ESCUELA POLITÉCNICA

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Detección en tiempo real de características poligonales en imágenes mediante la Transformada de Hough 3D

UNIVERSIDAD DE EXTREMADURA

ESCUELA POLITÉCNICA

GRADO EN INGENIERÍA INFORMÁTICA EN INGENIERÍA DE COMPUTADORES

TRABAJO FIN DE GRADO

Detección en tiempo real de características poligonales en imágenes mediante la Transformada de Hough 3D

Autor: José Miguel Moreno Marcelo

Tutor: Pilar Bachiller Burgos

Resumen

La detección de características es un aspecto importante a considerar en cualquier sistema de visión artificial. Una de las técnicas más populares para este fin es la Transformada de Hough Estándar, de la cual han surgido muchas variantes, siendo su principal función la detección de lineas rectas. En este proyecto se ha utilizado una variante denominada Transformada de Hough 3D (HT3D), que se caracteriza por representar segmentos de linea en lugar de lineas infinitas. Haciendo uso de un espacio de características tridimensional, HT3D puede detectar cualquier característica que pueda definirse a partir de segmentos, como esquinas, puntos extremos de segmento o polígonos, además de los propios segmentos.

El uso de un espacio tridimensional implica un coste computacional elevado, lo que hace que HT3D no pueda utilizarse para aplicaciones de tiempo real en su versión secuencial. El objetivo de este trabajo es desarrollar una implementación de HT3D que permita detectar las diferentes características en tiempo real. Para eliminar la dependencia del hardware, la implementación se ha desarrollado utilizando el framework OpenCL. Se presentan pruebas de rendimiento comparando la implementación paralela con la secuencial, así como con distintas unidades de procesamiento gráfico.

Abstract

Feature detection is an important problem to consider in many artificial vision system. One of the techniques more popular for this end is the Standard Hough Transform, being its main fuction the detection of strainght lines. In this proyect it has been used a variant called Hough Transform 3D (HT3D), which is characterized by representing line segments instead of infinites lines. Making use of a theree-dimensional feature space, HT3D can detect any feature that can be defined from segments, like corners, segment enpoints or polygons, in addition to line segments.

The use to a three-dimensional space implies a high computacional cost, which makes that HT3D cannot be used for real time applications in its sequential version. The objective of this work is to develop an implementation of HT3D that provides the detection of the different features in real time. To eliminate the hardware dependencies, the implementation has been developed using the OpenCL framework. Performance tests are presented, comparing the parallel implementation with the sequential one, as well as with diferent graphic processing units.

Índice general

Índice de tablas

Índice de figuras

Capítulo 1

Introducción

1.1. Descripción del problema

El desarrollo de unidades de procesamiento gráfico (GPUs) ha supuesto un avance importante en múltiples campos, entre los que se encuentra la visión artificial. Muchos algoritmos relacionados con el procesamiento y uso de imágenes pueden utilizarse hoy en día con respuestas de tiempo real a pesar de su complejidad computacional.

Entre los algoritmos existentes para detección de características en imágenes, la transformada de Hough es uno de los más utilizados para la detección de líneas. Una de las críticas más extendidas hacia esta técnica y también hacia sus distintas variantes es su alto coste computacional. Sin embargo, esta técnica se basa en la creación y análisis de un espacio alternativo al espacio de imagen (espacio de parámetros), en el que, por cada posición, se lleva a cabo el mismo procesamiento. Esto hace que se trate de una técnica altamente paralelizable y candidata ideal a su implementación en GPUs.

Entre las distintas variantes de la transformada de Hough (TH), la transformada de Hough 3D (HT3D) es una técnica reciente que extiende la aplicación de la TH original a la detección de múltiples características basadas en segmentos de línea (esquinas, segmentos, polígonos). Para ello utiliza un espacio de parámetros de 3 dimensiones, lo que supone una alta carga computacional. En un trabajo anterior [1], se desarrolló una implementación paralela de una primera versión de HT3D utilizando el framework CUDA para gráficas NVidia. En esta versión HT3D permitía detectar esquinas, puntos extremo de segmento, así como segmentos. El objetivo de este trabajo es actualizar dicha versión, extendiendo la detección a polígonos predefinidos y modificando cada fase del algoritmo considerando mejoras añadidas al método propuesto originalmente. Asimismo, para ampliar el uso de esta implementación a otras tarjetas gráficas, la implementación se llevará a cabo utilizando el framework OpenCL. Para validar la implementación, se presenta un análisis de rendimiento que permite comparar la implementación desarrollada con la versión secuencial y con la versión anterior en CUDA utilizando diferentes unidades de procesamiento gráfico.

1.2. Objetivos

Los objetivos planteados en este trabajo son los siguientes:

- Paralelizar todas las fases de HT3D para detección de esquinas, puntos extremos de segmento, segmentos de líneas y rectángulos utilizando el framework OpenCL.
- Construir una aplicación que permita gestionar la ejecución de HT3D a través de una interfaz gráfica, así como obtener resultados de tiempos de las distintas fases para conjuntos de imágenes. Dicha aplicación permitirá al usuario configurar los distintos parámetros que intervienen en esta técnica, así como guardar en fichero los resultados de la detección.
- Realizar un análisis de los tiempos de ejecución de la implementación desarrollada comparando con la versión secuencial y con la versión parcial desarrollada utilizando CUDA para distintas tarjetas gráficas.

1.3. Estructura del documento

El resto del documento está organizado en 5 secciones. La primera sección incluye una explicación detallada de la técnica de detección utilizado, dando una descripción introductoria e incluyendo los distintos algoritmos que intervienen. Como este trabajo se basa en la paralelización de esta técnica, se expone en la segunda sección los distintos aspectos relevantes de la paralelización con GPUs y del framework empleado. La tercera sección describe el proceso de implementación. A continuación, la siguiente sección expone un análisis de los resultados obtenidos con diferentes configuraciones, dispositivos y sistema de paralelización. Por último se incluye una sección de conclusiones y trabajos futuros.

Capítulo 2

Transformada de Hough 3D

2.1. La Transformada de Hough

La Transformada de Hough es una técnica ampliamente usada en procesamiento de imágenes para detectar características tales como lineas, círculos, y elipses. Su popularidad está relacionada con su capacidad de detección ante situaciones desfavorables como pueden ser un ruido moderado en la imagen o la ausencia de partes de las características a detectar debido a las condiciones de iluminación.

En su forma básica, la transformada de Hough es conocida como Transformada Estándar de Hough (TEH) y se utiliza para la detección de líneas rectas. Para ello, la TEH construye un espacio de parámetros, también denominado espacio de Hough, caracterizado por θ y d correspondientes a los parámetros de la ecuación polar de la recta 2D: $d = xcos(\theta) + ysin(\theta)$. A partir de una representación discreta de este espacio de parámetros, los puntos característicos de la imagen, normalmente los puntos de borde, votan por las celdas de esta representación asociadas con las líneas a las que pertenecen. Tras esta fase de votación, la TEH transforma la detección de líneas en una detección de picos en el espacio de parámetros. Así, aquellas celdas cuyo contenido (votos) representa un máximo local en el espacio de Hough definen los parámetros de las líneas presentes en la imagen.

Entre las distintas variantes que han surgido a partir de la Transformada Estándar de Hough, en este trabajo nos centramos en una propuesta reciente denominada HT3D (Transformada de Hough 3D) [2] [3].A diferencia de otras variantes y de la propia TEH, HT3D proporciona la detección de segmentos de línea en lugar de líneas infinitas. Para ello propone un espacio de parámetros tridimensional en el que cada celda almacena el número de puntos característicos de imagen pertenecientes a un segmento. Asimismo, este nuevo espacio tridimensional proporciona una representación canónica de características puntuales de imagen, concretamente, esquinas y puntos extremos de segmento. Así, HT3D no se limita exclusivamente a la detección de segmentos, sino que permite también detectar esquinas y extremos de segmento demostrando mayor precisión y capacidad de detección que otras propuestas. Por último, la estrategia de detección de segmentos puede extenderse a la detección de otras características de imagen que puedan definirse a partir de segmentos de línea.

Concretamente, se ha demostrado su aplicación a la detección de rectángulos. Las siguiente secciones de este capítulo, detallan el método empleado por HT3D en la detección de las características mencionadas: esquinas y extremos de segmento, segmentos y rectángulos.

2.2. La Transformada de Hough 3D

La transformada de Hough 3D (HT3D) propone un espacio de parámetros tridimensional para facilitar la representación de segmentos de línea en lugar de líneas infinitas. El espacio se organiza como un conjunto de planos, donde cada plano representa una orientación de línea (parámetro θ de la ecuación polar de la recta). A su vez, cada columna de un plano se asocia a una distancia determinada de una línea al origen (parámetro d de la ecuación polar de la recta), por lo que cada columna del espacio parametriza de manera completa una línea determinada de la imagen. La principal diferencia con la transformada estándar, es que en HT3D se utilizan varias celdas para representar una línea a través de un tercer parámetro denominado p. Este tercer parámetro (ver figura 2.1) permite definir posiciones relativas dentro de una línea de cada uno de sus puntos. En definitiva, p hace referencia a cada uno de los puntos que pertenecen a la recta parametrizada por θ y d, a través de la siguiente relación:

$$p = -xsin(\theta) + ycos(\theta) \tag{2.1}$$

De manera similar a la transformada estándar, para construir este nuevo espacio, se considera que $\theta \in [0, \pi)$ y d y $p \in [-R, R]$, siendo R la mitad de la diagonal de la imagen y considerando el origen de coordenadas situado en el centro de la imagen.

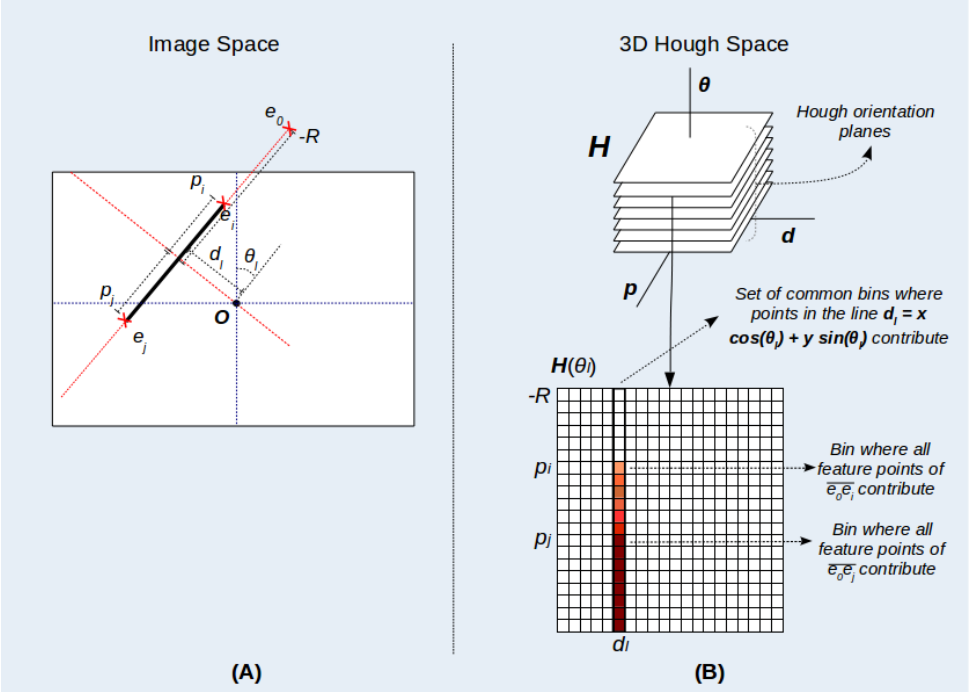


Figura 2.1: **Espacio de Hough 3D (figura extraída de [3])**. (A) Representación de un segmento de la línea $l(\theta_l, d_l)$ en el espacio de imagen; (B) Representación del segmento en el espacio de parámetros. Los puntos de la línea $l(\theta_l, d_l)$ contribuyen a un subconjunto de celdas situada en la columna d_l del plano $H(\theta_l$. Concretamente, un punto con posición relativa p_i en la línea l contribuye al subconjunto de celdas para las que $p >= p_i$.

A pesar de que la nueva dimensión utilizada en HT3D aporta información adicional sobre las posiciones de los puntos característicos de imagen, no es posible representar cualquier segmento de la imagen en una única celda del espacio, ya que para ello sería necesario una cuarta dimensión. Para evitar el uso de una nueva dimensión, HT3D considera que cada celda (θ, d, p) representa un segmento de la línea $l(\theta, d)$ definido por los puntos e y e_0, siendo e un punto con posición relativa a la línea p y e_0 un punto fijo situado dentro de la línea en una posición constante definida por $p_0 = -R$. Teniendo en cuenta esta representación, un punto $e_i = (x, y)$ vota por la celda (θ, d, p) del espacio de Hough si pertenece al segmento $\overline{e_0 e}$ de la línea $l(\theta, d)$, estando el punto e en la posición p de dicha línea. Formalmente, esta relación de pertenencia vendría dada por las siguientes expresiones:

$$d = x\cos(\theta) + y\sin(\theta) \tag{2.2}$$

$$p >= -xsin(\theta) + ycos(\theta) \tag{2.3}$$

Aunque directamente este espacio no representa todos los posibles segmentos, sí permite calcular el número de puntos de imagen pertenecientes a cualquier segmento. Así, dados dos puntos $e_i = (x_i, y_i)$ y $e_j = (x_j, y_j)$ de la línea $l(\theta_l, d_l)$, con posiciones relativas p_i y p_j, asumiendo $p_i < p_j$, el total de puntos pertenecientes al segmento $\overline{e_i e_j}$ viene dado por:

$$H_{i \Leftrightarrow j} = |H(\theta_l, d_l, p_i) - H(\theta_l, d_l, p_j)| \tag{2.4}$$

siendo H el espacio de Hough.

A partir de la expresión anterior, se puede cuantificar la probabilidad de que exista un segmento entre dos puntos e_i y e_j de la siguiente forma:

$$ss(e_i, e_j) = H_{i \Leftrightarrow j} / \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \tag{2.5}$$

Los valores de *ss* (*fuerza de un segmento*) están en el rango entre 0 y 1. Cuanto mayor es el valor de *ss*, mayor es la probabilidad de existencia de segmento entre los dos puntos.

A partir de las medidas anteriores, el método propone la detección de cualquier característica de imagen que pueda ser definida en términos de segmento, ya sean esquinas, segmentos propiamente dichos o incluso polígonos. Las siguientes secciones describen con más detalle las distintas fases empleadas para resolver la detección de las diferentes características.

2.3. El proceso de votación

Para formar el espacio de Hough 3D discreto H, los parámetros θ, d y p se tienen que discretizar asumiendo resoluciones de $\Delta\theta$ para θ, Δd para d y Δp para p. En este espacio discreto, cada punto característico de imagen vota, en cada plano de orientación, por aquellas celdas para las que se verifican las ecuaciones 2.2 y 2.3.

La inecuación de la expresión 2.3 implica que, en cada plano, un punto de imagen debe votar

por varias celdas. Esto supone un elevado coste computacional en la implementación secuencial del algoritmo, por lo que los autores del método proponen dividir el proceso de votación en dos fases:

En la primera fase, los puntos de imagen votan únicamente por el primer segmento al que pueden pertenecer, es decir, p se calcula utilizando exclusivamente la igualdad de la ecuación 2.3.

La segunda fase, conocida como acumulación, parte del segundo valor discreto de $p(p_d = p_{min} + 1)$ y actualiza cada celda $H(\theta_d, d_d, p_d)$ añadiendo a su contenido el contenido de $H(\theta_d, d_d, p_d - 1)$.

Los algoritmos 1 y 2 describen estas dos fases.

Algoritmo 1 Primera fase del proceso de votación.

for cada punto (x,y) perteneciente a un borde de imagen **do**
    for $\theta_d = \theta_{min}..\theta_{max}$ **do**
        Calcular el valor θ asociado con θ_d
        Calcular $d = xcos(\theta) + ysin(\theta)$
        Calcular el valor discreto d_d asociado con d
        Calcular $p = -xsin(\theta) - ycos(\theta)$
        Calcular el valor discreto p_d asociado con p
        Incrementar $H(\theta_d, d_d, p - d)$ en 1
    end for
end for

Algoritmo 2 Segunda fase del proceso de votación (*acumulación*).

if $H(\theta_d, d_d, p_{min}) > \Delta p$ **then**
    $H(\theta_d, d_d, p_{min}) \leftarrow \Delta p$
end if
for $p_d = p_{min} - 1...p_{max}$ **do**
    if $H(\theta_d, d_d, p_d) > \Delta p$ **then**
        $H(\theta_d, d_d, p_d) \leftarrow \Delta p$
    end if
    $H(\theta_d, d_d, p_d) \leftarrow H(\theta_d, d_d, p_d) + H(\theta_d, d_d, p_d - 1)$
end for

En la primera fase es posible reducir el número de iteraciones del segundo bucle utilizando la dirección local del borde para determinar el subconjunto de orientaciones de línea a las que puede pertenecer el punto. Esto reduce el coste computacional, pero, además, permite simplificar el proceso de detección de esquinas como se verá en el siguiente apartado.

En la segunda fase del proceso de votación, como puede observarse, cada celda de Hough se satura a Δp antes de la acumulación. Este paso de saturación obliga a que las celdas que representan un segmento de imagen en el espacio de Hough deban tener una mínima contribución para que el segmento sea detectado.

2.4. Detección de esquinas y puntos extremos

En HT3D se consideran dos tipos de puntos distintivos en una imagen definidos a partir de segmentos: esquinas y puntos extremos. Las esquinas se forman por la intersección de, al menos, dos segmentos. Los puntos extremos son puntos finales de segmento que no pertenecen a otro segmento.

El proceso de detección de los dos tipos de puntos tiene en cuenta que los segmentos de línea de imagen que los definen se transforman en segmentos verticales de celdas en los planos de orientación correspondientes del espacio de Hough 3D. Este hecho da lugar a las configuraciones de celdas de las figuras 2.2 y 2.3. En ambas figuras, las celdas negras representan celdas del espacio de Hough 3D cuyo contenido difiere de la celda situada en la posición anterior de su misma columna. El cambio de contenido implica la existencia de un segmento de línea en imagen en las posiciones asociadas con dichas celdas. Por otro lado, las celdas blancas representan la situación opuesta, es decir, celdas en las que no existe variación de contenido, lo que supone la ausencia de segmento en dichas posiciones de imagen. Por último, las celdas de color gris se asocian con celdas cuyo contenido no tiene relevancia en el proceso de detección. Para cada patrón de punto extremo o esquina, se considera una versión volteada que permite cubrir el rango completo de orientaciones entre 0 y 2π.

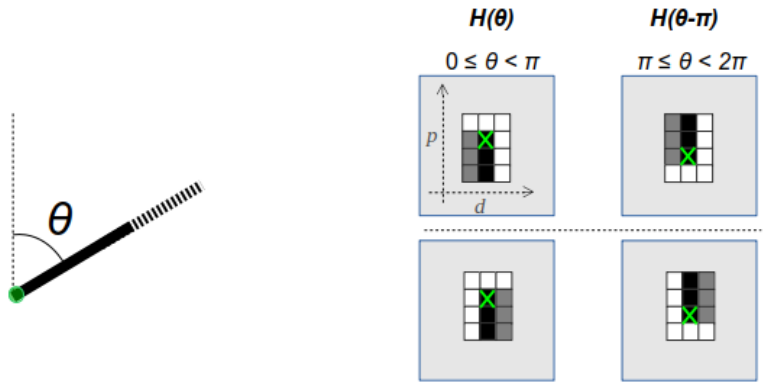


Figura 2.2: **Representación de un punto extremo de segmento en el espacio de Hough (figura extraída de [3])**. La imagen de la izquierda muestra la representación de un punto extremo de segmento en la imagen. La imagen de la derecha muestra la representación equivalente en el espacio de Hough 3D. La celda asociada con el punto aparece marcada con una "x".

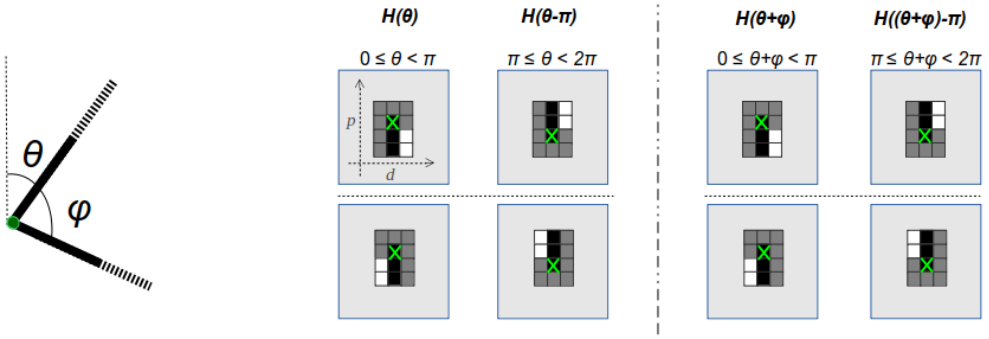


Figura 2.3: **Representación de una esquina en el espacio de Hough (figura extraída de [3])**. La imagen de la izquierda muestra la representación de una esquina en el espacio de imagen y a la derecha su representación equivalente en el espacio de Hough 3D. La celda asociada con el punto aparece marcada con una "x".

En base a estas configuraciones de celdas, el proceso de detección de esquinas y puntos extremos consiste en la verificación de estos patrones en el espacio de Hough 3D. Para ello, las celdas en cada patrón se agrupan en columnas de celdas completas (celdas negras) o vacías (celdas blancas). Una columna completa debe verificar que la diferencia de contenido entre su última y primera celda es superior a un umbral τ_F cercano a 1. Asimismo, para las columnas vacías, debe cumplirse que la diferencia entre sus celdas es menor o igual que un umbral (τ_E)

cercano a cero.

Por ejemplo, para detectar la existencia de una esquina en una posición del espacio de Hough (θ_d, d_d, p_d), tomando η como el número de celdas que representan una columna completa en el patrón de esquina, deben verificarse las siguientes ecuaciones para el plano $H(\theta_d)$:

$$H(\theta_d, d_d, p_d + \eta - 1) - H(\theta_d, d_d, p_d - 1) > \tau_F \tag{2.6}$$

y

$$H(\theta_d, d_d + 1, p_d + \eta - 1) - H(\theta_d, d_d + 1, p_d - 1) <= \tau_E \tag{2.7}$$

o

$$H(\theta_d, d_d - 1, p_d + \eta - 1) - H(\theta_d, d_d - 1, p_d) <= \tau_E \tag{2.8}$$

y las mismas ecuaciones para el plano $H((\theta + \varphi)_d)$, dado un cierto rango de φ y asumiendo θ y $(\theta + \varphi) < \pi$.

El valor de η representa la longitud mínima de segmento que define la esquina o el punto extremo. En el caso de un punto extremo, dicho valor es constante y se fija de acuerdo con la necesitad de detectar segmentos de corta longitud en la siguiente fase. En el caso de las esquinas, el valor de η es variable y dependiente del ángulo φ que define cada posible esquina a detectar. El motivo es que, como resultado de la discretización, para un cierto ángulo de esquina φ, hay una longitud de segmento (l_φ) por debajo de la cual cualquier punto de segmento es detectado como punto esquina. Concretamente, puede demostrarse que dicho valor viene determinado por las siguientes expresiones (ver figura 2.4):

$$l_\varphi = \Delta d / \sin(\varphi/2), \forall \varphi \in [0, \pi/2) \tag{2.9}$$

$$l_\varphi = \Delta d / \sin((\pi - \varphi)/2), \forall \varphi \in [\pi/2, \pi) \tag{2.10}$$

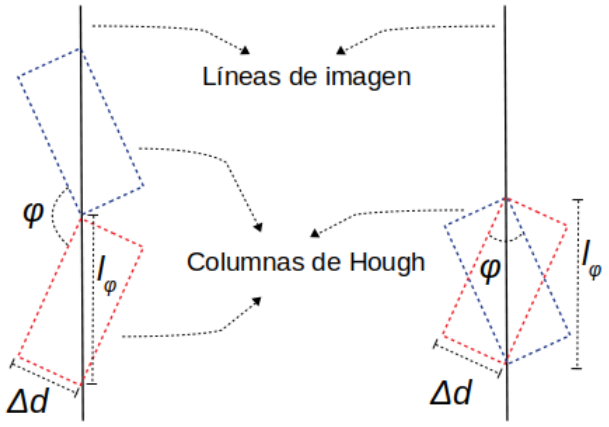


Figura 2.4: Relación entre el ángulo de esquina y la longitud de segmento por debajo de la cuál la detección proporciona resultados erróneos.

Así, cuánto más próximo es φ a $\pi/2$ menor es la longitud de segmento necesaria para detectar una esquina. A medida que el ángulo se aleja de $\pi/2$, se hace necesario ampliar la longitud del segmento que define la esquina.

Al valor de l_φ se le añade una constante l_{ms} para definir la longitud valida de segmento que permite detectar una esquina en un cierto ángulo. Finalmente, puesto que dicha longitud está expresada en píxels, el número de celdas η necesario para verificar un patrón de esquina, se calcula como:

$$\eta = (l_\varphi + l_{ms})/\Delta p \tag{2.11}$$

El proceso de verificación de patrones en todo el espacio de Hough 3D es muy costoso, especialmente en el caso de las esquinas. Así, la detección de una esquina implica la búsqueda del patrón correspondiente en un plano de orientación y la localización del segundo patrón en un rango de planos asociados con los posibles ángulos de esquina. Este coste computacional puede reducirse de 2 formas. La primera es modificando el proceso de votación de manera que cada punto sólo vote por un subconjunto de orientaciones de acuerdo con la dirección local del borde. Esto permite que las esquinas de ángulos aproximadamente rectos se transformen en puntos extremos de segmento sin intersección, que pueden detectarse a partir de los patrones de la figura 2.2. Así, la detección de esquinas se limitaría a un rango de posibles ángulos de esquina

más reducido, esto es, un rango que pudiera definir esquinas de ángulos obtusos y agudos con la apertura y cierre deseados. La segunda forma de reducir el coste computacional es aplicando la detección de patrones en un subconjunto de planos de orientación, en lugar de hacerlo en el total de planos. Así, puede demostrarse que sólo es necesario realizar dicha búsqueda en planos situados a una distancia angular de $\phi = \arctan(1/(\eta \Delta p))$ [2]. No obstante, para asegurar que la detección se lleva a cabo en más de un plano, el valor final utilizado en la implementación de este trabajo es la mitad del ángulo anterior:

$$\phi = \frac{arctan(1/\eta \Delta p)}{2} \tag{2.12}$$

A continuación se muestra el proceso de detección de puntos extremos y esquinas de acuerdo con lo especificado anteriormente. En concreto, la detección de puntos extremos se lleva cabo siguiendo los pasos especificados en el algoritmo 3.

Algoritmo 3 Detección de puntos extremos sin intersección.

```
1:  Inicializar θ_d a θ_min
2:  while θ_d < θ_max do
3:      for cada posición (d_d, p_d) del plano de orientación H(θ_d) do
4:          Comprobar el patrón en su forma normal en la celda (θ_d, d_d, p_d)
5:          if se confirma el patrón then
6:              Almacenar la posición de la celda en una lista de puntos detectados
7:          else
8:              Comprobar el patrón en su forma invertida en la celda (θ_d, d_d, p_d)
9:              if se confirma el patrón then
10:                 Almacenar la posición de la celda en una lista de puntos detectados
11:             end if
12:         end if
13:         Incrementar θ_d de acuerdo con φ (Ecuación 2.12)
14:     end for
15: end while
```

Para la detección de esquinas, en lugar de considerar un rango de posibles ángulos de esquina para la comprobación del segundo patrón, se considera un rango de incrementos/decrementos de ángulos sobre $\pi/2$. De esta forma, la longitud de segmento necesaria para confirmar un patrón (ecuaciones 2.9 y 2.10) se calculan de manera creciente, lo que permite detener el proceso de detección de la celda actual cuando el primer patrón no se

confirma para un determinado incremento de ángulo. Suponiendo que el rango de incrementos de ángulo viene dado por $[\Delta\varphi_{min}, \Delta\varphi_{max}]$, la detección de esquinas en el espacio de Hough 3D se lleva a cabo siguiendo los pasos del algoritmo 4.

