Universidad Autónoma Metropolitana
Unidad Azcapotzalco
División de Ciencias Básicas e Ingenieráa
Licenciatura en Ingenierá en Computación

Aplicación de la transformada de wavelet discreta de la familia Haar para la fusión de imágenes con cómputo acelerado por GPU

Modalidad: Proyecto Tecnológico

Trimestre 2018 Otoño

Alumno:
Manuel Fernando Maldonado Ramírez
2143033285

Asesores:
M. en C. Oscar Alvarado Nava
M. en C. Hilda María Chablé Martínez

14 de diciembre de 2018

Declaratoria

Yo, Oscar Alvarado Nava, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Yo, Hilda María Chablé Martínez, declaro que aprobé el contenido del presente Reporte de Proyecto de Integración y doy mi autorización para su publicación en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Yo, Manuel Fernando Maldonado Ramírez, doy mi autorización a la Coordinación de Servicios de Información de la Universidad Autónoma Metropolitana, Unidad Azcapotzalco, para publicar el presente documento en la Biblioteca Digital, así como en el Repositorio Institucional de UAM Azcapotzalco.

Resumen

En el presente trabajo se buscó crear un asistente de conducción nocturna, el cual mediante la implementación de la transformada wavelet discreta de Haar se fusionó una imagen convencional con una imagen infrarroja, obteniendo así una imagen con más información visible para el conductor. Debido a que a la hora de conducir se espera que el asistente muestre la imagen fusionada en tiempo real, se realizó una paralelización del programa secuencial del asistente sobre una GPU.

Las imágenes utilizadas para la fusión son imágenes en formato tiff con un alto y un ancho de 512, 1024,2048,4096 y 8192 pixeles, lo cual nos permite obtener una comparación de tiempos entre la implementación secuencial y la implementación paralela del asistente de conducción nocturna.

Tabla de contenido

1. Introducción

1.1. Introducción

Conducir un automóvil durante la noche puede ser de alto riesgo debido a la falta de visibilidad. La información que proporcionan los sensores de luz visible, como cámaras de video y ojos humanos, está limitada por la iluminación provista por los faros del vehículo y por la luz ambiental disponible. Obstáculos como peatones y animales pueden ser difíciles de detectar con la luz de un vehículo, pero pueden ser bastante fáciles de localizar usando una cámara infrarroja. Sin embargo, una imagen infrarroja no proporciona toda la información necesaria para operar un vehículo por la noche. Características como el color de la luz de la calle faltan por completo en las imágenes de infrarrojos y las características como las marcas de carril pueden ser difíciles de identificar en las imágenes de infrarrojos. A partir de estos hechos, parece que una fusión de las imágenes visibles e infrarrojas de una escena sería útil en una situación de conducción nocturna[1].

Con ayuda de la transformada discreta de *wavelet* de la familia Haar (DHWT por sus siglas en inglés) se realizará la fusión de la imagen infrarroja Figura 1 y la imagen convencional Figura 2 para así obtener una imagen con mayor información para la asistencia del conductor. Antes de fusionar las imágenes fuente, primero se descompondrán en sus canales RGB de tal forma que ambas imágenes queden en la misma escala de colores, una vez que ambas imágenes se encuentren en la misma escala de colores se descompondrán por medio de la DHWT en regiones de frecuencias altas y bajas [1]. Asimismo, se aplicarán reglas de selección, las cuales serán las encargadas de elegir los coeficientes de *wavelet* que conformarán la imagen combinada. Finalmente, una nueva imagen es creada realizando la transformada de *wavelet* inversa.

Este trabajo estará dividido en dos partes, en la primera se realizará la fase de fusión en forma secuencial; en la segunda parte, se buscará acelerar el proceso de fusión de imágenes paralelizando el programa secuencial y ejecutándolo sobre una Unidad de Procesamiento Gráfico (GPU), a este proceso se le conoce como cómputo acelerado por GPU, que no es más que el uso de una GPU en combinación con una CPU para acelerar aplicaciones de *Deep learning*, análisis e ingeniería. Esta paralelización es realizada para que la obtención de la imagen fusionada no tome mucho tiempo, pues se espera que este trabajo en un futuro pueda ser aplicado a la vida real, en donde el tiempo se vuelve en un factor importante.

Figura 1: Imagen Infrarroja obtenida de CVC-14:Dataset de secuencia peatonal de dá y noche de FIR visible

Figura 2: Imagen Convencional obtenida de CVC-14:Dataset de secuencia peatonal de dá y noche de FIR visible

1.2. Antecedentes

Como antecedentes al trabajo aquí presentado, el 21 de septiembre del 2016 se presentó un proyecto en la Universidad Autónoma Metropolitana llamado "Paralelización de Transformada Wavelet en GPUs" [5], en el cual se buscó acelerar el proceso del cálculo de la transformada de wavelet de la familia Haar, para ello se paralelizó un programa secuencial encargado de obtener la transformada de wavelet. La paralelización se hizo de dos formas, en la primera se usó una interfaz de programación de aplicaciones para la programación multiproceso de memoria compartida llamada OpenMP, la segunda forma fue hecha sobre la plataforma de computación en paralelo llamada CUDA, la cual permite realizar programas para en GPU de NVIDIA. Dicho proyecto fue fundamental para comprender y facilitar el entendimiento de todo el proceso que conlleva realizar la paralelización de la transformada de wavelet.

Por otro lado, el artículo titulado "Un análisis comparativo de fusión visual y térmica de imagen facial basada en diferentes familias de wavelets"[6], nos ayudó a comprender todo el proceso necesario para realizar la fusión de imágenes, ya que, en dicho artículo se realiza la fusión de imágenes infrarroja y visible aplicando diferentes familias de la transformada de wavelt, y así poder obtener una comparativa sobre cuál es el mejor método.

1.3. Justificación

El conducir un automóvil implica una responsabilidad muy grande para aquel que se encuentre detrás del volante, ya que sus acciones pueden llegar a afectar a terceros. Sin embargo, hay ocasiones en las que el ambiente o factores que rodean al conductor no son los más favorables. Un caso en el cual la acción de conducir se vuelve hasta cierto punto complicada, es cuando se maneja durante la noche, ya que algunas veces la iluminación de las calles no es suficiente o los faros del automóvil no iluminan el camino adecuadamente.

El no percatarse a tiempo de un peatón o animal que pudiera atravesársele al conductor podría dar origen a un accidente, por lo tanto, este trabajo pretende ofrecer un asistente de conducción nocturna que ayude a disminuir la posibilidad de tener un accidente al conducir en condiciones de poca luz, haciendo visible aquello que la persona que está conduciendo no es capaz de ver con claridad. Debido a lo mencionado anteriormente, es importante que el proceso de generación de la imagen fusionada sea acelerado y paralelizado en una unidad de procesamiento gráfico, de tal forma que dicha imagen sea obtenida en tiempo real.

1.4. Objetivo general

Implementar la transformada de *wavelet* discreta de la familia Haar en el proceso de fusión de imágenes con cómputo acelerado por GPU.

1.5. Objetivos específicos

- Diseñar e implementar un módulo de generación de coeficientes de *wavelet*.
- Diseñar e implementar un módulo de fusión de *wavelets*.
- Diseñar e implementar un módulo de construccińn de la imagen fusionada por medio de la transformada inversa de *wavelet*.

2. Marco teórico

2.1. Transformada *Wavelet*

Una vez dada la introducción y justificación del proyecto aquí realizado, se dará un breve recorrido por los temas más importantes, los mismos que nos darán las bases para comprender el comportamiento y el proceso de la fusión de imágenes con cómputo acelerado por GPU aplicando la transformada *wavelet* discreta de la familia Haar. En esta sección se hablará sobre los temas relacionados con la Transformada *Wavelet*, dando una breve introducción acerca de ella y finalmente explicando su cálculo tanto para datos unidimensionales como bidimencionales.

2.1.1. Introducción a la Transformada *Wavelet* Discreta

Antes de adentrarnos al mundo de la Transformada *Wavelet* Discreta de la familia Haar (DHWT) tenemos que conocer el comportamiento y algunas de las características importantes de la Transformada *Wavelet* Discreta (nos referiremos a la Transformada *Wavelet* Discreta como DWT) y el porque se ha escogido este tipo de transformada para la implementación del asistente de conducción nocturna.

La idea principal de la Transformada *Wavelet* es la descomposición multiresolución de señales e imágenes lo cual es ventajoso ya que objetos pequeños necesitan resoluciones altas, mientras que los objetos grandes necesitan bajas resoluciones. A su vez, el enfoque general de la descomposición multiresolución es el crear un componente de aproximación usando una función de escalamiento (filtro de paso bajo) y un componente de detalle usando funciones *wavelets* (filtro de paso alto).

Adicionalmente, la transformada *wavelet* es una herramienta que nos permite convertir una señal que se encuentra en el dominio del tiempo a una señal en el dominio de tiempo-frecuencia y regresarla a su dominio original. Esto nos permite localizar el momento en el que ocurre cierto evento en el dominio del tiempo, es decir, la correlación entre una señal dada y una señal base se puede determinar en varios instantes de tiempo. Además, el espectro de la señal está dividida en partes desiguales, lo cual se adapta al análisis de la señal ya que un ancho de banda más largo e intervalos de tiempo más cortos son apropiados para detectar un componente de alta frecuencia, mientras que un ancho de banda corto e intervalos de tiempo largos son adecuados para localizar un componente de baja frecuencia. Estas son las características principales del análisis de señales de la transformación *wavelet*, y las ventajas de la DWT en el análisis de la señal se derivan de estas características[7].

Una vez aclarado el comportamiento y las características de la DWT es fácil darnos

cuenta de que debido a su naturaleza, la DWT se adecua perfectamente al propósito del presente proyecto.

2.1.2. Transformada Wavelet Discreta de Haar

Tras haber introducido la Transformada *Wavelet* Discreta en la sección anterior, comenzaremos ahora por definir la Transformas *Wavelet* Discreta de Haar, ya que, durante el desarrollo del proyecto utilizaremos esta transformada para la descomposición de imágenes.

Debido a que la DHWT trabaja con señales discretas, empezaremos por definir que es una señal discreta. Entonces diremos que una señal discreta es una función en el tiempo con valores que ocurren en un instante de tiempo finito, expresaremos este tipo de señal de la siguiente forma:

$$f = (f_1, f_2, ..., f_N)$$

Donde N es un número entero par positivo el cual representa el tamaño o longitud de $\boldsymbol{f}$ y $f_1, f_2, ..., f_N$ son los N números reales de la señal discreta $\boldsymbol{f}$.

