Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Un enfoque híbrido del algoritmo *Inpainting* bajo CPU y GPU

Trabajo Especial de Grado
presentado ante la Ilustre
Universidad Central de Venezuela
Por la Bachiller
Karina G. Pedrique R.
para optar el título de
Licenciado en Computación

Tutor: Prof. Esmitt Ramírez
Co-Tutor: Prof. Reinaldo Astudillo

Caracas, 27 de Septiembre de 2011

Agradecimientos

Quiero agradecer en primer lugar a ese ser supremo que llamamos Dios, que me ha acompañado y protegido siempre, que cuando iba asustada a las 9 de la noche subiendo a mi casa me daba protección. Que siempre me ha mostrado la verdad y que en mis peores momentos me ha dado fuerzas para continuar y no dejar desmonorarme por situaciones de vida, y que por el contrario me ha hecho más fuerte. A veces pasan cosas que al momento no entendemos porque suceden, pero cuando pasa el tiempo y ocurren eventos que nos satisfacen de gran manera, nos damos cuenta que fueron necesarios esos momentos que no entendíamos. Estoy agradecida con Dios por todos aquellos momentos que no entendí, en los cuales sentí que la vida era injusta, pero que en cierta forma influyeron positivamente en el presente y que hacen que escriba estos agradecimientos con la satisfacción que di lo mejor de mí para este trabajo especial de grado.

A mi familia, que siempre me han apoyado. Mi familia es numerosa, pero nunca por más lejos que estemos uno del otro hemos dejado de amarnos o darnos apoyo. Mi madre querida, mi mejor amiga, que aunque no entendía muy bien lo que estaba haciendo, siempre me decía *el tiempo de Dios es perfecto, y todo ocurre para mejor*. Mi madre que cuando iba a su casa, hablábamos un poquito nada más y me atendía para que me concentrara solo en la tesis y me hacía mi arepita con huevito que tanto me gusta. Al padre de mi alma, que siempre tenía palabras de aliento para mí, que siempre me ha dado su apoyo incondicional. A mis hermanos, que fueron muy pacientes y entendían que no podía compartir con ellos del modo al que estaban acostumbrados. A mi abuelita, que es mi segunda madre, que cuando yo decía no se si pueda, ella me decía *no es creo que puedo, es que tú puedes* y que quería siempre hacerme sentir segura de mí misma. A mi tía Vigui y mi tío Juan, que me escuchaban y que siempre me han tratado como un hija. Mi adorada tía Yuni, que aparte de darme todo su apoyo, y motivarme siempre, me ha ayudado a formarme como profesional (siendo en su empresa y con la enseñanza de mi padre que aprendí a ensamblar computadoras y una primera vista del área de redes de computadoras) y que siempre pensó en mí cuando veía oportunidad de trabajo en época de vacaciones, que es una oportunidad que no tenemos todos y que la tuve desde el segundo semestre de esta carrera, mi tía que siempre vió potencial en mí. Mi Yeri linda, que escuchó mi exposición de tesis muchas veces y me ayudó a practicar, que me ayudó a escribir mi primer borrador de documento, que me apoyó de una manera muy especial los días más críticos y que me daba mucho ánimo. A Gaudy, que siempre estuvo pendiente de mí, que si no es por ella, nos quedamos sin pasapalos y que siempre cuida mucho la salud de mi papi y su alimentación y me hacía sentir tranquila que mis hermanas y

mi papi estaban en buenas manos. Bueno, si sigo nombrándolos a todos, esto me va a llevar muchas páginas, pero a quienes nombro y a quienes faltó por nombrar saben que los adoro con todo mi corazón y solo el hecho de saber que estaban apoyándome y que me quieren tuvo una gran influencia en mí, porque son importante en mi vida, porque Dios no me pudo dar una familia más maravillosa, pues no existe, son ustedes.

La Universidad Central de Venezuela, un lugar que conozco de pequeña cuando mi papá me llevaba a sus clases de Arquitectura, que cuando estudiaba bachillerato y pensaba en la universidad, pensaba en la UCV, que aunque comencé la vida universitaria en otro lugar, retorné al lugar que siempre estaba en mi mente y fue lo mejor que me pudo suceder, porque creo que lo que voy a sentir en el Aula Magna de la UCV, es una experiencia que no tiene otro lugar. Nunca usé una toga, ni un birrete, me saltaron de segundo nivel a primer grado y en el colegio cuando me gradué de bachiller, solo usé mi uniforme. Pero está bien que haya sido así, porque mi primera toga y mi primer birrete va a ser en en esa Aula Magna de mi casa de estudios, mi adorada UCV. Al Centro de Computación Gráfica, donde trabajo como Auxiliar docente y todos los profesores de una u otra forma me ayudaron, donde tuve todo el equipo necesario para desarrollar mi tesis, y el lugar donde pasé más horas que en ningún otro lado trabajando, y en el cual aprendí muchas cosas nuevas y que le doy gracias por darme la oportunidad de trabajar allí. A mi tutor Esmitt, que cuando me quedé sin tema de tesis, se sentó conmigo a buscar uno, que es el que estoy presentando con este documento, que me apoyó muchísimo y que junto al profesor Reinaldo me ayudaron a descifrar dudas matemáticas que se presentaron en el transcurso de la investigación. Al profesor Ernesto que me ayudó a comprender que es eso de hablar de energía en una imagen. Al profesor Héctor que me aclaró puntos importantes de las probabilidades y las variables aleatorias. Al profesor Rhadamés, que me dió recomendaciones y documentación del espacio del color en el cual trabajar. Al profesor Robinson que cuando todo me explotó en CUDA justo antes de terminar la versión paralela, me dió recomendaciones de como detectar el error y que muchas veces me dió la cola hasta el metro cuando me quedaba tarde trabajando en la tesis. En fin, profesionales que me daban valiosas ideas y a los cuales estimo mucho.

En la Facultad de Ciencias, hice mis mejores amigos y viví mis mejores momentos, mis amigos que han estado allí siempre. Daniel Romero, que siempre escuchaba específicamente como planeaba codificar algo, y aunque me decía *Kari estás loca* ó *Kari ya está bien, por favor escribe*, siempre estuvo apoyándome y siempre me daba su opinión constructiva. A mi amiga Vanessa y Jhon, que siempre me daban café *XD* y me ayudan a salir un poco del estrés. A Jorge, que junto a Vanessa, Jhon y Daniel, vieron 960 cookie monsters y les dieron una calificación, ayudándome en esta prueba tan importante en mi tesis. A Larry que siempre me cuidaba y no le gustaba que estuviera de noche sola y me enseñó otras maneras de trabajar sin arriesgarme tanto y que siempre me dió mucho ánimo, que siempre me dijo *poco a poco todo te va a salir bien*. A Pablo que me sacó varias veces la pata del barro con LaTeXy que me ayudó con un error de sincronización de los hilos con CUDA. A todos mis amigos, Iris, Roberto, Alejandro, Adriana, Edgar, Kijam, Draumari, Pedro, Astrid, Dayana, Jesús, Anais y otros *XD*.

Hay gente que quizás su intención en principio no fue precisamente ayudarme, sino todo lo contrario, pero que sus acciones influyeron positivamente. En primer lugar a la directora

de computación de la USB, quien me dijo que yo no merecía el cambio de carrera y que no hiciera una segunda solicitud, porque sencillamente no me la iba a dar. Gracias a su acción negativa, yo positivamente terminé estudiando mi carrera y mi especialidad en el lugar indicado. En segundo lugar, a mi ex-compañera de tesis, que cuando me puso contra la espada y la pared, viéndome obligada a terminar renunciando a mi tesis anterior, me empujó a ver una realidad, *yo suelo ver en las personas lo que quiero que sean y no lo que son*. Y por otro lado, positivamente escribo hoy mis agradecimientos en este trabajo especial de grado que me ha dejado mayor satisfacción.

Que viva la UUUCV!!!!

Karina Pedrique

Dedicatoria

A todos los niños del mundo, que necesitan educación, amor y protección para hacer un mundo mejor.

Resumen

En el procesamiento digital de imágenes se aplica un conjunto de técnicas que tienen como objetivo mejorar la calidad de la imagen o extraer algún tipo de información. Entre las técnicas utilizadas se encuentran la segmentación, corrección de imagen, análisis de la imagen, reconstrucción de partes perdidas, entre otras. En la reconstrucción de imágenes una de las técnicas empleadas se denomina *Inpainting*, la cual modifica una imagen para reconstruir áreas deterioradas o eliminadas de ésta. En la actualidad, las técnicas de Inpainting ocupan un amplio campo de investigación y desarrollo en el área de la Computación Gráfica. En este trabajo se plantea la implementación de un novedoso algoritmo de *Inpainting*. Nuestro algoritmo se denomina *k-Inpainting*, el cual está basado en iteraciones para lograr la reconstrucción total de la imagen. El algoritmo *k-Inpainting* se presenta como una nueva técnica híbrida ya que une dos enfoques distintos de investigación. Adicionalmente se desarrolla una versión secuencial (CPU) y dos variantes en la GPU para su versión en un ambiente de alto desempeño paralelo. Se realizaron diversas pruebas variando todos los parámetros de entrada del algoritmo, con el fin de obtener los valores óptimos. En los experimentos realizados, se obtuvieron resultados satisfactorios.

Palabras Claves: *Inpainting*, reconstrucción de imágenes, síntesis de textura, coherencia, difusión, CUDA.

Tabla de Contenidos

Lista de Figuras

Lista de Tablas

Introducción

La restauración de imágenes consiste en restituir zonas "perdidas" de una imagen de una forma que no sea detectada a simple vista. Como ejemplo de ello, se encuentra restaurar una fotografía antigua que ha sido digitalizada, remover artefactos notables, remover objetos, reconstruir áreas de un objeto específico dentro de una imagen, entre otros. Aparte de una técnica, es un arte y una práctica tan vieja como la creación artística en sí misma.

Actualmente, los restauradores profesionales de fotografías están reemplazando los métodos manuales por los digitales, y la técnica de *Inpainting* es una de ellas. Sin embargo, esta conlleva un problema muy costoso computacionalmente desde el punto de vista de procesamiento. El proceso consiste en aplicar diversos procedimientos matemáticos de alto nivel que sean estables respecto a las técnicas numéricas empleadas, que permitan reconstruir la zona de interés de forma que no sea fácilmente detectable por un observador. Debido a su complejidad se ha convertido en un campo amplio de investigación.

En este documento se presenta un novedoso enfoque híbrido que une 2 trabajos recientes con perspectivas distintas: *A comprehensive framework for image inpainting* [15] y *The generalized patchmatch correspondence algorithm* [9]. Nuestra propuesta se basa en 6 etapas bien definidas, que serán explicadas en detalle. En ella, se busca experimentar y proponer un algoritmo con enfoques diferentes a los existentes, que logre el objetivo de la técnica. Todo ello, tomando en cuenta que la calidad visual del resultado es primordial para nuestra propuesta. La intervención del usuario es mínima, siendo casi automática en su proceso.

El Capítulo 1 describe el algoritmo y las técnicas aplicadas en cada etapa. Al mismo tiempo, se contemplan conceptos básicos y se describen aspectos importantes de trabajos realizados bajo este paradigma. Luego, en el Capítulo 2 se específica la propuesta de la variante de algoritmo *Inpainting* llamada *k-Inpainting*. El algoritmo propuesto es costoso a nivel computacional; por ello en el Capítulo 3 se proponen dos variantes paralelas del algoritmo para agilizar el tiempo de respuesta del proceso de restauración. Finalmente, en el Capítulo 4 se muestran y analizan los resultados obtenidos a partir del enfoque planteado.

Capítulo 1

Inpainting

El proceso *Inpainting* se refiere a la restauración de imágenes mediante la aplicación de una o más técnicas numéricas, que permiten recuperar información en una imagen. En este capítulo se dará a conocer en qué consiste dicho proceso se realizará una revisión del estado-del-arte de los métodos empleados para su aplicación.

1.1. Definición

Es el arte de modificar una imagen o video de una forma que no sea fácilmente detectable por un observador ordinario, y ha sido un área fundamental de la investigación en el procesamiento digital de imágenes [15]. La palabra *Inpainting* ha sido un término genérico empleado para el proceso de restauración de partes dañadas o perdidas de una imagen. El problema puede ser descrito como sigue: Dada una región Ω a ser restaurada, se emplea la información conocida que rodea dicha región, para regenerar de la manera más aceptable los datos en Ω [16]. Grosso modo, el algoritmo de *Inpainting* debe buscar el mejor píxel q en Ω^c que sustituya el valor desconocido de un píxel p en Ω, tal como se muestra en la Figura 1.1.

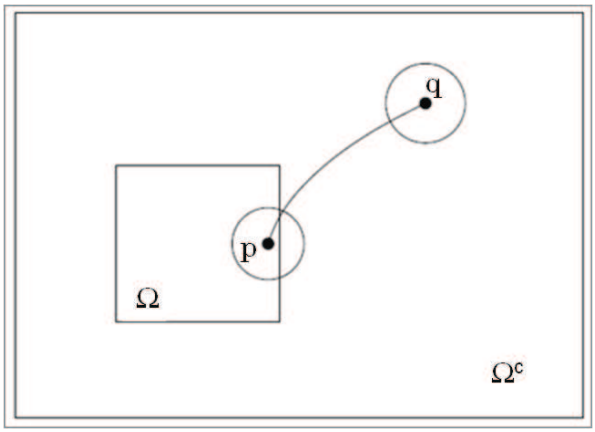


Figura 1.1: Objetivo del algoritmo *Inpainting* [1].

Un aspecto importante a resaltar es que los clásicos algoritmos de eliminación de ruido no se aplican a la técnica de *Inpainting*. Las diferentes aplicaciones de procesamiento digital de imágenes que se encargan de mejorar una imagen, tienen en común que los píxeles contienen información acerca de los datos reales y el ruido, mientras que en *Inpainting*, no hay información significativa en la región a ser reconstruida. La información se encuentra principalmente alrededor de las áreas a ser tratadas [17]. En consecuencia, han desarrollado técnicas específicas para estudiar este problema, las cuales serán detalladas en este capítulo. Antes de detallar las diferentes técnicas de inpainting es necesario definir algunos el término de sistema de vecindad, ya que en muchos trabajos de investigación relacionados, así como el planteado en este documento, se utiliza dicho sistema.

1.2. Sistema de vecindad

Los sistemas de vecindad permiten extraer la información que se encuentra alrededor de los píxeles a ser tratados. Dentro de este sistema se emplea un término denominado localidad S. Un conjunto de localidades pueden ser categorizadas en relación a su *regularidad*. Todas las localidades en una matriz se consideran de forma regular [2]. Una matriz regular para una imagen 2D de dimensiones $n \times n$ puede ser denotada como:

$$S = (i, j)|0 \leq i, j < n \tag{1.1}$$

La relación entre las localidades es mantenida mediante un sistema de vecindad [2]. Un sistema de vecindad para S es definido como:

$$N = \{N_i|\forall_i \in S\} \tag{1.2}$$

Donde N_i es el conjunto de localidades vecinas a i. La relación de vecinos tiene las siguientes propiedades:

- Una localidad no es un vecino de sí mismo: $i \notin N_i$
- La relación de vecindad es mutua: $i \in N_{i'} \Leftrightarrow i' \in N_i$

Para una matriz regular, el conjunto de vecinos de i es definido como el conjunto de posiciones dentro de un radio $\sqrt{r}$ desde i, tal como se observa en la ecuación 1.3

$$N_i = \left\{i' \in S|[dist(pixel_{i'}, pixel_i)]^2 \leq r, i' \neq i\right\} \tag{1.3}$$

Donde dist($pixel_{i'}$, $pixel_i$) se refiere a la distancia Euclidiana entre $pixel_{i'}$ y $pixel_i$, y r toma un valor entero. Note que las posiciones sobre o cerca de las fronteras tienen menos vecinos. En el sistema de vecindad de primer orden, también llamado sistema de 4 vecinos (Figura 1.2(a)), x es la localidad a tratar y los ceros sus vecinos. En el sistema de vecindad de segundo orden, también llamado sistema de vecindad de 8 vecinos, hay 8 vecinos como lo muestra la Figura 1.2(b). Los números $n = 1, \ldots, 5$ mostrados en la Figura 1.2(c) indican vecindades de posiciones periféricas en el sistema de vecindad del n-ésimo orden. La forma

de un conjunto de vecinos puede ser descrita como una región cerrada que encierra a todas las posiciones o sitios en dicho conjunto [2].

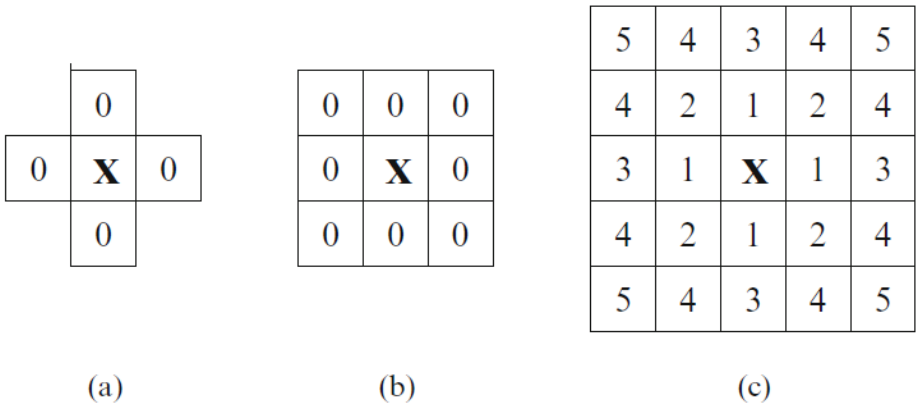


Figura 1.2: Vecindades sobre matrices regulares [2].

Los sistemas de vecindad empleados comúnmente por las técnicas de *Inpainting* emplean sistemas de vecindades de segundo o n-ésimo orden.

1.3. Técnicas de *Inpainting*

El proceso de *Inpainting* surge a partir de la importancia de restaurar y modificar imágenes y videos, pero también su uso es empleado para entender la validez de diferentes modelos de imagen (por ejemplo modelos de imagen estocásticos o determinísticos). Según [15], considerando estos modelos de imágenes, la técnica de *Inpainting* puede ser dividida en tres grupos básicos: auto-similitud y síntesis de textura; difusión y propagación (basada en emplear ecuaciones diferenciales parciales de orden superior); y coherencia. Además, se puede incluir un nuevo grupo según el trabajo mostrado en [18], donde se contemplan métodos basados en la continuación de bordes. Adicionalmente, existen diversos trabajos que combinan técnicas y crean nuevos algoritmos para *Inpainting*. En esta sección se presentan diversas investigaciones que estudian los métodos antes mencionados.

1.3.1. Auto-similitud y Síntesis de textura

Una textura en computación gráfica es definida como un mapa de bits que contiene un número finito y discreto de muestras de alguna imagen original [19]. También puede definirse como un conjunto de píxeles que se encuentran sobre un espacio 2D finito que muestran características de superficies tales como terrenos, plantas, minerales, pelaje, piel, etc [20]. Estas texturas pueden ser obtenidas de diferentes fuentes, sin embargo, algunas pueden resultar inadecuadas en el mapeo de textura. La síntesis de textura es otro mecanismo para crear texturas, el cual es conveniente debido a que se pueden originar texturas sintéticas de cual-

quier tamaño, evitando repeticiones visuales. Además la síntesis de textura puede producir imágenes de alta definición por el manejo adecuado de los bordes.

El problema de síntesis de textura puede ser definido como sigue: Dada una muestra de textura, se sintetiza una nueva textura que cuando sea percibida por un observador, parezca ser generada por el mismo proceso estocástico subyacente (ver Figura 1.3).

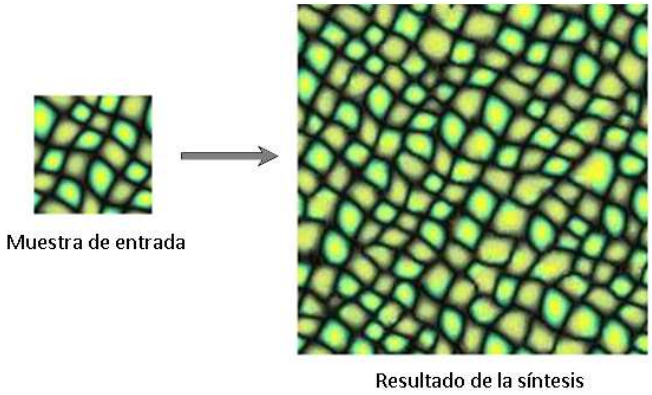


Figura 1.3: Resultado visual luego de sintetizar una muestra de textura de entrada [3].

La síntesis de textura ha sido un tópico activo de investigación empleado como mecanismo para verificar el método de análisis de textura. Contribuye exitosamente en una serie de aplicaciones como la restauración de partes perdidas de imágenes y video, remoción de *foreground*, entre otros. Hay dos formas básicas: Una textura puede ser sintetizada basada en píxeles o basada en parches. Ambas serán resumidas a continuación.

Síntesis de textura basada en píxeles

Diversos métodos sintetizan una textura haciéndola crecer de adentro hacia afuera píxel por píxel, a partir de una semilla inicial [21] o replicando un píxel a la vez [18]. Esta forma de sintetizar la textura fue introducida por Efros y Leung en su trabajo *Texture Synthesis by Non-Parametric Sampling* [21]. Para este tipo de síntesis, la textura es modelada como un Campo Aleatorio de Markov, asumiendo que la probabilidad de distribución del valor del brillo para un píxel, proporcionado por los valores de brillo de su vecindad espacial, es independiente del resto de la imagen. El método ayuda a preservar la estructura local y produce buenos resultados para una amplia variedad de texturas sintéticas y del mundo real. Según lo expuesto en [18], a pesar de ser un método extremadamente versátil reproduciendo texturas estocásticas y determinísticas, puede ser lento creando grandes texturas.

Aunque cada trabajo realizado bajo este concepto [21, 20, 22, 7] tiene diferentes aportes (que serán descritos posteriormente), todas las investigaciones poseen en común las siguientes características:

- Adoptan un campo aleatorio de Markov, es decir usan un proceso local y estacionario.

- Realizan muestreos sobre la imagen.
- A partir de una semilla inicial, seleccionan una textura de entrada, píxeles que tengan vecindades similares.

En ciertas ocasiones, se generan artefactos visuales, tal como lo reportan Efros y Leung [21]. En [23] se argumenta que estos artefactos se crean por dos razones: por el artefacto anterior (existente en la imagen o formado en una iteración previa), ya que un muestreo no paramétrico con vecindades parciales de un píxel puede no ser suficiente para capturar estructuras a gran escala de una muestra de textura. Por otro lado, si se usa un tamaño de ventana adecuado, éstas estructuras pueden ser captadas. Sin embargo, para texturas que contienen grandes estructuras se necesitan usar grandes sistemas de vecindad que pueden requerir mayor demanda computacional. Además, el tamaño de la ventana, probablemente sea demasiado grande para introducir la aleatoriedad suficiente.

Wei y Levoy [20] establecen que este problema puede ser solucionado utilizando un esquema de multiresolución con una pirámide de imágenes, que también exhibe el efecto de ampliar el tamaño de la ventana; ya que se buscan explotar sistemas de vecindad completos y amplios en los niveles más gruesos. Para acelerar la búsqueda de píxeles, adoptan una heurística llamada *tree-structured vector quantization (TSVQ)*. Esta heurística efectivamente acelera el proceso de búsqueda, pero también trae un efecto borroso que oculta la calidad obtenida a partir del esquema de multiresolución.

Ashikhmin [7] explota la coherencia espacial para remediar el artefacto borroso (*blurring*) que se observa en [20]. La propuesta de Ashikhmin será explicada con mayor detalle en la sección de la técnica de coherencia. Posteriormente, Hertzmann et al. [22] en su trabajo *Image Analogies* describe un *framework* para procesamiento de imágenes, el cual es aplicable para una amplia variedad de problemas de textura, incluyendo filtrado de imágenes, síntesis de textura, super resolución, transferencia de textura, entre otros. En el caso de síntesis de textura [22] combinan las ventajas de los dos métodos descritos anteriormente [20] y [7] para obtener mejores resultados. En la Figura 1.4 se comparan los métodos antes mencionados en cuanto a los resultados visuales obtenidos.

Demanet et al. [1], basado en el trabajo de Wei y Levoy [20], formaliza una solución global determinística que define un mapa de correspondencia para el método *Inpainting*, donde cada píxel *perdido* tiene un enlace o conexión con un píxel conocido de la imagen. Dada una imagen con una función de intensidad conocida $I(p)$ definida en los píxeles $p \in I \backslash \Omega$, para una cierta región desconocida Ω se busca recuperar las intensidades faltantes para $I(p)$ por cada $p \in \Omega$. Para ello se define un mapa de correspondencia $\varphi : \Omega \longrightarrow I \backslash \Omega$ que sustituya los valores de los píxeles desconocidos con el resto de la imagen $I(p) = I(\varphi(p))$ (ver Figura 1.1). El mapa φ debe ser escogido de tal forma que la región sintetizada se parezca lo más posible a las partes conocidas de la imagen. Este mapa se puede definir matemáticamente como sigue:

$$\varphi(p) = \begin{cases} p & \text{si } p \in \Omega^c \\ argmin_{q \in \Omega^c} dist(p, q) & \text{si } p \in \Omega \end{cases} \tag{1.4}$$

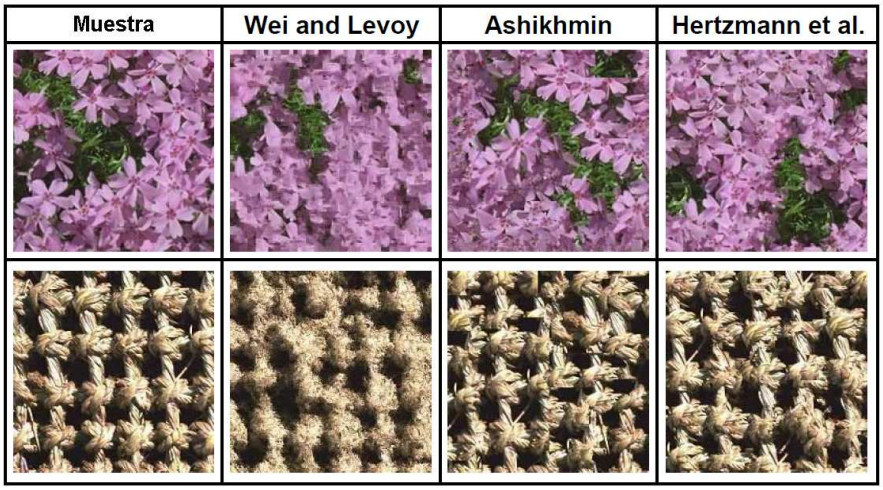


Figura 1.4: Comparación de algunos métodos de síntesis de textura basada en píxeles [4].

Demanet et al. [1] realizan diferentes experimentos y concluyen que la calidad del resultado no se degrada con sistemas de vecindades grandes (ventanas de gran tamaño). Este hecho soporta que el modelo detrás de su algoritmo sea determinístico y no probabilístico.

Síntesis de textura basada en parches

Se comienza a introducir la idea de sintetizar textura con parches mediante el trabajo titulado *Chaos Mosaic: Fast and Memory Efficient Texture Synthesis* presentado por Xu et al. [24]. En este trabajo se busca construir texturas procedurales, sintetizando grandes texturas a partir de una muestra inicial. Para ello, utilizan como base el algoritmo *chaos mosaic*, el cual permite sintetizar texturas con una distribución equilibrada y estocástica visual de las características locales de la muestra de entrada.

Efros y Freeman [5] presentan un método basado en parches llamado *image quilting*. Este método, permite sintetizar imágenes mediante la superposición de bloques cuadrados (obtenidos de la muestra de textura), de tal manera que cada bloque encaje con los vecinos semejantes en las regiones que se superponen. La Figura 1.5 detalla el proceso antes mencionado.

Liang et al. [26] propusieron un método similar. Su método aplica un algoritmo de *feathering*[1] para combinar un par de bloques. Aunque estos métodos estaban pensados para disminuir el ruido observado en los métodos que sintetizan textura basándose en píxeles, los métodos basados en parches algunas veces producen artefactos visuales tales como discontinuidades en las texturas y la copia fiel de un parche. Por otro lado, Cohen et al. [28] presentan en su trabajo *Wang tiles for image and texture generation* un método estocástico para cubrir el plano no periódico con un pequeño conjunto de restricciones sobre los bordes de los blo-

[1] Algoritmo empleado para reducir la discontinuidades de intensidad y color entre dos imágenes a ser compuestas. Para ello en vez de utilizar el algoritmo de *quilting*, se pesan los píxeles de cada imagen de manera proporcional a su distancia al borde [27]

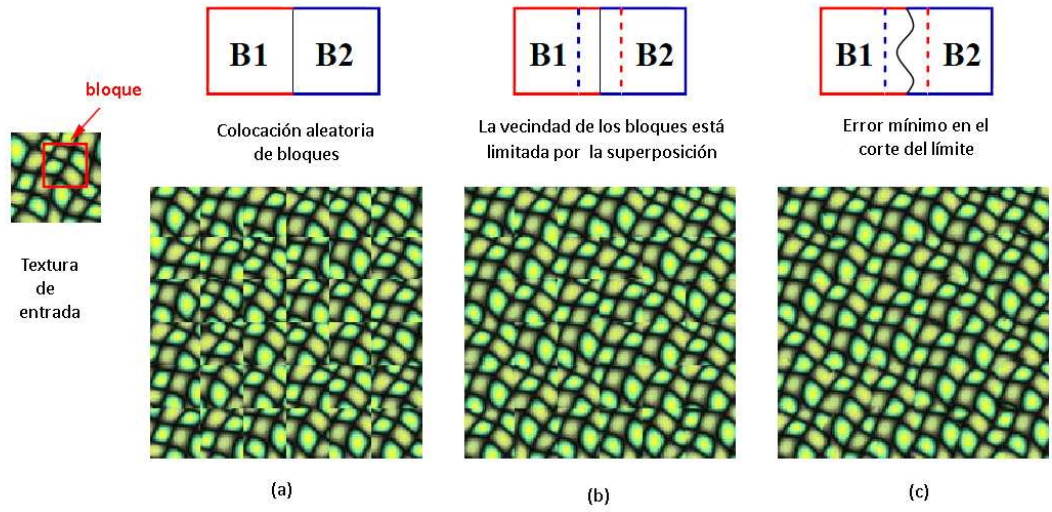


Figura 1.5: Técnica *Quilting texture* para síntesis de textura basada en parches [5]: Bloques cuadrados de la muestra entrante son parcheados juntos para sintetizar una nueva textura: (a) Los bloques son seleccionados aleatoriamente (como en [25, 20]), (b) Los bloques se solapan y cada nuevo bloque se selecciona de acuerdo a los vecinos en la región a ser solapada, (c) Entre los límites de los bloques se calcula una ruta de menor costo a través de la superficie que manifiesta errores producto de la superposición.

ques llamado *Wang tiles* [23]. En este trabajo sus autores extienden la definición de *Wang tiles* para incluir una codificación para las esquinas de los bloques a solaparse que permite sobreponer más de una arista [28].

En el trabajo denominado *Grapcut Textures: Image and Video Synthesis Using Graph Cuts* [29], se propone un nuevo método basado en parches para solucionar los problemas de los métodos [5, 26]. Este método de forma iterativa refina la calidad de las texturas sin necesidad de un conocimiento a priori del tamaño del parche. Igualmente, se demuestra un rendimiento excelente en cuanto a calidad y eficiencia. Sin embargo, es difícil encontrar los criterios adecuados para la síntesis automática de textura [23]. En la Figura 1.6 se comparan algunos de los métodos antes mencionados en cuanto a los resultados visuales.

La mayoría de los métodos basados en parches han utilizado sólo la información del color cuando se busca una ubicación adecuada de un parche [28, 5, 29, 26, 24]. Sin embargo [30] introducen un método híbrido que considera las características geométricas y de color simultáneamente para minimizar la discontinuidad visual entre parches adyacentes [30]. En los casos donde las discontinuidades generen artefactos inevitables, se remueve explícitamente la deformación del parche.

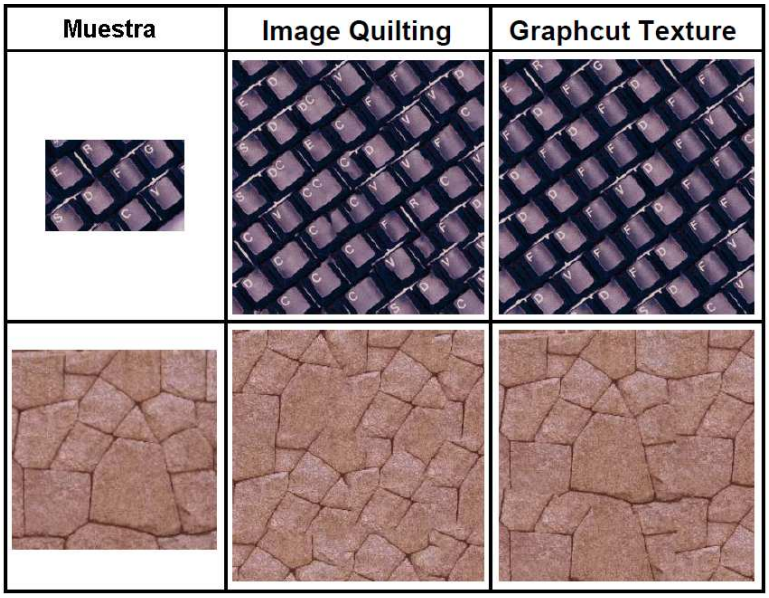


Figura 1.6: Comparación de algunos métodos de síntesis de textura basada en parches [4].

1.3.2. Difusión y propagación

Existen un grupo de algoritmos *Inpainting* basados en la difusión con ecuaciones parciales diferenciales (PDEs, por sus siglas en inglés: *partial differential equations*) y variaciones sobre las formulaciones de las mismas, las cuales han sido exitosamente usadas, particularmente para suavizar imágenes o cuando Ω es mezclado con los objetos que lo rodean en la imagen. Esta técnica de *Inpainting* surge de la necesidad de realizar *disocclusion* [31] en reconocimiento de objetos, visión robótica y restauración de imágenes y películas. Se llama *disocclusion* a la recuperación de áreas ocultas en imágenes digitales por la interpolación de su vecindad.

Sea Ω un área pequeña a ser reconstruida y sea $\delta\Omega$ su borde, el proceso de *Inpainting* puede ser aproximado a un proceso de difusión isotrópica que propaga la información desde el $\delta\Omega$ hacia Ω. La difusión isotrópica es una técnica que apunta a la eliminación del ruido en una imagen sin afectar los bordes, con la introducción de las ecuaciones de calor para el tratamiento de las imágenes. Suponiendo que se tiene una habitación con una fuente de calor en el medio, con el correr del tiempo, este calor va propagándose por la habitación en círculos concéntricos, los cuales van perdiendo intensidad a medida que se propagan. Es decir, el calor avanza alejándose de su fuente y va perdiéndose, debido a que la temperatura de la habitación tiende a homogeneizarse y equipararse a la de la fuente de calor.

Ahora, si se considera la habitación como una imagen y la fuente de calor como un punto originado por ruido aditivo, se puede imaginar como el ruido se propaga y se atenúa a través de la imagen [32]. Es decir, con esta técnica se tiene la intención de difuminar los colores en las áreas faltantes. Matemáticamente, los colores de cada píxel se promedian con una pequeña porción de su color a cada uno de sus vecinos, como se muestra en la Figura 1.7.

En el año 2000, *Inpainting* se introduce como una técnica por Bertalmio et al. [33] me-

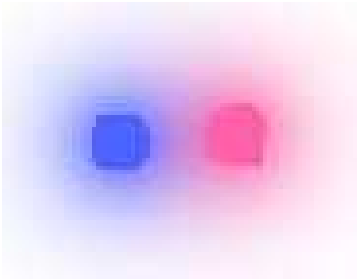


Figura 1.7: Efecto de difusión entre dos colores [6].

diante su trabajo *Image Inpainting*. En dicho trabajo, se restaura una imagen automáticamente mediante el uso de ecuaciones diferenciales parciales (PDEs) [33, 17]. El algoritmo es capaz de llenar simultáneamente pequeños agujeros, necesitando solamente la información que se encuentra alrededor de dichos hoyos. La idea básica consiste en propagar suavemente la información de las zonas circundantes al agujero en direcciones *isophotas*[2]. En su mayoría, se obtienen imágenes nítidas y sin artefactos de color, ó imágenes que pueden ser un punto de inicio para una restauración manual que requiera menos tiempo del usuario editor. Sin embargo, una de sus principales desventajas es que no puede reproducir largas regiones de texturas [33] ni reconstruir bordes agudos en el área ocluida de la imagen [18].

Luego en el trabajo mostrado en [34], se trata de mejorar las deficiencias del método *Inpainting* presentado en [33], diseñando un rápido método de *Inpainting* basado en un modelo de difusión isotrópica extendido con la noción de barreras de difusión proporcionadas por el usuario. Con el método de difusión de barreras el proceso de difusión se detiene sobre cierta posición predefinida. Este método es extremadamente rápido, pero no reconstruye adecuadamente grandes áreas y requiere una intervención previa del usuario.

Por otro lado, Chan y Shen [35, 36] presentan dos métodos diferentes de *Inpainting*. El primero basado en una aproximación variacional que hace uso de una ecuación diferencial de segundo orden. El segundo toma como referencia al anterior para crear un modelo basado en una ecuación parcial diferencial de tercer orden [35]. Tal modelo es llamado *curvature-driven diffusions*, donde la cantidad de difusión aplicada está basada en la cantidad de curvas *isophotas* en ese punto. Este trabajo mejora el algoritmo presentado en [33], ya que evita que el suavizado de las áreas se vea borroso. El problema más significativo de este método es que tampoco es capaz de reconstruir textura en la imagen. Otro trabajo similar, pero que usa ecuaciones parciales diferenciales de orden superior es el presentado por Tschumperlé y Deriche [37], el cual presenta el mismo problema en la reconstrucción de texturas, pero es muy bueno para llenar agujeros pequeños y estrechos.

Aproximaciones variacionales

Existen otros trabajos que tienen diferentes aproximaciones que tratan de mejorar las propuestas de [33] y [34] con nuevos algoritmos. Por ejemplo en [36], se introduce una variación

[2] *Isophota*: Un contorno de igual luminancia en una imagen.

total del modelo *Rudin-Osher Fatemi*, la cual elimina el ruido preservando los bordes, pero que elimina también la textura y detalles de escala fina. Este modelo fue desarrollado para aplicar la técnica de *Inpainting* en orificios pequeños. Una de sus principales fallas, es que es incapaz de reconectar un objeto que haya sido dividido por la oclusión, si la separación de las partes es muy grande.

Otra aproximación fue la realizada por Masnou [31], la cual es compatible con la teoría *Amodal completion de Kanizsa* [38]. Este método se aplica alrededor de *T-junctions* que impactan el área ocluida a través de curvas geodésicas [18]. En primer lugar detectan los píxeles que forman el límite de la oclusión, luego calculan las *T-junctions* que después serán conectadas según reglas de interpolación (descritas en el trabajo) y según una función de energía. Seguidamente, se calcula un camino geodésico para dos *T-junctions* compatibles. Finalmente, se aplica programación dinámica para minimizar la energía y así encontrar el conjunto de conexiones óptimas [31]. Los experimentos realizados arrojaron resultados aceptables a pesar del carácter puramente geométrico de esta aproximación. El uso de parámetros dentro su función de energía coaccionan las líneas para hacerlas más fuertes (o poligonales). Se pueden recuperar los bordes marcados en comparación a otros trabajos. Sin embargo, su trabajo de investigación también falla en la recuperación de texturas, debido a su naturaleza geométrica.

Para una revisión más profunda de varios métodos variacionales así como una buena introducción a la matemática detrás de éstos, se puede acudir al trabajo propuesto por Chan y Shein [36] donde se intenta fijar los fundamentos del campo variacional de *Inpainting* [18]. Existen además trabajos más recientes e innovadores que tratan de dar respuesta al problema señalado por la técnica de *Inpainting* con PDEs, ver [39], donde usan las ecuaciones de fluidos, específicamente las ecuaciones de *Navier-Stokes*, como una PDE base para el procesamiento de imágenes.

1.3.3. Coherencia

Una serie de autores han propuesto añadir una coherencia espacial (y temporal en video) al proceso de síntesis de textura. A partir de esta iniciativa de agregar tales características a dicho proceso, surge un nuevo tipo de técnica que ha sido denominado *coherencia*. El primero en introducir este concepto fue Ashikhmin [7], argumentando que el sistema visual humano es muy sensible a los bordes y esquinas y la medida utilizada en el proceso de síntesis y textura tiende a suavizar los bordes en algunos casos, por lo que no siempre se obtienen resultados suficientemente buenos. En [7] se genera una textura de mayor tamaño a partir de un patrón de menor dimensión. Con ello, es posible explicar una idea inicial para este método de coherencia. La idea consiste en solucionar el problema observado, buscando una mejor correspondencia para el píxel p en el conjunto de candidatos desplazados hacia la vecindad de p, luego de aplicar el método de síntesis de textura, ver Figura 1.8.