Algoritmo 4 Detección de esquinas.

Calcular η para $\varphi = \pi/2 - \Delta\varphi_{max}$
Calcular ϕ_{max} a partir del valor anterior de η
Inicializar θ_d a θ_{min}
while $\theta_d < \theta_{max}$ **do**
    for cada posición (d_d, p_d) del plano de orientación $H(\theta_d)$ **do**
        Inicializar $\Delta\varphi$ a $\Delta\varphi_{min}$
        $corner \leftarrow false$
        $stop \leftarrow false$
        while *not stop* **do**
            Calcular η para $\varphi = \pi/2 - \Delta\varphi$ (ecuación 2.9)
            Comprobar el primer patrón en su forma normal en la celda (θ_d, d_d, p_d)
            if se confirma el patrón **then**
                $\varphi \leftarrow \pi/2 - \Delta\varphi$
                Calcular el ángulo discreto (θ_p) asociado con el ángulo actual (θ_d) más φ
                Calcular la posición de imagen (x_c, y_c) asociada con la celda actual
                Calcular la posición de celda (d_p, p_p) en el plano $H(\theta_p)$ para la posición de imagen (x_c, y_c)
                Comprobar el segundo patrón en la celda (θ_p, d_p, p_p)
                if se confirma el patrón **then**
                    $corner \leftarrow true$
                    $stop \leftarrow true$
                else
                    Repetir el proceso anterior para $\varphi = \pi/2 + \Delta\varphi$
                end if
                Calcular ϕ para el valor actual de η (ecuación 2.12)
                $\Delta\varphi \leftarrow \Delta\varphi + \phi$
                if $\Delta\varphi > \Delta\varphi_{max}$ **then**
                    $stop \leftarrow true$
                end if
            else
                $stop \leftarrow true$
            end if
        end while
        if corner **then**
            Almacenar (θ_d, d_d, p_d) en una lista de puntos detectados
        else
            Repetir el proceso anterior comprobando el primer patrón en su forma invertida para la celda actual.
        end if
    end for
    Incrementar θ_d de acuerdo con ϕ_{max}
end while

Los dos algoritmos de detección generan una lista de posiciones de celda que contienen puntos distintivos (punto extremo o esquina). Tras obtener dicha lista, es necesario localizar las posiciones de los puntos detectados en el espacio de imagen. Para ello, se realiza un proceso de votación desde el espacio de Hough al espacio de imagen. En concreto, por cada posición de celda se calculan las posibles posiciones de imagen asociadas a ella. Las posiciones resultantes votan en el espacio de imagen de acuerdo con su distancia a la posición central de la celda en la imagen (ecuaciones 2.13 y 2.14).

$$x_c = d\cos(\theta) - p\sin(\theta) \tag{2.13}$$

$$y_c = d\sin(\theta) + p\cos(\theta) \tag{2.14}$$

Tras este proceso de votación, las posiciones de los puntos distintivos se localizan finalmente mediante una búsqueda de máximos locales en el espacio de imagen.

2.5. Detección de segmentos

Una vez detectados las esquinas y extremos de segmento, se puede pasar a detectar los segmentos utilizando como indicador la fuerza del segmento calculada a partir de cada pareja de puntos. Para facilitar este proceso, los puntos detectados se almacenan en un array tridimensional (*pointMap*) de acuerdo a sus posiciones en el espacio de Hough. Cada elemento de este array contiene la siguiente información:

1. *endpoint*: indica si la celda correspondiente del espacio de Hough contiene un punto distintivo.
2. *point*: coordenadas de imagen del punto.
3. *distBin*: distancia entre la columna de Hough actual (parámetro d) y la columna real del punto.

La discretización puede provocar que un segmento de línea de imagen se sitúe en dos columnas contiguas dentro del espacio de Hough, en lugar de en una única columna. Es

decir, los puntos finales del segmento se situarían en celdas de columnas diferentes, lo que hace imposible la localización del segmento según la ecuación 2.20. Por este motivo, en el almacenamiento de puntos distintivos en *pointMap*, si la posición de celda de un punto se encuentra cerca de otra columna de Hough (según el valor del parámetro d), dicho punto se almacena también en la columna vecina. Para tener en cuenta el hecho de que un punto no pertenezca con exactitud a la columna en la que se encuentra almacenado, se utiliza el atributo *distBin*. Dicho atributo debe contener la distancia, en posiciones discretas, a la columna que le corresponde realmente. Así, los posibles valores de *distBin* son -1, 0 y 1.

El procedimiento de detección de segmentos se describe en el algoritmo 5. En él se indica el procesamiento que se lleva a cabo para cada celda de *pointMap*. Concretamente, si existe un punto final de segmento en dicha celda, recorre las celdas siguientes de su misma columna para localizar el punto que maximiza el valor de *ss*. Si dicho valor supera un umbral mínimo μ_{ss}, los dos puntos definen un segmento de imagen. Dicho segmento se almacena en un array con las dimensiones de la imagen original (*segmentMap*) que contiene, por cada elemento, una lista con los segmentos (puntos finales que lo definen) "conectados" a ese píxel. Con cada segmento, se almacena también el valor de *ss*.

Algoritmo 5 Detección de segmentos.

if $pointMap(\theta_d, d_d, p_d).endpoint$ **then**
  $e_i \leftarrow pointMap(\theta_d, d_d, p_d).point$
  $b1 \leftarrow pointMap(\theta_d, d_d, p_d).distBin$
  $p2_d \leftarrow p_d + 1$
  $stop \leftarrow false$
  $maxSS \leftarrow 0$
  $insertSegment \leftarrow false$
  while $p2_d < p_{max}$ and $not\ stop$ **do**
    if $pointMap(\theta_d, d_d, p2_d).endpoint$ **then**
      $e_j \leftarrow pointMap(\theta_d, d_d, p2_d).point$
      $b2 \leftarrow pointMap(\theta_d, d_d, p2_d).distBin$
      Calcular $ss(e_i, e_j)$ con $d_d - 1$ y con $d_d + 1$
      if $ss_{d-1} < 0{,}25$ or $ss_{d+1} < 0{,}25$ **then**
        $curSS \leftarrow ss(e_i, e_j)$
        $impSS \leftarrow curSS$
        if $|b_1| \neq |b_2|$ and $curSS < \mu_{ss}$ **then**
          $impSS \leftarrow updateSS(curSS, \theta_d, d_d, p_d, p2_d, b1, b2)$
        end if
        if $impSS > \mu_s$ **then**
          if $curSS >= maxSS$ **then**
            $maxSS \leftarrow curSS$
            $e_{final} \leftarrow e_j$
            $insertSegment \leftarrow true$
          else
            $stop \leftarrow true$
          end if
        else
          $stop \leftarrow true$
        end if
      end if
    end if
    $p2_d \leftarrow p2_d + 1$
  end while
  if $insertSegment$ **then**
    $segmentMap(e_i).add(e_i, e_{final}, maxSS)$
    $segmentMap(e_{final}).add(e_i, e_{final}, maxSS)$
  end if
end if

Como se puede observar en el algoritmo, para confirmar un segmento es necesario que las celdas de las columnas vecinas a ambos lados no definan también un segmento. Esto permite reducir falsos positivos producidos por zonas de altas densidad de borde en la imagen original. Por este motivo, se incluye la comprobación de la línea 13, considerando un valor máximo de *ss* para los segmentos laterales de 0,25. Asimismo, si alguno de los puntos sobre los que se comprueba un segmento pertenece a una columna contigua y el valor correspondiente de *ss* no supera el umbral mínimo, el valor de *ss* es reestimado considerando la información de ambas columnas (línea 17). Dicho valor sólo se utiliza para comprobar si la intensidad de segmento supera el mínimo necesario, pero no es tenido en cuenta en la comprobación de máxima intensidad. De esta forma, si existe otro segmento con el mismo punto inicial cuyo valor de *ss* calculado sobre la columna es mayor, se descartará el estimado con puntos no pertenecientes a la columna actual.

Una vez que se han detectado los segmentos, es necesario llevar a cabo una fase de supresión de no-máximo para descartar aquellos falsos segmentos detectados debido a su cercanía a segmentos reales. Concretamente, en esta fase, se comprueban los segmentos conectados a un mismo punto con orientación similar, eliminando aquellos que muestran un menor valor de *ss*. En la siguiente imagen se muestra el resultado de realizar la detección de segmentos, antes y después de llevar a cabo la supresión del no-máximo. También se puede observar, la formación de polígonos que usan como intersección los puntos finales de los segmentos.

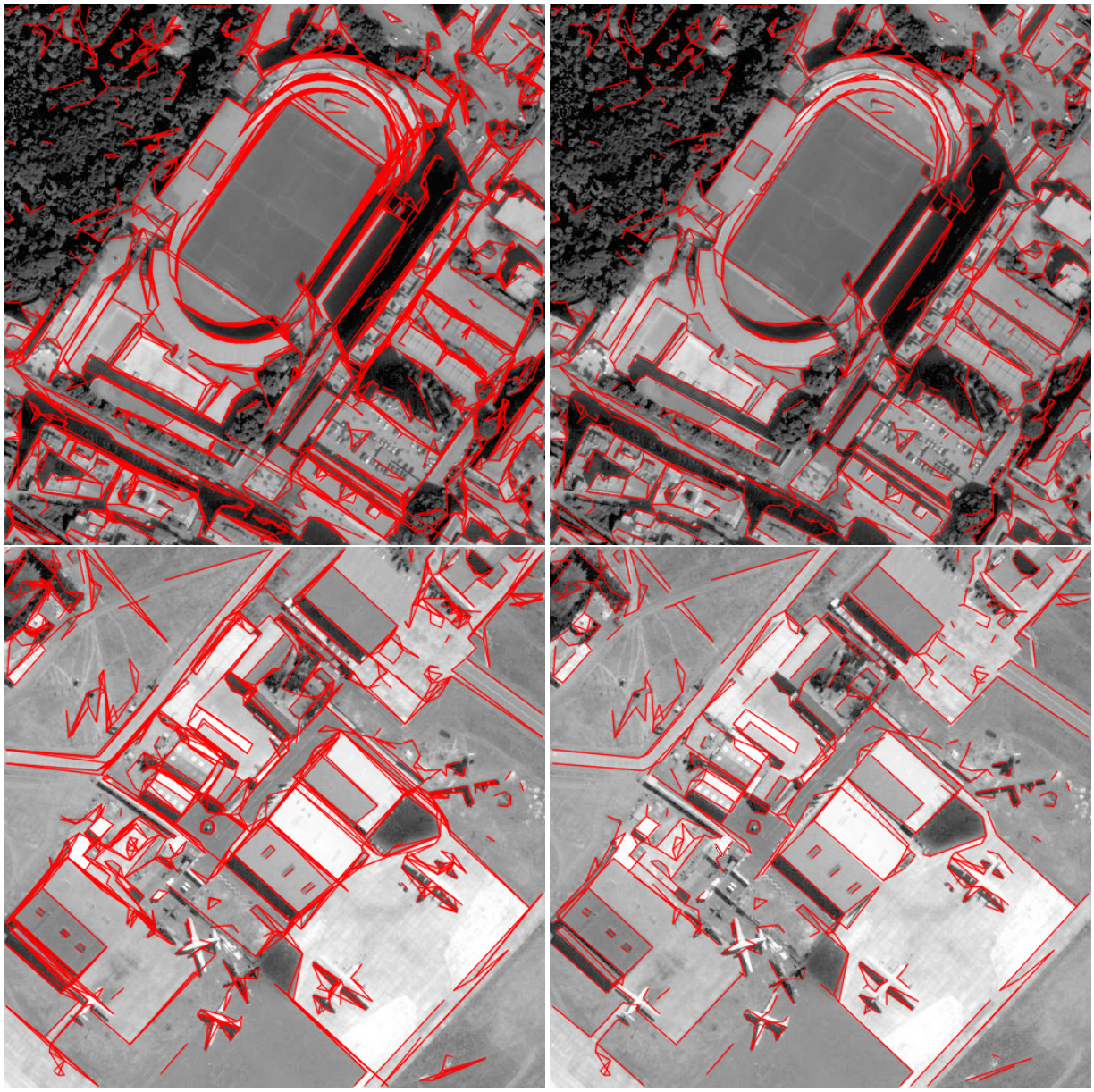


Figura 2.5: **Detección de Segmentos con y sin supresión del no-máximo.** En la imagen de la izquierda se muestra la detección de segmentos ignorando la fase de supresión del no-máximo. Como se puede observar en el resultado, alrededor de cada línea aparecen agrupaciones de varios segmentos. En la imagen de la derecha se aplica una fase posterior de supresión del no-máximo, produciendo segmentos de línea únicos coherentes con los contornos de la imagen.

2.6. Detección de rectángulos

La detección de segmentos a partir de sus puntos finales da como resultado cadenas de segmentos que pueden utilizarse para extraer información sobre los polígonos presentes en la imagen. No obstante, es posible utilizar directamente el espacio de Hough 3D para extraer polígonos predefinidos. En este apartado, este uso para el caso de los rectángulos.

Dado que un rectángulo está formado por cuatro segmentos de línea, es posible estimar el número de puntos de su contorno localizando los 4 segmentos en el espacio de Hough 3D (ver figura 2.6). Así, considerando un rectángulo definido por sus cuatro vértices, V_1, V_2, V_3 y V_4, el número de puntos de su contorno (H_r) puede calcularse como:

$$H_r = H_{1\leftrightarrow 2} + H_{2\leftrightarrow 3} + H_{3\leftrightarrow 4} + H_{4\leftrightarrow 1} \tag{2.15}$$

donde cada $H_{i\Leftrightarrow j}$ denota el número de puntos del segmento definido por V_i y V_j.

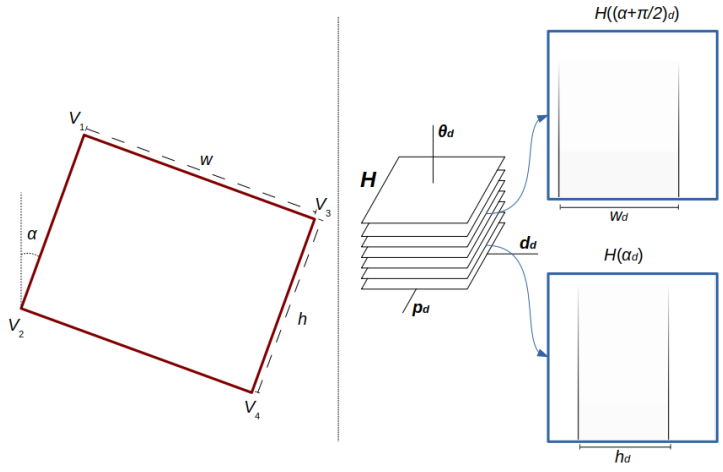


Figura 2.6: Representación de un rectángulo en el espacio de Hough 3D (figura extraída de [2])

La detección de rectángulos en el espacio de Hough 3D parte de la localización del primer segmento ($\overline{V_1V_2}$) siguiendo un proceso similar al indicado en el algoritmo 5. El primer vértice se localiza en la posición (α, d, p). En la misma linea $l(\alpha, d)$ del espacio de Hough 3D se detecta el segundo punto, pero este tiene un rango búsqueda determinado por el parámetro $p = \{p+1, ..., p_{max}\}$ siendo p_{max} el limite de la linea en el espacio de Hough 3D. En este rango se

comprueba todos los puntos que estén dentro de un valor de rango aceptable y con los que pueda formar un segmento, siempre que la fuerza del segmento calculado sea mayor que un umbral μ_{ss1} establecido. Se logra obtener a partir de este segmento, $H_{1\leftrightarrow 2} = |H(\alpha, d, p_2) - H(\alpha, d, p)|$ siendo p_2 la posición en la linea donde está alojado el segundo vértice (V_2).

Para la detección del tercer vértice, se parte desde una linea vecina $l(\alpha, d+n)$ en el mismo plano $H(\alpha)$ tomando para el parámetro p valores en el rango $[p-m, p+m]$, siendo m un valor reducido para asegurar que los vértices definen ángulos aproximadamente rectos. Si en alguna de las celdas establecidas por dicho rango existe un punto final de segmento, dicho punto es tomado inicialmente como tercer vértice del rectángulo. Teniendo en cuenta las propiedades del rectángulo, el cuarto vértice se localiza en la celda $(\alpha, d+n, p_2)$. Si entre ambas celdas se obtiene un valor de intensidad suficiente, se toma $H_{3\leftrightarrow 4} = |H(\alpha, d_n, p_2) - H(\alpha, d_n, p_m)|$, siendo p_m la posición en la línea donde se ha localizado el tercer vértice y $d_n = d+n$.

Con todos los vértices localizados se puede obtener $H_{4\leftrightarrow 1}$ y $H_{2\leftrightarrow 3}$, localizando estos dos segmentos de linea en el plano de orientación $\alpha + \pi/2$. La posición de celda (d', p') en dicho plano puede obtenerse a partir de la posición en el plano de orientación α siguiendo las siguientes transformaciones:

$$d' = p\Delta p/\Delta d \tag{2.16}$$

$$p' = p_{max} - d\Delta d/\Delta p \tag{2.17}$$

para $\alpha < \pi/2$ o

$$d' = d_{max} - p\Delta p/\Delta d \tag{2.18}$$

$$p' = d\Delta d/\Delta p \tag{2.19}$$

para $\alpha >= \pi/2$.

Utilizando estas transformaciones, se obtiene $H_{4\leftrightarrow 1} = |H(\alpha+\pi/2, d', p') - H(\alpha+\pi/2, d'_n, p'_m)|$

y $H_{2\leftrightarrow 3} = |H(\alpha+\pi/2, d'_n, p'_2) - H(\alpha+\pi/2, d', p'_2)|$, pudiendo finalmente calcular H_r.

A partir de H_r es posible cuantificar la fuerza o intensidad del rectángulo (*rs*) utilizando una expresión similar a la que se aplica para estimar la existencia de segmentos. Así, dado un rectángulo r, definido a partir de 3 vértices (V_1, V_2 y, V_3), puede cuantificarse la presencia de dicho rectángulo en la imagen utilizando la siguiente expresión:

$$rs(V_1, V_2, V_3) = H_r/2 \left(\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} + \sqrt{(x_3 - x_1)^2 + (y_3 - y_1)^2} \right) \tag{2.20}$$

Tal y como ocurre con *ss* para estimar la probabilidad de existencia de un segmento, los posibles valores de *rs* se encuentran en el rango $[0, 1]$. Un valor alto de *rs* para una tripleta de vértices, implica una probabilidad alta de presencia del rectángulo correspondiente en la imagen, mientras que valores reducidos de esta medida indican la no existencia del rectángulo.

La figura 2.7 muestra el resultado de aplicar el proceso de detección de rectángulos descrito a dos imágenes reales. Como puede observarse, el método detecta la mayor parte de los contornos rectangulares presentes.

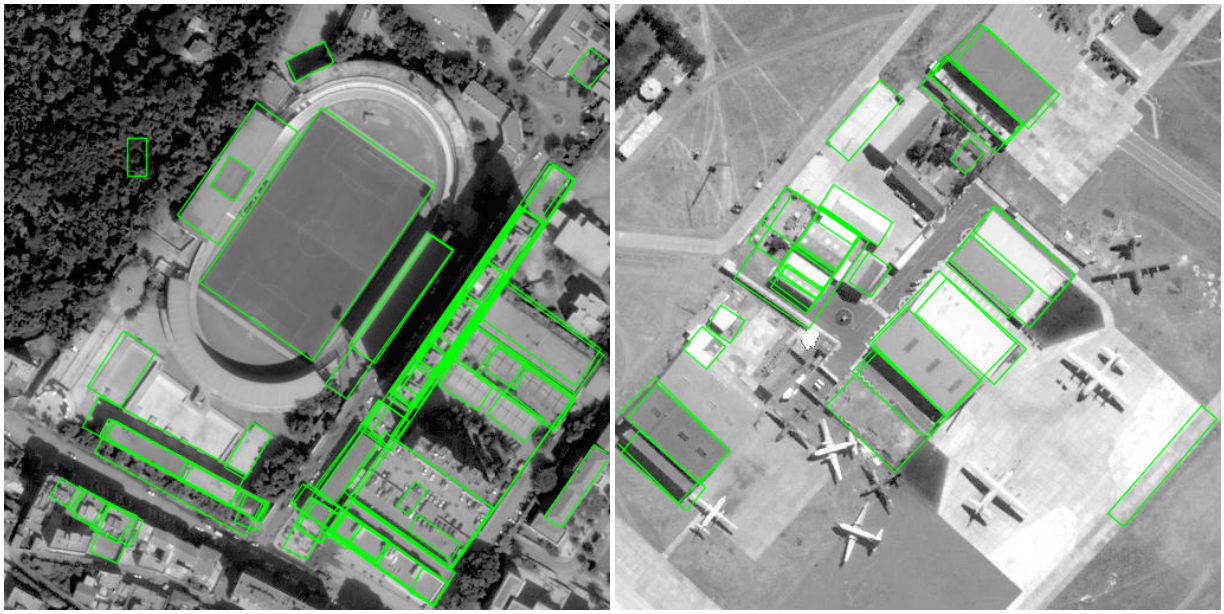


Figura 2.7: **Detección de Rectángulos.** En la figura se muestra el resultado de aplicar el proceso de detección de rectángulos explicado. La cantidad de rectángulos detectados varia según la configuración utilizada.

Capítulo 3

Paralelismo GPU

3.1. Computación Paralela

La computación paralela [4] es un paradigma que logra ejecutar muchas instrucciones simultáneamente. Utilizado para tratar el cómputo de grandes problemas, logra reducir el trabajo en tareas más pequeñas que son ejecutadas o resueltas simultáneamente en paralelo.

Aunque el paralelismo divide la tarea reduciendo el tiempo de computo, surge otro problema relacionado con el diseño y la implementación del código, ya que un programa paralelo es más complejo de escribir que uno secuencial. Así, la concurrencia en su ejecución produce nuevos errores de software, siendo el más común la secuencialidad de los datos en la ejecución y perdida de rendimiento debido a la sincronización y comunicación entre las diferentes subtareas.

Para poder paralelizar los problemas, la computadora dispone de dos unidades dispuestas a realizar procesamientos paralelos. Siendo estos la CPU y GPU. Como en este proyecto está basado en programación paralela orientada a la GPU sólo se centra en la explicación de esta.

El paralelismo en GPU se conoce como computación de propósito general en unidades de procesamiento gráfico GPGPU [5] (del inglés purpose computing on graphics processing units). Las GPUs son unidades con gran potencia de cálculo, gran paralelismo y con capacidad para realizar operaciones en coma flotante optimizadas. Pero no siempre es eficiente la paralelización con la GPU, sobre todo en los problemas con accesos a memoria, ya que el acceso está más restringido que en la CPU. Para solucionar este inconveniente se adaptan los accesos a memoria y las estructuras de datos, a las características de la GPU.

Aunque se puede implementar la ejecución de cualquier problema tanto en la CPU como en la GPU, esto no indica que sea igual de eficiente en los dos sistemas, centrándonos en los algoritmos paralelizados, con estructuras sencillas y con un alto porcentaje de cálculos aritméticos, se obtiene mejor rendimiento con la GPU.

Desde un principio, el desarrollo del software para la GPU se realizaba en lenguaje ensamblador

o en otros lenguajes alternativos como GLSL, Cg o HLSL. En la actualidad se han desarrollado herramientas que facilitan y mejoran la implementación de aplicaciones en la GPU realizando este desarrollo a más alto nivel. Entre las opciones destaca CUDA de Nvidia, una extensión de C que permiten la codificación de algoritmos en las GPU's de Nvidia. Otra alternativa semejante a CUDA es OpenCL, una combinación de interfaz y lenguaje de programación para la implementación de programas paralelos, pudiéndose desarrollar en unidades de procesamiento como CPU multinúcleo, GPU, etc. Siendo esta la alternativa utilizada en este trabajo.

3.2. OpenCL origen y actualidad

OpenCL [6] fue desarrollado inicialmente por Apple Inc. Con la colaboración de los equipos técnicos de AMD, IBM, Qualcomm, Intel y Nvidia lograron refinarlo. Apple presentó su propuesta inicial al Grupo Khronos. Este Grupo trabajó para refinar los detalles técnicos de la especificación para OpenCL 1.0, aprobando su lanzamiento público el 8 de diciembre de 2008.

Desde esta fecha el Grupo Khronos ha ido publicando diferentes versiones en las cuales ha realizado ratificaciones y mejoras. En la actualidad la versión más reciente de OpenCL es la 2.2 (lanzada en noviembre de 2017 y actualizada en mayo de 2018), aunque para este mismo año es probable que esté disponible una nueva versión con nuevas características.

La versión actual presenta las siguientes características:

- El lenguaje del kernel OpenCL C++ es un subconjunto estático del estándar C++ 14.
- Khronos proporcionar su nuevo lenguaje SPIR-V 1.1 que es compatible con OpenCL C++.
- Para el uso de las librerías permite usar el lenguaje C++ para evitar problemas con las funciones atómicas, iteradores, imágenes, muestreadores, tuberías y tipos de colas de dispositivos integrados y espacios de direcciones.

- El almacenamiento de tuberías hace que el tamaño y el tipo de conectividad se conozcan en el momento de la compilación, lo que permite una comunicación eficiente entre los kernel.

- Incluye optimizaciones en la generación de código.

- Es compatible con el hardware que soporta OpenCL 2.0.

Pero esta versión sólo es compatible con los dispositivos más modernos. La versión más establecida es la 1.2 siendo la más soportada por todos los dispositivos, pudiendo utilizar sus funciones por cualquier versión superior.

La versión 1.2 se lanzó el 15 de noviembre del 2011 y entre sus características más notables incluye:

- Poder dividir el dispositivo en sub-dispositivos logrando concentrar el trabajo en unidades de computo individuales reduciendo la latencia.

- Compilación separada y vinculación de objetos.

- Agrega soporte para imágenes 1D y matrices de imágenes 1D / 2D. Además, las extensiones para compartir OpenGL ahora permiten el uso de texturas OpenGL 1D y matrices de textura 1D / 2D para crear imágenes OpenCL.

- Permite utilizar en los Kernels las funciones especificas únicas que proporcionan los dispositivos dentro del marco de OpenCL.

- Funcionalidad de DirectX permitiendo la compartición perfecta entre las superficies OpenCL y DX11.

- La capacidad de forzar el cumplimiento de IEEE 754 para las matemáticas de punto flotante de precisión simple.

3.3. Modelo de memoria

El modelo de memoria [7] de OpenCL es muy parecido al modelo de memoria de una GPU. A continuación, se hace una comparativa visual con el modelo de memoria de una GPU AMD Radeon HD 6970.