Al igual que las demás transformadas *wavelet*, la transformada de Haar descompone una señal discreta en dos subseñales de la mitad del tamaño que la señal original. La primera de estas es un promedio o tendencia; la segunda por su parte es una diferencia o fluctuación.[8]

La subseñal de tendencia $a = (a_1, a_2, ..., a_{N/2})$ de $\boldsymbol{f}$ está compuesta por un conjunto de promedios, los cuales son calculados de la siguiente manera: su primer valor, a_1, es calculado tomando el promedio del primer par de valores de $\boldsymbol{f}$, y después multiplicando dicho promedio por $\sqrt{2}$. El valor a_2 es el resultado de tomar el promedio del siguiente par de valores de $\boldsymbol{f}$ y de igual forma, multiplicando el resultado por $\sqrt{2}$. En otras palabras, los valores de $\boldsymbol{a}$ son obtenidos tomando el promedio de pares sucesivos de valores de $\boldsymbol{f}$, y después multiplicando esos promedios por $\sqrt{2}$. La fórmula para obtener los valores de $\boldsymbol{a}$ es:

$$a_m = \frac{f_{2m-1}+f_{2m}}{\sqrt{2}}$$

Para $m = 1, 2, 3, ..., N/2$.

Para que quede claro el proceso de calcular la subseñal de tendencia daremos un ejemplo. Supongamos que $\boldsymbol{f}$ está compuesta por los siguientes 6 valores:

$$f = (10, 9, 6, 2, 1, 9)$$

Entonces, el promedio del primer par de valores es 8, el del segundo es 4 y el del último par es 5. Multiplicando esos promedios por $\sqrt{2}$ obtenemos que la subsñeal de tendencia es:

$$a = (8\sqrt{2}, 4\sqrt{2}, 5\sqrt{2})$$

En contraste, la subseñal de fluctuación de $\boldsymbol{f}$, denotada por $d = (d_1, d_2, ..., d_{N/2})$, está formada por una serie de restas o diferencias [8]. El primer valor de la señal de fluctuación, d_1, es obtenido tomando el promedio de la diferencia entre el primer par de valores de $\boldsymbol{f}$, y multiplicando ese promedio resultante por $\sqrt{2}$. Del mismo modo es obtenido el valor d_2, tomamos el promedio de la diferencia entre el segundo par de valores de $\boldsymbol{f}$, y multiplicamos el resultado por $\sqrt{2}$. Siguiendo este camino, obtenemos que los valores de $\boldsymbol{d}$ son el resultado de calcular la siguiente fórmula:

$$d_m = \frac{f_{2m-1} - f_{2m}}{\sqrt{2}}$$

Para $m = 1, 2, 3, ..., N/2$.

Por ejemplo, para la señal considerada anteriormente:

$$f = (10, 9, 6, 2, 1, 9)$$

La señal de fluctuación de $\boldsymbol{f}$ sería:

$$d = (0{,}5\sqrt{2}, 2\sqrt{2}, -4\sqrt{2})$$

2.1.3. Transformada Wavelet Discreta de Haar bidimensional

Una vez comprendida la DHWT y debido a que el objetivo del presente trabajo es aplicar la Transformada *Wavelet* de Haar a imágenes, es necesario saber como aplicar esta transformada a datos bidimensionales. Por tal motivo en esta sección abarcaremos dicho tema. Por simplicidad nos referiremos a la transformada unidimensional como "transformada 1D" y a la bidimensional como "transformada 2D".

La Transformada *Wavelet* bidimensional puede ser aplicada a una imagen, siempre y cuando el número de filas y columnas de la imagen sea par. Para obtener la Transformada bidimensional de una imagen $\boldsymbol{f}$, es necesario seguir los siguientes pasos:

1. Primero se debe realizar la transformada de *wavelet* 1D en cada fila de $\boldsymbol{f}$, derivando así una nueva imagen.

2. Sobre la imagen generada en el paso anterior aplicamos la transformada 1D pero ahora en cada una de las columnas.

Al aplicar los pasos anteriores obtenemos la Transformada *Wavelet* de una imagen, la cual podemos simbolizarla como se muestra en la Figura 3.

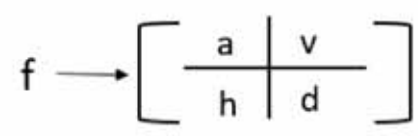

Figura 3: Representación de la Transformada *Wavelet* 2D de una imagen.

Donde $\boldsymbol{h}$,$\boldsymbol{d}$,$\boldsymbol{a}$ y $\boldsymbol{v}$ son subimagenes con M/2 filas y N/2 columnas. A continuación se explicará la naturaleza de dichas subimagenes.

La subimagen $\boldsymbol{a}$ es el resultado de obtener la tendencia a lo largo de las filas y columnas, dando como resultado una versión de menor resolución que la imagen original $\boldsymbol{f}$.

La subimagen $\boldsymbol{h}$ es creada calculando la tendencia a lo largo de las filas y la fluctuación a lo largo de las columnas. Como consecuencia, en cualquier parte dentro de una imagen en dónde haya bordes horizontales, las fluctuaciones detectarán dichos bordes. Esta subimagen tiende a enfatizar los bordes horizontales de una imagen, llamaremos a esta subimagen como fluctuación horizontal.

La subimagen $\boldsymbol{v}$ es similar a la $\boldsymbol{h}$, excepto que en la subimagen $\boldsymbol{v}$ la fluctuación es calculada a lo largo de las filas y la tendencia a lo largo de las columnas, obteniendo así una subimagen que resalta más los bordes verticales dentro de la imagen original, llamaremos a esta subimagen como fluctuación vertical.

Por último, hablaremos de la fluctuación diagonal, $\boldsymbol{d}$. Esta subimagen a diferencia de las demás, tiende a enfatizar las características diagonales de la imagen original, ya que es creada calculando la fluctuación de las filas y columnas de la imagen fuente.

2.2. Cómputo acelerado por GPU

Para finalizar la sección del marco teórico, en este apartado abordaremos el tema de las Unidades Gráficas de Procesamiento, comenzando con una pequeña reseña histórica de su evolución y el cómo ha llegado para quedarse el cómputo acelerado por Unidades Gráficas de Procesamiento. Finalizaremos el tema hablando sobre la Arquitectura CUDA y de sus jerarquías de memoria.

2.2.1. La era del procesamiento paralelo

Hace 30 años, uno de los métodos más importantes para mejorar el rendimiento de los procesadores era aumentar la velocidad en la que operaba el ciclo de reloj, sin embargo, debido a las restricciones de energía y calor de los circuitos integrados, así como a un rápido aproximamiento al limite físico del tamaño del transistor, las industrias e investigadores se vieron forzados a buscar nuevas alternativas de mejorar el poder de cómputo. Fue así como los investigadores e industrias del mercado voltearon a ver al mundo de las supercomputadoras. Las cuales en adición a este aumento astronómico de la velocidad del procesador, dieron un salto masivo en el rendimiento de cómputo al aumentar constantemente el número de procesadores. Esta tendencia, que a veces se conoce como la revolución multinúcleo, ha marcado un gran cambio en la evolución del mercado de la computadora personal.

Mientras todo este cambio revolucionario de los procesadores sucedía, el procesamiento gráfico sufrió una evolución dramática. Pues la compañía de tecnología NVIDIA lanzó la primer GPU en la que los cálculos de transformación e iluminación se podían realizar directamente en la unidad de procesamiento gráfico, lo cual, desde un punto de vista de cómputo paralelo representa posiblemente el avance más importante en la tecnología de GPU, ya que, fue la primer GPU que implementaba el entonces nuevo estándar DirectX 8.0 de Microsoft. Este estándar requería que el hardware compatible contuviera tanto vértices programables como sombreado de píxeles programables. Por primera vez, los desarrolladores tenían cierto control sobre los cálculos que se realizaban en sus GPU.

Esencialmente, las GPU de principios de la década del 2000 se diseñaron para producir un color para cada píxel en la pantalla utilizando unidades aritméticas programables conocidas como sombreadores de píxeles. En general, un sombreador de píxeles utiliza su posición (x, y) en la pantalla, así como información adicional para combinar varias entradas en el cálculo del color final. Debido a que la aritmética que se estaba realizando en los colores y texturas de entrada estaba completamente controlada por el programador, los investigadores observaron que estos "colores" de entrada podían ser realmente cualquier información. Entonces, si las entradas fueran en realidad datos numéricos que significan algo distinto al color, los desarrolladores podrían programar los sombreadores de píxeles para realizar cálculos arbitrarios en estos datos. Los resultados serían devueltos a la GPU como el "color" final del píxel, aunque los colores serían simplemente el resultado de cualquier cálculo que el programador le haya ordenado a la GPU que realizar. Este truco era muy inteligente pero también muy complicado, pues el modelo de programación era demasiado restrictivo y había serias limitaciones sobre cómo y dónde el programador podía escribir resultados en memoria. Además, era casi imposible predecir cómo una GPU en particular manejaría los datos de punto flotante, si es que manejaba datos de punto flotante. Finalmente, cuando el programa inevitablemente calculaba resultados incorrectos, no finalizaba o simplemente la máquina se quedaba ciclada, no existía un método razonablemente bueno para depurar el código que se estuviera ejecutando en la GPU.

Como si las limitaciones no fueran lo suficientemente graves, cualquiera que quisiera usar una GPU para realizar cómputo de propósito general necesitaba aprender OpenGL o DirectX, ya que eran el único medio por el cual se podía interactuar con una GPU. Esto no sólo significaba almacenar datos en texturas gráficas y ejecutar cálculos llamando a las funciones OpenGL o DirectX, sino que también significaba escribir los cálculos en lenguajes de programación especiales para gráficos. El hecho de pedirles a los investigadores que enfrentaran todas estas limitaciones antes de intentar aprovechar el poder de cómputo de su GPU, resultó ser un gran obstáculo.

No sería hasta noviembre del 2006, que NVIDIA presentó la primera GPU DirectX 10 de la industria, la GeForce 8800 GTX. La GeForce 8800 GTX fue la primera GPU que se construyó con la arquitectura CUDA de NVIDIA. Esta arquitectura incluía varios componentes nuevos diseñados estrictamente para el cómputo sobre GPU y destinados a

aliviar muchas de las limitaciones que impedían a sus antecesoras ser útiles para el cálculo de propósito general [10].

2.2.2. Arquitectura CUDA

CUDA es una arquitectura de coprocesamiento de hardware y software para cómputo paralelo que permite a las GPU de NVIDIA ejecutar código en C, C++, Fortran, OpenCL y otros lenguajes de programación. Debido a que la mayoría de los lenguajes fueron diseñados para un hilo secuencial, CUDA conserva este modelo y lo extiende con un conjunto minimalista de abstracciones para expresar el paralelismo. Además, debido a su diseño, CUDA permite el desarrollo de programas paralelos altamente escalables que puedan correr en decenas de miles hilos concurrentes y cientos de núcleos del procesador.