En otras palabras, el candidato $\varphi(p)$ es seleccionado de tal manera que la intensidad de su vecindad sea cercana a la intensidad de los candidatos desplazados a la vecindad de p [15]. Este término de coherencia, también ha sido empleado por otros autores que buscan hacer *Inpainting* en video, tal como es el caso de *Space-Time Completion of Video* [8], en donde se presenta un nuevo *framework* para realizar *Inpainting* sobre video, como se puede observar

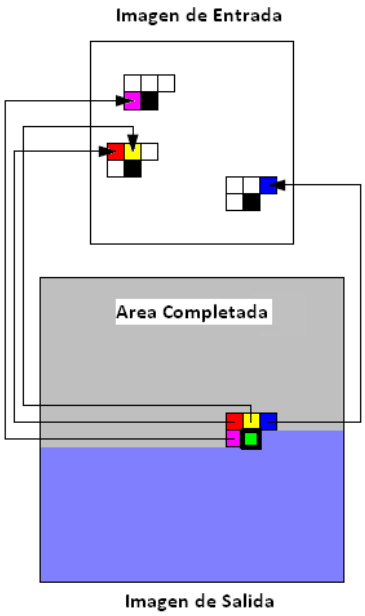


Figura 1.8: Coherencia entre píxeles: Cada píxel en el vecindario actual en forma de L genera un píxel candidato desplazado (píxel negro) de acuerdo a su posición original en la textura de entrada. El mejor candidato es escogido sobre solo éstos candidatos. Diferentes píxeles en el vecindario actual pueden generar el mismo candidato [7].

en la Figura 1.9. Un aspecto importante en su trabajo es la consistencia del espacio-tiempo en un nivel local y global, por ello cuentan con una función de coherencia que emplea un término S que consiste en la coherencia visual global, la cual puede estar asociada a alguna otra secuencia D en caso que S pudiera encontrarse en algún lugar dentro de la secuencia D [8]. Aquí se puede observar como la coherencia entra en un contexto espacial y temporal.

Existen otras propuestas que surgen a partir de [7] en cuanto al sistema visual humano, las cuales están basadas en detección de aristas, para reconstruir cada estructura o textura en el área ocluida [40, 41]. En el año 2008, [42] presenta un algoritmo basado en este concepto. En dicho trabajo, se extiende el modelo basado en *Inpainting* incorporando curvas de Bézier para reconstruir bordes perdidos. Para esto, se aplica un procedimiento para hacer *Inpainting* en el área dañada y la segmentación es usada para extraer contornos de cada región. Después que la estructura de una imagen es determinada, los contornos destruidos serán conectados en el proceso de ajustes de curvas y, las regiones perdidas serán reconstruidas empleando el modelo basado en *Inpainting*. Este enfoque basado en la detección de aristas es denominado en [18] como *algoritmos basados en aristas*.

1.4. Consideraciones finales

Como se mencionó anteriormente, hay muchos trabajos que abordan el problema de *Inpainting*, sin embargo los más exitosos son aquellos que se basan en la combinación de dos

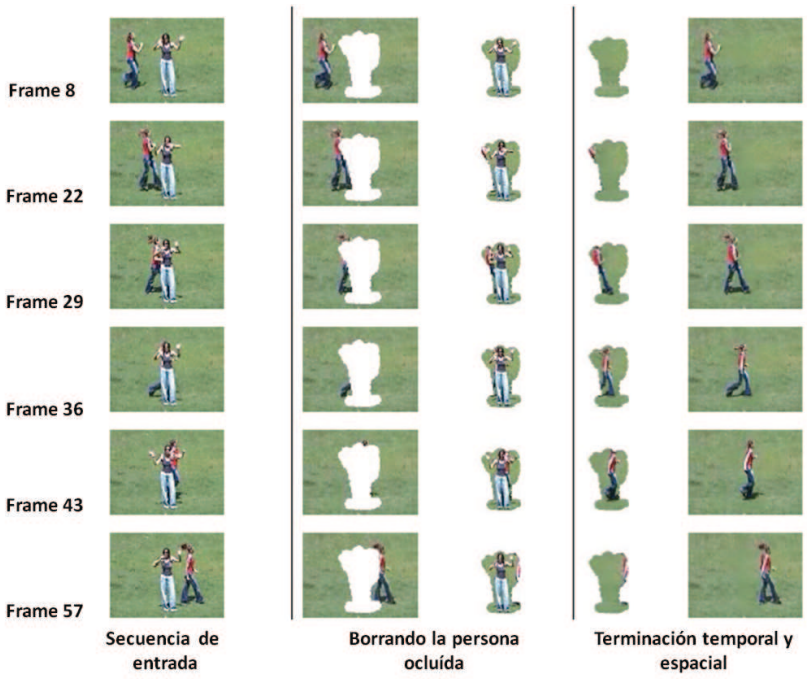


Figura 1.9: Ejemplo del algoritmo *Inpainting* sobre video, considerando la coherencia espacial y temporal [8].

o de las tres técnicas antes discutidas. En el trabajo presentado por Bugeau et al. [15], se combinan las tres técnicas descritas en este capítulo, obteniéndose muy buenos resultados. La técnica de *Inpainting* es un campo de investigación que continúa abierto a nuevas ideas, que busquen dar solución al problema que ataca el algoritmo de manera rápida y eficaz (en este caso que se consiga calidad visual en los resultados). Los capítulos siguientes describen nuestra propuesta para esta técnica.

Capítulo 2

Inpainting CPU

En este capťulo se va a describir nuestra propuesta de algoritmo inpainting en su versión secuencial. El algoritmo tiene como entrada dos imágenes y una serie de parámetros que serán detallados en el transcurso de este capítulo. La primera imagen I de entrada es la destinada a ser procesada, la cual tiene una región desconocida Ω a ser reconstruida y la segunda imagen es una máscara M que permite diferenciar en coordenadas cartesianas los píxeles que deben ser procesados. En este capítulo en primer lugar se habla del enfoque global del algoritmo que resume las características del proceso, para luego detallar los datos de entrada y la forma en que éstos son procesados hasta lograr el objetivo.

2.1. Enfoque global del algoritmo

La región Ω es reconstruida de afuera hacia adentro, siguiendo un esquema de capas (Figura 2.1). El color de cada píxel p en Ω es reemplazado por el color de algún píxel q en Ω^c. Este píxel q es escogido aplicando la técnica de mapa de correspondencia explicada previamente en el Capítulo 1, en la Sección 1.3.1. Adicionalmente, se utilizan diferentes métricas que en conjunto logran un mejor acabado en el resultado. La técnica de mapa de correspondencia es aplicada junto a otros algoritmos iterativos que permiten llevar a cabo el algoritmo planteado en este documento.

El algoritmo *k-Inpainting* surge de la combinación de dos trabajos de investigación: *A Comprehensive Framework for Image Inpainting* [15] y *The Generalized PatchMatch Correspondence Algorithm* [9], ambos publicados en el año 2010. Los aportes resultantes de la composición de estos trabajos, así como la forma en que fueron combinados, será explicado en el transcurso de este capítulo.

Cabe destacar que las métricas presentadas en este trabajo están aplicadas sobre el espacio L*a*b. Al igual que el espacio L*u*v, este espacio es más uniforme perceptualmente con respecto al espacio RGB, y por ello provee una buena estimación de la diferencia (distancia) entre dos vectores de color. Los sistemas L*a*b y L*u*v están basados en la percepción de la luz y un conjunto de ejes de colores opuestos, aproximadamente rojo-verde vs. amarillo-azul [43]. Los componentes u* y v* en L*u*v, así como a* y b* en L*a*b representan la

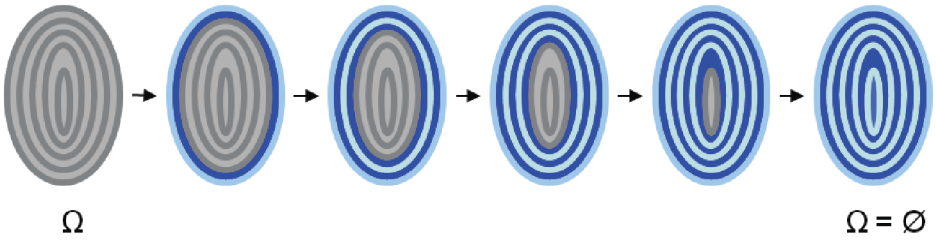


Figura 2.1: Proceso de reconstrucción: La región Ω es reconstruida de afuera hacia adentro siguiendo un sistema de capas.

crominancia del color[1].

La Figura 2.2 diagrama la estructura general del algoritmo y sirve de guía para ubicar y comprender la funcionalidad de los pasos que serán descritos en el resto del capítulo.

El primer paso en la Figura 2.2 es obtener los datos mediante la participación del usuario, quien selecciona la región a ser reconstruida. A partir de la selección, se llena el área con un algoritmo de búsqueda llamado DFS (*Depth-first search*), y se construye una pirámide de Gauss de dos niveles conformada por las imágenes I e I'. Seguidamente sobre la imagen más pequeña I' se aplica la etapa de inicialización que proporciona valores preliminares a Ω a partir del procesamiento de los píxeles recolectados mediante un muestreo. Luego sobre I' se aplica el procesamiento de textura para refinar la zona a tratar y la prepara para la siguiente etapa, en la cual se aplica una función de energía que considera las tres técnicas de *Inpainting* descritas en el capítulo anterior. Después, se realiza una proyección de los píxeles procesados en I' a la imagen original I. Finalmente, se aplica la función de energía sobre I y se obtienen los resultados después de aplicar el algoritmo.

Las sub-etapas *K-Propagación* y *Búsqueda Aleatoria* son utilizadas en diferentes etapas del algoritmo. Tales sub-etapas son algoritmos que trabajan en conjunto y tienen como objetivo encontrar los mejores candidatos para cada píxel que constituye la parte desconocida de la imagen. Se denomina candidato a cualquier píxel $q \in \Omega^c$ que pueda reemplazar el valor incógnita de algún píxel $p \in \Omega$. Otro aspecto importante, es que dichos algoritmos utilizan métricas diferentes dependiendo de la etapa en la que se estén ejecutando. Debido a esto, ambos son explicados posteriormente de forma genérica en la Sección 2.3.2.

A continuación se explicarán detalladamente cada etapa junto con otros aspectos importantes que serán discutidos en el apartado que describe el algoritmo propuesto.

2.2. Obtención de datos

El usuario selecciona la imagen a ser manipulada y luego con una herramienta de selección (por ejemplo, selección a mano alzada) señala el área que desea tratar. Cuando la zona es seleccionada a mano alzada, se busca un punto dentro del área a reconstruir y se aplica

[1]Es el componente que contiene la información del color o croma de una señal de video y viene definido por los parámetros de tono y saturación [43].

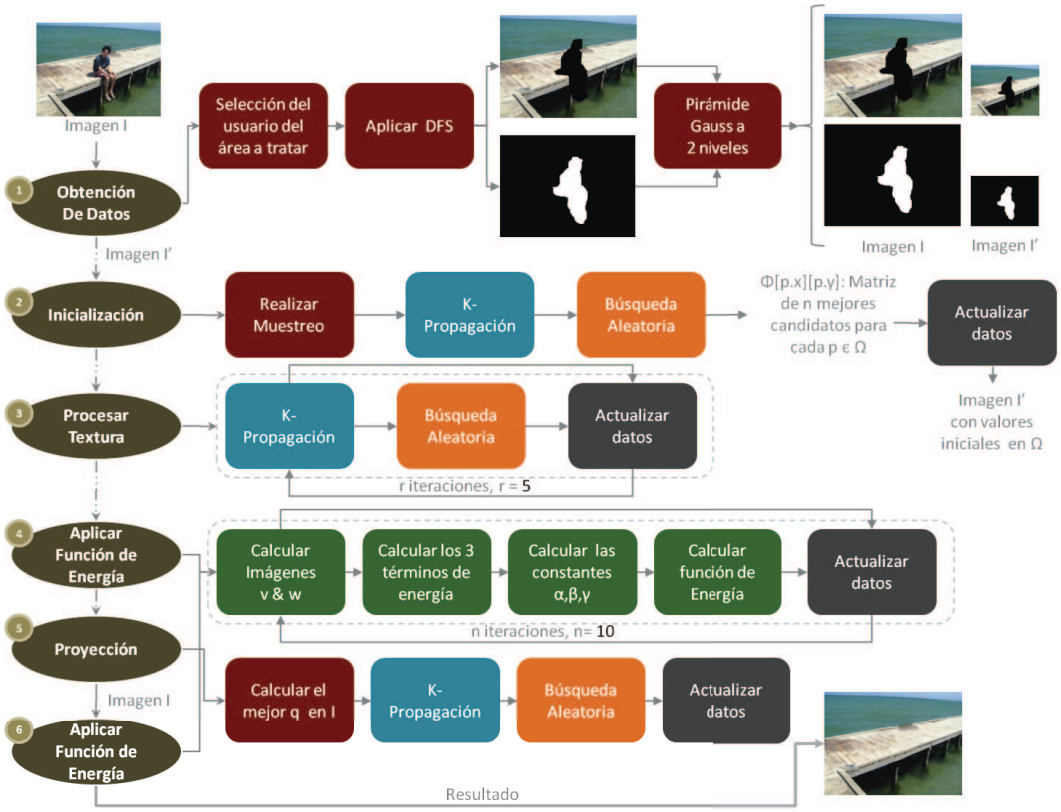


Figura 2.2: Estructura general del algoritmo propuesto.

un algoritmo conocido como DFS para llenar dicha región, tal como se puede observar en la Figura 2.3. La máscara generada para dicha imagen es la que se muestra en la Figura 2.4.

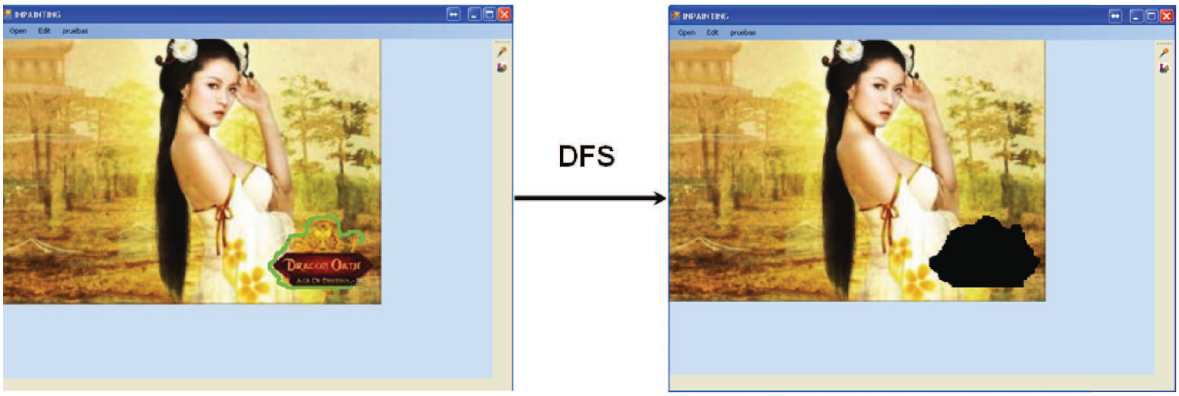


Figura 2.3: Selección del área a reconstruir en la imagen.

El algoritmo *k-Inpainting* utiliza un esquema de multiresolución empleado en varias investigaciones [20, 44, 45]. La estrategia de multiresolución[2] es lograda mediante la construc-

[2]Esquema frecuentemente usado para mejorar la velocidad, la exactitud y robustez. La idea básica consiste

Figura 2.4: Máscara de la imagen que permite diferenciar el área a ser tratada (Ω).

ción de una pirámide gaussiana [46] de 2 niveles, es decir se obtiene una nueva imagen I', que es dos veces más pequeña que la imagen I (ver Figura 2.5).

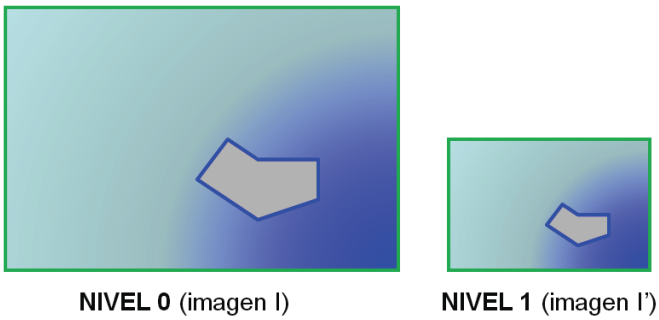


Figura 2.5: Pirámide Gaussiana de 2 niveles.

Para cada imagen se recopila la siguiente información:

- Máscara.
- Puntos máximos y mínimos del área a reconstruir.
- Módulo de los puntos máximos y mínimos con respecto a la imagen anterior: Para el paso de proyección es importante ubicar correctamente la correspondencia de los puntos entre las imágenes, especialmente para las coordenadas ubicadas dentro de Ω. En algunos casos las dimensiones de la imagen no son múltiplo de 2, por ello deben almacenarse el resto de la división. La forma de corresponder un punto de la imagen I a la imagen I' se tratará en la Sección 2.3.6 (*proyección*).

---

en aplicar un proceso específico sobre una escala de información burda y la transformación estimada se utiliza para inicializar el mismo proceso sobre la próxima escala de información más fina o con mayor detalle [45].

2.3. Algoritmo *k-Inpainting*

En la Sección 2.1 se explica grosso modo la Figura 3.4 que resume el algoritmo mediante una secuencia de etapas, las cuales serán explicadas en este apartado. La comprensión de éstas requiere entender previamente ciertos aspectos importantes que serán especificados en el siguiente punto a tratar. Luego en la Sección 2.3.2, se detallarán 2 sub-etapas comunes para varias etapas o fases de la Figura 3.4, que permiten seleccionar candidatos $q \in \Omega^c$ bajo ciertos criterios contemplados en dicho apartado. Después de lo anterior, se hablará con respecto a cada una de las fases del algoritmo.

2.3.1. Consideraciones previas

Para aplicar *k-Inpainting*, se requiere realizar un muestreo sobre la parte conocida de la imagen para seleccionar píxeles candidatos que pueden sustituir los valores desconocidos de los píxeles que constituyen a Ω; así como emplear un montículo (tambiín conocido como *heap*) con algunas variaciones que permita almacenar los mejores candidatos para cada píxel $p \in \Omega$.

En las diferentes etapas que conforman el algoritmo *k-Inpainting* se emplean diferentes métricas, algunas de ellas denominadas términos de energía, mientras que en otra sub-etapa se emplea una ecuación que acopla dichos términos. Por ello, previamente se deben conocer tales métricas, antes de explicar con detalle cada etapa que conforma el algoritmo. A continuación, se describirá la forma en que se efectúa el muestreo, así como las características de la estructura de datos (montículo) antes mencionada. También se hablará con respecto a los términos de energía y de la combinación de los mismos en una función de energía.

Implementación del *Heap*

La formulación de esta estructura está basada en [9], donde implementan un *max-heap*[3] que almacena la distancia D entre dos vectores de color, teniéndose la mayor distancia en el tope de la estructura. Cuando la estructura está llena y existe un candidato a insertar con una distancia menor que el valor en la raíz, ésta última es eliminada. Cuando se examinan los candidatos, se construye una tabla *hash* que verifica rápidamente si el candidato ya ha sido almacenado en la lista [9].

En base a lo anterior, se ha implementado un *max-heap* empleando arreglos con algunas variaciones. La estructura almacena dos datos, un tipo de dato *punto* que identifica cual píxel $q \in \Omega^c$ es candidato a reemplazar algún $p \in \Omega$ y un tipo de dato *float* que indica la diferencia entre las vecindades de p y q para alguna métrica. La estructura no admite repeticiones en el dato *punto* para evitar repetir candidatos para un p determinado. Por ello se dispone de una tabla *hash* al igual que en [9] para guardar los puntos que ya han sido escogidos. Antes de insertar un nuevo elemento se comprueba en la tabla si ya existe tal punto; en caso que ya exista se descarta y en caso contrario se inserta en el *max-heap* y en la tabla *hash* para

[3]Un *max-heap* es un árbol binario completo en el cual el valor en cada nodo es mayor o igual a sus hijos (en caso de existir alguno) [47].

evitar repeticiones futuras. El ordenamiento de los puntos viene dado por la distancia entre los colores de la vecindad de p y q (ver Figura 2.6).

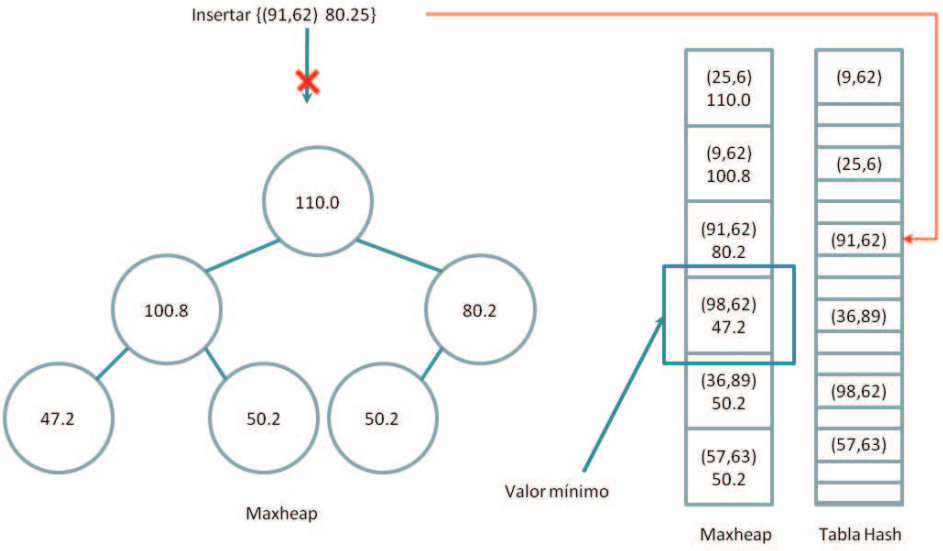


Figura 2.6: Funcionamiento del *max-heap*: El *heap* se ordena en función del valor distancia, teniéndose la mayor distancia en el tope del mismo. Si se inserta un valor con distancia menor a la que está en el raíz del *heap*, se inserta. En caso de que la estructura esté llena, si el valor de la distancia del nodo que se va a insertar es menor que la de la raíz, entonces el nodo de la raíz se elimina y se inserta el nuevo nodo. Si en el *heap* ya se encuentra un nodo con la misma posición del nodo que se desea insertar, el mismo no es insertado. Por ejemplo, en esta Figura se desea insertar el nodo con distancia 80.2 y posición 91,62. Sin embargo, antes de insertar se compara con la tabla *Hash*, como la posición 91,62 ya existe entonces esta inserción no es ejecutada.

Antes de actualizar los datos en la imagen en cada paso del algoritmo, se debe conocer cuál de los N candidatos es más conveniente para cada píxel p. El mejor candidato es el que tiene menor valor en la variable flotante dentro de la estructura, es decir es el píxel que tiene menor diferencia entre su vecindad y la vecindad del píxel para el cual es candidato. Debido a la importancia de este valor, la clase que implementa el *max-heap* tiene un atributo que almacena las características del mejor candidato. Este valor puede cambiar o actualizarse en cada operación de inserción.

Se estimó que si el parámetro N corresponde al 0.05 % de los píxeles totales de la imagen (ρ en 2.1), se obtienen resultados satisfactorios. Como se reseñará más adelante, en el algoritmo se maneja un atributo $16 \leq K \leq 32$ para dos algoritmos que aquí hemos llamado *K-Propagación* y *Búsqueda Aleatoria*. Si el valor de N es menor que el máximo valor que toma K, entonces N toma el máximo valor de K. Dicho esto, N es definido como lo indica la ecuación 2.1, dónde $\rho = (I.Width * I.Height * 0{,}05)/100$.

$$N = \begin{cases} \rho & \text{si } \rho \geq 2 * K \\ 2 * K & \text{de otro modo} \end{cases} \tag{2.1}$$

La definición de N es formulada de esta manera para que la dimensión del *heap* no sea muy pequeña con respecto a una selección previa de K candidatos.

Muestreo

El paso de inicialización que será explicado en la próxima sección, realiza un muestreo uniforme sobre la imagen a procesar, para escoger aleatoriamente los posibles candidatos de cada píxel $p \in \Omega$. Para llevar a cabo tal muestreo se requiere de un generador de números pseudo-aleatorios con distribución uniforme, en los que cada valor tiene la misma probabilidad. Para ello se implementaron 2 métodos que serán descritos brevemente a continuación:

- **Método de congruencia lineal:** Es el método más conocido para generar números aleatorios, utilizado casi exclusivamente desde que fue introducido por D. Lehmer en 1951, donde para obtener un nuevo número aleatorio, se toma el anterior, se multiplica por una constante *b*, se le suma *1* y se toma el resto de la división por una segunda constante *m*. El resultado es siempre un entero entre 0 y m-1. Este método ha sido objeto de volúmenes enteros de estudio detallado y complicados análisis matemáticos. En [48] se muestran formas adecuadas para elegir las constantes antes mencionadas.

- **Ran1:** Método que retorna un valor aleatorio de una desviación uniforme entre 0.0 y 1.0. Inicializa una constante con algún valor negativo para inicializar o reinicializar la secuencia [49].

Estos generadores fueron utilizados dentro de un algoritmo probabilista denominado *Algoritmo de Las Vegas*. Los algoritmos probabilistas están basados en probabilidades para tomar decisiones, ya que suele ocurrir que cuando un algoritmo debe tomar una decisión, si el costo asociado es muy grande, es preferible tomar la decisión aleatoriamente. Existen tres tipos de algoritmos probabilistas: Algoritmos Numéricos, Algoritmo de Monte Carlo y el Algoritmo de Las Vegas. El Algoritmo de Las Vegas es capaz de reconocer cuando una respuesta es incorrecta, por lo que ignora este tipo de respuestas ejecutándose nuevamente [50].

Es importante realizar un muestreo sobre la parte conocida de la imagen, ya que realizar el algoritmo a fuerza bruta (cada píxel desconocido por cada píxel de la parte conocida de la imagen) tiene un alto costo computacional que, para ciertas imágenes podría tomar muchas horas de procesamiento con PCs convencionales. En la próxima sección, se explicará cómo se inicializa la parte desconocida de la imagen con ayuda del muestreo, el cual solamente se emplea en la etapa de inicialización del algoritmo propuesto.

Combinación de los términos de energía

En el Capítulo 1, se ha descrito en qué consisten las técnicas de auto-similitud y síntesis de textura; difusión y propagación; y coherencia. La investigación realizada en [15] une éstas tres técnicas en una función que permite seleccionar un candidato no solo por la similitud de sus vecindades en cuanto a los colores que la constituyen, sino también en cuanto a la

forma en que están organizados dichos colores (coherencia espacial) y el cambio del tono del color de un vecino a otro (difusión). Cada uno de estos métodos fueron planteados como un término de energía, los cuales serán descritos a continuación:

Auto-similitud y síntesis de textura

Los trabajos [1, 16, 15] presentan una formulación variacional del método de síntesis y textura propuesto por Efros y Leung [21]. Esta variación consiste en calcular el mapa de correspondencia φ que minimiza la función de energía expresada en [1], para un píxel $p \in \Omega \subset \iota$, tal como sigue:

$$E(\varphi) = \sum_{\tau \in N_0} \|\iota(\varphi(p+\tau)) - \iota(q+\tau)\|^2 \tag{2.2}$$

La síntesis de textura suele ser un proceso de refinamiento iterativo, por lo que la ecuación 2.2 debe ser expresada en función a las r iteraciones aplicadas para procesar la textura (ver Ecuación 2.3).

$$E_1(\varphi, \varphi^{r-1}) = \sum_{\tau \in N_0} \left\|\iota(\varphi^{r-1}(p+\tau)) - \iota(q+\tau)\right\|^2 \tag{2.3}$$

En [15] proponen una variación de este término de energía en función del histograma RGB de cada vecindad p y q, sin embargo como en el algoritmo *k-Inpainting* se trabaja sobre el espacio L*a*b, tal modificación no fue considerada.

Difusión y propagación

Al igual que en [15], se toma la difusión Laplaciana como segundo término de energía. Dado un píxel $p \in \Omega$, la difusión Laplaciana de p es definida como:

$$\upsilon(p, \varphi) = \frac{1}{4}\left[\iota(\varphi(p^N)) + \iota(\varphi(p^S)) + \iota(\varphi(p^E)) + \iota(\varphi(p^O))\right] \tag{2.4}$$

A partir de 2.4, el segundo término de energía es formulado como:

$$E_2(\varphi, \varphi^{r-1}) = \sum_{\tau \in N_0} \left\|\upsilon(p+\tau, \varphi^{r-1}) - \iota(q+\tau))\right\|^2 \tag{2.5}$$

Coherencia

El tercer y último termino de energía es definido en [15] como sigue:

$$E_3(\varphi, \varphi^{r-1}) = \sum_{\tau \in N_0} \left\|\omega(p, \tau, \varphi^{r-1}) - \iota(q+\tau))\right\|^2 \tag{2.6}$$

Siendo ω el cálculo que favorece la similitud de los parches correspondiente a los píxeles vecinos. Para hacer esta estimación menos sensible a valores que no se encuentran en la

imagen (valores aberrantes), en [15] proponen emplear la mediana de los valores generados, tal como sigue:

$$\omega(p, \tau, \varphi^{r-1}) = median_{l \in N_0} \left\{ \iota(\varphi^{r-1}(p + l) - l + \tau) \right\} \tag{2.7}$$

Para calcular la mediana es necesario hacer una ordenación de los elementos involucrados, por ello los colores en el espacio L*a*b fueron ordenados comparando primero el componente de luz, en caso que para dos colores estos valores sean iguales, se ordena con respecto al componente *a, y por último por el componente *b. Se ordenaron de esta forma, debido a la sensibilidad del ojo humano a la luz.

Todos los términos de energía se basan en la suma de diferencias cuadradas. Es posible que para algunos píxeles q, parte de sus vecindades estén fuera de la imagen. Cuando esto ocurre, se le suma una constante de alto valor en las posiciones de la ventana que no se encuentran dentro de la imagen.

Combinación de los tres términos de energía

En la Sección 2.3.2 se describe como se escogen los N mejores candidatos para cada $p \in \Omega$, los cuales son almacenados en un conjunto $\Phi(p)$. Para cada conjunto $\Phi(p)$ se aplican los tres términos de energía combinados con respecto a cada píxel p en cada iteración r, tal combinación es mostrada en la Ecuación 2.8.

$$\varphi^{r-1}(p) = argmin_{q \in \Phi(p)}(\alpha E_1 + \beta E_2 + \gamma E_3) \tag{2.8}$$

Las constantes α, β y γ definen la influencia de cada uno de los tres términos en el resultado final. En [15] estos pesos han sido definidos dependiendo de las propiedades de cada píxel en la imagen según las métricas determinadas. El cálculo de dichos pesos viene dado en primer lugar por el valor mínimo de cada uno de los términos de energía (ver Ecuación 2.9)

$$m_i(p) = min_{q \in \Phi(p)} E_i \tag{2.9}$$

Estos mínimos son considerados en la estimación de las constantes, ya que se asume que los valores de sus pesos podrían depender de la validez del mejor parche candidato (píxel q con su vecindad definida según el parámetro L). Esto es porque si el parche candidato para un determinado píxel en un tiempo t no es suficientemente bueno, el valor del correspondiente peso podría ser pequeño, es decir, se decrementará en función del valor $m_i(p)$. En relación a lo antes descrito, α, β y γ se determinan numéricamente como sigue:

$$\alpha(p) = \epsilon^{\frac{-m_1}{\sigma}}, \beta(p) = \epsilon^{\frac{-m_2}{\sigma}}, \gamma(p) = \epsilon^{\frac{-m_3}{\sigma}}, \text{dónde } \sigma = \frac{m_1 + m_2 + m_3}{3} \tag{2.10}$$

La ecuación laplaciana aplicada sobre cada píxel $p \in \Omega \subset \iota$ genera una imagen denominada imagen de difusión v y el cálculo de ω sobre el mismo conjunto de píxeles produce una imagen de coherencia w. Por ello es conveniente construir en primer lugar la imagen

coherencia y difusión con las ecuaciones 2.7 y 2.4 respectivamente, y luego realizar todos los cálculos pertinentes para cada término de energía.

Como se mencionó en la Sección 2.1, en varias etapas del algoritmo se aplican los algoritmos de *K-Propagación y Búsqueda aleatoria* para obtener los candidatos de cada píxel desconocido de la imagen, por ello antes de detallar cada etapa se describirá el funcionamiento de estos dos algoritmos en conjunto a continuación.

2.3.2. Obtener candidatos

Dependiendo de la etapa del algoritmo que se esté aplicando, se puede tener una estructura Υ (lista de muestras) o Φ (matriz de *heaps*) con candidatos. Dichas estructuras son datos de entrada a dos algoritmos que encuentran los mejores N píxeles candidatos a sustituir el valor desconocido de cada píxel p.

Estos dos algoritmos reciben el nombre de *K-Propagación* y *Búsqueda Aleatoria*, el primero toma cada muestra y la compara con K vecinos en una dirección x y en una dirección y. El segundo algoritmo hace una búsqueda aleatoria en varias direcciones a diferentes radios de distancia del píxel escogido y lo compara con sus vecinos. Cabe destacar que estas funciones forman parte del algoritmo *NNF* (*Nearest Neighbors Field*) y *K-NNF* (*K-Nearest Neighbors Field*) presentados en [51] y [9] respectivamente, los cuales están enfocados en la síntesis de textura **basada en parches**. Aquí se han **adaptado** estos algoritmos para sintetizar la textura **basada en píxeles**.

Las métricas que permiten conocer cual píxel es mejor candidato que otro están basadas en la suma de las diferencias cuadradas de las vecindades del píxel incógnita y el píxel seleccionado. Suponiendo la utilización de alguna métrica que calcula una distancia D, a continuación se explicará el funcionamiento de los algoritmos antes mencionados.

K-Propagación

En este trabajo, hemos llamado a este algoritmo *K-Propagación* por ser una variación del algoritmo de propagación propuesto en [9] y su funcionamiento se describe a continuación:

Dado un píxel desconocido $p \in \Omega$, *K-Propagación* es aplicado a todos los píxeles $q \in \Upsilon$ ($q \in \Phi$ dependiendo de la etapa que esté ejecutando *k-Inpainting*). Sin embargo, antes se deben escoger las direcciones en el eje x y y. En el algoritmo de *Propagation* propuesto en [51] se establece que los candidatos $f(x, y)$ tratan de ser mejorados haciendo desplazamientos $f(x-1, y)$ y $f(x, y-1)$ y en algunas iteraciones los examinan de forma inversa $f(x-1, y)$ y $f(x, y-1)$. En esta propuesta, las direcciones son escogidas aleatoriamente, tal como se lo indican las ecuaciones 2.11 y 2.12, dónde $0 \leq k < K$.

$$q_{kx} = (x \pm k, y) \tag{2.11}$$

y

$$q_{ky} = (x, y \pm k) \tag{2.12}$$

La aleatoriedad de las direcciones viene dada por la elección del signo. Debido a esto, en esta propuesta, se escogen dos números aleatorios, uno para el eje x y otro para el eje y. En cada caso, si el número aleatoriamente seleccionado es par, el signo es negativo y es positivo en caso que el número sea impar. Para y se sigue la misma política, pero con el número seleccionado aleatoriamente para el eje y.

Se debe acotar que K (valor límite superior de la constante k en las ecuaciones 2.11 y 2.12) es un parámetro global del algoritmo. La Figura 2.7 muestra diferentes resultados de estudios realizados en [9] sobre las variantes del algoritmo *k-PatchMatch*. Los resultados de la Figura 2.7, indican que el valor ideal para K es el valor constante 16. Por lo que se tomará un rango de valores [8,32) para esta variable global en el algoritmo *k-Inpainting*.

En [9] se puede observar que NNF es más rápido, pero menos preciso, tal como se planteó en la idea general descrita anteriormente en este documento. Por otro lado, correr el algoritmo NNF K veces arroja resultados más precisos, pero con mayor tiempo de ejecución. Según la Figura 2.7, al procesar los candidatos con la estructura *heap* se logran resultados aún más precisos. De aquí se puede deducir que entre más grande sea el K, los resultados van a ser más exactos, pero con un alto costo computacional.

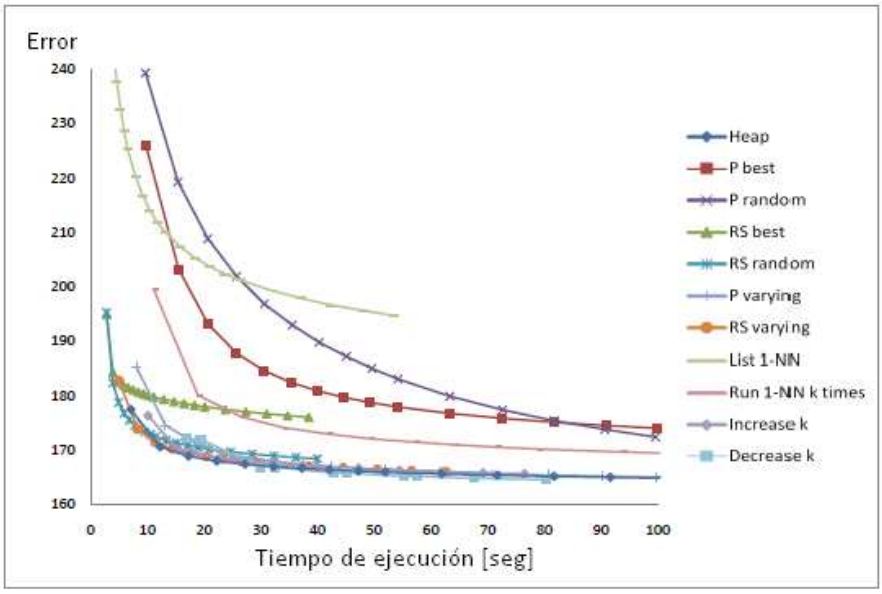


Figura 2.7: Convergencia de algoritmos NNF [9].

Una vez comprendido como se escogen las direcciones x y y y la influencia de la variable K, se puede describir el funcionamiento completo de la subetapa *K-Propagación*, que se resume en el Algoritmo 1.

Gráficamente se están examinando los puntos que se muestran en la Figura 2.8 para un píxel q y un píxel p. Para guardar los mejores K candidatos de los $2K$ candidatos generados, se utiliza una estructura *max-heap* auxiliar H que retorna los K valores en una lista λ.

Algoritmo 1 K-Propagación

procedure KPROPAGACION(Point $p \in \Omega$, Point $q \in \Upsilon$, Int K) ▷ Siendo $\Upsilon \subset \Omega^c$
    MaxHeap H(K) ▷ H máximo almacena K candidatos
    Escoger una dirección aleatoria en el eje de las x (derecha o izquierda + ó -)
    Escoger una dirección aleatoria en el eje de las y (arriba o abajo + ó -)
    Evaluar y obtener D para K candidatos electos con 2.11 y almacenarlos en H
    Evaluar y obtener D para K candidatos electos con 2.12 y almacenarlos en H
    List λ = H.List()
    return λ
end procedure

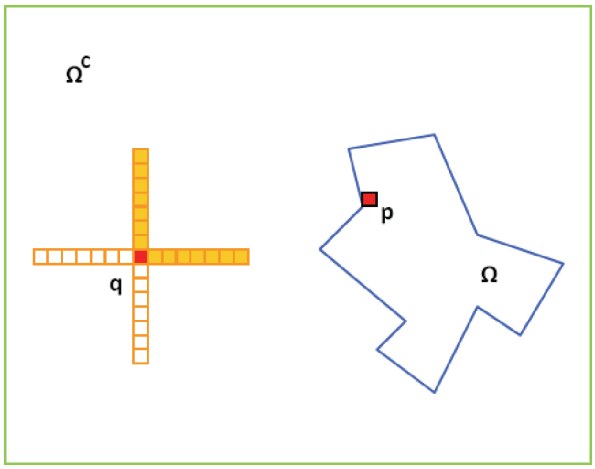


Figura 2.8: Se generan K píxeles hacia arriba y hacia la derecha para este caso. Como la elección de la dirección es aleatoria se pueden generar otras 3 combinaciones.

Búsqueda aleatoria

En *K-Propagación* se obtuvo la lista λ con los K mejores candidatos. Para evitar un mínimo local, se aplica el algoritmo de *Búsqueda Aleatoria* propuesto en [51], en donde para cada $q \in \lambda$, se aplica lo siguiente:

Dado $v_0 = \varphi(p)$, se intenta mejorar tal valor probando una secuencia de desplazamientos de candidatos en un decremento de distancia exponencial desde v_0, tal como se visualiza en la Figura 2.9. Esta descripción se traduce matemáticamente mediante la Ecuación 2.13 propuesta en [51].

$$u_i = v_0 + w\alpha^i R_i \tag{2.13}$$

La variable R_i es una variable aleatoria uniforme sobre la ventana $[-1, 1] \times [-1, 1]$, w es el máximo radio de búsqueda y α es un valor fijo que varía la distancia del radio de búsqueda en alguna dirección. Se examinan las vecindades de los nuevos puntos generados para i = 0,1,2, ..., hasta que $w\alpha^i < 1$. Al igual que en [51], w es la máxima dimensión de la imagen y $\alpha = 1/2$. Sin embargo cuando un nuevo punto generado está fuera de la imagen, este se

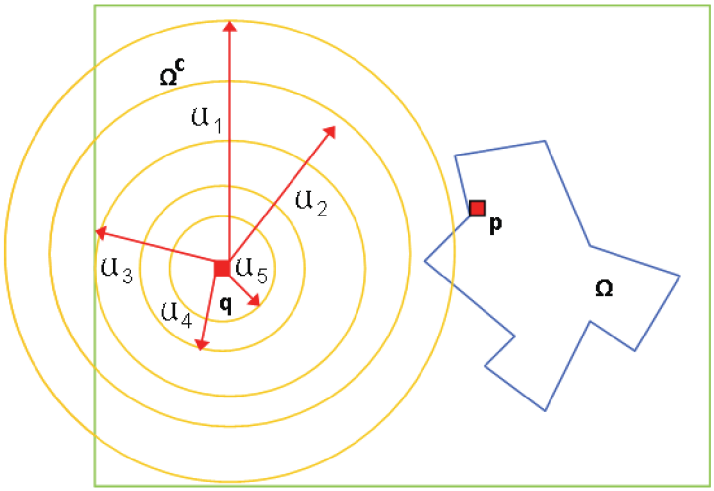


Figura 2.9: Búsqueda aleatoria de puntos a partir de una posición q.

descarta y se realiza la siguiente iteración, en vez de limitarlo al borde de la imagen en esa dirección (como ocurre en [51]). Esto es porque hay menos vecindades en los bordes de la imagen (ver Figura 2.10).