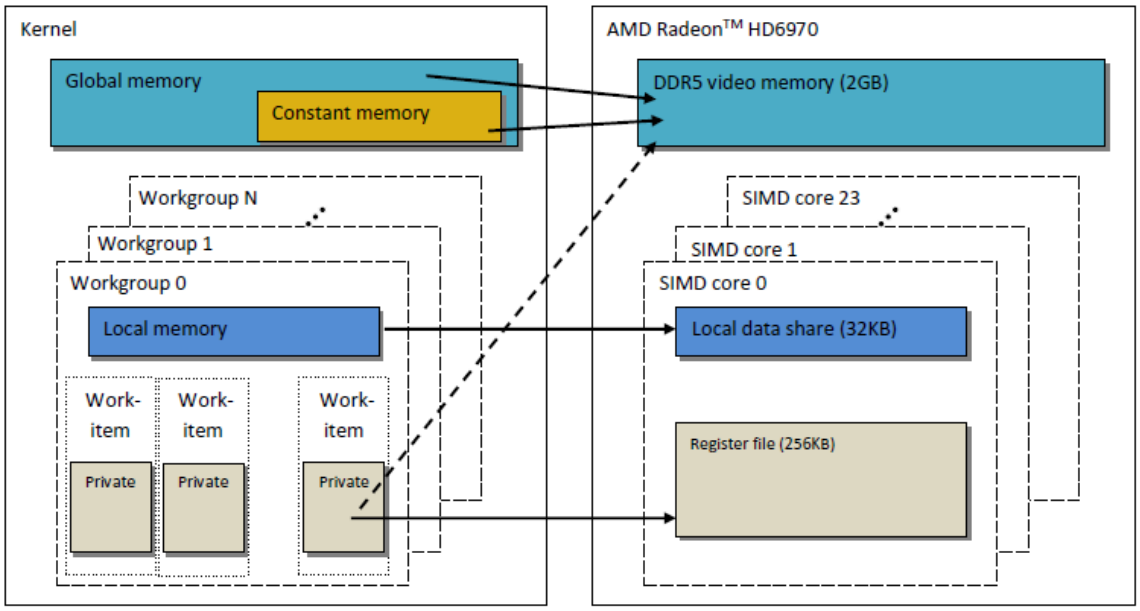


Figura 3.1: **Modelo de memoria (figura obtenida de [7])**. Como se puede apreciar en la imagen, aunque OpenCL crea su propia estructura de trabajo es muy similar a la que viene integrada en las GPU's, pudiendo diferenciar zonas de memoria global, local y privadas.

Como se aprecia en la imagen, el modelo está estructurado en cuatro tipos de memoria utilizadas en zonas de trabajo diferentes, las cuales son:

- La **memoria global** es una zona de memoria accesible por todas las unidades de trabaja, es como la memoria RAM del sistema anfitrión.

  En OpenCL se define con la palabra clave *"__global"* y se considera a la variable que lo usa como un puntero a la zona de memoria global. Se expresa en el kernel de la siguiente forma:

  __kernel void fuction(__global float *A) { /// cuerpo del kernel }

- La **memoria constante** contiene valores constantes de sólo lectura y es una zona de

memoria en la que se puede acceder de forma simultanea por diferentes unidades de trabajo. En OpenCL la memoria constante está alojada en la propia memoria global compartiendo la misma zona de memoria, pero esta es denominada con la palabra clave *"__constant"*. Se expresa en el kernel de la siguiente forma:

__kernel void function(__constant float *B) { /// cuerpo del kernel }

- La **memoria local** es una memoria auxiliar alojada en la unidad de trabajo y es única para cada dispositivo. Hay hardware que tienen un chip reservado para ella, pero esto no indica que OpenCL lo pueda utilizar para la misma finalidad.

  La memoria local es accesible únicamente para un grupo de trabajo determinado, cada grupo tiene la suya propia, descrita por la palabra clave *"__local"*. El acceso a la memoria local es mucho más rápida que la memoria global, ya que el tiempo de espera es menor y el ancho de banda es mucho mayor. Gracias a que tiene un menor tiempo de repuesta consigue optimizar el kernel. Este tipo de memoria se puede colocar tanto en el cuerpo del kernel como por indicación en sus parámetros como las anteriores:

  __kernel void fuction(__local float * variable) { /// cuerpo del kernel }

  o también

  ```
  __kernel void fuction( __global float *A )
  {
  /// cuerpo del kernel
  __local float vector[64];
  }
  ```

- La **memoria privada** es usada en el cuerpo del kernel, siendo única para cada unidad de trabajo y no puede ser accesible por las demás. En esta categoría se introducen las variables locales y argumentos del kernel no definidas como punteros, si no se especifica con el indicador *"__local"* o *"local"*.

Como los sistemas de memoria de las CPU y la GPU son diferentes, si se desea trabajar con las GPU's hay que realizar una portabilidad del código y así poder utilizar el hardware. Para esto OpenCL define una estructura de memoria que se adapta a las características de todos los dispositivos. Pudiendo realizar la migración de datos entre la CPU y GPU y viceversa de forma segura, alojando los datos transferidos en sus correspondientes zonas de memoria global.

En la programación de las GPU's existe un problema referentes a los registros, como el dispositivo esta formado por un mayor número de núcleos en una zona de chip reducida queda poco espacio para agregar zonas de memoria grandes, con lo cual la cantidad de registros están limitados.

3.4. Entorno OpenCL

3.4.1. Plataformas y dispositivos

Una plataforma se define como la conexión generada por un host con uno o más dispositivos OpenCL. Un dispositivo es una unidad de cómputo que puede dividirse en una o varias unidades, que a la vez se dividen en uno o más elementos de procesamiento.[8]

Cada plataforma esta definida por un modelo el cual indica los roles que tendrán el host y los dispositivos. Este modelo es esencial para realizar una portabilidad entre sistemas compatibles y desarrollar aplicaciones en OpenCL que puedan adaptarse y elegir de esta manera la mejor plataforma y dispositivos para la ejecución.

Los modelos de plataforma [9] proporcionan un modelo de hardware abstracto para los dispositivos que se dirigen los programadores, cuando implementan en lenguaje C sobre OpenCL. Esta arquitectura es asignada por los proveedores en el hardware físico, definiendo el modelo en el dispositivo como un grupo de múltiples unidades de computo, donde cada unidad es independiente y divida en elementos de procesamiento. Como se puede apreciar en la imagen siguiente.

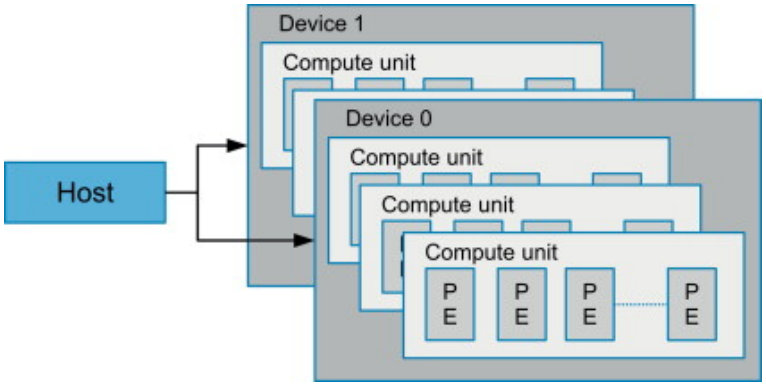


Figura 3.2: **Modelo de Plataforma (figura obtenida de [9])**. Muestra como es la estructura de una plataforma, representado como un host conectado con varios dispositivos (en esta imagen con dos). Y cada dispositivo se divide en varias unidades de computo, dentro de cada unidad se divide en elementos de procesamiento.

Las plataformas son consideradas por las aplicaciones como interfaz que tienen en común todos los dispositivos, siendo especificada por el proveedor. Estas especificaciones provocan la limitación de dispositivos que puedan dirigirse a la plataforma, ya que sólo pueden actuar aquellos dispositivos que estén indicados en las especificaciones. Pero también existe la excepción de exclusividad de una plataforma para un proveedor, se puede desarrollar una aplicación OpenCL que esté implementada en la misma plataforma con los dispositivos de AMD e Intel y poder interactuar entre ellos.

Una vez definido que es una plataforma y el uso de los dispositivos en ella, se indica su funcionamiento en OpenCL. El procedimiento a seguir es usar la función *clGetPlatformIDs()*, con esto OpenCL proporciona la capacidad de descubrir e identificar las distintas plataforma que existen en un sistema. La utilización de esta instrucción se basa en dos parte, la primera obtiene el número de plataformas disponibles, usando para este propósito un puntero de entero sin signo con el cual se almacena el número total de plataformas, y la segunda parte, activa una plataforma concreta que ha sido identificada anteriormente al obtener el número total de ellas. Si se quiere obtener información sobre las plataforma, OpenCL proporciona la función *clGetPlatformInfo()*.

Ya descubiertas y activadas las plataformas en el sistema hay que identificar la cantidad de dispositivos disponibles. En este paso, OpenCL proporciona la instrucción *clGetDeviceIDs()*

realizando la misma funcionalidad que *clGetPlatformIDs()* pero con resultados distintos, se realizan los mismos pasos, primero localizar y después activar los dispositivos que se van a utilizar. Pero con la diferencia de que *clGetDeviceIDs()* usa la plataforma activada y tipo de búsqueda a realizar, sólo GPU's, sólo CPU o todo los dispositivos. También para obtener información sobre los dispositivos se proporciona la función *clGetDeviceInfo()*.

3.4.2. Contexto, cola y programa

El contexto [10] en OpenCL es una interfaz creada para que los diferentes dispositivos del sistema puedan comunicarse entre sí durante la ejecución del programa, asignándose a una plataforma. También se encarga de ocultar todos los detalles referentes a los dispositivos sobre la información de bajo nivel a usar y no puede ser manipulada por el programa.

Una vez creado el contexto, este permite crear colas de comandos, programas que se pueden ejecutar en uno o más dispositivos, kernels, reservar memoria o imágenes, mandar datos a los diferentes dispositivos, ejecutar los kernels, leer los datos que se obtienen de la ejecución, etc.

Para poder crear un contexto, OpenCL proporciona la función *clCreateContext()* asignando unas propiedades para su creación, se proporcionan dos tipos de propiedades:

- CL_CONTEXT_PLATFORM especifica la plataforma a utilizar.
- CL_CONTEXT_INTEROP_USER_SYNC especifica si el usuario es el responsable de la sincronización entre OpenCL y otras API.

Y por último, se asigna la lista de dispositivos localizados en la plataforma y disponibles para actuar en la ejecución OpenCL.

La cola de comandos [11] es utilizada para operar en OpenCL sobre los programas e interactuar en los kernel implementados, usando dentro de estos los objetos de memoria previamente reservados en el dispositivo. La función principal de la cola de comando es añadir a una cola el conjunto de operaciones a utilizar, indicando el orden de ejecución.

Estas colas están creadas y localizadas dentro del contexto, pudiendo existir varias de ellas.

Si se crea más de una se consigue que las aplicaciones puedan ejecutar varios comando simultáneamente, pero con la condición de no poder compartir los objetos de memoria. Si se da el caso del intercambio de datos, es trabajo de la aplicación, realizar la sincronización entre ambas colas.

Para crear una cola de comandos, OpenCL proporciona la instrucción *clCreateCommandQueue()* indicando el contexto al que pertenece y los dispositivos en los que va ha actuar. También, se indica la propiedad con la que se creará la cola de comando, existiendo dos propiedades:

- CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE obliga a la cola de comando mandar ejecutar las operaciones en orden de entrada, si no se establece, la cola se ejecuta de forma aleatoria.
- CL_QUEUE_PROFILING_ENABLE habilita el perfilado del comando en la cola de comandos, si no se indica está deshabilitado.

Un objeto de programa [12] OpenCL, esta formada por un conjunto de kernels definidos en el código del programa e identificados por la nomenclatura "__kernel". A parte de los kernels, considerados como funciones públicas o externas, también puede tener desarrollado otras funciones auxiliares e incluso tener definidos datos constantes que completan a uno o varios kernels.

En la creación de un objeto de programa, OpenCL proporciona la herramienta *clCreateProgramWithSource()* siendo una de las ofrecidas y existiendo diferentes para este propósito. Con la cual se crea el objeto a partir del código fuente alojado en un archivo con extensión *"*.cl"*, en su interior contiene la implementación de los kernels más las diferentes funciones que los acompañan.

Con todos los recursos disponibles para crear el objeto de programa, se inicia su creación de forma online u outline destinado a los dispositivos apropiados, realizada esta tarea por el compilador de OpenCL. Se usa para este propósito la instrucción *clBuildProgram()*, comprobando que el código fuente es correcto. En situación de error se opta la posibilidad

de obtener dicha información a partir de la función *clGetProgramBuildInfo()*, mostrando los problemas detectados en la compilación.

Creado el objeto de programa,en él esta encapsulado un contexto asociado, la fuente de programa o binario y el ejecutable, además de todo lo requerido para su correcta ejecución como bibliotecas, opciones de compilación,etc.

3.4.3. Kernel

Como se ha mencionado en el apartado anterior, un kernel [13] es una función pública que esta implementada en el código del programa OpenCL, identificada por el calificador *"__kernel"*. La función del kernel es permitir a la aplicación poder ejecutar de forma paralela el código, logrando procesar de forma más rápida matrices de datos e incluso dividir el procesamiento de cálculos entre varias instancias, reduciendo así su tiempo de cálculo y obtener un resultado final tras la acumulación de cada hilo al terminar.

Los datos a usar por el kernel son trasferidos a través de sus argumentos localizados según su uso en memoria global, constante o local. Se almacenan en memoria privada aquellos datos externos al entorno OpenCL siendo transferidos en el momento de la ejecución desde el programa del host anfitrión.

En la creación de los objetos kernel, OpenCL proporciona la herramienta *clCreateKernel()*. Sólo se requiere para su creación, la instancia del programa OpenCL y el nombre impuesto al kernel definido en el código del programa. El resultado es un objeto *cl_kernel* utilizado en la ejecución de la aplicación.

3.4.4. Creación del entorno

OpenCL es un estándar abierto multiplataforma para la programación paralela. Permite acceder a múltiples procesadores de forma simultanea, pero para poder utilizar los distintos dispositivos con este framework se tiene que realizar una adaptación, el cual OpenCL proporciona para este propósito la creación de un entorno de trabajo, describiéndose los pasos

que se deben seguir a continuación.

Antes de usar las funciones descritas en los apartados anteriores hay que realizar la preparación de las bases del entorno, siendo estos los distintos objetos que se deben crear. Se comienza con la declaración de cada objeto, reservando espacio de memoria en el host anfitrión para aquellos que son necesarios, como es el caso de la plataforma y los dispositivos.

En su declaración, OpenCL indica en su manual los distintos tipos de objetos a utilizar en el estándar, aquí sólo se mencionan aquellos que se utilizan en el proyecto y en este caso en la creación del entorno. Los tipos de objetos a utilizar son:

- *cl_platform_id* tipo destinado para recoger las plataforma disponibles en la computadora, se declara como puntero y reserva memoria en el host.
- *cl_device_id* recoge los distintos dispositivos compatibles con la plataforma creada, como en el anterior, se tiene que establecer como puntero y reservar memoria.
- *cl_context* tipo que alberga la instancia del contexto OpenCL.
- *cl_command_queue* tipo de objeto destinado a la cola de comandos, el cual puede utilizarse para crear varias colas de comandos a utilizar en la aplicación o en el entorno OpenCL.
- *cl_program* tipo objeto usado para crea una instancia cuyo contenido ha recogido el código a utilizar por el entorno OpenCL.
- *cl_kernel* tipo de objeto utilizado para crear instancias de los distintos kernel implementados en el código.

A continuación, se muestra como se ha realizado la declaración de cada tipo en el proyecto.

```
/*Entorno opencl*/
cl_platform_id *platforms;
cl_device_id *devices;
cl_context context;
```

cl_command_queue queue;
cl_program program;

/*Funciones kernel*/
cl_kernel setTo0Device, votingHoughSpace, accumulateHoughSpace;
cl_kernel detectEndpointsFirst, detectEndpointsSecond, detectImageCorners;
cl_kernel setToNegCornerMap, transformCornersToHough, detectSegmentsSinMutex, validSegmentAfterNMS;
cl_kernel detectRectangles, validateRectangles;

Para la reserva de memoria, es obligatorio ejecutar antes *clGetPlatformIDs y clGetDeviceIDs*, obteniendo el número de plataformas y dispositivos, pudiendo así reservar correctamente su zona de memoria.

platforms = (cl_platform_id*)malloc(sizeof(cl_platform_id)*num_platforms);

devices = (cl_device_id*)malloc(sizeof(cl_device_id)*num_devices);

Definidos los objetos, se procede a su creación en orden, utilizando las instrucciones descritas en los subapartados del ***"Entorno OpenCL"***. También se indica como es la estructura de cada instrucción.

El primer paso es localizar las distintas plataformas existentes en la computadora, como se ha descrito, se debe de usar la función *cclGetPlatformIDs()* dos veces, la primera para obtener y la segunda para crear.

clGetPlatformIDs(cl_uint *num_entries*, cl_platform_id **platforms*, cl_uint **num_platforms*)[14]
num_entries número de plataformas a crear.
platforms puntero de la plataforma del sistema.
num_platforms número de plataformas disponibles.

Se comienza con la obtención del número de plataformas, usando un tipo dato *"cl_uint"*

equivalente a *"unsigned int"* obteniendo el total de plataformas, en la primera ejecución de la instrucción los dos primeros parámetros no se toman en cuenta siendo nulos en la ejecución.

err = clGetPlatformIDs(0, NULL, &num_platforms);

Una vez tomado el número de plataforma, se pasa a su creación. En este caso se anula el tercer parámetro, añadiendo la cantidad de plataformas detectadas y el puntero a plataforma.

err = clGetPlatformIDs(num_platforms, platforms, NULL);

La función retorna un valor *"cl_int"* equivalente a *"int"*, indica si a ocurrido un error y cual es.

Con la plataforma ya iniciada, hay que comprobar la existencia de dispositivos compatibles y lograr obtener cada uno de ellos. El procedimiento es el mismo que en la plataforma, primero se recoge la cantidad de dispositivos y despues se inicia uno o varios.

clGetDeviceIDs(cl_platform_id platform, cl_device_type device_type, cl_uint num_entries, cl_device_id *devices, cl_uint *num_devices)[14]
platform plataforma previamente creada.
device_type tipo de dispositivo a buscar.
num_entries cantidad de dispositivos a iniciar.
devices puntero que contienen los dispositivos encontrados en la plataforma.
num_devices obtiene el número de dispositivos compatibles con la plataforma.

Con el primer uso de esta instrucción, se localiza la cantidad de dispositivos compatibles. Para esto se debe de proporcionar la plataforma y el tipo de dispositivos a buscar, siendo en el caso del proyecto "CL_DEVICE_TYPE_GPU" (esta opción busca sólo los dispositivos GPU ignorando los demás). El tercer y cuarto parámetro no se toman en cuenta y el último se añade una variable de tipo *cl_uint* obteniendo el número total de dispositivos encontrados.

err = clGetDeviceIDs(*platforms, CL_DEVICE_TYPE_GPU, 0, NULL, &num_devices);

Para iniciar cada dispositivo, se vuelve a ejecutar la misma función. En este caso se anula el

último parámetro y se añade al tercero y cuarto la cantidad de dispositivos encontrados y el puntero de tipo *cl_device_id*.

err = clGetDeviceIDs(*platforms, CL_DEVICE_TYPE_GPU, num_devices, devices, NULL);

NOTA: Como se ha indicado en la reserva de memoria, para reservar espacio se puede realizar de dos formas, tomando un orden diferente cada una. El primer caso, sería ejecutar primero la primera parte de estas dos funciones, a continuación reservar memoria con el número total de plataformas y dispositivos, y después ejecutar la creación de la plataforma y los dispositivos. O el segundo caso, buscar primero las plataformas, reservar memoria para el objeto plataforma, crear la plataforma, buscar los dispositivos, reservar memoria para los dispositivos y por último iniciarlos.

El siguiente paso a seguir es la creación del contexto de trabajo OpenCL. Primero se tiene que indicar las propiedades del contexto, para esto, se declara un vector de 3 posiciones de tipo *cl_context_properties* que contiene la propiedad CL_CONTEXT_PLATFORM y el puntero a plataforma, el último valor se pone a 0.

clCreateContext(cl_context_properties *propeties, cl_uint num_devices, const cl_devide_id *devices, void *pfn_notify, void *user_data, cl_int *errcode_ret)[14]
propeties propiedades del contexto.
num_devices número de dispositivos iniciados.
devices dispositivos iniciados.
pfn_notify retorna información sobre errores en la generación del contexto.
user_data datos de usuario usados con el parámetro anterior, normalmente puesto a NULL.
errcode_ret devuelve el código de error.

En este trabajo se ha creado el contexto usando los tres primeros parámetros y el último para identificar errores. Siguiendo el orden de los parámetros se añade las propiedades para el contexto, el número de dispositivos y el puntero a los dispositivos, el resto de parámetros se pone a NULL excepto errcode_ret necesario para obtener el error usando una variable *"cl_int"*.

Esta función retorna el contexto creado.

context = clCreateContext (properties, num_devices, devices, NULL, NULL, &err);

Hasta aquí, es requerido seguir el orden indicado. Ya que es un procedimiento en cadena y cada objeto requiere del anterior. Lo último de crear es: la cola, el programa y los kernel. El orden no importa, excepto los kernel, siendo obligatorio disponer del objeto programa.

En el caso de la cola necesita para su creación el contexto al que pertenece, el dispositivo asignado, el tipo de orden de ejecución (ya explicado anteriormente) y una variable *cl_int* encargada de recoger el valor de un posible error. Esta función retorna como valor la cola de comando.

clCreateCommandQueue(cl_context context, cl_device_id device, cl_command _queue_properties properties, cl_int *errcode_ret)[14]
context contexto OpenCL.
device dispositivo asociado al contexto.
properties tipo de orden en la ejecución en la cola de comando.
errcode_ret indica el código de error.

queue = clCreateCommandQueue (context, devices[0], CL_QUEUE_PROFILING _ENABLE, &err);

El programa es un objetos destinado en OpenCL a contener el código a ejecutar, este se obtiene a partir de un fichero, almacenando su contenido en una cadena de tipo char dinámico, convirtiéndose en un argumento a añadir. Esta función retorna como valor el objeto programa OpenCL.

ClCreateProgramWithSource(cl_context context, cl_uint count, const char **strings, const size_t *lengths, cl_int *errcode_ret)
context contexto OpenCL.
count número de cadenas disponibles.

strings código del programa.
lengths tamaño de cada cadena pasada por el parámetro strings. Si es NULL se considera que las cadenas de caracteres acaban en NULL.
errcode_ret código del error.

Los parámetros a introducir en orden son: el contexto, el número de cadenas de código utilizadas siendo en este caso 1, la cadena del código, como no se indica el tamaño se coloca a NULL y por último la variable *"cl_int"* para recoge el código de error.

program = clCreateProgramWithSource(context, 1, (const char **)&code, NULL, &err);

Como cualquier programa, una vez disponible el código fuente se debe compilar y comprobar que esta libre de errores. Realizandose en OpenCL este paso en tiempo de ejecución.

clBuildProgram(cl_program program, cl_uint num_devices, const cl_device_id *device_list, const char options, void (pfn_notify)(cl_program, void *user_data), void *user_data)[14]
program objeto del programa OpenCL.
num_devices número total de dispositivos que contiene la lista.
device_list lista de todos los dispositivos asociados al programa.
Options opción de compilación.
pfn_notify puntero a una rutina de notificación. Si es NULL se espera a que el procedimiento acabe de compilar.
user_data dato de usuario pasado como argumento cuando pfn_notify no es NULL.

La compilación es necesaria, desvela posible errores en la implementación del código. En este proyecto sólo se le dá uso a sus tres primeros parámetros, los tres últimos no se usan. Para una correcta compilación se requiere del programa, el número total de dispositivos iniciados y asociados, y el puntero a dispositivos.

err = clBuildProgram (program, num_devices, devices, " ", NULL, NULL);

Por último los kernel. Por cada función implementada en el código del programa con el

calificador __kernel, se debe declarar un objeto en el cual se crea una instancia pudiendo realizar la llamada a la función.

clCreateKernel(cl_program program, const char *kernel_name, cl_int *errcode_ret)[14]
program programa construido con éxito.
kernel_name nombre del kernel en el código fuente.
errcode_ret código de error.

Esta función retorna como valor el objeto kernel, con la finalidad de poder realizar la llamada a la función. En la inicialización de la instancia se asigna por parámetros el programa creado y compilado, el nombre de la función a ejecutar como kernel y una variable *"cl_int"* que obtiene el código de error en caso de haberlo.

detectRectangles = clCreateKernel(program, "detectRectangles", &err);

3.5. Entorno de trabajo

OpenCL es un framework destinado a trabajar de forma paralela en dispositivos de cómputo, creado para reducir grandes problemas, pudiendo definir espacios de trabajos con identificación de zonas por índices, considerando cada espacio de trabajo global como una dimensión. OpenCL dispones de tres dimensiones de trabajo, considerando a cada una de ellas como:

- Dimensión 1 Array.
- Dimensión 2 Imágenes.
- Dimensión 3 Imágenes 3D.

Esto no indica que cada dimensión esté orientada a usarse sólo con esas estructuras, se utiliza para proporcionar una visión de como es la forma de trabajo en cada dimensión, es decir, indica cuantos ejes de orientación tiene cada dimensión.

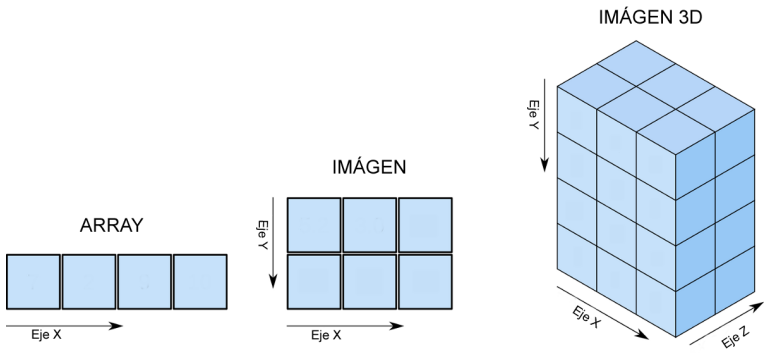


Figura 3.3: **Dimensiones OpenCL.** En la figura se muestra como se plantea los tres tipos de dimensiones, comenzando con un sólo eje de trabajo y acabando con tres orientaciones.

Los indices de localización en cada dimensión están estipulados por los work group [15] y los work item [16]. Un work item es un elemento de trabajo que actúa en el núcleo de cálculo o kernel, dentro de un dispositivo iniciado en el proceso de creación del entorno OpenCL, el cual dispone de un número limitado de work item dispuesto a lanzar. Cada elemento de trabajo se asigna y se ejecuta en un elemento de procesamiento, operando dentro del kernel de forma secuencial, cada work item se agrega a un work group, siendo la agrupación de varios elementos de trabajo. Estos grupos de trabajo son asignados por el programador a una unidad de computo, pudiendo actuar varios work group en la misma unidad. Como los work item, cada dispositivo tiene un máximo de work group que puede crear.

Cada work group utiliza para su propósito un espacio de memoria local, en el cual todos los work item pertenecientes al grupo pueden acceder a esa zona de memoria, siendo esta memoria ajena a los demás grupos. Si se desea compartir memoria se utiliza el espacio de memoria global. En el caso de los work item, a parte de poder usar esta memoria local, tiene su propia memoria privada, no permitiendo a los demás miembros poder alterar o ver el contenido.

Dentro de una unidad de trabajo para identificar al work item, OpenCL proporciona unas funciones útiles destinadas a este propósito. Para cada dimensión se da la opción de obtener el identificador global de cada work item, el identificador del grupo que está trabajando y el identificador del work item dentro del grupo, siendo estas algunas de las opciones disponibles.

size_t get_global_id(dimidx) identificador global del work item.
size_t get_group_id(dimidx) identificador del grupo.
size_t get_local_id(dimidx) identificador del work item en el work group.

Donde dimidx es la dimensión de la cual se quiere obtener el identificador.

Una vez explicado como son los elementos a utilizar en el entorno OpenCL, se pasa a indicar como se inicia la ejecución de cada kernel. La instrucción utilizada para iniciar cada ejecución es *clEnqueueNDRangeKernel()*, pero para un correcto funcionamiento, se debe asignar los objetos de memoria necesarios para su ejecución.

clSetKernelArg(cl_kernel kernel, cl_uint arg_index, size_t arg_size, const void *arg_value) [14]
kernel objeto kernel el cual se va a ejecutar.
arg_index posición en la lista de argumentos siguiendo el orden del encabezado del kernel.
arg_size tamaño del objeto de memoria o variable.
arg_value zona de memoria donde se localiza los datos del objeto o variable.