Un programa CUDA reside en un programa host, que consta de uno o más subprocesos secuenciales que se ejecutan en la CPU del host y uno o más núcleos paralelos adecuados para su ejecución en una GPU. Para ejecutar un programa CUDA, primero debemos realizar una llama al kernel, el cual se encargar de ejecutar un programa secuencial en un conjunto de hilos paralelos ligeros, estos hilos paralelos ejecutados en el kernel son organizados por el programador o compilador, de tal forma que cada hilo pertenezca a un Grid de bloques de hilos, esta jerarquía de entidades existente en un programa CUDA es mostrado en la Figura 4.

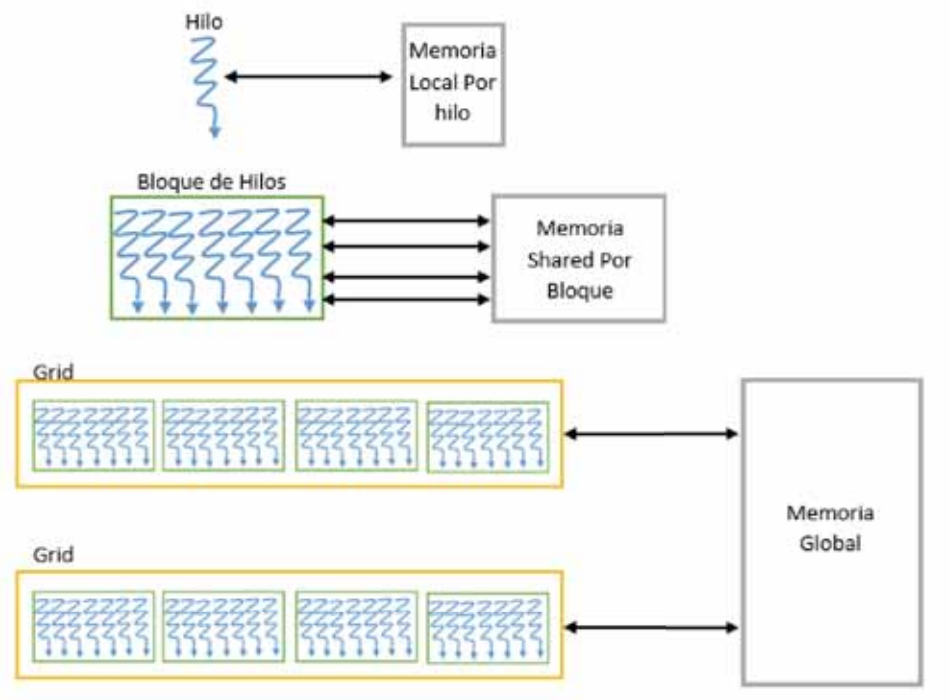

Figura 4: Representación jerárquica de memoria y entidades de CUDA.

Como se puede apreciar en la Figura 4, los hilos de diferentes bloques del mismo Grid pueden coordinarse mediante operaciones atómicas en un espacio de memoria compartido por todos los hilos, comúnmente llamada memoria Shared. Por su parte, los Grid del kernel dependientes secuencialmente se pueden sincronizar a través de barreras globales y coordinarse por medio de la memoria global. CUDA requiere que los bloques de subprocesos sean independientes, lo que proporciona escalabilidad a las GPU con diferentes números de núcleos y subprocesos de procesadores [11]. Una vez que hemos entendido lo mencionado anteriormente, ha llegado la hora de explicar algunas de las etiquetas, funciones y tipos de datos CUDA que estaremos utilizando en el presente proyecto.

- *__global__*: Este es un calificador o etiqueta que alerta al compilador de que una función debe ser ejecutada en la GPU en lugar del host.
- *CudaMalloc(void ** devPtr, size_t size)*: se comporta de forma similar al estándar C llamado Malloc, pero este le dice al tiempo de ejecución de CUDA que debe reservar memoria en el dispositivo. El primer argumento es un apuntador al apuntador de la dirección de memoria recién asignada, mientras que el segundo parámetro es el espacio de memoria a reservar.
- *CudaFree(void * devPtr)*: nos permite liberar el espacio memoria apuntada por devPtr.
- *CudaMemcpy(void * dst, void * src, size_t n,cudaMemcpyKind tipo)*: Copia *n* bytes del área de memoria apuntada por *src* al área de memoria apuntada por *dst*, donde *tipo* puede ser *cudaMemcpyHostToHost* , *cudaMemcpyHostToDevice* , *cudaMemcpyDeviceToHost* , o *cudaMemcpyDeviceToDevice* , y especifica la dirección en la que se copiaran los datos.
- *ThreadIdx*: contiene el índice del hilo que está corriendo actualmente.
- *BlockIdx*: contiene el índice del bloque que está corriendo actualmente.
- *BlockDIm*: es una constante para todos los bloques y almacena el número de hilos en cada dimensión del bloque.
- *GridDim*: contiene las dimensiones con las que se lanzó el Grid y es el mismo para todos los bloques.
- *dim3(int p1, int p2 , int p3)*: es una tupla tridimensional que nos permite especificar el número de bloques o hilos lanzados en tres dimensiones, sin embargo, actualmente CUDA no soportar Grids de tres dimensiones, por lo que sólo definiremos los primeros dos parámetros. CUDA en tiempo de ejecución llenará la tercera dimensión con el valor de 1.

Por último, para realizar una llamada al kernel debemos realizar lo siguiente:

nombreFuncion«bloques, hilos»(parámetros);

Donde,el primer parámetro dentro de los corchetes agudos hace referencia al número de bloques en paralelo que nos gustaría que el dispositivo ejecute en nuestro kernel. El segundo parámetro representa el número de hilos que queremos enviar por cada bloque.Para finalizar, al momento de realizar una llamada al kernel debemos de considerar que el número de bloques máximo que pueden ser lanzados es 65,535 y el número de hilos por bloque máximo que pueden ser lanzados es 512.

3. Desarrollo del proyecto

3.1. Descripción general

Durante el desarrollo del proyecto estaremos utilizando la estructura mostrada en la Figura 5, en la que podemos apreciar los módulos o bloques de los que está compuesto el proyecto.

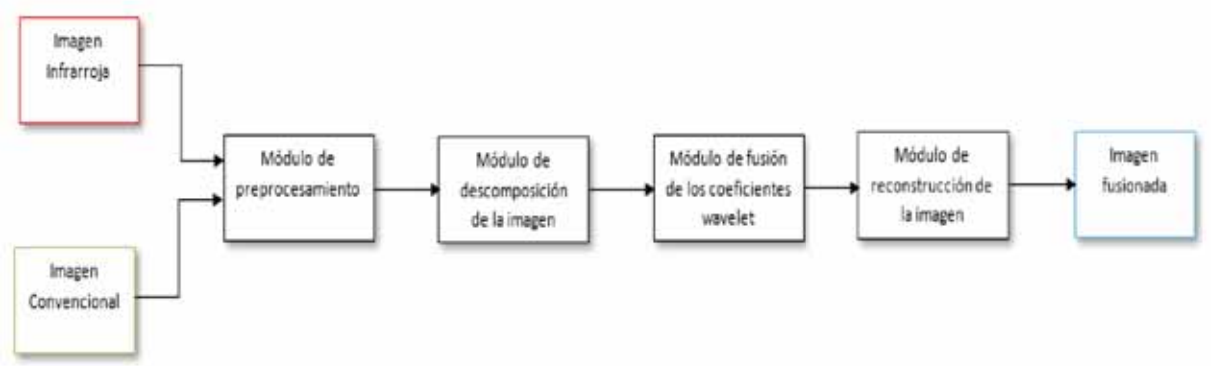

Figura 5: Módulos usados para la aplicación de fusión de imágenes por medio de la Transformada *Wavelet* de Haar.

3.2. Fusión de imágenes secuencial

En esta sección se detallará el proceso de fusión de imágenes de forma secuencial, explicando la funcionalidad de los módulos utilizados, así como los resultados esperados en cada uno de ellos. Los módulos que veremos en esta sección son los siguientes:

- Módulo de Preprocesamiento
- Módulo de descomposición de la imagen secuencial.
- Módulo de fusión de los coeficientes de *Wavelet* secuencial.
- Módulo de reconstrucción de la imagen secuencial.
- Módulo de fusión de canales RGB.

Una vez dada esta pequeña introducción comencemos.

3.2.1. Módulo de Preprocesamiento

Antes de aplicarle cualquier tipo de tratamiento a las imágenes fuente, primero debemos de prepararlas, de tal forma que estén completamente listas para aplicarles el procesamiento deseado. En este caso, dividiremos nuestras imágenes infrarroja y visible en sus canales RGB, dando como resultado una nueva imagen para cada uno de los canales, es decir, las imágenes resultantes son la representación de los canales RGB de la imagen original. Este proceso se muestra en la Figura 6. A partir de este momento, nos referiremos como "imagen fuente" a cada una de las imágenes obtenidas de la descomposición RGB de las imágenes originales.

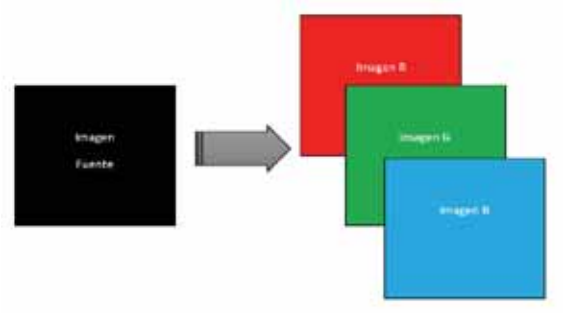

Figura 6: Descomposición de la imagen fuente en sus canales RGB.

3.2.2. Módulo de descomposición de la imagen secuencial

Como vimos en la sección 2.1.3 para calcular la Transformada *Wavelet* bidimensional de una imagen primero debemos calcular la DHWT de cada una de las filas de la imagen original, a la imagen resultante de este proceso debemos aplicarle nuevamente la Transformada *Wavelet* Discreta de Haar, pero en esta ocasión a cada una sus columnas, obteniendo así una imagen descompuesta en cuatro subandas. Para simplificar el cálculo de la Transformada *Wavelet* de Haar, hemos dividido este procedimiento en dos módulos, en el primero de ellos calcularemos el DHWT de las filas, mientras que en el otro calcularemos la DHWT de las columnas.

Transformada *Wavelet* Discreta de Haar de las filas.

Este módulo es el encargado de realizar el cálculo de la Transformada *Wavelet* Discreta de Haar en cada una de las filas de las imágenes fuente. Obteniendo como resultado la descomposición de las imágenes originales en dos subimagenes, en donde la primer subimagen es el resultado de calcular la tendencia a lo largo de las filas de la imagen fuente, mientras que la otra es creada al obtener la fluctuación de estas.