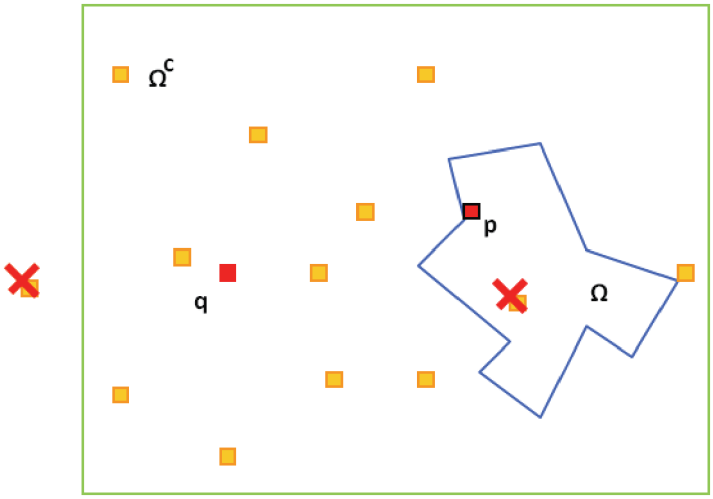


Figura 2.10: Píxeles fuera de las dimensiones de la imagen: Los puntos que se generan fuera de la imagen o dentro de Ω son descartados.

Una vez comprendido el funcionamiento de estos algoritmos es posible dar paso a la descripción de las etapas del diagrama 2.2.

2.3.3. Inicialización

El principal objetivo de este paso es encontrar desde un punto de vista espacial los mejores candidatos para los valores incógnitas de I. Por ello, este paso es llamado paso de

inicialización.

En primer lugar se busca $\varphi(p)$ definido en la ecuación 1.4 para cada píxel $p \in \Omega$. Una forma de encontrar este argumento empleado por diferentes trabajos [1, 44, 20, 23] es utilizando Campos Aleatorios de Markov (Markov Random Fields - MRF). Sin embargo, este mecanismo puede ser costoso en el tiempo para nuestro propósito, por ello en [44] se agiliza el tiempo de respuesta usando esta técnica con el algoritmo de Label Pruning. Esta técnica ha sido empleada por otros trabajos de investigación, tal como el caso de Bugeau et al. [15] que utiliza dicho procedimiento para acelerar el tiempo de respuesta del algoritmo resultante de su investigación.

Recientemente ha sido propuesto en [51] un algoritmo que busca rápidamente $\varphi(p)$ que recibe el nombre de *Nearest Neighbors Field* (NNF). Sobre este fue realizado una optimización en [9] que permite obtener resultados más precisos, dicha optimización recibe el nombre de *K-Nearest Neighbors Field* (*K-NNF*). En cualquier caso, Connelly Barnes et al. [51] en sus trabajos emplea en sus algoritmos de *PatchMatch* un paso de inicialización, en el cual realiza unas pocas iteraciones iniciales del algoritmo *NNF* o *K-NNF* usando una inicialización aleatoria.

Se toma esta idea como base y sobre $\Omega^c \in I'$ se realiza un muestreo uniforme, a partir del cual se aplicará el algoritmo *NNF*, pero con algunas variantes de las funciones *Propagation* y *Random Search*, que han sido explicadas previamente en la Secciones 2.3.2 y 2.3.2.

En *k-Inpainting* el muestreo se efectúa aplicando el algoritmo probabilístico de las Vegas junto a un generador pseudo-aleatorio tal como se especifica en las consideraciones previas. La función que aplica el Algoritmo de Las Vegas recibe dos parámetros importantes, el porcentaje de píxeles que serán tomados por fila o columna y un valor δ que indica cada cuantas filas o columnas va a ser realizado el muestreo. Si el ancho de la imagen es mayor que el alto de la misma, el porcentaje va dirigido a las filas y el valor δ indica cada cuantas filas se toma el muestreo de las mismas. En caso que el alto de la imagen sea mayor que el ancho, se realiza lo mismo descrito anteriormente pero hacia las columnas.

El porcentaje de píxeles tomados por cada fila o columna es del 25 %, mientras que el δ toma el valor constante 2. Estos valores fueron estimados de forma empírica, realizando diferentes pruebas. El algoritmo probabilístico aplicado devuelve una lista Υ con píxeles seleccionados, en dicha lista todos los píxeles son diferentes entre sí en cuanto a ubicación espacial en la imagen.

El muestreo es realizado cada vez que se detecta un borde $\delta\Omega$, por lo que los píxeles en Υ_j son los candidatos a sustituir a los píxeles $p_i \in \delta\Omega_j$ por cada $\delta\Omega_j \subset \Omega$, la Figura 2.11 gráfica el muestreo realizado para cada capa sobre la máscara de la imagen a procesar y la Figura 2.12 muestra los valores que toma Υ_j para $\delta\Omega_j$.

Para cada píxel p_i se buscan los N mejores píxeles que son candidatos a reemplazar el valor desconocido de cada p_i. Los N mejores candidatos para cada píxel $p \in \Omega$ son almacenados en un *max-heap* (Sección 2.3.1), por ello se construye una matriz de tipo *max-heap* de las dimensiones de la zona a reconstruir. Como la forma de la zona, puede ser irregular, dentro de la matriz pueden existir estructuras vacías para los píxeles que no pertenecen a Ω.

Una vez hallados los mejores N candidatos para cada píxel desconocido del agujero, se asigna el mejor candidato al píxel p_i de la zona a tratar de la imagen. En este caso, por ser

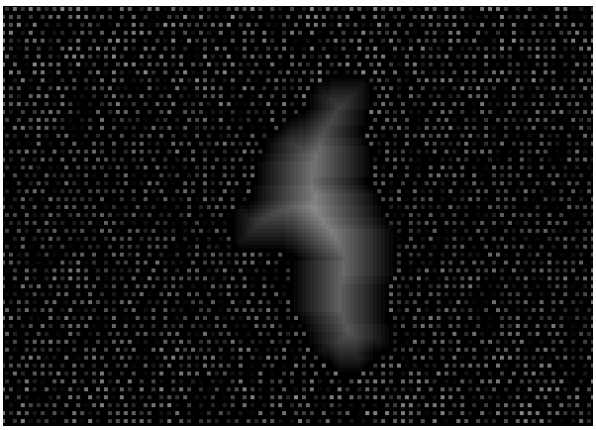


Figura 2.11: Por cada capa se realiza un muestreo.

Figura 2.12: Muestreo para una capa: En este caso para la primera capa más externa de Ω.

el paso de inicialización se está reconstruyendo el área Ω de I', y se indica en una máscara auxiliar cuales píxeles fueron sustituidos. Debido a que el proceso de reemplazar los valores de los píxeles es iterativo, siempre se usa una máscara auxiliar para no perder la localidad espacial del área de la imagen que se está procesando. El fragmento del Algoritmo 2 resume lo indicado en este párrafo.

Luego de obtener las muestras en Υ, se aplica el proceso descrito en la Sección 2.3.2 para hallar los N mejores candidatos y almacenarlos en Φ. En otras palabras, los algoritmos de *K-Propagación* y *Búsqueda Aleatoria* son aplicados sobre cada una de las muestras con respecto a cada píxel p_i y encuentran los mejores N píxeles a sustituir el valor desconocido de cada p.

Una vez realizados los dos algoritmos anteriores en el paso de inicialización, se ha calculado $\varphi(p)$ $\forall p \in \Omega \subset I'$. Para calcular $\varphi(p)$ en este paso, se utiliza una variación sobre la suma de diferencias cuadradas (abreviado como ssd), de manera similar a como la plantean en [44], donde la diferencia de los vecinos en posiciones desconocidas, no son calculadas. La definición de esta métrica para este trabajo se muestra en la ecuación 2.14, dónde $p \in \Omega$,

Algoritmo 2 Algoritmo para inicializar Ω

```
Array ϒ
MaxHeap Φ[widthHole][heightHole]
Image M = mask          ▷ Se asigna la máscara (mask) de I' a una máscara auxiliar
mientras Ω ≠ ∅ hacer
    Seleccionar δΩ_j
    RealizarMuestreo(ϒ_j, Mask, ...)
    para cada píxel p ∈ δΩ_j hacer
        Hallar los n mejores candidatos a partir de ϒ_j y almacenarlos en Φ[p.X][p.Y]
        I(p) = φ(p)
        M(p) = 0
    end para
end mientras
```

$q \in \Omega^c$ e $\iota = \Omega \cup \Omega^c$:

$$ssd(N_p, N_q) = \sum_{\tau \in N_0} M(p+\tau) * \|\iota(p+\tau) - \iota(q+\tau)\|^2 \tag{2.14}$$

2.3.4. Procesamiento de textura

Este paso tiene como entrada el conjunto Φ, por lo que ya se tendrían candidatos electos para cada píxel p. Estos píxeles han sido escogidos mediante la ecuación 2.14, siendo estos los primeros valores que toma Ω en el paso de inicialización, por elloen dicho paso se llama $\varphi^0(p)$ al mejor candidato q que sustituyen a cada píxel p. Esto debido a que el algoritmo *k-Inpainting* es iterativo y a partir de $\varphi^0(p)$ es que los términos de energía antes explicados son estimados.

Para procesar una textura hay que sintetizarla. Para esto, se utiliza nuestra implementación de K-NNF (Obtener candidatos) con r iteraciones y empleando la ecuación 2.2 en vez de 2.14. Esto es para refinar el procesamiento de la textura y obtener un mejor acabado.

Los candidatos a ser evaluados provienen del conjunto $\Phi(p)$ para cada p, como se mencionó anteriormente y estos son actualizados en cada iteración. En síntesis, se realizan las operaciones mostradas en el Algoritmo 3.

Así como en [51], cuando r toma como valor máximo 5, se obtienen resultados satisfactorios, sin embargo en el capítulo 4 se notará que este valor puede variar. Los puntos p pertenecientes a Ω siempre se toman de afuera hacia dentro.

2.3.5. Aplicar función de energía

En la Sección 2.3.1 se mostraron las diferentes ecuaciones que componen los términos de energía y la forma como los mismo son combinados. Ahora se procederá a detallar como son utilizados dichos términos y como es aplicada la función de energía formulada en 2.8.

Algoritmo 3 Algoritmo para procesar textura

```
1: r = 0                                   ▷ Sea valor el número de iteraciones
2: mientras r < valor hacer                ▷ Aplicar el algoritmo para Obtener candidatos usando
   como candidatos a los q ∈ Φ(p) y las ecuaciones 2.2 y 2.3
3:     mientras Ω ≠ ∅ hacer
4:         Seleccionarδ Ω_j
5:         para cada píxel p ∈ δΩ_j hacer
6:             λ = Kpropagacion(Φ(p))
7:             BusquedaAleatoria(λ, Φ(p))   ▷ Φ(p) se actualiza en la Búsqueda Aleatoria
8:             ι = φ(p)
9:         end para
10:    end mientras
11:    r = r + 1
12: end mientras
```

Primero se generan las imágenes de difusión v y coherencia w. Para este momento debe existir una matriz o conjunto de *max-heaps* Φ como lo expresa la ecuación 2.15.

$$\Phi|\Phi[p.X][p.Y] \vee \Phi(p) \equiv Mejores\ N\ candidatos\ para\ un\ p \in \Omega \tag{2.15}$$

Estos candidatos han sido escogidos utilizando pasos que emplean solo el primer término de energía para mejorar los candidatos mediante un refinamiento de la textura, tal como el paso 3 (Procesar textura) y 5 (Proyección) de la Figura 2.2.

Luego se intenta mejorar estos candidatos aplicando la función de energía antes mencionada. Para ello, se calculan las magnitudes 2.3, 2.5 y 2.6 para todos los píxeles $q \in \Phi(p)$, así como las constantes α, β y γ. Partiendo de estas estimaciones, se formula la suma reseñada en 2.8. Una vez calculada la ecuación 2.8, se actualiza la lista de $\Phi(p)$, tomándose como valor de distancia el dato calculado por la función de energía.

Seguidamente se toma el mejor valor de cada $\Phi(p)$ para cada píxel $p \in \Omega$, es decir $\varphi(p)$, y se actualiza la imagen a tratar. Una vez actualizada la imagen, se vuelve a calcular las imágenes de difusión (v) y coherencia (w) y se repite el proceso antes descrito, y así sucesivamente hasta que se hayan cumplido un número de n iteraciones. Para diferenciar las iteraciones del procesamiento de textura de éstas iteraciones, aquí se habla de n iteraciones en vez de r iteraciones.

La forma en que los píxeles p son seleccionados, es de la misma forma planteada anteriormente, de afuera hacia adentro, siguiendo un esquema de capas y contando con una máscara auxiliar que permita distinguir los píxeles que faltan por procesar de los que no. Una vez procesados todos los píxeles, a la máscara auxiliar le es asignada la máscara original para repetir el proceso de aplicar la función de energía en la siguiente iteración.

El número de iteraciones n para esta parte del algoritmo es un parámetro fijo que puede ser manipulado por el usuario, sin embargo para $n = 10$ se obtienen buenos resultados según [15], pero en *k-Inpainting* esta variable puede tomar otros valores que serán discutidos en el capítulo 4. Es importante destacar, que para la primera iteración del algoritmo la ecuación 2.3

no es calculada, si no que se toma el valor que se obtuvo del paso anterior, el cual ha buscado los mejores candidatos mediante este primer término de energía.

2.3.6. Proyección

En una proyección, es natural pensar que se proyectan cuatro puntos a una imagen grande a partir de un punto de la imagen de menor tamaño. Sin embargo, ese tipo de proyección puede provocar que se procesen puntos de Ω en un orden inadecuado, lo que conllevaría a que se presenten problemas en el algoritmo *k-Inpainting*.

Debido a lo anterior, la proyección se hace a partir de la imagen de mayor tamaño a la de menor tamaño, es decir, se toma un punto $p \in \Omega \subset I$ y se busca el punto correspondiente en I'. Una vez encontrado ese punto en I', se proyectan ahora de la forma $I' \implies I$ los candidatos $q' \in \Phi(p'), p' \in \Omega \subset I'$, donde cada q' proyecta cuatro puntos $q \in \Omega^c \subset I$.

Como no es deseable aumentar la cantidad de candidatos mediante la fórmula 2.14, se calcula cual de los cuatro puntos proyectados es el nuevo candidato para p. El mejor q para cada p, viene a ser $\varphi(p)^0$ para esta imagen y reemplazan en una iteración inicial a los valores incógnitas $p \in \Omega \subset I$, es decir a $I(p) = \varphi(p)^0$.

Posteriormente, como indica la Figura 2.13, se aplica una síntesis de textura con el algoritmo para *Obtener candidatos* con el primer término de energía. Sin embargo, como ya se tiene una buena estimación espacial, el algoritmo es realizado una única vez, es decir no se itera r veces. Para entender mejor este concepto, véase la Figura 2.13

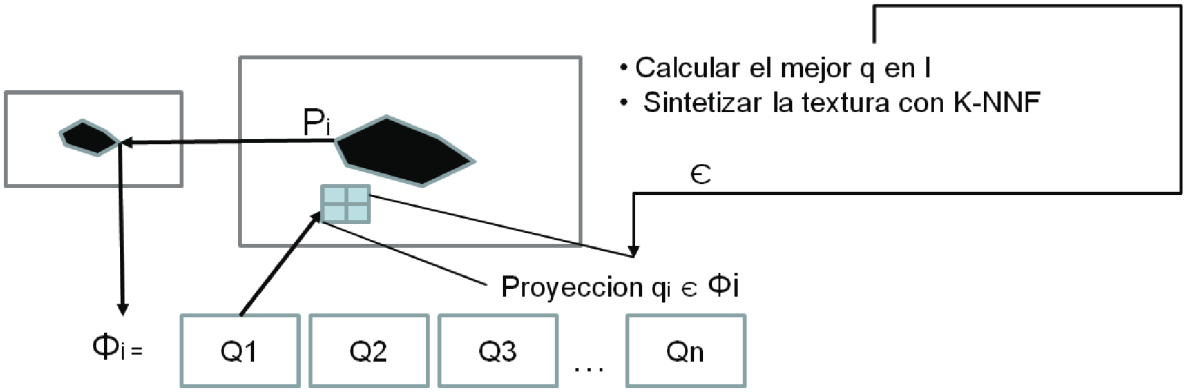


Figura 2.13: Pasos que llevan a cabo la etapa de proyección.

En la Sección de *Obtención de datos*, se menciona que para cada imagen son almacenados los puntos máximos y mínimos con respecto a la imagen anterior, así como sus módulos. Esto es, para hacer una buena correspondencia de puntos en el momento de proyectar. Sea p un punto dentro de $\Omega \subset I$ y sea $pmin$, $pmax$, $pminmod$ y $pmaxmod$ los puntos máximos y mínimos de Ω con sus correspondientes módulos, para buscar el punto p' correspondiente dentro de Ω de I', se realiza el cálculo de la ecuación 2.16, donde m es un punto dentro de la ventana $[0, 0] \times [1, 1]$.

$$p' = (p - m - pminmod)/2 \tag{2.16}$$

Luego de aplicar la proyección, Φ ha sido redimensionado y actualizado con los mejores candidatos para $\Omega \subset I$, los cuales han sido escogidos en primer lugar con la ecuación 2.14 y luego mejorados con la ecuación 2.2. Buscando mejorar los candidatos, se aplica nuevamente la función de energía, tal como se ilustra en el paso 6 y finalmente se obtienen los resultados de la imagen.

2.4. Consideraciones finales

El algoritmo de *Inpainting* aquí planteado se considera híbrido, debido a que se basa en dos trabajos que están concebidos de manera distinta, además de incluir modificaciones a dichos trabajos para que éstos se pudieran acoplar en un mismo algoritmo, que incluye entre otras cosas un proceso de proyección semi-invertido. Nótese que en el Capítulo 1 se hablan de dos tipos de síntesis de textura: basada en píxeles y basada en parches. Los trabajos de Connelly Barnes et al. [51, 9] tratan la síntesis de textura basada en parches, mientras que el trabajo de Bugeau et al.[15] emplea la síntesis de textura basada en píxeles, adoptándo para el proceso de inicialización un campo aleatorio de Markov, la cual es una característica común en los algoritmos de síntesis de textura basada en píxeles. Básicamente, en la propuesta planteada, se sustituye la inicialización mediante un campo aleatorio de Markov, por un algoritmo de síntesis de textura para procesar la misma mediante parches, pero adaptándo este último a su vez a píxeles.

Otro aspecto importante es que el cómputo más fuerte se realiza sobre la imagen pequeña, lo que ayuda a mejorar el tiempo de respuesta, así como a minimizar los cambios bruscos de color entre píxeles vecinos en la imagen original sobre la cual se realiza la proyección. El uso de la pirámide de dos niveles y la etapa de proyección evitan realizar un post-procesamiento de suavizado sobre el área a tratar, empleado por algunos trabajos de investigación en este campo, tal como [15].

Finalmente, cabe acotar que en cada etapa se procesa un píxel a la vez de la parte desconocida de la imagen, y para ciertas imágenes con áreas importantes a tratar, el algoritmo será lento. Por ello, se ha considerado implementar una versión que agilice el proceso con ayuda de hardware de alto rendimiento. En el siguiente capítulo se describirá una versión más ágil de este algoritmo, basándose en las ideas aquí establecidas.

Capítulo 3

Inpainting GPU

En el Capítulo 2 se describe como se implementa el algoritmo propuesto de *k-Inpainting*. Aunque este arroja buenos resultados, es necesario acelerar el tiempo de ejecución del algoritmo. Por ello se recurre al uso de la tarjeta gráfica, la cual ha sido empleada desde el año 2006 aproximadamente para resolver problemas de propósito general. Por ende, antes de describir como se llevó el algoritmo de CPU a GPU, se hablará acerca de algunos aspectos básicos de la GPU.

3.1. Unidad de Procesamiento Gráfico

La CPU (Unidad Central de Procesamiento) es el componente más importante del computador debido a que realiza casi todos los cálculos, tareas y procesos necesarios para tratar y desplegar información, siendo unos de estos procesos el despliegue de gráficos. Sin embargo, la incorporación de un potente procesador de construcción especializada y alto nivel de paralelismo denominado GPU (Unidad de Procesamiento Gráfico), permite separar el despliegue gráfico de los demás procesos, liberando a la CPU de esta carga.

La GPU es un chip o procesador SIMD (*Single Instruction Multiple Data*) diseñado para calcular operaciones relacionadas al despliegue de gráficos en 3D. Este chip integrado con transformaciones, iluminación, configuración y recorte de triángulos, y motores de renderización [52], es capaz de procesar millones de polígonos por segundo, realizar operaciones arbitrarias y programables sobre la data que le es enviada [53], además de procesar en paralelo masivas cantidades de números punto flotante [54].

La arquitectura GPU/CPU se encuentra situada dentro de la clasificación de arquitecturas paralelas[1]. Una arquitectura paralela posee sistemas de memoria distribuida (o compartida) [55] o sistemas de memoria por pase de mensaje [56]. En el caso de la GPU, ésta posee una arquitectura paralela con sistema de memoria distribuida que es capaz de comunicarse a otros sistemas como la CPU. La CPU en un sistema computacional moderno se logra comunicar con la GPU por medio de conectores gráficos, tales como un puerto PCI *Express* o una ra-

[1]Es una ampliación básica de la arquitectura del computador para soportar comunicación y cooperación entre los procesadores.

nura AGP en la tarjeta madre. Esta tecnología junto con la GPU también ha evolucionado a través de los años, debido a que dichos conectores gráficos son los responsables de transferir todos los comandos, texturas y datos de la CPU a la GPU. Para conocer más acerca de la arquitectura de la GPU, ver anexos 5.3.

La programación en la GPU se ha divido en la actualidad en dos vertientes: programación en shaders y programación de propósito general. Un shader es un programa que se ejecuta dentro de la GPU y define cómo los datos son recibidos desde el programa que está siendo procesado por las etapas programables del *rendering pipeline* [57]. La programación sobre shaders permite a los desarrolladores crear diferentes efectos gráficos y el efecto de los *scripts* se perciben directamente en el despliegue, ya que están asociados puramente a modificar vértices, geometrías y píxeles.

En la actualidad, existen principalmente dos (2) APIs empleados para la renderización 3D en tiempo real: DirectX propiedad de la corporación Microsoft y OpenGL soportado por la arquitectura ARB (*Architecture Review Board*), los cuales proporcionan los lenguajes para *shading* (*Shading Language*) HLSL y GLSL respectivamente. NVIDIA también incorporó un lenguaje para shaders llamado Cg que puede ser usado en cualquiera de los dos APIs mencionados [58]. Estos lenguajes están diseñados sobre una sintaxis parecida al lenguaje de programación C y permiten realizar scripts (comúnmente llamados shaders) sobre las etapas programables de *vertex shader*, *geometry shader* y *pixel shader* del *pipeline* gráfico.

Recientemente, por las características del GPU aparece el concepto de GPGPU (General-Purpose Computing on Graphics Processing Units). Este es un concepto asociado al empleo de la GPU para desarrollar aplicaciones de propósito general fuera del render de gráficos por computador, buscando aprovechar su potencia en el cálculo, su gran paralelismo y optimización en cálculos de punto flotante. En la actualidad existen dos APIs que permiten programar bajo este concepto en la GPU: CUDA de NVIDIA y OpenCL. Para la implementación del algoritmo *k-inpainting*, se ha decidido usar CUDA, por lo que se realizará una breve descripción de esta herramienta.

CUDA fue creado por NVIDIA en noviembre de 2006, como una arquitectura de cómputo paralelo de propósito general que permite incrementar el rendimiento de la GPU. Con las herramientas y arquitectura de CUDA los desarrolladores están logrando grandes avances en diversos campos como imágenes médicas, exploración de recursos naturales, creación de aplicaciones como reconocimiento de imágenes en tiempo real, así como la reproducción de video en alta definición. Estos avances se logran a través de lenguajes y APIs estándares tales como, OpenCL, DirectX Compute, C/C++, Fortran, Java, Phyton y .NET de Microsoft, ver la Figura 3.1.

Como se ha mencionado, la GPU es una arquitectura altamente paralela que contiene cientos de núcleos (*cores*) en los cuales se pueden ejecutar colectivamente miles de hilos concurrentes, por lo que es ideal aprovechar al máximo este nivel de cómputo simultáneo [11].

NVIDIA busca de aprovechar el paralelismo de la GPU mediante un modelo y un ambiente de programación paralelo que explote el poder de la unidad de procesamiento gráfico para programaciones no gráficas, de manera de obtener el máximo rendimiento de este chip con el mínimo esfuerzo. Por ello es creada la arquitectura CUDA que permite a los progra-

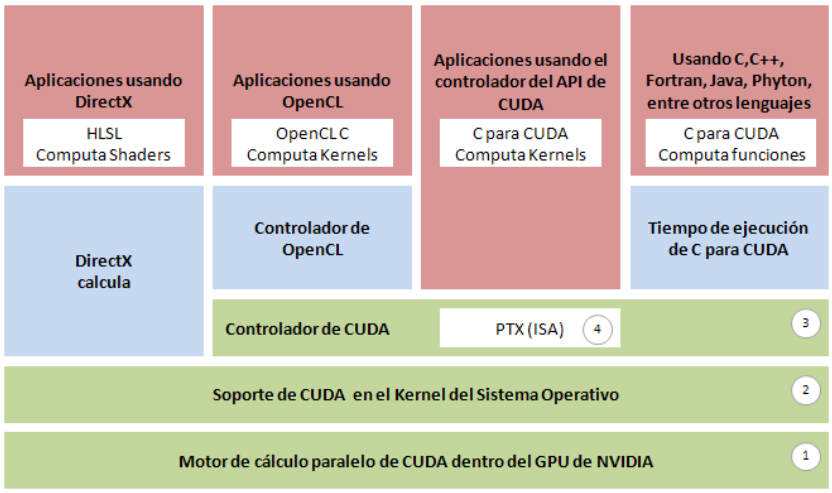


Figura 3.1: CUDA e interacciones con aplicaciones [10].

madores enfocarse en el desarrollo de algoritmos paralelos y no en la implementación de los mecanismos de paralelismo. Este modelo es esencialmente una extensión del lenguaje de programación C, por lo que la implementación de los algoritmos paralelos en CUDA es bastante directa, y requiere de una baja curva de aprendizaje respecto a la sintaxis y semántica del lenguaje por parte de los desarrolladores.

CUDA también está desarrollado para permitir programación heterogénea, en donde las aplicaciones usan el tanto la GPU como la CPU, siendo estos últimos tratados como dispositivos distintos con espacios de memorias privados o propios. Las porciones seriales del programa desarrollado en esta arquitectura se ejecutan en la CPU, a la que se ha denominado *host*, mientras que las porciones paralelas del programa se ejecutan la GPU, que ha sido llamado dispositivo (*device*), como núcleos (*kernels*) computacionales. En otras palabras, un *kernel* es una función que es llamada por un *host* y es ejecutada por un *device* [11]. Sólo un *kernel* puede ser ejecutado en un *device* a la vez, y cuando un *kernel* es ejecutado son lanzados varios hilos concurrentes (hilos de CUDA) encargados de ejecutar el mismo código presentado en dicho *kernel* en paralelo, pero en diferentes regiones de memoria.

Hay algunas distinciones que deben hacerse entre los hilos de CUDA y los hilos de CPU. Los hilos de CUDA son extremadamente ligeros en términos de creación y cambio de ejecución de un hilo a otro. Miles de hilos de CUDA pueden ser creados en solo algunos ciclos de reloj y como resultado no hay sobrecarga (*overhead*) que tenga que ser amortizada durante la ejecución del *kernel*. Dado que en CUDA el intercambio de hilos es de bajo costo, las penalizaciones de tiempo asociadas a un intercambio de hilo cuando se encuentra en el estado suspendido o bloqueado son mínimas.

Cuando un *kernel* de CUDA es lanzado, se ejecuta como un arreglo de hilos paralelos, donde cada hilo corre el mismo código, pero en regiones distintas de memoria que son asignadas a cada hilo y sobre las cuales se ejecuta alguna operación (Figura 3.2). Supóngase que se desea sumar dos arreglos A y B de 100 posiciones cada uno, en este caso, cada hilo i suma la posición $A_i + B_i$ y arroja el resultado en un arreglo C_i. En el caso que se acaba de

describir, cada hilo se ejecuta independientemente, pero esto no siempre es así, por lo tanto CUDA también permite realizar operaciones de cómputo dependientes. La pieza faltante en el ejemplo anterior son los medios de cooperación entre los hilos [11]. La cooperación de los hilos es valiosa por varias razones:

1. Los hilos pueden compartir resultado para evitar computaciones redundantes, de forma similar que sucede en la memorización que emplea la programación dinámica.

2. Los hilos pueden compartir el acceso a memoria, lo cual puede reducir los requerimientos de ancho de banda.

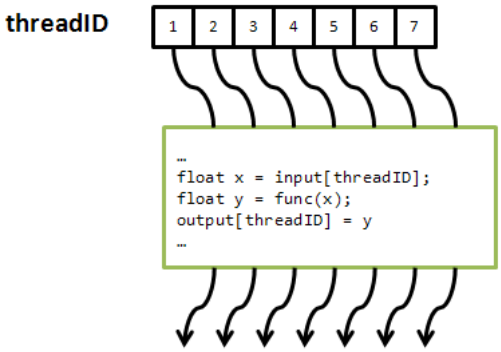


Figura 3.2: Hilos de un kernel [11].

Aún cuando la cooperación es deseable, la colaboración sin límites entre todos los hilos de un *kernel* puede influenciar de forma negativa el rendimiento, siendo a su vez no escalable. Sin embargo, reduciendo la cooperación de los hilos en pequeños grupos, la arquitectura alcanza cooperación y escalabilidad al mismo tiempo. Cuando un *kernel* es lanzado, los hilos son arreglados en una rejilla (*grid*) de bloques de hilos. Los hilos en el mismo bloque cooperan mediante memoria compartida y sincronización mediante barreras, mientras que hilos en bloques distintos no pueden cooperar. Esta ordenación permite a los programas escalar de forma transparente entre distintos modelos de GPU, de tal forma que CUDA distribuye los bloques de un *kernel* de forma apropiada a los multiprocesadores de la GPU, sin importar el número de bloques o multiprocesadores.

Por otro lado, OpenCL es un API de código abierto desarrollado en el 2008 que permite el desarrollo de aplicaciones en distintas tarjetas gráficas, sin importar su su marca comercial.

Teniendo estas bases, se procederá a describir como fue paralelizado el algoritmo *k-Inpainting*.

3.2. Algoritmo *k-Inpainting*

Como se describió en la sección anterior, el GPU tiene un alto nivel de paralelismo, por ello ahora se debe pensar en una implementación paralela en vez de secuencial. La programación concurrente o paralela se basa en la capacidad de diseñar y programar tareas que

puedan ser ejecutadas al mismo tiempo. Con ello, el programador busca algoritmos basados en la descomposición de tareas simultáneas, para problemas que tradicionalmente se han atacado de forma secuencial [59]. Diseñar algoritmos paralelos no es tarea fácil y es un proceso altamente creativo [12]. El diseño involucra cuatro etapas, las cuales se presentan como secuenciales, pero que en la práctica no lo son (ver Figura 3.3). Tales etapas se describen a continuación:

1. Partición: El cómputo y los datos sobre los cuales se opera se descomponen en tareas. Se ignoran aspectos como el número de procesadores de la máquina a usar y se concentra la atención en explotar oportunidades de paralelismo.

2. Comunicación: Se determina la comunicación requerida para coordinar las tareas. Se definen estructuras y algoritmos de comunicación.

3. Agrupación: El resultado de las dos etapas anteriores es evaluado en términos de eficiencia y costos de implementación. De ser necesario, se agrupan tareas pequeñas en tareas más grandes.

4. Asignación: Cada tarea es asignada a un procesador tratando de maximizar la utilización de los procesadores y de reducir el costo de comunicación. La asignación puede ser estática (se establece antes de la ejecución del programa) o en tiempo de ejecución mediante algoritmos de balanceo de carga.

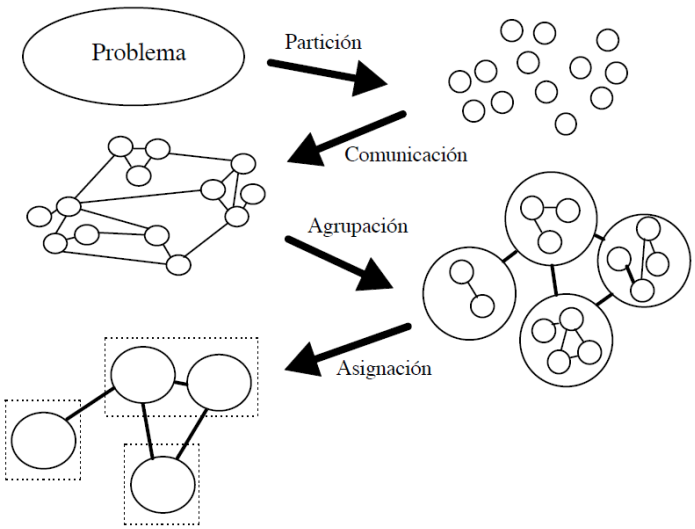


Figura 3.3: Etapas en el diseño de algoritmos paralelos [12].

En la siguiente sección se habla de forma general sobre la paralelización del algoritmo, contemplando las etapas antes discutidas.

3.2.1. Enfoque global del algoritmo

Para este caso, la etapa de comunicación es casi transparente para el programador, debido a que CUDA se encarga de manejar la sincronización y comunicación de los hilos. El programador solo debe indicar cuantos hilos se van a ejecutar, seguir ciertos parámetros que indica el API y estar consciente de algunos conceptos importantes para el desarrollo de la aplicación bajo CUDA, los cuales puede encontrar en [10] y en el anexo 5.3 presentes en este documento.

El diseño del algoritmo es totalmente transparente en relación a la etapa de asignación, ya que el programador no utiliza ningún tipo de instrucción que ayude con esta tarea, es responsabilidad neta del API la asignación de los *cores* de la GPU para diferentes tareas. Sin embargo, si el lector desea conocer el funcionamiento del API a nivel de hardware puede recurrir a [10]. Dicho esto, solo se van a considerar las etapas de partición y agrupación en nuestro enfoque.

En la etapa de partición de datos se trata de subdividir el problema lo más finamente posible, es decir, que la granularidad del problema sea fina. Una buena partición divide tanto los cálculos como los datos. Existen dos formas de particionar la tarea a ser paralelizada:

1. **Partición del dominio:** El centro de atención son los datos. Se determina la partición apropiada de los datos y luego se trabaja en los cómputos asociados con los datos.

2. **Partición funcional:** Es el enfoque alternativo a la descomposición anterior. Primero se descomponen los cálculos y luego se ocupa de los datos.

Estas técnicas son complementarias y pueden ser aplicadas a diferentes componentes de un problema e inclusive al mismo problema para obtener algoritmos paralelos alternativos [59]. En la versión paralela de *k-Inpainting*, se utiliza la partición de dominio, ya que los píxeles de una misma capa $\delta\Omega$ del área a reconstruir son independientes entre sí y se pueden procesar al mismo tiempo. No se pueden procesar todos los píxeles de la zona a reconstruir sobre la imagen, ya que una capa siempre depende de la capa adyacente más externa.

En la fase de partición se trata de establecer el mayor número posible de tareas con la intención de explorar al máximo las oportunidades de paralelismo [59]. Esto no necesariamente produce un algoritmo eficiente ya que el costo de comunicación puede ser significativo. En la mayoría de los computadores paralelos la comunicación es mediante el pase de mensajes y frecuentemente hay que parar los cálculos para enviar o recibir mensajes. Mediante la agrupación de tareas se puede reducir la cantidad de datos a enviar y así reducir el número de mensajes y el costo de comunicación.

En CUDA, el multiprocesador que crea los hilos, lo hace sin ningún tipo de *overhead* por la planificación, lo cual unido a una rápida sincronización por barreras y una creación de hilos muy ligera, consigue que CUDA pueda utilizarse en problemas de muy baja granularidad, incluso asignando un hilo a un elemento, como el caso del píxel de una imagen. Por ello la comunicación que preocupa al programador no es precisamente la que hay entre los hilos, sino la comunicación entre la GPU y la CPU, por ello se trata de hacer el menor intercambio de datos posibles.

CUDA ataca a problemas de granularidad fina, a diferencia de los sistemas paralelos que usan usualmente granularidad gruesa o media, para sistemas con memoria privada y compartida respectivamente. Debido a esto, la agrupación de tareas no se considera relevante en este caso.

El algoritmo paralelo es similar al secuencial, pero tiene algunas variaciones. La Figura 3.4 presenta el nuevo esquema de *k-Inpainting*, donde cada etapa en si misma se ejecuta en paralelo en sus sub-etapas, a excepción de *obtener datos de entrada*. En el diagrama presentado en el capítulo 2 en la Sección 2.1, se describe la obtención de datos. A estos datos se le aplica un procedimiento que los almacena en diferentes listas que luego son enviadas al algoritmo paralelo.

Una diferencia importante con respecto al diagrama secuencial, es la encapsulación de las sub-etapas *K-Propagación* y *Búsqueda Aleatoria* en una etapa llamada *K-NNF*. En la etapa de *Procesamiento de textura*, el ciclo de r iteraciones (en línea punteada roja), indica que ese proceso puede ser ejecutado una vez si se procesa con todas las muestras (implementación GPU T1) o r veces si se paraleliza siguiendo la misma idea presentada en el capítulo 2 en la Sección 2.3.4 (GPU T2). Ambas implementaciones se detallarán posteriormente.

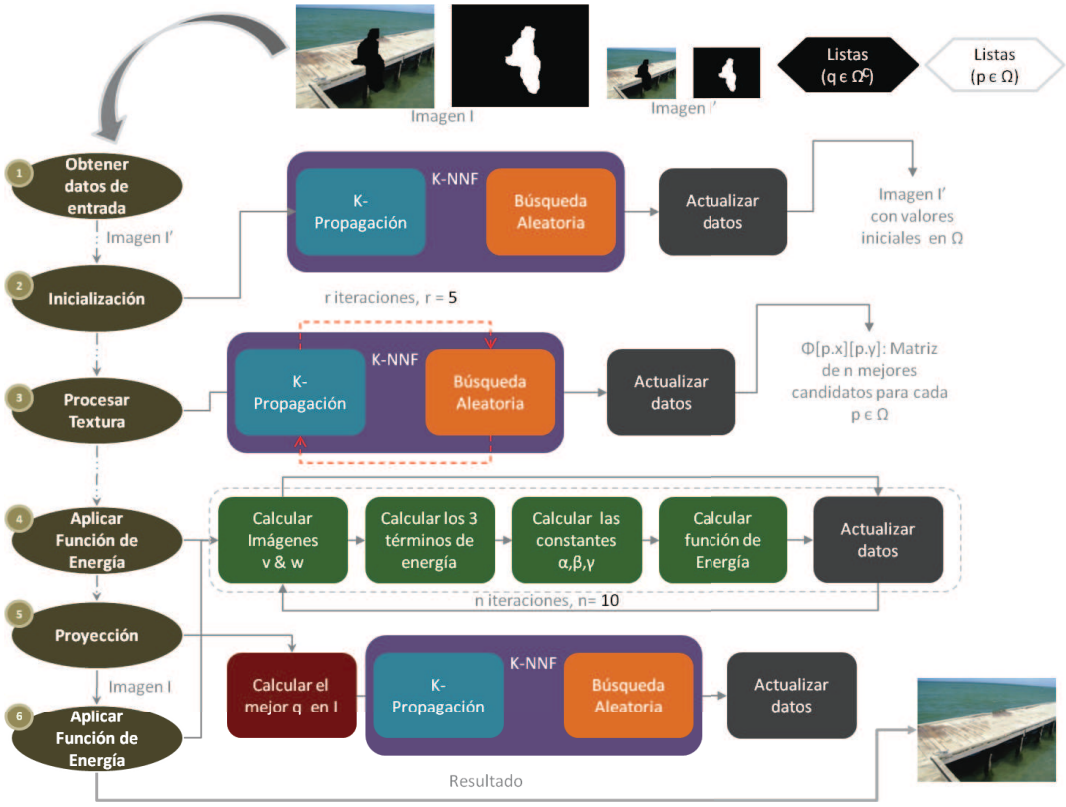


Figura 3.4: Estructura general del algoritmo paralelo.

A continuación se detallarán los datos de entrada del algoritmo paralelo.

3.2.2. Procesamiento de los datos

Los datos de entrada están compuesto por las imágenes I' e I con sus respectivas máscaras, y una serie de listas que serán descritas a continuación:

- P: Lista de los puntos que conforman a Ω.
- X: Lista de puntos candidatos. Es decir, los puntos que se obtienen a partir del muestreo.
- Ψ_1 y Ψ_2: Listas de índices.
- K para el algoritmo K-Propagación.
- N que indica la cantidad de candidatos para cada píxel p.

Como se ha discutido anteriormente, los puntos $p \in \Omega$ son procesados siguiendo un sistema de capas, por ello es necesario que la lista de puntos P este ordenada siguiendo dicho esquema (sección 2.1). Por otro lado, como se mencionó en la sección 2.3.3 del capítulo 2, cada vez que se detecta un borde $\delta\Omega$ se realiza un muestreo, por lo tanto X tiene tantos muestreos como cubiertas constituyen a P. Para diferenciar una capa de otra, y un muestreo de la capa $\delta\Omega_i$ del muestreo de la siguiente capa $\delta\Omega_{i+1}$, se emplea una lista de índices Ψ_1 y Ψ_2. En cada lista Ψ_x, se almacena la posición donde comienza el siguiente lote de muestras (ver Figura 3.5). La variable K es el parámetro que indica cuantos píxeles van a ser propagados en el algoritmo de *K-propagación* (sección 2.3.2). Finalmente N se refiere a la cantidad de mejores candidatos que a lo sumo puede tener cada píxel desconocido en la imagen.

