Diseño e implementación en hardware de un modelo de retina artificial

Autores:

Álvaro Gil Sánchez

Héctor Gutiérrez Palancarejo

Álvaro Morales Lozano

Proyecto de Sistemas Informáticos

Facultad de Informática

UNIVERSIDAD COMPLUTENSE DE MADRID

2014 / 2015

Directores:

Guillermo Botella Juan

Carlos García Sánchez

Autorización de difusión y utilización

Autorizamos al personal de la Biblioteca de la UCM a conservar, utilizar y, si es preciso, difundir, tanto el código de las aplicaciones desarrolladas en este trabajo, como el presente material de documentación; contando con la correspondiente garantía de conservación y protección de la totalidad del proyecto.

Álvaro Gil Sánchez

Héctor Gutiérrez Palancarejo

Álvaro Morales Lozano

A nuestros padres, por su incondicional apoyo, sin el cual no habríamos logrado cumplir con nuestro propósito.

Prólogo

La discapacidad en los seres humanos es el término que hace referencia a las limitaciones en las funciones corporales, y a las restricciones en el desarrollo de actividades individuales o la participación social. La discapacidad es la consecuencia de la deficiencia funcional o estructural de algún/os órganos corporales, tal y como establece la Organización Mundial de la Salud (OMS) [1], las deficiencias funcionales se agrupan en tres grandes grupos: deficiencias físicas, deficiencias mentales y deficiencias sensoriales; aunque se ha de considerar que en cada grupo existe un alto grado de homogeneidad, producto de la diversidad en los casos concretos de discapacidad.

Para valorar el grado de discapacidad visual se utiliza la Agudeza Visual (AV), que mide "la imagen retiniana más pequeña cuya forma pueda apreciarse", haciendo uso de lentes estandarizadas (optotipos). Una persona con visión normal tiene una agudeza visual de 1, que se suele indicar en escala francófona con 10/10. De entre los muchos métodos que existen para medir la agudeza visual, en Europa, y también en nuestro país, se conoce y utiliza el test de Snellen; aunque con niños o personas que no saben leer se emplean otros test, como el de Lea, o el de Landolt. La OMS establece[1] el límite de la discapacidad visual en los siguientes valores.

- Visión normal: AV = 10/10
- Discapacidad visual :
    - Moderada: AV < 3/10
    - Grave: AV < 1/10
    - Ceguera: AV < 1/20

Según la OMS, en el año 2015, en torno a 285 millones de personas presentan algún tipo de discapacidad visual, de las que 39 millones sufren ceguera. El 90% de los ciegos, se concentra en países en vías de desarrollo. Pese al esfuerzo de algunas organizaciones, como la propia OMS, aún es poco el esfuerzo social que fijan los políticos en combatir la ceguera, cuando se trata de una enfermedad en la que el 80% de los casos tienen margen de mejora. La discapacidad visual general, se divide en problemas de baja visión y problemas de ceguera; sin embargo, y pese al escaso esfuerzo en combatir esta clase de problemas, las sociedades se encuentran más sensibilizadas con el segundo tipo de dolencias, pasando casi inadvertidas las del primer grupo, el de la baja visión. Por todo esto, es vital introducir sistemas de mejora de la percepción en discapacitados visuales, que puedan ayudar al normal desarrollo de sus actividades individuales o sociales. Actualmente, en las sociedades, solo se implantan sistemas de mejora de la percepción a pacientes con problemas oculares del tipo reflexivo, es decir, a personas que sufren enfermedades como la miopía, hipermetropía o astigmatismo. Para los pacientes con problemas de baja visión, existen ayudas de tipo óptico, como las gafas de gran aumento las lupas, los telescopios, telemicroscopios y filtros ópticos; pero no suelen contar con ayudas de

tipo informático. Con los avances tecnológicos actuales en computación, que han hecho posible que personas con un poder adquisitivo moderado, puedan disponer de potentes procesadores de bajo consumo energético en sus terminales móviles; no es disparatado pensar en el desarrollo de aplicaciones informáticas que puedan apoyar la percepción visual de pacientes que sufren algún tipo de problema de baja visión, no relacionado con la reflexión de la luz en el ojo.

El envejecimiento de la población mundial, que aumenta el número de casos de degeneración macular asociada a la edad (DMAE), hace cada vez más presente la discapacidad visual en los países desarrollados; así como el crecimiento de los casos de diabetes, causante del tipo de ceguera producida por la retinopatía diabética. Otras dolencias que incrementan el número de pacientes con enfermedades de baja visión son las cataratas no operadas, la retinitis pigmentaria, la miopía magna, el glaucoma y la aniridia.