Transformada Discreta de *Wavelet* de Haar de las columnas.

A diferencia del módulo anterior, este es el encargado de obtener la Transformada *Wavelet* Discreta de Haar a lo largo de las columnas de la imagen obtenida en el módulo pasado. Como consecuencia de este proceso surge una imagen dividida en cuatro subimagenes, las cuales hemos explicado con anterioridad en la sección 2.1.3.

Los algoritmos usados en el cálculo de la DWHT en las filas y en las columnas se muestran en la Figura 7.

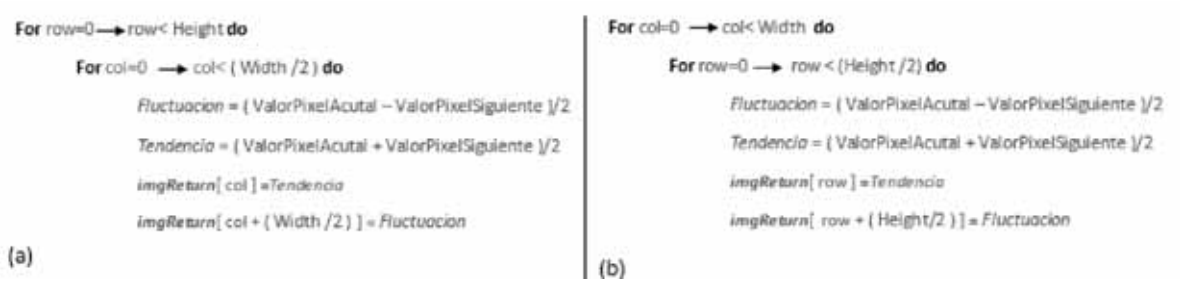

Figura 7: (a) Cálculo de la Transformada Discreta *Wavelet* de Haar para cada una de las filas; (b) Cálculo de la Transformada Discreta *Wavelet* de Haar en cada una de las Columnas.

3.2.3. Módulo de fusión de los coeficientes de *Wavelet* secuencial.

Una vez que hemos descompuesto las imágenes fuente infrarroja y visible en el apartado anterior, procedemos a combinar y seleccionar los coeficientes *wavelet* de cada una de las imágenes de acuerdo a un conjunto de reglas de selección. Las reglas utilizadas para realizar el proceso de fusión son:

1. Para las regiones de bajas frecuencias se obtiene el promedio de los coeficientes *Wavelet* de las imágenes fuente.

2. Para las regiones de altas frecuencias se comparan los coeficientes *Wavelet* de las imágenes fuente y se seleccionará la de mayor valor.

Al aplicar estas reglas, obtendremos como resultado una nueva imagen con regiones de altas y bajas frecuencias que contienen coeficientes tanto de la imagen infrarroja como de la imagen visible.

3.2.4. Módulo de reconstrucción de la imagen secuencial.

Al igual que hicimos con el cálculo de la DHWT bidimensional, obtendremos la Transformada Inversa mediante dos módulos, en el primero de ellos calcularemos la inversa de la transformada en cada una de las filas, mientras que, en el segundo módulo determinaremos la Transformada Inversa *Wavelet* en cada una de las columnas.

Transformada Inversa *Wavelet* de Haar de las filas

Este módulo será el encargado de elaborar el primer paso en la reconstrucción de la imagen fusionada. Para ello se procederá a determinar la Transformada Inversa de Haar en cada una de las filas de la imagen.

Transformada Inversa *Wavelet* de Haar de las comlumnas

Como último paso en la reconstrucción de la imagen fusionada, este módulo es el encargado de efectuar el cálculo de la Transforma Inversa a lo largo de las columnas de la imagen mostrada en la Figura anterior. Consiguiendo así, la reconstrucción de la imagen fusionada Figura 8.

Figura 8: Imagen fusionada finalmente reconstruida

Los algoritmos aplicados para el cálculo de la Transformada Inversa en las filas y en las columnas se muestran en la Figura 9.

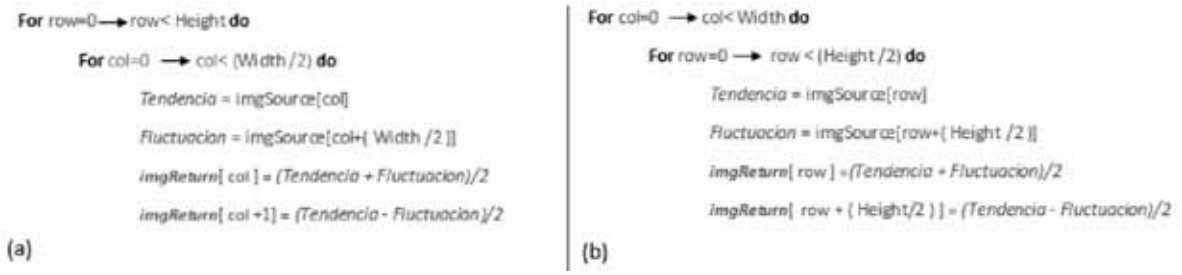

Figura 9: (a) Cálculo de la Transformada Inversa Discreta de *Wavelet* de Haar para cada una de las filas; (b) Cálculo de la Transformada Inversa Discreta de *Wavelet* de Haar en cada una de las Columnas.

3.2.5. Fusión de los canales RGB

Para finalizar el proceso de fusión de las imágenes infrarroja y visible, mezclaremos de nueva cuenta cada uno de los canales RGB de las imágenes originales , terminado así el procesamiento de fusión de imágenes por medio de las Transformada Discreta *Wavelet* de Haar.

Puedes consultar el apéndice A para observar con mayor detalle la implementación secuencial de la fusión de imágenes.

3.3. Fusión de imágenes con cómputo acelerado por GPU

En el apartado anterior analizamos el proceso secuencial de la fusión de imágenes, sin embargo, se espera que la presente aplicación de la Transformada de *Wavelet* sea usada para asistir a los conductores nocturnos en tiempo real. Debido a esto, en esta sección hablaremos sobre la paralelización de la fusión de imágenes mediante el uso de Unidades Gráficas de Procesamiento, buscando de esta forma obtener en tiempo real la imagen fusionada.
Los módulos que se presentan a continuación serán paralelizados, ya que se consideran que son la parte de la fusión de imágenes que más tiempo toman en llevarse a cabo.

- Módulo de descomposición de la imagen.
- Módulo de fusión de los coeficientes de *wavelet*.
- Módulo de reconstrucción de la imagen.

3.3.1. Módulo de descomposición de la imagen en paralelo.

Como observamos anteriormente dividiremos el módulo en dos, uno para el cálculo de la DHTW en filas y otro en columnas.

Transformada Discreta de *Wavelet* de Haar de las filas paralelizada.

Este módulo realizara el cálculo de la DHWT en cada una de las filas de la imagen fuente, para ello debemos de calcular el número de hilos y de bloques que queremos que se ejecuten en nuestra GPU. En este caso, enviaremos un arreglo bidimensional de hilos de 32 x 32, dando un total de 1024 hilos por bloque. Al igual que con los hilos, lanzaremos bidimensionalmente *(WIDHT/2)/(hilos en X) * HEIGHT/(hilos en Y)* bloques.

Para lograr una mayor comprensión de lo mencionado en el párrafo anterior, daremos un pequeño ejemplo. Imaginemos que tenemos una imagen de 512 x 512 píxeles y el número de hilos lanzados por bloque será de 1024, entonces para calcular el número de bloques realizamos lo siguiente:

Bloques en X=(WIDHT/2)/hilos en X = (512/2)/32=8 bloques en X
Bloques en Y= HEIGH/hilos en Y =512/32=16 bloques en Y
Bloques lanzados en total=128
Hilos enviados en total= hilos por bloque * bloques totales=1024x128=131072 hilos.

Como nuestra imagen es de 512x512 píxeles, necesitaremos 262,144 hilos ejecutándose al mismo tiempo, sin embargo, nosotros sólo estamos enviando 131,072, lo cual es la mitad de los hilos que necesitamos. Esto se debe a que en nuestra implementación cada hilo se encargará de obtener la tendencia y la fluctuación de sus respectivos píxeles.

La representacón de los hilos y de los bloques lanzados en nuestro ejemplo puede verse en la Figura 10.

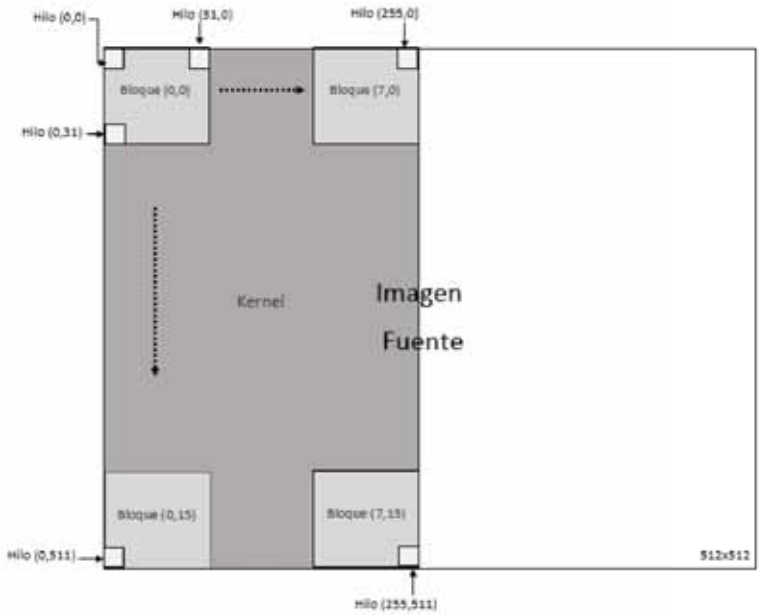

Figura 10: Representación de los hilos y de los bloques lanzados para calcular la Transformada Discreta *Wavelet* de Haar en cada una de las filas.

Transformada Discreta de *Wavelet* de Haar de las columnas paralelizada.

De forma similar al módulo que calcula la DHWT en las filas, tenemos que estimar el número de hilos y de bloques que serán ejecutados en nuestro kernel que se encargará de calcular la DHWT de cada una de las columnas. Para este caso en específico el número de hilos por bloque será el mismo que en el módulo anterior (1024 hilos por bloque), lo que sí cambiaremos esta vez será la forma en que enviaremos lo bloques a nuestro kernel, ya que en este caso dispararemos *WIDHT /(hilos en X) * (HEIGHT /2)/(hilos en Y)* bloques.

Los bloques e hilos lanzados en esta sección se ven representados en la Figura 11.