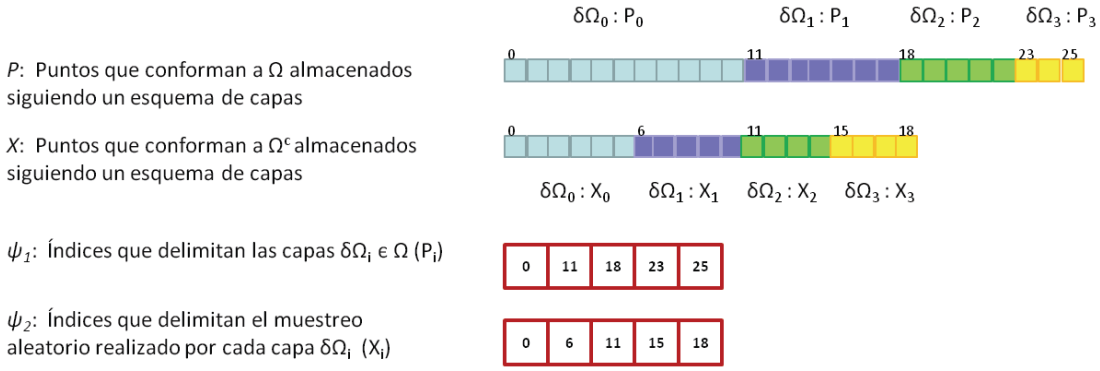


Figura 3.5: Listas de entrada al algoritmo paralelo.

Conociendo como están estructurados los datos de entrada, ahora se procederá a describir algunas funciones comunes que son usadas en cada etapa, en el siguiente apartado.

3.2.3. Consideraciones previas

Como se mencionó anteriormente, los puntos y las muestras se encuentran almacenadas en listas como se describe en la Figura 3.5, con el objetivo de realizar los cálculos de forma eficiente. De esta manera, cada hilo procesa y modifica regiones distintas de memoria y lo hace independiente de los otros hilos durante la ejecución de un *kernel*. Por ello, se emplearán matrices o arreglos, donde cada hilo acceda a una posición de estas estructuras.

En el capítulo 2, se requiere mantener dos datos importantes: la distancia D que indica la diferencia de vecindades entre dos píxeles para alguna métrica, y la posición q de un píxel candidato. Por ello, se crearon estructuras para diferentes propósitos. Durante la ejecución de cada una de las etapas o sub-etapas, se describirá como son empleadas estas estructuras de datos mencionadas a continuación:

- **points**: Es un arreglo 1D que almacena puntos representados como un tipo de dato **int2**. Tiene un atributo llamado *size* para indicar el tamaño del mismo. Por ejemplo, las variable denominadas P y X (mencionadas en la Sección 3.2.2), son de tipo **points**.

- **node**: Se refiere a una estructura que contiene dos arreglos 1D que almacenan en cada posición un tipo de dato **float** para guardar la distancia D y un punto *int2* = (x,y). Cuenta con un atributo *size* que indica el tamaño de la estructura. Esta estructura fue creada para almacenar el mejor candidato $\varphi(P_i)$ para cada punto desconocido de la imagen.

- **node2D**: Se refiere a una estructura que contiene dos arreglos 2D que almacenan una distancia D y un punto candidato q. Cuenta con dos atributos w y h para indicar el ancho y el alto de las matrices. Esta estructura fue pensada para almacenar a lo sumo los N mejores candidatos $q \in \Omega^c$ para cada píxel p o los cálculos realizados por la sub-etapa *K-NNF* mostrada en el diagrama de la Figura 3.4. El valor de w es la cantidad de puntos de alguna capa de Ω y h puede ser de la cantidad de puntos que constituyen un muestreo X_i para alguna capa, o puede tomar el valor de la variable N.

- **matrixf**: Es un arreglo 2D que almacena en cada posición un tipo de dato **float3**. Igualmente posee dos atributos w y h para indicar el ancho y alto de la matriz. Esta estructura fue creada para almacenar las tres distancias correspondientes a los tres términos de energía (2.3, 2.5, 2.6) para cada píxel p. Los valores w y h son descritos en la estructura **node2D**.

En el capítulo 2, antes de describir el algoritmo secuencial, se trataron tres puntos importantes: La implementación del *heap* que almacena los N mejores píxeles seleccionados para sustituir un píxel $p \in \Omega$ (Sección 2.3.1), la aplicación del muestreo sobre la parte conocida de la imagen (Sección 2.3.1), y la definición y combinación de los tres términos de energía (Sección 2.3.1). En esta versión del algoritmo, el muestreo es el mismo realizado que su versión secuencial y almacenado en la lista X, tal como se especificó anteriormente. Al mismo tiempo, en esta sección se emplean los mismos términos de energía los cuales son calculados para un píxel p_i por un hilo T_i.

En la implementación del *heap* descrito en la versión secuencial, se utiliza una estructura de datos. En esta versión, se construye una función que emula el funcionamiento del *heap* para ordenar los candidatos en una tabla auxiliar y guardar los mejores candidatos para cada píxel p. El siguiente apartado describe detalladamente como se implementa y ejecuta dicha función en tiempo real.

Función de ordenamiento

Dadas las siguientes estructuras:

- Dos estructuras **node2D** llamadas *A* y *B*. La definición de los atributos de *A* es como sigue: $w = size(P)$ y $h = size(X)$. Se puede deducir que cada índice en j corresponde a un punto P_j, mientras que cada índice i corresponde a un punto X_i. Así, en cada posición (i,j) se encuentra un valor D y un punto q resultante de un cálculo realizado para buscar candidatos para cada punto $p \in \Omega$, de alguna de las etapas mostradas en la Figura 3.4. La estructura *B* tiene el mismo w de *A*, pero con $h = N$, ya que solo se quieren almacenar los N píxeles $q \in \Omega^c$ que tengan menor diferencia entre su vecindad y la vecindad del píxel $p \in \Omega$. La estructura *B* puede estar vacía o tener candidatos previamente almacenados en una iteración anterior.

- Dos arreglos de **enteros** denominados *sizes* y *sizes_A*. El arreglo *sizes* guarda la cantidad de mejores candidatos para cada píxel p del orificio a llenar. En principio deberían ser N candidatos para cada píxel incógnita, sin embargo, no siempre ocurre de esta forma. Por ejemplo, si al aplicar *K-propagación* hay muchos candidatos repetidos, la cantidad de píxeles candidatos puede ser menor que N. Ahora bien, si la estructura *B* se encuentra llena antes de entrar a esta función, el arreglo *sizes* contiene la cantidad de candidatos por cada punto. En caso que *B* sea vacía, el arreglo *sizes* tendrá cero (0) en todas sus posiciones. El arreglo *sizes_A* contiene la cantidad de píxeles $q \in \Omega^c$ de la variable *A*, en principio todas las posiciones tiene el valor h, pero para la etapa de proyección, esta estructura es vital, ya que en esta etapa *A* tiene los mejores candidatos de la imagen I' y como se señalo anteriormente no siempre se tiene por cada píxel exactamente N candidatos.

- Una estructura **node** llamada *C* de longitud w que contendrá el mejor de candidato $\varphi(p)$ para cada p luego de aplicar la función de ordenamiento.

La función de ordenamiento modifica las estructuras *B*, *sizes* y *C*, para obtener los N candidatos de cada punto $p \in \Omega$ en *B*, ordenados como un *heap* (descrito en la Sección 2.3.1), la cantidad de píxeles q seleccionados en *sizes* y el mejor candidato $\varphi(p)$ en *C* para cada p.

La función trabaja de la siguiente forma. Cada hilo se ocupa de una columna j de *A* y *B*, y de una posición i de *sizes*, *sizes_A* y *C*. Un hilo procesa los datos almacenados en la columna de *A* que le corresponde, emulando un *heap*, y almacena los resultados en *B*. Por lo tanto, se procesan los candidatos de cada $p \in \Omega$ de forma paralela.

El proceso que emula el *heap* utiliza la misma lógica de programación y las mismas operaciones del *heap* explicado en el Capítulo 2, pero sin emplear la tabla hash. Esto es debido a que la existencia de un elemento en una columna de *B* se realiza con búsqueda lineal. Se asume que una columna j de *B* es el *heap* y una columna j de *A* es una lista de elementos de tamaño *sizes_*A_j a insertar en el *heap*, donde solo se tomarán los mejores N candidatos sin admitir repeticiones. Para comprender mejor lo antes descrito, véase la Figura 3.6.

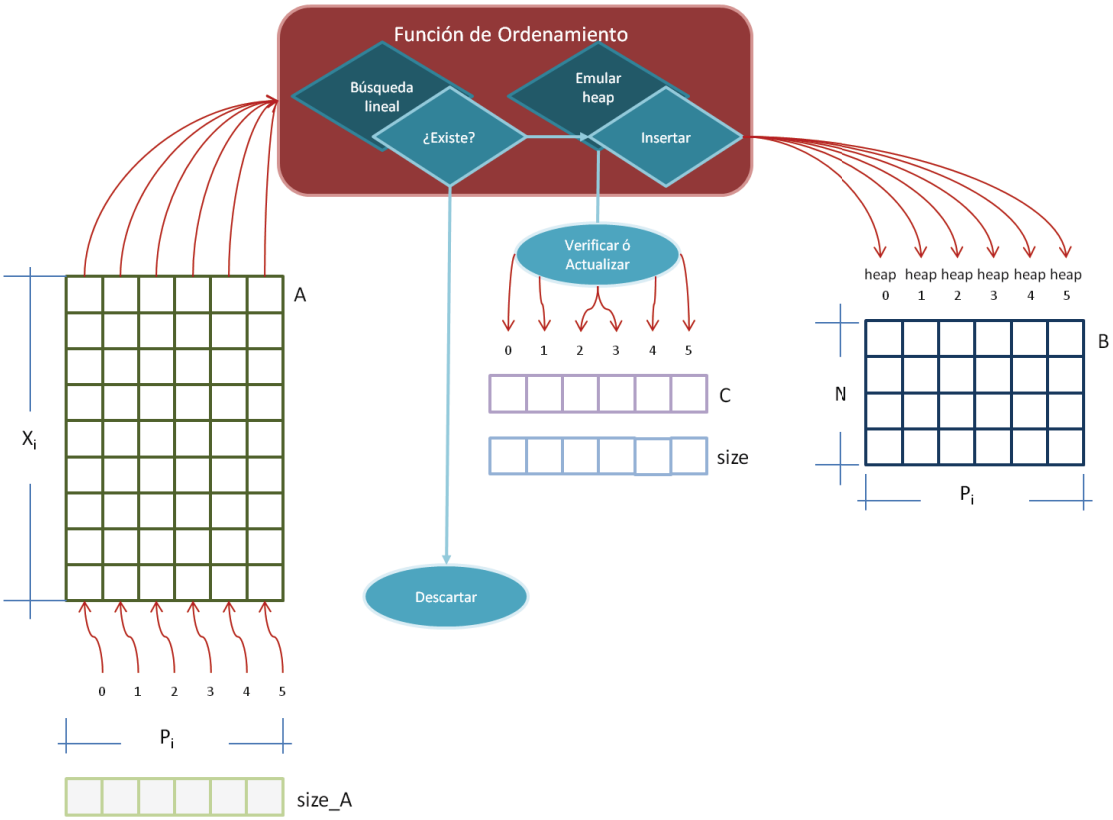


Figura 3.6: Diagrama de función de ordenamiento para una capa de 6 píxeles incógnita y $N = 4$. El hilo con identificador 0, recorre $sizes\_A_j$ elementos de la columna 0 de *A*, y por cada posición en la columna, aplica la función de ordenamiento. El hilo 0 para una posición $A_{i,0}$, evalúa la existencia del punto candidato en B_j mediante búsqueda lineal. Si el elemento existe, se descarta y continúa con la próxima posición $A_{i+1,0}$. Si no existe, procede a insertar el elemento en B_j emulando un *heap* de la Sección 2.3.1 y se actualiza el tamaño del *heap* B_j en $sizes_j$. Si el candidato a insertar es mejor que el candidato almacenado en C_j, entonces se reemplazan los datos almacenados en *C* en esa posición por el resultado actual. El proceso descrito para el hilo con identificador 0, lo realizan todos los hilos de forma paralela.

Esta función es empleada en algunas sub-etapas y etapas que serán descritas a continuación y debido a esto era necesario informar previamente su funcionamiento.

3.2.4. Paralelización de sub-etapas

En esta versión paralela del algoritmo, todas las sub-etapas han sido paralelizadas en la GPU, por ello se procederá a explicar cómo se ha realizado ese proceso para luego resumir su funcionalidad en cada etapa. En otras palabras, la ejecución de una etapa es secuencial, pero los procesos que la llevan a cabo son paralelos.

K-NNF

La sub-etapa *K-NNF* procesa un nivel de *K-Propagación* y realiza una *Búsqueda Aleatoria*. Tanto el algoritmo de *K-Propagación* como el momento de aplicar *Búsqueda Aleatoria* se ejecuta de una forma diferente al algoritmo secuencial.

En el algoritmo secuencial, se toma un píxel p, y por cada píxel q (ya sea muestreados o procesados en Φ) se realiza el proceso de *K-Propagación*. En el algoritmo de *K-Propagación* a partir de un píxel candidato q, K píxeles son propagados en una dirección x y hacia una dirección y, obteniendo $2K$ candidatos. Y sobre cada uno de estos candidatos se aplica el algoritmo de *Búsqueda Aleatoria*. Este proceso se aplica para cada píxel $p \in \Omega$ con respecto a todos los píxeles candidatos, uno a uno.

En la versión paralela, se toman simultáneamente todos los píxeles incógnitas de una capa P_i y sus muestras correspondientes (X_i) o, N candidatos ya procesados en Φ. Nótese, que se toman todas las muestras en el paso de inicialización y se compara luego con los N píxeles $q \in \Omega^c$ en la etapa de *Procesamiento de textura* y en la etapa de proyección (ver Capítulo 2).

A cada candidato se le suma una posición k de la forma ($\pm(0 \leq i \leq K)$,0) en dirección x, o de la forma (0,$\pm(0 \leq j \leq K)$) en dirección y, y en esa nueva posición $q + k$ se aplica *Búsqueda Aleatoria*. Lo anterior se aplica de forma simultánea a todos los candidatos con respecto a todos los píxeles $p \in \delta\Omega_i$. Este proceso se repite secuencialmente $2K + 1$ veces, es decir, cada nivel representa una posición k, resultante de la propagación. Entre cada nivel de los k niveles, se aplica una función de ordenamiento (ver Sección 3.2.3). Como resultado de este proceso, para aplicar el algoritmo de *Búsqueda Aleatoria* no se requiere esperar a que finalice el algoritmo de *K-propagación* como sucede en la versión secuencial, si no que en se aplica a la vez que se va propagando el candidato en cuestión.

Otra forma de describir lo que se ha discutido en el párrafo anterior, es la que sigue: Dada una estructura tipo **node2D** llamada *Aux* con ancho $w = size(P_i)$ y alto $h = size(X_i)$ o $h = N$. En dicha estructura se almacenan las distancias D (resultantes de la diferencia entre dos vecindades, utilizando la ecuación 2.14 ó la 2.3) y el candidato q en cada nivel k del algoritmo *K-propagación*. Un hilo con identificador de coordenadas (j,i), selecciona dos puntos:

- Un punto desconocido guardado en la posición j de una capa de Ω ($P_i : \delta\Omega$), es decir accede a $P_i[j]$.
- Un punto candidato almacenado en la posición i de X_i si se está trabajando con un muestreo para una capa $\delta\Omega_i$; ó se toma un punto candidato Φ[i][j] si se está tratando a los candidatos ya procesados y almacenados en una matriz Φ para un píxel p en j y un píxel q en i.

Ahora bien, dado el punto k, cada hilo toma su candidato q y le suma la posición k, obteniendo un nuevo candidato $kq = q + k$. A dicho candidato se le aplica el algoritmo de *búsqueda aleatoria* explicado en la sección 2.3.2 y la distancia y el punto obtenido a partir de dicho algoritmo es almacenado en la fila i con respecto a la columna j. El valor de i y j es obtenido a partir del identificador (j,i) del hilo, tal como lo indica la Figura 3.7.

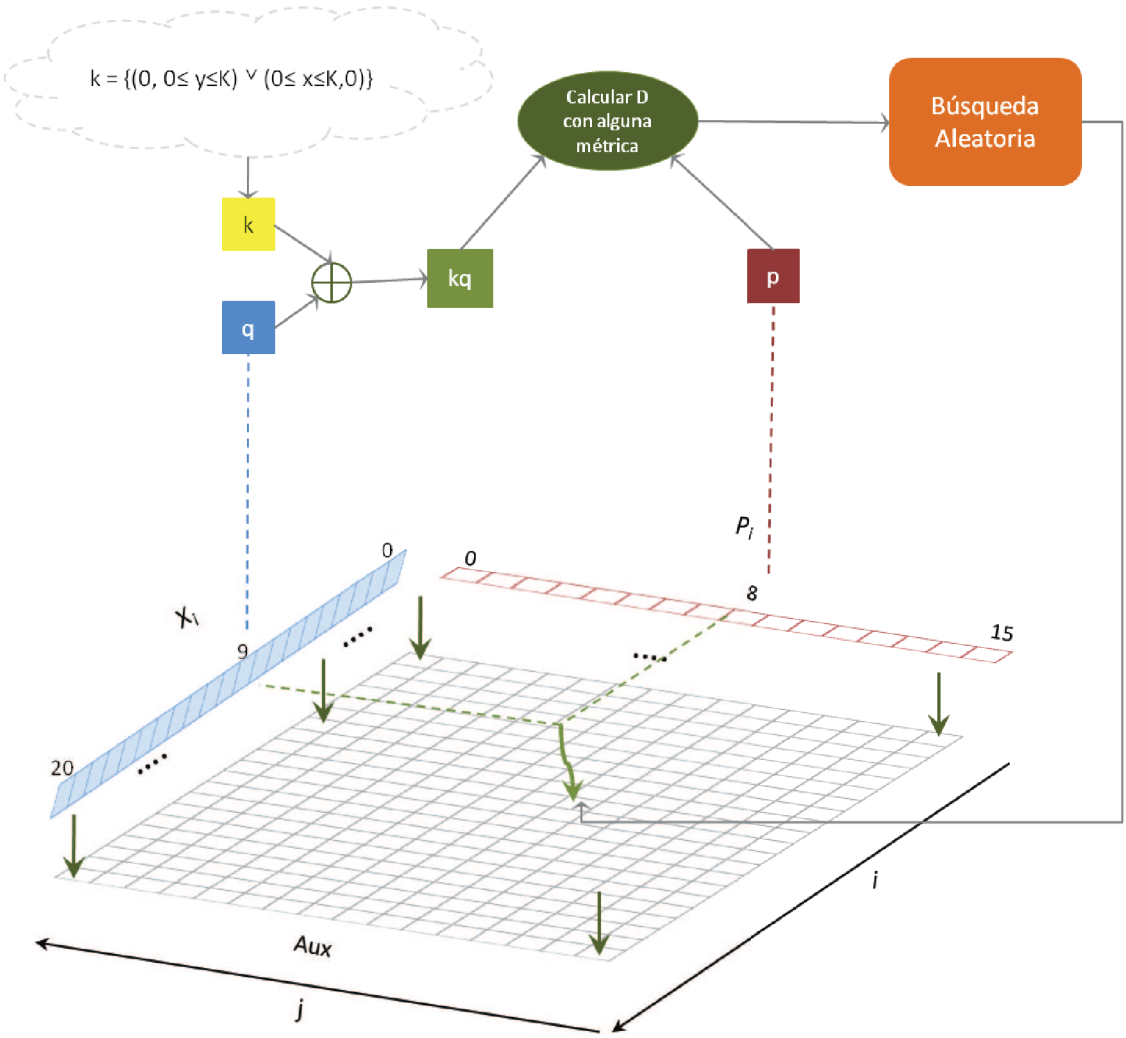


Figura 3.7: Procesar un nivel de *K-Propagación* en paralelo: El punto k puede tomar los valores indicados en la nube. En cada nivel de *K-Propagación*, el k es diferente y la llamada de los niveles es secuencial. Cada hilo, toma su punto q correspondiente en X_i, así como el punto p que le corresponde indexar en P_i. A q le suma el valor de k (el primer nivel siempre es $k = (0, 0)$ para no descartar el candidato que se tiene almacenado) y a este valor se le calcula la distancia D y este último punto es la posición en la imagen ι a tratar que ingresa al procedimiento *búsqueda aleatoria*. Finalmente, el resultado es guardado en la posición (j,i) de *Aux*.

Una vez comprendida la paralelización para un nivel k, es posible ver la variación propuesta del algoritmo *K-NNF* en el Algoritmo 4. Para este código se usa la lista de muestras

X_i, empleada en la implementación *GPU T1*. Sin embargo se puede utilizar una matriz Φ con candidatos ya procesados por esta u otra etapa para la implementación *GPU T2*.

Algoritmo 4 Algoritmo Pseudo-formal K-NNF

```
1: procedure K-NNF(points P_i,X_i; node mejorCandidato; node2D Nmejores; int K, N, * sizes)
2:     node2D Aux;
3:     int size_Aux;
4:     Llenar size_Aux con size(X_i) en todas sus posiciones
5:     int dir_x, dir_y;
6:     Escoger una dirección aleatoria en dirección x y y, y guardarla en dir_x y dir_y respectivamente
7:     para k = 0 hasta k <= K hacer
8:         int2 k_y = (dir_y %2==0)? (0,-k):(0,k);
9:         int2 k_x = (dir_x %2==0)? (-k,0):(k,0);
10:        ProcesarNiveldeKpropagación(P_i, X_i, k_x,Aux);
11:        FunciónOrdenamiento(Aux,Nmejores,size_Aux,sizes,N);
12:        si k > 0 entonces
13:            ProcesarNiveldeKpropagación(P_i, X_i, k_y,Aux);
14:            FunciónOrdenamiento(Aux,Nmejores,size_Aux,sizes,N);
15:        end si
16:    end para
17: end procedure
```

El Algoritmo 4 se aplica tantas veces como capas tenga Ω. La *función de ordenamiento* explicada en la Sección 3.2.3, permite obtener los mejores candidatos después de procesar los $2K+1$ niveles de *K-Propagación*. Luego de aplicar este proceso en el algoritmo, hay que trasladar los píxeles seleccionados como mejores para cada p a la posición de dicho píxel. La forma en que la imagen es actualizada, se contempla en el siguiente apartado.

Actualizar datos

Dada una estructura tipo **node** denominada $Mejor\ q$ y una imagen ι con una máscara M. Se generan tantos hilos como sea la longitud de la estructura $Mejor\ q$. La longitud de esta estructura indica la cantidad de puntos p en alguna cubierta de Ω. Cada hilo generado, con su identificador idx indexa un punto en P_i ($P_i[idx]$), luego toma el candidato q en la posición idx de $Mejor\ q$ y reemplaza el color RGB de píxel p por el color RGB del píxel q, tal como muestra la Figura 3.8.

Calcular Imágenes difusión y coherencia

Las imágenes de difusión y de coherencia se generan aplicando las ecuaciones 2.4 y 2.7 respectivamente en cada píxel de la imagen ι a tratar y recopilando el resultado en las

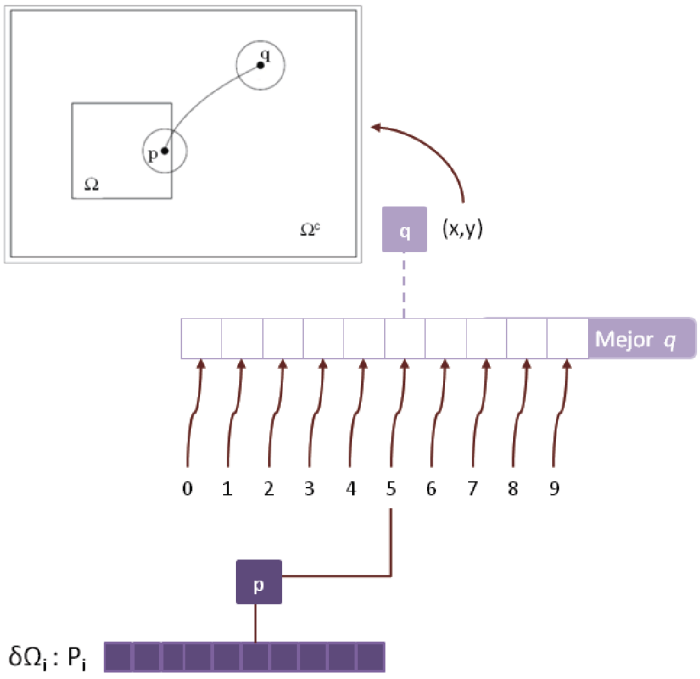


Figura 3.8: Actualizar datos en imagen ι: Los hilos son identificados del 0 al 9, para una capa de 9 puntos incógnitas. Cada hilo procesa una posición del vector *Mejor q*.

imágenes de difusión y coherencia antes mencionadas, las cuales se encuentran vacías en el estado inicial.

La región crítica a tratar es aquella donde se encuentra el orificio. Por ello se crean dos estructuras vacías tipo **matrixf** de las dimensiones del área a tratar. Las dos estructuras corresponden a la imagen v para la difusión e imagen w para la coherencia en el espacio L*a*b. Luego, se generan tantos hilos como píxeles cubra el área ξ de la Figura 3.9, y cada hilo con identificador (i,j) procesa el píxel en la posición (i,j) de las estructuras v y w, tal como lo indica la Figura 3.9. Para indexar correctamente los píxeles correspondientes en ι, el identificador del hilo se desplaza para encontrar la posición correcta ($p = \iota[i+pmin.y][j+pmin.x]$).

Calcular los términos de energía

Teniendo las imágenes de coherencia (w) y difusión (v), es posible calcular los términos de energía de las ecuaciones 2.5 y 2.6. Las ecuaciones 2.3, 2.5 y 2.6 deben ser aplicadas a cada píxel p a tratar con respecto a los mejores N candidatos recopilados en una estructura de tipo **node2D** llamada $Nmejores$. Cada columna j de la estructura $Nmejores$ contiene a la sumo N mejores candidatos (la cantidad de candidatos se encuentra en un arreglo de enteros que guarda la cantidad de píxeles $q \in \Omega^c$ recopilados para cada $p \in \Omega$) para cada punto $P_i[j]$.

Para almacenar las distancias que resultarán del empleo de las ecuaciones antes mencionadas, se construye una estructura tipo **matrixf** denominada Φ con las dimensiones de $Nmejores$, es decir $w = size(P_i)$ y $h = N$. Para paralelizar el proceso descrito en el párrafo

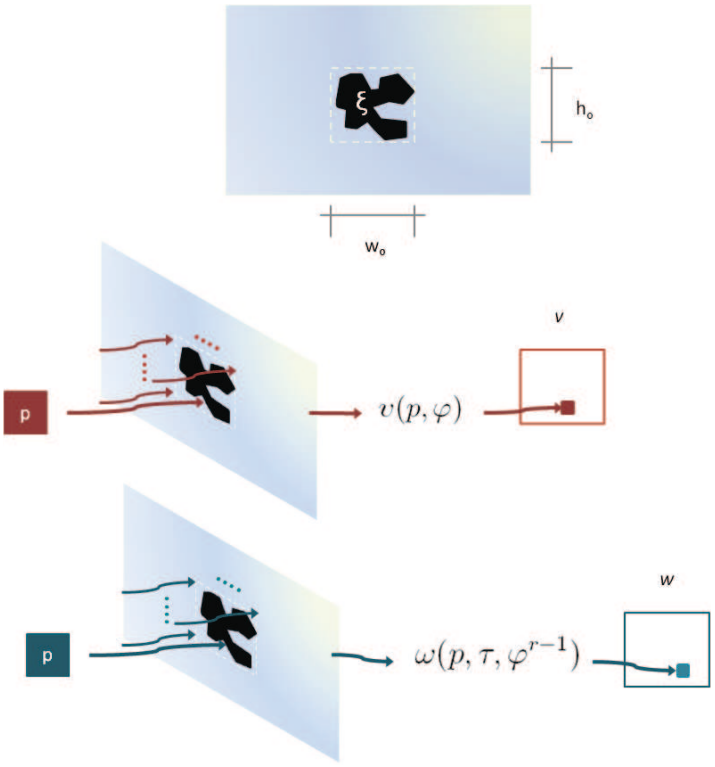


Figura 3.9: Calcular las imágenes de difusión (v) y coherencia (w).

anterior, se generan tantos hilos (con identificador (j,i)) como posiciones tenga la estructura $Nmejores$. Cada hilo indexa con j un punto desconocido p en P_i ($P_i[j]$) y el candidato que le corresponde con el índice idy en la columna idx de $Nmejores$, calcula tres términos de energía y los almacena en la fila i con respecto a la columna j de Φ. Este proceso se puede visualizar en la Figura 3.10.

Calcular las constantes α, β y γ

En la sección 2.3.1 se definieron las constantes α, β y γ en 2.10, las cuales son dependientes de las propiedades de cada píxel de la imagen con respecto a los tres términos de energía. Estas propiedades han sido definidas en la ecuación 2.9. Como aquí se contemplan tres términos de energía, estas propiedades han sido llamadas m_1, m_2 y m_3 en relación a los términos de síntesis de textura, difusión y coherencia respectivamente. Tal como se mencionó en el capítulo 2, m_1, m_2 y m_3 corresponden al mínimo valor del término de energía que le corresponde.

Entonces, se debe calcular m_1, m_2 y m_3 para cada píxel p del área a tratar. Para ello, se crean tantos hilos como tenga alguna capa de Ω que se esté procesando. Cada hilo secuencialmente examina las tres distancias almacenadas en la estructura Φ contemplada en el apartado anterior (*Calcular los términos de energía*) y verifica si el valor de cada término es menor al que se encuentra almacenado. Por ejemplo, al seleccionar el primer término, se debe verificar

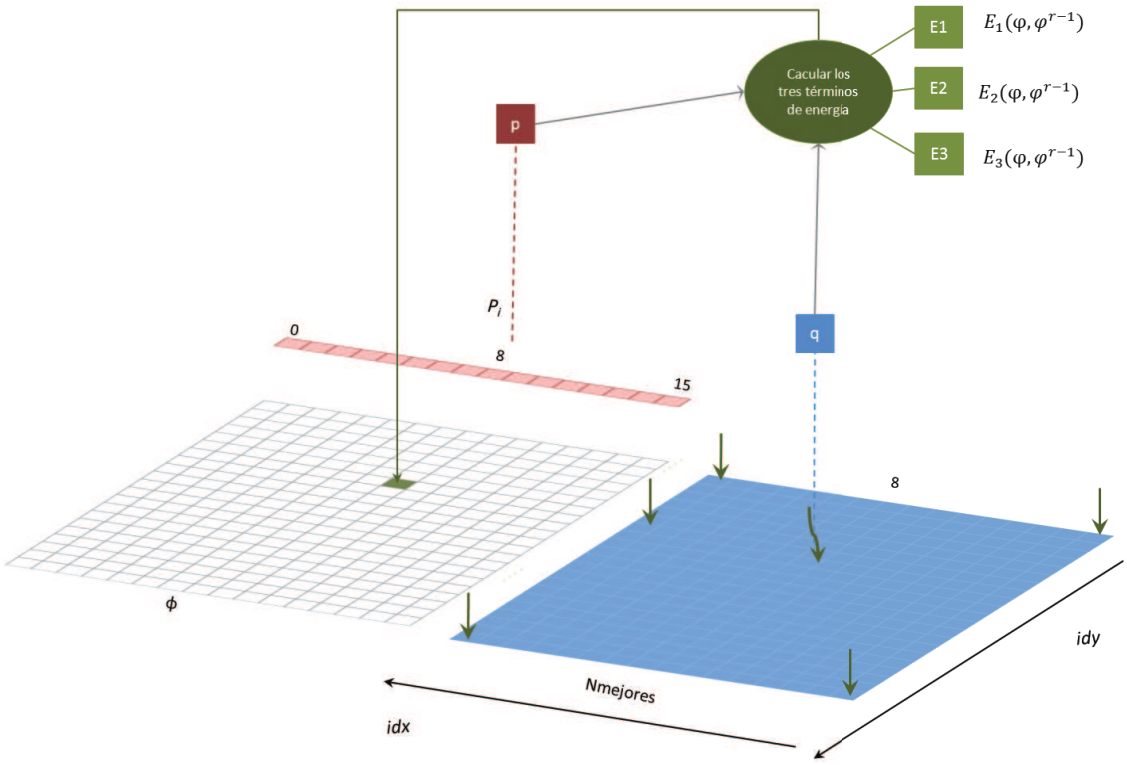


Figura 3.10: Calcular los términos de energía: Dada una estructura **points** P_i, las imágenes v, w e ι, una matriz **node2D** $Nmejores$ que tiene los mejores N candidatos en cada columna j para cada punto $P_i[j]$, un arreglo de enteros con la cantidad de candidatos para cada punto p. Cada hilo calcula los tres términos de energía antes mencionados y los almacena en Φ.

si es menor al valor contenido en m_1, si es así entonces se reemplaza el valor existente en m_1 por la distancia originada del primer término para ese candidato. Seguidamente, el hilo realiza el mismo procedimiento para los otros dos términos de energía almacenados en esa posición con respecto a m_2 y m_3. Antes de ejecutarse esta sub-etapa, los valores m_1, m_2 y m_3 son inicializados con ∞.

Luego que un hilo ha procesado la columna de Φ que le corresponde, calcula el valor de α, β y γ a partir de los valores m_1, m_2 y m_3 obtenidos y los almacena en la posición que le corresponde en arreglos tipo **float** llamados igualmente α, β y γ, como se puede observar en la Figura 3.11.

Calcular función de energía

Para calcular la función de energía se necesitan las constantes α, β y γ contempladas en la sección 2.3.1 del capítulo 2. Una vez halladas estas constantes se puede calcular la ecuación 2.8 que combina los tres términos de energía para cada píxel p a tratar con respecto a los mejores N candidatos recopilados para ese punto p en ese momento. Por ello, también es necesario contar con las distancias correspondientes a cada término de energía almacenados en

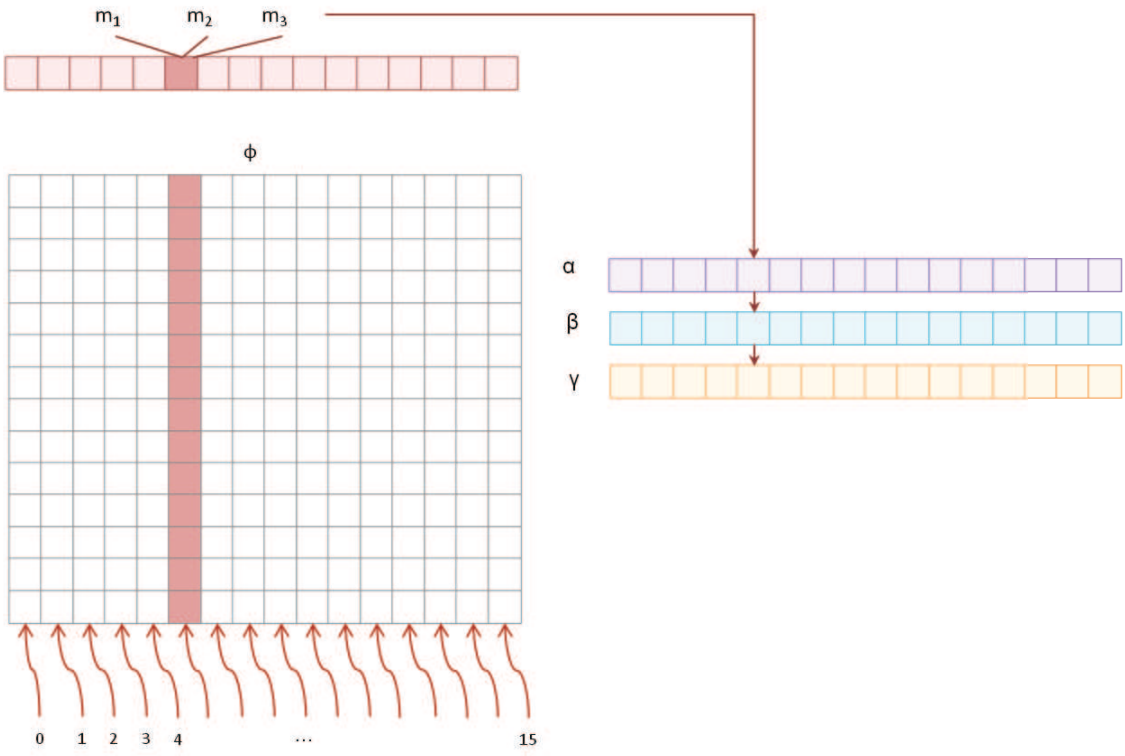


Figura 3.11: Calcular α, β y γ: Cada hilo verifica secuencialmente una columna y almacena la menor distancia de cada término de energía en m_1, m_2 y m_3 para esa columna de Φ.

Φ y los N mejores candidatos para cada píxel almacenados en la estructura $Nmejores$. Por otro lado, para recopilar la nueva distancia generada luego de aplicar la función de energía, se crea una estructura con las mismas características de la estructura $Nmejores$, denominada Φ_{aux}. En cada posición de Φ_{aux} un hilo almacena el valor que se obtiene luego de aplicar la métrica mostrada en la ecuación 2.8 para de un candidato q obtenido de la columna j de $Nmejores$ y el punto p obtenido de la lista de puntos P_i en la posición j. La Figura 3.12 muestra gráficamente la paralelización de esta sub-etapa.

Posteriormente se trasladarán la información en Φ_{aux} a $Nmejores$, pero ordenadas como un *heap* del capítulo 2.

Calcular el mejor q en I

En la sección 2.3.6 se explicó como se proyectaban los candidatos de la imagen I' a la imagen I para un píxel $p \in \Omega$. La proyección tiene dos pasos: Calcular el mejor q en I y sintetizar la textura con *K-NNF*. Como se ha mencionado anteriormente, cada píxel tiene a lo sumo N mejores $q \in \Omega$, cada uno de esos píxeles se proyectan, pero al proyectarse se hallan cuatro (4) puntos en I, de los cuales se debe tomar el mejor de ellos (menor diferencia de vecindad con respecto al píxel p a tratar). En la versión secuencial, se toma un píxel p, se calcula el mejor q para cada candidato uno a uno, luego se procesa el siguiente píxel p con

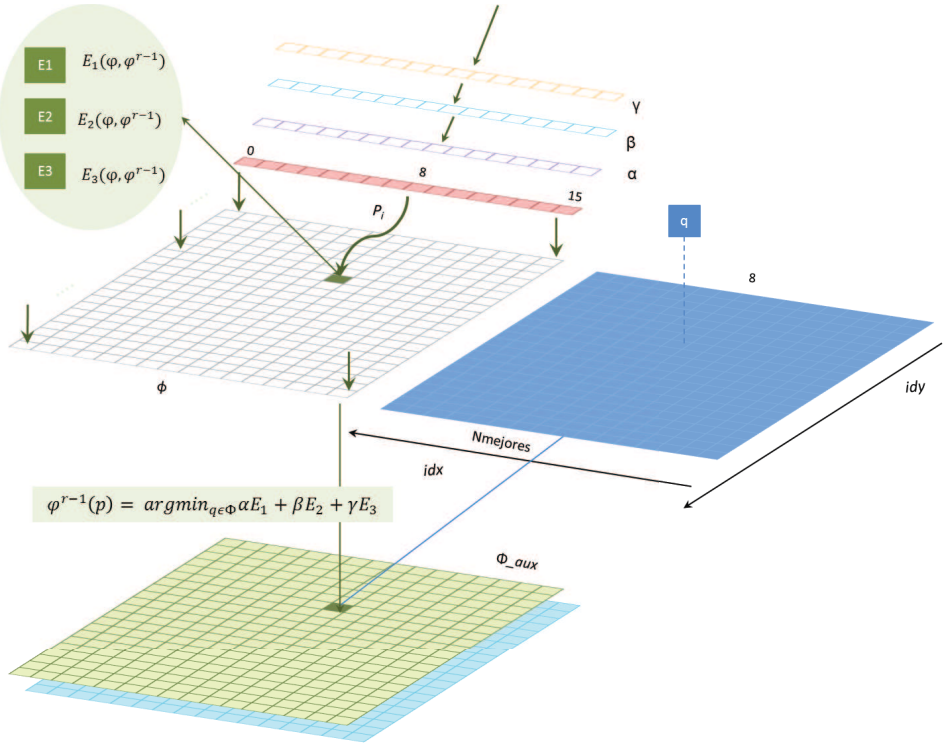


Figura 3.12: Cálculo de la función de energía.

los candidatos que le corresponde y así sucesivamente. En la versión paralela, cada hilo se ocupa de los candidatos del píxel p que indexa en P_i para una capa de Ω.

3.2.5. Resumen del algoritmo

A continuación se explicarán las sub-etapas antes descritas.

Etapa de Inicialización:

Para cada capa $\delta\Omega_i \subset \Omega$ se aplica el algoritmo K-NNF (ver sección 3.2.4) con las listas de puntos P_i y las listas de muestras X_i. El resultado queda en una estructura tipo **node2D** con los N mejores candidatos y una estructura tipo **node** con el mejor candidato para cada píxel incógnita en ese momento. Los candidatos son hallados con la métrica SSD de la ecuación 2.14. Luego de aplicar el algoritmo antes mencionado, se ejecuta la sub-etapa actualizar datos con los valores almacenados en la estructura que tiene el mejor candidato (llamado anteriormente $Mejorq$).