Con esta instrucción es la forma que tiene OpenCL para proporcionar datos al kernel y realizar las operaciones de computo sobre ellos, como ejemplo se utiliza uno de los kernel creados en el proyecto.

```
clSetKernelArg(validateRectangles, 0, sizeof(cl_mem), &dev_rectangleMap);
clSetKernelArg(validateRectangles, 1, sizeof(cl_mem), &IMAGE_WIDTH_OCL);
clSetKernelArg(validateRectangles, 2, sizeof(cl_mem), &IMAGE_HEIGHT_OCL);
```

Como se aprecia, primero se indica a qué kernel se asigna el dato (con el mismo nombre de la función a la que está asociado), su posición en el encabezado, el tamaño de la variable (en este caso son objeto de memoria) y por último se hace referencia a la zona de memoria donde están localizados los datos, siendo la referencia a la zona de memoria del dispositivo donde están alojados y donde se realizarán los cálculos.

Otro paso a realiza antes de iniciar la ejecución es saber cuantas dimensiones se van a utilizar

en el kernel, para acelerar el proceso y dividir el trabajo, en esta decisión influye la cantidad de work group encargados de dividir el trabajo junto a la cantidad de work item que trabajaran en cada grupo. Hay que tener en cuenta la cantidad de work group y work item que puede usar en cada dispositivo, para esto se tiene que consultar en el manual de OpenCL y ver que herramientas se utilizan para obtener la información.

Para asignar la cantidad total de work item a utilizar en la ejecución, usando su forma más básica de cálculo es: *número total = cantidad de grupos x cantidad de work item por grupo*. Pero con esta opción se da más cuando se trabaja con una dimensión. En el proyecto el número total de dimensiones utilizadas son 2, por lo cual el número total de item a utilizar se ha planteado de la siguiente forma:

espacio = espacio de trabajo
tamaño local[2] = {ancho, alto}
tamaño grupo[2] = {espacio/ancho, espacio/alto}
tamaño global[2] = {tamaño local[0] x tamaño grupo[0], tamaño local[1] x tamaño grupo[1]}

Este sistema es utilizado en el proyecto para poder dividir el trabajo de un plano del espacio de Hough. Ya que el espacio de trabajo total seria:

Espacio total = número de planos x espacio de cada plano

Con lo que para calcular el total, en la primera dimensión se utiliza la forma básica y se toma para la segunda dimensión el segundo espacio del tamaño global calculado, quedando como resultado final:

tamaño global [2] = {número de planos, tamaño local [1] x tamaño grupo [1]}
work item grupo [2] = {1, tamaño local [1]}

clEnqueueNDRangeKernel(cl_command_queue command_queue, cl_kernel kernel, cl_uint work_dim, const size_t *global_work_offset, const size_t *global_work_size, const size_t

*local_work_size, cl_uint num_events_in_wait_list, const cl_event *event_wait_list, cl_event *event) [14]

command_queue cola de comando a utilizar en el entorno OpenCL.

Kernel objeto kernel.

work_dim dimensión de trabajo.

global_work_offset actualmente es un valor NULL. Que utilizará en el futuro OpenCL.

global_work_size cantidad total de work item que se emplean en la ejecución.

l*ocal_work_size* cantidad de work item a utilizar en cada grupo de trabajo.

event_wait_list y num_events_in_wait_list especifica los eventos que deben completarse antes de ejecutar este comando en particular. Puede estar a NULL.

event retorna un objeto evento que identifica la ejecución del kernel. Puede estar a NULL.

Con todo ya preparado sólo queda ejecutar el kernel, a la función anterior se agrega por parámetros la cola de comandos, el objeto kernel, la dimensión en la cual se trabaja, la cantidad de work-item a utilizar y el tamaño de los work-group, lo demás se establece a NULL. En el momento de la ejecución la cola se encarga de elegir el orden según las propiedades que se han especificado en su creación. Como ejemplo se expone un inicio de ejecución de un kernel del proyecto.

wi número total de work item.

wg tamaño del grupo de trabajo o cantidad de work item pertenecientes a un work group.

clEnqueueNDRangeKernel(queue, validateRectangles, 1, NULL, &wi, &wg, 0, NULL, NULL);

3.6. Alternativa paralelismo GPU

Una alternativa a OpenCL para gráficas Nvidia es Cuda [17]. Es el sistema de programación paralela en GPU más potente y extendido, aunque está limitado a ciertas unidades gráficas. OpenCL surgió como alternativa a Cuda para proporcionar un framework de desarrollo independiente del hardware, proporcionando incluso la posibilidad de paralelizar a nivel de CPU.

Cuda es una plataforma de computación paralela y un modelo de programación creado por Nvidia, el cual permite desarrollar problemas de computo elevado, favoreciendo la resolución en sus unidades de procesamiento gráfico. Debido al gran poder de computo del que disponen estos dispositivos, por su gran cantidad de núcleos de procesamiento, los desarrolladores lo utilizan para garantizar el mayor rendimiento de sus aplicaciones.

Cuda permite desarrollar programas con diferentes lenguajes de programación como C, C++, Fortran, Python y MATLAB.

A continuación se muestra una comparación de características entre OpenCL y Cuda, realizada por AMD en 2010 [18], en la cual se muestran algunas de las diferencias principales.

La forma de compilar el código del programa en Cuda sólo es permite realizarlo de forma Offline, al contrario que OpenCL, proporciona al programador la posibilidad de realizar dicha compilación tanto de forma Online como Offline.

Referente a la precisión matemática a aplicar durante la ejecución, OpenCL está bien definida al contrario a Cuda. En el uso de bibliotecas matemáticas OpenCL utiliza estándar definidos al contrario a Cuda que usa bibliotecas propietarias, al ser OpenCL un framework de programación paralelo abierto y Cuda un sistema propietario, considerándose otra diferencia.

Una gran diferencia entre ambos modelos es la posibilidad de actuar en diferentes dispositivos. Cuda sólo está diseñado para trabajar en GPU de Nvidia, puede comunicarse con otros dispositivos pero no actuar sobre ellos. OpenCL, al contrario, es un sistema multiplataforma pudiendo utilizar GPU de diferentes fabricantes e incluso de realizar la computación paralela sobre CPU, tanto Intel como AMD.

Los mecanismos de extensión de Cuda están establecidos como propietarios y OpenCL definidos.

En el caso de uso de lenguajes de programación soportado, son muy similares ya que pueden usar tanto C, C++ y Python.

Desviando la atención a la cantidad de vendedores que dan soporte como plataforma, se decanta por OpenCL como el sistema entre los dos más soportado, se proporciona soporte de plataforma por AMD, Apple e incluso Nvidia, al contrario de Cuda que sólo dispone de Nvidia.

Capítulo 4

Implementación y desarrollo

4.1. Introducción a la implementación

Para poder operar en la aplicación se han creado diferentes estructuras de datos, que almacenan información sobre los diferentes aspectos del programa con lo que se realiza la funcionalidad. OpenCL sólo admite estructuras lineales, con lo cual todas las estructuras son de ese tipo. Lo correcto es usar estructuras de doble puntero, separando mejor los diferentes planos por los que están formados las estructuras asemejadas al espacio de Hough. No es del todo cierto que OpenCL admita sólo estructuras lineales, ya que proporciona un tipo de estructura *"image"* la cual puede ser 1D, 2D ó 3D. En este proyecto encajaría la estructura *"image 2D"* siendo parecida al doble puntero, pero no permiten realizar operaciones de forma atómica, con lo que esta opción ha sido descartada.

Como se ha explicado en el apartado *"Creación del Entorno"* primero se reserva memoria en el entorno OpenCL creado en el dispositivo. A diferencia del dispositivo host, se reserva espacio para un tipo de dato objeto, identificado como *"cl_mem"* u objeto de memoria OpenCL, donde la forma de reservar espacio en memoria es la misma que para una estructura lineal. Después en el kernel se considera a este objeto como un puntero. Para realizar esta reserva de memoria, OpenCL pone a disposición la siguiente función que devuelve el objeto de memoria reservado.

clCreateBuffer(cl_context context, cl_mem_flags flags, size_t size, void *host_ptr, cl_int *errcode_ret) [14]
context contexto OpenCL.
flags indica como se utiliza la memoria, si es lectura, escritura o ambas y también si proviene del host anfitrión.
size cantidad de memoria a reservar.
**host_ptr* indica la zona de memoria donde están los datos a usar en host, se coloca a NULL si no se utiliza la memoria del host anfitrión.
errcod_ret código de error.

Para este propósito se ha creado una función más orientativa que realiza una llamada a la función anterior. Esta función es *"oclMalloc"* (oclMalloc(cl_mem_flags, site_t size, void

*host_ptr, cl_mem &var)) y, a diferencia de la función de OpenCL, retorno el código del error y devuelve en el parámetro &var el objeto de memoria.

cl_mem dev_HoughSpace;

*cl_int error = oclMalloc(CL_MEM_READ_WRITE, (sizeof(int) * (HOUGH_DISTANCES) * (HOUGH_PDISTANCES)) * HOUGH_ANGLES, NULL, dev_HoughSpace);*

En este ejemplo se ha representado como reservar la memoria para el espacio de Hough 3D, indicando el tamaño que tiene cada plano, formado por los valores $HOUGH\_DISTANCES$ (H_D considerado como el número total de filas) y $HOUGH\_PDISTANCES$ (H_PD siendo el número total de columnas), multiplicado por la cantidad de ángulos o planos por los que va a estar formado el espacio de Hough 3D. En este caso, se otorga acceso a memoria con privilegios de escritura y lectura. Para los objetos de memoria que contienen valores constantes, se utiliza en su lugar CL_MEM_ONLY_READ, dando privilegios de sólo lectura.

Como es una estructura lineal, los planos están consecutivos. El acceso a una posición concreta en un plano se lleva a cabo por tres parámetros ya descritos en HT3D, usando θ para identificar el plano, d localiza una linea en el plano y p una posición en la linea. Para realizar el desplazamiento se considera el tamaño del plano o el número total de celdas por las que está formado, siendo $\theta * (H\_D * H\_PD)$, accediendo de esta manera al plano indicado por θ. La localización de una linea se consigue a partir del total de celdas que forman una linea, siendo $d * H\_PD$, y dentro de esta linea se sitúa en la celda correspondiente p, quedando como resultado en el uso de la estructura como:

$$HoughSpace[\theta * (H\_D * H\_PD) + d * H\_PD + p].$$

Dentro de cada kernel los valores de θ, d y p son obtenidos a partir de los identificadores de cada hilo o work-item que se ejecuta en ese momento. Para este propósito se utiliza las herramientas explicadas en el apartado *"Entorno de trabajo"*, siendo identificado por los identificadores de grupo y de hilos locales. A continuación se explica por cada procedimiento realizado en la HT3D, que estructura usa, cuantas dimensiones se ha decidido utilizar para trabajar, como se

divide el trabajo, etc.

4.2. Creación del espacio de Hough 3D

La creación del espacio de Hough 3D es un requisito principal para el uso de la HT3D, separando su inicialización y recolección de datos en tres partes: puesta a cero, proceso de votación y proceso de acumulación. En este proceso se inicializan cinco estructuras de datos, a parte del espacio de Hough (HoughSpace): puntos detectados en el espacio de Hough (pointSpace), con las mismas característica que *"HoughSpace"*, el espacio de votación de esquinas y puntos extremos (pointVotes), el mapa de puntos (pointMap), el mapa de segmentos (segmentMap) y el mapa de rectángulos (rectangleMap), siendo estas cuatro últimas del mismo tamaño que la imagen.

La puesta a cero se encarga de inicializar tanto *"HoughSpace"* como *"pointSpace"*, asignando en cada celda de HoughSpace el valor a cero y en pointSpace a -1 (indica la no existencia de esquina en esa celda). Esta tarea se ejecuta en dos dimensiones, cada hilo de la primera dimensión esta dirigido a un plano. El desarrollo de cada plano se ejecuta en la segunda dimensión, asignando a cada celda un hilo del grupo de esta segunda dimensión.

θ = *get_group_id(0)*
d = *get_group_id(1)*
p = *get_local_id(1)*

$$HoughSpace[\theta * (H\_D * H\_PD) + d * H\_PD + p] = 0$$
$$cornersImg[\theta * (H\_D * H\_PD) + d * H\_PD + p] = -1$$

En el proceso de votación, como la búsqueda de características se realizan sobre la imagen, sólo se requiere del uso de una dimensión. Como se trabaja en una dimensión, se proporciona el valor de "*y*" a partir del identificador del grupo y a "*x*" el identificador del hilo local.

Para identificar una posición se utiliza el ancho de la imagen *y * (ancho imagen) + x*

considerándose el valor obtenido como índice, utilizado para inicializar cada posición de pointMap, pointVotes, segmentMap y rectangleMap. La funcionalidad está bien descrita en el algoritmo 1. Como cada celda de la imagen es asignada a un hilo no se desempeña el primer bucle. De esta manera si una celda no contiene un punto de borde no se efectúa ningún cálculo, terminando la ejecución para ese hilo.

El proceso de acumulación es igual que la puesta a cero en relación al reparto de trabajo y a la asignación de los parámetros θ, d y p. El funcionamiento se basa en el algoritmo 2 cada hilo es asignado a una línea del espacio de Hough y ejecuta el algoritmo tal y como esta definido. Como los hilos se ejecutan de forma concurrente, para realizar la acumulación se almacena los datos de cada celda actual en una estructura reservada como local en la posición p, cada hilo realiza una espera hasta que los demás hilos del grupo realicen la misma operación, tras lo cual continúa con la acumulación.

4.3. Detección de esquinas y puntos extremos

Para llevar a cabo la detección de puntos característicos (esquinas y puntos extremos) en la imagen se ha organizado el procesamiento en tres partes: detección de patrones de esquinas y puntos extremos, votación de los puntos detectados en el espacio de imagen y detección final de los puntos en la imagen.

Se utiliza como soporte tres de las estructuras inicializadas en el apartado anterior, a parte del espacio de Hough: pointSpace, contiene los puntos detectados en el espacio de Hough, pointVotes, almacena los votos en el espacio de imagen de las celdas que contienen un patrón de esquina o punto extremo, y pointMap para almacenar los puntos finalmente detectados.

En la detección de patrones de esquinas y puntos extremos se opera sobre dos dimensiones, dando el control de cada plano a un hilo de la primera dimensión y otorgando dentro de cada plano, un hilo a cada celda en la segunda dimensión. Así, el identificador de grupo de la primera dimensión se asocia a θ, el de la segunda dimensión a d y el identificador del hilo dentro del grupo a p.

Esta función está definida por los algoritmos 3 y 4. Para la detección de un punto extremo el procesamiento de cada celda (código incluido en los bucles) es realizado por un hilo. En el caso de las esquinas, aunque es un algoritmo más complejo, el trato que realizan los hilos es el mismo que en la detección de puntos extremos. También, se reduce el tiempo de detección comprobando antes si hay que realizar el procedimiento en su forma normal o invertida. Además, como no se realizan los primeros bucles del algoritmo, los últimos cálculos que se efectúan sobre η y ϕ son realizados una vez, antes de pasar a la detección de esquinas.

Una vez detectados estos patrones y marcadas sus posiciones de celda en *pointSpace*, el procesamiento continúa con la votación de estas celdas sobre el espacio de imagen. En esta función el reparto de los hilos y el uso de las dimensiones es igual que en la fase anterior, ya que es necesario realizar el procesamiento por cada celda. Si sobre una celda no se ha detectado esquina o punto extremo, el hilo finaliza. En caso contrario, la celda vota por la posición correspondiente de imagen tal y como se indicó en la sección 2.4.

La detección final de los puntos en la imagen realiza el procesamiento en una dimensión ya que actúa sobre la imagen. A cada pixel de la imagen se le asigna un hilo, del cual se obtiene la identificación de posición sobre la imagen en los ejes x e y, dando al eje x el identificador del grupo y a cada posición del eje y el identificador del hilo en el grupo, siendo asignada cada fila de la imagen a un grupo de trabajo. Esta función se encarga de buscar en las posiciones (x, y) de la imagen los puntos con mayor número de votos en su entorno local y con un valor de votación superior a un umbral.

4.4. Transformación de esquinas y puntos extremos al espacio de Hough

Las esquinas y puntos finales localizados en una imagen son utilizados para la detección de otras características de imagen mediante el espacio de Hough 3D. Para ello, se debe de transformar la posición de imagen de estos puntos a posiciones de celda en el espacio de Hough. Esta transformación se realiza mediante dos funciones: inicialización del espacio de

puntos *pointSpace* (aloja los puntos detectados en el espacio de Hough 3D) y transformación de coordenadas.

Como se ha indicado anteriormente, una de las estructura utilizadas es el espacio de puntos (pointSpace), un espacio tridimensional donde se encuentran marcadas las posiciones de los puntos detectados en el espacio de Hough 3D, y el mapa de puntos (pointMap), donde están los puntos detectados en el espacio de la imagen.

La inicialización del espacio de puntos se encarga de recorrer por completo el espacio de Hough 3D y asignar a cada celda un valor de -1 que indica que la celda no contiene ningún punto característico. Para paralelizar este proceso, se utilizan dos dimensiones de grupos como en el resto de funciones que trabajan sobre el espacio de Hough 3D. Así, los dos identificadores de grupo indican una posición de línea en el espacio de Hough (valores de θ y d) y el identificador de hilo hace referencia a una celda concreta dentro de la línea (valor de p). De esta manera cada hilo se encarga de la inicialización de una celda concreta.

La fase de transformación de coordenadas se lleva a cabo en una segunda función que trabaja sobre el mapa de puntos (pointMap). Así, a cada píxel de la imagen se le asigna un hilo que se encarga de marcar todas las celdas del espacio de Hough asociadas a esa posición de imagen (ecuaciones 2.2 y 2.3). Es decir, cada hilo trabaja sobre un único píxel que codifica su posición en las celdas que le corresponden en *pointSpace*.

4.5. Detección de segmentos

La paralelización del método de detección de segmentos se realiza a través de dos funciones: detección de segmentos propiamente dicha y validación de segmentos.

Para llevar a cabo este proceso, se utilizan tres estructuras: el espacio de Hough 3D (HoughSpace), el espacio de puntos (pointSpace) y el mapa de segmentos (segmentMap), del tamaño de la imagen y encargado de almacenar información de los segmentos detectados dentro de la imagen en las posiciones indicadas por sus puntos extremos.

La detección de segmentos se lleva a cabo siguiendo el algoritmo 5. El procesamiento paralelo se lleva a cabo a partir de 2 dimensiones de grupos para el acceso al espacio de Hough 3D y a la estructura tridimensional pointSpace. Así, los planos del espacio son identificados en su primera dimensión por el identificador del grupo, anotándose en θ_d, realizando el tratamiento de cada línea de plano en la segunda dimensión (d_d) y proporcionando un hilo por celda (p_d). Al inicio de la ejecución del kernel, los hilos que no detectan en su posición de una esquina o punto extremo finalizan. El resto de hilos continuarían con el procesamiento siguiendo los pasos descritos en el algoritmo 5. Para agilizar el tiempo ejecución se han utilizado varias estructuras locales que recogen tres lineas verticales del espacio de Hough, es decir, la linea actual $l(\theta_d, d_d)$ y sus líneas vecinas, además de la linea $l(\theta_d, d_d)$ del espacio de puntos (pointSpace). Cada hilo copia el contenido de su celda en la posición p de estas estructuras realizando una espera hasta que todos los hilos finalicen la copia.

Para almacenar los segmentos detectados, se utiliza la estructura *segmentMap*. Esta estructura tiene las mismas dimensiones que la imagen original y, en cada posición, almacena información sobre los segmentos conectados a ese píxel. Concretamente, cada píxel contiene un vector (cada elemento hace referencia a un segmento conectado a dicho punto) y el tamaño actual del vector. La información de un segmento se almacena duplicada, para sus dos puntos extremos. Hay que tener en cuenta que los distintos segmentos conectados a un mismo punto se detectan desde hilos diferentes, por lo que es posible que 2 hilos diferentes almacenen sus segmentos en una misma posición del vector. Para evitarlo, el tamaño del vector asociado a un punto se incrementa utilizando una operación atómica. El nuevo tamaño obtenido hace referencia a la primera posición libre del vector de segmentos asociados a ese punto. El segundo problema a considerar en esta fase de almacenamiento del resultado es que un mismo segmento puede detectarse desde hilos diferentes. Esto implica que el mismo segmento puede almacenarse repetido varias veces. Puesto que el tamaño del vector es limitado y debe ser establecido de antemano, es deseable que esto no ocurra, por lo que, antes de almacenar un segmento, se recorren los vectores de segmento asociados con sus puntos extremos. Tras este recorrido, si se determina que el segmento no se encuentra ya almacenado, se guarda en la posición obtenida por la operación atómica indicada anteriormente.

La validación de segmentos es un proceso de supresión del no-máximo que se lleva cabo en cada vector de segmentos asociado con un punto de imagen. Para ello, se asocia un hilo a cada posición (x, y). El procesamiento que realiza cada hilo se encarga de recorrer el vector de segmentos conectados a ese punto para determinar si existe un segmento similar (en orientación) con mayor valor de *ss*. De ser así, el segmento se marca como inválido en el vector del punto actual y en el vector asociado con el otro punto extremo. Los segmentos no marcados como inválidos constituyen el resultado final de la detección.

4.6. Detección de rectángulos

Para llevar a cabo la detección de rectángulos se utilizan dos funciones: detección de rectángulos y validación de rectángulos. El proceso detección hace uso de 3 estructuras: el espacio de Hough 3D (HoughSpace), el espacio de puntos (pointSpace) y el mapa de rectángulos (rectangleMap) del tamaño de la imagen y utilizado para almacenar información de los rectángulos detectados.

En la detección de rectángulos el reparto de hilos se lleva a cabo utilizando 2 dimensiones de grupos. La primera dimensión se encarga de identificar a cada plano del espacio de Hough (θ), mientras que la segunda dimensión identifica a cada línea de un plano (d). De esta manera un plano es organizado por tantos grupos de trabajo como lineas verticales formen el plano de Hough, identificando cada celda de la linea por el hilo local para p. Así, cada hilo iniciado trabaja sobre una celda realizando la búsqueda del resto de puntos o vértices que forma el rectángulo, si la celda de la cual parte contiene una esquina.

El almacenamiento de cada rectángulo detectado sigue una estrategia similar a la empleada para el almacenamiento de segmentos. Cada píxel del mapa de rectángulos contiene un vector con los rectángulos detectados y asociados a ese punto y un contador con el número actual de rectángulos. En este caso un punto del mapa identifica el centro del rectángulo.

Para la validación de rectángulos, puesto que sólo utiliza el mapa de rectángulos y este tiene el tamaño de la imagen, sólo se requiere para realizar las operaciones una dimensión de grupos.

Así, el identificador de grupo se utiliza para hacer referencia a la fila de imagen (y) y el identificador del hilo local para identificar la columna (x). Con esto cada hilo realiza, por cada celda de forma independiente, la validación de los rectángulos asociados con ese punto (centro del rectángulo), siguiendo un proceso de supresión del no-máximo en el que se consideran rectángulos del entorno local a otro dado aquellos que presentan un tamaño y una orientación similar.

4.7. Aplicación desarrollada

Para poner en práctica la técnica explicada y comentada en este documento se ha desarrollado una aplicación que muestra en tiempo real sobre la imagen, las esquinas y puntos extremos de segmento, los segmentos de línea y rectángulos detectados.

La estructura software está soportada sobre tres clases. Una de ellas esta destinada a definir la configuración a aplicar sobre una imagen o conjunto de imágenes con la que se puede extraer más o menos características según la configuración empleada. Por otro lado, nos encontramos con una clase principal encargada de realizar todo el control donde fluye toda la ejecución. Ésta se encarga de realizar las llamadas a las funciones tanto de configuración como las destinadas a aplicar los algoritmos de la Transformada de Hough sobre las imágenes. Por último, siendo la más importante, la implementación de OpenCL junto al desarrollo de la HT3D. Básicamente en esta clase sólo está implementada la creación del entorno OpenCL y su manipulación, ya que toda la implementación de la HT3D se ha realizado en un fichero ajeno a la aplicación, y este es asignado al entorno OpenCL por lectura de ficheros cuando se inicia la aplicación.

A continuación se muestran varias capturas indicando la funcionalidad de la aplicación.

En la figura 4.1 se muestra la ventana principal de la aplicación y los resultados de detección sobre una imagen determinada. En la parte superior se encuentra imagen original. En orden de izquierda a derecha y de arriba a abajo se muestran las esquinas y puntos extremos de segmento, los segmentos de linea, la imagen canny y los rectángulos localizados. En esta ventana se dispone de tres botones y tres opciones. Las opciones sirven para indicar posibilidad

de detectar distintos conjuntos de características sobre la imagen (segmentos, rectángulos o ambos). Entre los botones esta “Save Result” encargado de guardar las imágenes resultantes. Este botón permanece oculto cuando no hay imagen y sólo se activa cuando se opera sobre una imagen. El botón “Configuration” permite abrir la ventana de configuración y el botón “Resize Views” proporciona una visualización de los resultados con la resolución completa de la (Figura 4.3).

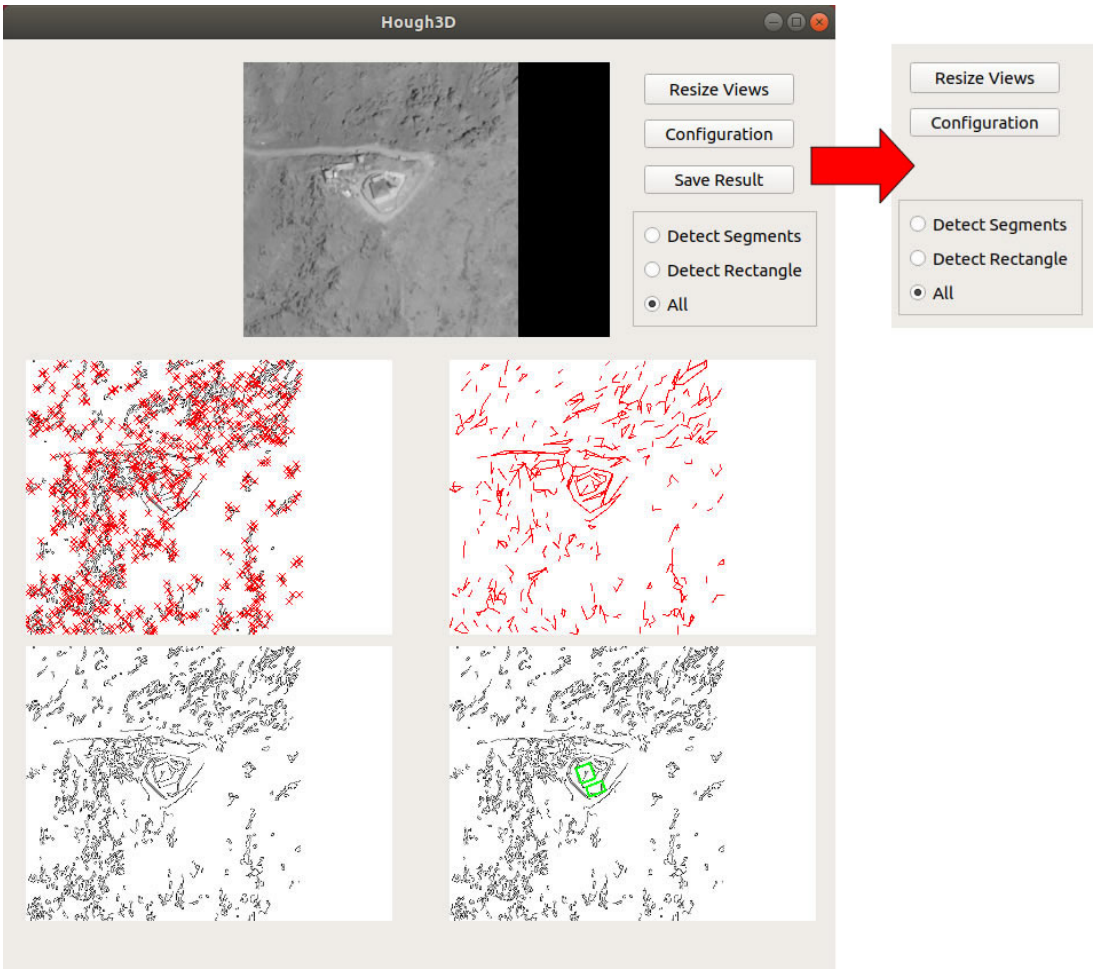


Figura 4.1: **Ventana principal de la aplicación.** Zona donde se centra la ejecución de la aplicación y la encargada de mostrar los resultados conseguidos. También se indica como estaría el botón “Save Result” al iniciar la aplicación.

