos que serán empleados para la medición del grado de enfoque en las imágenes obtenidas mediante microscopio. Asimismo se describirán las adecuaciones requeridas a algunas técnicas para su correcta implementación en la GPU.

Recordemos que las imágenes reales son continuas y pueden representarse como funciones $f(x, y)$. Para poder trabajar con ellas en un ordenador es preciso realizar la digitalización de las mismas. Las imágenes digitales son obtenidas mediante un proceso de *muestreo* y *cuantización* de las señales de video adquiridas a través de sensores especializados, ya sean cámaras o algún otro tipo de dispositivo como un escáner.

El *muestreo* de una imagen consiste en la discretización de los valores de sus coordenadas que ocurre en el plano del sensor. El proceso de muestreo sobre una imagen que varía de forma espacial se realiza midiendo los valores de la señal continua cada t unidades de tiempo; a este intervalo se le denomina como *intervalo de muestreo*. Es decir, a partir de este proceso la imagen es convertida en una secuencia de números llamados muestras que se interpretan como una matriz discreta de $M \times N$. Por ende el muestreo se encuentra directamente relacionado

con la resolución espacial de una imagen.

Por ejemplo la Figura 3.1 muestra la misma escena adquirida a diferentes resoluciones 256×256, 128×128, 64×64, 32×32 pixeles. Note como varía el grado de enfoque conforme se disminuye la resolución espacial sobre la misma superficie.

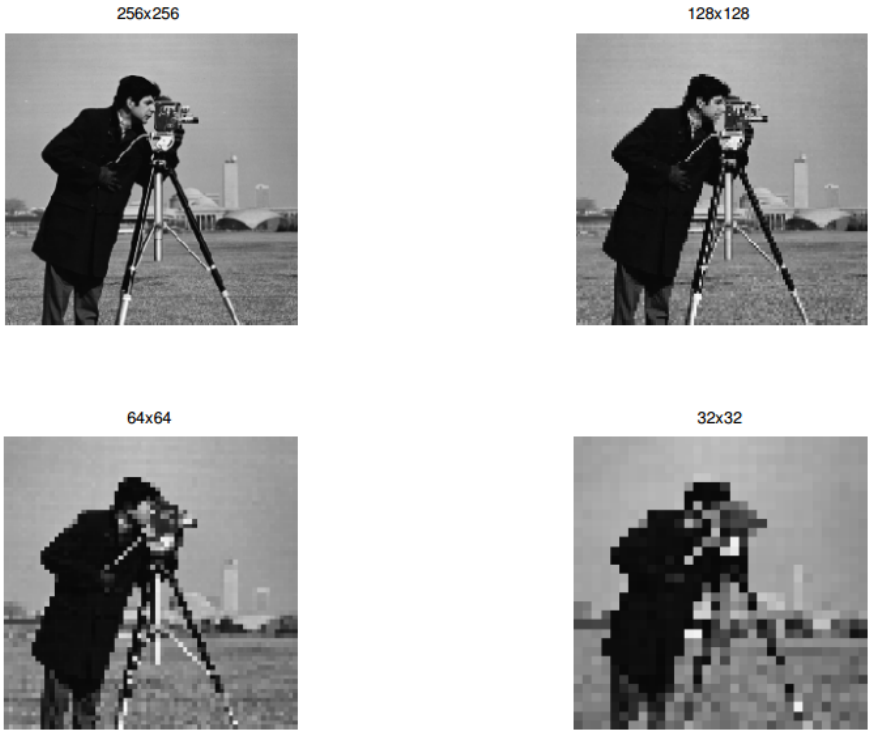


Figura 3.1: Diferentes niveles de muestreo de una escena

Una imagen puede ser continua respecto al eje de coordenadas x y y, pero también puede ser continua en amplitud. Para convertir una imagen continua a su forma digital, denominamos a la digitalización de los valores de amplitud como *cuantización* [1]. Usualmente los niveles de cuantización se definen en potencias de 2 para facilitar su almacenamiento en un equipo de cómputo. Así una imagen "estándar", utiliza un byte de cuantización (1 byte=8 bits), que es el equivalente a $2^8 = 256$, es decir, se tienen 256 niveles de gris, donde el 0 corresponde al color negro y el 255 al color blanco. La figura 3.2(a) muestra una imagen con 256 niveles de gris o con 8 bits de profundidad; mientras que el inciso (b) de la misma figura representa una imagen con 64 niveles de gris o 6 bits; la parte (c) corresponde a una cuantización en 16 niveles de

gris; y finalmente (d) corresponde a una imagen con 4 niveles de gris o 2 bits de cuantízación. Se dice que cuando una imagen solo posee niveles de grises, la cuantización es propiamente un escalar. Si dicho escalar posee solo dos niveles se habla entonces de una imagen binaria donde el cero representa al color negro y uno al blanco.

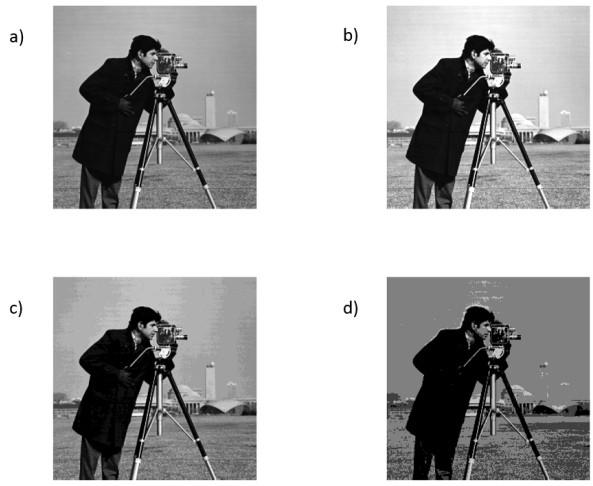


Figura 3.2: Cuantización de una imagen digital. a) Imagen con 8 bits, b) 6 bits, c) 4 bits, d) 2 bits

Para el caso del color, la cuantización se vuelve vectorial; por cada pixel se representa una terna de valores que hagan reflejar la luminancia de cada pixel. En el sistema estándar de color RGB *(Red, Green, Blue)*, se emplea un byte por cada color lo que se traduce en 256 niveles de intensidad para el *Rojo*, 256 niveles de intensidad para el *Verde* y 256 niveles de intensidad para el *Azul*, de tal forma que mediante la operación $256 \times 256 \times 256$ se obtendrán $16,777,216$ millones de colores.

Un algoritmo de enfocamiento o medida de contraste en imágenes digitales pretende medir el grado de detalle a partir de una imagen digital de tamaño $M \times N$. Recordemos que los detalles en un imagen digital se encuentran en los "bordes" que son aquellos píxeles alrededor de los cuales la imagen presenta una variación brusca en los niveles de gris y que estos bordes están representados por las altas frecuencias en el espacio de Fourier.

3.2. Algoritmos de Enfocamiento

Acorde a lo mencionado con anterioridad se puede decir que una métrica de enfocamiento debería de ser efectiva en términos de fiabilidad y precisión. La mayoría de los algoritmos de enfoque pueden catalogarse en alguno de los siguientes grupos: i) derivables, ii)basados en transformaciones, iii) estadísticos y iv) basados en histograma [2]. En este capítulo se evalúan diferentes métodos que recaen en las categorias de *derivables* y *basados en transformación*. Dichas métricas fueron elegidas puesto que proveen un bajo porcentaje de error en el enfocamiento y a su vez implican una elevada carga computacional dichas métricas han sido probadas específicamente en imágenes con especímenes de bacterias [1, 3]. Para su implementación se usará la GPU como procesador principal. Finalmente cabe mencionar que, las métricas explicadas a continuación se aplican sobre un conjunto de imágenes con diferentes focos, con el objetivo de "medir" el contraste de las mismas. Así pues, la imagen mejor enfocada corresponderá a la imagen que una vez aplicada la métrica devuelva el mayor valor numérico.

Para representar estas métricas matemáticamente, se define una imagen digital G de tamaño $M \times N$, cuyas posiciones o pixeles serán representados por $g(i, j)$, que representará la intensidad de la imagen en la posición i, j para $i = 1, ..., M$ y $j = 1, ..., N$ respectivamente; el operador $\otimes$ denota la operación de convolución.

3.2.1. Métricas Derivables

Frecuencia Espacial (FS)

Este algoritmo es ampliamente utilizado, fue introducido por [4] como una criterio para la fusión de imágenes digitales con diferentes planos de enfoque. Esta técnica mide el nivel de cambios en filas (RF) y columnas (CF) de una imagen y se describe matemáticamente por las siguientes expresiones:

$$F_{FS} = \sqrt{RF^2 + CF^2} \tag{3.1}$$

donde CF corresponde a la *Frecuencia Espacial* en las columnas y esta definido por:

$$CF = \sqrt{\frac{1}{MN}\sum_{j=1}^{N}\sum_{i=2}^{M}[g(i,j) - g(i-1,j)]^2} \tag{3.2}$$

y RF corresponde a la frecuencia espacial para las filas.

$$RF = \sqrt{\frac{1}{MN}\sum_{i=1}^{M}\sum_{j=2}^{N}[g(i,j) - g(i,j-1)]^2}. \tag{3.3}$$

Laplaciano(LAP)

Originalmente fue utilizado para encontrar errores en el enfoque ocasionados por el ruido, este algoritmo posee varias características deseables, como su sencillez, su simetría rotacional, y la eliminación de información innecesaria. El algoritmo se basa en una convolución de una máscara Laplaciana con la imagen de entrada y está definida por [5]:

$$F_{LAP} = \sum_{i=1}^{M}\sum_{j=1}^{N}[g(i-1,j) + g(i+1,j) + g(i,j-1) + g(i,j+1) - 4g(i,j)]^2 \tag{3.4}$$

Tenengrad (TEN)

Acordé con [6], esta técnica emplea un operador *Sobel* para convolucionar la imagen original con el operador S y con su transpuesta S^T el cual está definido por:

$$F_{TEN} = \sum_{i=1}^{M}\sum_{j=1}^{N}[g(i,j) \otimes S]^2 + [g(i,j) \otimes S^T]^2 \tag{3.5}$$

donde S corresponde al $Kernel$ de Sobel y S^T es su traspuesta definidas por:

$$S = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad S^T = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Vollath-4 (VOL4)

Vollath propone una métrica de autoenfoque basada en la auto-correlación [7], dicha métrica esta definida por:

$$F_{VOL4} = \sum_{i=1}^{M-1}\sum_{j=1}^{N} g(i,j)g(i+1,j) - \sum_{i=1}^{M-2}\sum_{j=1}^{N} g(i,j)g(i+2,j) \tag{3.6}$$

donde $g(i,j)$ es el nivel de gris del pixel de la imagen G localizado en las coordenadas i,j. Esta métrica es robusta ante el ruido y eficiente en tiempo de cómputo.

Filtro Gaussiano (G1,G2)

Esta métrica consiste en realizar la convolución de una imagen con una derivada Gaussiana de primer orden en las direcciones x e y [8]. La información relacionada con las altas frecuencias se mide de la forma:

$$F_{GS}(\sigma) = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N}[g(i,j) \otimes G_x(i,j,\sigma)]^2 + [g(i,j) \otimes G_y(i,j,\sigma)]^2. \tag{3.7}$$

donde G_x y G_y son la derivada Gaussiana de primer orden con una escala de σ en la dirección x y y respectivamente. El valor de σ depende en gran medida de los objetos presentes en la imagen a analizar. El ajuste en la escala de σ resulta en un incremento de la robustez frente al ruido como pueden ser partículas de polvo presentes en la superficie de preparación y artefactos ópticos. Para este trabajo se eligieron valores de $\sigma = 1$ y $\sigma = 2$ para implementar las métricas *GS1* y *GS2* respectivamente.

3.2.2. Métricas basadas en transformación

Momentos de Hu (Hu)

Los momentos de Hu son una herramienta que se utiliza usualmente como invariantes a la rotación, traslación y cambio de escala, en el contexto del reconocimiento de patrones. Recientemente fueron introducidos como un criterio para medir el grado de enfoque de una

imagen [9, 10]. En particular es posible utilizar una métrica basada en una combinación lineal de los momentos de segundo orden tal como:

$$F_{Hu} = \mu_{20} + \mu_{02} = \sum_{i}^{M}\sum_{j}^{N}(i - i_c)^2 g(i,j) + \sum_{i}^{M}\sum_{j}^{N}(j - j_c)^2 g(i,j). \tag{3.8}$$

donde $i_c = \frac{c_{10}}{c_{00}}$ y $j_c = \frac{c_{01}}{c_{00}}$ corresponden al centroide horizontal y vertical de una imagen obtenidos apartir de los momentos centrales definidos como:

$$c_{pq} = \sum_{i}^{M}\sum_{j}^{N} i^p \cdot j^p \cdot g(i,j) \tag{3.9}$$

Transformada Coseno Discreta de Frecuencias Medias (MDCT)

En la implementación original usando la Transformada Coseno Discreta de este algoritmo propuesto por [11], las imágenes de las que se desean obtener información se dividen en bloques de pixeles de tamaño $K \times K$; posteriormente la métrica de enfoque se basa en el análisis de 4 bandas diagonales de una imagen que representan las frecuencias altas y medias, una mejor descripción se puede observar en la Figura 3.3.

Los autores en [12] proponen el uso de una máscara de convolución de tamaño 4×4 para extraer el coeficiente central $c(4,4)$ de la DCT, entonces la sumatoria de dicha convolución a lo largo de toda la imagen es empleada como una métrica de enfoque, dicho operador puede ser calculado como:

$$F_{MDCT} = \sum_{i}^{M}\sum_{j}^{N}[g(i,j) \otimes K_{MDCT}]^2, \tag{3.10}$$

donde K_{MDCT} representa la máscara de convolución definida por:

$$K_{MDCT} = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & 1 & -1 & -1 \\ -1 & -1 & 1 & 1 \\ -1 & -1 & 1 & 1 \end{bmatrix}$$

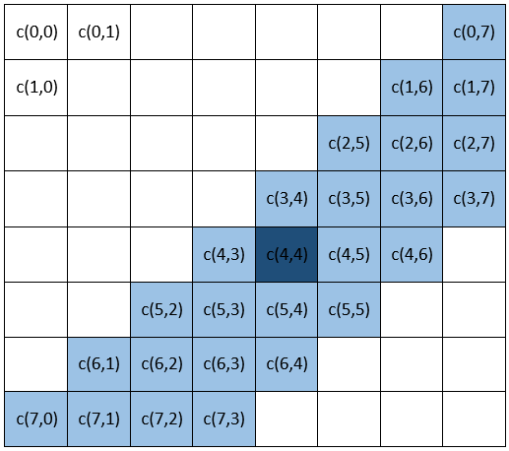


Figura 3.3: Coeficientes centrales de la transformada coseno discreta correspondientes a un bloque de tamaño 8×8. En nuestro caso se realiza el análisis de toda la imagen empleando la máscara de convolución

Distancia Euclidiana (DE)

La distancia euclidiana o euclídea es la distancia "ordinaria" entre dos puntos en un espacio euclídeo la cual se obtiene a partir del teorema de pitágoras. Acorde con [13], es posible cuantificar la diferencia entre dos colores calculando la distancia geométrica euclidiana entre ellos.

Supongamos por ejemplo dos pixeles de color cualesquiera $P_1 = (P_{1r}, P_{1g}, P_{1b})$ y $P_2 = (P_{2r}, P_{2g}, P_{2b})$, la distancia entre ellos está definida por:

$$D(P_1, P_2) = \sqrt{(P_{1r} - P_{2r})^2 + (P_{1g} - P_{1g})^2 + (P_{1b} - P_{2b})^2} \tag{3.11}$$

donde los subíndices r, g y b corresponden a los componentes del modelo de color RGB.

En base a lo anterior el autor propone medir la distancia euclidiana (DE) entre los vectores de color de una imagen en las direcciones i, j, acorde con la ecuación (3.12).

$$DE(i,j) = \sqrt{\begin{array}{c}(g_r(i,j) - g_r(i,j+1))^2 + (g_g(i,j) - g_g(i,j+1))^2 + (g_b(i,j) - g_b(i,i+1))^2 + \ldots \\ +(g_r(i,j) - g_r(i+1,j))^2 + (g_g(i,j) - g_g(i+1,j))^2 + (g_b(i,j) - g_b(i+1,i))^2\end{array}} \tag{3.12}$$

donde g_r, g_g, g_b corresponden, respectivamente, a los componentes de color de una imagen.

De acuerdo con el autor la matriz de transformación anterior se puede utilizar como métrica de autoenfoque en base a la expresión siguiente:

$$F_{DE} = \sum_{i}^{M} \sum_{j}^{N} DE(i,j). \tag{3.13}$$

Otro resultado importante es que, el uso de la matriz DE permite mejorar sustancialmente el desempeño de cualquiera de los algoritmos de autoenfoque antes mencionados, para el caso de imágenes en color.

3.3. Modificaciones a los algoritmos para su implementación en GPU

La programación paralela y en general la programación en GPU posee algunas restricciones entre las que destacan la imposibilidad de anidar ciclos *for* y el uso de condicionales anidados; de tal forma que para llevar a cabo la ejecución de algunos algoritmos mencionados en este capítulo, es necesario modificar la lógica de programación de los mismos para su correcta ejecución en GPU. Dichas modificaciones se detallan a continuación para algunas métricas.

3.3.1. Frecuencia Espacial

La frecuencia espacial esta representada por las ecuaciones (3.1), (3.2) y (3.3),en particular la ecuación (3.2) representa la frecuencia espacial para las columnas y esta dada por:

$$CF = \sqrt{\frac{1}{MN} \sum_{j=1}^{N} \sum_{i=2}^{M} [g(i,j) - g(i-1,j)]^2}. \tag{3.14}$$

es posible apreciar de la ecuación anterior que la implementación requiere de una *doble sumatoria* y de una resta pixel a pxiel, que de forma discreta recae su aplicación en ciclos *for*; para poder implementar esta operación de forma paralela, sustituimos esta doble sumatoria

por una desplazamiento en el sentido de las filas, que nos permite realizar la resta componente a componente necesaria para implementar la métrica descrita en (3.11). Para ello considere a A como una matriz de 5×5 pixeles

$$A = \begin{pmatrix} 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \end{pmatrix} \tag{3.15}$$

$$ARF = \begin{pmatrix} 23 & 5 & 7 & 14 & 16 \\ 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \\ 17 & 24 & 1 & 8 & 15 \end{pmatrix} \tag{3.16}$$

la cual se puede pensar como una imagen de tamaño 5×5; la matriz ARF corresponde a la misma desplazada en el sentido de las filas 1 posición hacia arriba. Para realizar la resta de los pixeles en las posiciones i menos el $i-1$ de acuerdo a la expresión $[g(i,j) - g(i-1,j)]^2$, únicamente es necesarios restar ambas matrices $(ARF - AR)$, tal que el resultado de dicha operación se expresa en la matriz (3.17.)

$$ARF - A = \begin{pmatrix} -6 & 19 & -6 & -6 & -1 \\ 19 & -1 & -6 & -6 & -6 \\ -6 & -6 & -6 & -1 & 19 \\ -1 & -6 & -6 & 19 & -6 \\ -6 & -6 & 24 & -6 & -6 \end{pmatrix} \tag{3.17}$$

El resto de la métrica implica aplicar potencia *al cuadrado* a cada elemento de dicha matriz, realizar la sumatoria de los elementos, un producto por el término $\frac{1}{MN}$, donde $M = N = 5$.

Finalmente se normaliza el procedimiento anterior obteniendo la raíz cuadrada del término obtenido.

De forma análoga el procedimiento explicado en líneas anteriores se aplica para obtener RF que representa la frecuencia espacial en las filas, con la diferencia de que el desplazamiento se hace en el sentido de las columnas *una posición* hacia la izquierda.