Etapa de Procesamiento de textura:

En la versión presentada en el capítulo 2, se aplican los procedimientos de *K-propagación* y *búsqueda aleatoria* r veces, lo que se traduce en la Figura 3.4 en realizar la sub-etapa

K-NNF r veces, empleando una estructura auxiliar Φ (que tiene la misma información de $Nmejores$) para conocer los candidatos a tratar , permitiendo actualizar la estructura $Nmejores$ sin perder los datos obtenidos en una etapa o iteración previa. Al igual que en el algoritmo secuencial, se emplea la métrica correspondiente al primer término de energía (ver ecuación 2.3) para esta etapa.

En la versión secuencial, aplicar el algoritmo de *K-propagación* y *búsqueda aleatoria* en esta etapa con todas las muestras, en vez de N candidatos ya escogidos, es muy costoso. Sin embargo, para esta versión paralelan tiene más tiempo computacional iterar r veces con N mejores candidatos obtenidos en la etapa anterior, que aplicar el algoritmo con todas las muestras una sola vez, utilizando la métrica antes mencionada. Debido a lo anterior se ha colocado una línea roja punteada en el diagrama de la Figura 3.4, ya que se puede aplicar el *Procesamiento de textura* (ver Sección 2.3.4) fielmente a la versión secuencial o con la variación propuesta. Debido a que se obtienen los mismos resultados visuales con tal variación, y en menor tiempo, el *Procesamiento de textura* para este algoritmo paralelo ejecuta la sub-etapa *K-NNF* una vez con todas las muestras de entrada junto con la ecuación 2.3, en vez de emplear Φ.

Como en la etapa anterior se actualiza la imagen con el mejor candidato obtenido para ese momento de la ejecución del algoritmo, la imagen ya tiene el valor del color de $\varphi(p)$ para cada píxel p que constituye el área a reconstruir, este valor es necesario para el primer término de energía utilizado en esta etapa. Luego se actualiza nuevamente la imagen.

Aplicar función de energía

Como lo indica la Figura 3.4, en *Aplicar la función de energía* se sigue la secuencia de sub-etapas allí indicadas, las cuales han sido explicadas anteriormente. Sin embargo, antes de actualizar la imagen, se ejecuta la *función de ordenamiento* sobre la estructura que contiene los N mejores candidatos hallados con la distancia D que corresponde a al empleo de la ecuación 2.8. Luego de aplicar la función antes mencionada, se tiene en una estructura tipo **node** el mejor candidato para cada píxel p, la cual es necesaria para ejecutar la sub-etapa *actualizar datos*. Este proceso se ejecuta n veces.

Proyección

En la etapa de proyección, se ejecuta en primer lugar la sub-etapa *calcular el mejor* q *en I* con la métrica SSD mencionada previamente, de la cual se obtiene en una estructura tipo **node2D** la proyección de los N mejores $q \in \Omega^c$ hallados en I'. Luego de ejecutar esta sub-etapa, se aplica la función de ordenamiento y se actualiza la imagen. Seguidamente se aplica el algoritmo de *K-NNF* empleando el primer término de energía y con una matriz Φ que es una copia fiel de los N mejores candidatos obtenidos en el paso anterior (*calcular el mejor* q *en I*).

Finalmente se aplica la función de energía sobre la imagen I.

3.3. Consideraciones finales

En todas las sub-etapas descritas en este capítulo se ha hablado de paralelizar procesos para los puntos P_i que pertenecen a una capa. Esto es, por lo comentado en el enfoque global de esta versión (ver Sección 3.2.1) que indica que las capas son dependientes de la capa más externa que le sigue. En consecuencia, las sub-etapas antes descritas se repiten tantas veces como cubiertas tenga el área de la imagen a tratar, antes de pasar a la siguiente sub-etapa o etapa.

Se puede notar que al paralelizar el algoritmo secuencial, se realizaron varias modificaciones. Las modificaciones importantes están relacionadas a la aplicación de los algoritmos *K-Propagación* y *Búsqueda Aleatoria*, los cuales en esta versión coexisten y trabajan entre sí de una manera diferente, como se pudo notar en la descripción del algoritmo *K-NNF*.

La implementación *GPU T1* y *GPU T2* se diferencian en la etapa de *Procesamiento de textura*. Donde la implementación *GPU T1* utiliza las muestras y realiza una sola iteración, mientras que la implementación *GPU T2* emplea la matriz Φ con los mejores candidatos obtenidos para ese momento y realiza r iteraciones, al igual que en la versión secuencial.

Por ende, una modificación importante se encuentra en la implementación *GPU T1*, ya que en el algoritmo final de esta versión no se trabaja con los N mejores candidatos r veces, si no de nuevo con todas las muestras de entrada aplicando la misma métrica de la versión secuencial, lo cual implica que se están procesando más candidatos y que en algunos casos podría arrojar mejores resultados. Sin embargo la comparación de resultados visuales entre *GPU T1*, *GPU T2* y CPU para la propuesta híbrida de este documento será discutida en el próximo capítulo.

Capítulo 4

Pruebas y Resultados

En capítulos anteriores, se detalla la implementación del algoritmo *k-Inpainting* en sus dos versiones. Ahora es necesario poner a prueba dichas versiones para evaluar su rendimiento y determinar parámetros adecuados que ayuden a lograr el objetivo de la técnica. En el transcurso de este capítulo se presentarán las diferentes pruebas realizadas para reconstruir el área desconocida de cada imagen, así como el ámbito de ejecución de las mismas, para finalmente realizar un análisis de los resultados obtenidos.

4.1. Ambiente de pruebas

En este apartado se específica el entorno de hardware y software en el cual se realizaron pruebas.

4.1.1. Requerimientos de hardware

El algoritmo en su versión CPU fue probado en tres equipos con especificaciones de hardware distintas, las cuales se pueden observar en la Tabla 4.1. En la versión de GPU, las características de la tarjeta gráfica son un factor muy importante, ya que sobre dicho hardware es donde se explota el paralelismo, por ello en la Tabla 4.2 se indica que tarjetas fueron utilizadas para efectuar las pruebas, haciendo su correspondencia en el campo *sistema* a los equipos especificados en la primera tabla.

Sistema	Procesador	Memoria RAM	Sistema Operativo
1	Intel Core i3 3.07Ghz	4 GB	Windows 7
2	Intel Core 2 Quad 2.40 GHz	4 GB	Windows XP
3	Intel Pentium 4 3.20 GHz	1 GB	Windows XP

Tabla 4.1: Descripción de los sistemas empleados en los experimentos.

El equipo debe contar con al menos 68 MB de disco para que el algoritmo pueda ser ejecutado en su versión secuencial. Para la versión paralela, se necesita poder contar además

Sistema	Tarjeta	Memoria	Núcleos de CUDA	Capacidad
1	GeForce GTX 470	1280 MB	448	2.0
2	GeForce GT 8800	640 MB	112	1.1
3	GeForce GT 9600	512 MB	64	1.0

Tabla 4.2: Descripción tarjetas gráficas utilizadas en los experimentos.

con una tarjeta gráfica a partir de la GeForce 8800 en adelante. Adicionalmente para la versión paralela se necesita 1.36 GB de disco duro para instalar el API sobre el cual se ejecuta el programa.

4.1.2. Requerimientos de software

Los requerimientos de software son los siguientes:

- Sistema Operativo Windows XP en adelante
- Microsoft .NET Framework, versión 3.5 en adelante

El sistema fue desarrollado en Microsoft C# para la interfaz de usuario y varios módulos del programa en CPU. También se codificaron módulos en C++ que se integran al programa principal. Estos módulos son incorporados mediante librerías de enlace dinámico (DLL). El ambiente de desarrollo fue Microsoft Visual Studio 2008, junto a la librería EmguCv para el manejo de imágenes.

El algoritmo en GPU desarrolla todos sus módulos en C++ en el ambiente de desarrollo de Microsoft Visual Studio 2008. Para la implementación en paralelo del algoritmo se utiliza el API CUDA, que puede emplearse desde su versión 3.1 en adelante.

Para los sistemas antes mencionados se utilizaron diferentes versiones de software en cuanto a las librería EmguCV y la versión del SDK de CUDA, las cuales se específican a continuación en la Tabla 4.3.

Sistema	Software	Versión
1	EmguCV	2.1.0.707
1	CUDA *Toolkit*	3.2
2	EmguCV	2.1.0.793
2	CUDA *Toolkit*	3.1.1
3	EmguCV	2.1.0.793
3	CUDA *Toolkit*	3.0.1

Tabla 4.3: Versiones de software para los equipos empleados en las pruebas.

4.2. Descripción de los escenarios

Se efectuaron experimentos en diferentes niveles. El primer nivel hace un estudio a fondo de los parámetros, realizando tantas ejecuciones como resulten de la combinación de todos éstos. Luego, en un segundo nivel se elaboran pruebas con las mejores combinaciones obtenidas de los experimentos de primer nivel en 4 imágenes distintas. Por último, a partir de las pruebas anteriores se toma un nuevo subconjunto de combinaciones de parámetros, para realizar pruebas sobre 9 imágenes en la versión paralela del algoritmo. Antes de revisar a fondo los experimentos efectuados, se especificarán los parámetros del algoritmo.

4.2.1. Parámetros

Existen diferentes parámetros que afectan el resultado luego de aplicar el algoritmo en cualquiera de sus versiones. La mayoría de éstos han sido mencionados previamente en los capítulos 1 y 3. Algunas de éstas variables pueden tener influencia significativa en el producto final, por ello las mismas se retoman y se describe que valores pueden tomar a continuación:

Lado de la ventana (L)

El lado de la ventana está relacionado con el sistema de vecindad, tal como se muestra en la Sección 1.2. En esta propuesta se emplea un sistema de vecindad de segundo y n-ésimo orden, ver Figura 1.2. Estos tienen la forma de una ventana cuadrada de lado L. En *k-Inpainting* este parámetro puede ser de tamaño: 3, 5, 9 y 11.

Sobre estas ventanas N_0 se opera para extraer la distancia D que indica la diferencia entre dos vecindades en el espacio L*a*b.

Parámetro de propagación (K)

En los Capítulos 2 y 3 se detalla la sub-etapa *K-Propagación* (ver Secciones 2.3.2 y 3.2.4), en donde se verifican K píxeles vecinos en una dirección x y en una dirección y a partir de un punto obtenido mediante muestreo. En el Capítulo 2 se menciona que K debe tomar valores entre 16 y 32, sin embargo se realizan pruebas con un valor inferior y superior al rango para evaluar el rendimiento del algoritmo en comparación con otros parámetros. Los valores que toma K en la propagación son: 8, 16, 32 y 64.

Cantidad de candidatos (N)

El valor de N está relacionado con la cantidad de candidatos que se recopila para cada píxel. Para dar valor a este parámetro, se multiplica la cantidad total de píxeles en la imagen por un pequeño porcentaje, como lo indica la ecuación 2.1. En dicha ecuación, el valor de N depende de K, ya que como se explicó anteriormente N debe ser mínimo dos veces K. Los porcentajes a partir de donde se calcula N son: 0.01 %, 0.05 % y 0.1 %.

Cantidad de iteraciones para el procesamiento de textura (r)

En los diagramas de las Figuras 2.2 y 3.4, se observa que en la etapa de *Procesamiento de textura* se realizan una serie de iteraciones sobre los algoritmos *K-Propagación* y *Búsqueda Aleatoria*. En los diagramas de estas figuras se indica que se deben realizar 5 iteraciones, sin embargo se ha decidido probar con un valor inferior, para indagar como afecta los resultados visuales. Por consiguiente los valores de este parámetro r son: 3 y 5.

Cantidad de iteraciones para aplicar la función de energía (n)

En los diagramas de las Figuras 2.2 y 3.4, se observa que en la etapa de *aplicar función de energía* se realizan una serie de iteraciones sobre las sub-etapas que la conforman. En dichas figuras se indica que se deben iterar 10 veces, sin embargo se ha decidido probar con un valor inferior, para observar su influencia en los resultados visuales en comparación a los demás parámetros que son objeto de estudio. Por consiguiente los valores del parámetro n son: 5 y 10.

Generadores aleatorios

En el Capítulo 2 antes de describir el algoritmo propuesto, se explica que es necesario hacer un muestreo uniforme sobre la imagen a partir de generadores aleatorios, por ello se comentan dos tipos de generadores. A partir de alguno de ellos, se lleva a cabo el muestreo junto con el Algoritmo de Las Vegas, tal como se mencionó en la Sección 2.3.1. Estos generadores aleatorios, también son considerados parámetros del algoritmo. En este trabajo se emplearon los generadores Congruencia Lineal y Ran1.

Una vez comprendidos los parámetros, se procederá a estudiar los diferentes experimentos realizados a continuación.

4.2.2. Consideraciones previas

En la Sección 4.1.1 se especifican los distintos sistemas donde se realizaron las pruebas. En la mayoría de los experimentos presentados se utiliza el sistema 1. Esto se debe a su capacidad de procesamiento (superior a los otros sistemas), lo cual permite ejecutar el algoritmo en la CPU en menor tiempo. Por las características del procesador central del sistema 3, es fácil percibir que al mismo le toma más tiempo procesar el algoritmo. Sin embargo, los sistemas 1 y 2 tienen alto rendimiento computacional, por lo tanto se efectuaron un lote de pruebas en ambos sistemas para recopilar el tiempo de ejecución del mismo en su versión secuencial.

Se procesó una imagen de la Figura 4.1 de dimensiones 228×250 píxeles, fijándose el el valor $0.01\,\%$ para calcular N, mientras los demás parámetros toman todos sus posibles valores. Así, el algoritmo *k-Inpainting* es ejecutado 128 veces en cada sistema. La duración del algoritmo en el tiempo para cada hardware se refleja en la Figura 4.2. En esta gráfica, se aprecia que el sistema 1 es el más rápido de los dos. En consecuencia, se toma el mismo para llevar a cabo las pruebas de rigor.

Figura 4.1: Imagen para aplicar el algoritmo en su versión secuencial sobre el área roja.

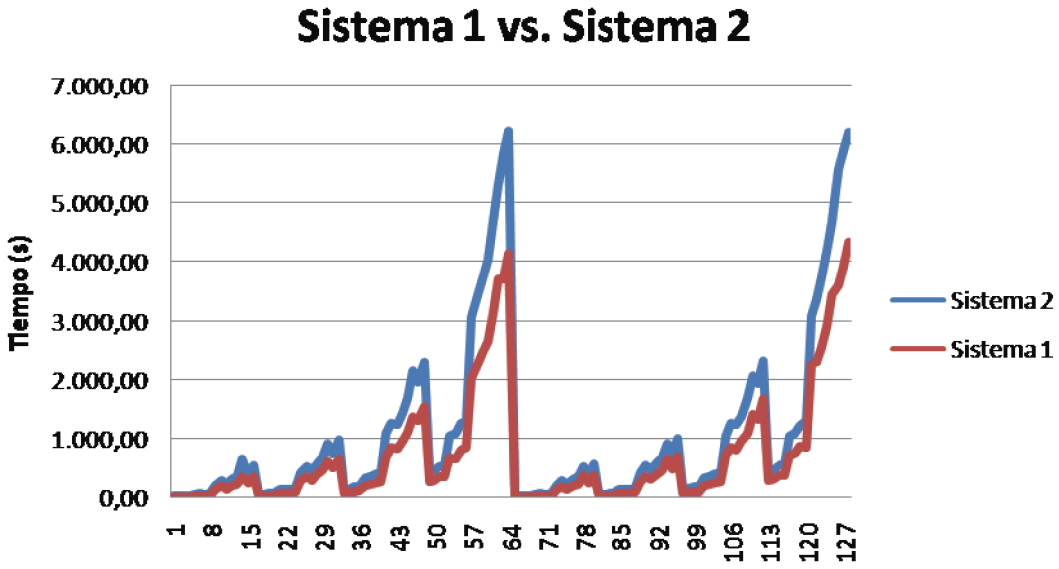


Figura 4.2: Rendimiento en cuanto al factor tiempo de los sistemas de prueba 1 y 2. En el eje de las x, se encuentra el número de corrida y en el eje y se refleja el tiempo en segundos que toma cada ejecución para cada sistema de la gráfica.

La etapa de procesamiento de textura (ver Capítulo 3) se puede llevar a cabo mediante dos implementaciones. La primera (*GPU T1*) ejecutando una sub-etapa a la vez, pero teniendo como entrada las muestras que se emplearon en la etapa de inicialización. La segunda (*GPU T2*), iterando r veces en las sub-etapas que la componen, teniendo como entrada a dicha etapa los mejores candidatos seleccionados en la etapa anterior.

En los experimentos presentados, se contemplarán ambas implementaciones de la versión paralela de *k-Inpainting*. Por otra parte, en la Sección 4.1.1 se presentan tres tipos de tarjetas gráficas, cada una con especificaciones técnicas y poder de cálculo distinto. En la Tabla 4.2

es notable que la tarjeta GeForce GTX 470, tiene mayor poder de cómputo que las demás y por ello, ésta ha sido seleccionada para realizar los diferentes experimentos correspondientes a la versión paralela.

4.2.3. Experimentos (nivel 1)

Los experimentos de nivel 1 prueban el algoritmo en sus dos versiones, considerando todas las combinaciones posibles de los diferentes parámetros descritos en la Sección 4.2.1. Este es el estudio más engorroso, debido a que se deben realizar 384 ejecuciones (resultantes de las 384 combinaciones que surgen a partir de la composición de dichos parámetros) para la versión secuencial y la la implementación *GPU T2* de la versión paralela. Adicionalmente se realizan 192 ejecuciones para la implementación *GPU T1* del algoritmo en la GPU.

El tiempo de ejecución del algoritmo depende de tres factores: la complejidad del algoritmo, el tamaño de la imagen y el tamaño del agujero. Debido a la cantidad de pruebas a realizar y a la complejidad del algoritmo, se decide emplear una imagen relativamente pequeña de dimensiones 228×250 píxeles con un orificio pequeño. Esta imagen corresponde a la imagen de la Figura 4.1 y la Figura 4.3, siendo ésta última donde el usuario ya ha seleccionado el área a tratar.

Figura 4.3: Imagen electa para efectuar los experimentos de nivel 1, con la zona a tratar previamente seleccionada, pintada en color negro.

El objetivo de este experimento es elegir el sub-conjunto de las mejores combinaciones de los diferentes valores, para luego aplicar estas combinaciones a otros experimentos. En primer lugar se estudiarán los resultados visuales del algoritmo para cada versión. Después se realiza una nueva selección sobre estos datos, para tomar las combinaciones con mejores resultados.

Para hacer una distinción objetiva de los parámetros, se hace un estudio visual y métrico. Para el estudio métrico se necesita una imagen de referencia que sea correcta, por ello se tomó una imagen sin ningún tipo de pérdida de información, como la mostrada en la Figura 4.4a y se edita para causar pérdida de información como se observa en la Figura 4.4b. Luego, se hace una selección del área dañada como se observa en la Figura 4.4c y esa es la región

a reconstruir. Con la Figura 4.4a, es posible tener un punto de referencia adecuado y así realizar una diferencia píxel a píxel con cada imagen resultante. Además, con dicha imagen de referencia también existe una asociación visual para decidir si una imagen resultante es mejor a otra.

(a) Imagen original (b) Imagen editada para causar pérdida de información (c) Zona dañada seleccionada

Figura 4.4: Procedimiento realizado para los experimentos de nivel 1.

Para evaluar los resultados de la imagen se aplican dos métodos:

1. Calcular la diferencia promedio entre la imagen 4.4a y la imagen reconstruida a partir de 4.4c en la región tratada con la técnica. Dada la imagen original O y la imagen resultante R, se calcula en el espacio L*a*b la diferencia euclidiana promedio de los η píxeles sobre Ω. La diferencia se calcula mediante la ecuación 4.1.

$$Dif(\varphi) = \frac{\sum_{i \in \Omega} \|\iota(O_i) - \iota(Ri)\|^2}{\eta} \tag{4.1}$$

2. Mediante la observación humana. Se realizó una encuesta a cinco usuarios, para que los mismos dieran una calificación a cada uno de los resultados obtenidos con respecto a la imagen de la Figura 4.4. El sistema de calificación se muestra en la Tabla 4.4.

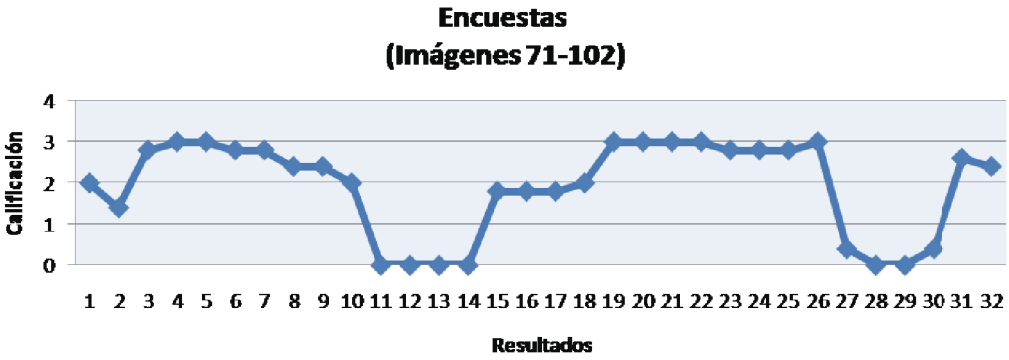
Característica	Valoración
Deficiente	0
Regular	1
Aceptable	2
Excelente	3

Tabla 4.4: Calificación de imágenes, según sus resultados.

La Figura 4.5 contiene dos gráficas con respecto a las imágenes obtenidas desde la ejecución número 71 a la ejecución número 102 de la versión secuencial. La gráfica 4.5a indica en

promedio, la calificación otorgada por los diferentes usuarios a las imágenes resultantes y en la gráfica 4.5b se observa la diferencia euclidiana con respecto a la imagen original.

Un aspecto importante, es que cuando hay picos en la gráfica de diferencias, el resultado en las encuestas es bajo o cercano a cero, mientras que cuando la gráfica se mantiene en valores bajos, la aceptación de la imagen por parte de los usuarios es alta. En consecuencia, existe una correspondencia entre los resultados obtenidos. La imagen 4.5 muestra solo una parte de los datos recopilados, el resto de los datos pueden observarse en la Sección 5.4.

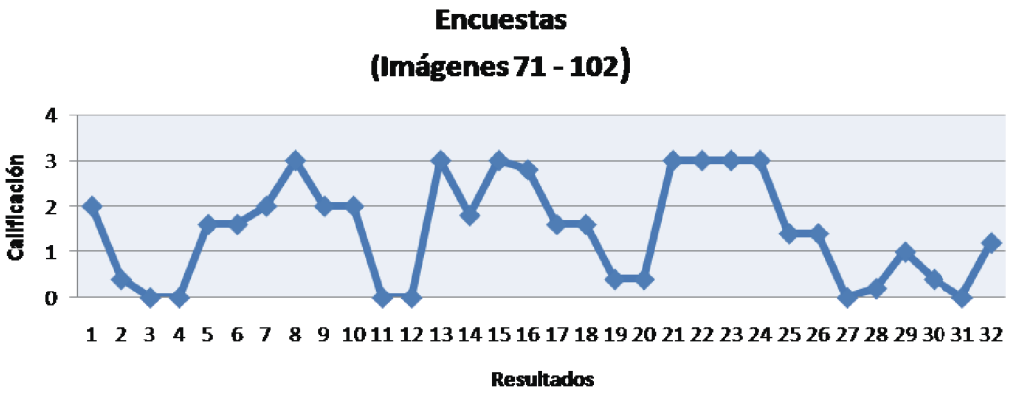


(a)

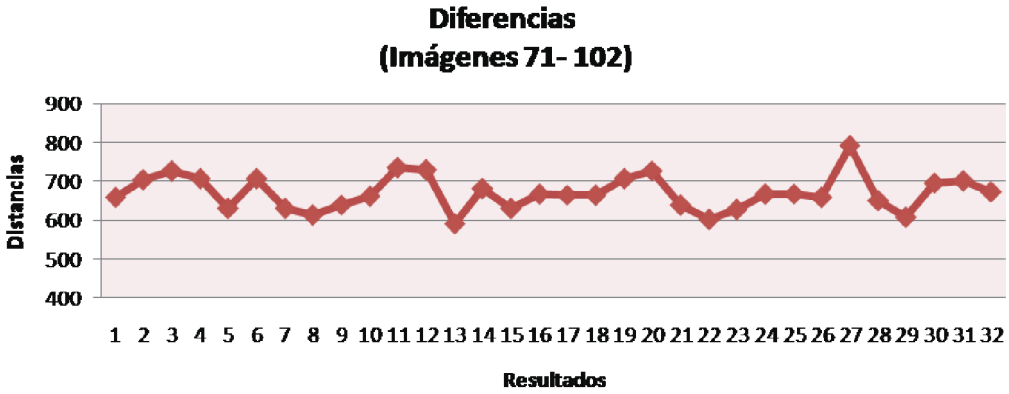
(b)

Figura 4.5: Resultados de los métodos de evaluación empleados para escoger la mejor combinación de parámetros, desde la ejecución 71 a la 102 (versión secuencial).

Las Figuras 4.6 y 4.7 corresponden a los resultados obtenidos para un lote de ejecuciones del algoritmo de la primera y segunda implementación (*GPU T1* y *GPU T2* respectivamente) de la versión paralela. En ambos casos, en la gráfica de diferencias se observa que los valores son más nivelados con respecto a la gráfica de la versión secuencial, eso es debido a que los valores de la mencionada versión oscilan entre 594 a 1868, mientras que las tendencias de la versión paralela oscilan entre 628 a 839 y 562 a 858 respectivamente. A pesar que no hay picos evidentes, se puede observar un comportamiento similar, ya que en las diferencias más elevadas, la encuesta corresponde a una calificación regular o deficiente.



(a)



(b)

Figura 4.6: Resultados de los métodos de evaluación empleado para escoger la mejor combinación de parámetros, desde la ejecución 71 a la 102 (primera implementación de la versión paralela).

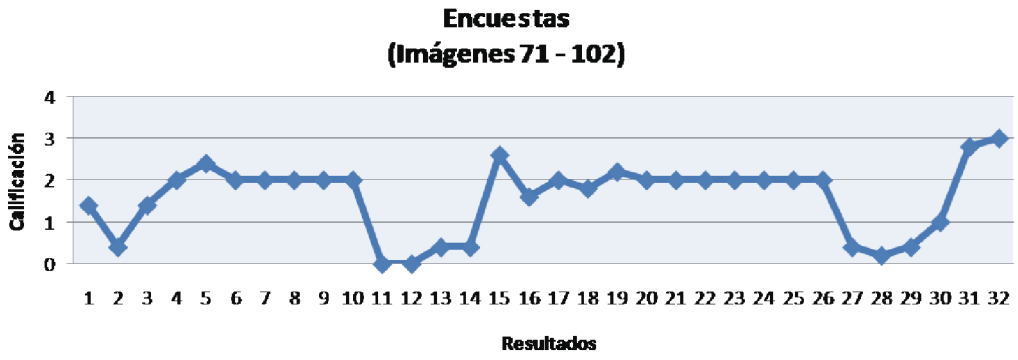
Por otro lado, la correspondencia no es total, ya que la imagen que tuvo menor diferencia con respecto a la imagen original en cada versión, no tiene un nivel de aceptación promedio de 3 (excelente). En la Figura 4.8 se muestra uno de los mejores resultados obtenido de la versión secuencial y de las tendencias de la versión paralela.

En la Tabla 4.5, se muestra el tiempo que tardó cada versión en alcanzar el resultado observado en la Figura 4.8.

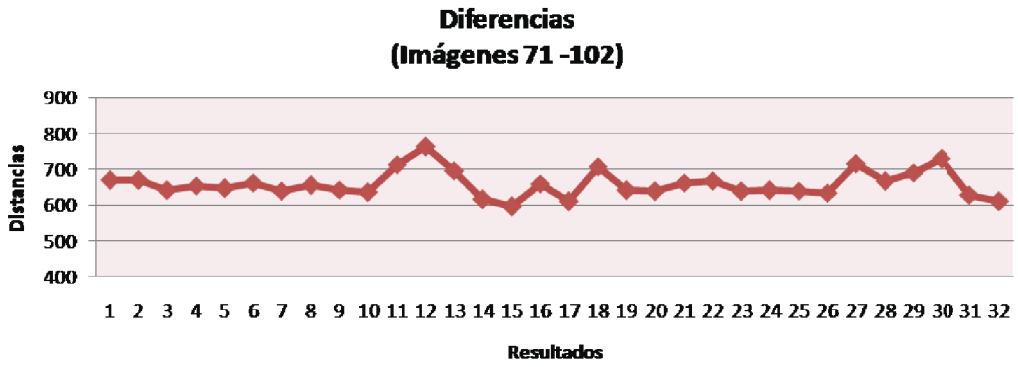
Versión del algoritmo	Tiempo	Parámetros
Secuencial	3 minutos y 19 segundos	$L = 9$, $K = 8$, $r = 3$, $n = 10$
Paralela (GPU T1)	1 minuto y 18 segundos	$L = 9$, $K = 32$, $n = 5$
Paralela (GPU T2)	33 segundos	$L = 5$, $K = 64$, $r = 3$, $n = 10$

Tabla 4.5: Tiempos de ejecución

En la Figura 4.9 se puede visualizar el tiempo que tardó la versión secuencial en ejecutar



(a)



(b)

Figura 4.7: Resultados de los métodos de evaluación empleado para escoger la mejor combinación de parámetros, desde la ejecución 71 a la 102 (segunda implementación de la versión paralela).

(a) CPU (b) *GPU T1* (c) *GPU T2*

Figura 4.8: Resultados visuales de los diferentes algoritmos *k-Inpainting*.

las 384 combinaciones, el tiempo en ejecutarse las 192 combinaciones de la implementación *GPU T1* de la versión paralela y el tiempo en ejecutarse las 384 combinaciones en la implementación *GPU T2* de la versión paralela.

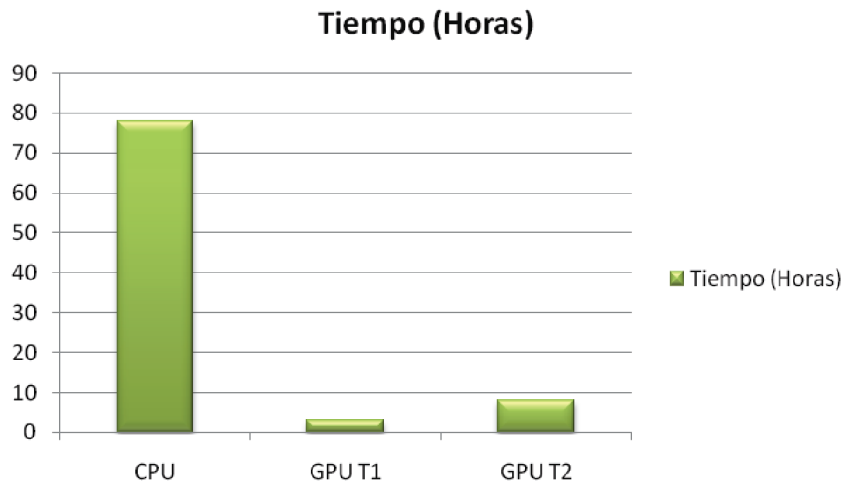


Figura 4.9: Tiempos de ejecución de las versiones de *k-Inpainting* en reconstruir el área desconocida de la Figura 4.3.

A partir del experimento realizado, se extraen las mejores combinaciones para cada caso, las cuales se muestran en las Tablas 4.6, 4.7 y 4.8. El campo frecuencia, indica cuantas veces esas combinaciones obtuvieron resultados de calificación promedio 3. La frecuencia de las combinaciones ocurren debido a que las pruebas se realizan con dos tipos de muestreos y diferentes porcentajes para calcular N, tal como se mencionó anteriormente.

Para escoger el subconjunto de combinaciones, se tomarán las que arrojan mejores resultados (la mayoría presente en las Tablas 4.6, 4.7 y 4.8) que tengan mínimo 2 repeticiones. En este punto se han seleccionado las combinaciones de los parámetros L, K, r y n. Para fijar el resto de los parámetros se toman las siguientes consideraciones:

- En valor de N depende del parámetro K. Si el valor de N es menor que $2K$, entonces N tomará dicho valor. El porcentaje 0.01 % suele dar un N siempre más pequeño que K. Por ello se contemplará para las próximas pruebas los porcentajes 0.05 % y 0.1 %, los cuales se trabajarán de la siguiente manera: Si para el porcentaje 0.05 % no se obtienen los resultados esperados, se realiza nuevamente la prueba con porcentaje 0.1 %.

- El muestreo no muestra una influencia notable. Por lo tanto, se probará en primera instancia con el tipo de muestreo de Congruencia Lineal y en caso de no obtener resultados esperados con los parámetros seleccionados y una vez probados los dos porcentajes para calcular N, se realiza la prueba con el muestreo de Ran1.

Debido a que la región reconstruida en esta imagen es pequeña (1.9 % de la imagen), en imágenes complejas o con mayor área a reconstruir se pueden tomar las combinaciones con los valores más grandes de la tabla, aunque éstos no sean tan frecuentes.

L	K	r	n	Frecuencia
9	8	5	5	2
9	16	3	5	3
9	16	3	10	3
9	16	5	5	5
9	16	5	10	3
9	32	3	5	3
9	32	3	10	4
9	64	3	10	2
9	8	3	10	2
9	8	5	5	4
9	32	5	5	2
11	8	3	5	2
9	32	5	10	1
9	64	3	5	1
11	16	5	10	1
9	8	3	10	1
11	8	5	10	1
11	16	5	5	1
11	8	3	10	1

Tabla 4.6: Combinación de datos (versión secuencial)

L	K	n	Frecuencia
5	8	5	4
5	16	10	2
9	32	5	4
9	32	10	4
9	64	5	5
9	64	10	4
5	64	5	2
5	64	10	3
5	32	5	2
5	32	10	3
5	16	5	2
11	8	10	1
9	16	5	1
3	8	10	1
9	16	10	1
9	8	5	1
9	8	10	1
5	64	10	1

Tabla 4.7: Combinación de datos (implementación *GPU T1* de la versión paralela)

L	K	r	n	Frecuencia
5	32	5	10	4
9	32	3	10	2
9	32	5	10	5
5	64	3	5	4
5	64	5	10	4
9	64	5	10	4
5	32	3	10	2
5	32	3	5	5
9	32	3	5	5
9	64	5	5	5
5	16	5	10	2
9	16	3	5	2
9	16	5	5	3
9	32	3	5	3
9	64	3	5	4
9	8	5	10	3
5	16	5	5	1
9	8	5	5	1
5	64	3	10	1

Tabla 4.8: Combinación de datos (implementación *GPU T2* de la versión paralela)

4.2.4. Experimentos (nivel 2)

En los experimentos nivel 2, se tratarán las imágenes agrupadas en la Figura 4.10. Dichas imágenes serán procesadas con los parámetros escogidos en los experimentos de nivel 1, los cuales se encuentran en las Tablas 4.6, 4.7 y 4.8. Como las imágenes son distintas, pueden existir casos especiales, en los cuales haya que variar el valor del tamaño de la ventana L o el valor de propagación K. Por otro lado, la Tabla 4.9 resume las características de las imágenes a procesar.

Imagen	Dimensiones en píxeles
A	301×226
B	292×291
C	206×308
D	256×162

Tabla 4.9: Ejecuciones de la imagen A con el algoritmo secuencial

Las imágenes serán tratadas por el orden indicado en la Figura 4.10, es decir, siguiendo la secuencia alfabética A,B,C,D. Por ende la primera imagen a procesar es la imagen A, la segunda imagen con la cual experimentar es la B y así sucesivamente. Para cada caso, se mostrará el resultado en la versión secuencial y las dos tendencias de la versión paralela con

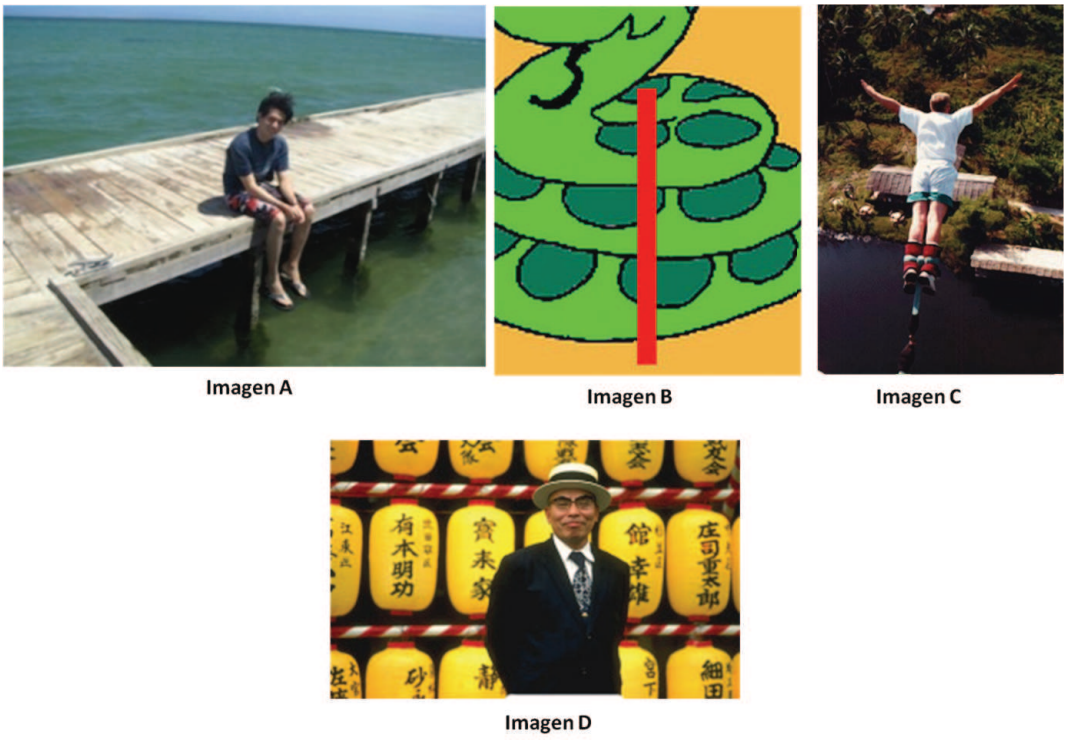


Figura 4.10: Imágenes de prueba para los experimentos nivel 2

la combinación de parámetros que más se adecua en cada caso y el tiempo de ejecución de las mismas. También se indicará cuantos píxeles se reconstruyeron y que porcentaje de la imagen fue reconstruida.

Imagen A

En la imagen A, se observa una persona sentada en un puente. Para esta imagen se busca remover a dicha persona para visualizar el paisaje de la fotografía. Luego que el usuario selecciona el área a restaurar rodeando al individuo en la foto, en la imagen queda un agujero tal como se observa en la Figura 4.11. Este agujero está constituido por 6.398 píxeles, que forman el 9,4 % de la imagen.

En la Tabla 4.10 se listan el subconjunto de parámetros seleccionados de las Tabla 4.6 para procesar la imagen mediante el algoritmo secuencial. También se muestra el tiempo de ejecución para cada caso. En la Figura 4.12 se observan los resultados obtenidos en relación a la información mostrada en la Tabla 4.10. La imagen 4 muestra el mejor resultado visual en un menor tiempo.

En la Tabla 4.11 se despliega el subconjunto de paŕametros tomados de la Tabla 4.7 para procesar la región desconocida de la imagen con su respectivo tiempo de ejecución. La Figura 4.13 muestra los resultados obtenidos para cada combinación. En esta figura podemos observar que los resultados son muy parecidos entre las imágenes 2, 3, 4, 5 y 6, por consiguiente el mejor resultado es aquel que tomó menos tiempo de ejecución, es decir, para este caso, el

Figura 4.11: Agujero en la imagen A

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,05 %	9	16	3	5	31 minutos
2	0,05 %	9	16	5	10	45 minutos
3	0,1 %	9	16	5	10	1 hora y 9 minutos
4	0,05 %	9	32	3	5	1 hora y 15 minutos
5	0,05 %	9	32	5	5	1 hora y 32 minutos
6	0,05 %	9	32	5	10	1 hora y 45 minutos

Tabla 4.10: Ejecuciones de la imagen A con el algoritmo secuencial.

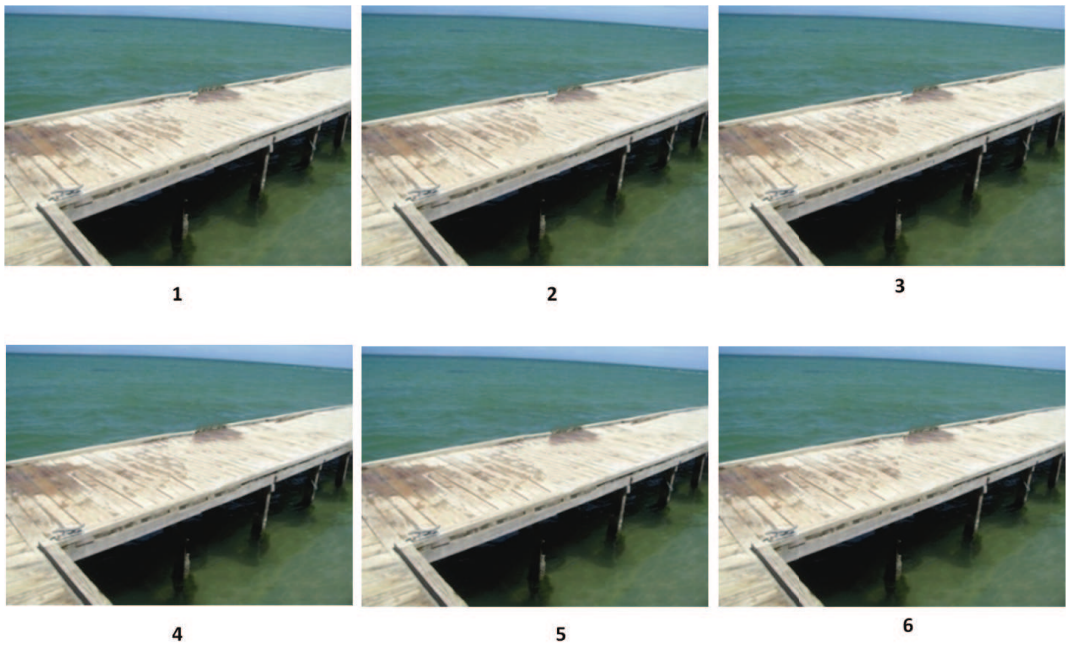


Figura 4.12: Resultados luego de procesar la imagen A con los parámetros de la Tabla 4.10.

mejor resultado corresponde a la imagen 6.