Los factores de riesgo comunes de la discapacidad visual en el mundo son[2]: la edad, el género y la condición económica:

En cuanto a la edad, la discapacidad visual es más habitual en personas mayores de 50 años, que representan un 65% del total de casos de discapacidad visual en el mundo. A partir de esta edad, la visión se degrada, produciéndose, por ejemplo, pérdidas de sensibilidad a la iluminación, pérdidas de la capacidad de enfoque, opacidad en el cristalino, etc. Por estas razones, la degradación natural del sistema visual aumenta la probabilidad de sufrir patologías relacionadas con la baja visión.

El género puede ser un factor de riesgo de sufrir discapacidad visual, pero se debe tener en cuenta que la esperanza de vida de las mujeres es mayor que en el caso de los hombres; y puede tratarse de un factor relacionado con el primero. Por ello, existe división de opiniones en la comunidad científica, existiendo expertos que sostienen que la genética de la mujer la hace más propensa a sufrir este tipo de enfermedades; y expertos que explican el mayor número de casos de discapacidad visual en mujeres, como consecuencia de la longevidad femenina.

En las sociedades en vías de desarrollo, el acceso a servicios médicos por parte de la población más pobre suele ser muy escaso o inexistente; por lo que la condición social y económica de las personas en cada país, es un factor de riesgo de padecer discapacidad visual. Las cifras lo avalan diciendo que el 86% de los casos de discapacidad en el mundo tienen lugar en países en vías de desarrollo.

Aproximadamente 979.200 personas son deficientes visuales en nuestro país (el 2,4% de la población), de las que 920.900 sufren enfermedades de baja visión [2]. El número de ciegos en España es cercano a los 58.300. Las comunidades más afectadas por discapacidad visual son Extremadura, Castilla la Mancha, Castill y León y Galicia. Si la medida es atendiendo a la edad, hasta los 65 años existen más casos de discapacidad visual en hombres; sin embargo, a partir de esa edad, la situación se invierte y los casos se hacen más habituales en las mujeres.

En definitiva, existen multitud de razones para ampliar el conocimiento en este campo, tanto a nivel mundial, como en nuestro país. Sería deseable que los gobiernos aumentaran sus esfuerzos en la lucha contra la discapacidad visual, primero, informando correctamente a los ciudadanos de la situación de estos colectivos en nuestro país, y en el mundo; segundo, creando planes para el estudio de las patologías de la baja visión, para el desarrollo de tratamientos y para la implantación de sistemas de ayuda a discapacitados visuales.

Índice

Índice de figuras

Resumen

El presente trabajo se enfoca en el estudio de las posibilidades del software de apoyo a la visión humana, orientado a pacientes con enfermedades de baja visión, es decir, enfermedades que disminuyen la agudeza visual afectando tanto a la precisión lograda en la retina, como a la amplitud del campo visual. En concreto, nos centramos en los efectos de la degeneración macular asociada a la edad (DMAE). En un primer momento, introducimos al lector en el problema de la discapacidad visual en el mundo, y más concretamente, en el subconjunto de los pacientes de DMAE. Atendiendo a la anatomía del ojo, presentamos las características de la dolencia, sus síntomas. Continuamos el capítulo indicando las tecnologías informáticas más utilizadas en el mundo de la visión artificial; para terminar con él, describiendo los algoritmos de realce de imágenes que, consideramos, pueden ayudar a los pacientes de DMAE. Llegados al capítulo 3, introducimos una descripción de nuestro modelo de retina; junto a una breve descripción anatómica del ojo humano. Explicamos cómo opera nuestro modelo de retina, estableciendo, donde proceda, los correspondientes paralelismos con la función del ojo humano; y describiendo cada una de sus cinco etapas. El capítulo 4 versa sobre el entorno de experimentacion donde se ha implementado, tanto del modelo de retina descrito en el capítulo anterior, como de los algoritmos de realce presentados en el capítulo 2; en cada uno de los dos lenguajes de programación en que se ha desarrollado la Retina, a saber, Octave y C. En el capítulo 5 describimos las metodologias usadas y como se ha paralelizado el código del. En concreto, hablamos de la paralelización realizada en OpenMP. Seguidamente presentamos los resultados de la implantación del software en varios dispositivos, observando y contrastando los tiempos de ejecución de los diferentes desarrollos (desarrollos seriales y desarrollos paralelos), en las diferentes máquinas, finalizando así el último capítulo, el sexto; y concluyendo el documento con las referencias bibliográficas que hemos consultado para este fin.