3.3.2. Vollath-4

Vollath4 requiere del uso de ciclos for o de bucles de repetición para desplazarse en las posiciones *i,j-ésimas* (ec 3.6). En particular, es necesario obtener la sumatoria del producto de los elementos *i,j* de una determinada imagen $g(i,j)$ multiplicados por los elementos *i+1*, con lo cual se resuelve la primera parte de la ecuación expresado como sigue:

$$\sum_{i=1}^{M-1}\sum_{j=1}^{N} g(i,j)\cdot g(i+i,j). \tag{3.18}$$

Con el objeto de evitar nuevamente los ciclos for se realizó un desplazamiento hacia arriba en las filas de la imagen original $g(i,j)$ y enseguida se realizó la multiplicación elemento a elemento de ambas matrices. Por ejemplo, considere la matriz A y su versión des´lazada $AR1$ que se presentan a continuación de tamaño 5×5.

$$A=\begin{bmatrix} 17 & 24 & 1 & 8 & 15\\ 23 & 5 & 7 & 14 & 16\\ 4 & 6 & 13 & 20 & 22\\ 10 & 12 & 19 & 21 & 3\\ 11 & 18 & 25 & 2 & 9 \end{bmatrix} \quad AR1=\begin{bmatrix} 23 & 5 & 7 & 14 & 16\\ 4 & 6 & 13 & 20 & 22\\ 10 & 12 & 19 & 21 & 3\\ 11 & 18 & 25 & 2 & 9\\ 17 & 24 & 1 & 8 & 15 \end{bmatrix}$$

Ambas matrices A y $AR1$ se multiplican elemento a elemento y posteriormente se obtiene la sumatoria de dicho producto, acorde con la primera parte de la ecuación (3.6). En nuestra implementación la sumatoria se realiza hasta el elemento $M-1$ para dicha métrica; obteniendo así un valor escalar.

La métrica se termina de resolver acorde a la segunda parte de la expresión la cual se

presenta a continuación:

$$\sum_{i=1}^{M-2}\sum_{j=1}^{N} g(i,j)\cdot g(i+2,j). \tag{3.19}$$

Para su implementación ahora es necesario realizar dos desplazamientos hacia arriba a la matriz A. La matriz siguiente representa la imagen A desplazada dos veces en dirección de las filas.

$$AR2 = \begin{pmatrix} 4 & 6 & 13 & 20 & 22 \\ 10 & 12 & 19 & 21 & 3 \\ 11 & 18 & 25 & 2 & 9 \\ 17 & 24 & 1 & 8 & 15 \\ 23 & 5 & 7 & 14 & 16 \end{pmatrix} \tag{3.20}$$

El procedimiento descrito continúa como esta indicado en la ecuación (3.6) mediante el *producto elemento a elemento* entre las matrices; nuevamente se obtiene la sumatoria de los elementos de dicho producto, en nuestro caso hasta el elemento $M-2$. De esta manera se obtiene un segundo valor escalar que habrá de restarse con el obtenido del procedimiento correspondiente de la ecuación (3.18) para obtener la métrica de autoenfoque.

La implementación de los desplazamientos en las métricas antes descritas permite la correcta implementación de los algoritmos en la GPU ya que debido a las restricciones impuestas en la mismas, requieren una lógica diferente para su funcionamiento.

Bibliografía

[1] R. Gonzalez, and P. Woods, *Digital Image Processing*, Addison Wesley, Segunda Edición, 2002.

[2] S. Pertuz-Arroyo, and H.R. Ibanez-Grandas, H. R., " Automated image acquisition system for optical microscope," *Ing. Desarrollo*, Vol. 22, pp. 23-37,(2007).

[3] R. Redondo, G. Bueno, J.C. Valdiviezo, R.Nava, G. Cristóbal, O. Deniz, M. García, J. Salido, M. Del Milagro, J.Vidal, B.Escalante-Ramírez, "Autofocus evaluation for brightfield microscopy pathology," *Journal of Biomedial Optics*, Vol. 17, No.3, pp. 1–9 (2012).

[4] S. Li , J.T Kwok, Y. Wang, "Combination of images with diverse focuses using the spatial frequency," *Journal of Information Fusion*, Vol. 2, No.3, pp. 169–176, (2001).

[5] M. J. Russell and T. Douglas, "Evaluation of autofocus algorithms for tuberculosis microscopy" *Proc.International Conference of the IEEE EMBS*, pp. 3489–3492, (2007).

[6] E. Krotkov, "Focusing," *International Journal of Computer Vision*, Vol. 1, pp. 223-237 (1987).

[7] D. Vollath, "The influence of the scene parameters and of noise on the behavior of automatic focusing algorithms," *Journal of Microscopy*, Vol. 151, pp. 133–146, (1988).

[8] J. Geusebroek, F. Cornelissen, A. W. N. Smeulders and H. Geerts, "Robust autofocusing in microscopy," *Cytometry*, Vol. 39, pp. 1-9 (2000).

[9] Y. Zhang, and C. Wen, "New focus measures method using moments," *Image and Vision Computing*, Vol. 18, pp. 959-965, (2000).

[10] J. Flusser, T. Suk, and B. Zitovan, *"Moments and moments invariant in pattern recognition"*, John Wiley and Sons, UK (2009).

[11] M. Subbarao, T. Choi, and A. Nikzad, "Focusing techniques", *Opt. Eng.* 32(11), 2824-2836 (1993).

[12] S. Y. Lee, Y. Kumar, J.M. Cho, S.W. Lee, and S.W. Kim, "Enhanced autofocus algorithm using robust focus measure and fuzzy reasoning," *IEEE Trans. on Circuits and Systems for Video Technology*, Vol. 18, pp. 1237–1246 (2008).

[13] R. Hurtado-Perez, "Análisis Comparativo de algoritmos de enfocamiento en imágenes a color para microscopía óptica", Tesis de Maestría, Universidad Politécnica de Tulancingo, (2013)

[14] J.M. Mateos-Pérez, R. Redondo, R. Nava, J.C. Valdiviezo, G. Cristóbal , B. Escalante-Ramírez, M.J. Ruiz-Serrano, J. Pascau, M. Desco , "Comparative evaluation of autofocus algorithms for a real-timesystem for automatic detection of Mycobacterium tuberculosis," *Journal of Cytometry Part A*, Vol. 81, No. 3, pp. 213–221 (2012).

Capítulo 4

Programación en paralelo usando CUDA

4.1. Introducción

En la actualidad el método principal para agilizar las operaciones realizadas por un computadora se centra en aumentar la velocidad de reloj del CPU *(Unidad Central de Procesamiento)* principal, permitiendo a su vez incrementar el rendimiento; esta práctica ha llevado a incrementar el número de núcleos de procesamiento disponibles en un mismo CPU, para aumentar el rendimiento del mismo, de tal forma que la CPUs actuales han llegado a tener 4, 8, 12 o hasta 16 núcleos.

Gracias al incremento de núcleos ha surgido la programación paralela, cuyo punto fuerte es el uso de varios procesadores trabajando en conjunto para dar solución a una tarea en común, dividiendo el trabajo entre el número total de núcleos de procesamiento disponibles. La programación paralela permite resolver problemas que requieren cálculos altamente complejos y que adicionalmente no se solucionan en un tiempo razonable, lo anterior tambíen permite que se puedan ejecutar programas con mayor complejidad más rápidamente [1].

4.2. Arquitectura CUDA para cómputo en paralelo

La arquitectura de dispositivos de cómputo unificado (*Computed Unified Device Arquitecture,* CUDA),es una arquitectura de cómputo paralelo que hace referencia tanto a un compilador como a un conjunto de herramientas de desarrollo creadas por la empresa NVIDIA. Permite a los programadores usar una variación de C/C++ para codificar algoritmos en una GPU de NVIDIA. Mas aún, mediante el uso de *wrappers* o clases envoltorio se puede utilizar Python, Fortran, Java, OpenGL y Direct3D para el desarrollo de aplicaciones, así como software dedicado como puede ser MATLAB o LabView.

La arquitectura CUDA está diseñada para ejecutarse sobre un Unidad de Procesamiento Gráfico (*Graphic Processing Unit.* GPU), la GPU es un coprocesador dedicado al procesamiento de gráficos y operaciones de punto flotante, para aligerar la carga de trabajo del procesador central en aplicaciones como los videojuegos y/o aplicaciones 3D interactivas. Un ejemplo de una tarjeta gráfica se muestra en la Figura: 4.1.

Figura 4.1: GPU de Sobre Mesa Tesla

CUDA tiene como objetivo aprovechar el paralelismo y la enorme capacidad de transferencia de datos que una GPU es capaz de ofrecer en operaciones con un gran costo aritmético. Ésto, gracias a la cantidad de hilos independientes de procesamiento que es capaz de ejecutar a la vez; un hilo (*thread*) puede ser visto como un programa de ejecución ya que en general un programa posee usualmente un solo hilo de ejecución.

Cuando se ejecuta un programa, el sistema operativo crea un hilo para ejecutar el programa. Todos los sistemas operativos actuales son *multihilo*; es decir pueden ejecutar simultáneamente

varios hilos lo que se traduce a la ejecución de varios programas [2].

El número de hilos en la CPU está limitado principalmente por el número de procesadores en el mismo Chip del procesador, así un procesador de 2 núcleos tendrá a su disposición dependiendo del modelo de procesador al menos 2 hilos de procesamiento puesto que existen procesadores que poseen hasta dos hilos de procesamiento por núcleo. En los procesadores más modernos se pueden encontrar equipos con hasta 16 hilos en 8 núcleos. En su lugar un GPU de arquitectura moderna como una GPU *NVIDIA K4000* puede ofrecer *768* núcleos para procesamiento, divididos en *Streaming Multiprocessor* (SM)[1] de 32 núcleos cada uno, por lo que una GPU dedica esencialmente un número mayor de transistores al procesamiento de datos. Una descripción gráfica al punto anterior se puede apreciar el la Figura: 4.2a que muestra la arquitectura de un núcleo de CPU con sus componentes básicos la Unidad Aritmética lógica (ALU); encargada de las operaciones aritméticas y lógicas del procesador, la memoria Cache del mismo; empleada para reducir el tiempo de acceso a datos ubicados en la memoria principal, la memoria dinámica de acceso aleatorio (**DRAM** o simplemente RAM); y la unidad de control (UC) encargada de controlar el flujo de datos a través del procesador.

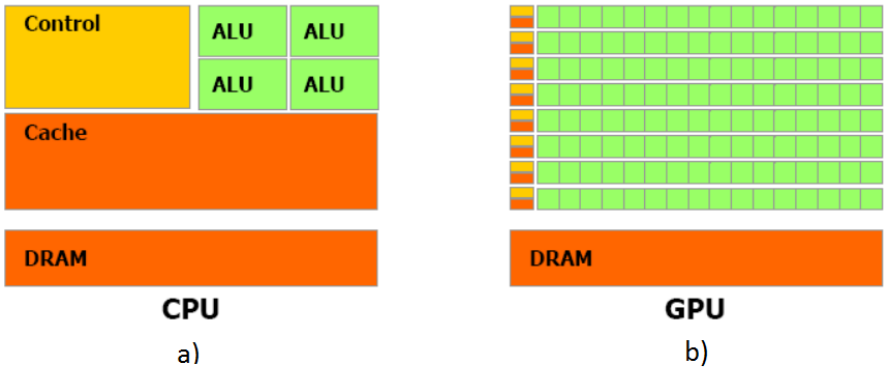


Figura 4.2: Numero de Transistores CPU vs GPU

En contraparte 4.2b se aprecia la arquitectura de un núcleo CUDA, compuesto por un numero mayor de ALU compartiendo en cada grupo de las mismas una memoria Cache, así

[1]Se le conoce con este nombre a una arreglo de núcleos de CUDA en la GPU donde cada núcleo recibe el nombre de Streaming Processor.

como una unidad de control; y de forma global comparten la memoria RAM.

Una de las ventajas principales de CUDA radica en la escalabilidad de las aplicaciones diseñadas para la misma, es posible combinar diversas GPUs en un mismo equipo sin importar su arquitectura, su número de núcleos, la cantidad de memoria, etc. En pocas palabras es posible expandir la capacidad de cómputo agregando una mayor cantidad de GPUs que pueden ser en esencia más o menos potentes y logicamente heterogéneas entre sí. Adicionalmente el uso de varias GPUs no implica algún procedimiento especial pues el aprovechamiento de las mismas es totalmente transparente por lo que es posible que el usuario trabaje con dos o más tarjetas. al mismo tiempo.

Una explicación de esto se aprecia en la Figura 4.3 donde un solo programa es dividido en dos GPU distintas cuya diferencia principal entre ellas es el número de núcleos que las componen.

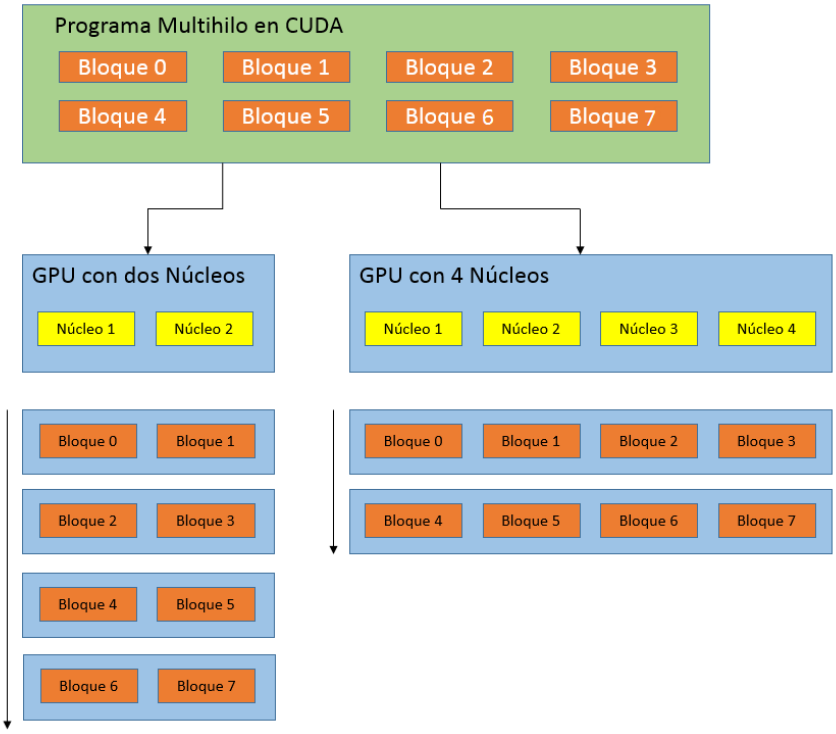


Figura 4.3: La carga de trabajo se distribuye equitativamente entre las GPUs y el número de núcleos disponibles

4.2.1. Kernel

Un *kernel* es una función que al ejecutarse lo hará de forma paralela mediante la ejecución de un determinado número de hilos [3], a su vez un kernel es lanzado directamente en la GPU. En base a lo anterior decimos que un programa o aplicación en CUDA consta de las siguientes partes:

- Un código que se ejecuta del lado de la CPU y hace de interfaz con la GPU.
- Un kernel o código que se ejecuta en paralelo y por tanto se ejecuta del lado de la GPU.

A partir de ahora a la CPU se le definirá como el anfitrión *(host)* que será el inicio de cualquier aplicación en CUDA y a la GPU la denominaremos como *device*, por lo que ser concluye que todo programa que se deseé ejecutar en CUDA consistirá en un intercambio de información entre estos dos elementos.

Cada kernel se dice que se organiza mediante una *Malla (Grid)*, la cual se puede definir propiamente como una forma de estructurar *bloques* o conjuntos de hilos en el kernel. Cada malla contiene un número finito de bloques de hilos, y a su vez cada bloque puede tener hasta 32 hilos ejecutados físicamente en paralelo. Esta explicación es posible observarla en la Figura 4.4.

La figura 4.4 demuestra que a nivel lógico una aplicación en CUDA se invoca mediante una malla, donde a su vez en cada malla hay un número finito de *bloques de hilos* ***(thread blocks)*** que serán los encargados de ejecutar físicamente un determinado número de operaciones en paralelo.

4.2.2. Operaciones Básicas en Memoria

La velocidad de ejecución de un programa se basa en la explotación de los datos y al número de procesos que se puedan realizar simultáneamente. CUDA maneja diferentes tipos de *memoria*, dentro de las cuales se almacenará la información a ser procesada por la GPU o el CPU.

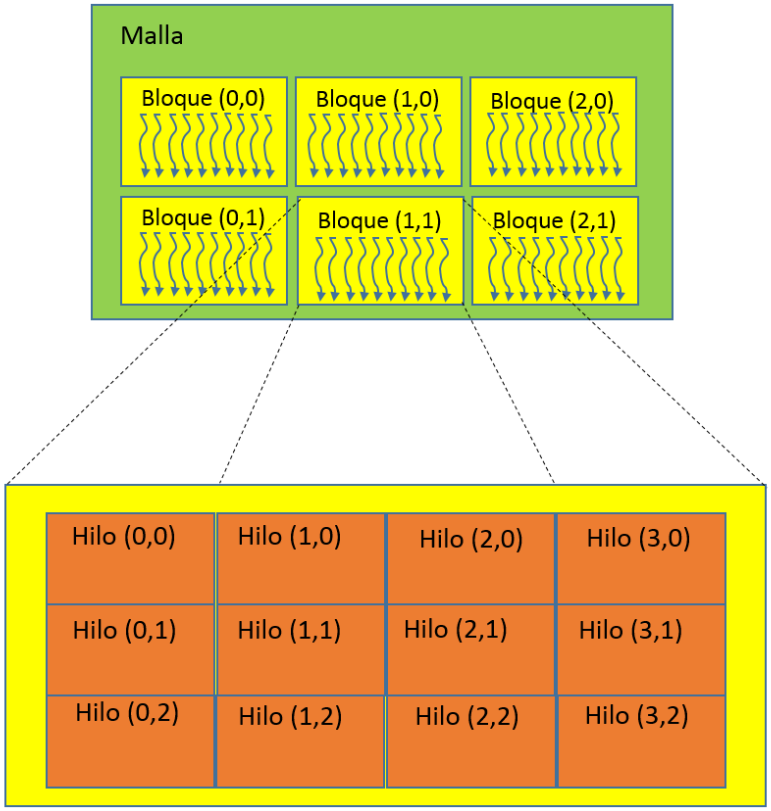


Figura 4.4: Representación Lógica de bloques e hilos en una aplicación en paralelo

Tipos de Memoria

- **Registros**: Es el tipo de memoria mas rápida, los registros son de 32 bits y es manejada por el compilador.
- **Memoria Local**: Igualmente usada por el compilador si existen muchas variables locales, presenta la desventaja de ser mucho más lenta que los registros, por tanto, debe usarse en raras ocasiones.Así mismo no es posible impedir su uso; si se aprecia una fuerte degradación en el rendimiento se debería rediseñar el código.
- **Memoria Global**: Es la memoria donde se almacenan los datos transferidos desde el host por lo que se considera la memoria más lenta de la GPU, para utilizarla se puede llamar con el indicador o palabra reservada __global__.

- **Memoria Compartida**: Es rápida y se recomienda su uso para minimizar el acceso a la memoria global, se usa principalmente para intercambiar datos entre los hilos de un mismo bloque. Esté tipo de memoria difiere de la memoria global se identifica mediante la palabra reservada __shared__.
- **Memoria constante**: Es rápida y solo se puede escribir en ella desde el host, desde el lado del device solo se puede leer, para su manipulación se emplea un identificador diferente al de la memoria global y compartida, se utiliza el identificador __constant__. También se pueden guardar arreglos pero no se pueden reasignar.

Las operaciones en memoria tienen un tipo especial de métodos para su manipulación puesto que la información a procesar siempre se almacenará en algún tipo de memoria (usualmente memoria global) y, basados en el hecho de que tanto la GPU como la CPU mantienen su propio espacio de memoria, es necesario hacer intercambios entre la memoria alojada en el host y en el device; entre las operaciones que podemos realizar se especifican las siguientes:

1. Copiar Memoria entre CPU y GPU
   a) Copiar Memoria de CPU a GPU (HostToDevice)
   b) Copiar Memoria de GPU a CPU (DeviceToHost)
   c) Copiar Memoria entre Diferentes GPU (DeviceTo Device)
2. Reservar Memoria para su uso posterior (CUDAMalloc) (Similar a Memory Allocate de C)
3. Liberar Memoria(CUDAFree)

Reservar Memoria cudaMalloc

Para poder realizar operaciones de memoria entre el CPU y la GPU es necesario solicitar a la Tarjeta Gráfica la memoria que será utilizada. Para ello se emplea la instrucción: *Cuda Memory Allocate (cudaMalloc)*, que será la encargada de garantizar que exista la memoria suficiente para realizar un copiado de memoria del host hacia el device.

Su sintaxis es la siguiente:

```
cudaMalloc(void**)&dev_a,N*sizeof(int));
```

donde, el primer argumento *(dev_a)* es un puntero que va hacia el espacio nuevo de memoria que se alojara en la GPU , y el segundo parámetro es el tamaño de la memoria que se reservara *(N*sizeof(int))*. Sin embargo aun falta especificar la variable de retorno de la función, si nos basamos en la función malloc propia de C/C++ es completamente idéntica. Por tanto añadimos un tipo de retorno de la función es este caso *(void**)*.