Resultado	Porcentaje	L	K	n	Tiempo
1	0,1 %	9	16	10	6 minutos
2	0,05 %	9	32	5	10 minutos
3	0,05 %	5	64	5	8 minutos y 40 segundos
4	0,05 %	9	32	10	10 minutos y 10 segundos
5	0,05 %	9	64	5	27 minutos
6	0,1 %	5	16	10	1 minuto y 40 segundos

Tabla 4.11: Ejecuciones de la imagen A con la implementación *GPU T1* de la versión paralela.

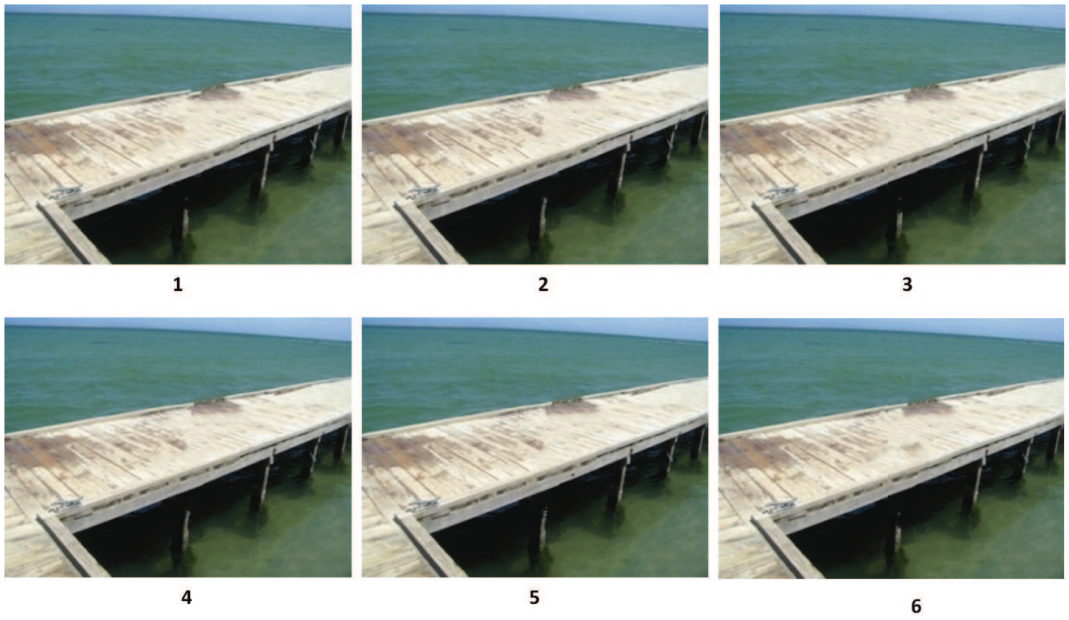


Figura 4.13: Resultados luego de procesar la imagen A con los parámetros de la Tabla 4.11.

La Tabla 4.12 lista los parámetros empleados para las diferentes ejecuciones de la implementación *GPU T2* de la versión paralela. En la Figura 4.14 se observan las imágenes obtenidas luego de procesar la Figura 4.10 con este algoritmo. Los mejores resultados se observan en la imágenes 2 y 6. Como la imagen 2 toma menos tiempo de ejecución, se selecciona ésta como el mejor resultado para este conjunto de imágenes procesadas con esta variación del algoritmo paralelo.

Imagen B

La imagen B tiene un área dañada (rectángulo rojo) en la parte central de la imagen, afectando de forma transversal la estructura de la serpiente en dicha imagen. En este caso se busca restaurar la región. Luego que el usuario selecciona el área a tratar rodeando el área

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,1 %	9	16	5	10	10 minutos
2	0,1 %	5	16	5	10	3 minutos
3	0,1 %	5	32	3	5	5 minutos
4	0,05 %	9	32	3	5	14 minutos
5	0,05 %	9	16	3	5	7 minutos
6	0,05 %	5	64	3	5	13 minutos

Tabla 4.12: Ejecuciones de la imagen A con la implementación *GPU T2* del algoritmo paralelo.

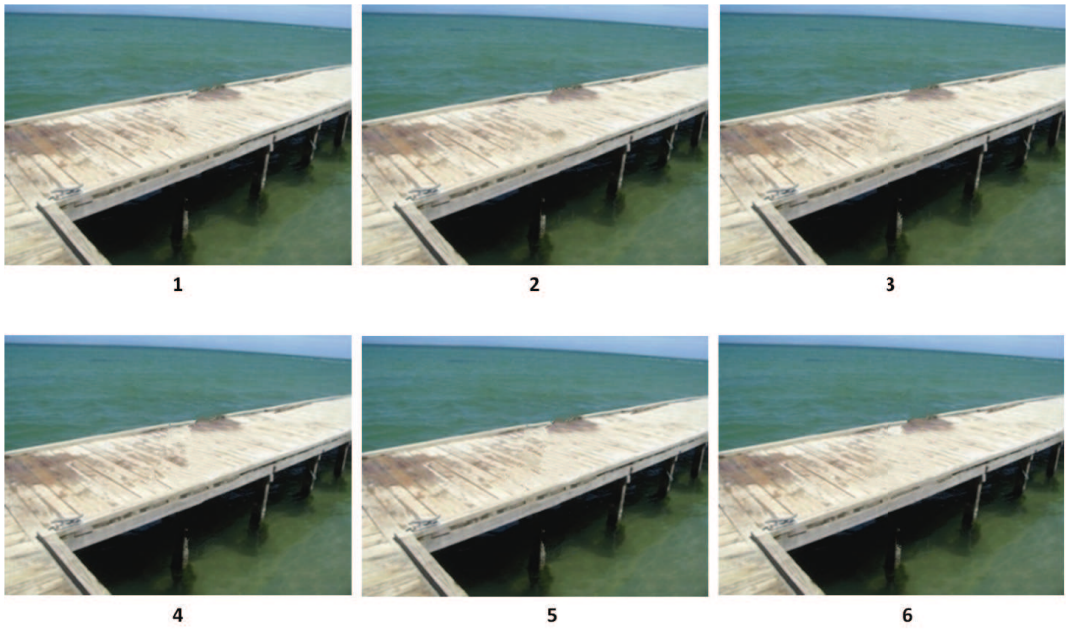


Figura 4.14: Resultados luego de procesar la imagen A con los parámetros de la Tabla 4.12.

dañada, en la misma queda un orificio tal como se observa en la Figura 4.15. Este orificio está constituido por 5.617 píxeles, que forman el 7,9 % de la imagen B.

Los parámetros empleados para procesar la imagen a nivel de CPU, se muestran en la Tabla 4.13, en la cual también se encuentran los tiempos de ejecución para cada combinación seleccionada de la Tabla 4.6. En la Figura 4.16 se visualizan los resultados para los diferentes parámetros de la Tabla 4.13. En este caso, toma prioridad la calidad visual, por lo que el mejor resultado es evidentemente la imagen 2, en la cual no se generaron artefactos.

Ahora se procederá a mostrar los resultados de la versión paralela en su primera implementación (*GPU T1*). El subconjunto de parámetros extraídos de la Tabla 4.7, se despliegan en la Tabla 4.14. En la Figura 4.17 se observa el resultado para cada conjunto de parámetros en la tabla. Los resultados son muy parecidos entre las primeras cuatro imágenes, por lo que se selecciona como mejor resultado la que se ejecuta en menor tiempo entre ellas, que corresponde a la imagen 2.

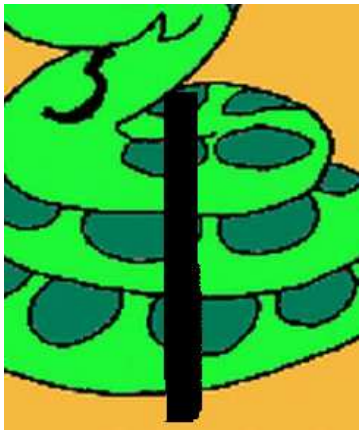


Figura 4.15: Orificio en la imagen B

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,1 %	11	8	3	5	44 minutos
2	0,05 %	11	8	3	5	29 minutos
3	0,05 %	11	8	3	10	46 minutos
4	0,1 %	9	16	3	10	50 minutos
5	0,05 %	11	8	5	10	46 minutos
6	0,05 %	11	16	5	10	1 hora y 13 minutos

Tabla 4.13: Ejecuciones para procesar la imagen B con el algoritmo secuencial.

Resultado	Porcentaje	L	K	n	Tiempo
1	0,1 %	11	8	10	5 minutos
2	0,05 %	11	8	10	3 minutos 45 segundos
3	0,05 %	9	32	5	9 minutos
4	0,05 %	9	32	10	9 minutos y 20 segundos
5	0,05 %	5	64	10	8 minutos y 17 segundos
6	0,1 %	9	16	10	5 minutos y 25 segundos

Tabla 4.14: Ejecuciones de la imagen B con el algoritmo para la implementación *GPU T1* de la versión paralela.

En la Tabla 4.15 se listan los parámetros seleccionados para procesar la imagen a tratar con el algoritmo paralelo en su segunda implementación (*GPU T2*). Para cada combinación se visualiza el resultado obtenido en la imagen 4.18. Se observa que los mejores resultados visuales se encuentran entre las imágenes 1,2 y 5. Como la que se ejecuta en menor tiempo entre éstas es la imagen 2, seleccionamos dicha imagen como mejor resultado.

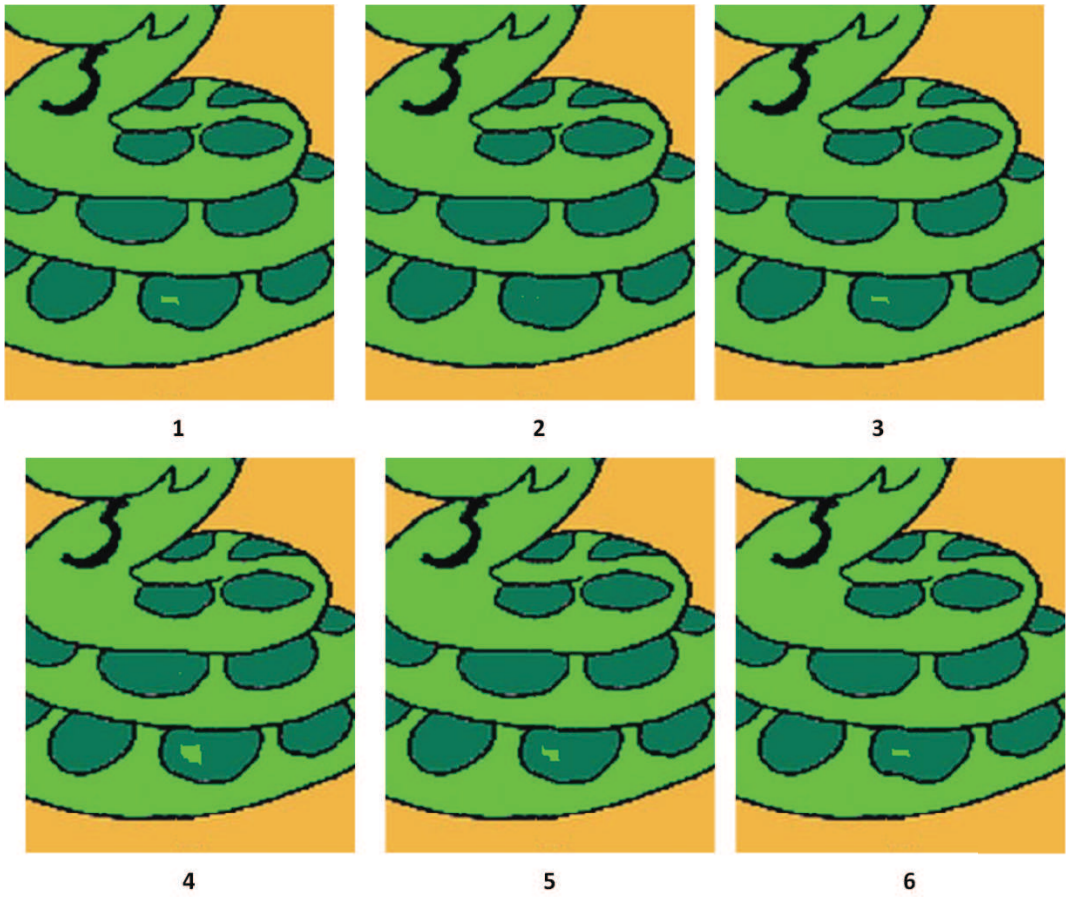


Figura 4.16: Resultados luego de aplicar el algoritmo secuencial con los parámetros de la Tabla 4.13.

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,1 %	9	8	5	10	5 minutos
2	0,1 %	9	8	5	5	4 minutos y 30 segundos
3	0,1 %	5	16	5	5	3 minutos y 30 segundos
4	0,1 %	5	16	5	10	2 minutos y 40 segundos
5	0,05 %	9	16	3	10	7 minutos
6	0,05 %	5	32	3	10	4 minutos

Tabla 4.15: Ejecuciones de la imagen B con el algoritmo paralelo en su implementación *GPU T2*.

Imagen C

En la imagen C se busca remover la persona que bloquea el paisaje, la cual forma el agujero verde de la Figura 4.19. Esta área tiene 12.117 píxeles y constituye el 19,09 % de la imagen. En las Tablas 4.16, 4.17 y 4.18, se listan los parámetros utilizados para procesar la imagen en la versión secuencial y las tendencias 1 y 2 de la versión paralela. Los distintos resultados para cada caso se muestran en las Figuras 4.20, 4.21 y 4.22.

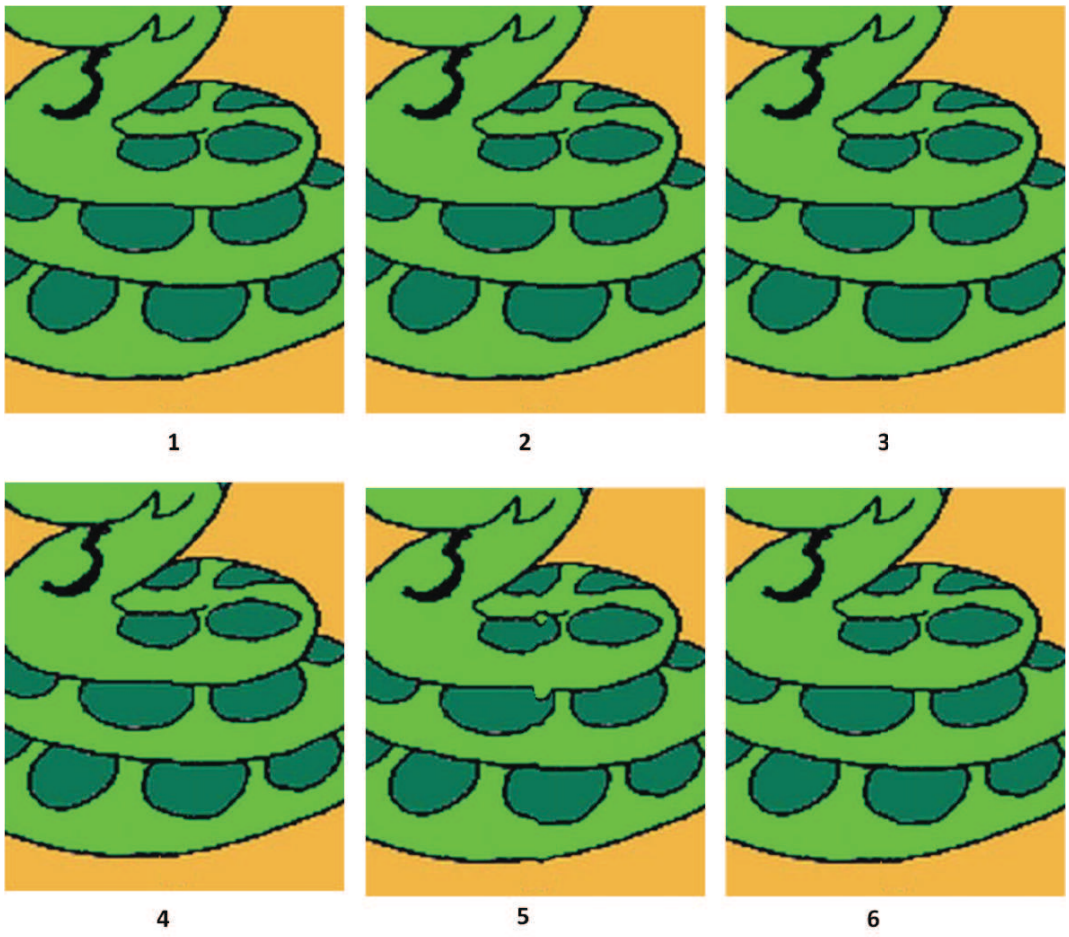


Figura 4.17: Resultados después de aplicar la primera versión del algoritmo paralelo con los parámetros de la Tabla 4.14.

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,1 %	11	16	5	10	2 horas y 30 minutos
2	0,05 %	9	16	5	10	1 hora y 50 minutos
3	0,1 %	9	16	10	5	2 horas

Tabla 4.16: Ejecuciones de la imagen C con el algoritmo secuencial.

Resultado	Porcentaje	L	K	n	Tiempo
1	0,1 %	5	16	10	2 minutos y 41 segundos
2	0,1 %	11	8	10	7 minutos
3	0,1 %	9	32	10	16 minutos

Tabla 4.17: Ejecuciones de la imagen C con la primera variación (*GPU T1*) del algoritmo paralelo.

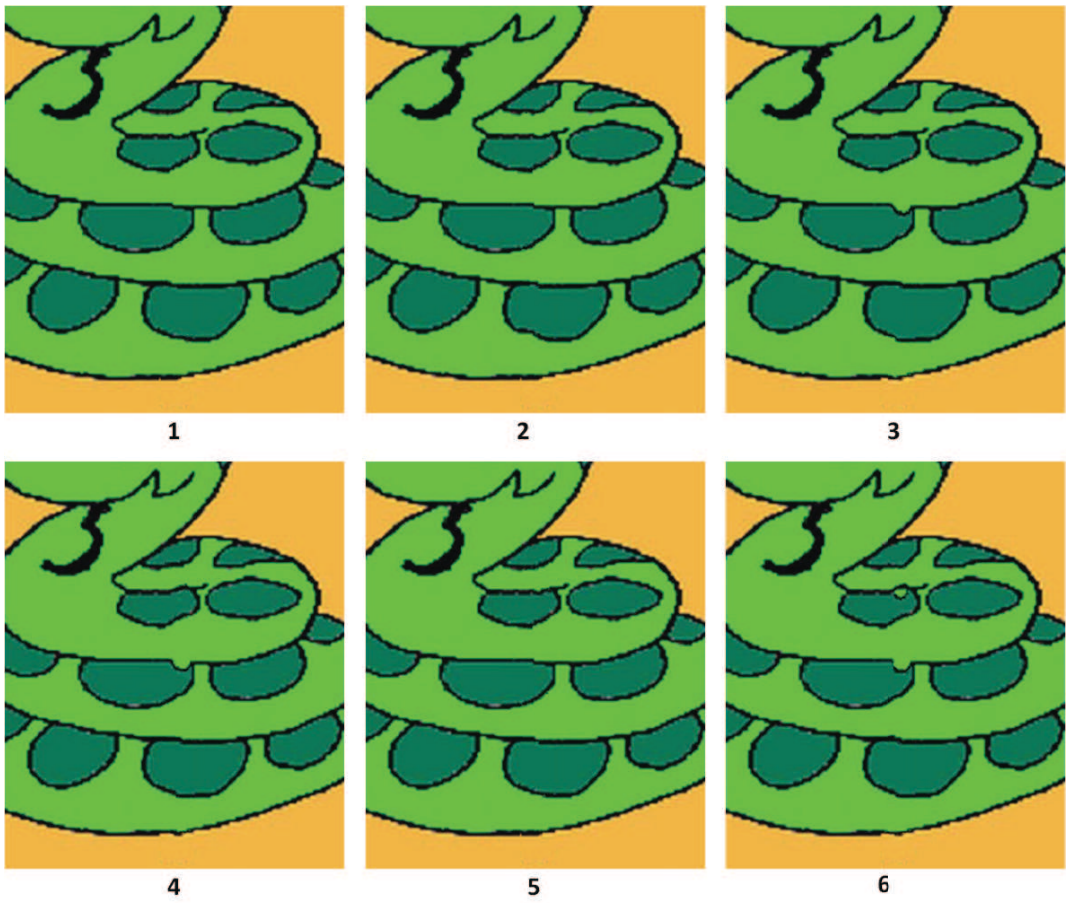


Figura 4.18: Resultados luego de procesar la imagen B con la implementación *GPU T2* del algoritmo paralelo con los parámetros de la Tabla 4.15.

Figura 4.19: Zona a reconstruir en la Imagen C

Resultado	Porcentaje	L	K	r	n	Tiempo
1	0,1 %	9	16	3	5	12 minutos
2	0,1 %	9	32	5	10	29 minutos
3	0,1 %	9	16	5	5	15 minutos

Tabla 4.18: Ejecuciones de la imagen C con la segunda variación (*GPU T2*) del algoritmo paralelo.

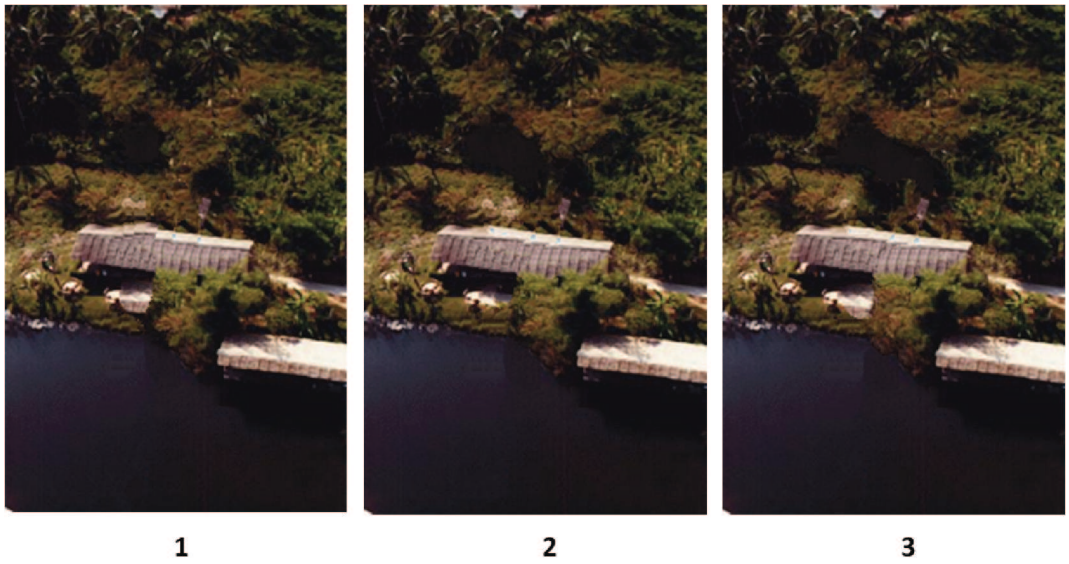


Figura 4.20: Resultados posteriores a la aplicación del algoritmo secuencial con los parámetros de la Tabla 4.16.

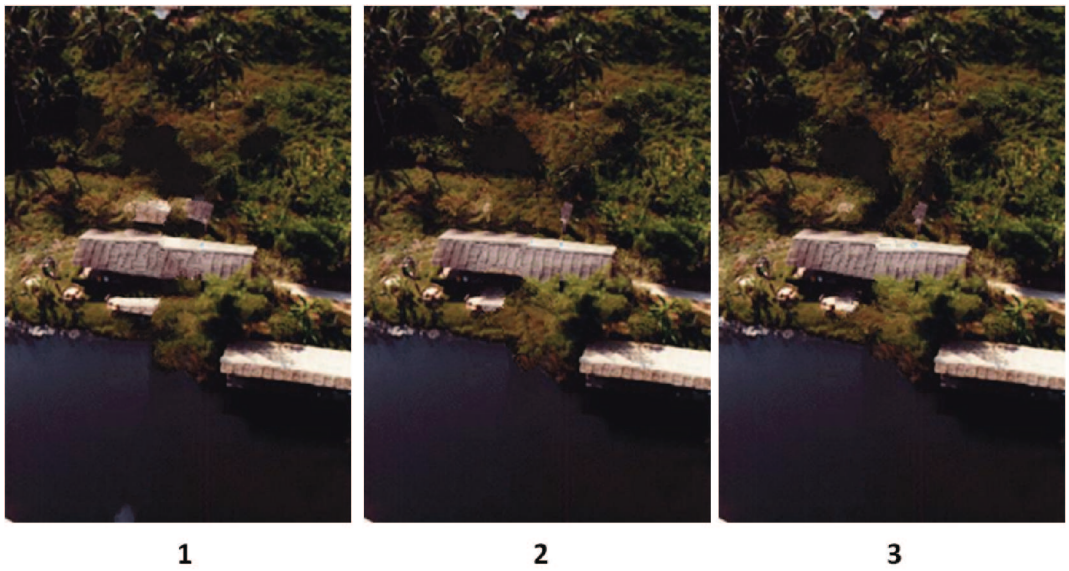


Figura 4.21: Resultados luego de procesar la imagen C con los parámetros indicados en la Tabla 4.17.

Figura 4.22: Resultados luego de procesar la imagen C con los parámetros de la Tabla 4.18.

Es notable que para cada caso los mejores resultados son: La segunda combinación de parámetros para CPU, la segunda combinación para la implementación *GPU T1* de *k-Inpainting* programado bajo GPU y la segunda combinación para la implementación *GPU T2* de la versión paralela. En este caso, la mejor selección se basa en la observación de los resultados.

Imagen D

Se busca remover el individuo presente en la foto y reconstruir el patrón detrás de él. Esta imagen es compleja, ya que el tamaño del agujero representa un porcentaje importante y tiene un fondo complicado. El agujero que queda en la imagen luego de seleccionar el área a tratar es el que indica la Figura 4.23, el cual ocupa el 26,67 % de la imagen con 11.061 píxeles. A continuación se despliega en la Tabla 4.19 los parámetros utilizados para procesar esta imagen con cada variante del algoritmo.

Figura 4.23: Área a reconstruir en la imagen D.

Versión	Porcentaje	L	K	r	n	Tiempo
CPU	0,05 %	11	16	5	10	2 horas
GPU T1	0,05 %	11	32	-	10	44 minutos
GPU T2	0,05 %	11	32	5	5	50 miuntos

Tabla 4.19: Ejecuciones de la imagen D con los diferentes algoritmos propuestos.

A diferencia de las imágenes anteriores, solo la primera combinación es extraída de la Tabla 4.6, debido a que la imagen arroja mejores resultados con una ventana $L = 11$ para cualquier versión. Los parámetros de la segunda y tercera combinación de la tabla se escogieron a partir del resultado obtenido para la primera. En las Figuras 4.24, 4.25 y 4.26 se observan las imágenes obtenidas a partir de estos parámetros.

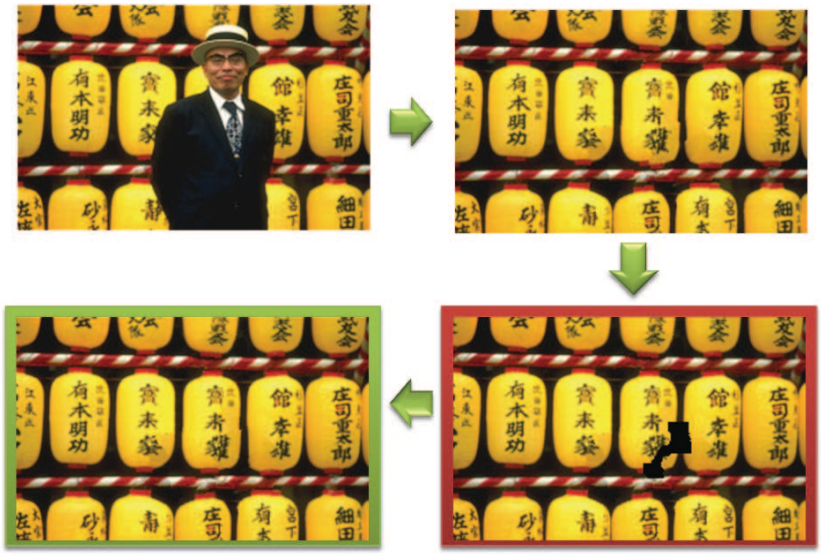


Figura 4.24: Resultado con CPU: Imagen D.

Dado que no se obtuvo una reconstrucción exacta, se aplica el algoritmo nuevamente en el área que tuvo fallas, para cada caso se seleccionan las áreas que se resaltan en rojo. La imagen resaltada con color verde muestra los resultados finales obtenidos. Cabe destacar, que los tiempos que se indican en la Tabla 4.19 es la suma de los tiempos de ejecución del algoritmo que permite alcanzar el producto deseado. En esta imagen se puede observar que las variantes en paralelo tiene fallas, especialmente en el área que está cercana y sobre los límites de la imagen. Por otro lado, debido a los parámetros utilizados en cada caso y el tamaño de Ω el tiempo de cómputo es alto, incluso en las tendencias de la versión paralela.

4.2.5. Experimentos (nivel 3)

A partir del experimento anterior se puede notar que los algoritmos en paralelo arrojan buenos resultados en la mayoría de los casos con un tiempo de ejecución medianamente acep-

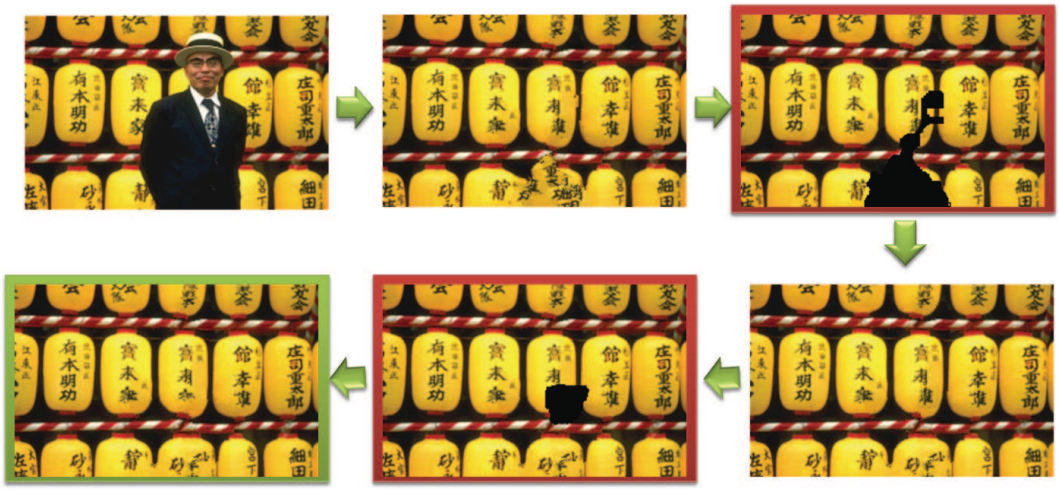


Figura 4.25: Resultado con GPU T1: Imagen D.

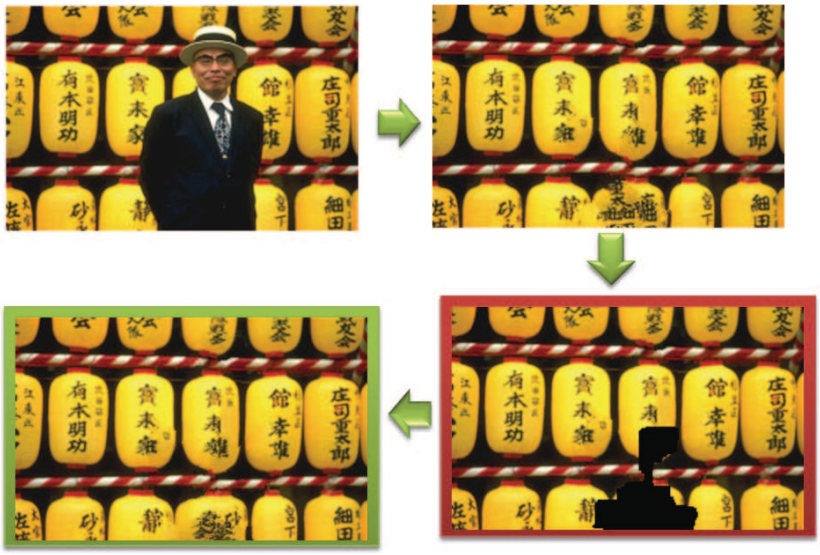


Figura 4.26: Resultado con GPU T2: Imagen D.

table en comparación a la versión secuencial. Por ello, para estudiar el comportamiento del algoritmo en otras imágenes, se realizarán diferentes pruebas a nivel de GPU. Para este tipo de experimento, se compararán los resultados de las dos tendencias de la versión paralela en cuanto a calidad visual y tiempo, tratando de estimar que tendencia es mejor desde una perspectiva general. Los parámetros empleados no serán exactamente los mismos, ya que como se ha mencionado en el capítulo 3 la implementación *GPU T1* no cuenta con el parámetro r, así que los parámetros no serán siempre similares para todos los casos.

La Figura 4.27 muestra el conjunto de imágenes contempladas para este experimento, ordenadas por nivel de dificultad, según un concepto de estructura, tamaño de la imagen y tamaño del agujero. El agujero se puede visualizar en cada caso en la Figura 4.28. Seguida-

mente se muestra en la Tabla 4.21 el tipo de tendencia utilizada (1 para la *GPU T1* y 2 para *GPU T2*), el identificador de la imagen asociado a la Figura 4.27, los parámetros empleados y el tiempo de respuesta. Luego para cada imagen se discuten sus resultados comparando las dos implementaciones paralelas propuestas en este documento.

En la Tabla 4.20 se observa para cada imagen, cuantos píxeles se reconstruyeron y el porcentaje que dicho orificio afecta en la imagen.

Imagen	Dimensiones en píxeles	# píxeles reconstruidos	% de la imagen
1	373×434	7376	4,6 %
2	470×342	8249	5,1 %
3	307×230	3257	4,6 %
4	230×173	3210	8 %
5	448×458	9860	4,8 %
6	447×333	6638	4,5 %
7	600×398	15437	6,5 %
8	500×334	17354	7 %
9	640×425	8590	3 %

Tabla 4.20: Dimensiones de Ω para cada imagen de la Figura 4.28.

Implementación	Imagen	Parámetros	Tiempo
1	1	$L = 9$, $K = 8$, $n = 10$, 0.1 %	4 minutos y 44 segundos
2	1	$L = 9$, $K = 8$, $r = 3$, $n = 10$, 0.05 %	4 minutos y 4 segundos
1	2	$L = 9$, $K = 16$, $n = 10$, 0.05 %	8 minutos
2	2	$L = 9$, $K = 8$, $r = 3$, $n = 10$, 0.05 %	5 minutos
1	3	$L = 5$, $K = 16$, $n = 10$, 0.05 %	57 segundos
2	3	$L = 5$, $K = 16$, $r = 5$, $n = 10$, 0.05 %	1 minuto y 18 segundos
1	4	$L = 9$, $K = 16$, $n = 10$, 0.05 %	2 minutos y 3 segundos
2	4	$L = 9$, $K = 8$, $r = 3$, $n = 10$, 0.1 %	3 minutos y 30 segundos
1	5	$L = 5$, $K = 16$, $n = 10$, 0.05 %	4 minutos y 34 segundos
2	5	$L = 9$, $K = 8$, $r = 5$, $n = 10$, 0.05 %	11 minutos
1	6	$L = 11$, $K = 8$, $n = 10$, 0.1 %	11 minutos y 15 segundos
2	6	$L = 11$, $K = 8$, $r = 3$, $n = 10$, 0.05 %	9 minutos y 30 segundos
1	7	$L = 5$, $K = 16$, $n = 10$, 0.05 %	8 minutos
2	7	$L = 5$, $K = 8$, $r = 5$, $n = 10$, 0.05 %	11 minutos y 38 segundos
1	8	$L = 9$, $K = 16$, $n = 10$, 0.05 %	18 minutos
2	8	$L = 9$, $K = 16$, $r = 5$, $n = 10$, 0.05 %	29 minutos
1	9	$L = 9$, $K = 16$, $n = 10$, 0.05 %	13 minutos y 40 segundos
2	9	$L = 9$, $K = 16$, $r = 5$, $n = 10$, 0.05 %	22 minutos

Tabla 4.21: Ejecuciones para procesar las imágenes de la Figura 4.28.

Los resultados obtenidos para ambas tendencias paralelas propuestas con respecto a cada

Figura 4.27: Imágenes a tratar en los experimentos nivel 3.

imagen a procesar se visualizan en las Figuras 4.29, 4.30, 4.31, 4.32, 4.33, 4.34, 4.35, 4.36 y 4.37 respectivamente.

En general se obtienen buenos resultados por parte de las dos tendencias paralelas. Sin embargo, en algunos casos la implementación *GPU T1* suele ser mejor que *GPU T2* y en otras situaciones ocurre lo contrario. Cuando los parámetros L, K, n y el porcentaje para calcular N son los mismos para cada algoritmo, *GPU T1* se ejecuta más rápido, tal como lo indica la Tabla 4.21 en el tratamiento de las imágenes 3, 8 y 9.

Cuando algunos de los parámetros en *GPU T2* son menores, como en el procesamiento de las imágenes 1, 2 y 6, dicho algoritmo se ejecuta más rápido. En las demás imágenes contempladas, los parámetros son variables. Un aspecto notable es el presentado en la imagen 5, en la cual la implementación *GPU T1* se ejecuta más rápido que la segunda, teniendo presente que en *GPU T1* el tamaño de la ventana es menor y el valor de K es menor en *GPU T2* con respecto a la implementación *GPU T1*. A partir de aquí, se deduce que el tamaño de la ventana tiene mayor influencia que la variable de propagación K en el factor tiempo. Otra situación interesante es la que ocurre en la imagen 4, en donde el valor K de la implementación *GPU T2* es menor que la primera, pero el porcentaje para calcular N es más grande y la implementación *GPU T1* gana en tiempo.

Entonces, se deduce que el parámetro que representa el tamaño del sistema de vecindad

Figura 4.28: Áreas a tratar en cada imagen de la Figura 4.27.

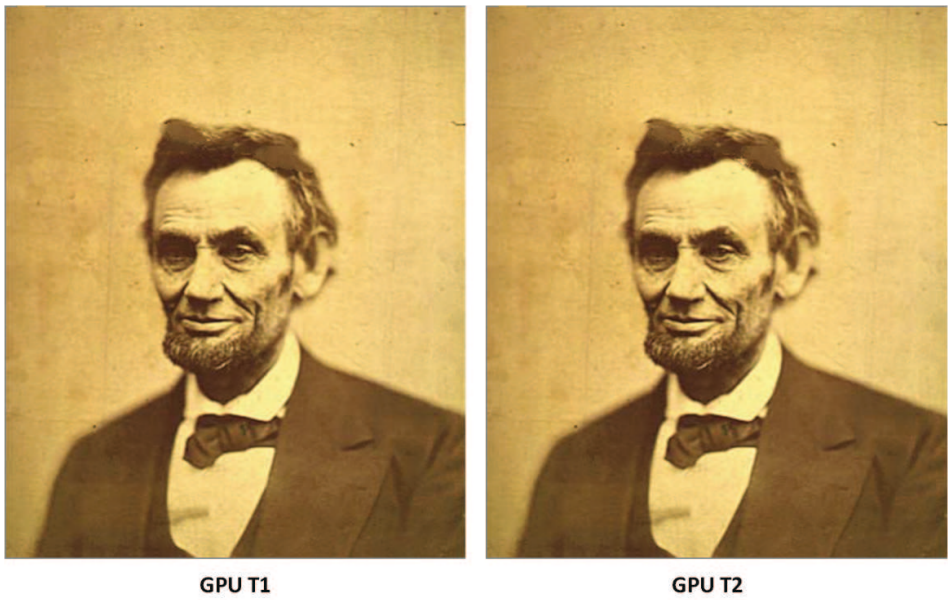


Figura 4.29: Versión paralela: resultado de la imagen 1.

Figura 4.30: Versión paralela: resultado de la imagen 2.

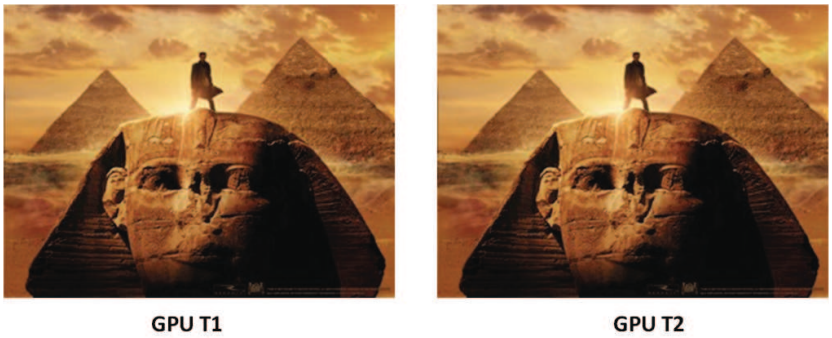


Figura 4.31: Versión paralela: resultado de la imagen 3.

Figura 4.32: Versión paralela: resultado de la imagen 4.

influye directamente en el tiempo de ejecución. Al mismo tiempo, otro factor influyente es el porcentaje para calcular N y por último el valor K. Pero es importante contemplar que la variable K afecta el tiempo de ejecución cuando se sintetiza la textura, además del número de iteraciones r.