La figura 4.2 muestra la ventana de configuración. En la parte inferior se sitúan tres paneles de configuración (de izquierda a derecha): parámetros para crear el espacio de Hough, parámetros para detectar las esquinas y puntos extremos de segmentos y por último parámetros para detectar

segmentos de linea. En la mitad superior derecha están los botones de aceptación, cancelación y “Search Images” utilizado para realizar la búsqueda de un directorio que contenga imágenes. También se dispone de tres opción de ejecución con las imágenes: ejecutar una sola imagen, ejecutar todo el directorio y ejecutar todo el directorio y guardar tiempos de ejecución. Siendo lo más visual de esta ventana la representación de la imagen elegida con el titulo encima y a su derecha la lista de imágenes recogidas del directorio seleccionado, pudiendo elegir cualquiera de ellas.

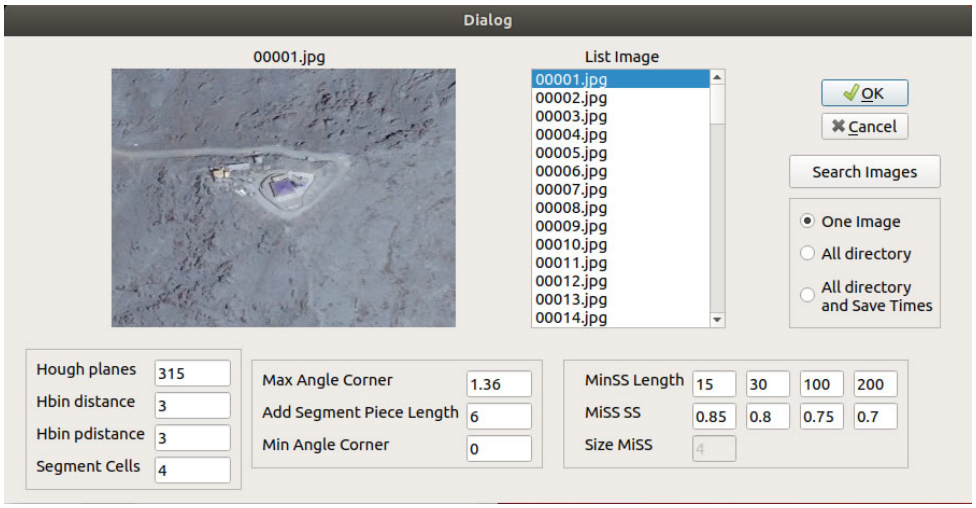


Figura 4.2: **Ventana de configuración de la aplicación.** Muestra la ventana de configuración, donde se selecciona la imagen o conjunto de imágenes y los parámetros de configuración para el espacio de Hough, la detección de esquinas y segmentos de linea.

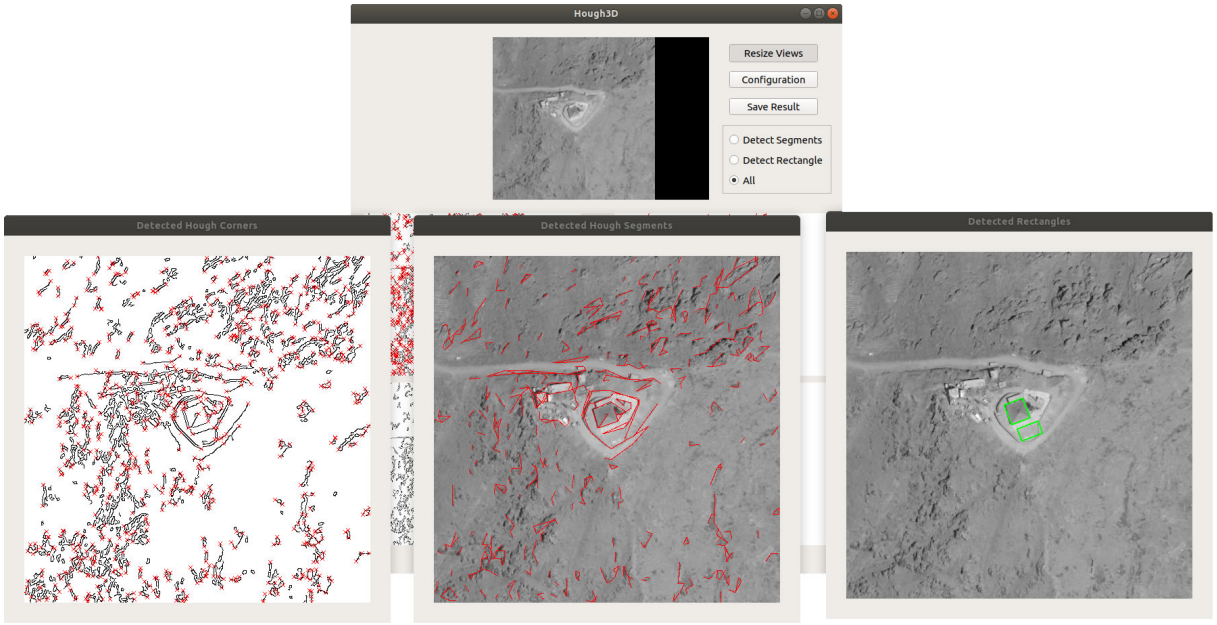


Figura 4.3: **Ventanas Resize Views.** Realiza una representación más amplia de los resultados obtenidos de la detección de esquinas, detección de segmentos de linea y detección de rectángulos.

Capítulo 5

Análisis de Resultados

5.1. Descripción General

5.1.1. Plataformas hardware empleadas

Las pruebas se han realizado sobre tres plataformas, utilizando la gráfica integrada de Intel Skaylake de un i7 y dos GPU's de Nvidia: Nvidia GTX 1080 y Nvidia RTX 2080, de las cuales se indican sus especificaciones en las tablas siguientes.

NVIDIA GTX 1080	
Especificaciones de la memoria:	
Frecuencia de la memoria	10 Gb/s
Configuración de memoria estándar	8 GB GDDR5X
Ancho de la interfaz de memoria	256-bits
Ancho de banda de memoria (GB/s)	320
Pantallas:	
Máxima resolución digital[2]	7680x4320 a 60Hz
Conectores de pantalla	DP 1.43, HDMI 2.0b, DL-DVI
Capacidad multimonitor	Sí
HDCP	2.2

Tabla 5.1: Especificaciones NVIDIA 1080 (Parte 1)

NVIDIA GTX 1080	
Especificaciones del motor de GPU:	
NVIDIA CUDA® Cores	2560
Frecuencia de reloj normal (MHz)	1607
Frecuencia acelerada (MHz)	1733
Tecnologías soportadas:	
Simultaneous Multi-Projection	10Gb/s
VR Ready	Sí
NVIDIA Ansel	Sí
NVIDIA SLI® Ready1	Sí, compatible con puente SLI HB
NVIDIA G-SYNC™-Ready	Sí
NVIDIA GameStream™-Ready	Sí
NVIDIA GPU Boost™	3.0
Microsoft DirectX	Versión 12 nivel 12_1
Vulkan API	Sí
OpenGL 4.5	4.5
Soporte de bus	PCIe 3.0
Certificación para SO	Windows 7-101, Linux, FreeBSDx86

Tabla 5.2: Especificaciones NVIDIA 1080 (Parte 2)

NVIDIA RTX 2080	
Especificaciones del motor de GPU:	
NVIDIA CUDA® Cores	2944
RTX-OPS	57T
Giga Rays/s	8
Frecuencia acelerada (MHz)	1710
Frecuencia de reloj normal (MHz)	1515
Especificaciones de memoria:	
Frecuencia de la memoria	14 Gbps
Configuración de memoria estándar	8 GB GDDR6
Ancho de la interfaz de memoria	256-bit
Ancho de banda de memoria (GB/s)	448 GB/s
Pantalla:	
Máxima resolución digital (2)	7680x4320
Conectores de pantalla	DisplayPort 1.4, HDMI 2.0b
Capacidad multimonitor	4
HDCP	2.2

Tabla 5.3: Especificaciones NVIDIA 2080 (Parte 1)

NVIDIA RTX 2080	
Tecnologías soportadas:	
Trazado de rayos en tiempo real	Sí
NVIDIA® GeForce Experience	Sí
NVIDIA Ansel	Sí
NVIDIA® Highlights	Sí
Compatible con NVIDIA G-SYNC™	Sí
Controladores Game Ready	Sí
API de Microsoft® DirectX® 12, API de Vulkan, OpenGL 4.6	Sí
DisplayPort 1.4, HDMI 2.0b[3]	Sí
HDCP 2.2	Sí
NVIDIA® GPU Boost™	4
NVIDIA NVLink (SLI-Ready)	Sí - Con NVIDIA RTX NVLink Bridge
VR Ready	Sí
Diseñada para USB Type-C™ and VirtualLink™[5]	Sí
NVIDIA Encoder (NVENC)	Sí (Turing)

Tabla 5.4: Especificaciones NVIDIA 2080 (Parte 2)

5.1.2. Configuraciones del espacio de Hough 3D

En la realización de cada prueba se ha utilizado una configuración especifica del espacio de Hough tanto en la detección de segmentos como en la detección de rectángulos. En concreto, se han utilizado tres tipos de configuraciones para la detección de segmentos y dos tipos para la detección de rectángulos, creando un espacio de Hough 3D diferente con cada una de las pruebas realizadas con cada configuración.

	$\triangle\theta$	$\triangle d$	$\triangle p$
Configuración 1	0.01 (315 planos)	2	2
Configuración 2	0.01 (315 planos)	3	3
Configuración 3	0.015 (210 planos)	2	2

Tabla 5.5: Configuraciones Detección Segmentos

	$\triangle\theta$	$\triangle d$	$\triangle p$
Configuración 1	0.01 (315 planos)	3	3
Configuración 2	0.015 (210 planos)	3	3

Tabla 5.6: Configuraciones Detección Rectángulos

5.2. Pruebas Detección de Segmentos

5.2.1. Comparación tiempo implementación secuencial y OpenCL en Nvidia

Esta prueba ha sido realizada sobre tarjetas gráficas Nvidia utilizando la configuración 1 de la tabla 5.5. Se realiza una comparación de los resultados obtenidos por la gráfica Nvidia RTX 2080 y Nvidia GTX 1080, mostrando la comparativa del Speedup en relación a la

ejecución secuencial, los tiempos de las fases de votación, detección de esquinas y detección de segmentos, además del tiempo total de ejecución.

La mejora obtenida por la implementación paralela es bastante notable. Así, como se puede observar en la figura 5.1 se ha conseguido una mejora, según la imagen tratada, entre cinco y veinte veces mejor dependiendo del hardware empleado, obteniendo estos resultados en su gran mayoría gracias a los cambios realizados en el software.

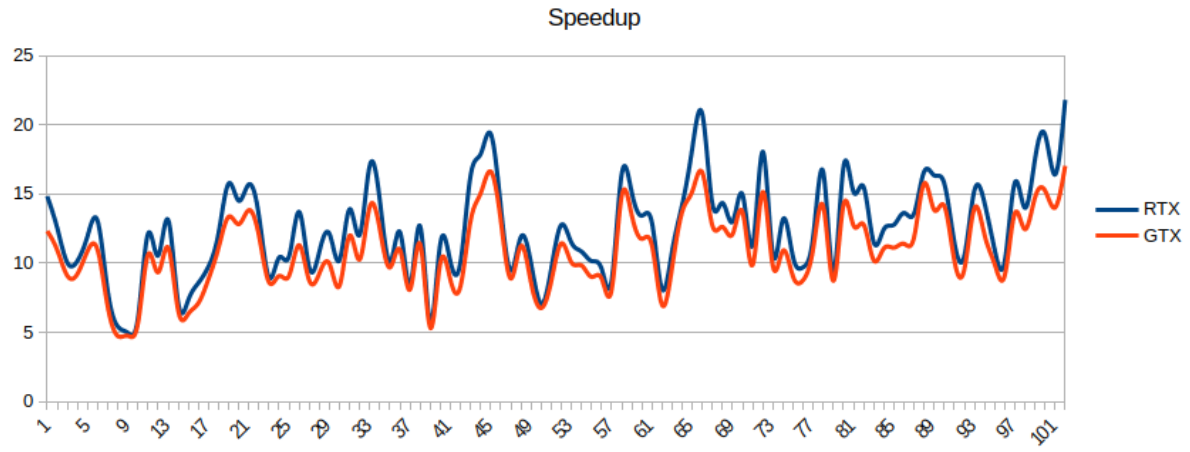


Figura 5.1: **Speedup global de OpenCL en relación a la versión secuencial entre Nvidia RTX 2080 y Nvidia GTX 1080 en la detección de segmentos.** En la gráfica se realiza una comparación del Speedup global de OpenCL en relación a la versión secuencial entre las dos GPU's mencionadas, representando de forma lineal la eficiencia obtenida marcada en el eje vertical al analizar cada una de las imágenes de un conjunto identificado por el eje horizontal.

A continuación, se muestran los tiempos empleados en el tratamiento de cada una de las imágenes para las distintas fases de la detección. La diferencia de tiempos del modelo secuencial al paralelo es muy elevada. Asimismo, los tiempos de la ejecución secuencial tienen mucha mayor variabilidad, causada por la diferencia de características detectables de cada imagen. La distribución de procesamiento entre los diferentes hilos en la ejecución paralela evita esta dependencia con la complejidad de la imagen, proporcionando tiempos aproximadamente constantes.

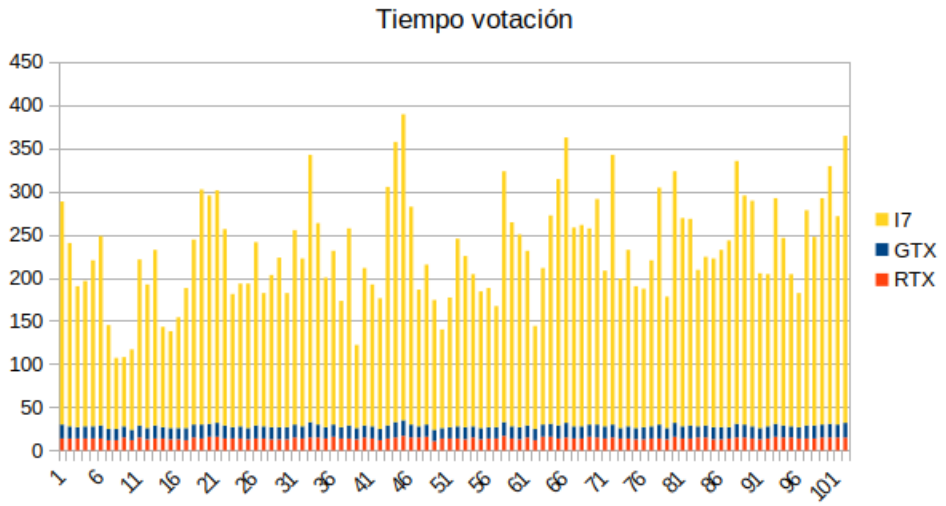


Figura 5.2: **Tiempos fase votación entre i7, Nvidia RTX 2080 y Nvidia GTX 1080 en la detección de segmentos.** Se muestran los tiempos empleados al realizar la fase de votación por cada uno de los dispositivos indicados, en cada una de las imágenes de un conjunto, indicando el tiempo en su eje vertical y el identificador de la imagen en su eje horizontal.

Una diferencia detectada en la representación gráfica de los tiempos en cada una de sus fases, es la variación entre las dos gráficas Nvidia debido a sus características, aunque se considera que la gráfica RTX 2080 es mejor que la GTX 1080, si nos fijamos en la representación de los tiempo de votación la GTX realiza antes la tarea que la RTX, esto se debe a su velocidad de trabajo ya que en esta fase se realiza un proceso de calculo y relleno donde sólo se accede una vez a cada celda. Al contrario de sus posteriores fases, siendo mejor la RTX debido a que el ancho de banda de acceso a memoria es mayor y aunque en la fase de votación no había mucha diferencia ahora con este factor reduce más su tiempo, ya que tanto para la detección de esquinas como segmentos se accede varias veces a memoria.

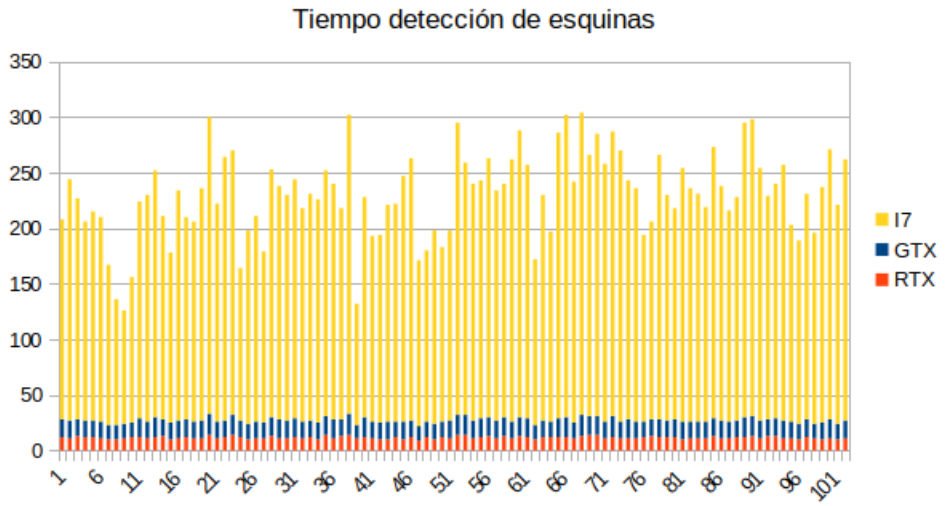


Figura 5.3: **Tiempos fase detección de esquinas entre i7, Nvidia RTX 2080 y Nvidia GTX 1080 en la detección de segmentos.** Representa el tiempo empleado en realizar la detección de esquinas y puntos extremos en cada imagen de un conjunto dado por los tres dispositivos mencionados. Marcando el tiempo empleado por cada imagen en el eje vertical y en el eje horizontal se muestra en orden numérico su identificador.

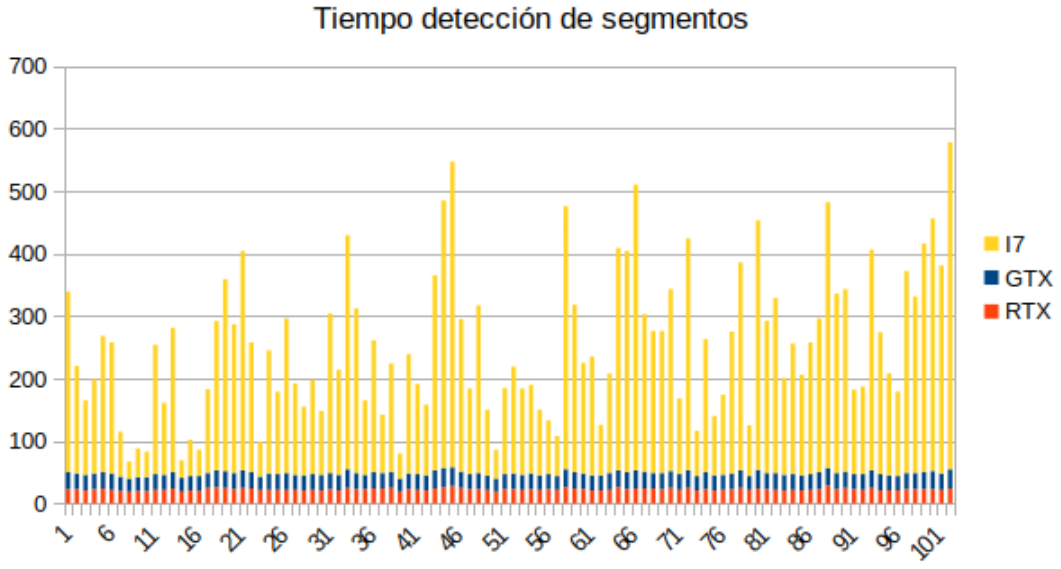


Figura 5.4: **Tiempos fase detección de segmentos entre i7, Nvidia RTX 2080 y Nvidia GTX 1080 en la detección de segmentos.** Se visualiza los distintos tiempos consumidos por cada uno de los dispositivos empleados para realizar la fase de detección de segmentos en una imagen. Realizando la misma operación en un conjunto de imágenes siendo identificadas por su valor numérico, indicado en su eje horizontal y el tiempo requerido en el eje vertical.

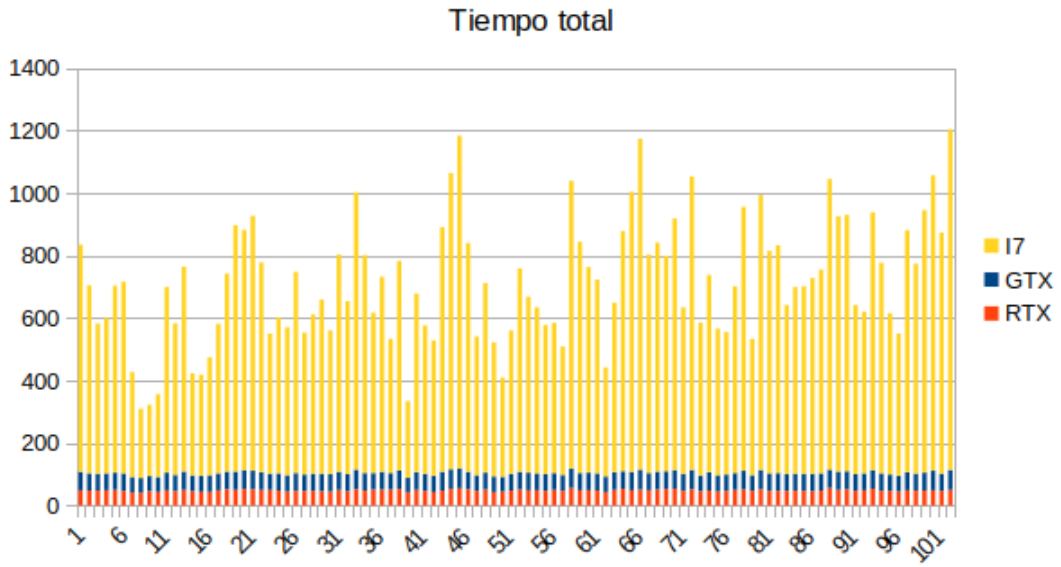


Figura 5.5: **Tiempo total empleado entre i7, Nvidia RTX 2080 y Nvidia GTX 1080 en la detección de segmentos.** Representa el tiempo total empleado por cada dispositivo para realizar las fases de votación, detección de esquinas y detección de segmentos en cada imagen. Representando el tiempo en su eje vertical y el identificador de cada imagen en su eje horizontal.

Como se ha mencionado, aunque el software realiza un papel importante en la mejora, el uso del hardware con unas mejores características incrementa esta eficiencia, tomando esta opinión en base a los resultados obtenidos a partir de los tres dispositivos empleados.

5.2.2. Comparación tiempos implementación secuencial y OpenCL en Intel Skylake

Esta prueba ha sido realizada utilizando una gráfica intergrada de Intel (Intel Skylake). La comparación de tiempos se ha realizado a partir de los resultados obtenidos por el modelo secuencial y los obtenidos al usar la versión paralela de OpenCL utilizando la configuración 1. Se muestra el Speedup tanto del global como por fases, así como la comparativa de tiempos en las fases de votación, detección de esquinas, detección de segmentos y tiempo total de ejecución.

Durante la realización de las pruebas con este dispositivo se ha planteado ejecutar dos variantes de la implementación, elegida para la representación la opción más optima. En concreto se han

planteado dos versiones, con memoria compartida y sin memoria compartida, en la ejecución. El uso explícito de memoria compartida no está aconsejado en este tipo de gráficas, ya que la memoria local es una porción de la caché L3 de la CPU que es gestionada por hardware. Su utilización explícita por software provoca operaciones innecesarias que aumentan los tiempos de ejecución. Este hecho puede observarse claramente en la figura 5.6.

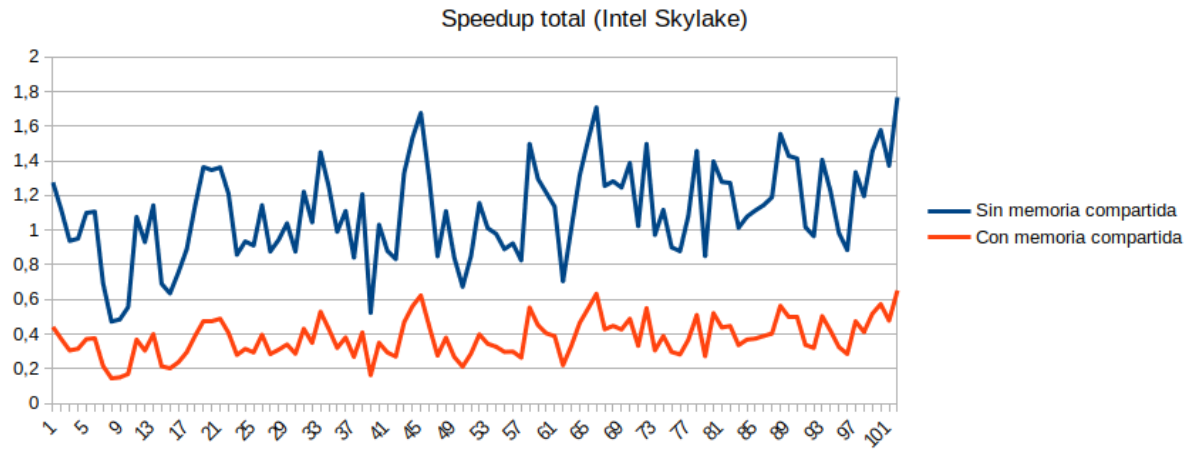


Figura 5.6: **Speedup global de las dos variantes en relación a la versión secuencial con Intel Skylake en la detección de segmentos.** Se muestra el Speedup obtenido al ejecutar un conjunto de imágenes con el uso de memoria compartida y sin memoria compartida. Indicando su eficiencia en el eje vertical y la imagen tratada en el eje horizontal.

Tal y como se muestra en dicha figura, la diferencia en eficiencia entre las dos versiones no es la misma para todas las imágenes, siendo mayor cuando aumenta el número de características a detectar. Aun con este factor se consigue con la variante sin memoria compartida en el peor de los casos una mejora de 0.4 veces mejor en relación a la versión secuencial y una unidad más en el mejor de los casos en la versión con memoria compartida. Si se contempla por fases en la figura 5.7, donde se produce perdida de eficiencia es en la fase de detección de segmentos consumiendo más tiempo de computo debido a las búsquedas realizadas para localizar dos esquinas candidatas a formar un segmento.

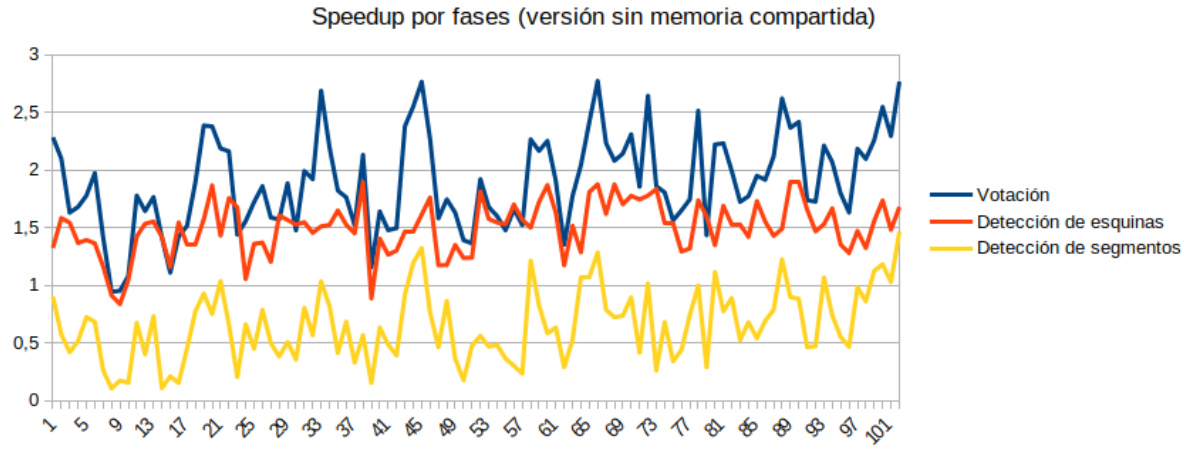


Figura 5.7: **Speedup por fases de la variante sin memoria compartida con Skylake en la detección de segmentos.** Se representa el Speedup obtenido por cada fases tras ser aplicado a un conjunto de imágenes usando la variante de implementación sin memoria compartida. Indicando su eficiencia en el eje vertical y la imagen tratada en el eje horizontal.