Copiar Memoria entre Host y Device (cudaMemcpy)

La instrucción CUDA Memory CopycudaMemcpy se utiliza para transportar o mover la información que se desea procesar entre la memoria de la CPU y la GPU, entre GPUs e incluso entre diferentes espacios de memoria, las anteriores operaciones además de ser la base de un programa en GPU permitirán intercambiar información entre GPUs en caso de poseer mas de una. La sintaxis para la instrucción cudaMemcpy es la siguiente:

```
cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
```

Esta instrucción es bastante similar a Memcpy en C/C++. El primer parámetro es el destino al que se copiará la memoria, el segundo es el origen de esta memoria, el tercer parámetro es el tamaño de la misma y por último un parámetro adicional propio de CUDA que indica el destino de la memoria; dicho parámetro puede tomar uno de los siguientes valores:

1. cudaMemcpyHostToDevice: de la CPU a la GPU
2. cudaMemcpyDeviceToHost: de la GPU a la CPU
3. cudaMemcpyDeviceToDevice: entre GPUs en caso de existir más de una en un sólo equipo o un *Cluster* de las mismas.

Liberar Memoria (cudaFree)

De forma similar a cuando se trabaja con espacios de memoria en C/C++ en cada ocasión que se reserve un bloque de memoria, el mismo debe de liberarse una vez que su uso llegue a su

fin de tal forma que se garantice memoria para futuras operaciones. La sintaxis que presenta es la siguiente:

```
cudaFree(dev_a),
```

cuyo único parámetro es el nombre del puntero de la memoria que se va a liberar, cabe mencionar que al terminar la ejecución de nuestro programa la memoria utilizada dentro de la GPU es liberada.

4.2.3. Llamada a un Kernel

Un Kernel es esencialmente una función diseñada para ser ejecutada sobre una GPU y puesto que CUDA es una extensión al lenguaje C se pueden definir dos tipos esenciales de variables: el calificador función tipo *type function* que se utiliza para especificar donde se ejecutará la aplicación, bien sea en el *host* o en el de *device* y el calificador de tipo de variable que se emplea para especificar la localidad de memoria en el dispositivo.

En el momento de ejecutarse un kernel es necesario especificar para cualquier llamada a una función el calificador función tipo *__global__*, la dimensión y el tamaño de la malla y de los bloques. Para ello es necesario definir la función de la siguiente forma:

```
__global__ void Mi_Funcion(float*parametro);
```

Para llamar la función Mi_Función se empleara la siguiente instrucción:

```
Mi_función<<<gd,bd,nb>>>(parámetro);
```

Donde **gd** es una variable que se encarga de especificar la dimensión y tamaño de la malla (numero de bloques); **bd** es una variable que se utiliza para especificar el número de hilos de cada bloque y finalmente **nb**, es un argumento opcional que especifica el número de bytes en la memoria compartida, su valor por default es 0. Una ejecución típica de un kernel sería entonces mediante la siguiente instrucción donde a **gd** se le asigna un valor de 10 y, a la variable **bd** se le asigna un valor de 20 se obtiene un kernel corriendo en paralelo con 10 bloques donde cada bloque ejecutaría 20 hilos.

```
Mi_función<<<20,10>>>(parametro);
```

Ahora es necesario definir el comportamiento de una aplicación en CUDA, básicamente toda aplicación en CUDA implica los siguientes pasos:

1. Inicio del Algoritmo: todo algoritmo deberá iniciar con una declaración de variables, todo esto se realiza del lado del Host.

2. Copiado de Memoria CPU a GPU: una vez que se tiene la información a procesar es necesario copiar la memoria del lado del host al device.

3. Ejecución del kernel del GPU: realizamos una llamada a la función Kernel.

4. Procesamiento del algoritmo: ejecución de las instrucciones que el algoritmo debe llevar a cabo.

5. Copiado de resultados de la device al host: puesto que CUDA carece de métodos o funciones para almacenamiento o muestreo de resultados, una vez que el algoritmo finalice copiamos los resultados obtenidos de la memoria del device hacia el host.

6. Muestreo o Almacenamiento de los resultados: ahora del lado del host mostramos o almacenamos los resultados obtenidos del algoritmo ejecutado en la GPU

Todos los pasos anteriores se pueden apreciar en la Figura: 4.5

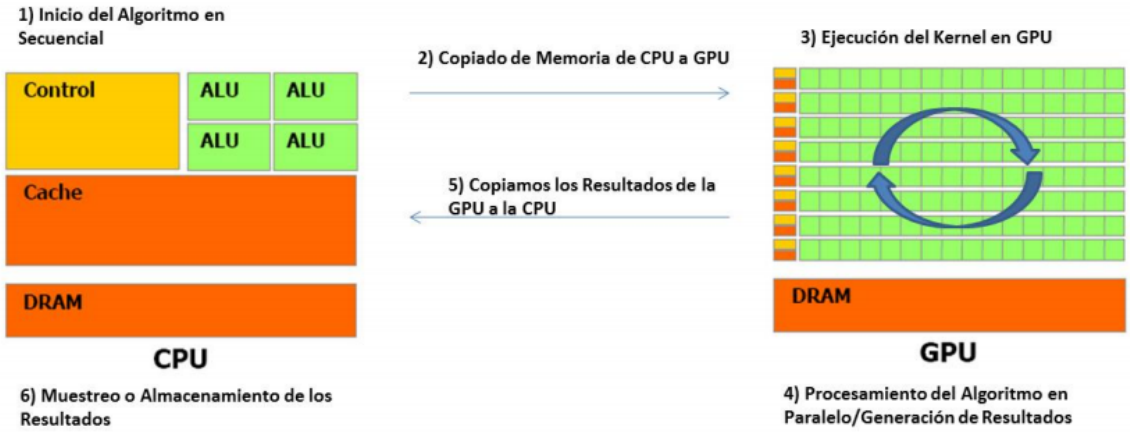


Figura 4.5: Comportamiento de un Programa en CUDA

En base a lo anterior es posible concluir que, una aplicación en CUDA no se ejecuta por completo sobre el device, la manipulación de memoria es necesaria para el manejo (copiado), de los datos almacenados en memoria, además al usarse un lenguaje de bajo nivel como C hay que ser cuidadosos al momento de copiar o transferir la memoria.

4.2.4. Suma de vectores en GPU

A lo largo de este capítulo se han discutido los fundamentos de programación empleando CUDA con el objetivo de optimizar los tiempos de procesos brindados por una CPU; para comprobar la ventaja que presenta CUDA gracias a su parecido con C se revisara un ejemplo común en programación que requiere el uso de ciclos repetitivos *mientras (while)* para su correcta solución.

Básicamente para resolver la suma de 2 vectores **a** y **b** con tamaño de 0 hasta $N-1$ se requiere para su realización en la CPU de un ciclo que le indique al procesador que debe realizar la siguiente instrucción; aprovechando la creación de funciones, suponga entonces la función siguiente a la que llamaremos *add*.

Algoritmo I. Función *add* en CPU que realiza la suma de dos vectores **a**+**b**

```
void add( int *a, int *b, int *c ) {
        int i = 0;
        // Necesitamos un ciclo para recorrer cada
    //posicion lo inicializamos en 0
        while (i < N) {
            c[i] = a[i] + b[i];
            i += 1;
      //Incrementos de 1 para recorrer cada posición
        }
}
```

El incremento en el ciclo *while* de 0 hasta $N-1$, donde $N = 10$) es el encargado de recorrer cada posición del vector **a** y **b** para colocar el resultado de la suma en un tercer arreglo **c**. Los

incrementos unitarios en i son requeridos pues el modelo de programación es secuencial.

A continuación se analizarán las partes esenciales de este mismo algoritmo pero ahora para su implementación en GPU. A diferencia de C u otros lenguajes de programación de bajo nivel donde no es obligatorio (aunque es una práctica recomendada) realizar una reserva de memoria, en CUDA será necesario reservar la memoria donde se alojarán los arreglos para su manipulación en la GPU. Dicho sea el siguiente código ilustra la reserva de dicha memoria mediante la instrucción *cudaMalloc* donde se reserva la memoria para los arreglos de entrada *dev_a*, *dev_b* y el arreglo *dev_c* donde se almacenará el resultado de la suma de vectores.

```
int a[N], b[N], c[N];
int *dev_a, *dev_b, *dev_c;

// Reservamos memoria en la GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_b, N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_c, N * sizeof(int) ) ;
```

Los lenguajes de alto nivel como JAVA o MATLAB no requieren de este tipo de instrucciones ya que el manejo de la memoria es transparente para el usuario. Por otra parte el "llenado" o inicialización de los vectores *dev_a*, *dev_b* se realiza inicializando los vectores *a[]* y *b[]* del lado de la CPU. Como se menciono anteriormente, ahora estos dos vectores se copiaran a la memoria de la GPU mediante la instrucción *cudaMemcpy* y quedarán almacenados tanto en *dev_a* como en *dev_b*.

Algoritmo II. Función *add* en GPU encargada de realizar la suma de vectores.[2]

```
// Copiamos los arreglo a y b de la CPU a la GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),
                                cudaMemcpyHostToDevice ) ;
    cudaMemcpy( dev_b, b, N * sizeof(int),
```

[2]Los códigos completos tanto del algoritmo en CPU como en GPU se anexan en el apéndice A

```
                                    cudaMemcpyHostToDevice ) ;
```

Para proceder a la ejecución de dicho algoritmo, se diseño también una función *add*, dicha función al igual que su contraparte en C está definida por un identificador void, pero también por la palabra reservada **__global__**, que indica que dicha función debe ser ejecutada en GPU.

```
__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;

    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
```

Como se observa en el algoritmo II la función *add* no emplea un cilo *while* puesto que su ejecuci´n se realiza en paralelo, está función también es el kernel del programa que será ejecutado en paralelo. Supongamos entonces que la dimensión de los vectores a sumar es de $N = 10$, para ejecutar el kernel se manda llamar con la siguiente instrucción.

```
add<<<N,1>>>( dev_a, dev_b, dev_c );
```

Si sustituimos la N se obtienen 10 bloques con un hilo corriendo en paralelo lo que garantiza que al menos un hilo será el encargado de procesar cada elemento de los vectores.

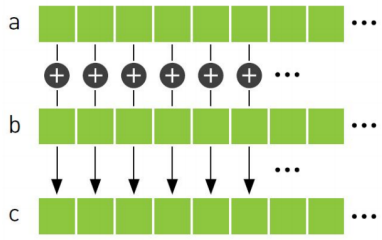


Figura 4.6: comportamiento lógico de una suma de vectores en paralelo, cada elemento es sumado por un hilo independiente

Finalmente se hace hincapié en que la función *add* del Algoritmo II carece completamente de una estructura repetitiva *while*, ésto es porque que todas las operaciones son ejecutadas simultáneamente en paralelo por un hilo independiente, evitando de tal forma un ciclo incremental para resolver la siguiente iteración, una descripción gráfica a está operación puede ser vista en la figura 4.6.

Es importante mencionar que en el número máximo de hilos corriendo en paralelo en una GPU depende del número de unidades de procesamiento físicos del dispositivo, suponiendo que una GPU posea 4 procesadores y cada procesador es capaz de correr 768 núcleos cada uno en un determinado momento el número máximo de hilos corriendo en paralelo será de 4×768, si se organiza un número mayor de hilos de forma lógica estos deberán espera su turno para ser ejecutados una práctica muy recomendada es sincronizar el número de hilos en ejecución de tal suerte que de garantiza que acabe un conjunto de hilos antes de ejecutar un segundo.

4.3. Programación en GPU Usando Matlab

MATLAB es un lenguaje de alto nivel que proporciona un entorno interactivo especial para cómputo numérico. A su vez es ampliamente utilizado en un gran número de aplicaciones que incluyen desde el procesamiento de señales y comunicaciones, procesamiento de imágenes y video, sistemas de control, análisis y medición de cómputo financiero, y biología por computadora.

El presente trabajo se realizó teniendo en cuenta el lenguaje de alto nivel de MATLAB en su versión 2013 A para el desarrollo y programación de algoritmos, a continuación se detallarán las ventajas y desventajas de su uso.

4.3.1. Ventajas y Desventajas del uso de Matlab

Facilidad de Uso

MATLAB posse un lenguaje bastante flexible que brinda un relativa facilidad de uso. MATLAB es un lenguaje basado en *Scripts* de muy alto nivel, esto implica que no hay necesidad como tal de añadir la inclusión de librerías, la declaración de variables, el mantenimiento/administración

de memoria o otras cuestiones de programación de bajo nivel. Esto implica entre otras cosas que no es necesario reservar la memoria que se va utilizar como en el caso de usar CUDA con C, así pues en MATLAB no existe una instrucción similar a *cudaMalloc* que como se mencionó anteriormente es la encargada de reservar la memoria sobre la que se va a trabajar, está práctica es muy común en lenguajes de bajo nivel.

El código siguiente representa las instrucciones necesarias para copiar memoria a la GPU usando CUDA; como se puede observar el nivel de complejidad usando un lenguaje de bajo nivel es bastante alto, el número de parámetros necesarios, así como los diversos tipos de datos exigen un alto nivel de programación, por otra parte el uso de código fuente en MATLAB presenta una mayor simpleza, lo que reduce drásticamente el porcentaje de errores.

```
//Proceso de Reserva de Memoria en GPU
cudaMalloc((void**)&dev_a,N*sizeof(int));
//Copiado de Memoria en la GPU
cudaMemcpy(dev_a,a,N*sizeof(int),cudaMemcpyHostToDevice);
```

En MATLAB la información almacenada en la GPU se denomina *gpuArray*, literalmente Arreglo de GPU, acorde con la descripción de MATLAB, este método corresponde al nombre de un función para realizar el copiado de memoria en la GPU, es decir realizar la transferencia de datos de CPU a GPU. El código siguiente ilustra el almacenamiento de un vector **a**, a la GPU, dicho vector se almacena en la variable **a_gpu**.

```
//Inicializamos el Arreglo
a=1:10;
//Copiamos la variable a de la CPU a la GPU
a_gpu=gpuArray(a);
```

Comparando el código en CUDA c con el de MATLAB, destaca claramente la facilidad de uso de MATLAB, además al ser un lenguaje de tipo *dinámico* no es necesario definir el tipo de dato de *a* ni de *a_GPU*, de igual forma no es necesario reservar la memoria en la GPU, mejorando entre otras cosas la declaración de variables y la escritura de código.

Como se analizó en la sección anterior, también es necesario copiar la información ya procesada de regreso a la memoria de la CPU, para está acción existe la instrucción reservada **gather**, que transfiere la memoria almacenada en GPU de regreso en el espacio de trabajo de MATLAB (CPU). El código siguiente en MATLAB regresa el vector **a_gpu** de regreso a la CPU.

```
//Copiamos la variable a de la CPU a la GPU
a_gpu=gpuArray(a);
//La instrucción gather regresa la memoria de GPU a CPU
a_return=gather(a_gpu);
```

Por ejemplo si se deseará sumar dos vectores **a** y **b**, solo se requiere el código siguiente para todo el proceso.

```
clc
clear all
close all
//Inicialización de los Vectores
a=1:10;
b=a.*a;
//Copiado de Memoria CPU a GPU
a_gpu=gpuArray(a);
b_gpu=gpuArray(b);
//Suma de Vectores
c_gpu=a_gpu+b_gpu;
//Copiado de Memoria GPU a CPU
c_cpu=gather(c_gpu);
```

Donde las variables **a_gpu** y **b_gpu**, tienen almacenadas en la memoria de la GPU los vectores originales **a** y **b** respectivamente, posteriormente el resultado de la suma se almacena en la variable c_gpu, y el valor almacenado se retorna a la CPU mediante la instrucción gather, para almacenarse en la variable **c_cpu**. Es claro que la diferencia en cuánto a codificación

entre un lenguaje de alto nivel y un lenguaje de bajo nivel es bastante, además MATLAB es un software dedicado, y hecho especialmente para el procesamiento de señales.

Velocidad y Recursos

MATLAB es un lenguaje que tiene sus bases en el lenguaje de programación JAVA y al ser un lenguaje de alto nivel no es necesario preocuparse por detalles concernientes a lenguajes de bajo nivel como pueden ser la administración y manejo de memoria; sin embargo, puesto que JAVA es un lenguaje interpretado, MATLAB es de igual forma un lenguaje que requiere de un interprete no así el lenguaje *C* que es más cercano al lenguaje máquina.

Al hablar de recursos nos referimos inmediatamente a la parte física (hardware) del equipo de cómputo, detalles como la memoria y el procesador son factores a considerar en la velocidad de ejecución de los algoritmos. Nuevamente la naturaleza de alto nivel de MATLAB exige una mayor cantidad de recursos para ejecutar un programa. A raíz de las definiciones anteriores es posible concluir que un programa escrito en lenguaje *C* será más rápido que un escrito en MATLAB. Sin embargo, MATLAB ha ido mejorando su programación en los últimos años además de contar con un multiprocesamiento natural al estar basado en JAVA.

Administración de Memoria

MATLAB presenta una notoria ventaja por encima de lenguajes de bajo nivel como *C*, en los lenguajes de bajo nivel es necesario en repetidas ocasiones *reservar zonas* de memoria y posteriormente es necesario liberarla después de su utilización. Por ejemplo, en caso de tener un ciclo *for* donde se reserva memoria una y otra vez sin liberar dicha memoria ocurrirá una fuga de memoria. Esto se origina en un crecimiento de la memoria en uso, provocando que el programa colapse al no tener más memoria libre para trabajar. Puesto que MATLAB es un lenguaje de alto nivel posee la capacidad para liberar automáticamente memoria en segundo plano reduciendo errores de memoria, muy comunes en el lenguaje *C*.

Depuración

Los estándares comunes de depuración pueden ser aplicados tanto en MATLAB como en C *(breakpoints, depuración por pasos)*. Sin embargo, en MATLAB existe la posibilidad de detener el programa y añadir líneas extras al código sin necesidad de compilar otra vez el código *(on the fly)*. De lo anterior, se concluye que MATLAB proporciona una herramienta poderosa para depuración de código.

Con el objetivo de realizar un análisis cuantitativo de los efectos sobre los tiempos de cómputo, se llevó a cabo una prueba donde se evalúan algunas de las métricas de autoenfoque que se discutieron en el capítulo 3, estás pruebas pretenden medir los tiempos de cómputo de una algoritmo implementado tanto en C/C++ como en MATLAB.

4.4. Comparativa de tiempos de Cómputo entre C/C++ y MATLAB

Para definir un lenguaje así como un entorno de programación para este trabajo de tesis se realizó una comparativa de tiempos de cómputo entre MATLAB y C++ puesto que como se discutió a lo largo de este capítulo ambos lenguajes pueden ser utilizados para codificar algoritmos en una GPU de NVIDIA. Para las pruebas se empleo una computadora con las siguientes especificaciones: CPU Xeon E3-1230 V3 @ 3.30 Ghz con 4 núcleos físicos y un máximo de 8 threads, 16 Gb de Memoria RAM DDR3 a 798.1 MHz, GPU Nvidia Quadro K400, 4 Multiprocesadores con 768 núcleos CUDA y 3 Gb GDDR5 . En la parte del software se emplea MATLAB 2013a en 64 bits, el toolkit de CUDA corresponde a la versión 5.0 y la versión de Microsoft Visual Studio 2010 para el uso de Visual C++ igualmente en 64 bits. Finalmente ambos IDE se ejecutaron sobre Microsoft Windows 8.

4.4.1. Frecuencia Espacial

La Frecuencia Espacial se define por la ecuación (3.3), que fue implementada bajo el lenguaje *C++* para su funcionamiento en GPU de la misma forma en que se explica en el

presente capítulo. Puesto que *C++* no es un lenguaje de alto nivel, carece de métodos que permitan la lectura y muestreo de imágenes. Para poder ejecutar dichas métricas en GPU fue necesario el uso de la librería abierta de visión por computadora (**OpenCV**) que posee en su arquitectura los métodos requeridos para leer, guardar y mostrar imágenes, entre otras funciones de alto nivel especiales para visión por computadora [4].

En comparación, una de las características mas destacables de MATLAB es la simpleza al momento de codificar y la creación dinámica de variables que vuelven innecesario la declaración e inicialización previa de una variable como sucede en otros lenguajes.