En algunos casos se logra un buen equilibrio en los parámetros, como en el caso de la imagen 3, donde se usa en las iteraciones el valor más alto, en la variable de propagación un valor intermedio y en el L un valor pequeño, obteniéndose buenos resultados y un buen

Figura 4.33: Versión paralela: resultado de la imagen 5.

Figura 4.34: Versión paralela: resultado de la imagen 6.

Figura 4.35: Versión paralela: resultado de la imagen 7.

Figura 4.36: Versión paralela: resultado de la imagen 8.

Figura 4.37: Versión paralela: resultado de la imagen 9.

tiempo de respuesta. Así, resulta difícil encontrar la combinación adecuada para cada imagen. Los valores obtenidos en las Tablas 4.6, 4.7 y 4.8 han sido útiles para escoger que parámetros utilizar en los diferentes experimentos nivel 3. Sin embargo, pueden existir imágenes que exijan valores que no están presentes en dichas tablas. Los tiempos de respuesta dependen también del número de píxeles en Ω y la forma que esta posea, más que en las dimensiones de la imagen, como se estimaba en un principio.

Observando las Tablas 4.20 y 4.21, se observa que la imagen 7 de tamaño 600×938 píxeles se ejecuta más rápido que la imagen 8, y la diferencia en tiempo entre ellas es considerable. Aquí notamos que la cantidad de píxeles y la forma del Ω tiene gran importancia. A partir de aquí se deduce que en primer lugar influye la cantidad de píxeles a restaurar y segundo el porcentaje que el área a tratar ocupe de la imagen. Cuando el área es amplia y con un porcentaje importante, se requiere parámetros con valores más altos en las dimensiones del sistema de vecindad, de iteraciones y en algunas ocasiones de N para que se puedan obtener los resultados deseados.

4.3. Comparaciones

En esta sección se efectuarán dos tipos de comparaciones: los tiempos de respuestas de las versiones paralelas con respecto a la versión secuencial y comparaciones visuales entre los mejores resultados de tres imágenes (B,C y D de los experimentos nivel 2) y resultados obtenidos en otros trabajos de esta índole.

4.3.1. CPU vs. GPU

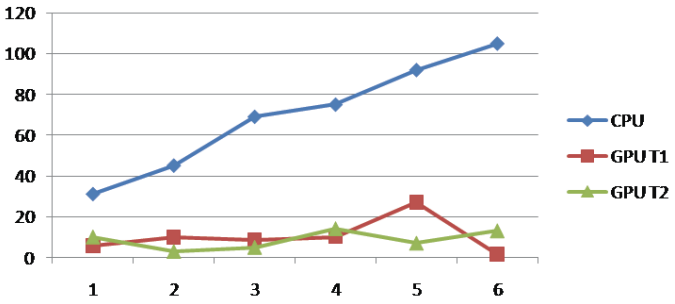
Se ha visto que los algoritmos paralelos proporcionan resultados satisfactorios y en muchos casos iguales a la versión secuencial, como ocurre con la reconstrucción de la imagen A del experimento nivel 1. En otras ocasiones produce resultados mejores como el caso de la imagen B. Sin embargo. existen sus excepciones como el caso de la imagen D. Con esto se hace notar, que en la mayoría de los casos la calidad es superior o igual a la versión secuencial. Esto es un aspecto positivo, debido a que en absolutamente todos los experimentos presentados, la versión paralela en la GPU en sus distintas tendencias siempre es significativamente más rápida que el algoritmo en la CPU, tal como lo indica la Figura 4.38.

Dado que comparar la calidad visual de la imagen es un elemento subjetivo, hay aspectos obvios que dependen de la percepción del usuario. Esto fue algo notable en los experimentos nivel 1, donde las personas varían su percepción para ciertos resultados obtenidos. Pero coinciden en rechazar una imagen que notablemente no fue reconstruida como se esperaba o en aceptar los mejores resultados. Un aspecto interesante en los resultados de los experimentos nivel 1 es que en algunas combinaciones de CPU, la boca del personaje de la Figura 4.3 se unía en muchos casos, algo que es significativamente incorrecto, mientras que en la versión paralela en *GPU T2*, sucedió una vez ligeramente y en la primera versión nunca ocurrió. Esto hace pensar que existen más probabilidades que la calidad visual sea mejor en la versión paralela que en la versión secuencial, aunque puedan existir excepciones, como el caso de la imagen D del experimento nivel 2.

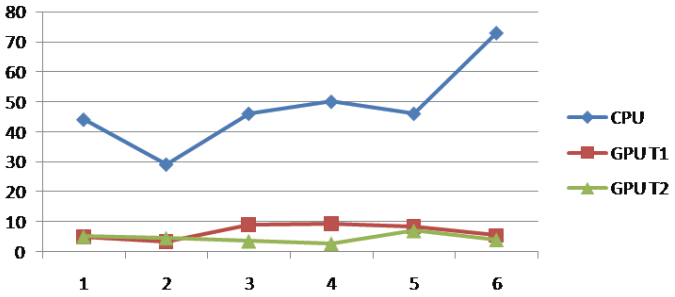
4.3.2. *k-Inpainting* vs. otros algoritmos

Las imágenes B, C y D presentadas en los experimentos nivel 2, han sido también objeto de estudio en otras investigaciones de esta índole. Como se desconoce el tiempo de ejecución de dos de dichas imágenes, se realizarán comparaciones a nivel visual. En las Figuras 4.39, 4.40 y 4.41, se comparan los mejores resultados del algoritmo secuencial y las variantes paralelas con el resultado obtenido de otros trabajos de investigación.

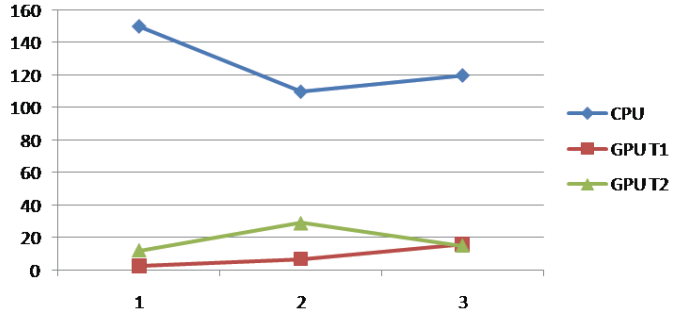
Observando los resultados, se puede decir que los algoritmos presentados en este documento proporcionan buenos resultados, especialmente los algoritmos de la versión secuencial y de la implementación *GPU T1* de la versión paralela.



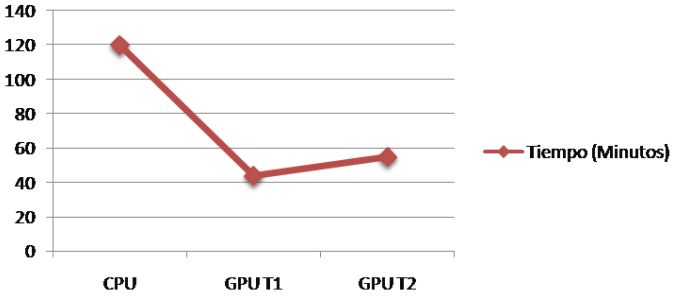
(a) Imagen A



(b) Imagen B en CPU

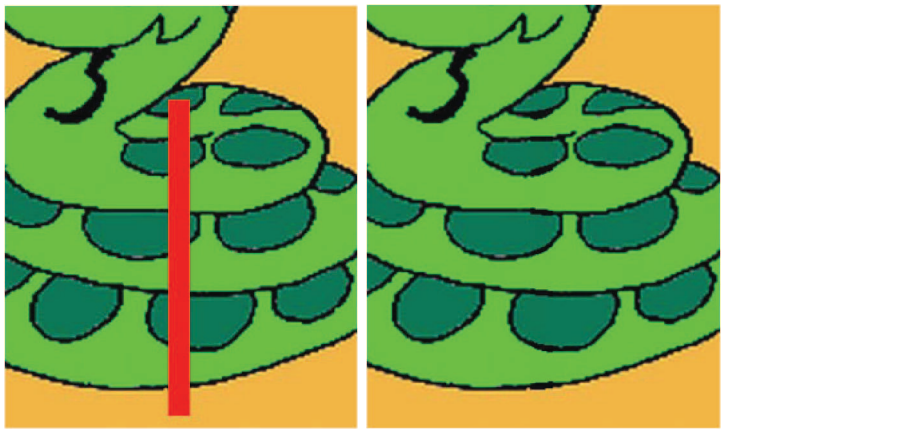


(c) Imagen C



(d) Imagen D

Figura 4.38: Comparación de tiempos de ejecución de las imágenes del experimento nivel 2.



(a) Área a tratar (b) *Inpainting* en [42]

(c) *k-Inpainting* en CPU (d) *k-Inpainting* con GPU T1 (e) *k-Inpainting* con GPU T2

Figura 4.39: Resultados vs. otros algoritmos: Imagen B.

4.4. Consideraciones finales

El enfoque híbrido logra el objetivo, sin embargo el tiempo de respuesta no es inmediato y en algunos casos puede tardar más de lo previsto. Esto en parte es debido a que la propuesta realiza todas sus funciones a nivel de píxeles, lo que hace más lento el tratamiento de la región a reconstruir. Las versiones en GPU ciertamente aceleran el proceso, como se ha visto en el transcurso de todo el capítulo, pero el API utilizado tiene ciertas características a nivel de paralelismo y de memoria que limitan su uso. La principal limitación a nivel de paralelismo es la falta de jerarquía a nivel de hilos, que sería ideal emplear en nuestra implementación. El uso de la memoria y el poder de cómputo está restringido por el hardware de la tarjeta.

Un detalle importante en el algoritmo, como se ha resaltado previamente, es la combinación de los parámetros. No siempre funciona utilizar valores altos, ya que muchas iteraciones en la síntesis de textura o del tamaño de la ventana, puede generar artefactos. Si la imagen es de texturas simples y sin tanta estructura, se pueden usar valores bajos. Si tiene estructura y textura simple, se toma un valor alto en las dimensiones de la ventana, pero bajos en el K y

(a) Área a tratar (b) *Inpainting* en [60]

(c) *k-Inpainting* en CPU (d) *k-Inpainting* con GPU T1 (e) *k-Inpainting* con GPU T1

Figura 4.40: Resultados vs. otros algoritmos: Imagen C.

en las iteraciones. Si tiene estructura intermedia y variaciones de textura, se pueden emplear valores intermedios. En caso de exigir en todos los sentidos, se necesitan valores altos. El punto de referencia es la configuración: $L = 9$, $K = 16$, $r = 5$ y $n = 10$, esta combinación generalmente funciona para la mayoría y en caso de que no, se puede deducir a partir de un resultado qué valores realmente convienen a una imagen.

El algoritmo es totalmente automatizado, el usuario solo debe indicar los parámetros y la región a procesar.

(a) Área a tratar

(b) *Inpainting* en [60]

(c) *Inpainting* en [44]

(d) *Inpainting* en [15]

(e) *k-Inpainting* en CPU

(f) *k-Inpainting* con GPU T1

(g) *k-Inpainting* con GPU T1

Figura 4.41: Resultados vs. otros algoritmos: Imagen D.

Capítulo 5

Conclusiones y Trabajos futuros

5.1. Conclusiones

El algoritmo de *Inpainting* es empleado para reconstruir partes dañadas o indeseadas de una imagen. Es un algoritmo complejo, especialmente cuando es automático y requiere una intervención mínima del usuario. Existen aplicaciones donde dicho usuario señala segmentos a seguir en la imagen para la coherencia en el área ma reconstruir u otros donde se indica al algoritmo que parches usar o sobre que área de la imagen extraer los candidatos. Con esto se quiere enfocar que entre más automatizado es el sistema, requiere de mayor nivel de complejidad.

En este trabajo se presentó un nuevo algoritmo híbrido denominado *k-Inpainting*. Nuestro algoritmo puede ser más lento que otros algoritmos propuestos basados en síntesis de textura. Sin embargo, *k-Inpainting* provee buenos resultados y logra el objetivo de la técnica. La implementación funciona eficientemente cuando no hay partes importantes del agujero en los bordes de la imagen, completa bien los contornos, realiza una buena síntesis de textura y no es estrictamente necesario aplicar una técnica de difusión al área que ha sido tratada o un post-procesamiento adicional. Adicionalmente, su rapidez depende más de las dimensiones de la región Ω a ser reconstruida y de los parámetros que del tamaño mismo de la imagen. Los parámetros que mas influyen en el tiempo son las dimensiones del sistema de vecindad, el parámetro de propagación y el parámetros N. Hasta ahora no existe para este algoritmo una combinación que funcione para todas las imágenes, por lo que sería necesario estimar estos valores también de forma automática. Un aspecto importante es que la selección a mano alzada no es la ideal, pues se pierden píxeles que podrían ayudar a reconstruir mejor el área a tratar.

En esta aplicación solo se contemplan las imágenes, sin embargo, pensando en una extensión para video, cuando se desea retirar un cuerpo en movimiento, se proveen más datos a las imágenes del video a reconstruir, ya que a través de las imágenes en otros momentos del mismo se puede extraer información. Por ello se considera que la reconstrucción de imágenes estáticas tiene una mayor complejidad a reconstruir un área donde había un cuerpo en movimiento.

La primera implementación de la versión paralela *GPU T1*, es un algoritmo que es al-

tamente paralelizable en el procesamiento de texturas. Este arroja buenos resultados en un tiempo menor, en caso que se utilicen los mismos parámetros que la segunda implementación.

Por otro lado, la segunda implementación *GPU T2*, es una algoritmo muy similar a la versión secuencial del algoritmo, pero no es una traducción exacta y esto lo certifican los resultados obtenidos en las pruebas realizadas. Sin embargo, el empleo de los mismos parámetros para una imagen en específico, tienen alta similitud.

El cálculo de la diferencia de vecindades es costoso en el tiempo, cuando hay que procesar una cantidad considerable de píxeles. Este fenómeno ocurre incluso en las variantes paralelas, ya que esta función es secuencial en si misma, es decir que la realiza un proceso o hilo a la vez. En CUDA no se pueden crear hilos a partir de otros ya que no hay un esquema jerárquico de hilos. El uso de la tarjeta gráfica para propósito general aporta un gran beneficio en diversos campos de la investigación para algoritmos que necesitan paralelismo masivo a un bajo costo.

En este trabajo se presentó un primer paso de un enfoque híbrido, el cual permitirá explotar diversas variantes en cualquier etapa de *k-Inpainting*.

5.2. Trabajos futuros

Diversos cambios se proponen con el objetivo de mejorar nuestra propuesta. Hasta ahora, se ha construido el algoritmo de forma secuencial y dos variantes paralelas. También se ha construido una interfaz sencilla para que el usuario seleccione el área a tratar. Este trabajo logra el objetivo de la técnica *Inpainting*; pero se proponen un conjunto de mejoras importantes:

- Integrar las versiones paralelas y secuencial en una misma interfaz de usuario, a través del uso de librerías de enlace dinámico DLL.
- Elaborar otras herramientas para el usuario, tal que faciliten la selección de la región a tratar, así como hacerla más precisa.
- Permitir reconstruir varias áreas a la vez (la versión actual solo realiza un área por ejecución del algoritmo).
- Hallar un método automático para detectar los parámetros más adecuados de una imagen de entrada.
- Incluir el uso de OpenMP [61] en el algoritmo secuencial para acelerar los tiempos de respuesta.
- Optimizar las variantes paralelas. Por ejemplo, utilizar estructuras de datos que exploten más la capacidad de la tarjeta gráfica.

Para acelerar el tiempo de respuesta, se propone elaborar una versión basada en parches, en vez de basada en píxeles. En cuanto a la implementación, se propone migrar el código

actual desarrollado en CUDA a un ambiente independiente de la arquitectura (e.g. OpenCL, OpenGL v. 4.2).

Bibliografía

[1] L. Demanet, B. Song, y T. Chan, "Image Inpainting by Correspondence Maps: a Deterministic Approach," Instituto Tecnológico de Caligornia y Universidad de California, Los Angeles, USA, Rep. Tec., 2003.

[2] S. Li, *Markov Random Field Modeling in Image Analysis*. Londres: Springer, 1999.

[3] Texture Analysis and Synthesis. [En línea]. Disponible en: http://graphics.stanford.edu/projects/texture/

[4] D. Lanman, "Texture synthesis and manipulation," 2006, Brown University. [En línea]. Disponible en: http://web.media.mit.edu/ dlanman/courses/en256/Lanman-Quilting-Proposal.pdf

[5] A. Efros y W. Freeman, "Image quilting for texture synthesis and transfer," en *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 341–346. [En línea]. Disponible en: http://doi.acm.org/10.1145/383259.383296

[6] Inpainting. [En línea]. Disponible en: http://iat.ubalt.edu/summers/math/inpainting.htm

[7] M. Ashikhmin, "Synthesizing natural textures," en *Proceedings of the 2001 symposium on Interactive 3D graphics*, I3D '01. New York, NY, USA: ACM, 2001, pp. 217–226. [En línea]. Disponible en: http://doi.acm.org/10.1145/364338.364405

[8] Y. Wexler, E. Shechtman, y M. Irani, "Space-Time Completion of Video," *IEEE Transactions On Pattern Analysis And Machine Intelligence*, vol. 29, pp. 463–476, 2007.

[9] C. Barnes, E. Shechtman, D. Goldman, y A. Finkelstein, "The generalized patchmatch correspondence algorithm," en *Proceedings of the 11th European conference on computer vision conference on Computer vision: Part III*, ECCV'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 29–43. [En línea]. Disponible en: http://dl.acm.org/citation.cfm?id=1927006.1927010

[10] Nvidia Corporation, "NVIDIA CUDA Programming Guide," U.S.A, 2009. [En línea]. Disponible en: www.nvidia.com/

[11] N. Corporation, "CUDA Programming Model Overview." [En línea]. Disponible en: www.nvidia.com/

[12] H. Hoeger, "Introducción a la Computación Paralela," pp. 26–31, 2010.

[13] Nvidia Corporation, "Cuda programming basics - part i." [En línea]. Disponible en: www.nvidia.com/

[14] ——, "Cuda programming basics - part ii." [En línea]. Disponible en: www.nvidia.com/

[15] A. Bugeau, M. Bertalamío, V. Caselles, y G. Shapiro, "A comprehensive framework for image inpainting," *IEEE Transactions on image processsing*, vol. 19, pp. 2634–2645, 2010.

[16] J. F. Aujol, S. Ladjal, y S. Masnou, "Exemplar-based inpainting from a variational point of view," pp. 2634–2645, 2009.

[17] M. Bertalmio, G. Sapiro, V. Caselles, y C. Ballester, "Image inpainting," en *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 417–424. [En línea]. Disponible en: http://dx.doi.org/10.1145/344779.344972

[18] M. Taschler, "A comparative analysis of image inpainting techniques," Ph.D. dissertation, 2006.

[19] Texture Aliasing Problems. [En línea]. Disponible en: http://www.relisoft.com/science/graphics/alias.html

[20] L.-Y. Wei y M. Levoy, "Fast texture synthesis using tree-structured vector quantization," en *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 479–488. [En línea]. Disponible en: http://dx.doi.org/10.1145/344779.345009

[21] A. Efros y T. Leung, "Texture synthesis by non-parametric sampling," *IEEE International Conference Vision*, 1999.

[22] A. Hertzmann, C. Jacobs, N. Oliver, B. Curless, y D. Salesin, "Image analogies," en *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01. New York, NY, USA: ACM, 2001, pp. 327–340. [En línea]. Disponible en: http://doi.acm.org/10.1145/383259.383295

[23] S. Shin, T.Ñishita, y S. Y. Shin, "On pixelbased texture synthesis by non-parametric sampling," *Computer & Graphics*, vol. 30, pp. 767–778, 2006.

[24] Y.-Q. Xu, B. Guo, y H. Shum, "Chaos mosaic: Fast and memory efficient texture synthesis, Rep. Tec. MSR-TR-2000-32, 2000.

[25] E. Praun, A. Finkelstein, y H. Hoppe, "Lapped textures," en *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '00. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000, pp. 465–470. [En línea]. Disponible en: http://dx.doi.org/10.1145/344779.344987

[26] L. Liang, C. Liu, Y.-Q. Xu, B. Guo, y H.-Y. Shum, "Real-time texture synthesis by patch-based sampling," *ACM Trans. Graph.*, vol. 20, pp. 127–150, July 2001. [En línea]. Disponible en: http://doi.acm.org/10.1145/501786.501787

[27] R. Szeliski y H.-Y. Shum, "Creating full view panoramic image mosaics and environment maps," en *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '97. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1997, pp. 251–258. [En línea]. Disponible en: http://dx.doi.org/10.1145/258734.258861

[28] M. Cohen, J. Shade, S. Hiller, y O. Deussen, "Wang tiles for image and texture generation," *ACM Trans. Graph.*, vol. 22, pp. 287–294, July 2003. [En línea]. Disponible en: http://doi.acm.org/10.1145/882262.882265

[29] V. Kwatra, A. Schödl, I. Essa, G. Turk, y A. Bobick, "Graphcut textures: image and video synthesis using graph cuts," *ACM Trans. Graph.*, vol. 22, pp. 277–286, July 2003. [En línea]. Disponible en: http://doi.acm.org/10.1145/882262.882264

[30] Q. Wu y Y. Yu, "Feature matching and deformation for texture synthesis," *ACM Transactions on Graphics*, vol. 23, pp. 364–367, August 2004. [En línea]. Disponible en: http://doi.acm.org/10.1145/1015706.1015730

[31] S. Masnou, "Disocclusion: a variational aproach using level lines," en *IEEE Transaction on Image Processing*, vol. 11, 2002, pp. 68–76.

[32] Universidad de Buenos Aires. [En línea]. Disponible en: http://www-2.dc.uba.ar/materias/ipdi/difusion.pdf

[33] M. Bertalmio, G. Sapiro, V. Caselles, y C. Ballester, "Processing of flat and non-flat image information on arbitrary maniflods using partial differential equations," Ph.D. dissertation, 2001.

[34] M. Oliveira, B. Bowen, R. McKenna, y Y.-S. Chang, "Fast digital image inpainting," *the Proceedings of the International Conference on Visualization, Imaging and Image Processing*, pp. 261–266, 2001.

[35] T. Chan y J. Shen, "Non-texture inpainting by curvature-driven," *Journal of Visual Communication and Image Representation*, vol. 12, pp. 436–449, 2001.

[36] ——, "Matematical models for local non-texture inpainting," *SIAM Journal of Applied Mathematics*, vol. 62, pp. 1019–1043, 2001.

[37] D. Tschumperlé y R. Deriche, "Vector- value image regularization with pdes: A common framework for different applications," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 27, pp. 506–517, 2005.

[38] R. Day y R. Kasperczyk, "Amodal completion as a basis for illusory contours," *Attention, Perception, & Psychophysics*, vol. 33, pp. 355–364, 1983, 10.3758/BF03205882. [En línea]. Disponible en: http://dx.doi.org/10.3758/BF03205882

[39] D. Fishelov y N. Sochen, "Image inpainting via fluid equations." en *ITRE'06*, 2006, pp. 23–25.

[40] L. Atzori y F. DeÑatale, "Reconstruction of missing or occluded contour segments using bezier interpolations," *Signal Process.*, vol. 80, pp. 1691–1694, August 2000. [En línea]. Disponible en: http://dx.doi.org/10.1016/S0165-1684(00)00130-4

[41] A. Rares, M. Reinders, y J. Biemond, "Edge-based image restoration," *IEEE Transaction on Image Processing*, vol. 14, pp. 1454–1468, 2005.

[42] J. Yu-Sung, C. Hwang, Y. Liao, N. Tang, y T. Chen, "Exemplar-based image inpainting base on structure construction," *Journal Of Software*, vol. 3, pp. 57–64, 2008.

[43] K. Plataniotis y A. Venetsanopoulos, *Color Image Processing and Applications*. Berlín: SpringerVerla, 2000.

[44] N. Komodakis, "Image completion using global optimization," en *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 1*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 442–452. [En línea]. Disponible en: http://dl.acm.org/citation.cfm?id=1153170.1153498

[45] K. Cao, K. Ding, G. Chirstensen, *et al.*, "Unifying vasular information in Intensity - Based Norigid Lung CT Registration," *Bioedical Image Registration*, pp. 5–6, 2010.

[46] P. Burt y E. Adelson, "The Laplacian Pyramid as a Compact Image Code," *IEEE Transactions On Communications*, vol. 31, pp. 532–540, 1983.

[47] A. Akepogu y R. Palagiri, *Data Structures And Algorithms Using C++*. India: Pearson, 2011.

[48] R. Sedgewick, *Algoritmos en C++*. Estados Unidos de América: Adinson - Wesley/ Diaz de Dantos, 1995.

[49] W. Press, S. Teukolsky, W. Vetterling, y B. Flannery, *Numerical Recipes in C*, 2da. ed. USA: Cambridge, 2002.

[50] H.Ñavarro, "Notas de docencia tap," 2007.

[51] C. Barnes, E. Shechtman, A. Finkelstein, y D. Goldman, "Patchmatch: a randomized correspondence algorithm for structural image editing," *ACM Trans. Graph.*, vol. 28, pp. 24:1–24:11, July 2009. [En línea]. Disponible en: http://doi.acm.org/10.1145/1531326.1531330

[52] Nvidia Corporation, "Graphics processing unit (gpu)," http://www.nvidia.com/object/gpu.html, 2010.

[53] J. Fung y S. Mann, "Computer Vision Signal Processing on Graphics Processing Units," en *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing*, Montreal, Quebec, Canada, 2004, pp. 83–89.

[54] T. Hagen, A. Lie Knut, y J.Ñatvig, "Solving the Euler Equations on Graphics," en *Proceedings of the 6th International Conference on Computational Science*. Noruega: Springer, 2006, pp. 220–227.

[55] A. Cuno, "Wordpress," 2010. [En línea]. Disponible en: http://compinformatidf.files.wordpress.com/2010/01/gpus1.pdf

[56] O. Plata, "Difusión isotrópica," Universidad de Málaga, 2005. [En línea]. Disponible en: http://www.ac.uma.es/educacion/cursos/ informatica/Multip/traspas/02-ArqParalelas - Introduccion.pdf

[57] A. Santos, *Beginning XNA 3.0 Game programming: From Novice to Professional*. New York: Apress, 2009.

[58] A. Lovesey, "A Comparison of Real Time Graphical Shading, Rep. Tec. CS4983, 2005.

[59] A. Gómez, I. De Jesús, y A. Briceño, *Introducción a la computación*. México: Coordinadores Editoriales, 2008.

[60] A. Criminisi, P. Perez, y K. Toyama, "Object Removal by Exemplar-based Inpainting," Madison, pp. II–721–II–728, 2003.

[61] OpenMP Architecture Review Board, "Open multi-processing (openmp)." [En línea]. Disponible en: www.openmp.org

[62] Y. Wexler, E. Shechtman, y M. Irani, "Space-time video completion," en *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2004, pp. I120–I127.

[63] M. Oliveira, B. Bowen, R. MacKenna, y Y.-S. Chang, "Fast digital image inpainting," en *Proceedings of the IASTED International Conference on Visualization, Imaging and Image Processing (VIIP 2001)*, España, 2001, pp. 261–266.

[64] J. MacConnell, *Analysis of Algorithms An Active Learning Approach*. India: Pearson, 2011.

Anexos

5.3. CUDA

5.3.1. Instalación y ejecución

Al instalar CUDA, este trae consigo tres componentes[13]:

- *Drivers*.
- Herramientas del compilador.
- Ejemplos en el SDK de CUDA.

En primera instancia se debe conocer como compilar un programa en CUDA, por ello se presentan en la Tabla A.2 las 4 configuraciones posible de compilación.

Configuración	Característica
nvcc <nombre del archivo>.cu [o- ejecutable]	Se refiere al modo de compilación y construcción por defecto
nvcc g- <nombre del archivo>.cu	Compilación en modo *debug*. En este caso se se depura el código de *host*, pero no el del *device*
nvcc -deviceemu <nombre del archivo>.cu	Genera una emulación del *device* sobre la CPU, pero no carga los símbolos de *debug*
nvcc -deviceemu -g <nombre del archivo>.cu	Genera una emulación del código del *device* en la CPU con los símbolos de *debug* correspondientes a dicho código.

Tabla 5.1: Construcción del código de CUDA [13].

5.3.2. Jerarquía de memoria

Una GPU posee N cantidad de multiprocesadores según su especificación. Cada multiprocesador del *device* (GPU) contiene generalmente 8 procesadores de hilos, de manera que

N*8 procesadores de hilos ejecutan los hilos de *kernel*[1]. Cada hilo de un multiprocesador posee un pequeño espacio de memoria denominado registros y cada multiprocesador posee una memoria compartida para habilitar la cooperación entre los hilos. Además de los espacios de memoria de cada multiprocesador dentro del *device*, éste también posee un espacio de memoria (DRAM) el cual se encuentra dividido en memoria local y global [11]. Así cada *kernel* posee diferentes accesos de memoria en diferentes niveles, que serán descritos brevemente a continuación:

- *Por hilo*: Puede acceder a los registros dentro del chip del multiprocesador y a la memoria local no cacheada fuera de dicho chip.
- *Por bloque*: Cada bloque puede acceder rápidamente a la pequeña memoria compartida dentro del chip.
- *Por dispositivo(device)*: Cada *kernel* tiene acceso a la memoria global fuera del chip que no es cacheada. Esta memoria global es consistente en el tiempo y permite operaciones de entrada y salida entre *kernels*.

La CPU y la GPU se comunican a través de la memoria global, la cual puede ser accedida por la CPU para lectura y escritura, tal como se muestra en el diagrama de la Figura 5.1

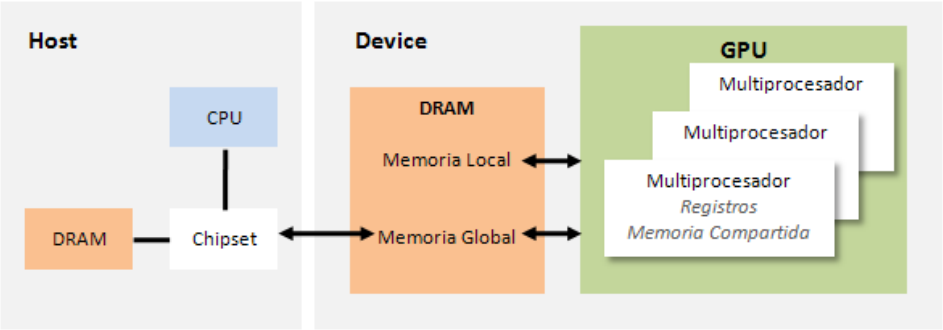


Figura 5.1: Memoria física de una tarjeta gráfica [11].

Manejo de memoria

El manejo de la memoria global del dispositivo se realiza mediante las siguientes instrucciones:

- `cudaMalloc`(void ** pointer, int value, size_t nbyte): Reserva un espacio de memoria en el *device* de tamaño nbyte y devuelve en pointer, el puntero a la memoria asignada.

[1]Al hablar de hilos de kernels en este contexto se refiere a los hilos de *kernel* de CUDA y no del sistema operativo.

- `cudaMemset`(void * pointer, int value, size_t count): Asigna el valor value a *count* bytes a partir de la dirección de dispositivo *pointer*.
- `cudaFree`(void * pointer): Libera la memoria apuntada por pointer en el dispositivo.
- `cudaMemCpy`(void * dest, void * src, size_t nbytes, enum cudaMemcpyKind direction): Copia datos en la dirección de memoria dest desde la dirección de memoria src. El destino (dest) y la fuente (src) puede ser una dirección de *host* o *device*, esto depende del enumerado `cudaMemcpyKind` *direction*, el cual es de tres tipos:

  1. `cudaMemcpyHostToDevice`: Copiar datos desde el *host* al *device*.
  2. `cudaMemcpyDeviceToHost`: Copiar datos desde el *device* al *host*.
  3. `cudaMemcpyDeviceToDevice`: Copiar datos desde un *device* a otro.

  Esta instrucción garantiza la seguridad entre hilos, debido a que esta instrucción es síncrona y los hilos se bloquean mientras esta se ejecuta por completo.

Las variables declaradas en memoria tienen asociado un tipo de clasificador que especifica el tipo de memoria en la cual se encuentran. Los tipos de variables calificadas se listan a continuación [10]:

- `_device_`: Declara una variable que reside en el *device*. Esta variable puede ser definida con el próximo clasificador a explicar para especificar con mayor detalle el espacio de memoria al que la variable pertenece. Si el otro clasificador no está presente, la variable:
  - Reside en el espacio de la memoria global.
  - Posee el mismo tiempo de vida de la aplicación.
  - Es accedido por todos los hilos.
- `_shared_`: Calificador usado opcionalmente junto con `_device_`, el cual declara una variable que:
  - Reside en el espacio de memoria compartida (muy baja latencia)
  - Tiene el tiempo de vida del bloque de hilos.
  - Accedido solo por los hilos dentro del bloque de hilos.

5.3.3. Ejecución

Por otro lado, el *software* de desarrollo de CUDA comprende un conjunto de librerías optimizadas, encontrándose entre las más importantes: math.h, FFT (*Fast Fourier Transform*) y BLAS (*Basic Linear Algebra Subprograms*). Además proporciona un código fuente C que integra la CPU con la GPU, un compilador C de NVIDIA, un lenguaje intermedio denominado

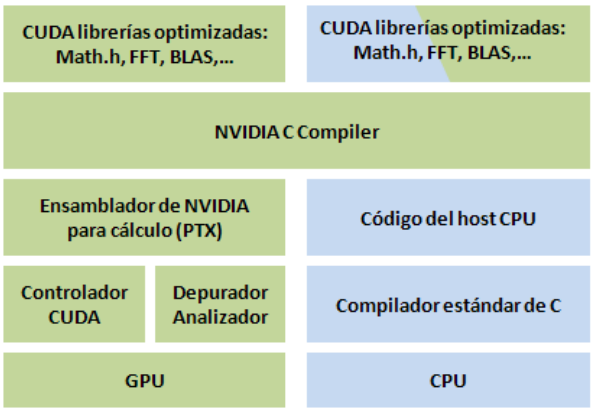


Figura 5.2: *Software* de desarrollo de CUDA [13].

PTX, los drivers de CUDA, un depurador [13] y demás componentes que pueden visualizarse en la Figura 5.2.

CUDA compila una aplicación C/C++ de CUDA enviándola al NVCC (compilador C de NVIDIA), en donde se generan dos códigos objetos, uno para la CPU y otro para la GPU. El código que va a la GPU es pasado previamente por el lenguaje intermedio PTX independiente de la plataforma, el cual transforma el código al vuelo al momento de la ejecución, a un binario que pueda ser ejecutado por la tarjeta gráfica sin importar cual sea esta (Ver Figura 5.3).

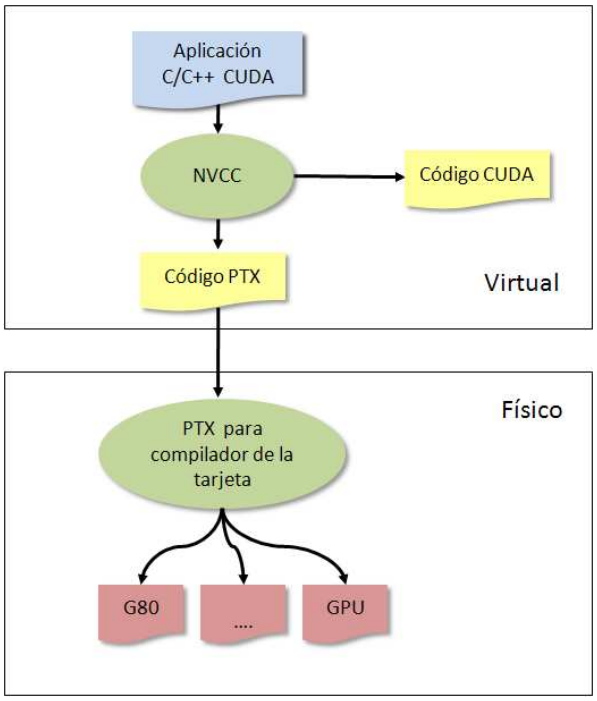


Figura 5.3: Compilando CUDA [13].

El modelo de ejecución de CUDA se resume en los siguientes ítems:

- Los hilos son procesados por un procesador de hilos.
- Los boques de hilos son ejecutados en multiprocesadores.
- Los bloques de hilos no migran.
- La cantidad bloques de hilos concurrentes que residen en un multiprocesador están limitados por la memoria compartida y los registros de cada hilo.
- Un *kernel* es lanzado como un *grid* de bloques de hilos.
- Solo un *kernel* puede ejecutarse sobre un dispositivo a la vez.

Todos los kernels convocados por el *host* son asíncronos, por lo que el control retorna a la CPU inmediatamente, pero los *kernels* se ejecutan después que todas las llamadas previas a CUDA han sido completadas. En el caso de la sincronización del *device*, los bloques de un *kernel* pueden cooperar entre ellos mediante una memoria compartida y sincronizando su ejecución para coordinar los accesos a memoria. En cada caso se emplean diferentes comandos que el programador puede utilizar para cooperar con la sincronización de estos dispositivos, tales comandos se discutirán a continuación.

Lanzar *kernels* en CUDA

Un *kernel* es definido mediante el clasificador `_global_` y es llamado desde el *host*. La sintáxis para llamar a dicho *kernel* está modificada con respecto a una llamada de una función de C. Esta sintáxis es como sigue[14]:
`Kernel«<dim3 dG, dim3 dB»>(parametros)`, donde:

- dG define la dimensión del *grid* y la cantidad de bloques que esta contiene. El *grid* tiene dos dimensiones como máximo *"x"* y *z"*. La cantidad de bloques generados es dG.x*dG.y.
- dB define la dimensión del bloque y la cantidad de hilos por bloque. El bloque puede tener hasta tres dimensiones *"x"*, *z"* y *"z"*. Y como es de esperarse, la cantidad de hilos por bloque es dB.x*db.y*db.z.

Las variables *x*,*y* para un grid o *x*,*y*,*z* para un bloque que no sean definidas, tendrán como valor por defecto 1.

Los valores *x*, *y* o *z*, según sea el caso son establecidos desde el *host*, la Figura 5.4 muestra tres formas de establecerlo en los cuadros numerados en esta. Para el primer cuadro el establecimiento de los valores se realiza a través de la indexación de *x*, *y* o *z*. En el segundo cuadro, las dimensiones son establecidas en la misma declaración del *grid* y/o bloque. Y en el tercer caso se puede observar que el *grid* y el bloque son definidos mediante un solo número, por lo cual es un *grid* y un arreglo unidimensional.

Todas las funciones que poseen los clasificadores `_global_` o `_device_`, tienen acceso a las siguientes variables automáticamente definidas por CUDA:

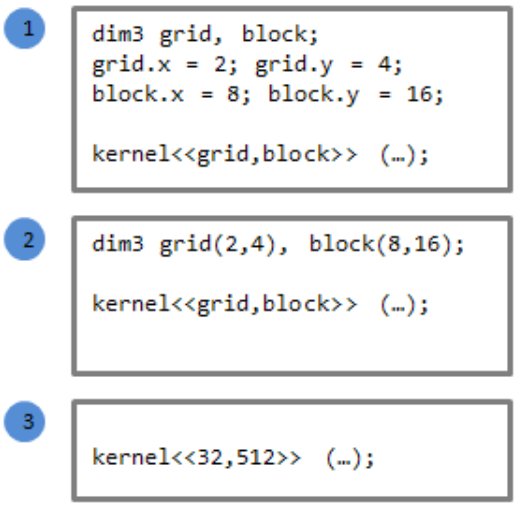


Figura 5.4: Compilando CUDA [14].

- dim3 gridDim: dimensión del *grid* en bloques (a lo sumo 2D).
- dim3 blockDim: dimensión del bloque en hilos (a lo sumo 3).
- dim3 blockIddx: Id del bloque en un *grid*.
- dim3 threadIdx: Id del hilo en el bloque.

Cada hilo tiene un ID único global que puede ser empleado como índice de arreglos de hasta tres dimensiones. Este ID único es mapeado mediante las variables antes mencionadas. En un caso sencillos de un *grid* 1D con N bloques unidimensionales, el ID global de un hilo sería blockId.x*blockDim.x + threadIdx.x. En la Figura 5.5, se muestra una comparación de un programa escrito en la CPU y un programa CUDA que son equivalentes en cuanto a resultado, visualizando como en el programa CUDA es empleado el ID global del hilo para asignar un valor a una posición específica de un arreglo. Es importante observar que en la definición del *grid* se emplea la función `ceil` para definir la dimensión de la misma, asegurando de esta forma que la cantidad de bloques en el *grid* ocupe un espacio menor o igual al espacio que dimensiona dicha malla.

En CUDA también se manejan tipos de clasificadores de funciones, que especifica si una función se ejecuta en el *host* o en el *device* y si esta función puede ser llamada desde el *host* o el *device*. Los tipos de calificadores para las funciones se muestran a continuación [14]:

- `_global_`: Permite declarar una función como un núcleo. Función que es llamada desde el *host* y ejecutada en el *device*. Debe retornar vacío.
- *_device_*: Función auxiliar para funciones que son corridas sobre *device*, y que por ende no pueden ser llamadas desde el *host*.
- `_host_`: Funciones llamadas solamente desde *host* y que son ejecutadas en el *host*.

Programa en CPU

```
void inc_cpu(int * a, int N)
{
    int idx;
    for(idx = 0; idx<N; idx++)
        a[idx]+=1;
}

void main()
{
    ...
    inc_cpu(a,N);

}
```

Programa en GPU

```
_global_ void inc_gpu(int *a_d, int N)
{
    int idx = blockIdx.x*blockDim.x
    + threadIdx.x;
    if(idx<N)
        a_d[idx] = a_d[idx] + 1;
}

void main()
{
    dim3 dimBlock(blocksize);
    dim3 dimGrid(ceil(N/(float)blocksize);
    inc_gpu<<<dimGrid,dimBlock>>>(a_d,N);
}
```

Figura 5.5: Ejemplo: Incrementar arreglo [14].