Los tiempos recogidos por cada fase sobre la versión paralela implementada en OpenCL y ejecutada por el dispositivo gráfico de Intel Skylake consigue valores por imagen casi constantes, manteniéndose en unos rangos de tiempo muy similares, sin importar mucho la cantidad de características que puedan ser detectadas. La versión secuencial, al contrario, genera grandes oscilaciones en tiempos por cada imagen que analiza, produciendo en algunas mejores resultados en su implementación secuencial que en OpenCL. Esto se debe a que la implementación secuencial, cuando no se detectan más características, concluye su ejecución. Sin embargo, en la paralela, la ejecución no termina hasta que todos sus hilos finalizan. Este efecto se aprecia más en la detección de segmentos representada en la figura 5.10, observando al final tiempos muy igualados en su ejecución global.

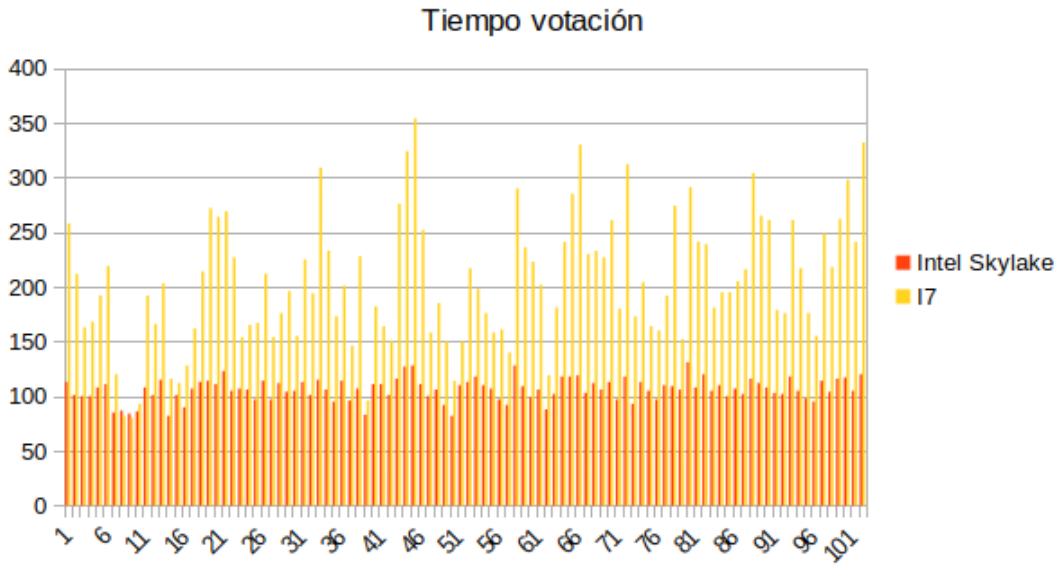


Figura 5.8: **Tiempo fase votación con Intel Skylake en la detección de segmentos** Muestra los tiempos empleados al realizar la fase de votación por el dispositivo Intel Skylake en su implementación en OpenCL frente a su implementación secuencial en la CPU i7 en cada una de las imágenes de un conjunto, indicando el tiempo en su eje vertical y el identificador de la imagen en su eje horizontal.

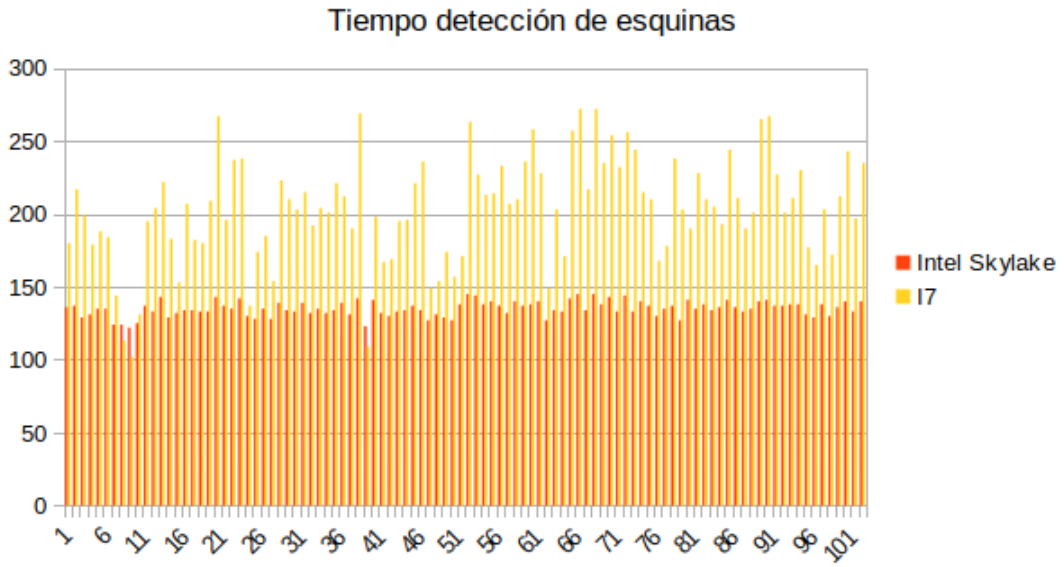


Figura 5.9: **Tiempo fase detección de esquinas con Intel Skylake en la detección de segmentos.** Representa el tiempo empleado en realizar la detección de esquinas y puntos extremos en cada imagen de un conjunto por su implementación secuencial y OpenCL, siendo esta realizada por el dispositivo Intel Skylake. Marcando el tiempo empleado por cada imagen en el eje vertical y en el eje horizontal se muestra en orden numérico su identificador.

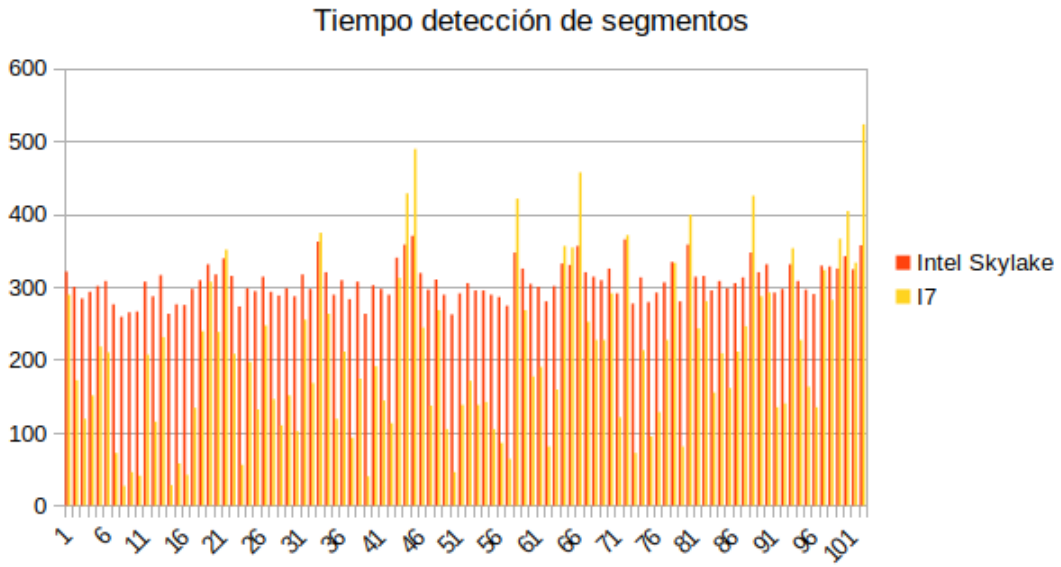


Figura 5.10: **Tiempo fase detección de segmentos con Intel Skylake en la detección de segmentos.** Se visualiza los tiempos consumidos por la implementación secuencial en la CPU i7 y paralela en la i7 Intel Skylake empleadas para realizar la fase de detección de segmentos en una imagen. Realizando la misma operación en un conjunto de imágenes siendo identificadas por su valor numérico, indicado en su eje horizontal y el tiempo requerido en el eje vertical.

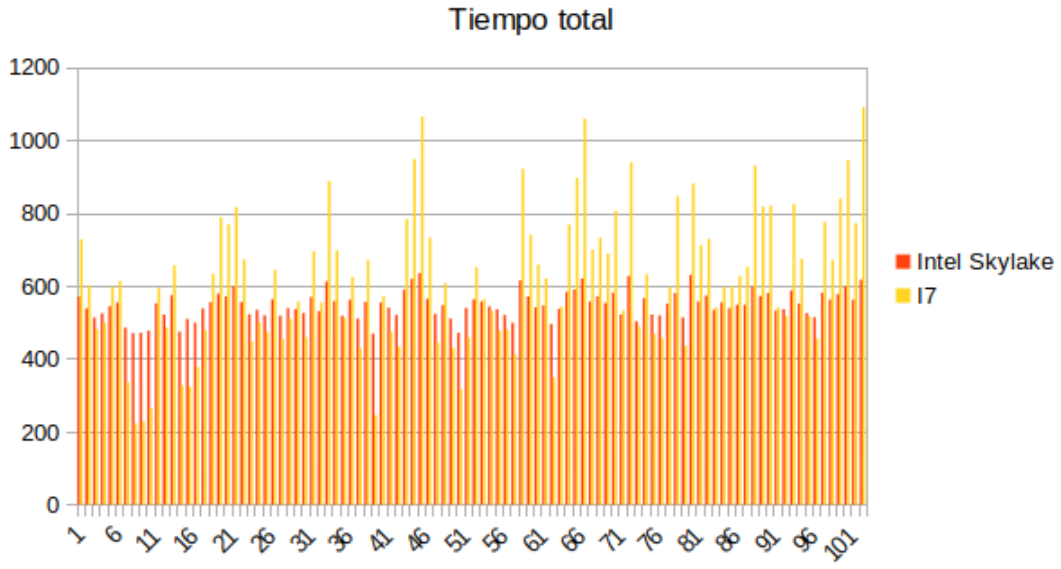


Figura 5.11: **Tiempo total con Intel Skylake en la detección de segmentos.** Representa el tiempo total empleado para realizar las fases de votación, detección de esquinas y detección de segmentos en cada imagen. Representando el tiempo en su eje vertical y el identificador de cada imagen en su eje horizontal.

5.2.3. Comparación tiempos diferentes configuraciones del espacio Hough

Para realizar las pruebas en la detección de segmentos, se han usado tres tipos de configuraciones distintas indicadas en la tabla 5.5, se ha ejecutado en este caso sobre una tarjeta Nvidia RTX 2080 obteniendo de esta prueba el Speedup de cada una de las configuraciones, representado su resultado en la figura 5.12. Como se puede apreciar, la configuración donde se observa una mayor mejora de rendimiento es la primera, consiguiendo mejores resultados sobre una cantidad de imágenes del conjunto mayor que las otras dos configuraciones empleadas. También, hay partes del conjunto donde las tres configuraciones están más equilibradas poniendo como ejemplo las imágenes comprendidas entre la 36 y 41. Aunque hay partes del conjunto donde se produce una rivalidad en eficiencia entre las tres, se considera a la tercera configuración la que peores mejoras proporciona.

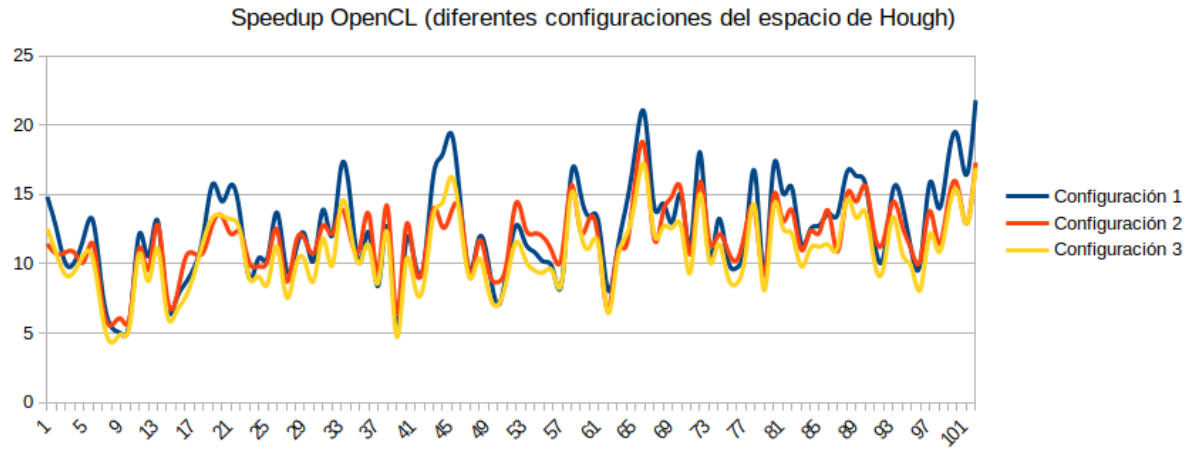


Figura 5.12: **Speedup OpenCL de tres configuraciones con Nvidia RTX 2080 en la detección de segmentos.** Representa el Speedup obtenido al aplicar cada una de las configuraciones ejecutadas sobre la tarjeta gráfica Nvidia RTX 2080. Mostrando la eficiencia en el eje vertical y el identificador de cada imagen en el eje horizontal.

5.2.4. Comparación de tiempos entre las implementaciones de Cuda y OpenCL

La realización de esta prueba se ha llevado a cabo para comprobar como se desenvuelve una aplicación ejecutada sobre OpenCL frente a la misma sobre CUDA en gráficas Nvidia, en este caso se ha utilizado una Nvidia GTX 1080 usando los parámetros de la configuración 1 del espacio de Hough indicada en la tabla 5.5. Para el ejecutable de CUDA se ha utilizado la versión 9 de cuda toolkit. En la versión 10 se obtienen peores resultados. Se muestran la comparativa del Speedup, los tiempos de las fases de votación, detección de esquinas y detección de segmentos, además del tiempo total de ejecución.

Las dos versiones utilizadas en la prueba se han diseñado para mejorar una implementación secuencial convirtiéndose en una implementación paralela, desarrollada sobre dos entornos diferentes pero con la misma finalidad. Comparando los resultados de la figura 5.13 se considera para este caso concreto, que la versión OpenCL proporciona mejores resultados en relación a la versión CUDA.

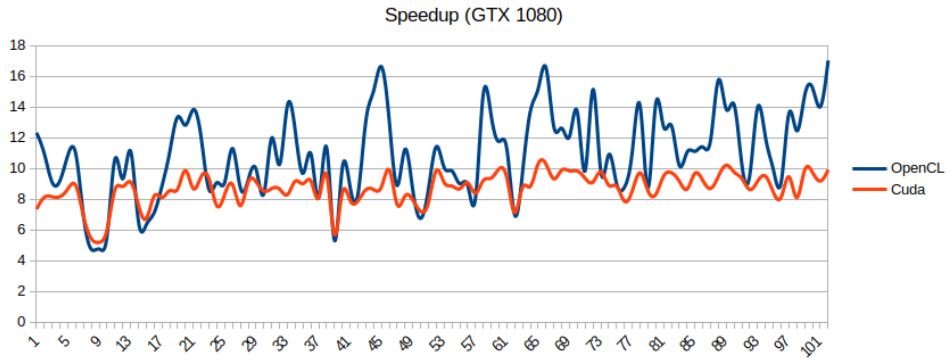


Figura 5.13: **Speedup de Cuda y OpenCL en relación a la ejecución secuencial con Nvidia GTX 1080 en la detección de segmentos.** Representa el Speedup obtenido al ejecutar la aplicación tanto en el entorno OpenCl como Cuda indicando la eficiencia en relación a la versión secuencial marcada en el eje vertical y el identificador de la imagen en el eje horizontal.

Considerando los tiempos individuales de cada fase, se puede observar que durante las fases de votación (Figura 5.14) y detección de esquinas (Figura 5.15) el rendimiento logrado por CUDA es superior al de OpenCL, sin embargo, no ocurre así en la detección de segmentos (Figura 5.16). Mientras que en OpenCL los tiempos de detección de segmento se mantienen similares con independencia del número de características a detecta, en CUDA se observan variaciones significativas de tiempo relacionadas con la complejidad de la imagen. Las distintas pruebas realizadas para tratar de buscar una causa a este hecho no han permitido obtener ninguna conclusión. La implementación original en CUDA se ha adaptado hasta coincidir con su equivalente en OpenCL, utilizando las mismas estructuras de datos, distribución de hilos, operaciones atómicas, etc. Asimismo se han probado distintas opciones de compilación que permitieran generar un código lo más optimizado posible desde las herramientas de CUDA. Sin embargo, ninguno de los cambios realizados ha tenido un efecto significativo, manteniéndose en todos los casos diferencias de tiempo entre los dos frameworks muy parecidos a los mostrados en las figuras 5.16 y 5.17.

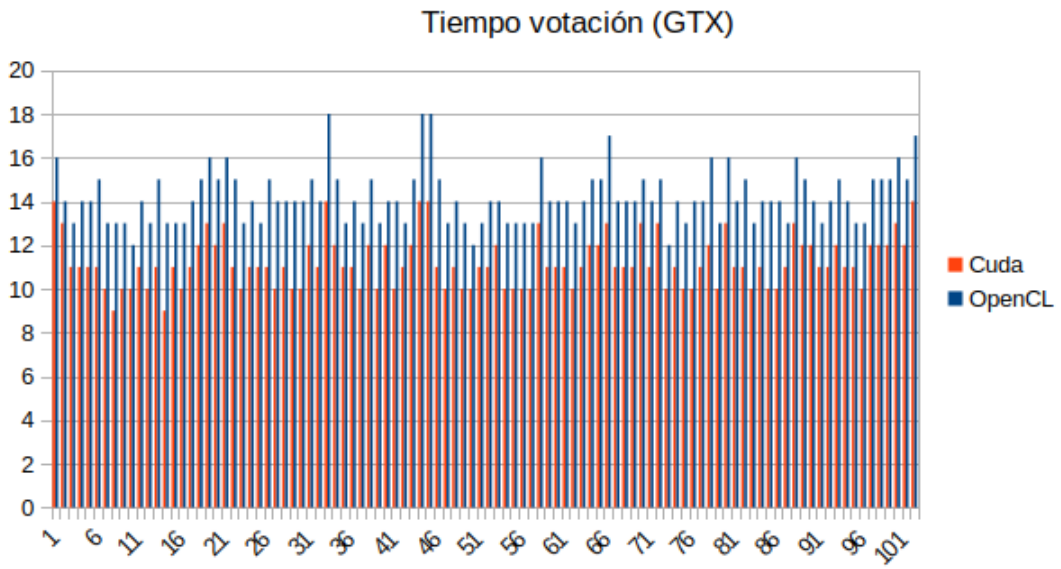


Figura 5.14: **Tiempo fase votación con CUDA y OpenCL en la detección de segmentos.** Se muestran los tiempos empleados al realizar la fase de votación por cada uno de las versiones implementadas sobre OpenCL y CUDA en cada una de las imágenes de un conjunto, indicando el tiempo en su eje vertical y el identificador de la imagen en su eje horizontal.

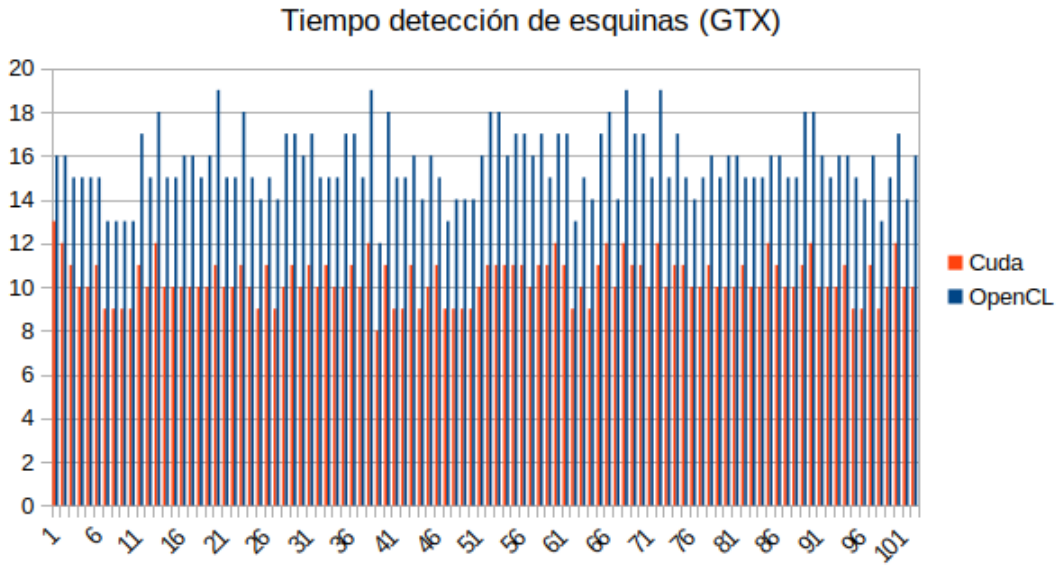


Figura 5.15: **Tiempo fase detección de esquinas con CUDA y OpenCL en la detección de segmentos.** Representa el tiempo empleado en realizar la detección de esquinas y puntos extremos en cada imagen de un conjunto por su implementación CUDA y OpenCL. Marcado el tiempo empleado por cada imagen en el eje vertical y en el eje horizontal se muestra en orden numérico su identificador.

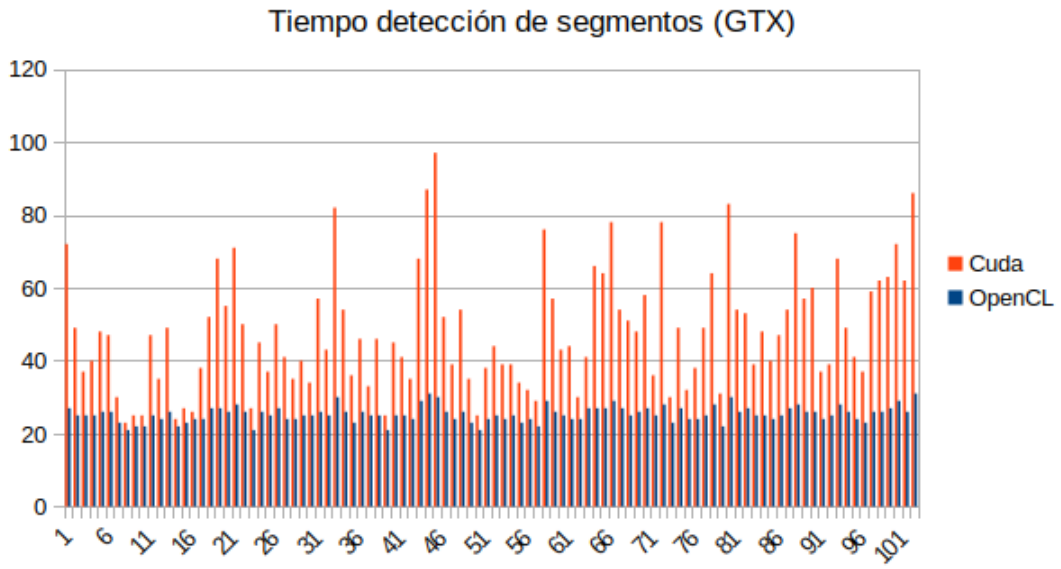


Figura 5.16: **Tiempo fase detección de segmentos con CUDA y OpenCL en la detección de segmentos.** Se visualiza los tiempos utilizados por la implementación de CUDA y OpenCL en la fase de detección de segmentos en un conjunto de imágenes siendo identificadas por su valor numérico, indicado en su eje horizontal y el tiempo requerido en el eje vertical.