El código fuente en C++ que se presenta en el Algoritmo III necesita inicializar la variable **IMG** como un tipo de dato primitivo para almacenar la imagen *"lena.bmp"* y la instrucción posterior convierte todos los valores en **IMG** a valores numéricos con precisión doble.

Algoritmo III. Lectura de una imagen usando C++ y OpenCV

```
#include "stdafx.h"
#include <iostream>
#include "opencv2/opencv.hpp"
#include "opencv2/gpu/gpu.hpp"

using namespace std;
using namespace cv;

int main()
{

        cv::Mat IMG;
        IMG= cv::imread("lena.bmp",0);//Lectura en escala de grises
        IMG.convertTo(IMG, CV_64FC1);
        return 0;
}
```

Las librerías *opencv2/opencv.hpp* y *opencv2/gpu/gpu.hpp* corresponden a archivos de encabezado de OpenCV(Open Computer Vision), se trata de una librería que ofrece los métodos para manipulación de imágenes, dicha librería está escrita en C++ y se libera bajo la licencia *BSD* de software libre permitiendo su uso en aplicaciones comerciales.

Para poder interactuar con las imágenes dentro de cualquiera de los dos lenguajes empleados, y poder manipular las mismas dentro de la GPU, es necesario un tipo especial de arreglo propio de cada lenguaje. En MATLAB encontramos el tipo de dato *gpuArray* y en C++ se tiene el tipo de dato *gpu::Mat* (haciendo uso de la librería OpenCV), ambos con la característica de que para su uso es necesario copiar o subir la información que se va procesar, dicha operación se realiza en C/C++ mediante el código siguiente.

```
cv::gpu::gpuMat IMG_GPU;
IMG_GPU.upload(IMG);
```

Obviando los archivos de inclusión y la función *main()*, la parte destacable del código anterior radica en la instrucción **upload** equivalente como la instrucción de copiado de memoria a la GPU discutida en la sección 4.4.2. Así que, a partir de la ejecución de dicha instrucción la información contenida en IMG_GPU es tratada como un arreglo de tipo GPU.

En el caso de MATLAB, la instrucción *gpuArray* explicada en el presente capítulo produce los mismo resultados.

La falta de inicialización de la variable IMG_GPU facilita la creación de objetos de forma dinámica; no así su contraparte en C/C++ que requiere que los elementos estén declarados e inicializados.

La implementación de la métrica de Frecuencia Espacial se realizó en C/C++ de la misma forma que fue descrito en la sección **3.3.1**, permitiendo así realizar una comparativa en los tiempo de cómputo. Para las pruebas mencionadas se empleo un stack de 10 imágenes adquiridas por microscopio, dichas imágenes presentan una resolución de 2584×1936 pixeles. Una muestra de dichas imágenes se aprecia en la figura 4.7.

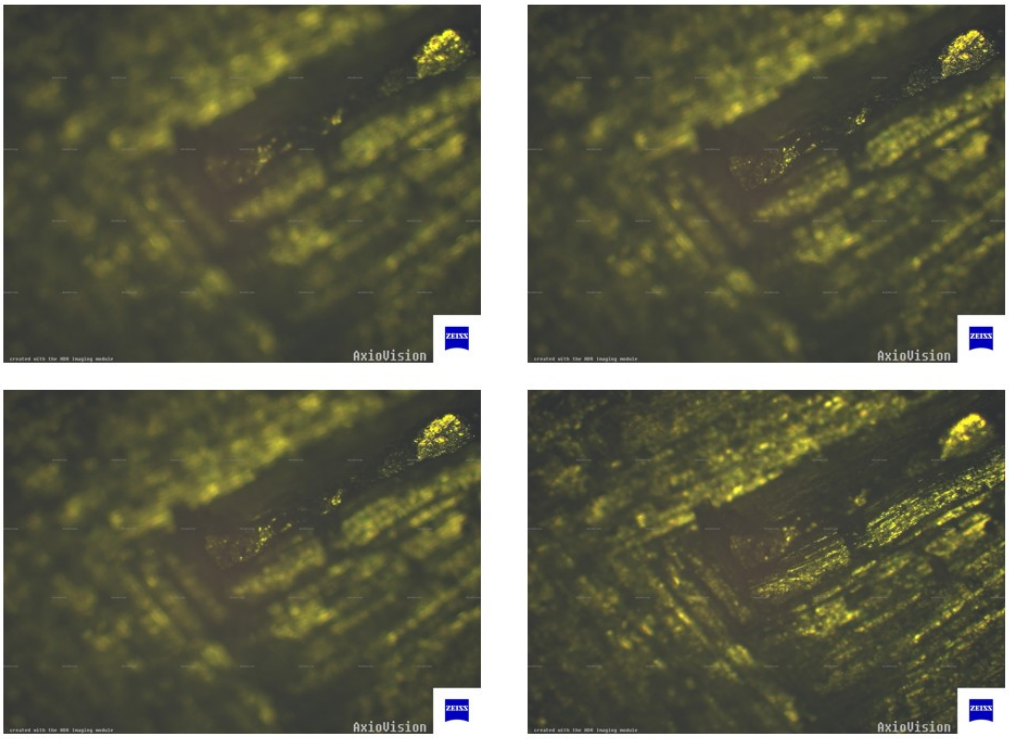


Figura 4.7: Imágenes Obtenidas por Microscopio con un aumento de 2.5x

El objetivo final de este experimento es medir los tiempos de cómputo para determinar el lenguaje de programación para la implementación de los algoritmos. El algoritmo fue implementado bajo la misma lógica siguiendo fielmente cada paso en ambos casos resultando los tiempo de cómputo en MATLAB de 1,9153 segundos y en el caso de C/C++ de 5.27 segundos, si bien es cierto que un lenguaje compilado como C++ es computacionalmente más eficiente en la mayoría de los procesos y operaciones, hay que considerar que MATLAB está altamente optimizado para operaciones de forma matricial, este lenguaje resulta idóneo para la manipulación y procesamiento de imágenes digitales así como de algoritmos programados para GPU. en este caso y justificando en base a los puntos anteriores el uso de MATLAB es hasta 2.8× más eficiente en tiempos de cómputo.

Recordemos que la imagen mejor contrastada o enfocada se localiza en el **máximo** valor numérico de la métrica que se implementó. El registro de este experimento se aprecia en la Tabla 4.1,dicha tabla muestra las diferencias numéricas entre los valores obtenidos en la imple-

mentación en C/C++ y MATLAB respectivamante. Dichas diferencias se deben al hecho de que para el manejo de imágenes es necesario manipular los valores numéricos de una imagen con una precisión doble; acorde al estándar de la IEEE una representación doble implica una representación en 64 bits contraria a lo que implica una representación simple que involucra únicamente 32 bits para su representación, básicamente la representación doble permite almacenar un rango mayor de números positivos y negativos.

Cada lenguaje maneja su propia versión del estándar para la representación de números dobles, por lo que existe una sensible diferencia numérica entre el tipo de dato **double** que maneja MATLAB y el tipo de dato **CV_64FC1** o de precisión doble que maneja OpenCV.

Tabla 4.1: Valores de Contraste para la pila de imágenes de la figura 4.7 usando FS

Imagen	FS usando MATLAB	FS usando C/C++ y Opencv	Error Absoluto
1	13.9554	13.8908	0.0645
2	14.0090	13.9601	0.0489
3	14.0465	13.9923	0.0541
4	14.1670	14.1099	0.0571
5	14.4257	14.3750	0.0507
6	14.4900	14.4483	0.0416
7	14.4814	14.4251	0.0563
8	14.3546	14.3124	0.0422
9	14.5220	14.4716	0.0504
10	**14.9467**	**14.8824**	**0.0642**

Pese a las diferencias numéricas ilustradas en la tabla 4.1 ambas implementaciones concluyen que la imagen mejor enfocada en base a la frecuencia espacial radica en la **imagen** 10 al haber obtenido el mayor valor numérico en ambas implementaciones.

4.4.2. Vollath 4

Otra métrica que se probó cuantitativamente en este trabajo es Vollath 4 que esta definida por la ecuación (3.7). Se implemento en C++ usando OpenCV bajo la misma lógica discutida en la sección 3.3.1.

Los resultados obtenidos son los siguientes. El tiempo de cómputo consumido por la versión de MATLAB requiere de 0,259933 segundos; mientras que su contraparte en OpenCV requiere de 5,17 segundos. Un tiempo de computó muy por arriba de la implementación en MATLAB que resulto ser hasta 20× más rápida. Es importante mencionar nuevamente que parte de la aceleración de este algoritmo se debe a que MATLAB está optimizado para operaciones con matrices, caso contrario al lenguaje C++.

Los valores numéricos de VOL4 se muestran en la Tabla 4.2

Tabla 4.2: Valores de contraste para la pila de imágenes de la figura 4.7, usando VOL4

Imagen	FS usando MATLAB	FS usando C/C++ y Opencv	Error Absoluto
1	104352325	104352634	309
2	107694874	107694909	35
3	117044435	117044574	139
4	132230604	132230685	81
5	**142117320**	**142117380**	**60**
6	136843610	136843826	216
7	126325927	126326011	84
8	118764373	118764298	75
9	115238060	115238177	117
10	121871061	121871300	239

La Tabla anterior muestras el error absoluto entre ambas implementaciones causado por la diferencia en el tipo de dato de cada lenguaje. Sin embargo, tanto la implementación en C como la de MATLAB coinciden en que la imagen mejor enfocada esta en la imagen que se encuentra en la posición 5 del stack de imágenes.

De las pruebas anteriores es posible concluir que:

- Puesto que C++ no esta diseñado de forma nativo para el procesamiento de matrices, los tiempos de cómputo al menos en el caso de manejo de imágenes digitales los tiempos de cómputo en MATLAB resultaron muy bajos en comparación al lenguajes C++.
- MATLAB es un entorno dedicado al cómputo científico por lo que sus rutinas o "métodos" se encuentran mejor diseñados para su ejecución, hablando en tiempo de cómputo.

Finalmente en base a estas pruebas elegimos MATLAB como nuestro entorno de desarrollo ya que presenta varias ventajas por encima de C/C++ entre las que destacan:

- La administración de memoria
- La posibilidad de depurar código *on the fly*
- La marcada facilidad de uso entre un lenguaje y otro.

Y por supuesto como se comprobó en estas pruebas una marcada mejora en tiempos de cómputo **al menos** para el *procesamiento digital de imágenes.*

Bibliografía

[1] John Nickolls, William J. Dally, *The GPU Computing Era* ,*Micro, IEEE* , Vol.30, No.2, pp. 59-69(2010).

[2] J Sanders, E. Kandrot, *Cuda By Example,An introduction to General-Purpose GPU Programming*, 1a. Edición, Addison Wesley, pp. 21-57, 2011.

[3] S. Tariq, *An Introduction to GPU Computing and CUDA Architecture*, disponible al (23/01/2014) en: http://on-demand.gputechconf.com/gtc-express/2011/presentations/GTC_Express_Sarah_Tariq_June2011.pdf.

[4] OpenCV Documentation disponible al (24/02/2014) en: http://docs.opencv.org/.

Capítulo 5

Resultados de Fusión de Imágenes Multifoco

5.1. Introducción

La fusión de imágenes es el proceso de combinar la información complementaria de multiples imágenes de la misma escena, tomadas en diferentes planos, en una sola imagen, de tal forma que la imagen resultante tendrá una información más detallada de la escena que la información que proporciona cada imagen de forma independiente.

En el caso de imágenes macroscópicas cuando enfocamos a una persona u objeto, la parte que corresponde al fondo se encuentra parcialmente desenfocada; en el caso de imágenes de microscopía, al incrementar el aumento, la cantidad de planos o zonas de la imagen que se encuentran desenfocados aumenta de forma considerable. Por lo que, la información de interés se encuentra "dispersa" en varias imágenes. La fusión de imágenes tiene como objetivo combinar las zonas de interés de una muestra para un microscopista, sin alterar la información original presente en los diferentes planos de enfoque.

Existen varias técnicas basadas en subdividir las imágenes por bloques de tamaño $M \times N$ como la propuesta por Li. et al [1]. Dicha métrica mide el nivel de contraste de los bloques de las imágenes empleando como criterio la Frecuencia Espacial, descrita en el capítulo 3, para medir el nivel de contraste de cada bloque y determinar cuál es el mejor. Además existen

otras métricas como las basadas en Wavelets [2], sin embargo, el principal problema de ambas técnicas reside en los elevados tiempos de cómputo requeridos para determinar el tamaño de bloque óptimo para realizar la fusión.

En este capítulo se presentarán los resultados tanto de tiempos de cómputo como de nivel de "contraste" de las imágenes de prueba empleadas en el desarrollo de este trabajo de tesis. Se hará un análisis cuantitativo respecto de la calidad de la fusión de imágenes así como del tiempo computacional empleado para lograr una fusión. Asimismo se identificará en qué imágenes funciona el método propuesto. De igual forma se explicará a detalle los métodos de fusión empleados para obtener las imágenes resultantes.

5.2. Fusión de Imágenes basada en subdivisión por bloques

Una de las técnicas básicas para fusión de imágenes consiste en dividir las imágenes a fusionar en bloques de tamaño $M \times M$ y, en base a alguna de las técnicas definidas en el capítulo 3, se mide el nivel de contraste de cada bloque. Específicamente dadas dos imágenes A y B de tamaño $M \times N$, el proceso de fusión consiste en:

1. Descomponer las imágenes originales A y B en bloques de tamaño M x N denotados A_i y B_i respectivamente.

2. Calcular la métrica de enfoque en cada bloque y denotar la métrica para A_i y B_i como ME^A y ME^B, respectivamente.

3. Comparar el resultados entre los bloques A_i y B_i y determinar el *i-ésimo* bloque de la imagen fusionada F_i de la forma descrita en la expresión siguiente:

$$F(A,B)_i = \begin{cases} A_i & ME^A > ME^B \\ B_i & ME^A < ME^B \\ \frac{A_i + B_i}{2} & otro \end{cases} \tag{5.1}$$

En pocas palabras, si el bloque *i-ésimo* de la imagen A es mayor que el de la imagen B, el bloque correspondiente a la imagen A se conserva para la nueva imagen fusionada F. En caso contrario, si el bloque *i-ésimo* de la imagen A es menor que el bloque de la imagen B, se conserva el bloque de la imagen B. En caso de ser iguales se hace un promedio con el valor de ambos bloques y se coloca en la imagen fusionada F. La figura 5.1 ilustra dicho proceso.

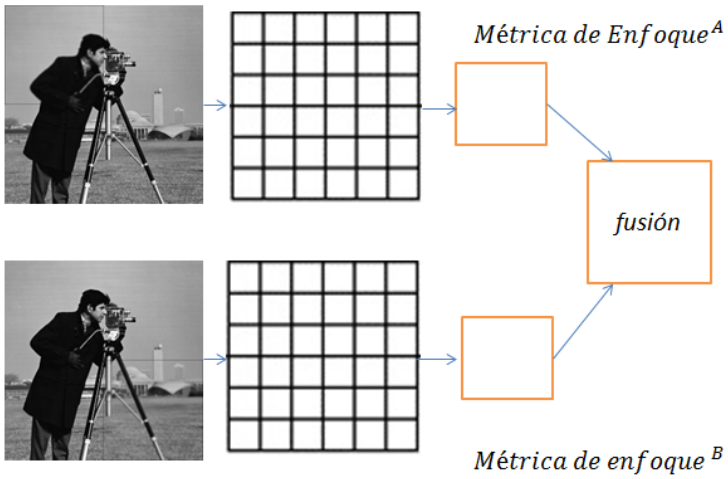


Figura 5.1: Ilustración del proceso de fusión pro bloques para dos imágenes

Dicha técnica presenta buenos resultados usando como métrica de enfoque la frecuencia espacial, sin embargo, una de sus principales desventajas radica en el tiempo de cómputo y en la marcada aparición de *artefactos*, que corresponden a zonas irregulares en la imagen resultante.

En el desarrollo de este trabajo de tesis se probó la fusión de bloques en GPU con el objetivo de mejorar los tiempos de cómputo de este proceso. Sin embargo, dadas las limitaciones de la programación en GPU discutidas en el capítulo 4 sobre el uso no recomendado de ciclos *for*, el proceso resultó en tiempos de cómputo mucho más elevados en la GPU que en la CPU. Ésto debido a la imposibilidad de dividir por bloques en la GPU, siendo entonces necesario copiar cada bloque a la memoria de la GPU (una vez se haya dividido la imagen en la CPU), y posteriormente copiar de regreso a la CPU la información procesada. En la siguiente subsección se presentan los resultados de realizar una fusión por bloques usando el GPU.

5.2.1. Resultados Fusión por Bloques empleando Frecuencia Espacial

Las pruebas siguientes se realizaron mediante la implementación de la métrica FS, descrita en el capítulo 3. Para ello se realizó una comparación en cuanto al tiempo de cómputo necesario para realizar el proceso completo tanto en CPU como en GPU.

La primera imagen de prueba se muestra en la Figura 5.2, dicha imagen corresponde a un bloque de grafito con tres diferentes planos de enfoque tomada a un aumento de 10×, cuyas imágenes tienen un tamaño de 1388×1040 pixeles. Para la fusión se usaron bloques de 4×4 y el proceso de fusión se realizó en cada canal.

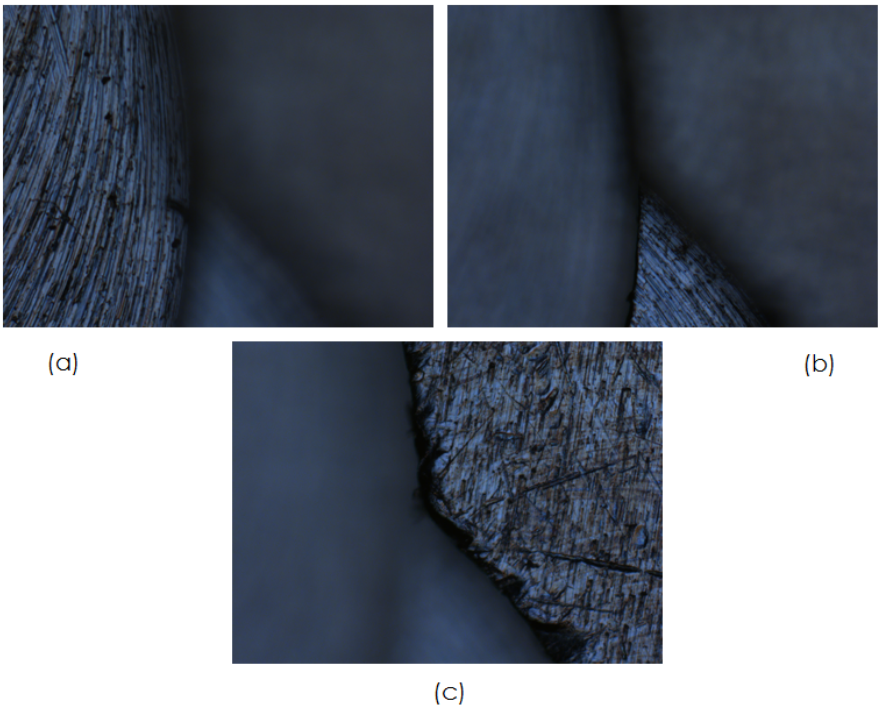


Figura 5.2: Muestra de Grafito cuya región contrastada esta en (a) primer plano (imagen A), (b) segundo plano (imagen B) y (c) tercer plano (imagen C)

El resultado de fusionar las imágenes anteriores se presenta en la figura 5.3. Puesto que se tienen tres planos de enfoque, se requiere primero fusionar las imágenes de (a) y (b) y posteriormente fusionar la imagen resultante con (c). Para verificar el tiempo de cómputo requerido, el tiempo se midió respecto a cada fusión. Se observa que la fusión de la imagen

(a) y (b) requirió un tiempo de **66.40 segundos** en CPU, mientras en la GPU el tiempo aumentó drásticamente a **2311.5456 segundos**. Una vez que se la imagen fusionada, la imagen resultante se fusionó com la imagen (c). La ejecución en CPU de dicha imagen toma un tiempo de **67.98 segundos** en la CPU contra un tiempo de **2345.123 segundos** en la GPU. La tabla 5.1 muestra los tiempos de cómputo.

Tabla 5.1: Tiempo de fusión requerido para la muestra de grafito usando bloques de 4×4

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Muestra de grafito (A+B+C)	134.38	4656.4678

Ambos tiempos de cómputo son elevados, sin embargo, esta prueba indica que una fusión por bloques no es adecuada para su ejecución en una GPU, pues el tiempo de cómputo aumenta hasta 34 veces más en el caso de la GPU.