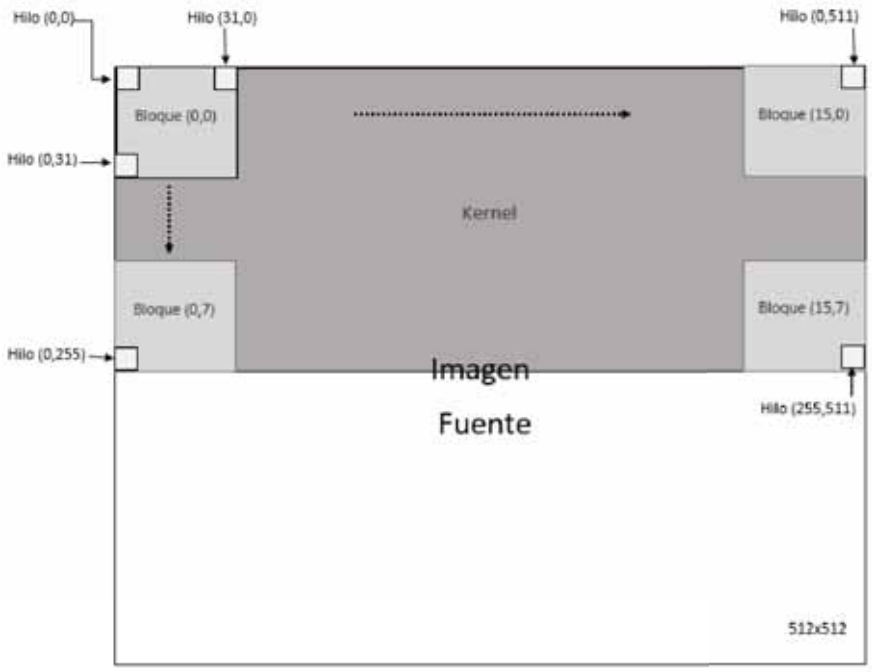

Figura 11: Representación de los hilos y de los bloques lanzados para calcular la Transformada Discreta *Wavelet* de Haar en cada una de las columnas.

De forma similar al apartado anterior estamos enviando solamente 131,072, lo cual es la mitad de los hilos que necesitamos, pues en nuestra implementación cada hilo se encargará de obtener la tendencia y la fluctuación de sus respectivos píxeles.

3.3.2. Módulo de fusión de los coeficientes de *wavelet* en paralelo.

Este módulo como su nombre lo indica, es el encargado de realizar la fusión o selección del coeficiente wavelet de las imágenes fuente. Para esta implementación lanzaremos en nuestro kernel *WIDHT /(hilos en X) * HEIGHT/8hilos en Y)* bloques.

La representación de los bloques e hilos enviados a nuestro kernel podemos observarlo en la Figura 12.

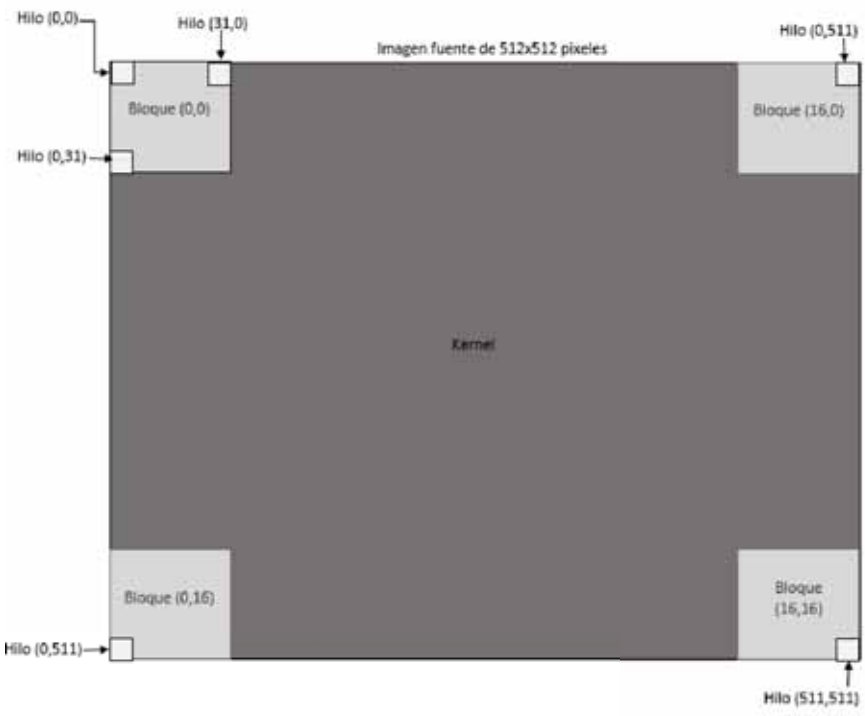

Figura 12: Representación de los hilos y de los bloques lanzados para fusión de los coeficientes de *wavelet*.

Como podemos notar, en esta ocasión hemos enviado un hilo para cada uno de los píxeles de la imagen fuente, esto se debe a que cada hilo es el encargado de fusionar los coeficientes wavelet de la imagen infrarroja con los de la visible, colocando finalmente el resultado de esta fusión en el píxel correspondiente a cada hilo.

3.3.3. Módulo de reconstrucción de la imagen

Para finalizar está sección abordaremos sin entrar en detalles el módulo de reconstrucción de la imagen fusionada, el cual está compuesto por el módulo de Transformada Inversa de *Wavelet* de Haar de las columnas y por el módulo Transformada Inversa de *Wavelet* de Haar de las filas.

El funcionamiento de estos dos módulos es idéntico a los módulos de descomposición de la imagen paralelizado, es decir, el kernel enviado en el módulo de la DHWT de las filas es igual al de la Figura 10, mientras que para el módulo de la DHWT de las columnas es idéntico al de la Figura 11.

Puedes consultar el apéndice B para observar con mayor detalle la implementación paralela de la fusión de imágenes.

4. Resultados

A continuación se presenta en la Figura 13 la imagen obtenida al fusionar la imagen infrarroja Figura 18 y la imagen visible Figura 19.

Figura 13: Imagen Fusionada.

Los tiempos de ejecución de la implementación paralela y de la secuencial se muestran en la siguiente tabla. Estos tiempos son el resultado de procesar diferentes resoluciones de las imágenes fuentes. Además dichos tiempos se pueden ver reflejados en las Figuras 14 y 15.

Las tarjetas utilizadas para la ejecución en paralelo son la tarjeta Quadro k420 y GeForce GTX TITAN X de Nvidia. Puedes consultar información detallada acerca de las tarjetas en los apéndices B.1 y B.2. La información de las tarjetas fue consultada en https://images.nvidia.com/content/pdf/quadro/data-sheets/75509_DS_NV_Quadro_K420_US_NV_HR.pdf y https://la.nvidia.com/object/geforce-gtx-titan-x-la.html#pdpContent=2 respectivamente.

Resolución	Ejecución Secuencial	Quadro k420	GeForce GTX TITAN X
512x512	0.40932	0.0605867	0.0947108
1024x1024	1.62611	0.113456	0.1223858
2048x2048	6.96076	0.3280217	0.2394597
4096x4096	28.73117	0.84539	0.7084557
8192x8192	145.7755	2.7775	2.1040057

Tabla 1: Tiempos de ejecucón secuencial y paralelo

Figura 14: Tiempos de ejecución de la implementaicón secuencial.

Figura 15: Tiempos de ejecución de la implementaicón paralela.

Acontinuación se puede observar en la Figura 16 la aceleración de los tiempos de ejecución obtenida al paralelizar el proceso de la fusión de imágenes.

Figura 16: Gráfica de aceleración de los tiempos de ejecución.

Finalmente, en la Figura 17 se muestra una comparativa gráfica de los tiempos de ejecución de las diferentes implementaciones.

Figura 17: Tiempos de ejecución secuancial y paralelo.

5. Análisis y discusión de resultados

Para un mejor análisis de los resultados obtenidos se han resaltado las características más importantes de la imagen visible Figura 19 e infrarroja Figura 18, de tal forma que las diferencias que existen entre ellas sean expuestas de forma clara. Podemos observar fácilmente que a diferencia de la imagen visible, en 1 y 2 de la imagen infrarroja los señalamientos de tránsito no se logran percibir. Sin embargo, en 3 y 4 de la imagen infrarroja las personas que transitan son fácilmente identificables, lo cual no sucede en 3 y 4 de la imagen visible.

Una vez que hemos identificado los principales elementos de cada una de las imágenes fuente, podemos apreciar lo que hemos ganado con la fusión imágenes. En 1 y 2 de la Figura 20 podemos identificar que los señalamientos de tránsito no se pierden como sucede en la imagen infrarroja Figura 18, además, en 3 y 4 de la Figura 20 las personas que transitan en la vialidad son fácilmente identificables lo cual era imposible de apreciar en la imagen visible Figura 19.

Figura 18: Imagen fuente visible.

Figura 19: Imagen fuente visible.

Figura 20: Imagen fusionada.

Por otro lado, en cuanto a los resultados obtenidos en el rendimiento de la implementación de la fusión de imágenes sobre las Unidades Gráficas de Procesamiento fue muy superior a la implementación secuencial, lo cual cumple con el objetivo de conseguir la imagen fusionada en un tiempo cercano al real.Una vez mencionado lo anterior deberíamos plantearnos la pregunta "¿Si obtenemos un mejor rendimiento con una GPU, por qué no realizamos todas las aplicaciones sobre estas?", es verdad que en el proyecto aquí presentado obtuvimos un rendimiento claramente mayor con la implementación en Unidades Gráficas de Procesamiento, sin embargo, existen aplicaciones en las que la dependencia de datos es tan grande que resulta complicado o incluso imposible realizar una paralelización, es aquí cuando las CPU entran en acción y resuelven este tipo de problemas. Entonces, debemos tener en claro que las GPU y CPU están diseñadas para realizar distintas tareas, en donde incluso y muy posiblemente tengan que trabajar una del lado de la otra haciendo lo que mejor saben hacer.

6. Conclusiones

En esta primera versión de la fusión de imágenes, considero que los resultados fueron satisfactorios, sin embargo, pueden mejorarse aún más. Esto puede lograse implementado y buscando reglas de selección de coeficientes de *wavelet* que nos permitan apreciar los elementos principales de las imágenes fuente con mayor claridad. Además, en las pruebas realizadas durante el proyecto, las imágenes fuente convencionales están en blanco y negro, lo cual altera considerablemente la imagen final fusionada, pues no se logran aprovechar al cien porciento las características y los colores que podrían ofrecernos imágenes convencionales a color. En cuanto al tiempo que se tomó para conseguir la imagen fusionada, puede ganarse mayor rendimiento si en lugar de utilizar la memoria global usamos la memoria *Shared*, lo cual nos reduciría la latencia. No obstante, los resultados obtenidos en el presente trabajo cumplieron con los propósitos planteados inicialmente.