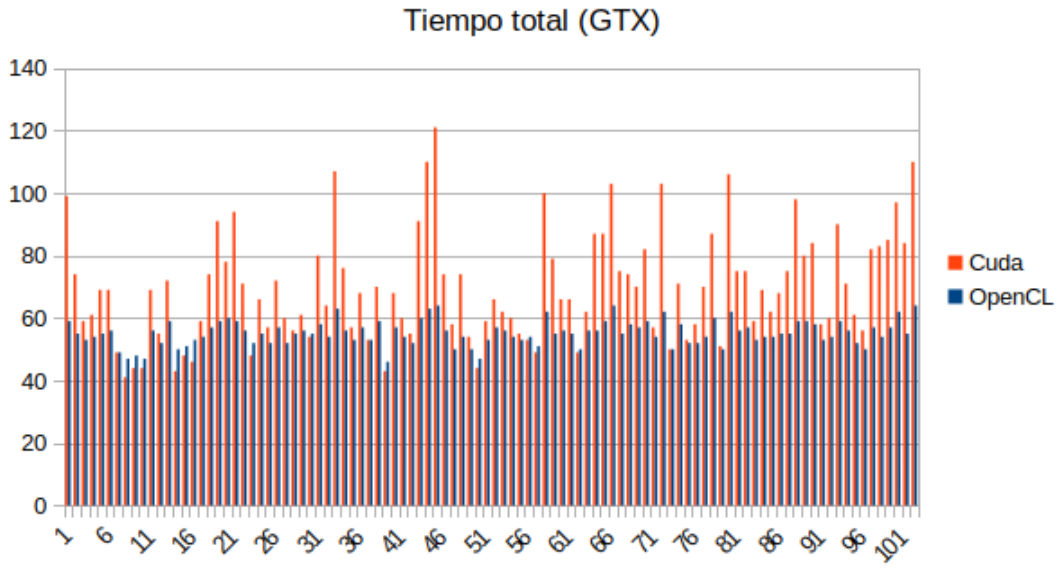
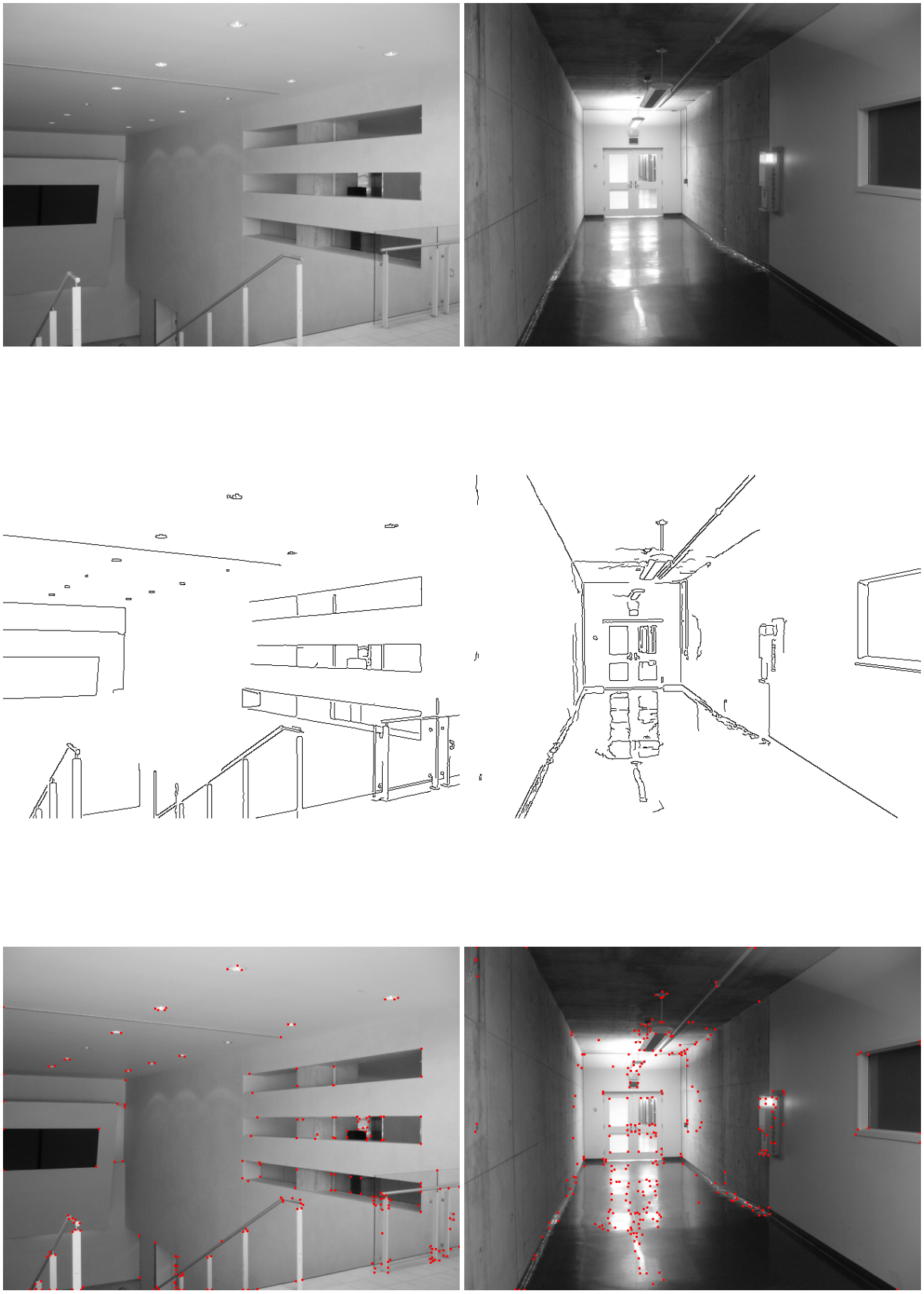
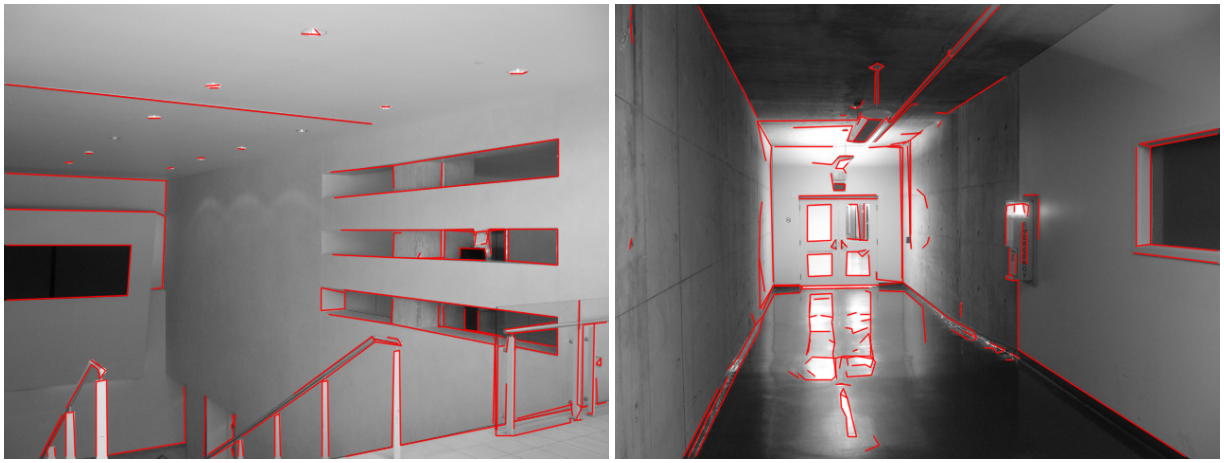


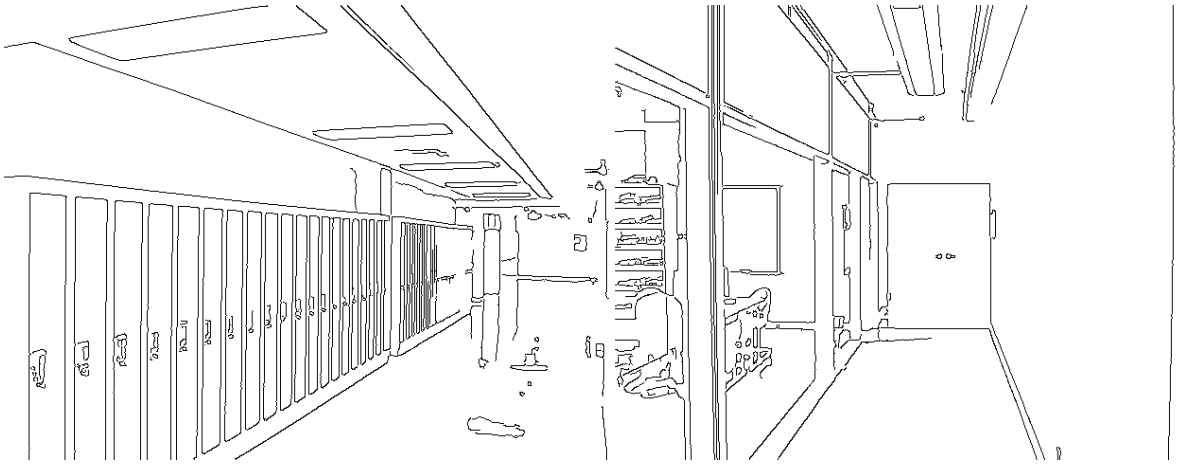
Figura 5.17: **Tiempo total con versión CUDA y OpenCL en la detección de segmentos.** Representa el tiempo total empleado por cada dispositivo para realizar las fases de votación, detección de esquinas y detección de segmentos en cada imagen. Representando el tiempo en su eje vertical y el identificador de cada imagen en su eje horizontal.

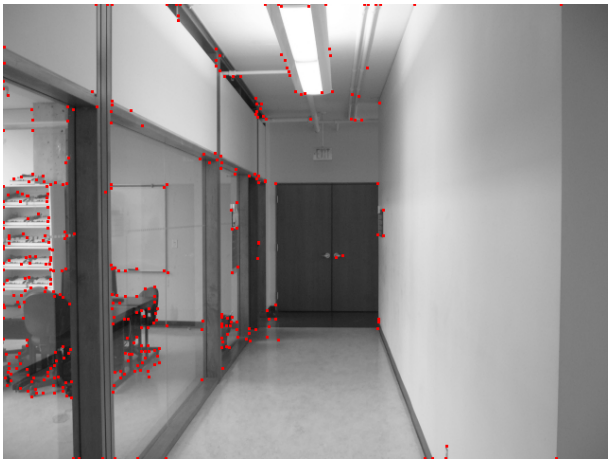
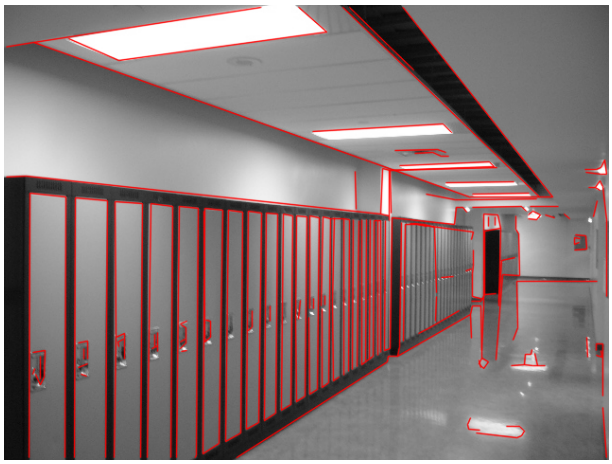
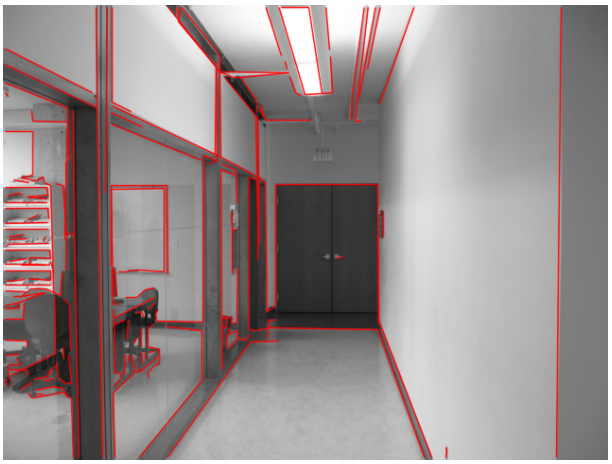
5.2.5. Resultados de detección de esquinas y segmentos

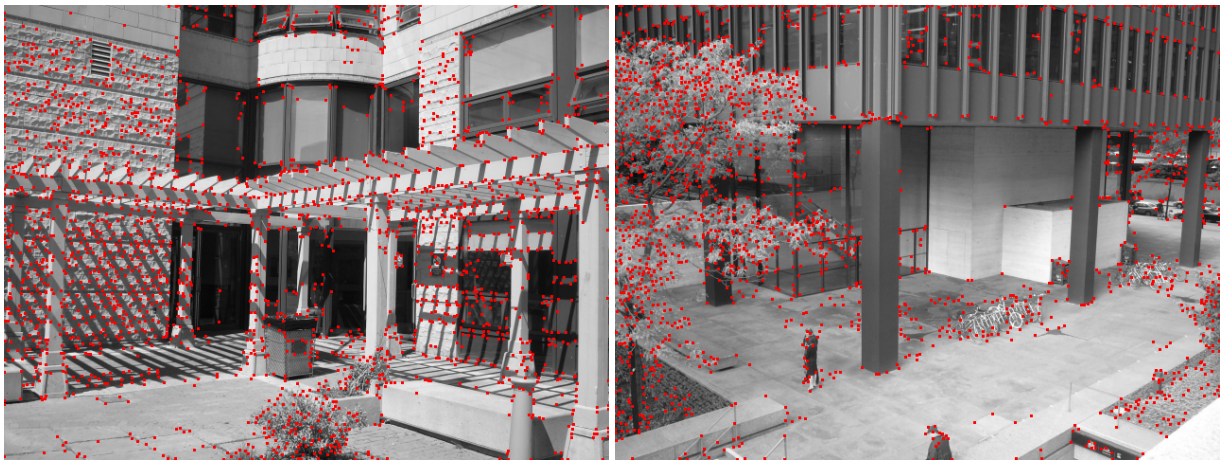
Los resultados de detección que se muestran en esta sección se han obtenido aplicando la configuración 1 del espacio de Hough indicada en la tabla 5.5. Por cada resultado, se muestra la imagen original, la imagen de bordes y, sobre la imagen original, la detección de esquinas y segmentos por separado. Se han utilizado para esta muestra tres parejas de imágenes, representativas de los distintos rangos de Speedup obtenidos, siendo los rangos de la primera pareja valores de Speedup entre 4 y 5, los de la segunda pareja entre 7 y 9, y los de la tercera pareja 14 y 16. Como puede observarse, el incremento del rendimiento está directamente relacionado con la complejidad de las imágenes y el número de características a detectar.

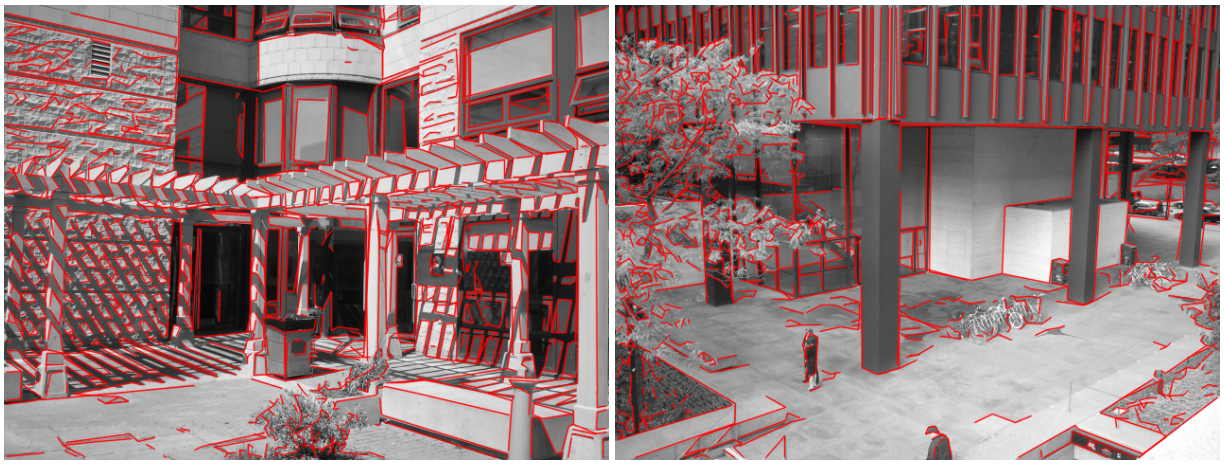










5.3. Pruebas Detección de Rectángulos

5.3.1. Comparación de tiempos entre la implementación secuencial y OpenCL en Nvidia

Esta prueba ha sido realizada sobre tarjetas gráficas Nvidia utilizando la configuración 1 de la tabla 5.6. Se realiza una comparación de los resultados obtenidos por la gráfica Nvidia RTX 2080 y Nvidia GTX 1080, mostrando la comparativa del Speedup, los tiempos de las fases de votación, detección de esquinas y detección de rectángulos, además del tiempo total de ejecución.

Al igual que en la detección de segmentos, la mejora en la detección de rectángulos se debe a la distribución del procesamiento entre los distintos hilos. Se produce una mejora global en relación a la versión secuencial entre cinco y veinte veces más eficiente, dependiendo de la imagen analizada y del dispositivo empleado. Según se aprecia en la figura 5.18, los resultados conseguidos por la RTX son siempre mejores considerando si se mejora el hardware se pueden lograr mejores resultados.

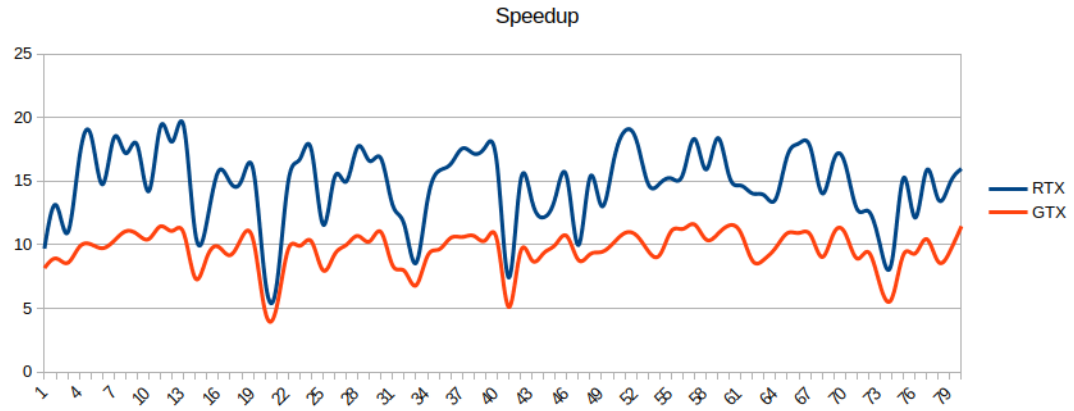


Figura 5.18: **Speedup global de OpenCL en relación a la versión secuencial entre RTX 2080 y GTX 1080 en la detección de rectángulos.** Representa el Speedup obtenido en relación al secuencial tras operar sobre las gráficas Nvidia GTX 1080 y RTX 2080. Indicando el nivel de eficiencia en el eje vertical e identificando a cada imagen del conjunto en el eje horizontal.

En las siguientes gráficas se muestran el tiempo empleado por cada una de las fases al ejecutar su funcionalidad sobre cada una de las imágenes. Comparando los resultados se observa

la gran ganancia en tiempo de un la implementación paralela frente a la secuencial. Así, en la fase de votación (Figura 5.19) y en la fase de detección de esquinas (Figura 5.20) los tiempo conseguidos por las GPUs RTX y GTX son ínfimos en relación a la versión secuencial ejecutada sobre el procesador i7.

Aunque la diferencia de tiempos entre la ejecución secuencial y la paralela es muy alta, se pueden ver situaciones en la fase de detección de rectángulos (Figura 5.21) donde esta diferencia es más reducida. Concretamente, estos casos están relacionados con imágenes con pocos puntos de borde y un número reducido de características detectables. A pesar de ello, la ejecución paralela proporciona siempre mejores resultados de tiempo que la secuencial.

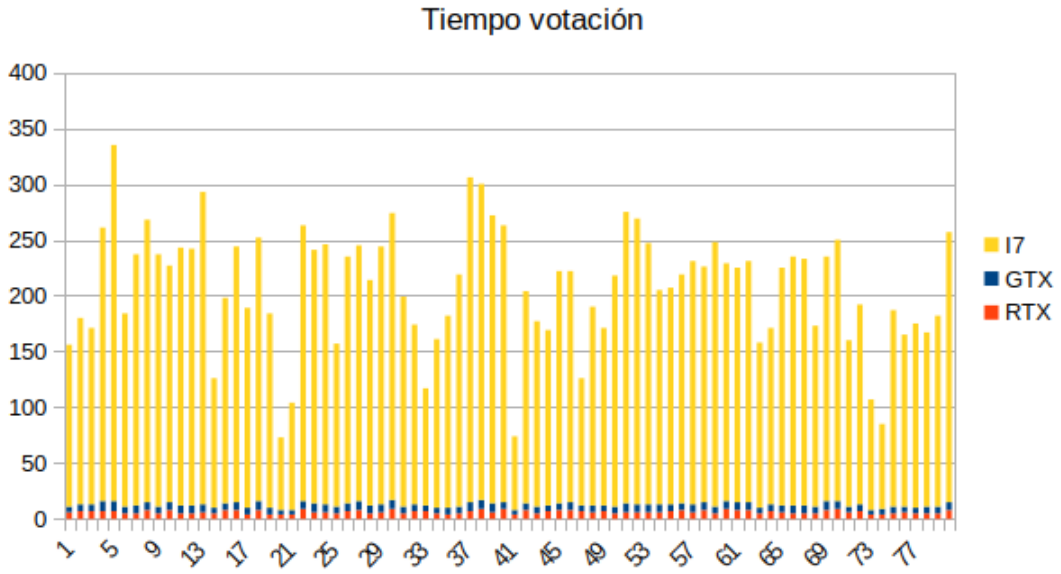


Figura 5.19: **Tiempo fase votación entre RTX 2080 y GTX 1080 en la detección de rectángulos.** Se muestran los tiempos empleados al realizar la fase de votación por cada uno de los dispositivos indicados, en cada una de las imágenes de un conjunto, indicando el tiempo en su eje vertical y el identificador de la imagen en su eje horizontal.

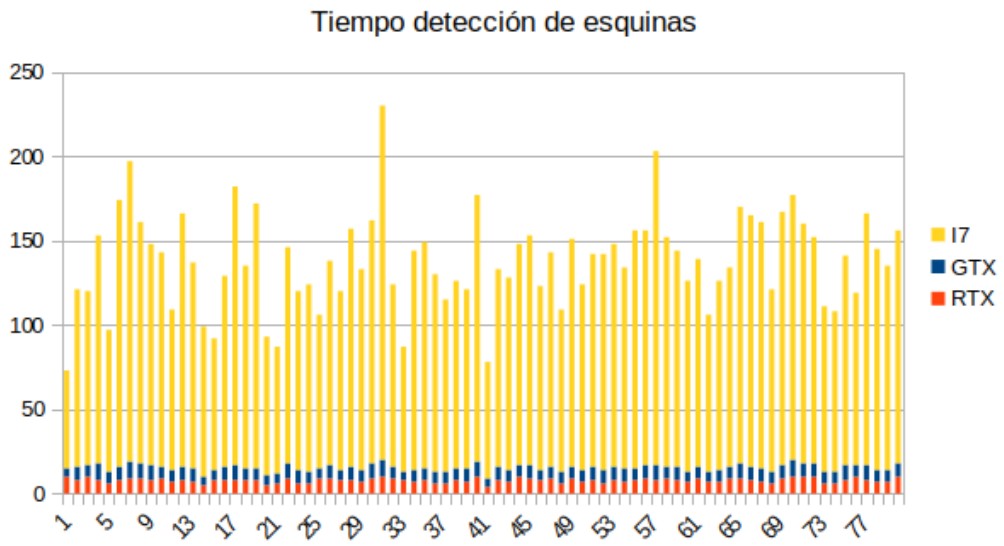


Figura 5.20: **Tiempo fase detección de esquinas entre RTX 2080 y GTX 1080 en la detección de rectángulos.** Representa el tiempo empleado en realizar la detección de esquinas y puntos extremos en cada imagen de un conjunto dado por los tres dispositivos mencionados. Marcando el tiempo empleado por cada imagen en el eje vertical y en el eje horizontal se muestra en orden numérico su identificador.

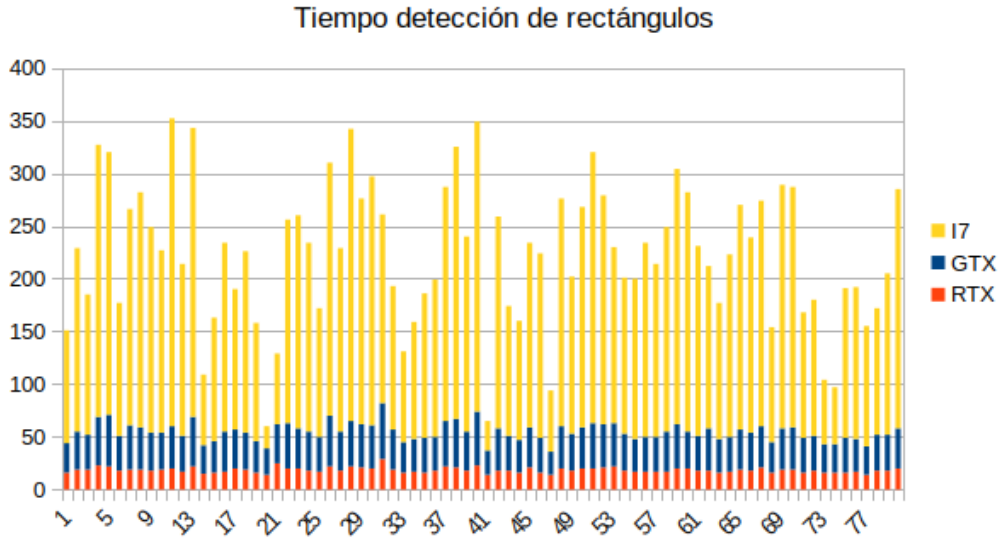


Figura 5.21: **Tiempo fase detección de rectángulos entre RTX 2080 y GTX 1080 en la detección de rectángulos.** Muestra el tiempo consumido en realizar la detección de rectángulos en cada una de las imágenes de un conjunto dado comparándose con el tiempo secuencial obtenido por un i7. Se indica el tiempo consumido en el eje vertical y el identificador de la imagen en el eje horizontal.

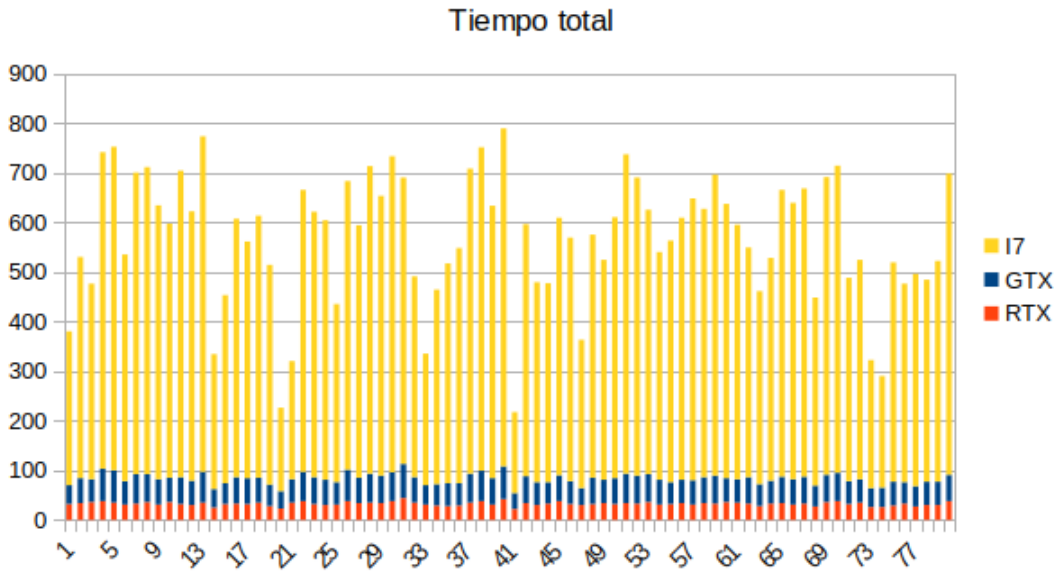


Figura 5.22: **Tiempo total de ejecución entre RTX 2080 y GTX 1080 en la detección de rectángulos.** Representa el tiempo total empleado por cada dispositivo para realizar las fases de votación, detección de esquinas y detección de segmentos en cada imagen. Representando el tiempo en su eje vertical y el identificador de cada imagen en su eje horizontal.

5.3.2. Comparación tiempos implementación secuencial y OpenCL en Intel Skylake

Esta prueba ha sido realizada sobre una gráfica integrada de Intel (Intel Skylake) donde se realiza la comparación de los resultados obtenidos por el modelo secuencial y la versión paralela de OpenCL utilizando la configuración 1 indicada en la tabla 5.6. Se muestra el Speedup tanto global como por fases, así como los tiempos en las fases de votación, detección de esquinas, detección de rectángulos y tiempo total de ejecución.

Para comentar los resultados obtenidos es necesario realizar una referencia a la gráfica representada en la figura 5.7 de la detección de segmentos, realizando la comparación con la figura 5.23 existiendo la diferencia en la ejecución de un algoritmo los resultados obtenidos han mejorado, partiendo de que el algoritmo de detección de rectángulos es más complejo y la gráfica de Intel Skylake no es tan potente, si se confrontan la detección de rectángulos de la figura 5.24 con la detección de segmentos de la figura 5.7 se obtienen resultados similares en

eficiencia, consiguiendo que sea favorable la paralelización.

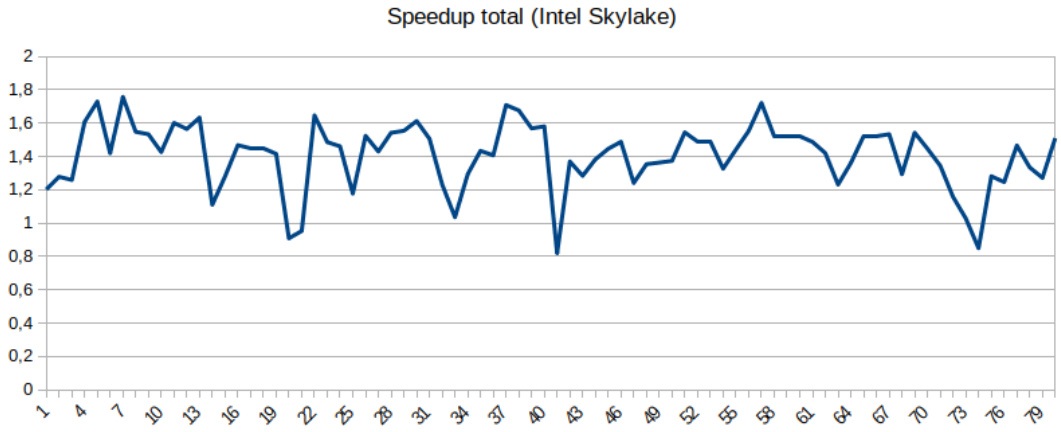


Figura 5.23: **Speedup global en relación a la versión secuencial con Intel Skylake en la detección de rectángulos.** Visualiza el Speedup conseguido tras concluir la ejecución sobre el conjunto de imágenes dadas identificadas en el eje horizontal y su respectiva eficiencia en el eje vertical.