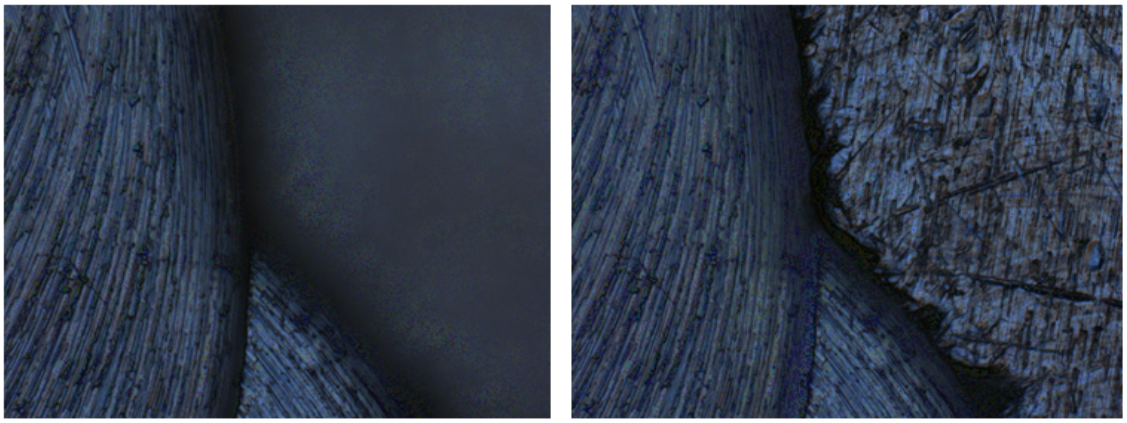


Figura 5.3: Resultados de la Fusión; (a) primer y segundo planos, (b) Imagen resultante y tercer plano

Como se puede observar en la imagen fusionada, esta técnica de fusión integra las regiones mejor contrastadas proporcionando buenos resultados en imágenes macroscópicas, sin embargo en imágenes de microscopía los resultados muestran artefactos. La figura 5.4 muestra que al aplicar un aumento en una región de la imagen se aprecia una pérdida de contraste y al

observar bien la zona aparecen "artefactos" que corresponden a zonas donde la frecuencia espacial parece no haber medido el contraste ya que dichos artefactos son bloques del mismo o aproximadamente del mismo tamaño que el tamaño de los bloques de división.

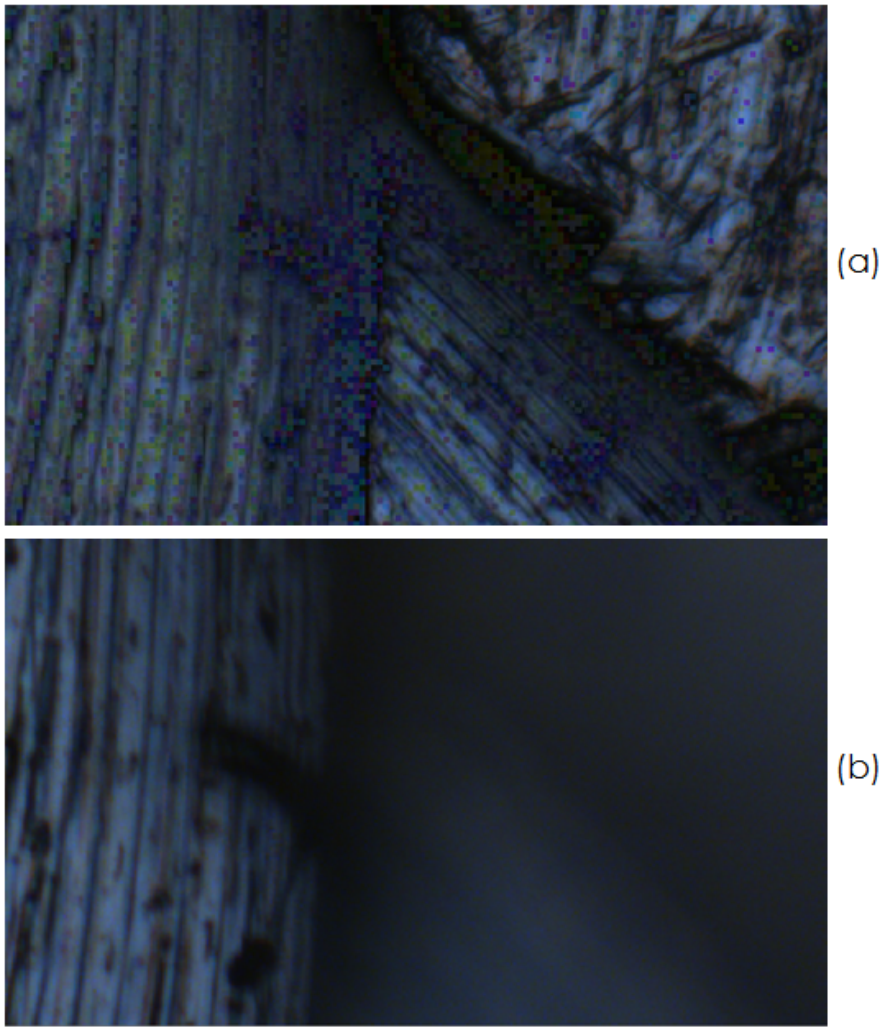


Figura 5.4: Aumento a una región de interés, (a) imagen fusionada, (b) imagen original

De la prueba anterior es posible concluir que la técnica de fusión por bloques presenta las desventajas siguientes:

1. La fusión de imágenes basada en bloques **no** es recomendable para su ejecución en GPU

2. Aún ejecutándose en CPU, la pérdida de contraste y la aparición de bloques sin color definido alteran la imagen original.

5.3. Fusión de Imágenes Pixel a Pixel usando la Distancia Euclidiana

De la sección anterior se comprobó que la fusión de imágenes mediante división por bloques no es recomendable puesto que la morfología de las imágenes de microscopia carece de simetría regular, así pues en este capítulo se explica una técnica de fusión de imágenes multifoco pixel a pixel donde se empleará la distancia euclidiana como medida de contraste para las imágenes en cuestión.

5.3.1. Procedimiento

Para está fusión siempre será necesario tener dos o más imágenes de la misma zona de interés de tal suerte que sea posible medir las diferencias entre ellas. Los pasos necesarios se listan a continuación:

1. Aplicar la métrica de Distancia Euclidiana (descrita en el Capítulo 3), a las imágenes de interés. Una vez aplicada dicha transformación se tienen dos imágenes con los bordes resaltados en sentidos horizontal y vertical, ambas escaladas en el intervalo $[0, 255]$.

2. Aplicar un criterio de selección: el criterio de selección se basa en elegir el pixel con el mayor valor en niveles de gris de las matrices transformadas que fueron obtenidas en el paso 1. El mayor de dos pixeles se tomará en cuenta para la nueva imagen fusionada denotada por F.

3. Repetir el procedimiento del paso 2 para todos los pixeles de la imagen.

Para explicarlo mejor supongamos las imágenes de la Figura 5.5. Las partes (a) y (c) presentan las imágenes originales con diferentes planos de enfoque. La imagen (b) presenta

enfocado el primer plano de la muestra, mientras que en (d) se tiene enfocado el segundo plano de la imagen. Las imágenes (b) y (d) muestran la matriz DE que se obtiene en cada caso.

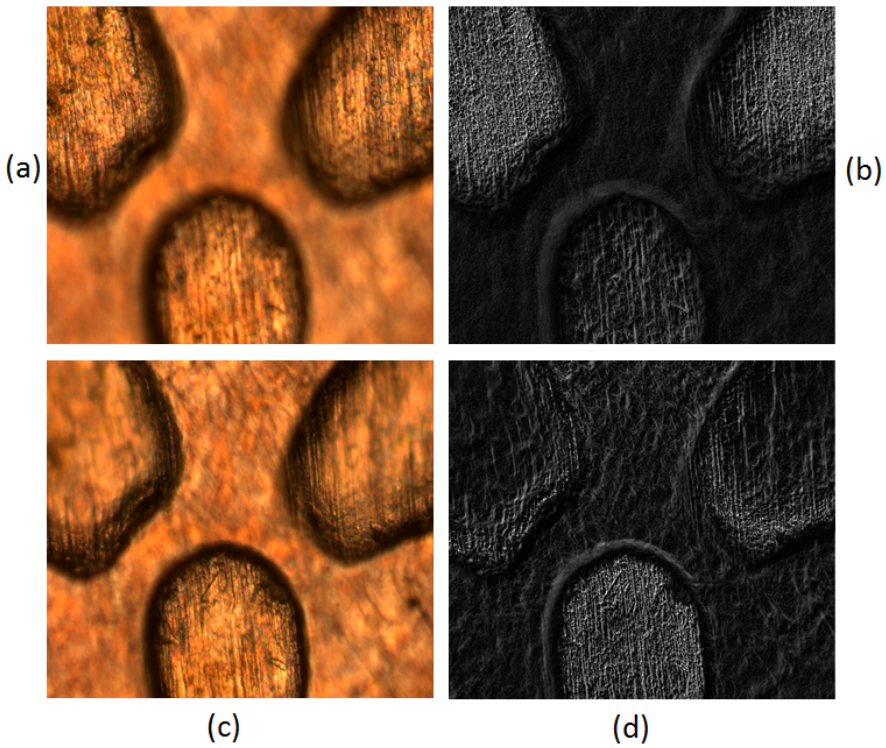


Figura 5.5: Transformación DE para una muestra con dos planos de enfoque; (a) y (c) imágenes originales, (b) y (d) transformación DE.

Como se observa en la Figura 5.6, que corresponde a un aumento de la imagen 5.5(b) y (d), las zonas con el nivel más alto de grises (255 o muy cercano), varían en cada imagen transformada. En base a dicha figura se aprecia que el nivel de grises es mayor en el primer plano de la imagen, justo antes de las orillas del mismo. Así, se concluye que la imagen fusionada para estas dos primeras imágenes estará compuesta en su mayoría por los pixeles con mayor nivel de intensidad, correspondientes a la figura 5.6(a).

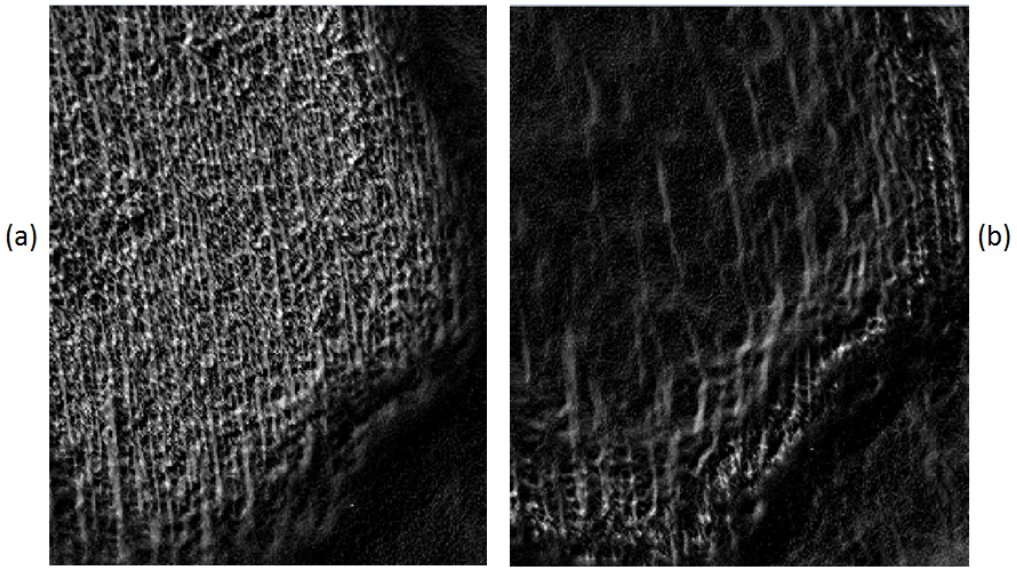


Figura 5.6: Amplificación de las imágenes transformadas en la figura 5.5; (a) imagen con los pixeles resaltados en el primer plano, (b) imagen con pixeles resaltados en el segundo plano.

Una vez realizada la implementación anterior, la imagen resultante de fusionar los dos primeros planos se aprecia en la figura 5.7, donde aún hace falta añadir el último plano de la imagen que corresponde a la parte mas baja de los tres planos de está imagen. El procedimiento de fusión se repite con la imagen resultante de la primera fusión y el último plano de la imagen de interés.

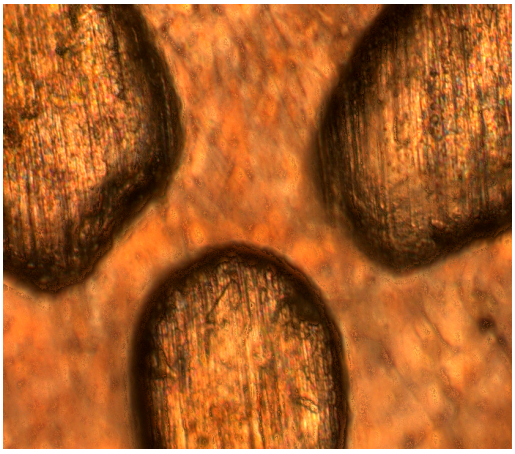


Figura 5.7: Imagen resultante de fusionar los dos primeros planos de la imagen

El resultado es una imagen de tamaño $M-1 \times N-1$ con los tres planos enfocads, donde M y N corresponden al número de filas y columnas de la imagen original. Es importante destacar que la fusión se realiza pixel a pixel tomando en cuenta los pixeles RGB de las imágenes originales.
Los incisos (a) y (b) de la figura 5.8 muestran las matrices de distancia euclidiana para el último paso de la fusión mientras que el inciso (c) muestra la imagen resultante.

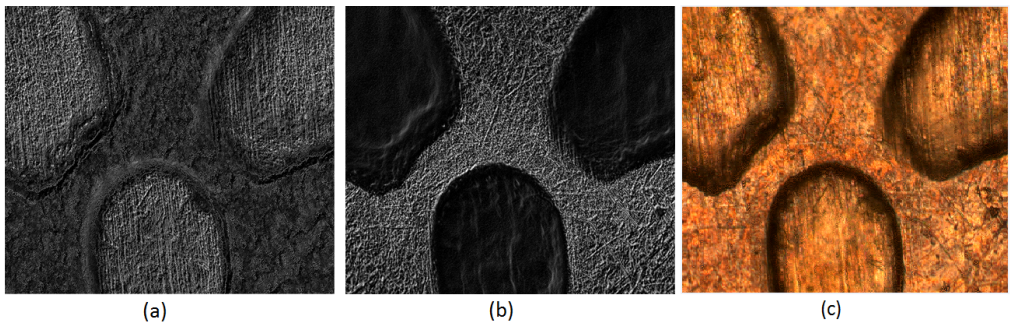


Figura 5.8: (a) matriz para los primeros dos planos fusionados, (b) matriz correspondiente al último plano, (c) imagen resultante.

La imagen resultante posee un contraste correcto así como la minima aparición de artefactos en comparación con la fusión por bloques como se puede apreciar en la figura 5.9. Computacionalmente, hablando los tiempos de cómputo para esta imagen de 833×728 pixeles, toman en la CPU 105.45 segundos para las primeras dos imágenes y 108.45 segundos para terminar la fusión, mientras que en la GPU demora para las primeras dos imágenes 0.4228 segundos y para terminar la fusión 0.4282 s. La tabla 5.2 muestra dichos tiempos de cómputo.

Tabla 5.2: Tiempo Requerido para la fusión de la muestra de la moneda mexicana basada en DE.

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Moneda Mexicana (A+B+C)	213.9	0.851

Por lo tanto, la técnica antes descrita presenta mejores resultados que más adelante se medirán cuantitativamente empleando las métricas descritas en el capítulo 3. La siguiente sección muestra una mejora a está técnica empleando las máscaras de convolución de Kirsch como un

detector de bordes previo al cálculo de las matrices de distancia euclidiana, resaltando aún más los detalles de las imágenes originales.

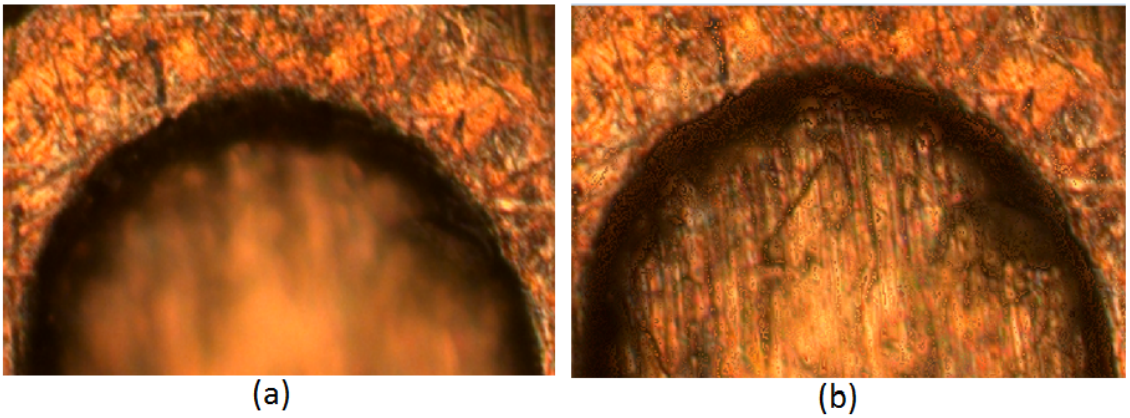


Figura 5.9: Aumento a una zona de interés; (a) imagen original, (B) imagen fusionada

5.4. Fusión de imágenes empleando la Distancia Euclidiana y las máscaras de Kirsch

5.4.1. Detección de Bordes

La detección de bordes en visión por computadora se emplea como herramienta para la segmentación de imágenes y extracción de características. Esencialmente un borde en un imagen es la zona que corresponde a las "orillas" del objeto. Concretamente los bordes son la zona donde la intensidad de la imagen cambia de forma drástica [3].
La información correspondiente a un borde en una imagen puede hallarse, evaluando la relación que un determinado pixel tiene con sus vecinos. Si un pixel comparte niveles de gris muy similares alrededor suyo, probablemente en ese punto no se trate de un borde. Por el contrario, si los valores vecinos de un pixel varían demasiado ese punto corresponde a un borde.
La primera derivada de una función imagen $f(x, y)$ puede usarse para detectar la presencia de un borde. La gran mayoría de métodos están implementados como la convolución de una imagen con máscaras discretas basadas en aproximaciones a operadores diferenciales [4]. Dichos operadores son los encargados de medir el cambio en el brillo de la función imagen.
Algunos operadores como es el caso de las máscaras de *Kirsch* proporcionan información adicional sobre la orientación de los bordes, en comparación con otros operadores que solo proporcionan información acerca de la existencia o no de un borde en un determinado punto.

Máscaras de Kirch

Mediante la operación de convolución de una máscara sencilla y sus rotaciones en 8 direcciones: Norte (N), Noroeste (NO), Noreste (NE), Sur (S), Suroeste (SO), Sudeste (SE), Este (E) y Oeste (O), es posible detectar la información de los bordes de una imagen. El valor máximo del operador se calcula como el máximo de todas las direcciones [5]. La figura 5.10 muestra las ocho máscaras de Kirsch.
Las imágenes obtenidas con la detección de bordes de Kirsch pueden ser utilizadas para obtener una imagen con los bordes remarcados en las 8 direcciones. De esta manera, para una imagen

$$N = \begin{bmatrix} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \end{bmatrix} \quad NO = \begin{bmatrix} -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{bmatrix} \quad O = \begin{bmatrix} 5 & 5 & 5 \\ -3 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix} \quad SO = \begin{bmatrix} 5 & 5 & -3 \\ 5 & 0 & -3 \\ -3 & -3 & -3 \end{bmatrix}$$

$$S = \begin{bmatrix} 5 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & -3 & -3 \end{bmatrix} \quad SE = \begin{bmatrix} -3 & -3 & -3 \\ 5 & 0 & -3 \\ 5 & 5 & -3 \end{bmatrix} \quad E = \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & -3 \\ 5 & 5 & 5 \end{bmatrix} \quad NE = \begin{bmatrix} -3 & -3 & -3 \\ -3 & 0 & 5 \\ -3 & 5 & 5 \end{bmatrix}$$

Figura 5.10: Máscaras de Kirsch en 8 direcciones cardinales

$G(i,j)$, la imagen resultante $K(i,j)$ es el resultado de la operación de convolución con las máscaras de Kirsch, acorde con la siguiente expresión:

$$K(i,j) = \text{máx}(G(i,j) \otimes L_p) \quad con \quad p = 1, ..., 8. \tag{5.2}$$

El objetivo de aplicar las máscaras de Kirsch es contrastar con mayor detalle los bordes o cambios de intensidad de una imagen, como se indicó anteriormente cada operador de Kirsch proporciona información diferente. La figura 5.11 muestra la información que proporciona cada máscara aplicada a la imagen de referencia "lena".