Palabras clave: DMAE, Retina, Octave, C, OpenMP

Abstract

This work focuses on the study of the possibilities of software to support human vision, aimed at patients with diseases of low vision, that is, diseases that decrease visual acuity affecting both the accuracy achieved in the retina, as the breadth of the visual field. Specifically, we focus on the effects of macular degeneration (AMD). At first, we introduce the reader to the problem of visual impairment in the world, and more particularly in the subset of patients with AMD. In response to the anatomy of the eye, we present the characteristics of the disease, its symptoms. Chapter indicating continued information technologies most used in the world of machine vision; to finish with him, describing the image enhancement algorithms that consider, they can help patients with AMD. We arrived to chapter 3, we introduce a description of our model retina; and a brief anatomical description of the human eye. We explain how to operate our model retina, establishing, where appropriate, the corresponding parallels to the function of the human eye; and describing each of its five stages. Chapter 4 deals with the experimentation environment where it is implemented, both retina model described in the previous chapter, as enhancement algorithms presented in Chapter 2; in each of the two programming languages that have been developed Retina, namely Octave and C. In section 5 we describe the methodologies used and as parallelized code. In particular, we discuss OpenMP parallelization performed in. Then we present the results of the implementation of the software on multiple devices, observing and contrasting the execution times of the different developments (developments serial and parallel developments), on different machines, ending the last chapter, the sixth; and finalizing the document with the references that we consulted for this purpose.

Keywords: DMAE, Retina, Octave, C, OpenMP

Capítulo 1. Introducción

1.1 Motivación

Los problemas visuales afectan a una importante parte de la población mundial. Según cifras de la OMS, 39 millones de personas padecen ceguera, y otros 246 millones presentan baja visión.

Existen multitud de enfermedades visuales. Las más comunes y su incidencia en la población mundial son:

- Errores de refracción (miopía, hipermetropía, astigmatismo...) que afecta al 43% de la población
- Cataratas no operadas que es sufrida por un 33%
- Glaucoma, la padece un 2% en todo el mundo

Todas estas patologías tienen cura o un tratamiento específico, al igual que el 80% de las enfermedades visuales catalogadas; sin embargo, no es el caso de la enfermedad que nos interesa en este proyecto: la degeneración macular.

Antes de hablar de degeneración macular, hay que hablar sobre la parte del ojo a la que afecta: la mácula[3]. La mácula es una mancha amarilla localizada en la retina, que está especializada en la visión de los detalles, es decir, en la visión de precisión. Nos sirve entre otras cosas para leer, o para reconocer un rostro humano. Se sitúa en la parte posterior de la retina, con una extensión aproximada de 5 mm de diámetro y está enfrente del disco óptico; luego procesa la información visual del centro de la imagen. Una característica importante de esta zona es que no distingue la intensidad de la luz, básica para la detección del movimiento. Para ello existen unas células llamadas bastones, que no se encuentran en la mácula. En lugar de bastones, existe una superpoblación de conos, que son las células encargadas de identificar el color, y transmitir al cerebro las variaciones de éste en la imagen.

La degeneración macular conlleva un deterioro progresivo de la mácula. La enfermedad suele estar asociada a la edad, puesto que la padece aproximadamente una de cada tres personas que superan los 75 años. Esto es porque esta membrana se va deteriorando con el paso de los años, haciendo que el paciente sufra una pérdida de visión que aumenta con el tiempo; sin embargo, también se puede dañar esta membrana de forma artificial, si se expone a una luz muy intensa. Un factor de riesgo de la enfermedad es el consumo de tabaco, ya que una persona que fume, al menos, una cajetilla diaria, aumenta sus posibilidades de padecer DMAE.

Existen dos variantes de la enfermedad. La primera es la degeneración macular seca, en la que el deterioro de la mácula está asociado a la pérdida progresiva de función de los conos. La segunda variante se llama degeneración macular húmeda, y se desarrolla con la formación de vasos sanguíneos anormales en la parte posterior del ojo. Cuando

estos vasos empiezan a presentar fuga líquida; ocasionan distorsión en la retina. Es el tipo más agresivo. Cabe destacar el papel de unas manchas que pueden aparecer en la mácula, las drusas. Las drusas son manchas amarillas en la mácula, que consisten en la formación de pequeños depósitos de material extracelular. Suelen formarse por encima de los 50 años y están relacionadas con la aparición de DMAE.

Cuando una persona padece esta enfermedad, lo que percibe es una pérdida de nitidez en la parte central de la imagen; pero no en la parte externa, por lo que se percibe perfectamente el movimiento. Es característico en pacientes con estados avanzados de la enfermedad, el hecho de mantener la visión periférica, pero ser incapaces de distinguir elementos detallados, como rasgos faciales o formas de letras.