Lista de Figuras

Bibliografía

[1] B. K. P. H. W. F. Herrington y I. Masaki, "Application of the discrete haar wavelet transform to image fusion for nighttime driving," en *IEEE Proceedings. Intelligent Vehicles Symposium, 2005*, Las Vegas, NV, USA, 2005, pp. 273–277.

[2] A. R. Pal y A. Singha, "A comparative analysis of visual and thermal face image fusion based on different wavelet family," en *2017 International Conference on Innovations in Electronics, Signal Processing and Communication (IESC)*, Shillong, 2017, pp. 213–218.

[3] K. K. y S. Arumuga Perumal, "Optimal decomposition level of discrete wavelet transform for pixel based fusion of multi-focused images," en *Department of Computer Science*, St. Hindu College, Tamilnadu, India, ene 1997.

[4] S. A. y C. Köse, "Multi-focus image fusion using stationary wavelet transform (swt) with principal component analysis (pca)," en *Department of Computer Engineering*, Karadeniz Technical University, Trabzon,Turquía, ene 1997.

[5] B. M. Sanchez, "Paralelización de transformada de wavelet en gpus," Proyecto terminal, División de Ciencias Básicas e Ingeniería, Universidad Autónoma Metropolitana Azcapotzalco, México, 2016.

[6] A. R. P. y A. Singha, "A comparative analysis of visual and thermal face image fusion based on different wavelet family," en *Department of Computer Science and Engineering*, Tripura University Agartala, India, sep 1987.

[7] D. Sundararajan, *Discrete Wavelet Transform : A Signal Processing Approach*. Beaverton,WA: John Wiley Sons, Incorporated, 2015.

[8] J. S. Walker, *A primer on wavelets and their scientific applications*. Boca Raton, Fla: Chapman Hall/CRC, 2008.

[9] J. P. Li, *Proceedings of the Third International Conference on Wavelet Analysis and its Applications*. Singapore: World Scientific., 2003.

[10] J. Sanders y E. Katndrot, *CUDA by example; an introduction to general-purpose GPU programming*. Boston, MA: Addison-Wesley, 2011.

[11] J. Nickolls y W. J. Dally, "The gpu computing era," *in IEEE Micro*, vol. 30, no. 2, pp. 56–69, March-April 2010.

[12] D. L. S. G. J. E. S. J. D. Owens, M. Houston y J. C. Phillips, "Gpu computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879–899, May 2008.

A. Apéndice Códigos fuente

A.1. Código Secuencial

```
#include<iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <time.h>
using namespace cv;
using namespace std;
#define HEIGHT 512
#define WIDTH 512

void descomposicionRGB(Mat imageSource, double* imgReturnR, double*
    imgReturnG, double* imgReturnB);
Mat MatImage(double* image_buffer);
void HaarFilas(double*  imgSource,double*  imgReturn);
void HaarColumnas(double*  imgSource,double*  imgReturn);
void fusion(double* imgSourceA, double* imgSourceB, double* imgReturn);
void HaarInversaCols(double* imgSource,double* imgReturn);
void HaarInversaFilas(double* imgSource, double* imgReturn);
Mat procesar(double* imgVisible, double*  imgInfrarroja);

int main(int argc,char** argv)
{
  Mat imgInf,imgVisible;
  clock_t t_ini,t_fin;
  double segs;
  //Lectura de las imagenes fuente
  imgInf=imread("/home/mmaldonado/Documents/ProyectoTerminal/imagenes/
    infrarrojo/img1_512x512.tiff");
  imgVisible=imread("/home/mmaldonado/Documents/ProyectoTerminal/imagenes/
    visible/img1_512x512.tiff");

  double*Inf_ChannelR=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
    double*Vis_ChannelR=(double*)malloc((WIDTH*HEIGHT)*sizeof(double));
    double*Inf_ChannelG=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));

    double*Vis_ChannelG=(double*)malloc((WIDTH*HEIGHT)*sizeof(double));
    double*Inf_ChannelB=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
    double*Vis_ChannelB=(double*)malloc((WIDTH*HEIGHT)*sizeof(double));
    t_ini=clock();

    //Descomposicion de las imagenes fuente en sus canales RGB
    descomposicionRGB(imgInf,Inf_ChannelR,Inf_ChannelG,Inf_ChannelB);
    descomposicionRGB(imgVisible,Vis_ChannelR,Vis_ChannelG,Vis_ChannelB);

    //Fusion de los canales R de las imagenes fuente
    Mat fusionR(HEIGHT,WIDTH, CV_8UC1);
    fusionR=procesar(Vis_ChannelR,Inf_ChannelR);
    free(Inf_ChannelR);
```

```
45      free(Vis_ChannelR);
46
47      //Fusion de los canales G de las imagenes fuente
48      Mat fusionG(HEIGHT, WIDTH, CV_8UC1);
49      fusionG=procesar(Vis_ChannelG,Inf_ChannelG);
50      free(Inf_ChannelG);
51      free(Vis_ChannelG);
52
53      //Fusion de los canales B de las imagenes fuente
54    Mat fusionB(HEIGHT, WIDTH, CV_8UC1);
55      fusionB=procesar(Vis_ChannelB,Inf_ChannelB);
56      free(Inf_ChannelB);
57      free(Vis_ChannelB);
58
59      //Combinacion de los canales RGB fusionados
60    Mat imgFinal(HEIGHT, WIDTH, CV_8UC1);
61      vector<Mat> channels;
62      channels.push_back(fusionB);
63      channels.push_back(fusionG);
64      channels.push_back(fusionR);
65      merge(channels, imgFinal);
66
67      t_fin=clock();
68      segs=(double)(t_fin-t_ini)/CLOCKS_PER_SEC;
69      cout << "Tiempo de ejecucion: " << segs <<" segundos"<< endl;
70      imwrite( "/home/mmaldonado/Documents/ProyectoTerminal/fusionOk/cuda.jpg
        " , imgFinal);
71
72    waitKey(0);
73
74    return 0;
75  }
76  //Modulo para convertir de tipo double* a tipo Mat
77  Mat MatImage(double* image_buffer)
78  {
79    Mat imgShow(WIDTH,HEIGHT, CV_8UC1);
80    for (int i=0;i<WIDTH;i++)
81          {
82                  for(int j=0;j<HEIGHT;j++)
83                  {
84            imgShow.at<uchar>(i,j)=(unsigned char)image_buffer[(i * WIDTH) +
        j];
85                  }
86          }
87      return imgShow;
88  }
89
90  Mat procesar(double* imgVisible, double*  imgInfrarroja)
91  {
92  //TRANSFORMADA DE HAAR de la  IMAGEN INFRARROJA
93    double*inf_HaarRows=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
94    double*inf_Haar=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
```

```
95
96    HaarFilas(imgInfrarroja,inf_HaarRows);
97    HaarColumnas(inf_HaarRows,inf_Haar);
98    free(inf_HaarRows);
99
100 //TRANSFORMADA DE HAAR de la IMAGEN VISIBLE
101   double*Vis_HaarRows=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
102   double*Vis_Haar=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
103
104   HaarFilas(imgVisible,Vis_HaarRows);
105   HaarColumnas(Vis_HaarRows,Vis_Haar);
106   free(Vis_HaarRows);
107
108 //FUSION DE IMAGENES
109   double*imgFusion=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
110   fusion(Vis_Haar,inf_Haar,imgFusion);
111   free(inf_Haar);
112   free(Vis_Haar);
113
114 //TRANSFORMADA INVERSA DE HAAR
115   double*inversaRows=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
116   HaarInversaFilas(imgFusion,inversaRows);
117   free(imgFusion);
118
119   double*inversaCols=(double *)malloc((WIDTH*HEIGHT)*sizeof(double));
120   HaarInversaCols(inversaRows,inversaCols);
121   free(inversaRows);
122
123   Mat ImageFusion(HEIGHT, WIDTH, CV_8UC1);
124  //Convertir de double* a Mat
125   ImageFusion=MatImage(inversaCols);
126   free(inversaCols);
127   return ImageFusion;
128 }
129
130 //Descomposicion en canales RGB de la imagen fuente
131 void descomposicionRGB(Mat imageSource, double* imgReturnR, double*
      imgReturnG, double* imgReturnB)
132 {
133   for (int i=0;i<imageSource.rows;i++)
134   {
135     for(int j=0;j<imageSource.cols;j++)
136     {
137       Vec3b pixel = imageSource.at<Vec3b>(i,j);
138       imgReturnB[i * imageSource.cols + j]=pixel[0];
139       imgReturnG[i * imageSource.cols + j]=pixel[1];
140       imgReturnR[i * imageSource.cols + j]=pixel[2];
141     }
142   }
143 }
144
145
```

```
146 /*Transformada de Haar aplicada a las filas de la imagen*/
147
148 void HaarFilas(double*  imgSource,double*  imgReturn){
149     int pix_act,pix_sig;
150     double val_act, val_sig;
151   int m=WIDTH/2;
152   double a,d;
153
154   for (int row = 0; row < WIDTH; row++)
155   {
156         for (int col = 0; col < m; col++)
157         {
158             pix_act = 2*col;
159             pix_sig = (2*col)+1;
160             val_act =imgSource[(row*WIDTH)+pix_act];
161             val_sig =imgSource[(row*WIDTH)+pix_sig];
162
163             //Senal de tendencia
164             a=(val_act + val_sig)/2;
165             //Senal de fluctuacion
166             d=(val_act - val_sig)/2;
167             imgReturn[(row*WIDTH)+col]=a;
168             imgReturn[(row*WIDTH)+(col+m)]=d;
169         }
170   }
171 }
172
173 /*Transformada de Haar aplicada a las columnas de la imagen*/
174 void HaarColumnas(double*  imgSource,double*  imgReturn)
175 {
176   int pix_act,pix_sig;
177   double val_act, val_sig;
178   int n=HEIGHT/2;
179   double a,d;
180
181   for (int col = 0; col < HEIGHT; col++)
182   {
183         for (int row = 0; row < n; row++)
184         {
185             pix_act = 2*row;
186             pix_sig = (2*row)+1;
187             val_act = imgSource[col+(HEIGHT*pix_act)];
188             val_sig = imgSource[col+(HEIGHT*pix_sig)];
189
190             //Senal de tendencia
191             a= (val_act + val_sig)/2;
192             //Senal de fluctuacion
193             d=(val_act - val_sig)/2;
194             imgReturn[(row*HEIGHT)+col]=a;
195             imgReturn[(HEIGHT*(row+n))+col]=d;
196         }
197   }
```

```
198
199 }
200
201
202 //Fusion de los coeficientes wavelet
203 void fusion(double* imgSourceA, double* imgSourceB, double* imgReturn)
204 {
205   int m, n;
206   m=HEIGHT/2;
207   n=WIDTH/2;
208   for(int col=0;col<HEIGHT;col++)
209   {
210     for(int row=0;row<WIDTH;row++)
211     {
212       if( col<m)
213       {
214         if(imgSourceA[(col*WIDTH)+row] > imgSourceB[(col*WIDTH)+row])
215                 {
216                   imgReturn[(col*WIDTH)+row]=imgSourceA[(col*WIDTH)+row];
217                 }
218                 else
219                 {
220                   imgReturn[(col*WIDTH)+row]=imgSourceB[(col*WIDTH)+row];
221                 }
222       }
223       else{
224         imgReturn[(col*WIDTH)+row]=(imgSourceA[(col*WIDTH)+row] +
    imgSourceB[(col*WIDTH)+row])/2;
225       }
226     }
227   }
228 }
229
230 //Transformda wavelet inversa de Haar de las columnas
231 void HaarInversaCols(double* imgSource,double* imgReturn)
232 {
233   int m;
234   double a, d;
235   m=HEIGHT/2;
236
237   for(int col=0;col<HEIGHT;col++)
238   {
239     for(int row=0;row<m;row++)
240     {
241       //Senal de tendencia
242       a=imgSource[(HEIGHT*row)+col];
243       //Senal de fluctuacion
244       d=imgSource[(HEIGHT*(row+m))+col];
245       imgReturn[((row*2)*HEIGHT)+col]= (a+d)/2;
246       imgReturn[(((row*2)+1)*HEIGHT)+col]= (a-d)/2;
247     }
248
```

```
249   }
250 }
251
252 //Transformda wavelet inversa de Haar de las filas
253 void HaarInversaFilas(double* imgSource, double* imgReturn)
254 {
255     int n;
256     double a, d;
257   n=WIDTH/2;
258
259         for(int row=0;row<WIDTH;row++)
260         {
261                 for(int col=0;col<n;col++)
262                 {
263                   //Senal de tendencia
264                     a= imgSource[(row*WIDTH)+col];
265                     //Senal de fluctuacion
266                     d= imgSource[(row*WIDTH)+(col+n)];
267                     imgReturn[(row*WIDTH)+(col*2)] =a+d;
268                     imgReturn[(row*WIDTH)+(col*2)+1]=a-d;
269                 }
270
271         }
272 }
```