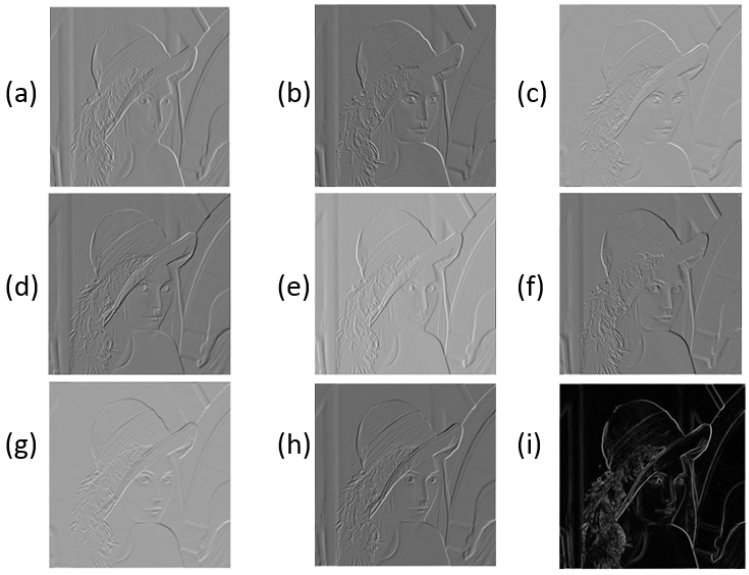


Figura 5.11: Imagen resultante de la convolución de la imagen original con la máscara de Kirch a: (a)0°,(b) 45°,(c) 90°,(d) 135°,(e) 180°, f) 225°,(g) 270°,(h) 315°, (i) máximo en todas las direcciones.

5.4.2. Procedimiento de fusión

El proceso se basa en una fusión pixel a pixel; donde las imágenes a fusionar pasan por un preprocesado previo usando las máscaras de Kirsch para saber cuál es la zona mejor enfocada de cada imagen en base a sus niveles de gris. El procedimiento completo se lista a continuación:

1. Obtener la imagen filtrada de los planos a fusionar empleando las máscaras de Kirsch mencionadas con anterioridad.

2. De las imágenes obtenidas calcular las matrices de distancias euclidianas.

3. Aplicar un criterio de selección: el criterio de selección se basa en elegir el pixel con el mayor valor en niveles de gris de las matrices de distancia euclidiana obtenidas en el paso anterior.

4. La posición (i, j) del mayor de los pixeles, elegido en el paso 3, será empleada para extraer el pixel RGB de la imagen original. Este pixel formará la imagen fusionada.

Dicho procedimiento se puede repetir para fusionar n imágenes digitales, considerando que el procedimiento se debe repetir un número de veces correspondiente al número de planos a fusionar. Para ilustrar mejor, suponga las imágenes de la Figura 5.12 que corresponden a las imágenes de una moneda mexicana con tres planos de enfoque la Figura 5.12(d)−(f) muestran las imágenes con los bordes detectados.

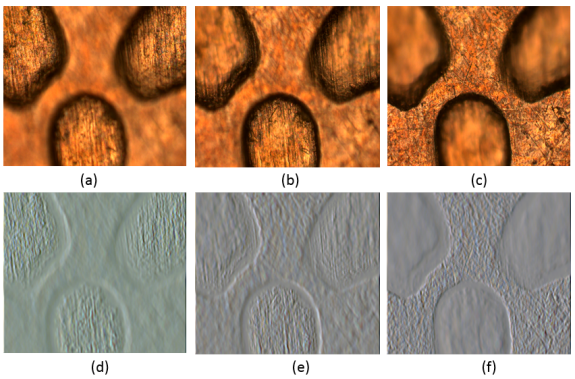


Figura 5.12: (a)-(c) imágenes originales, (d)-(f) imágenes resultantes de aplicar las máscaras de Kirsch

El siguiente paso es aplicar la transformación de distancias euclidianas para cada una de las imágenes anteriores. En la figura 5.13(a)-(c) se muestra el efecto de aplicar el proceso de distancias euclidianas, mientas que las imágenes de la figura 5.13(d)-(f) corresponden al proceso completo de aplicar las máscaras de Kirsch y la transformación de distancias euclidianas.

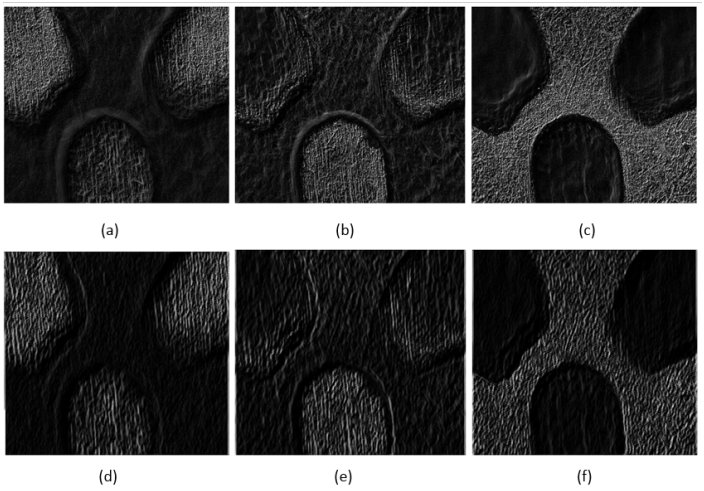


Figura 5.13: Resultado de aplicar el paso dos a las imágenes de entrada; Arriba: Solo Distancia Euclidiana, Abajo: Kirsch + Distancia Euclidiana

Un aumento a una zona a las imágenes anteriores permitirá observar las diferencias entre ellas, demostrando que la aplicación previa de la máscara de Kirsch a las imágenes originales genera un efecto de suavizado a los bordes; de esta manera al calcular la matriz DE los bordes se resaltan nuevamente y se elimina el ruido extra. La figura 5.14(a), muestra el efecto de aplicar DE, mientras que el inciso (b) muestra el efecto de aplicar el método aunado a las máscaras de Kirsch. De acuerdo con el procedimiento de fusión, se toma el valor máximo de las imágenes transformadas, el valor asociado RGB se coloca en la imagen fusionada. El resultado de la fusión se observa en la figura 5.15.

Finalmente, en la Figura 5.16 se muestra un aumento a una región de las imágenes fusionada por los métodos antes mencionados, la imagen (a) corresponde a la implementación con distancia euclidiana y la imagen (b) corresponde a métrica de distancia euclidiana junto con las máscaras de Kirsch.

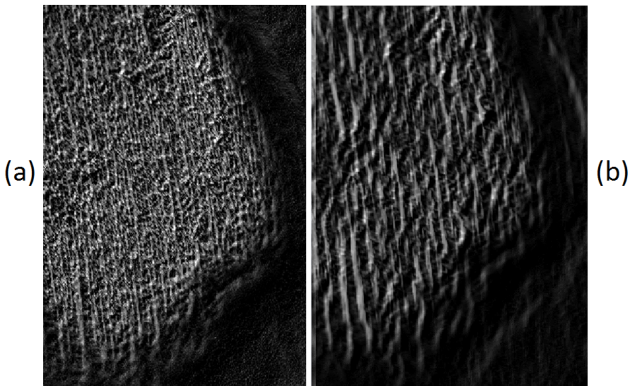


Figura 5.14: (a) Solo Distancia Euclidiana, (b) Kirsch + Distancia Euclidiana

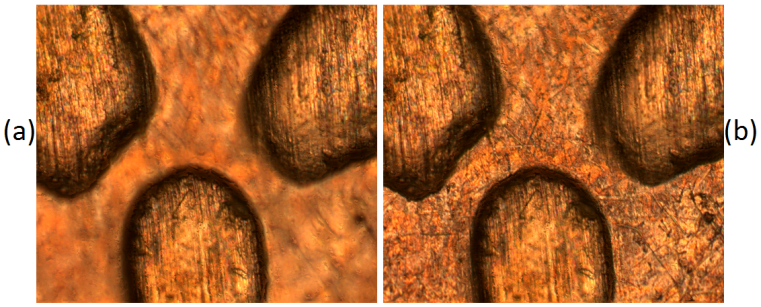


Figura 5.15: Imágenes resultantes fusionadas usando DE y las máscaras de Kirsch

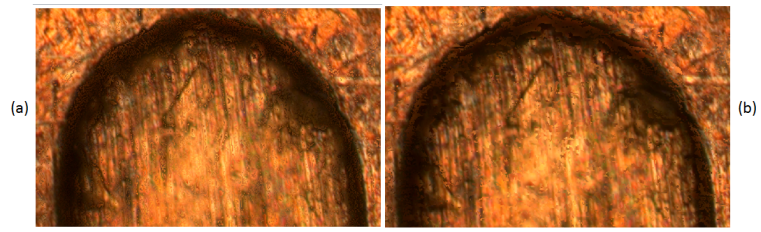


Figura 5.16: Aumento a una región de interés en ambas fusiones, Inciso (a) fusión por DE e Inciso(b) fusión DE + Kirsch

En cuanto a la carga computacional, la diferencia es mínima en ambas implementaciones. Los tiempos de cómputo para está implementación se presentan en la tabla 5.3.

Tabla 5.3: Tiempo requerido para la fusión de la moneda mexicana usando DE + Kirsch

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Moneda Mexicana (A+B+C)	216.995	1.659

5.5. Fusión de Imágenes usando Ángulo Espectral

El ángulo espectral en el espacio de color RGB es una medida para determinar cuán similares son dos vectores P_1 y P_2. El ángulo de espectral entre dos vectores P_1 y P_2 se denota como:

$$\theta_{P_1,P_2} = \cos^{-1} \frac{P_1 P_2}{\|P_1\|\|P_2\|} \tag{5.3}$$

Así en base a la ecuación 5.3 se deduce que cuando dos pixeles sean similares el ángulo entre ellos será muy pequeño, cuando la diferencia sea mayor el ángulo crecerá y cuando los pixeles sean iguales el ángulo será de 0°

Supongamos por ejemplo los siguientes valores para 2 pixeles $P_1 = (255, 255, 255)$ y $P_2 = (255, 255, 255)$, sustituyendo en la ecuación (5.3) se tienen los siguiente valores.

$$\theta_{P_1,P_2} = \cos^{-1}\left[\frac{(255,255,255)\cdot(255,255,255)}{\sqrt{255^2+255^2+255^2}\cdot\sqrt{255^2+255^2+255^2}}\right]$$

$$\theta_{P_1,P_2} = \cos^{-1}\left(\frac{195075}{(441{,}6730)\cdot(441{,}6730)}\right) = \left(\frac{195075}{195075}\right)$$

$$\theta_{P_1,P_2} = \cos^{-1}(1) = 0$$

Entonces acorde al resultado previo o el ángulo espectral entre los vectores de color P_1 y P_2 cuyos niveles de gris corresponden al color blanco es de 0. Por tanto, el Ángulo Espectral (AE) puede ser empleado para medir la similitud entre dos pixeles de una imagen a color. Supongamos nuevamente la imagen de prueba "lena"; la resultante de aplicar el ángulo espectral se muestra en la Figura 5.17. De dicha prueba es posible observar que el ángulo espectral registra los cambios en los niveles de gris para el espacio de color RGB. En el caso de imágenes

multifoco esta técnica brinda información de las zonas mejor enfocadas en cada plano de la imagen.

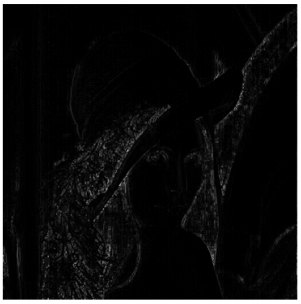
(a)   (b)

Figura 5.17: imagen de prueba Lena, (a) imagen en color, (b) ángulo espectral de la imagen de prueba

5.5.1. Procedimiento de fusión Ángulo Espectral

El procedimiento para la fusión es similar al procedimiento de la sección 5.4 donde se realiza una fusión escogiendo en pixel cuyo valor de gris es el mayor. El procedimiento se detalla a continuación:

1. Aplicar la transformación del **ángulo espectral** a las imágenes de interés. Una vez aplicada dicha transformación se tienen dos matrices con los bordes resaltados de cada una de las imágenes escaladas en el intervalo [0,255].

2. Aplicar un criterio de selección: el criterio de selección se basa en elegir el pixel con el mayor valor en niveles de gris de las matrices de ángulo espectral obtenidas en el paso anterior.

3. La posición (i, j) del mayor de los pixeles, elegido en el paso 2, será empleada para extraer el pixel RGB de la imagen original. Este pixel se colocará en la posición (i, j) de la imagen fusionada.

Para explicar dicho proceso suponga las imágenes de la figura 5.18. Los incisos (a), (b) y (c) corresponden a las imágenes originales, mientras que (d), (e) y (f) corresponden a la imagen una vez aplicado el ángulo espectral.

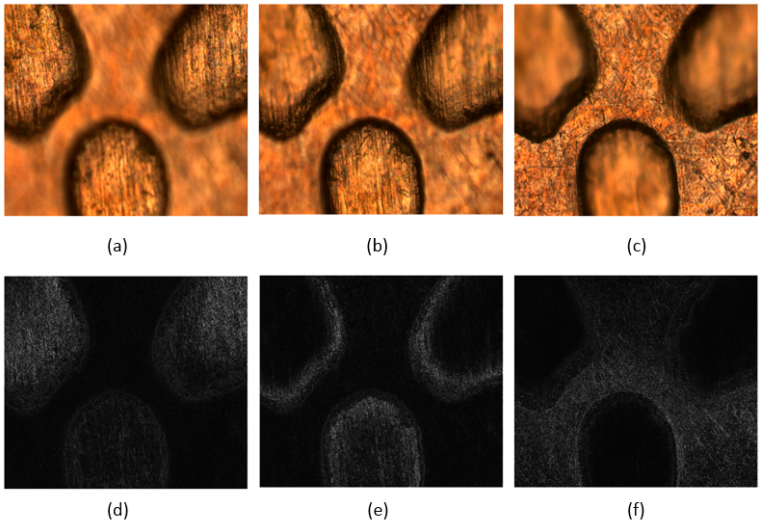


Figura 5.18: Ángulo espectral de las imágenes de prueba, (a-c) imágenes originales, (d-f) transformada ángulo espectral

Para la fusión de imágenes nuevamente se elige el pixel cuyo niveles de gris sea el mayor en las imágenes transformadas. Al comparar las zonas amplificadas de la figura 5.18 (a) y (b), mismas que se observan en la figura 5.19, observando detalladamente es lógico pensar que la imagen fusionada que resulta del proceso anterior corresponderá en su mayoría al grupos de pixeles de la primera imagen.
En la Figura 5.21 se muestran las imágenes resultantes de las dos fusiones para llegar a la imagen final con los tres planos fusionados. Es importante destacar que la fusión en todos los casos se realiza tomando los valores RGB de las imágenes originales, asociados a los máximos de intensidad de las imágenes transformadas.
La figura 5.20 muestra una comparativa de una zona de interés de los tres métodos de fusión. De izquierda a derecha: Distancia Euclidiana, Distancia Euclidiana + Máscara de Kirsch, Ángulo Espectral.

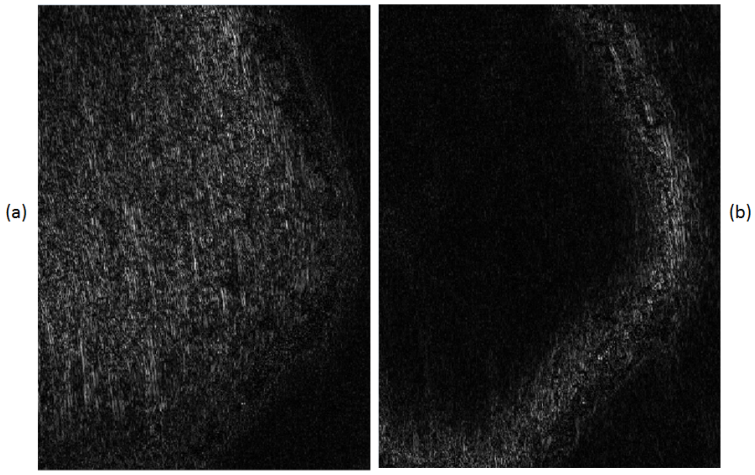


Figura 5.19: Aumento de la matriz de ángulo espectral de dos zona distintas del área de interés

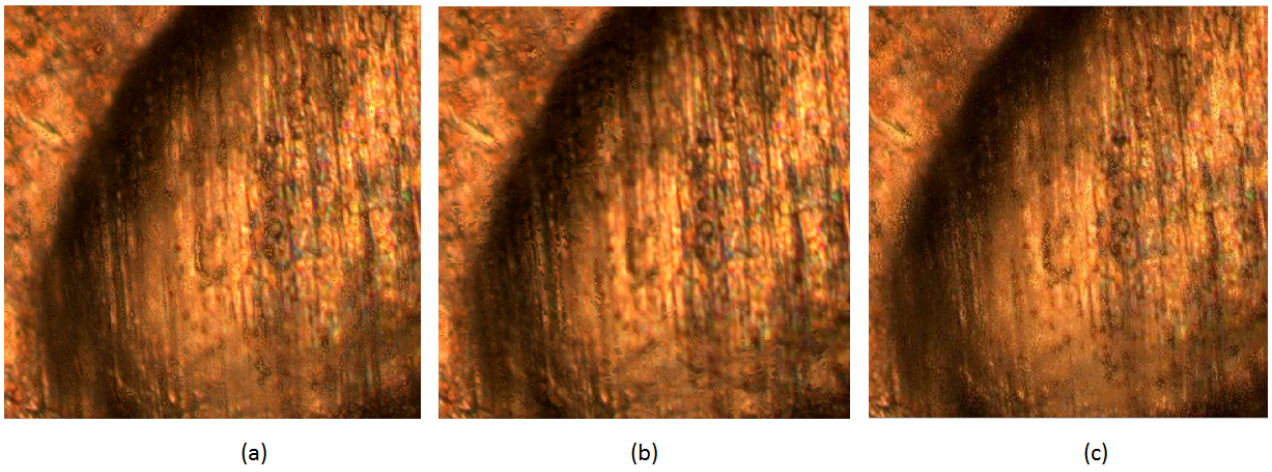


Figura 5.20: Aumento a una zona de la imágenes resultantes de los tres métodos, (a) AE, (b) DE, (c) DE+Kirsch

Finalmente los tiempos de cómputo de está métrica tanto en CPU como en GPU se observan en la Tabla 5.4, comparados con la tabla 5.3 los tiempos de cómputo del ángulo espectral en CPU son menores, sin embargo, los tiempos en GPU incrementan en un 40 %.

Tabla 5.4: Tiempo de fusión para la moneda mexicana usando ángulo espectral

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Moneda Mexicana (A+B+C)	114.269827	0.299824

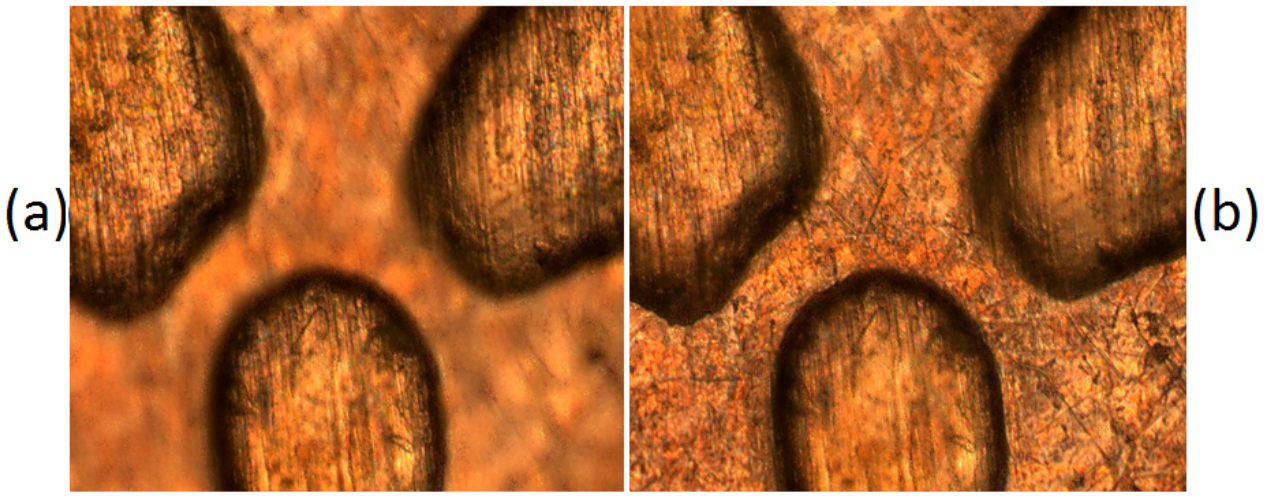


Figura 5.21: Imágenes resultantes de la fusión empleando ángulo Espectral, (a) primeros dos planos fusionados, (b) fusión de los tres planos.