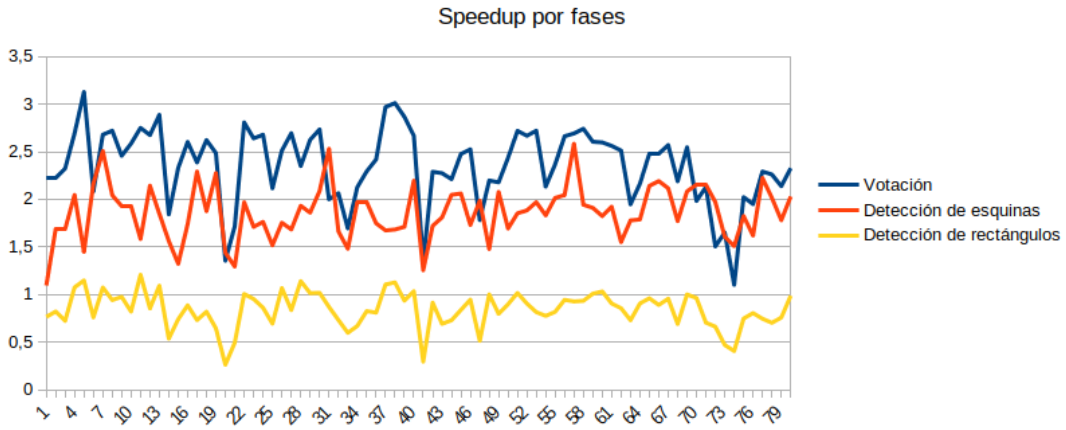


Figura 5.24: **Speedup por fases en relación a la versión secuencial con Intel Skylake en la detección de rectángulos.** Representa el Speedup obtenido por cada fase tras aplicar a un conjunto de imágenes cada uno de los algoritmos, siendo representado la eficiencia de cada una de las fases en el eje vertical de la gráfica y la identificación de la imagen tratado en el eje horizontal.

Si se contempla los tiempo conseguidos en cada fase por separado, siempre se consigue mejores resultados en tiempo con la versión OpenCL frente a la secuencial en las fases de votación y detección de esquinas (Figura 5.25 y Figura 5.9) siendo los obtenidos por Skylake

la mitad de la versión secuencial, sólo son próximos aquellas imágenes con poca densidad de rangos distintivos.

En base a todas la pruebas realizadas, donde se ha visto una mejora de rendimiento es en la fase de detección de rectángulos, como ya se ha comentado, aunque es un algoritmo complejo ha conseguido igualar su resultados al algoritmo secuencial, caso que no es igual en la situación dada en la detección de segmentos de la figura 5.10 siendo los tiempos conseguidos por la versión de OpenCL peores. Estos tiempos son la causa de la transformación de una versión a otra, ya que la versión paralela utiliza estructuras diferentes a la implementación secuencial consiguiendo mejorar el rendimiento como se puede observar en las dos primeras fases, aunque en algunas ocasiones puede perjudicar.

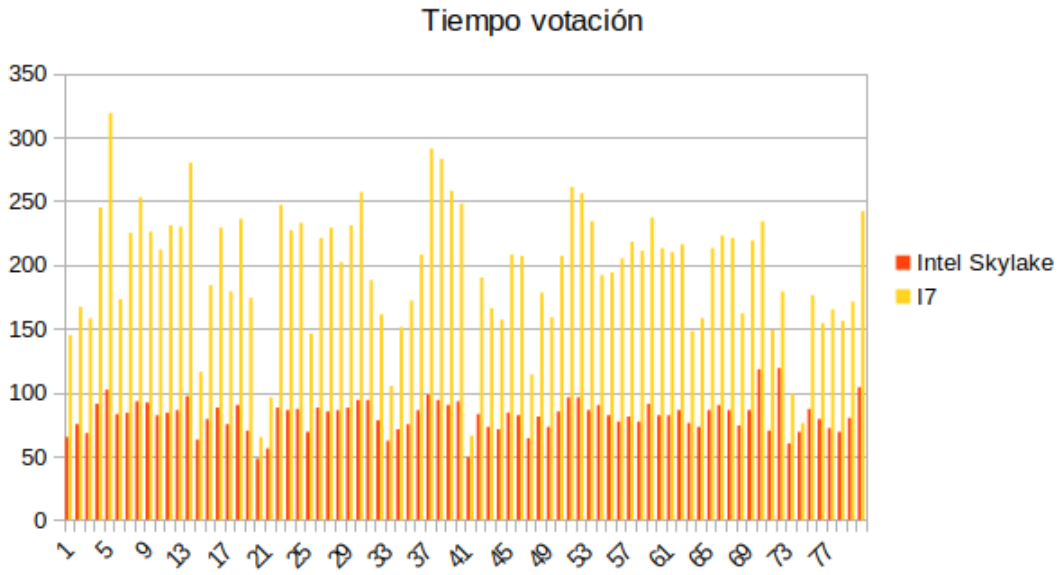


Figura 5.25: **Tiempo de votación con Intel Skylake en la detección de rectángulos.** Muestra los tiempos obtenidos al aplicar el algoritmo de votación sobre cada imagen de un conjunto, ejecutado de forma secuencial por el procesador i7 y por la versión OpenCL paralela en la gráfica Intel Skylake. Indicando el tiempo empleado por imagen en el eje vertical y la imagen tratada en el eje horizontal.

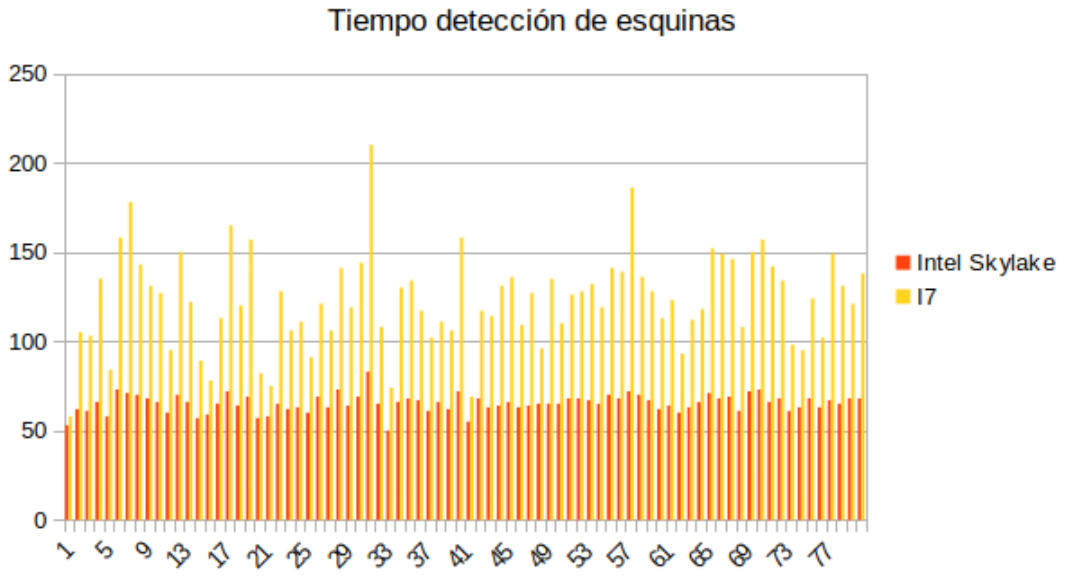


Figura 5.26: **Tiempo detección de esquinas con Intel Skylake en la detección de rectángulos.** Se imprime en la gráfica los resultados de tiempos obtenidos por Intel Skylake de la versión OpenCL y de la CPU i7 en su versión secuencial. Se marca en el eje vertical el tiempo empleado y en el eje horizontal la imagen analizada.

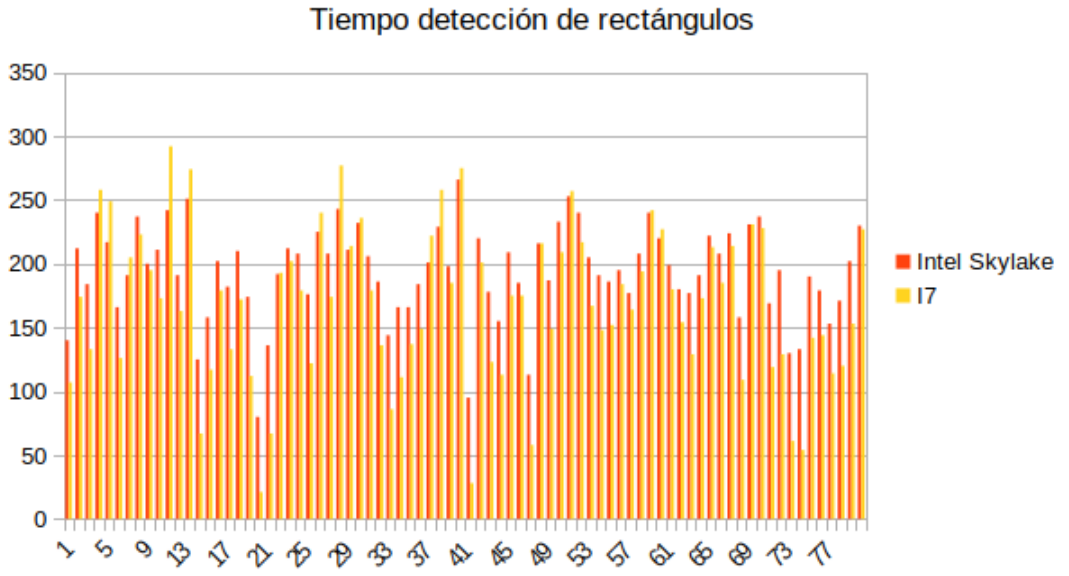


Figura 5.27: **Tiempo detección de rectángulos con Intel Skylake en la detección de rectángulos.** Se representan los tiempos empleados al ejecutar la versión secuencial y paralela realizada esta última por Intel Skylake, indicándose el tiempo usado en el eje vertical y la identificación de la imagen en el eje horizontal.

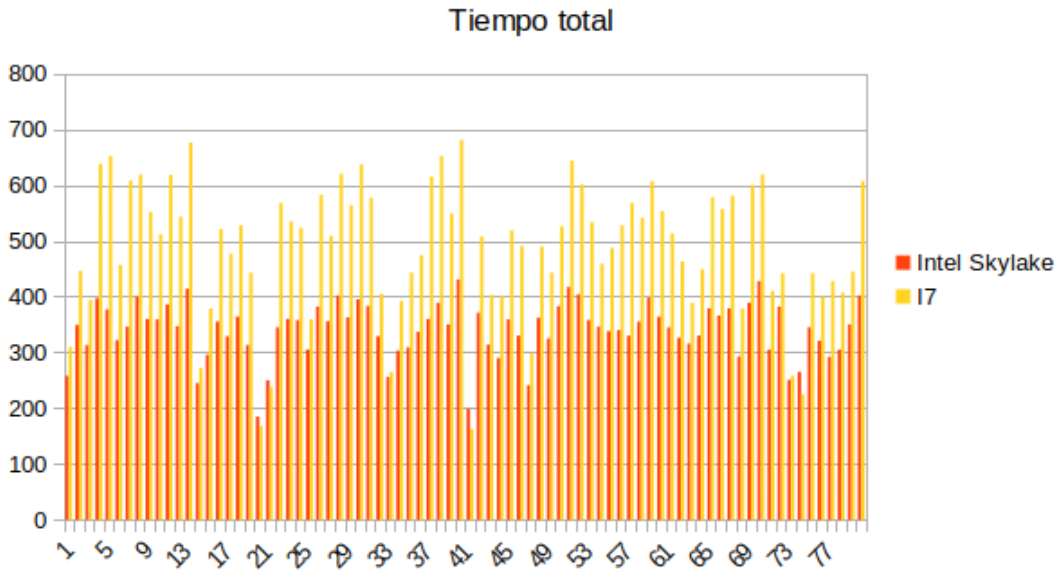


Figura 5.28: **Tiempo total con Intel Skylake en la detección de rectángulos.** Se muestra el tiempo total empleado por cada dispositivo para realizar las fases de votación, detección de esquinas y detección de segmentos en cada imagen. Representando el tiempo en su eje vertical y el identificador de cada imagen en su eje horizontal.

5.3.3. Comparación tiempos diferentes configuraciones del espacio Hough

Para realizar las pruebas en la detección de rectángulos, se han usado dos tipos de configuraciones distintas indicadas en la tabla 5.6. Se ha ejecutado en este caso sobre una tarjeta Nvidia RTX 2080 obteniendo el Speedup de cada una de las configuraciones (Figura 5.29). Si se comparan los resultados generados por las dos configuraciones se aprecia claramente que la diferencia de rendimiento entre la versión secuencial y la paralela aumenta cuanto mayor es el número de celdas del espacio de Hough. Así, para la configuración 1 se obtiene un Speedup que supera hasta en 5 puntos al equivalente con la configuración 2.

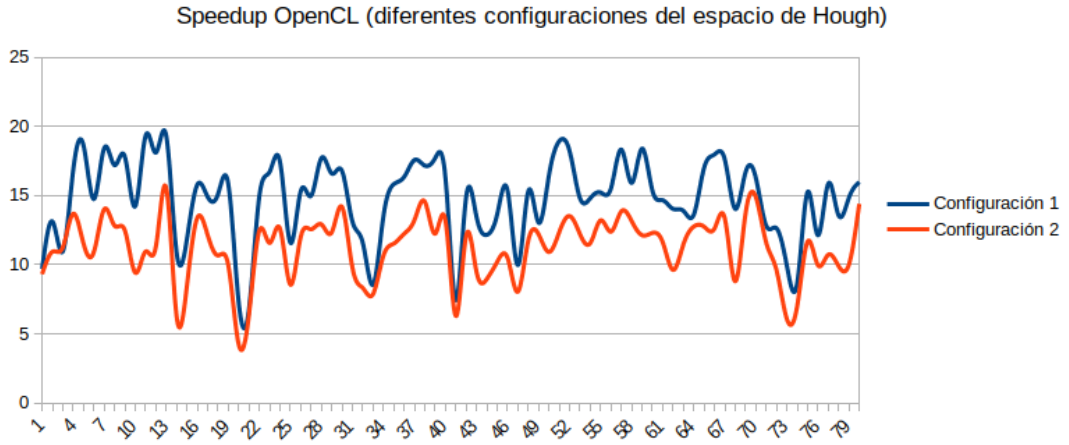
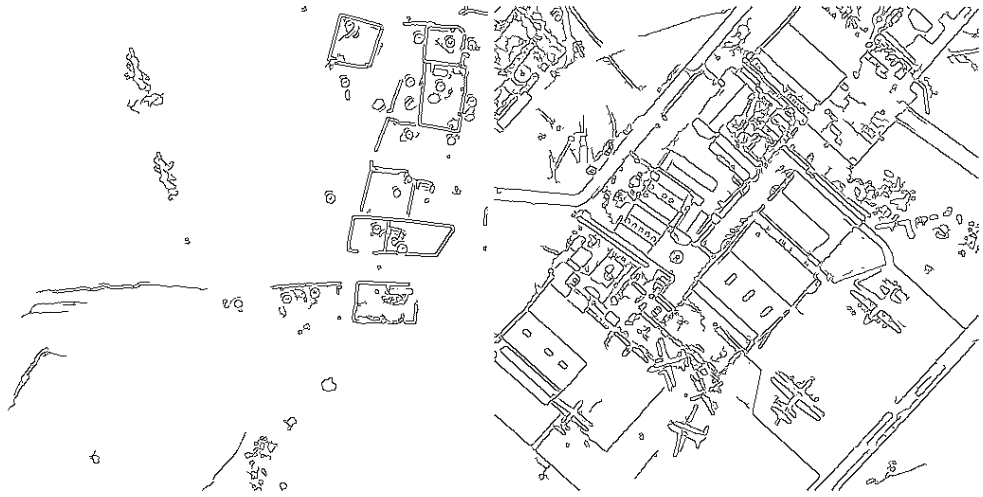
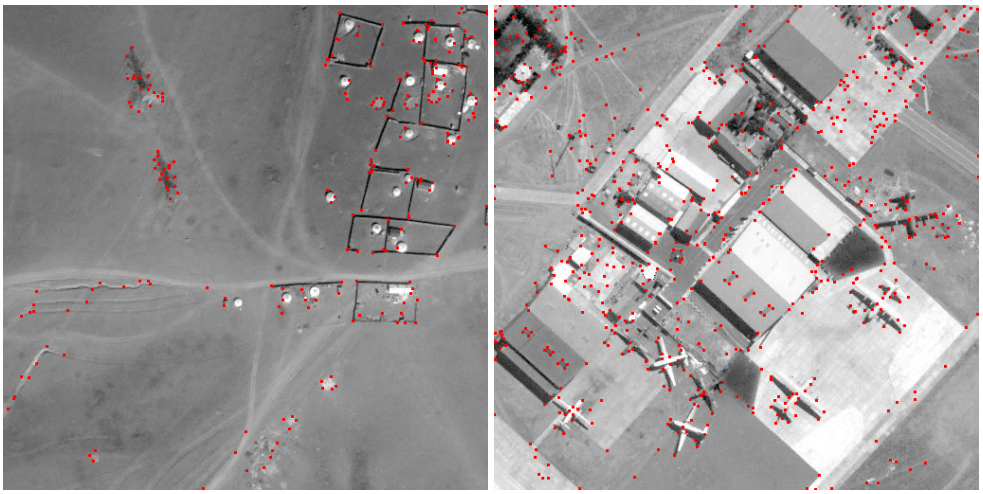


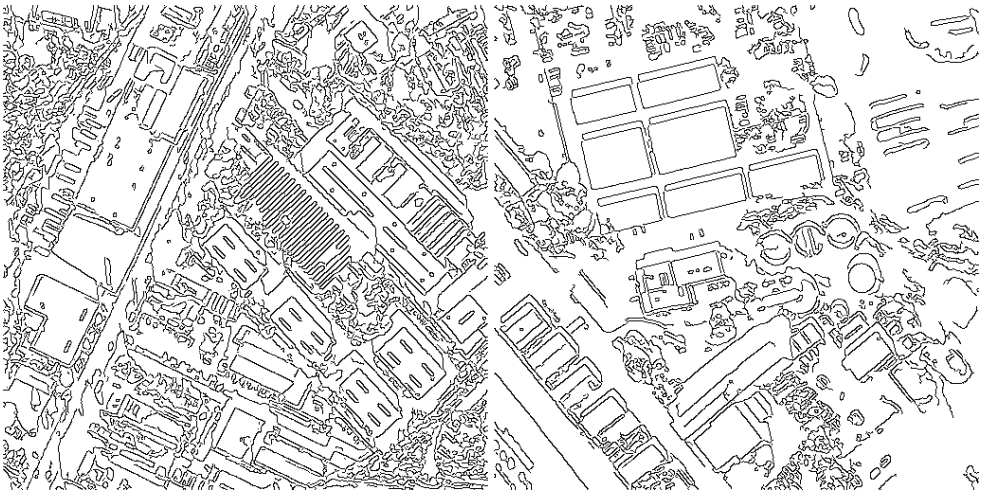
Figura 5.29: **Speedup OpenCL con distintas configuraciones en la detección de rectángulos.** Se representa el Speedup conseguido con dos configuraciones diferentes, obtenidas por gráfica Nvidia RTX 2080. Representado la eficiencia en el eje vertical y el conjunto de imágenes en el eje horizontal.

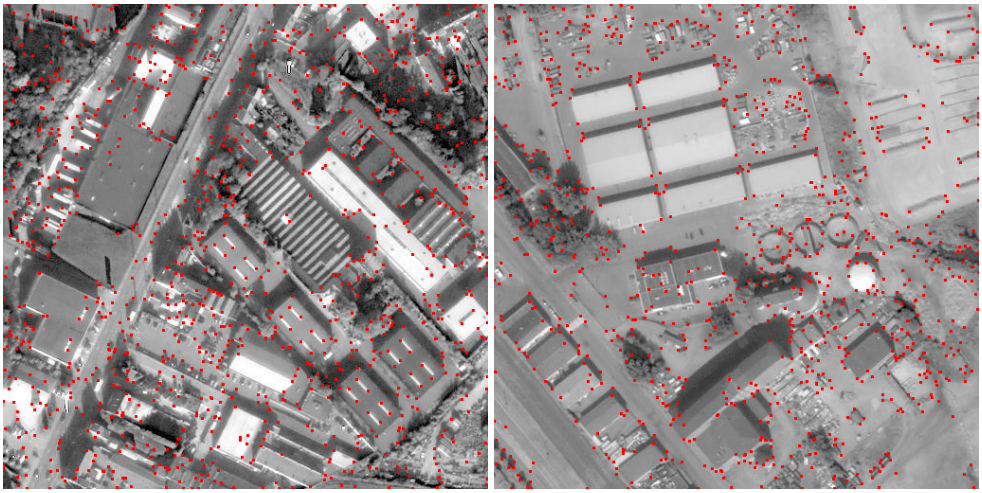
5.3.4. Resultados de detección de esquinas y rectángulos

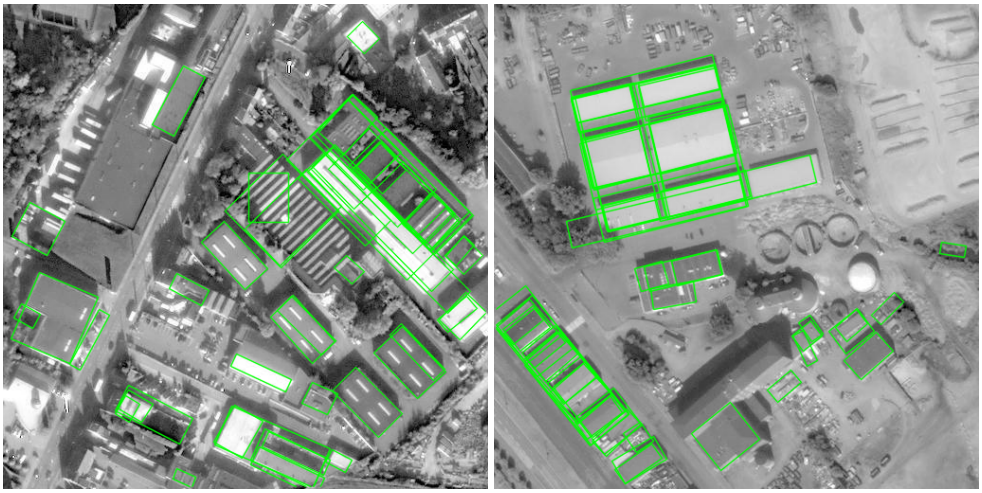
Los resultados de detección de rectángulos se han obtenido aplicando la la configuración 1 del espacio de Hough indicada en la tabla 5.6. Se incluyen en esta sección dos parejas de imágenes representativas de los distintos rangos de Speedup. La primera pareja se corresponde con un Speedup entre 7 y 8 y la segunda entre 15 y 16. Por cada imagen, se muestran la imagen original, la imagen de bordes y, sobre la imagen original, la detección de esquinas y rectángulos por separado.











Capítulo 6

Conclusiones y trabajos futuros

En este trabajo se ha desarrollado una implementación paralela de la transformada de Hough 3D, conocida como HT3D, para la detección en tiempo real de esquinas, segmentos y rectángulos en imágenes. Para detección de esquinas y segmentos, los resultados muestran que la velocidad de procesamiento en gráficas Nvidia se sitúa entre los 16 y los 20 f.p.s en función del modelo, mientras que la versión secuencial presenta una velocidad de procesamiento media de 1.25 f.p.s. Además de esta notable diferencia de rendimiento, la versión paralela muestra resultados de tiempo muy similares para imágenes de diferente complejidad (número y distribución de puntos de borde), mientras que la secuencial es claramente dependiente del número de características presentes en la imagen. Para la detección de rectángulos, pueden sacarse conclusiones similares, aunque en este caso la variación de tiempos es mayor que en la detección de segmentos.

Además de la ganancia en rendimiento, el uso del framework OpenCL permite independizar en gran medida la implementación del hardware utilizado, lo que supone una clara ventaja frente a una implementación con CUDA. No obstante, son necesarias pruebas adicionales para conocer el rendimiento de la implementación desarrollada en tarjetas gráficas de otros fabricantes que presenten características comparables a las gráficas de Nvidia. Este análisis se plantea dentro de las posibles líneas de trabajo futuro.

Otra cuestión abierta a partir del trabajo desarrollado es la inesperada diferencia de rendimiento

entre las implementaciones CUDA y OpenCL para la detección de segmentos. Tal y como se ha comentado, las distintas pruebas realizadas para tratar de encontrar una causa al empeoramiento de la versión CUDA en relación a la versión OpenCL no han permitido establecer ninguna conclusión. Una línea de actuación en este sentido podría consistir en implementar en ambos frameworks un problema que incluya los distintos elementos presentes en la detección de segmentos (bucles, operaciones atómicas, accesos concurrentes a varias estructuras de datos), incluyendo estos elementos de manera progresiva para poder determinar que aspectos de la implementación son causantes de la diferencia de rendimiento.

Bibliografía

[1] Pedro Domínguez Moralejo and Tutor Pilar Bachiller Burgos. Paralelización de un método de detección de esquinas, segmentos y polílineas en imágenes, 2016. [TFG; Septiembre 2016].

[2] Pilar Bachiller-Burgos, Luis J. Manso, and Pablo Bustos. A variant of the hough transform for the combined detection of corners, segments, and polylines. *EURASIP Journal on Image and Video Processing*, 2017(1):32, May 2017.

[3] Pilar Bachiller-Burgos, Luis J. Manso, and Pablo Bustos. A spiking neural model of ht3d for corner detection. *Frontiers in Computational Neuroscience*, 12:37, 2018.

[4] Wikipedia. Computación paralela — wikipedia, la enciclopedia libre, 2019. [Internet; descargado 16-julio-2019].

[5] Wikipedia. Gpgpu — wikipedia, la enciclopedia libre, 2019. [Internet; descargado 16-julio-2019].

[6] Wikipedia contributors. Opencl — Wikipedia, the free encyclopedia, 2019. [Online; accessed 16-July-2019].

[7] Sceptic Philozoff. Opencl:de una programación simple a una más intuitiva, 2014. [Internet; descargado 16-julio-2019].

[8] Khronos® OpenCL Working Group. Khronos opencl registry - opencl 2.2 api specification (platform model), 2019. [Internet; descargado 16-julio-2019].

[9] David Kaeli and Dong Ping Zhang. Sciencedirect opencl platform (introduction to opencl), 2015. [Internet; descargado 16-julio-2019].

[10] Ying Zhu. Basic concepts in opencl (context), 2018. [Internet; descargado 16-julio-2019].

[11] Khronos® OpenCL Working Group. Khronos opencl registry - opencl 2.2 api specification (command queues), 2019. [Internet; descargado 16-julio-2019].

[12] Khronos® OpenCL Working Group. Khronos opencl registry - opencl 2.2 api specification (program objects), 2019. [Internet; descargado 16-julio-2019].

[13] Khronos® OpenCL Working Group. Khronos opencl registry - opencl 2.2 api specification (kernel objects), 2019. [Internet; descargado 16-julio-2019].

[14] kronos group Editor: Aaftab Munshi. Opencl 1.0 specification, 2009. [Online pdf; accessed 19-Julio-2019].

[15] Ying Zhu. Basic concepts in opencl (work-group), 2018. [Internet; descargado 26-julio-2019].

[16] Ying Zhu. Basic concepts in opencl (work-item), 2018. [Internet; descargado 26-julio-2019].

[17] Nvidia CUDA Zone. Cuda zone, 2019. [Internet; descargado 16-julio-2019].

[18] CHW CHW. Amd: Pasado, presente y futuro del gpgpu, 2010. [Internet (FayerWayer); descargado 16-julio-2019].