1.2 Alcance

La degeneración macular no tiene tratamiento, solo se puede frenar su evolución. Nuestro propósito en este proyecto ha sido ayudar a las personas que padecen la enfermedad, mediante la creación de un sistema de apoyo a la visión humana, concebido pensando en las limitaciones que impone la DMAE.

Para ello hemos diseñado un sistema que trata una imagen de entrada, para simular en ella la visión normal del ojo humano y la degeneración macular. Además, aplicamos ciertos algoritmos para resaltar bordes de la imagen y simplificar su contenido, favoreciendo así el reconocimiento de las formas que contiene. Por último, el sistema procesa la imagen modificada como lo haría un ojo humano. Para conseguirlo nos basamos en el modelo artificial de retina propuesto por Al-Atabany[4], en el que se aproximan los procesos reales que suceden en el ojo con modelos matemáticos. Nuestra aplicación toma dichos modelos y los integra en un módulo software, desarrollado en C. Con nuestro modelo, pretendemos facilitar a las personas con degeneración macular la tarea de reconocer caras en la parte central de la imagen, o formas de objetos, asistiendo así a su visión. El objetivo final de este sistema es ser instalado en un dispositivo portable de forma que pueda mejorar la visión de un paciente en su día a día, capturando las imágenes que él vea y procesándolas en tiempo real.

Con todo esto pretendemos facilitar a las personas con degeneración macular la tarea de reconocer caras y formas en la parte central de la imagen, mejorando así su visión. El objetivo final de este sistema es ser instalado en un dispositivo portable de forma que pueda mejorar la visión de un paciente en su día a día, capturando las imágenes que él vea y procesándolas en tiempo real.

Capítulo 2. Visión artificial

En los mamíferos, la visión es uno de los sentidos más importantes, por ser de los que más información aporta al proceso de percepción de lo que nos rodea. La automatización del sentido de la vista es una tarea complicada, ya que implica reproducir sobre imágenes digitales los procesos que tienen lugar en los distintos elementos del sistema visual. El sistema visual está compuesto por el subsistema encargado de codificar la luz en impulsos eléctricos, el ojo; y el subsistema encargado de procesar los impulsos, la corteza visual del cerebro. La visión artificial o visión por computador, se centra principalmente en reproducir dichos procesamientos, y por ello, es una rama de la inteligencia artificial. Estudia la ardua tarea de dotar a un computador de la capacidad de ver imagenes y comprender su contenido. El procesamiento de imágenes que se realiza en la visión artificial persigue:

- Reconocimiento de formas incluidas caras humanas.
- Busqueda de patrones.
- Clasificacion de imagenes segun su contenido.
- Seguimiento de objetos.
- Mapeo de zonas para poder crear modelos tridimensionales.

Las tareas anteriores implican enfrentarse a problemas complejos como la detección de bordes, regiones o elementos importantes; reconocimiento de objetos y extracción de propiedades. En resumen, requiere conocer las estructuras que contiene la imagen para diferenciarlas y razonar sobre ellas. La resolución de estos problemas permite la aplicación de la visión artificial en:

- Computación: Los procesamientos con imágenes suelen ser altamente costosos en tiempo de ejecución, además de buenos ejemplos de procesos paralelizables. También, es posible que un sistema pueda necesitar visión artificial en tiempo real, y que los tiempos de procesamiento sean imperceptibles al ojo humano. Por esta razón, son buenos candidatos a ser acelerados mediante técnicas de procesamiento paralelo.
- Robótica: La navegación de un robot implica el reconocimiento de objetos y de distancias en el entorno de acción del robot. El objetivo del robot es construir un mapa 3D del entorno, recurriendo generalmente a técnicas de visión estereoscópica. Las mismas técnicas pueden aplicarse al guiado automático de máquinas en usos agrícolas, militares, de exploración, etc; así como a la detección del movimiento de vehículos.
- Biología: Es el área, junto con medicina, que más se beneficia de la visión artificial, porque a menudo se trabaja con imágenes microscópicas, o macroscópicas, de organismos biológicos. Las estructuras de las imágenes este tipo son observadas por biólogos expertos, que atendiendo características de forma, color, tamaño o excentricidad son capaces de identificar estados,

organismos, tejidos, malformaciones, etc. La visión artificial puede ser utilizada para contar poblaciones de microorganismos o células en una región; que pueden ser aisladas mediante técnicas de segmentación. También puede ser empleada para comprobar el estado de floración de determinadas plantaciones, o el estado de los tejidos en animales o vegetales.