5.6. Análisis cuantitativo de imágenes fusionadas

Esta sección corresponde a la comparación de los resultados de las pruebas realizadas en imágenes adquiridas en el microscopio, dichas imágenes fueron sometidas a pruebas que miden su nivel de contraste en base a 4 métricas: Frecuencia Espacial (FS), Laplaciano (LAP), Transformada Coseno Discreta de Frecuencias Medias (MDCT), y Vollath 4 (VOL4). Dichas métricas son computacionalmente eficientes y exactas en cuanto al nivel de contraste de una imagen [6].

Las pruebas corresponden a dos secciones: en la primera se mencionará las características de las imágenes a procesar y en la segunda se expondrá su nivel de contraste acorde con las métricas antes mencionadas. Además se incluyen los tiempos de cómputo tanto en CPU como en GPU de cada una de las muestras y técnicas descritas. Cada subsección contendrá solo la imagen mejor contrastada resultante de las fusiones.

5.6.1. Especificaciones de Hardware

Para este trabajo se utilizó una computadora con las siguientes especificaciones: CPU Xeon E3-1230 V3 @ 3.30 Ghz con 4 núcleos físicos y un máximo de 8 threads. Este equipo cuenta 16 Gb de Memoria RAM DDR3 a 798.1 MHz. Como GPU posee una tarjeta gráfica Nvidia Quadro K4000, esta tarjeta tiene en su interior 4 Multiprocesadores con 768 núcleos CUDA, asimismo cuenta con 3 Gb de memoria GDDR5. En cuánto al software se emplea MATLAB 2013a en 64 bits, y el el toolkit de CUDA 5.0. Todo bajo Microsoft Windows 8 Pro como sistema operativo.

5.6.2. Muestra de grafito

La muestra total consta de tres planos con diferente enfoque: la zona a la izquierda, central y a la derecha.

Aquí la Frecuencia Espacial, Laplaciano y la MDCT coinciden entre sí, concluyendo que la imagen mejor fusionada corresponde a la técnica de distancias euclidianas en conjunto con la técnica de Kirsch. La figura 5.22 muestra dicha imagen fusionada, en tanto que la tabla 5.6

Tabla 5.5: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	1	0.8842	1	0.9677
DE	0.6979	1	**0.5617**	1
DE +Kirsch	**0.6468**	**0.1468**	0.6646	**0.3636**

muestra los tiempos de cómputo para cada caso.

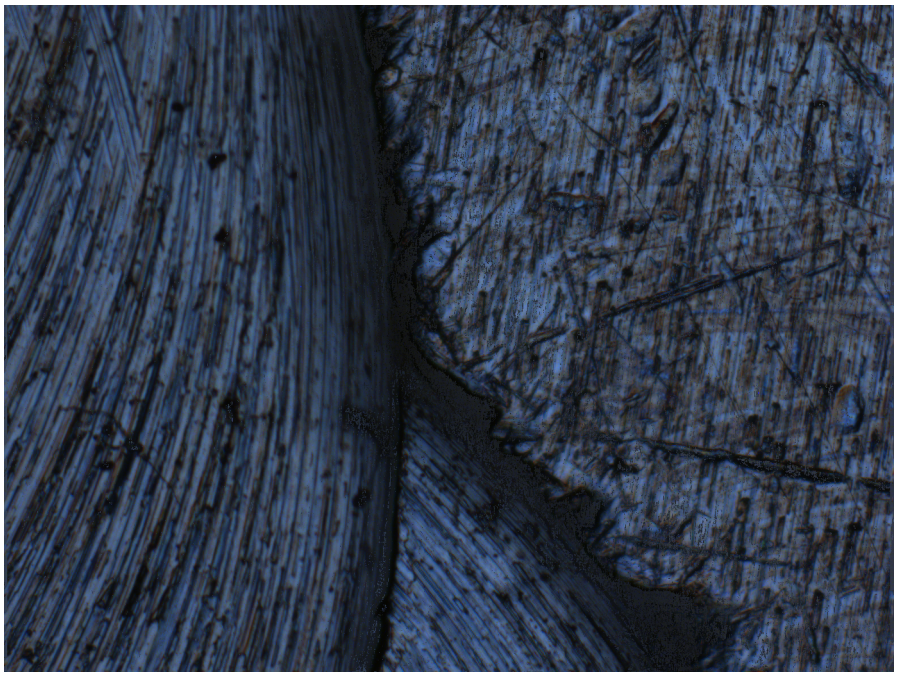


Figura 5.22: Imagen fusionada empleando la Distancia Euclidiana + Kirsch

Tabla 5.6: Tiempo requerido para la muestra de grafito

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	260.2833	0.481592
DE	523.1254	1.0442
DE+ Kirsch	483.6126	1.0551

5.6.3. Muestra de metal desvastado

La muestra consta de 23 planos de enfoque. La figura 5.23 muestra la imagen resultante y las tablas 5.7 y 5.8 muestran cuantitativamente la imagen mejor enfocada, y los tiempos de cómputo respectivamente. Acorde con la Tabla 5.7 la mejor fusión se logra empleando DE+Kirsch.

Tabla 5.7: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	1	**0.4032**	0.6720	0.9624
DE	**0.8255**	0.4577	1	1
DE +Kirsch	0.6740	1	**0.4086**	**0.6808**

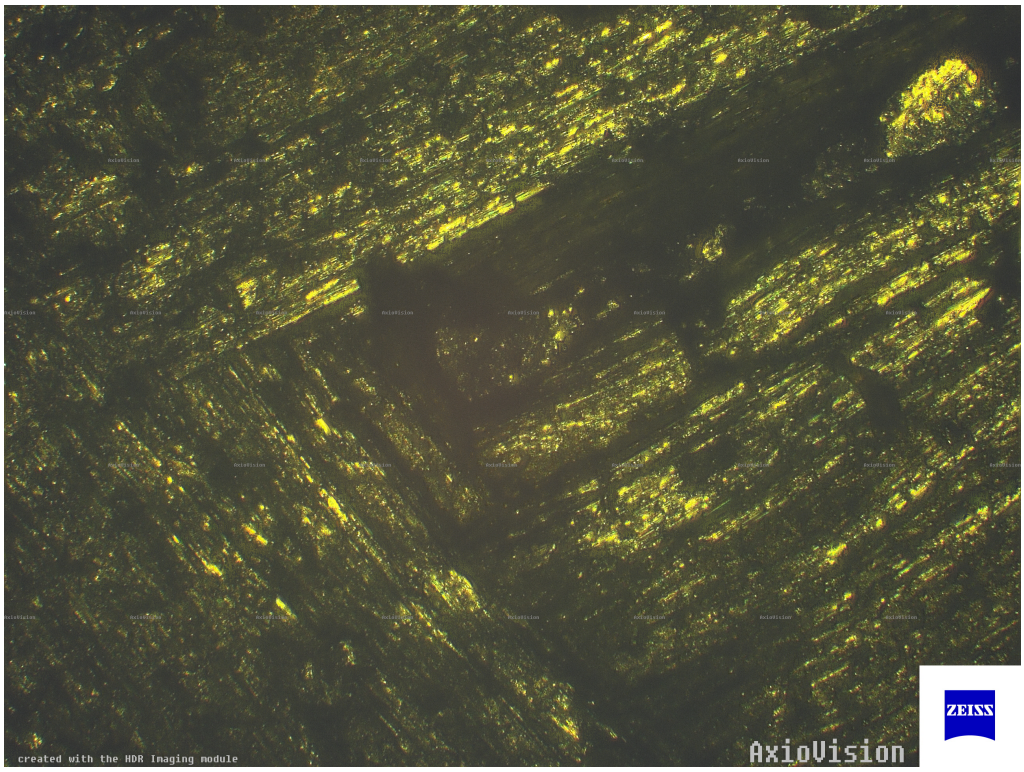


Figura 5.23: Imagen fusionada empleando la Distancia Euclidiana + Kirsch

Tabla 5.8: Tiempo requerido para el metal desvastado

Método	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	10116.60796	9.575215
Distancia Euclidiana	18301.46	10.3741
DE+ Kirsch	18068.90578	16.241444

5.6.4. Tejido de pez

La muestra consta de tres planos con diferente enfoque. La imagen fusionada resultante se muestra en figura 5.24, la medición del contraste se muestra en la tabla 5.9 y los tiempos de cómputo tanto en CPU como GPU en la tabla 5.10.

Tabla 5.9: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	**0.9069**	0.9121	0.8512	0.9082
DE	1	**0.7635**	1	1
DE +Kirsch	0.9462	1	**0.6057**	**0.8469**

Es posible concluir entonces que la mejor métrica corresponde a la uso de las distancia euclidianas + Kirsch, siendo esta la imagen con mejores resultados. Esta técnica consume mayor tiempo de cómputo como se aprecia en la tabla 5.10.

Tabla 5.10: Tiempo requerido para la muestra de pez

Método	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	266.93087	0.423361
Distancia Euclidiana	487.26	0.505624
DE+ Kirsch	487.39876	0.578383

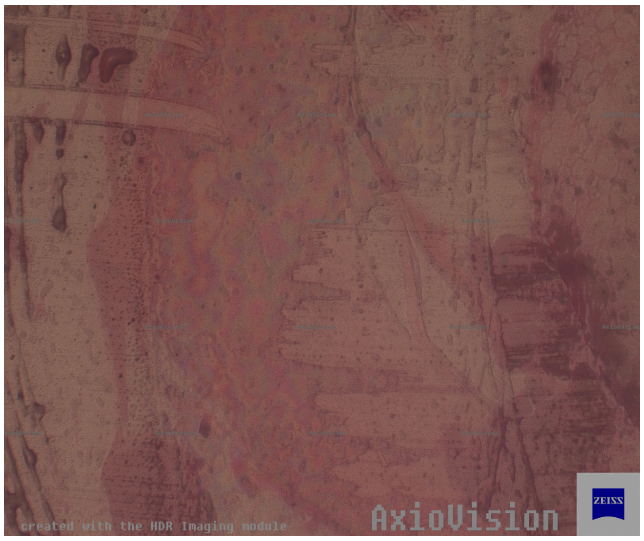


Figura 5.24: Imagen fusionada correspondiente a la espina dorsal se un pez empleando la Distancia Euclidiana + Kirsch

5.6.5. Aluminio

La muestra se compone por 19 planos con diferente enfoque. La imagen mejor fusionada nuevamente corresponde a la métrica que emplea las distancias euclidianas y las máscaras de Kirsch, como lo muestra la tabla 5.11.

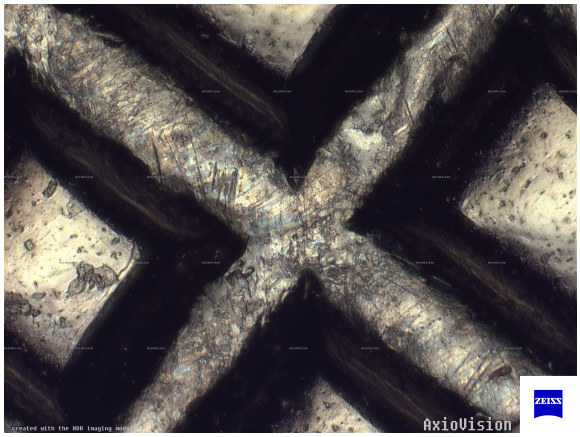


Figura 5.25: Imagen fusionada correspondiente a una llave de aluminio empleando la Distancia Euclidiana + Kirsch

Tabla 5.11: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	0.9214	0.9946	0.6787	1
DE	1	1	**0.4742**	0.9878
DE +Kirsch	**0.6757**	**0.4888**	1	**0.8423**

En cuanto a tiempos de cómputo el ángulo espectral se mantiene como en los demás casos con el menor tiempo; los tiempos de las tres técnicas aparecen en la siguiente tabla. Cabe recalcar que para todas las tablas de medida de enfoque los valores en **negrilla** corresponde a la técnica que presenta mejores resultados.

Tabla 5.12: Tiempo requerido para la muestra de aluminio

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	9249.55	6.7
Distancia Euclidiana	15559.58	6.15
DE+ Kirsch	15491.79	9.08

5.6.6. Moneda mexicana

La muestra se compone por 18 diferentes planos de enfoque. Nuevamente en cuánto a contraste la mejor métrica es aquella basada en la distancia euclidiana y las máscaras de Kirsch (Tabla 5.14), mientras que; en tiempos de cómputo la mejor métrica corresponde al igual que en casos anteriores al ángulo espectral (tabla 5.13). La figura mejor contrastada se muestra en la figura 5.26.

Tabla 5.13: Tiempo requerido para la muestra de la moneda mexicana

Método	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	8195.81	6.03
Distancia Euclidiana	14345.89	7.3415
DE+ Kirsch	14566.68	10.8913

Figura 5.26: Imagen fusionada correspondiente a una moneda mexicana empleando la Distancia Euclidiana + Kirsch

Tabla 5.14: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	**0.4335**	0.7414	0.519	0.8291
DE	0.7508	1	**0.2864**	1
DE + Kirsch	1	**0.3404**	1	**0.6846**

5.6.7. Cobre

En este caso se tienen 12 diferentes planos de enfoque. La mejor imagen fusionada se muestra en la figura 5.27.

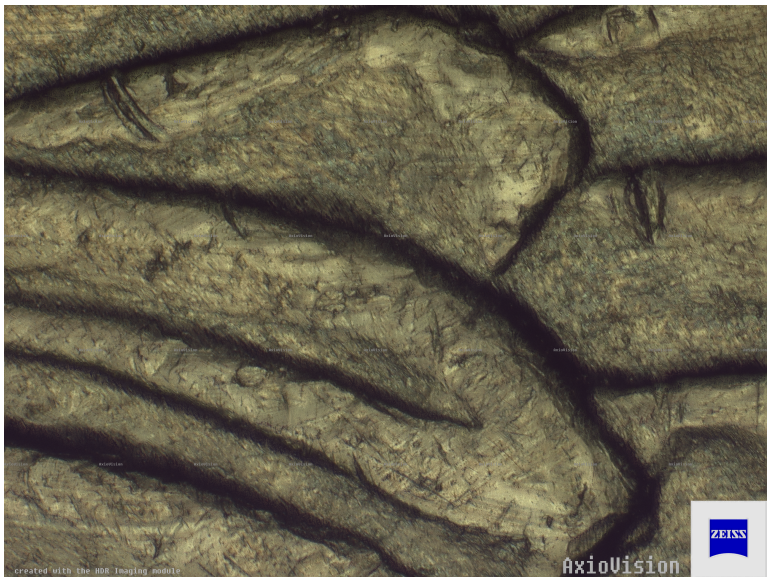


Figura 5.27: Imagen fusionada (Moneda Mexicana) empleando la Distancia Euclidiana + Kirsch

La mejor imagen acorde al análisis es nuevamente la que se obtiene de combinar las máscaras de Kirsch con la distancia euclidiana; finalmente los tiempos de cómputo se muestra en la tabla

Tabla 5.15: Imagen mejor contrastada acorde con:

Método	Frecuencia Espacial (s)	Laplaciano (s)	Vollath 4	MDCT
Ángulo Espectral	0.994	0.7287	0.6895	0.819
DE	1	1	**0.4826**	1
DE +Kirsch	**0.8416**	**0.452304**	1	**0.8583**

donde se observa nuevamente que los tiempos más bajos corresponden al ángulo espectral.

Tabla 5.16: Tiempo requerido para la muestra de Cobre

Imagen	Tiempo CPU (s)	Tiempo GPU (s)
Ángulo Espectral	4723.75	3.7564
Distancia Euclidiana	9453.67	4.4639
DE+ Kirsch	9578.85	6.6649

Con el propósito de analizar el comportamiento de los tiempos de cómputo de los tres métodos, se midió el tiempo de cómputo de cada método respecto del número de planos de las diferentes muestras empleadas. La figura 5.28 muestra el comportamiento de los tiempos de cómputo de la CPU. Dicha figura ilustra el incremento de la carga computacional respecto del número de planos a fusionar, siendo el mínimo de muestras 3 y el máximo 23. Se aprecia que en general la métrica que consume el menor tiempo de cómputo es aquella basada en el ángulo espectral. De igual forma las métricas DE y DE+Kirsch presentan tiempos de cómputo muy similares.

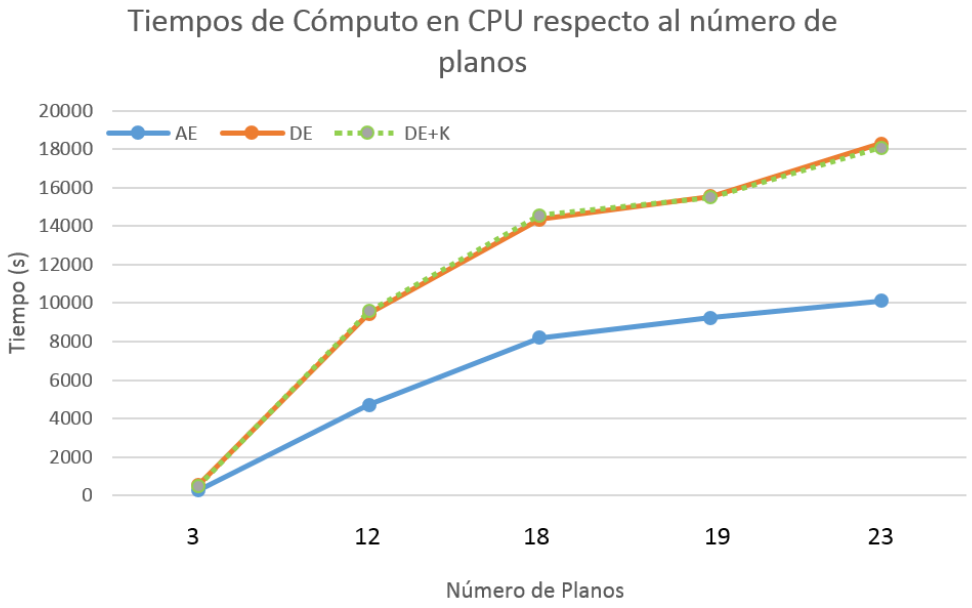


Figura 5.28: Gráfica donde se observa el comportamiento de la CPU respecto del número de planos.

Como complemento en la figura 5.28 se observa el comportamiento de cómputo en la GPU donde adicional a la disminución en tiempos de cómputo entre la CPU y la GPU, se observa que en GPU la distancia euclidiana y el ángulo espectral son más cercanos en tiempos de cómputo e incluso cuando la carga computacional aumenta en 19 muestras a fusionar la distancia euclidiana es inferior en tiempos de cómputo al ángulo espectral. Mientras que los tiempos de cómputo mas elevados corresponde la DE+Kirsch.