Listing 1: C example

A.2. Código Paralelo

```
#include<iostream>
#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <cuda.h>
using namespace cv;
using namespace std;

#define WARP 32
#define HEIGHT 512
#define WIDTH 512

void descomposicionRGB(Mat imageSource, double* imgReturnR, double*
    imgReturnG, double* imgReturnB);
Mat MatImage(double* image_buffer);
void mostrar(double* image_buffer);
__global__ void HaarFilas(double*  imgSource, double*  imgReturn);
__global__ void HaarColumnas(double*  imgSource,double*  imgReturn);
__global__ void fusion(double* imgSourceA, double* imgSourceB, double*
    imgReturn);
__global__ void HaarInversaCols(double* imgSource,double* imgReturn);
__global__ void HaarInversaFilas(double* imgSource, double* imgReturn);
Mat procesar(double* imgVisible, double*  imgInfrarroja);

int main(int argc,char** argv)
{
  Mat imgInf,imgVisible;
  cudaEvent_t start,stop;
  cudaEventCreate(&start);
  cudaEventCreate(&stop);

  float segs;
  //Lectura de las imagenes fuente
  imgInf=imread("/home/phoenix/Descargas/PT/imagenes/infrarrojo/
    img1_512x512.tiff");
  imgVisible=imread("/home/phoenix/Descargas/PT/imagenes/visible/
    img1_512x512.tiff");

  double*Inf_ChannelR=(double *)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));
  double*Vis_ChannelR=(double*)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));
  double*Inf_ChannelG=(double *)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));

  double*Vis_ChannelG=(double*)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));
  double*Inf_ChannelB=(double *)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));
  double*Vis_ChannelB=(double*)malloc((imgVisible.rows*imgVisible.cols)*
    sizeof(double));
```

```
41
42    cudaEventRecord(start,0);
43    descomposicionRGB(imgInf,Inf_ChannelR,Inf_ChannelG,Inf_ChannelB);
44    descomposicionRGB(imgVisible,Vis_ChannelR,Vis_ChannelG,Vis_ChannelB);
45
46    //Fusion de los canales R de las imagenes fuente
47    Mat fusionR(HEIGHT,WIDTH, CV_8UC1);
48    fusionR=procesar(Vis_ChannelR,Inf_ChannelR);
49    free(Inf_ChannelR);
50    free(Vis_ChannelR);
51
52    //Fusion de los canales G de las imagenes fuente
53    Mat fusionG(HEIGHT, WIDTH, CV_8UC1);
54    fusionG=procesar(Vis_ChannelG,Inf_ChannelG);
55    free(Inf_ChannelG);
56    free(Vis_ChannelG);
57
58    //Fusion de los canales B de las imagenes fuente
59    Mat fusionB(HEIGHT, WIDTH, CV_8UC1);
60    fusionB=procesar(Vis_ChannelB,Inf_ChannelB);
61    free(Inf_ChannelB);
62    free(Vis_ChannelB);
63
64    //Combinacion de los canales RGB fusionados
65    Mat imgFinal(HEIGHT, WIDTH, CV_8UC1);
66    vector<Mat> channels;
67    channels.push_back(fusionB);
68    channels.push_back(fusionG);
69    channels.push_back(fusionR);
70    merge(channels, imgFinal);
71
72    cudaEventRecord(stop,0);
73    cudaEventSynchronize(stop);
74    cudaEventElapsedTime(&segs,start,stop);
75
76    cout << "Tiempo de ejecucion: " << segs <<" segundos"<< endl;
77    cudaEventDestroy(start);
78    cudaEventDestroy(stop);
79    imwrite( "/home/phoenix/Descargas/PT/fusionOk/cuda.jpg" , imgFinal);
80    waitKey(0);
81    return 0;
82 }
83
84 Mat MatImage(double* image_buffer)
85 {
86    Mat imgShow(WIDTH,HEIGHT, CV_8UC1);
87    for (int i=0;i<WIDTH;i++)
88          {
89                for(int j=0;j<HEIGHT;j++)
90                {
91             imgShow.at<uchar>(i,j)=(unsigned char)image_buffer[i * WIDTH + j
      ];
```

```
92                  }
93          }
94      return imgShow;
95  }
96
97
98  Mat procesar(double* imgVisible, double*  imgInfrarroja)
99  {
100   dim3 hilos(WARP,WARP);
101 //Bloques lanzados para la DHWT de las Filas
102   dim3 bloquesFilas((WIDTH/2)/hilos.x,HEIGHT/hilos.y);
103 //Bloques lanzados para la DHWT de las Columnas
104   dim3 bloquesCols(WIDTH/hilos.x,(HEIGHT/2)/hilos.y);
105 //Bloques lanzados para la fusion de los coefiecientes wavelet
106   dim3 bloquesFusion(WIDTH/hilos.x,HEIGHT/hilos.y);
107
108 //PROCESAMIENTO DE LA IMAGEN INFRARROJA;
109
110   double*dev_imgInfrarroja;
111   double*dev_inf_HaarRows;
112
113   cudaMalloc((void **)&dev_inf_HaarRows,(WIDTH*HEIGHT)*sizeof(double));
114   cudaMalloc((void **)&dev_imgInfrarroja,(WIDTH*HEIGHT)*sizeof(double));
115   cudaMemcpy(dev_imgInfrarroja,imgInfrarroja,(WIDTH*HEIGHT)*sizeof(double),
        cudaMemcpyHostToDevice);
116
117   HaarFilas<<<bloquesFilas,hilos>>>(dev_imgInfrarroja,dev_inf_HaarRows);
118   cudaFree(dev_imgInfrarroja);
119
120   double*dev_inf_Haar;
121   cudaMalloc((void **)&dev_inf_Haar,(WIDTH*HEIGHT)*sizeof(double));
122
123   HaarColumnas<<<bloquesCols,hilos>>>(dev_inf_HaarRows,dev_inf_Haar);
124   cudaFree(dev_inf_HaarRows);
125
126
127 //PROCESAMIENTO DE LA IMAGEN VISIBLE
128
129   double*dev_imgVisible;
130   double*dev_vis_HaarRows;
131
132   cudaMalloc((void **)&dev_vis_HaarRows,(WIDTH*HEIGHT)*sizeof(double));
133   cudaMalloc((void **)&dev_imgVisible,(WIDTH*HEIGHT)*sizeof(double));
134   cudaMemcpy(dev_imgVisible,imgVisible,(WIDTH*HEIGHT)*sizeof(double),
        cudaMemcpyHostToDevice);
135
136   HaarFilas<<<bloquesFilas,hilos>>>(dev_imgVisible,dev_vis_HaarRows);
137   cudaFree(dev_imgVisible);
138
139   double*dev_vis_Haar;
140   cudaMalloc((void **)&dev_vis_Haar,(WIDTH*HEIGHT)*sizeof(double));
141
```

```
142   HaarColumnas<<<bloquesCols,hilos>>>(dev_vis_HaarRows,dev_vis_Haar);
143   cudaFree(dev_vis_HaarRows);
144
145 //FUSION DE IMAGENES
146
147   double*dev_imgFusion;
148   cudaMalloc((void **)&dev_imgFusion,(WIDTH*HEIGHT)*sizeof(double));
149
150   fusion<<<bloquesFusion,hilos>>>(dev_vis_Haar,dev_inf_Haar,dev_imgFusion);
151
152   cudaFree(dev_inf_Haar);
153   cudaFree(dev_vis_Haar);
154
155 //TRANSFORMAR INVERSA
156
157   double*dev_inversaCols;
158   cudaMalloc((void **)&dev_inversaCols,(WIDTH*HEIGHT)*sizeof(double));
159
160   HaarInversaCols<<<bloquesCols,hilos>>>(dev_imgFusion,dev_inversaCols);
161   cudaFree(dev_imgFusion);
162
163   double*dev_imgReconstruida;
164   cudaMalloc((void **)&dev_imgReconstruida,(WIDTH*HEIGHT)*sizeof(double));
165
166   HaarInversaFilas<<<bloquesFilas,hilos>>>(dev_inversaCols,
        dev_imgReconstruida);
167   cudaFree(dev_inversaCols);
168
169   double*imgReconstruida=(double*)malloc((WIDTH*HEIGHT)*sizeof(double));
170
171   cudaMemcpy(imgReconstruida, dev_imgReconstruida, WIDTH * HEIGHT * sizeof(
        double), cudaMemcpyDeviceToHost);
172   cudaFree(dev_imgReconstruida);
173
174   Mat ImageFusion(HEIGHT, WIDTH, CV_8UC1);
175   ImageFusion=MatImage(imgReconstruida);
176   free(imgReconstruida);
177
178   return ImageFusion;
179
180 }
181
182 //Descomposicion en canales RGB de la imagen fuente
183 void descomposicionRGB(Mat imageSource, double* imgReturnR, double*
      imgReturnG, double* imgReturnB)
184 {
185   for (int i=0;i<imageSource.rows;i++)
186   {
187     for(int j=0;j<imageSource.cols;j++)
188     {
189       Vec3b pixel = imageSource.at<Vec3b>(i,j);
190       imgReturnB[i * imageSource.cols + j]=pixel[0];
```

```
191         imgReturnG[i * imageSource.cols + j]=pixel[1];
192         imgReturnR[i * imageSource.cols + j]=pixel[2];
193       }
194     }
195 }
196
197 /*Transformada de Haar aplicada a las filas */
198
199 __global__ void HaarFilas(double*  imgSource,double*  imgReturn){
200
201   int col;
202   int row;
203
204   int hiloX = blockIdx.x * blockDim.x + threadIdx.x;
205   int hiloY = blockIdx.y * blockDim.y + threadIdx.y;
206   int tid = hiloY * (gridDim.x * blockDim.x) + hiloX;
207   //Posicion del hilo en las columnas
208   col = tid / HEIGHT;
209   //Posicion del hilo en las filas
210   row = tid %WIDTH;
211
212   int pix_act,pix_sig;
213   double val_act, val_sig;
214   int m=WIDTH/2;
215   double a,d;
216
217   pix_act = 2*col;
218   pix_sig = (2*col)+1;
219   val_act =imgSource[(row*WIDTH)+pix_act];
220   val_sig =imgSource[(row*WIDTH)+pix_sig];
221
222   //Senal de tendencia
223   a=(val_act + val_sig)/2;
224   //Senal de fluctuacion
225   d=(val_act - val_sig)/2;
226
227   imgReturn[(row*WIDTH)+col]=a;
228   imgReturn[(row*WIDTH)+(col+m)]=d;
229
230 }
231
232 /*Transformada de Haar aplicada a las Columnas */
233 __global__ void HaarColumnas(double*  imgSource,double*  imgReturn)
234 {
235   int col;
236   int row;
237
238   int hiloX = blockIdx.x * blockDim.x + threadIdx.x;
239   int hiloY = blockIdx.y * blockDim.y + threadIdx.y;
240   int tid = hiloY * (gridDim.x * blockDim.x) + hiloX;
241   //Posicion del hilo en las columnas
242   col = tid %HEIGHT;
```

```
243   //Posicion del hilo en las filas
244   row = tid / WIDTH;
245
246   int pix_act,pix_sig;
247   double val_act, val_sig;
248   int n=HEIGHT/2;
249   double a,d;
250
251   pix_act = 2*row;
252   pix_sig = (2*row)+1;
253   val_act = imgSource[col+(HEIGHT*pix_act)];
254   val_sig = imgSource[col+(HEIGHT*pix_sig)];
255   //Senal de tendencia
256   a= (val_act + val_sig)/2;
257   //Senal de fluctuacion
258   d=(val_act - val_sig)/2;
259   imgReturn[(row*HEIGHT)+col]=a;
260   imgReturn[(HEIGHT*(row+n))+col]=d;
261 }
262
263 //Fusion de los coeficientes wavelet
264 __global__ void fusion(double* imgSourceA, double* imgSourceB, double*
      imgReturn)
265 {
266   int col;
267   int row;
268
269   int hiloX = blockIdx.x * blockDim.x + threadIdx.x;
270   int hiloY = blockIdx.y * blockDim.y + threadIdx.y;
271   int tid = hiloY * (gridDim.x * blockDim.x) + hiloX;
272   col = tid / HEIGHT;
273   row = tid %WIDTH;
274
275   int m, n;
276   m=HEIGHT/2;
277   n=WIDTH/2;
278   if(col<m)
279   {
280       imgReturn[(col*WIDTH)+row]=(imgSourceA[(col*WIDTH)+row] + imgSourceB
        [(col*WIDTH)+row])/2;
281   }
282   else{
283
284       if(imgSourceA[(col*WIDTH)+row] > imgSourceB[(col*WIDTH)+row])
285       {
286             imgReturn[(col*WIDTH)+row]=imgSourceA[(col*WIDTH)+row];
287       }
288       else
289       {
290             imgReturn[(col*WIDTH)+row]=imgSourceB[(col*WIDTH)+row];
291       }
292   }
```

```
293 }
294
295 //Transformda wavelet inversa de Haar de las columnas
296 __global__ void HaarInversaCols(double* imgSource,double* imgReturn)
297 {
298   int col;
299   int row;
300
301   int hiloX = blockIdx.x * blockDim.x + threadIdx.x;
302   int hiloY = blockIdx.y * blockDim.y + threadIdx.y;
303   //calculo de la posicion del hilo en el arreglo
304   int tid = hiloY * (gridDim.x * blockDim.x) + hiloX;
305   col = tid %HEIGHT;
306   row = tid / WIDTH;
307
308   int m;
309   double a, d;
310   m=HEIGHT/2;
311   //Senal de tendencia
312   a=imgSource[(HEIGHT*row)+col];
313   //Senal de fluctuacion
314   d=imgSource[(HEIGHT*(row+m))+col];
315   imgReturn[((row*2)*HEIGHT)+col]= (a+d)/2;
316   imgReturn[(((row*2)+1)*HEIGHT)+col]= (a-d)/2;
317 }
318
319 //Transformda wavelet inversa de Haar de las filas
320 __global__ void HaarInversaFilas(double* imgSource, double* imgReturn)
321 {
322   int col;
323   int row;
324
325   int hiloX = blockIdx.x * blockDim.x + threadIdx.x;
326   int hiloY = blockIdx.y * blockDim.y + threadIdx.y;
327   //calculo de la posicion del hilo en el arreglo
328   int tid = hiloY * (gridDim.x * blockDim.x) + hiloX;
329   col = tid / HEIGHT;
330   row = tid %WIDTH;
331
332   int n;
333   double a, d;
334   n=WIDTH/2;
335   //Senal de tendencia
336   a= imgSource[(row*WIDTH)+col];
337   //Senal de fluctuacion
338   d= imgSource[(row*WIDTH)+(col+n)];
339   imgReturn[(row*WIDTH)+(col*2)] =a+d;
340   imgReturn[(row*WIDTH)+(col*2)+1]=a-d;
341 }
```