Los calificadores `_host_` y `_device_` pueden ser combinados para generar código tanto en la CPU como en la GPU.

5.3.4. Comunicación entre la CPU y la GPU

En la Figura 5.1 se observa como la CPU y la GPU tienen espacios de memoria separados y a su vez se observa que tanto la GPU como la CPU pueden acceder al espacio de memoria global DRAM del *device*, lo que permite a la CPU manejar la memoria del *device* para acceder a datos, crear o liberar espacios de la misma y escribir datos en ella. Estos accesos se realizan mediante instrucciones especificadas en el apéndice A.3.

La comunicación entre la CPU y la GPU es lo que permite la definición de los kernels desde un programa principal, debido a que los argumentos de la funciones de kernels son copiadas directamente desde el *host* al *device*, siempre y cuando los parámetros sean datos simples, como *int*, *char*, entre otros. Si se desea pasar un arreglo o tipos de datos no elementales, debe emplearse la instrucción de acceso `cudaMemCpy` especificado en B.

En CUDA se manejan tipos de clasificadores de funciones que especifican si una función es ejecutada o es llamada por el *host* o por el *device*. De la misma forma, las variables declaradas en memoria tienen asociado un tipo de clasificador que especifica el tipo de memoria en la cual se encuentran. Cabe acotar que también existen variables desclasificadas, tales como los escalares y tipos de vectores que son almacenados en los registros, y en caso de no poder ser alojadas en el registro son establecidas en la memoria local del *device*.

En este punto se sabe en concreto que esta arquitectura extiende C al permitir al programador definir funciones en dicho lenguaje denominadas "*kernels*", que cuando son llamadas son ejecutadas N veces en paralelo por N hilos de CUDA diferentes mediante la cooperación y sincronización entre los hilos, en lugar de ejecutarlo una sola vez como regularmente

lo hace C. También se conoce el manejo de memoria en este modelo y cómo los *kernels* intervienen en este manejo de memoria.

Sincronización

El `cudaMemCpy` mencionado con anterioridad y manejado desde el *host* es síncrono, por consiguiente el control retorna a la CPU luego que esta instrucción de copia ha sido completada. Esta instrucción inicia luego que todas las llamadas previas a CUDA han sido completadas.

`cudaThreadSynchronize()` actúa como una barrera que bloquea el *host* hasta que todas las llamadas previas de CUDAD hayan sido finalizadas.

En cuanto a la sincronización del *device*, los bloques de un *kernel* pueden cooperar entre ellos mediante una memoria compartida y sincronizando su ejecución para coordinar los accesos a memoria. Los puntos de sincronización se pueden especificar en el *kernel* llamando la función intrínseca `_syncthreads()`, que actúa como una barrera en la cual los hilos en el bloque deben esperar antes que proseguir.

5.3.5. Restricciones

Las restricciones y limitaciones de los hilos de CUDA se listan a continuación:

- No pueden acceder a la memoria de *host*.
- Siempre debe retornar vacío.
- El número de argumentos no puede ser variable.
- No son recursivas.
- No emplean variables estáticas.

5.4. Pruebas y Resultados

5.4.1. Experimentos (nivel 1)

En los experimentos nivel 1 en la sección 4.2.3 se realiza pruebas rigurosas con la Figura 4.4(c). A continuación se muestran los resultados de las 384 ejecuciones realizadas para la versión secuencial y la segunda tendencia de la versión paralela y los resultados de las 192 ejecuciones de la primera tendencia de la versión paralela. Para cada versión del algoritmo, los resultados se dividen en grupos de 48.

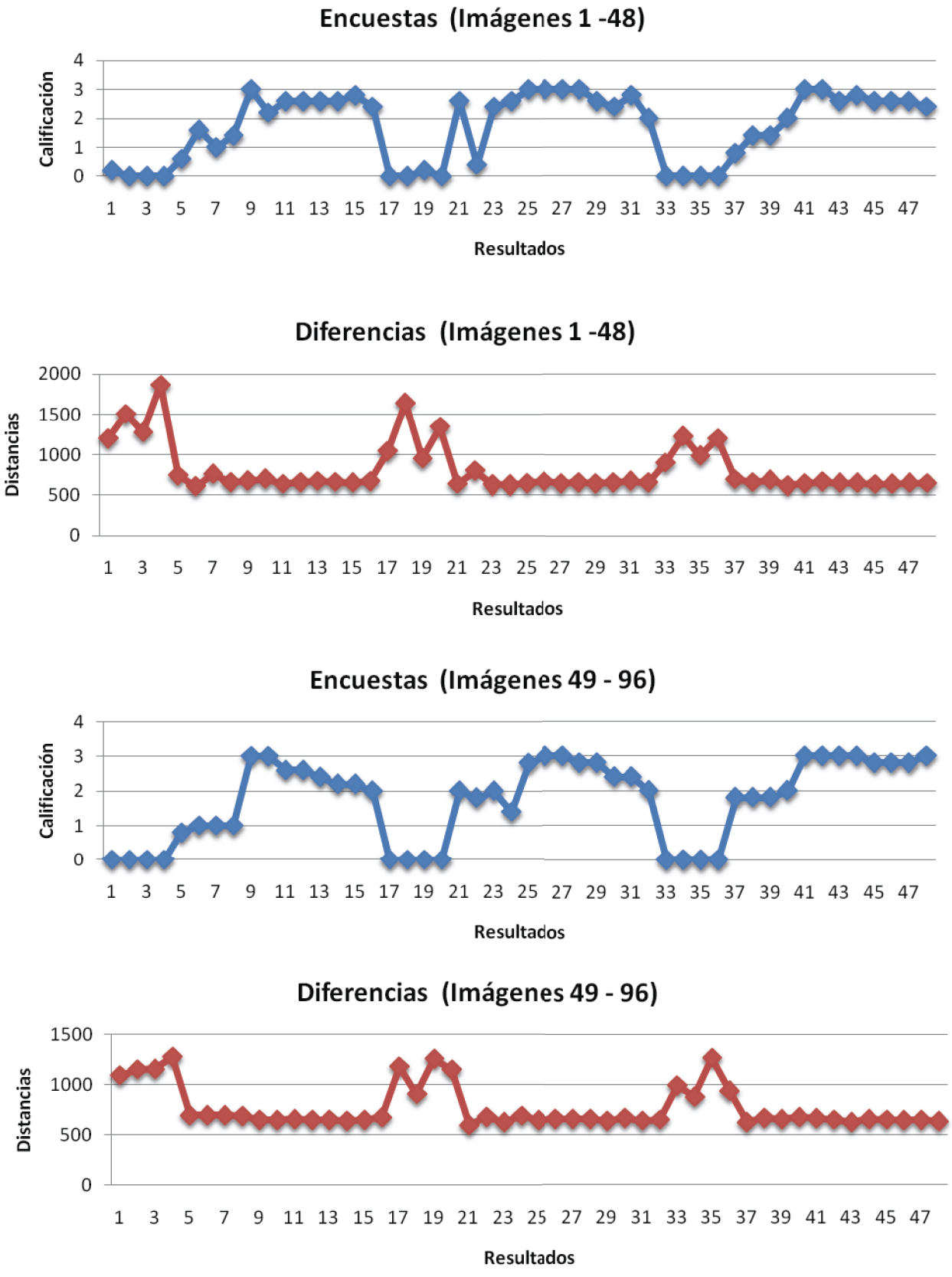


Figura 5.6: Versión secuencial: resultados 1 - 96.

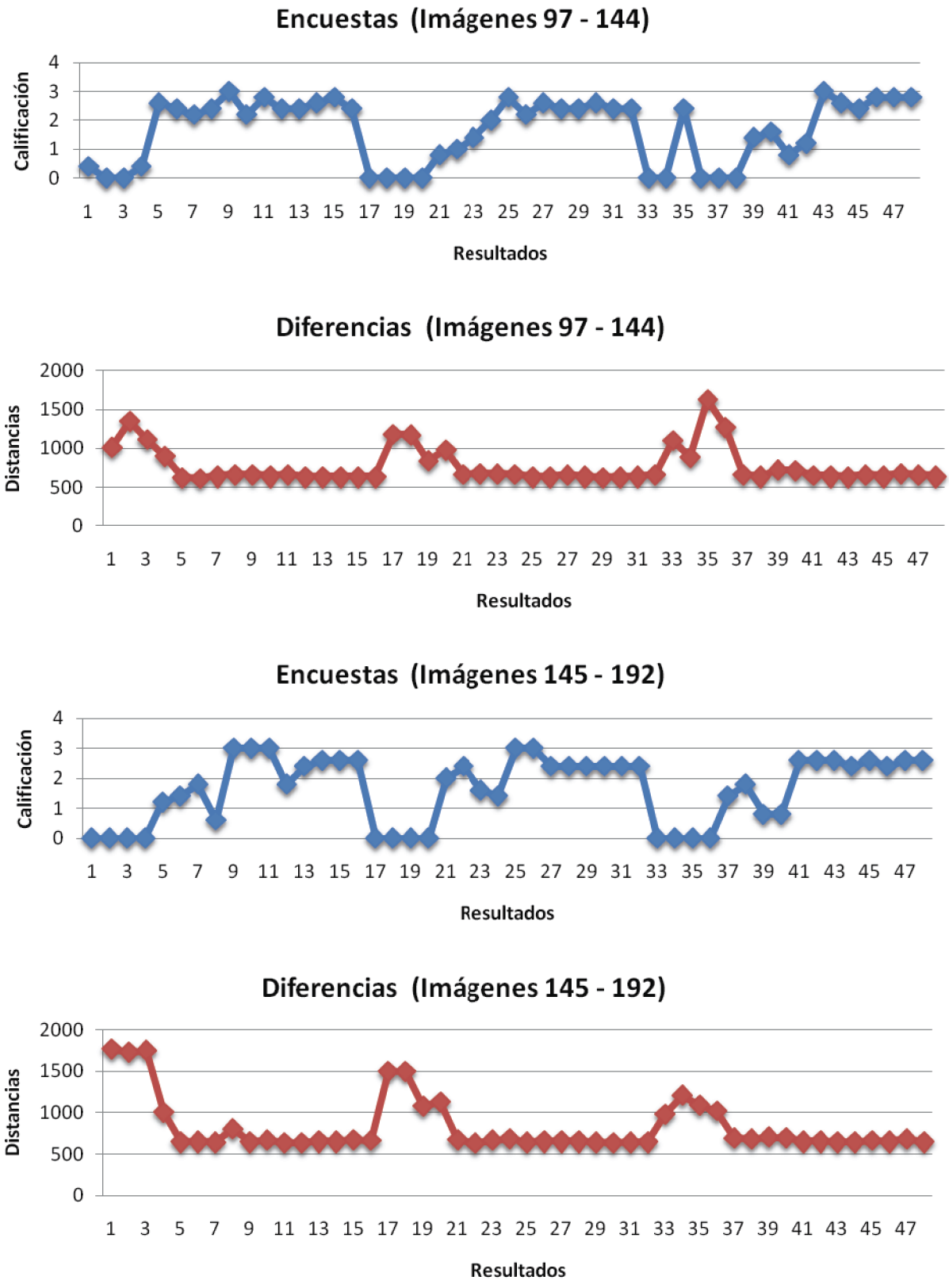


Figura 5.7: Versión secuencial: resultados 97 - 192.

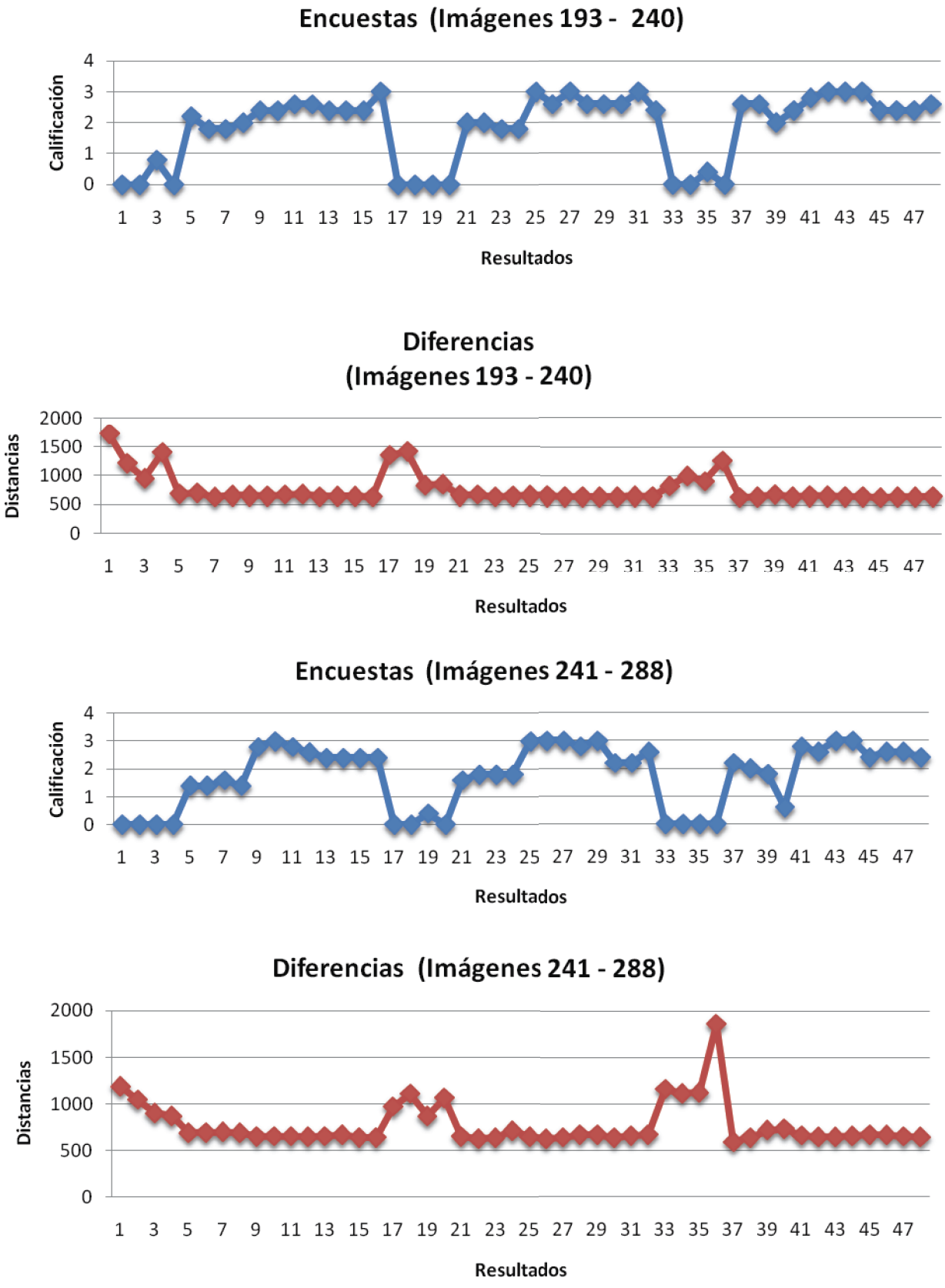


Figura 5.8: Versión secuencial: resultados 193 - 288.

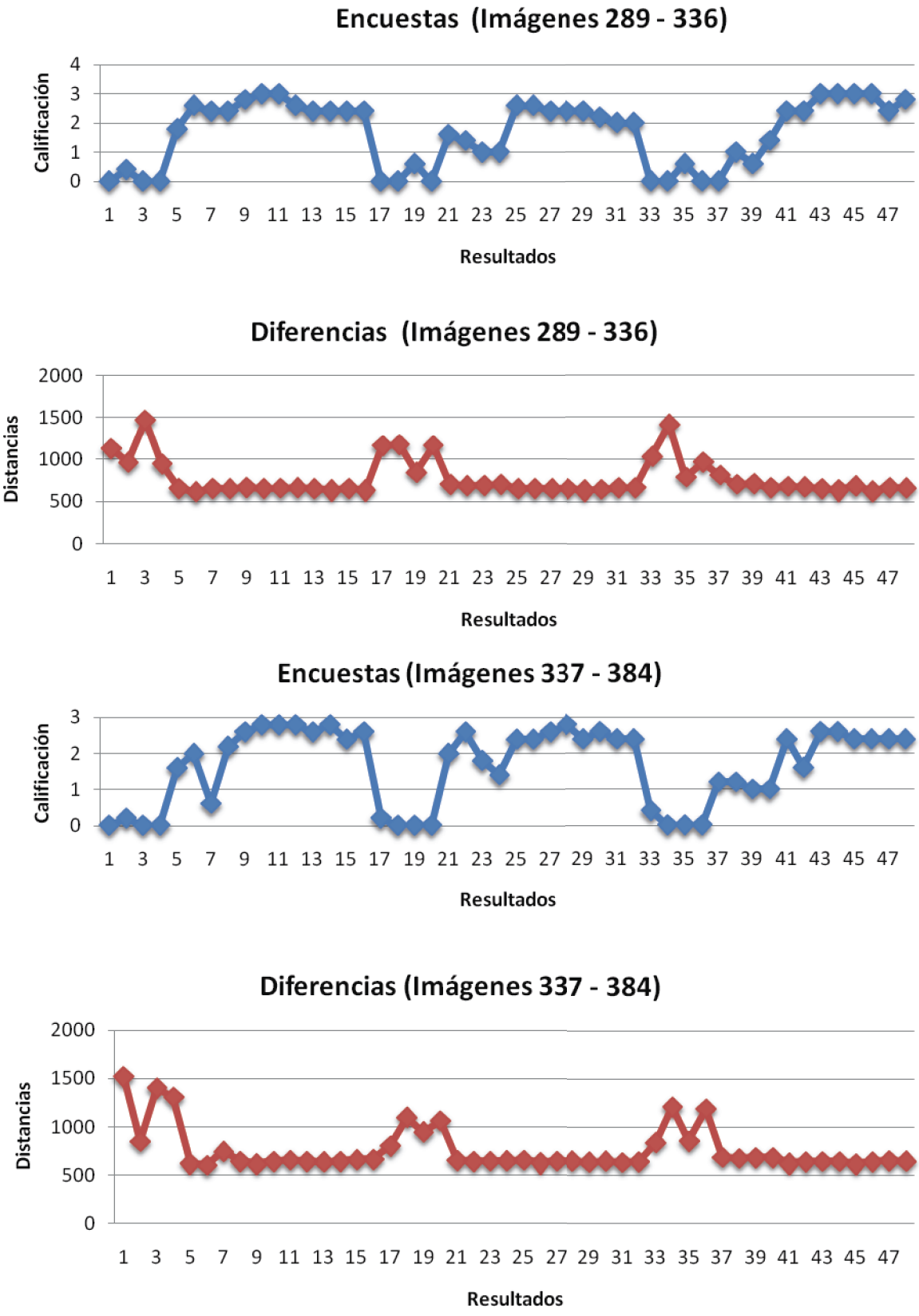


Figura 5.9: Versión secuencial: resultados 289 - 384.

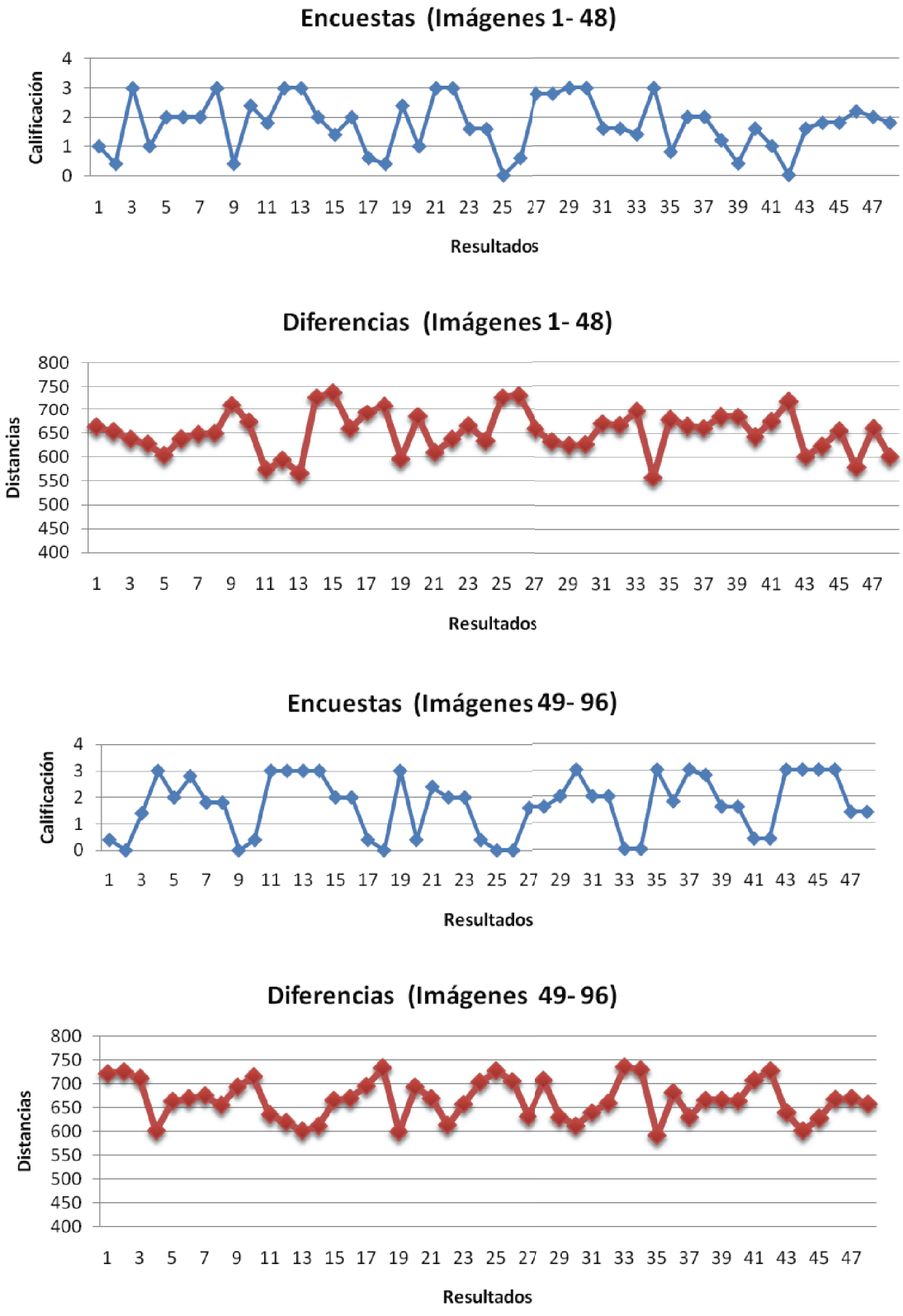


Figura 5.10: Versión paralela (*GPU T1*): resultados 1 - 96.

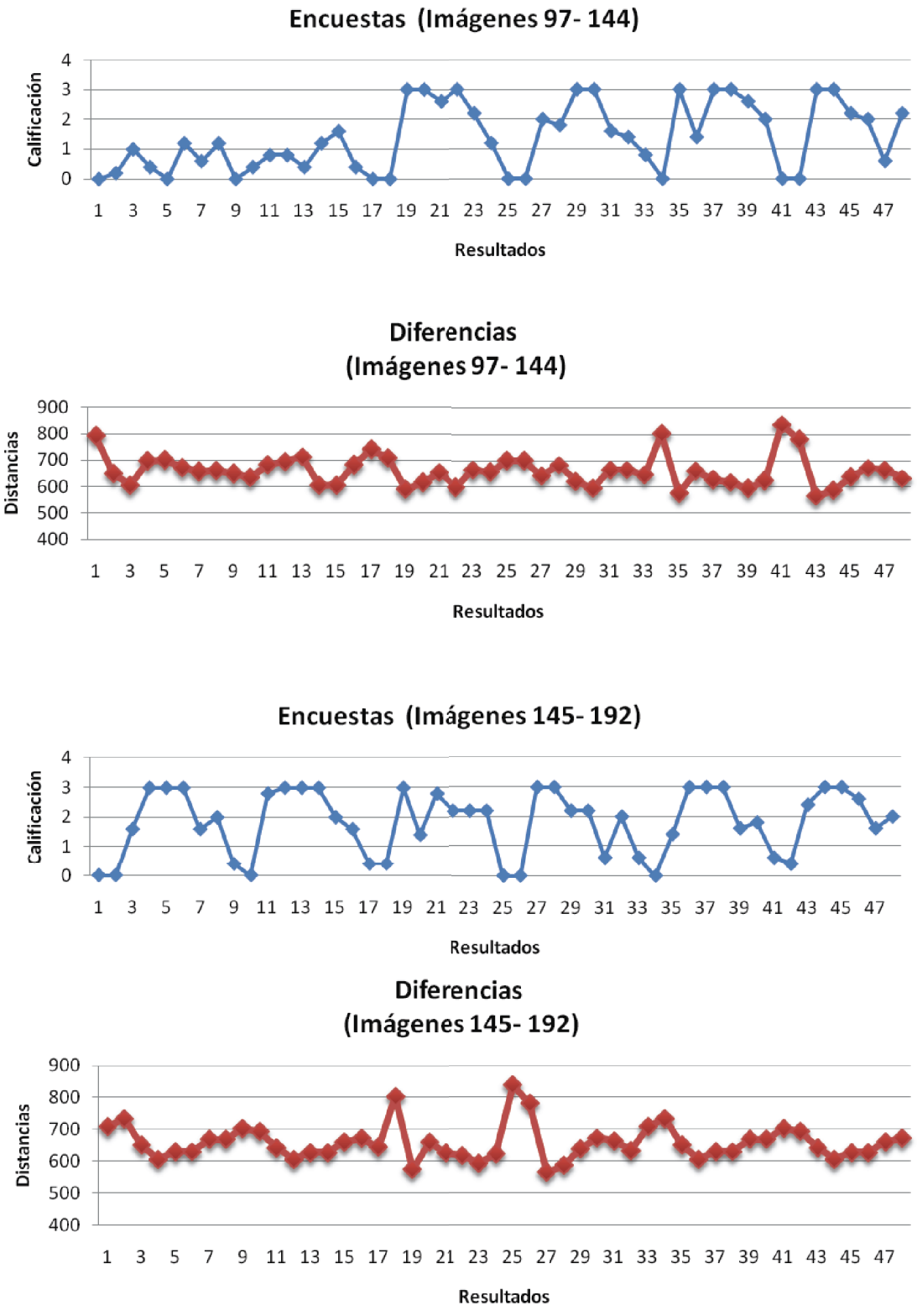


Figura 5.11: Versión paralela (*GPU T1*): resultados 97 - 192.

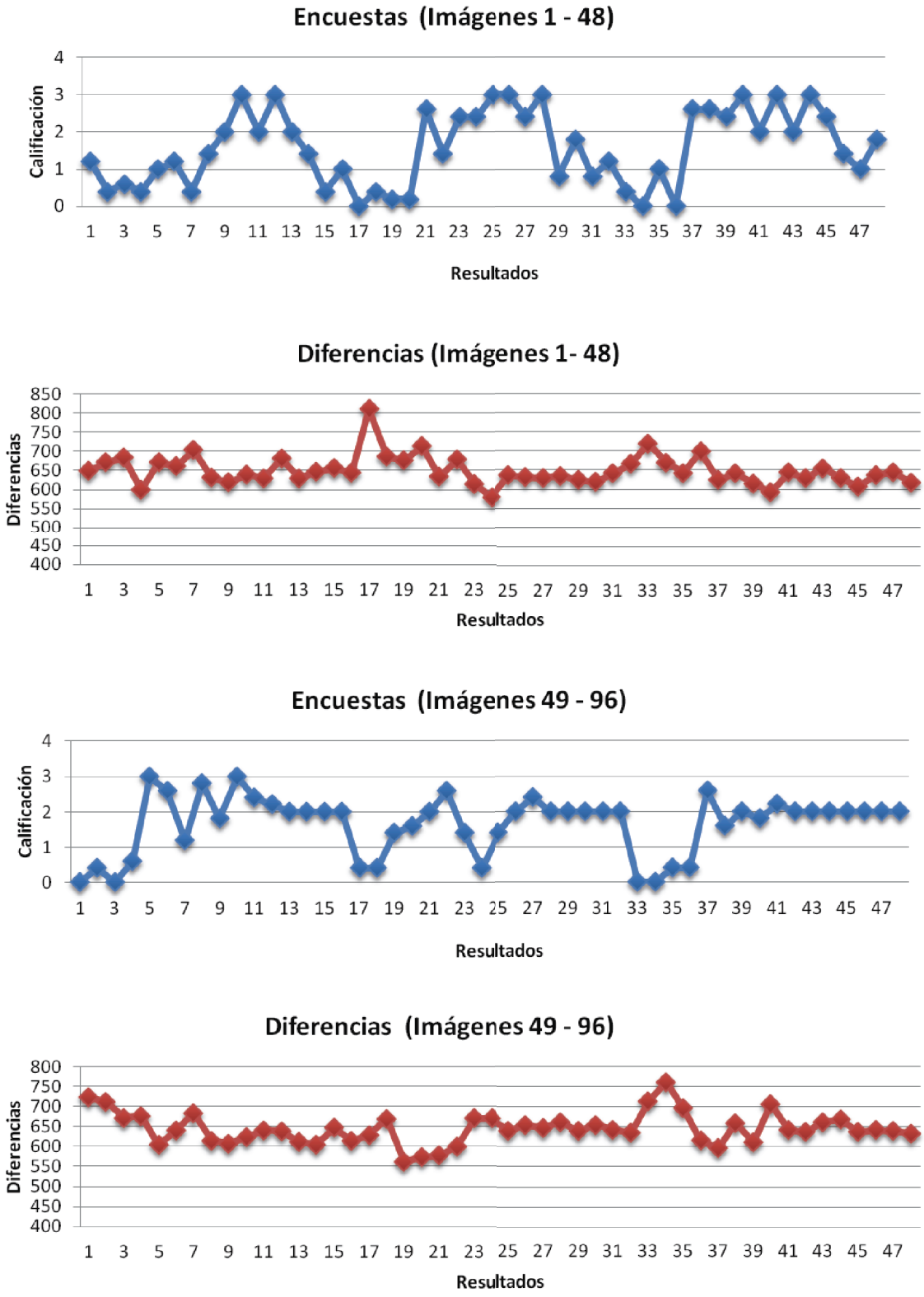


Figura 5.12: Versión paralela (*GPU T2*): resultados 1 - 96.

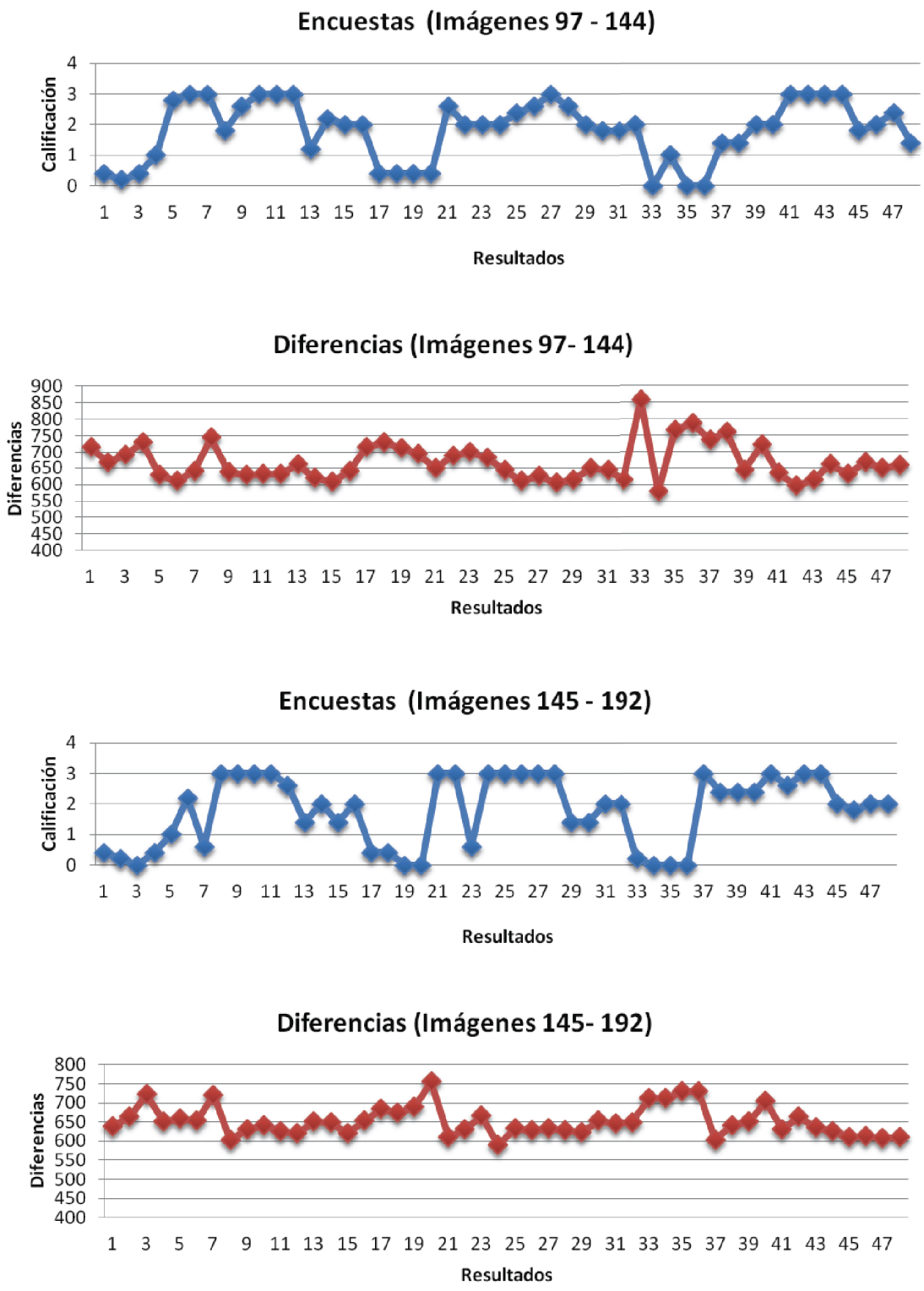


Figura 5.13: Versión paralela (*GPU T2*): resultados 97 - 192.

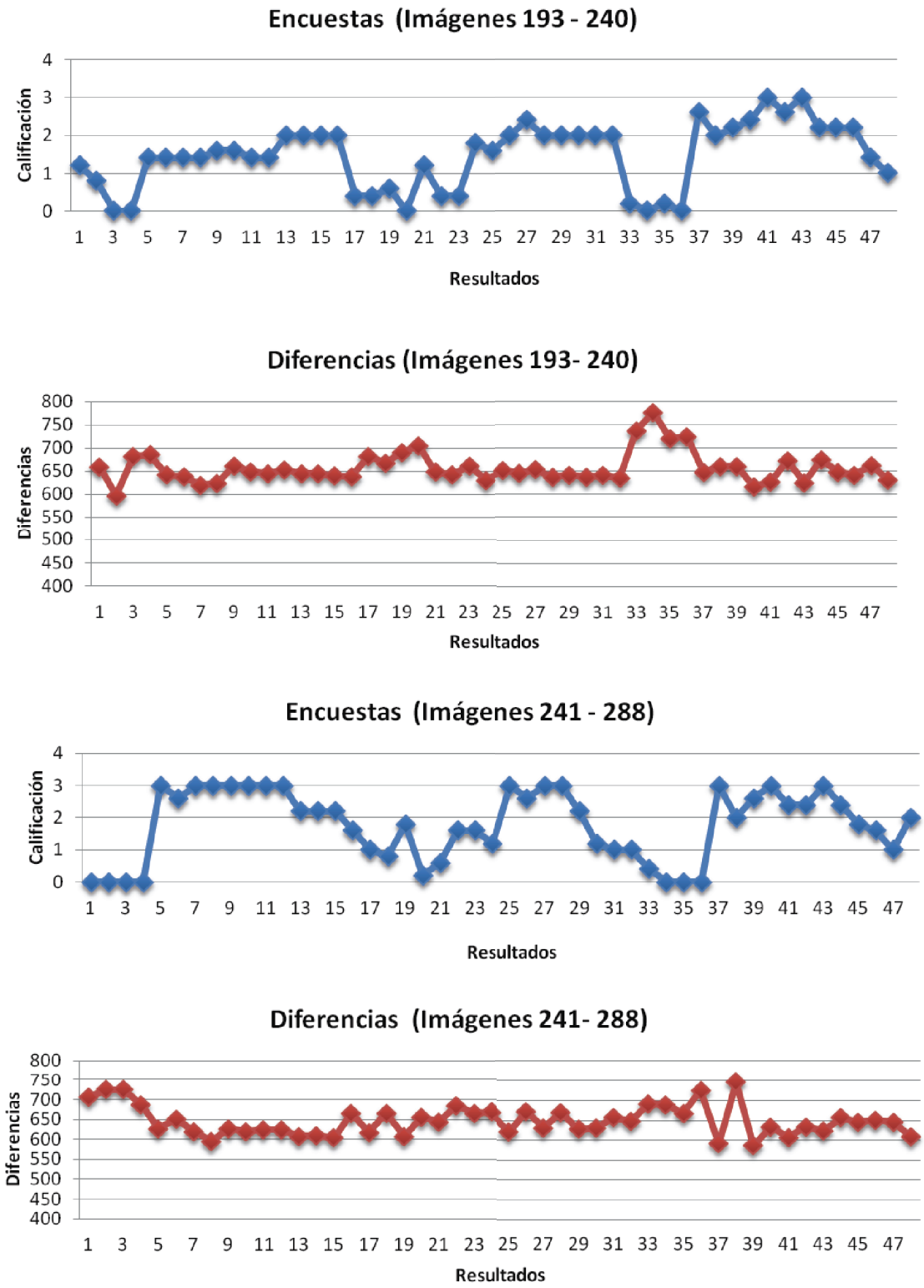


Figura 5.14: Versión paralela (*GPU T2*): resultados 193 - 288.

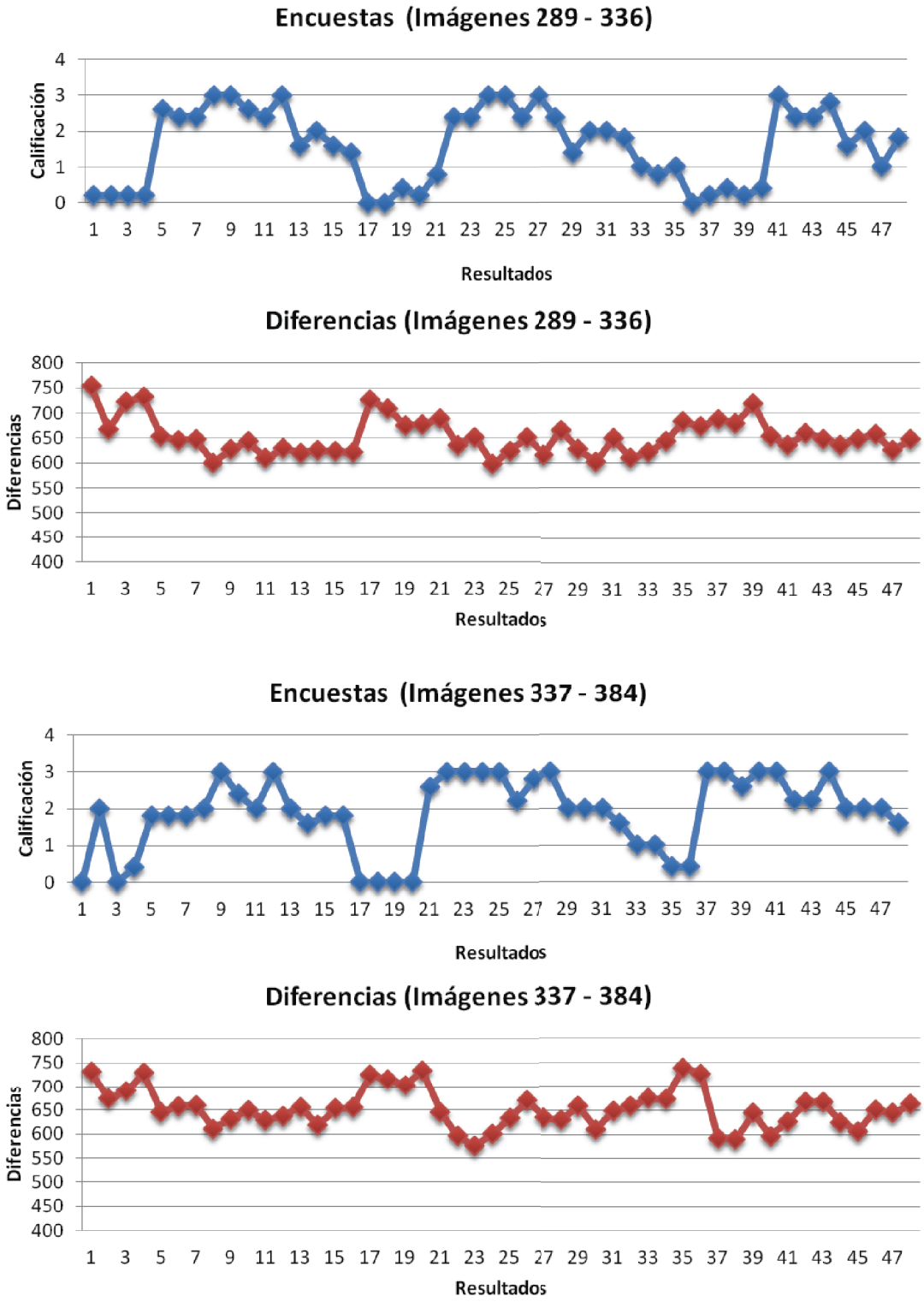


Figura 5.15: Versión paralela (*GPU T2*): resultados 289 - 384.