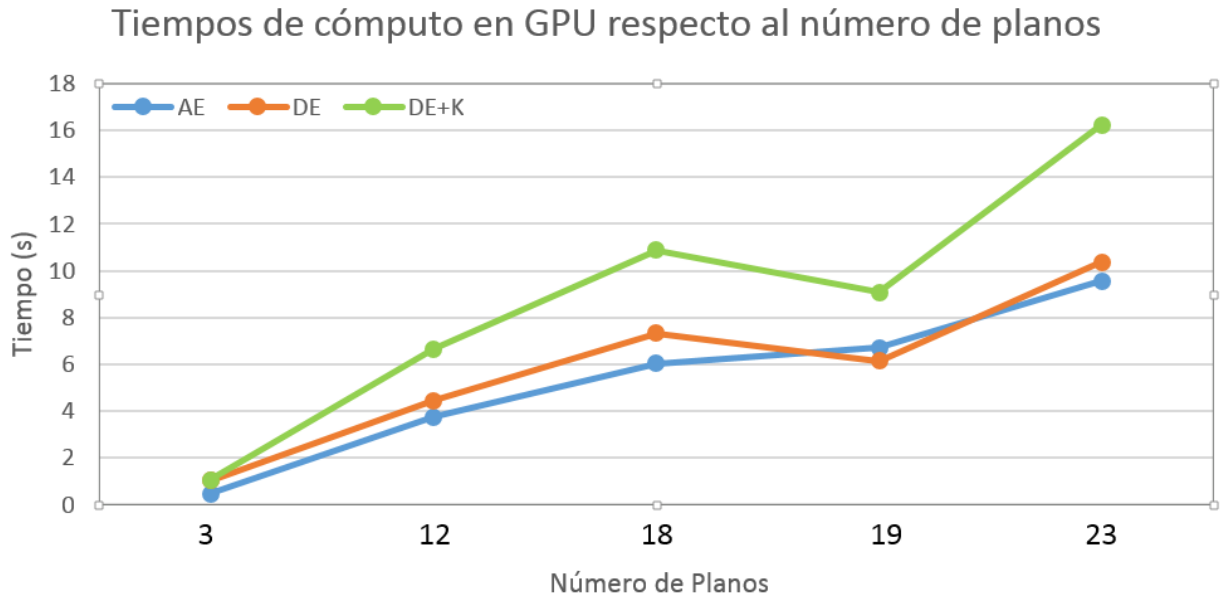


Figura 5.29: Gráfica donde se observa el comportamiento de la GPU respecto del número de planos.

Para finalizar la figura muestras los tiempos de cómputo tanto de la CPU como de la GPU para el caso DE+Kirsch, donde se observa que la GPU ha sido capaz de disminuir los tiempos de cómputo hasta 3 órdenes de magnitud, demostrando que el uso de la GPU es una alternativa para disminuir los tiempos de procesamiento de la fusión de imágenes.

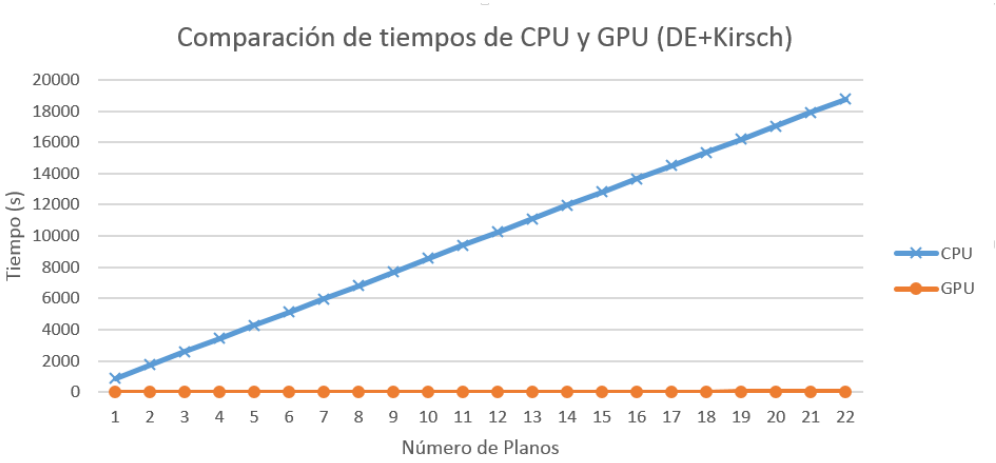


Figura 5.30: Tiempos de cómputo de la GPU respecto del CPU para el caso de fusión de la figura 5.23

Finalmente es importante mencionar que las imágenes obtenidas tanto en CPU como en GPU son imágenes idénticas. Por lo tanto se puede concluir que en el caso de la fusión de imágenes el uso de la GPU no altera el resultado. Para demostrarlo de tomaron las imágenes correspondiente a la figura 5.20 para el caso de la CPU y la GPU y se realizó una resta entre ellas sabiendo de antemano que si las imágenes son iguales el resultado será una matriz en color negro. La figura 5.31 (a) muestra la imagen resultante del proceso en CPU, el inciso (b) la imagen en GPU, y el inciso (c) el resultado de la resta entre ellas.

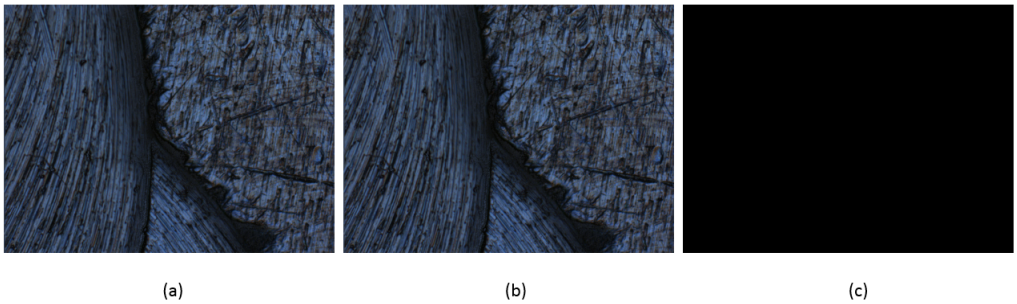


Figura 5.31: (a) imagen correspondiente a la técnica DE+Kirsch generada usando la CPU, (b) DE+Kirsch usando la GPU, (c) resta de las imágenes.

En la figura 5.32(c) se observa la diferencia entre la imagen fusionada DE+Kirsch (a)

obtenida en la GPU y una imagen obtenida modificando la máscara de Kirsch empleada, esta prueba sirve solo como referencia para ejemplificar cual sería el resultado en caso de que las imágenes no fueran iguales.

Figura 5.32: (a) imagen correspondiente a la técnica DE+Kirsch generado usando la CPU, (b) DE+Kirsch empleando la máscara correspondiente a E (0°), (c) resta de las imágenes.

Bibliografía

[1] Li, S., Kwok, J.T., Wang, Y., "Combination of images with diverse focuses using the spatial frequency,"*Journal of Information Fusion*, 2(3), pp. 169-176, 2001.

[2] A. Padilla-Vivanco, I. Tellez-Arriaga, C. Toxqui-Quitl, C. Santiago-Tepantlan, "Multifocus microscope color image fusion based on Daub(2) and Daub(4) kernels of the Daubechies wavelet family", *Proc. SPIE: Applications of Digital Image Processing XXXII*, Vol. 7443, pp. 1-7 (2009).

[3] Konstantinos N. *Color Image Processing and Applications*, Springer, Primera Edicion, pp. 197-181, (2000).

[4] R. Gonzalez, and P. Woods, *Digital Image Processing*, Addison Wesley, Segunda Edición, 2002.

[5] Kirsch, R., "Computer determination of the constituent structure of biological images", *Computers and Biomedical Research*, 315-328, (1971).

[6] J. C. Valdiviezo-N., J. Hernández-Tapia ; L. Mera-González, C. Toxqui-Quitl, A. Padilla-Vivanco; "Autofocusing in microscopy systems using graphics processing units", *Proc. SPIE: Applications of Digital Image Processing XXXVI*, Vol. 8856, pp.1-9 (2013).

Capítulo 6

Conclusiones Generales

El desarrollo del presente trabajo de tesis tuvo como objetivo lograr una fusión en imágenes multifoco adquiridas por microscopio que poseen diferentes planos de enfoque, buscando lograr una imagen con una calidad visualmente aceptable en el menor tiempo de cómputo posible.

Para lograr dicho objetivo, se realizaron pruebas con diversas métricas de enfoque que han mostrado resultados notables en tiempos de cómputo y exactitud. Dichas métricas fueron implementadas en un *Unidad de Procesamiento Gráfico* GPU, con el propósito de emplear dicha unidad como procesador principal en lugar de la *Unidad Central de Procesamiento* (CPU). La elección de la GPU como unidad principal de procesamiento se debe en gran medida a que ésta es capaz de realizar varias actividades y procesos de forma simultánea, que se traduce en un cómputo paralelo. De igual forma se analizó la tecnología CUDA que corresponde al compilador y las herramientas necesarias para desarrollar programas en una GPU de NVIDIA.

Adicionalmente se analizaron las técnicas usuales de fusión de imágenes y se observo que, las mismas van orientadas primordialmente a imágenes en escala de grises, y en la mayoría de los casos se emplean imágenes cuyo defoco es generado de manera artificial. Asimismo se concluyó que la fusión de imágenes basada en subdivisión por bloques es una técnica poco conveniente para imágenes adquiridas por microscopio dada la forma antisimétrica e irregular de la mayoría de los objetos una vez amplificados. De la mano de este análisis se concluyó que la misma técnica de división por bloques no es conveniente para su implementación en GPU puesto que en una imagen de 5Mpx los tiempos de cómputo se incrementan en la GPU hasta

35× respecto de la CPU. De lo anterior se concluyó que las ténicas basadas en división por bloques no son convenientes al menos para su implementación en una GPU.

Asimismo otro problema que se estudió va encaminado al hecho de que las imágenes empleadas en microscopía se manejan en el espacio de color RGB, a diferencia de la mayoría de los trabajos que existen donde las imágenes empleadas corresponden a la escala de grises. En otros casos la técnica de fusión en imágenes de color solo corresponde a una extensión de la técnica en escala de grises al caso de color, manejando cada canal de forma independiente.

Para finalizar se concluyó que lo mejor para implementar un algoritmo de fusión en la GPU, es una fusión a la que es posible denominar como "pixel a pixel", ya que se evalúa cada vector de color de una imagen en el espacio de RGB, en lugar de analizar cada canal por separado. Estás técnicas llevan a una reducción dramática en los tiempos de cómputo (hasta 1000 veces más rápido) gracias al uso de una GPU como unidad principal de procesamiento, lo que se traduce en una posible aplicación muy cercana a un procesamiento en tiempo real. La primera de estás técnicas se basa en la **distancia euclidiana** trabajando cada pixel de color como un vector. La técnica mide la similitud entre dos vectores (pixeles vecinos) de color, retornando al final una imagen consistente en los bordes o detalles finos de la imagen. Esta imagen permite identificar los pixeles más importantes de cada uno de los planos adquiridos de la imagen a fusionar. Por otro lado está técnica se fortaleció al implementar como un procesado previo las **máscaras de detección de bordes de Kirsch**, estas máscaras funcionan de igual forma como un detector de bordes con la característica de poder resaltar en una de las ocho direcciones cardinales o en todas si así se desea, aplicando a las imágenes de interés un procesado previo que acentúa los detalles finos (o bordes) de la imagen. Estas dos técnicas en conjunto proporcionan buenos resultados en tiempos de cómputo y en calidad visual de la imagen.

Por otro parte la segunda técnica se basa en medir el ángulo espectral de las imágenes a fusionar trabajando de igual forma cada pixel de forma independiente como un vector de color acorde con la ecuación 5.3 expuesta en el capítulo 5, dicha métrica devuelve de forma angular (0°-360°), la similitud entre dos pixeles vecinos y de igual forma se emplea para identificar los pixeles más importantes de cada uno de los planos adquiridos. Está técnica presenta resultados en tiempos de computó hasta 50 % menor respecto de la distancia euclidiana, pero la calidad

visual se ve disminuida.

6.1. Conclusiones particulares

La fusión de imágenes, enfocada a imágenes adquiridas por microscopio, es un tema que se encuentra poco explorado; en su mayoría los trabajos y técnicas actuales se enfocan principalmente a objetos macroscópicos. Del capítulo 1 se aprecia que los trabajos existentes orientados a microscopía presentan bajos niveles de contraste y pérdida de color, de ahí la necesidad de buscar otra alternativa a la fusión de imágenes.

Por otra parte, los algoritmos que permiten medir el contraste que se revisan en el capítulo 3, requieren de una modificación para poder ser implementados de forma paralela. Esto debido en gran medida a la nula productividad del uso de ciclos for en una tarjeta gráfica, lo que conllevó un análisis de estas métricas para resolver esta situación. Al resolver dicho problema las métricas se lograron implementar de forma exitosa en una GPU, logrando disminuir considerablemente los tiempos de cómputo en varias técnicas. Por ejemplo en *Vollath-4* se logró una reducción de tiempo del 63 %, de estas pruebas se concluye que la GPU es factible como unidad de procesamiento.

De igual forma se tuvo que elegir un lenguaje de desarrollo para la implementación de dichas técnicas, en el capítulo 4 se discute la programación en paralelo empleando como lenguajes de desarrollo tanto C/C++ como MATLAB. Se discutieron ventajas y desventajas del uso de ambos lenguajes, llegando a la conclusión de que MATLAB es radicalmente más rápido que C/C++ (hasta 2,8×), pese a que MATLAB es lenguaje interpretado, por el contrario C/C++ es un lenguaje compilado. Se estima que dicha situación es debido en gran parte a que MATLAB es un software optimizado y diseñado para procesamiento de señales.

Finalmente, de las técnicas que se implementaron en el capítulo 5 se llega a la conclusión de que la fusión de imágenes por bloques presenta, además de elevados tiempos de cómputo en la GPU, la aparición de "artefactos" rectangulares debido a la forma no simétrica de las muestras de microscopía, así como una marcada perdida de contraste en las imágenes. Por otra parte, las métricas analizadas en dicho capítulo trabajan cada pixel como un vector de

color en el espacio de color RGB lo que demuestra que la mejor forma de trabajar una imagen en color para su fusión es trabajando un pixel en sus tres componentes de forma simultanea y no de forma independiente. La técnica basada en el ángulo espectral ha demostrado ser computacionalmente más efectiva en todos los casos, disminuyendo su tiempo de cómputo respecto a la métrica de distancia euclidiana con máscaras de Kirsch hasta en un 45 % en el caso de la implementación de la CPU y una disminución de entre el 30 y 40 % en el caso de la implementación en GPU. Sin embargo en términos de la mejor imagen resultante la métrica basada en la distancia euclidiana en conjunto con las máscaras de Kirsch logra el mejor resultado en todos los casos probados, puesto que la diferencia de tiempo en la implementación en GPU entre la distancia euclidiana con la máscara de Kirsch y el ángulo espectral, es de sólo 6,67 segundos en la muestra con más imágenes que se analizó en este trabajo se concluye que la distancia euclidiana con las máscaras de Kirsch es una excelente alternativa para la fusión de imágenes de microscopía ya que proporciona una imagen resultante aceptable de forma visual, y además presenta un tiempo de cómputo bajo.

Para finalizar, se concluyó que la GPU en general, es una excelente alternativa como unidad principal de procesamiento. La programación en paralelo requiere una lógica nueva respecto a la lógica secuencial para poder desarrollar aplicaciones en un GPU. Asimismo las limitaciones lógicas de la GPU requieren un análisis detallado y la descomposición de algunos de los métodos presentados en este trabajo de tesis para su correcta ejecución en paralelo. Más importante aún "no todo problema es paralelizable" (léase el caso de la fusión por bloques) pues algunos programas requieren forzosamente ser ejecutados de forma secuencial.

De igual forma de las tres métricas analizadas en este trabajo de tesis (DE, DE+Kirsch y AE), se concluye que la mejor técnica evaluada corresponde a la Distancia Euclidiana en conjunto con las máscaras de Kirsch (DE+Kirsch). Dicha conclusión queda sustentada en las pruebas realizadas, puesto que las métricas de contraste propuestas concluyen de forma terminante en todos los casos que la técnica antes mencionada es la que mejores resultados brinda.

6.2. Productos derivados de la investigación

Derivado de este proyecto de tesis se participó en las siguientes reuniones académicas:

- 7mo. Congreso Universitario en Tecnologías de Información y Comunicaciones 2012, Universidad Autónoma del Estado de Hidalgo, 3, 4 y 5 de Octubre 2012.
- SPIE Optics + Photonics 2013, San Diego, California, USA, 25-29 Agosto 2013.

Con la participación en dichos congresos se logró la publicación de las siguientes memorias en extenso:

- J. Hernández Tapia, J. C. Valdiviezo-N, C. ToxquiQuitl; "Fusión de Imagenes Usando Frecuencia Espacial en Imágenes de Microscopía", *7mo. Congreso Universitario en Tecnologías de Información y Comunicaciones 2012*, pp. 1-6 (2012).
- J. C. ValdiviezoN., J. HernándezTapia ; L. MeraGonzález, C. ToxquiQuitl, A. Padilla-Vivanco; "Autofocusing in microscopy systems using graphics processing units", *Proc. SPIE: Applications of Digital Image Processing XXXVI*, Vol. 8856, pp.1-9 (2013).

6.3. Trabajo a Futuro

Como investigación posterior a esté trabajo de tesis se pretende analizar otro tipo de detectores de bordes, con el objeto de mejorar los resultados obtenidos empleando las máscaras de Kirsch, así mismo se pretende analizar un número mayor de muestras biológicas, de igual forma se pretende combinar la técnica de ángulo espectral con algún detector de bordes diferente de las máscaras de Kirsch ya que los resultados preliminares combinando estas técnicas presentan resultados poco favorecedores en la calidad de la imagen resultante.

De igual forma se pretende buscar una futura implementación en hardware, que permita a un microscopista obtener una imagen que le permita observar todos los planos que sean de interés de una muestra tan pronto sean adquiridas las imágenes.

Apéndices

Apéndice A

Códigos Fuente Capítulo 4

El presente capítulo contiene los códigos fuentes correspondientes al capítulo 4 donde se discuten los principios de la programación en paralelo en GPU usando CUDA. Los códigos fuente se pueden ejecutar en cualquier versión del Microsoft Visual Studio superior a la versión 2008 y una versión de CUDA superior a la 2.0.

A.1. Código fuente: Suma de Vectores en CPU usando C++

El presente código fuente ejecuta la suma de dos vectores a, b de tamaño N para almacenarlos posteriormente en un tercer vector c, cabe mencionar que la ejecución de este código fuente es secuencial y corresponde a la implementación en CPU.

```
#include "stdio.h"
#include "conio.h"
#define N   10

void add( int *a, int *b, int *c ) {
        int i = 0;
        // Necesitamos un ciclo para recorrer cada
```

```
    //posicion lo inicializamos en 0
        while (i < N) {
                c[i] = a[i] + b[i];
                i += 1;
    //Incrementos de 1 para Recorrer Cada Posición
    //del vector
        }
}
int main( void ) {
        int a[N], b[N], c[N];
        //Llenamos los Arreglos a,b
        for (int i=0; i<N; i++) {
                a[i] = i;
                b[i] = i * i;
        }
    add( a, b, c );
        //Mostramos los Resultados
        for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
        }
        _getch();
        return 0;
}
```

A.2. Código fuente: Suma de Vectores en GPU usando CUDA y C++

```
#include <stdio.h>
```

```
#include <conio.h>
#define N   10

__global__ void add( int *a, int *b, int *c ) {
    int tid = blockIdx.x;
// Este hilo maneja la información como su threadId
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}
//Función Principal

int main( void ) {
    int a[N], b[N], c[N];
    int *dev_a, *dev_b, *dev_c;

    // Reservamos memoria en la GPU
cudaMalloc( (void**)&dev_a, N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_b, N * sizeof(int) ) ;
cudaMalloc( (void**)&dev_c, N * sizeof(int) ) ;



    // llenamos los arreglos a,b en el CPU
    for (int i=0; i<N; i++) {
        a[i] = i;
        b[i] = i * i;
    }
    // Copiamos los arreglo a y b de la CPU a la GPU
    cudaMemcpy( dev_a, a, N * sizeof(int),
                                cudaMemcpyHostToDevice ) ;
```

```
    cudaMemcpy( dev_b, b, N * sizeof(int),
                                cudaMemcpyHostToDevice ) ;

    add<<<N,1>>>( dev_a, dev_b, dev_c );
    // Copiamos el arreglo c de la GPU a la CPU
     cudaMemcpy( c, dev_c, N * sizeof(int),
                                cudaMemcpyDeviceToHost ) ;
    // Mostramos los resultados utilizando un ciclo for
    for (int i=0; i<N; i++) {
        printf( "%d + %d = %d\n", a[i], b[i], c[i] );
    }
        _getch();
    // Liberamos la memoria almacenada en la GPU
    cudaFree( dev_a );
    cudaFree( dev_b );
    cudaFree( dev_c );
    return 0;
}
```

La parte principal del código fuente se discutió previamente en el capítulo 4, la intención de estos códigos es presentar las diferencias entre la programación "estándar" en CPU y la ´programación paralela en GPU.