Listing 2: C example

B. Características de las tarjetas

B.1. Quadro k420

ACCELERATE YOUR CREATIVITY NVIDIA® QUADRO® K420

Accelerate your creativity with NVIDIA® Quadro®—the world's most powerful workstation graphics.

The NVIDIA Quadro K420 delivers power-efficient 3D application performance and capability. 1 GB of DDR3 GPU memory with fast bandwidth enables you to create complex 3D models, and a flexible single-slot and low-profile form factor makes it compatible with even the most space and power-constrained chassis. Plus, an all-new display engine drives up to four displays with DisplayPort 1.2 support for ultra-high resolutions like 3840x2160 @ 60 Hz with 30-bit color.

Quadro cards are certified with a broad range of sophisticated professional applications, tested by leading workstation manufacturers, and backed by a global team of support specialists, giving you the peace of mind to focus on doing your best work. Whether you're developing revolutionary products or telling spectacularly vivid visual stories, Quadro gives you the performance to do it brilliantly.

FEATURES

- DisplayPort 1.2 Connector
- DisplayPort with Audio
- DVI-I Dual-Link Connector
- VGA Support[1]
- NVIDIA nView™ Desktop Management Software Compatibility
- HDCP Support
- NVIDIA Mosaic[2]

SPECIFICATIONS

GPU Memory	**1 GB DDR3**
Memory Interface	**128-bit**
Memory Bandwidth	**29.0 GB/s**
NVIDIA CUDA® Cores	**192**
System Interface	**PCI Express 2.0 x16**
Max Power Consumption	**41 W**
Thermal Solution	**Ultra-Quiet Active Fansink**
Form Factor	**2.713" H × 6.3" L, Single Slot, Low Profile**
Display Connectors	**DVI-I DL + DP 1.2**
Max Simultaneous Displays	**2 Direct, 4 DP 1.2 Multi-Stream**
Max DP 1.2 Resolution	**3840 x 2160 at 60 Hz**
Max DVI-I DL Resolution	**2560 × 1600 at 60 Hz**
Max DVI-I SL Resolution	**1920 × 1200 at 60 Hz**
Max VGA Resolution	**2048 × 1536 at 85 Hz**
Graphics APIs	**Shader Model 5.0, OpenGL 4.5[3], DirectX 11.2[4], Vulkan 1.0[3]**
Compute APIs	**CUDA, DirectCompute, OpenCL™**

[1] Via supplied adapter/connector/bracket | [2] Windows 7, 8, 8.1 and Linux | [3] Product is based on a published Khronos Specification, and is expected to pass the Khronos Conformance Testing Process when available. Current conformance status can be found at www.khronos.org/conformance | [4] GPU supports DX 11.2 API, Hardware Feature Level 11_0

© 2016 NVIDIA Corporation. All rights reserved. NVIDIA, the NVIDIA logo, Quadro, nView, CUDA, Kepler, and 3D Vision are trademarks and/or registered trademarks of NVIDIA Corporation in the U.S. and other countries. OpenCL is a trademark of Apple Inc. used under license to the Khronos Group Inc. All other trademarks and copyrights are the property of their respective owners.

B.2. GeForce GTX TITAN X

GeForce GTX TITAN X

Especificaciones del motor de la GPU	
Núcleos CUDA	3072
Base Clock	1000 MHz
Boost Clock	1075 MHz
Tasa de llenado de textura	192 GigaTexels/s
Especificaciones de la memoria	
Velocidad de la memoria (Gbps)	7.0
Cantidad de memoria	12 GB
Interfaz de memoria	384-bit GDDR5
Ancho de banda máx.	336.5 GB/s
Características de la tarjeta	
Apta para SLI	✓
Tipo de SLI	4 -way
Entorno de programación	CUDA
DirectX	12
Tipo de bus	PCI-E 3.0

Especificaciones	
Resolución Digital Máxima	5120x3200
Resolución máxima de VGA	2048x1536
Conexión de medios	• HDMI • Dual Link DVI-I • DisplayPort
Múltiples pantallas	✓
HDCP	✓
Audio HDMI	Internal
Dimensiones	
Altura	4.376 inches
Longitud	10.5 inches
Ancho	Dual Slot
Energía y temperatura	
Temperatura máxima	91 C
Consumo de energía	250 W
Requisitos mínimos de energía	600 W
Conexiones de alimentación	6-pin & 8-pin