- Medicina: En medicina, el uso del procesamiento de imágenes es útil para realizar diagnósticos de enfermedades sobre radiografías, resonancias magnéticas o tomografías. Se utiliza para monitorizar el bombeo de la sangre, atendiendo a los cambios de tamaño de los vasos sanguíneos con los movimientos de sístole y diástole; para la detección de cánceres de piel en base al color y forma de los melanomas, o en general, para la distinción entre tejidos sanos y tejidos enfermos, mediante el color.
- Reconocimiento aéreo: El procesamiento de las imágenes aéreas obtenidas por aviones o satélites, permite diferenciar entornos humanos de entornos naturales. Permite identificar edificios, poblaciones o infraestructuras, mediante técnicas de segmentación de regiones.
- Control y calidad: El reconocimiento de objetos puede ser muy útil en metalurgia para comprobar la validez de los productos manufacturados. En general, ayuda a controlar los acabados de las superficies, la geometría de los productos. También es importante a lo largo del proceso de fabricación de circuitos impresos, ya que permite verificar distribución de los componentes, integridad de las pistas y acabado de soldaduras.

2.1 Procesado de imágenes: Convolución y Filtrados

En ocasiones, las imágenes digitales presentan altos niveles de ruido, producidos por los dispositivos de captura, que se deben eliminar para que no entorpezcan el procesamiento. El ruido en imágenes digitales se manifiesta en píxeles que presentan niveles de brillo o color aleatoriamente distintos a los de sus vecinos; y se eliminan aplicando filtros a las imágenes. En consecuencia, se suele entender el filtrado de imágenes como un proceso previo a la segmentación.

En esencia, un filtrado es la transformación de una señal de entrada, para producir una señal de salida; y muy íntimamente ligado al concepto de señal, está el concepto de frecuencia. En lo que a frecuencia se refiere, solemos estar más familiarizados con el concepto de frecuencia temporal, pero si entendemos las imágenes como señales, debemos trabajar con otra frecuencia: la frecuencia espacial. De una señal temporal periódica, decimos que tiene alta frecuencia temporal, cuando altera periódicamente su valor de amplitud en un corto periodo de tiempo; pero de una imagen, diremos que tiene alta frecuencia espacial, cuando altere rápidamente el valor de los niveles de intensidad de píxel, o de los niveles de gris, en un intervalo espacial pequeño. En contraposición, las imágenes de baja frecuencia espacial, alteran su nivel de gris más lentamente, y los cambios se producen de forma algo más gradual.

Las imágenes se pueden filtrar en el dominio de la frecuencia, o en el dominio espacial:

- Si el filtrado se realiza en el dominio de la frecuencia el proceso consiste en una multiplicación de transformadas, por ejemplo, la multiplicación de la transformada de Fourier de la imagen, con la transformada del filtro. La transformada inversa del resultado, será la imagen filtrada.
- Si el filtrado se realiza en el dominio espacial, la operación necesaria es la convolución. Este es el tipo de filtrado que nos interesa en

2.1.1 Convolución

La convolución es una operación matemática que se define como Se define como la integral del producto de dos funciones después de desplazar una de ellas una distancia i, j (en ejes x e y respectivamente), como podemos ver en la siguiente fórmula:

$$[f * h][(x, y)] = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} f(m, h)h(x - m, y - n)dmdn \ (1)$$

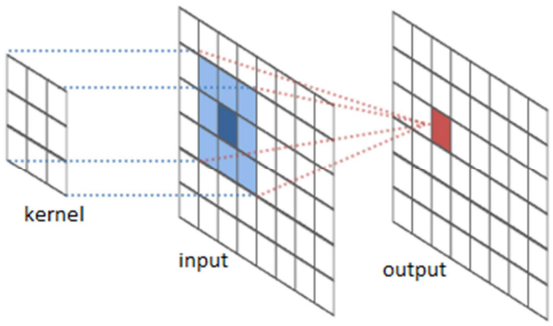
donde f * h es la convolución de dos imágenes f y h.

Esta operación tiene muchas aplicaciones en diversas ramas de ingeniería y física. En electrónica la salida de un sistema lineal es la convolución de la entrada con la respuesta del sistema a un impulso. En óptica, se puede representar una sombra como la convolución de la forma del punto de luz con la del objeto que proyecta dicha sombra. En nuestro caso nos interesa su uso en procesamiento digital de imágenes.

En este campo, se puede utilizar para crear efecto de desenfocado o detectar bordes en una imagen. Pero aquí no sirve la fórmula anterior, ya que nos encontramos en un dominio discreto, debido a que no hay valores entre el i-ésimo píxel y el (i+1)-ésimo. Más adelante se comentará con detenim*iento. La versión de la* (1) en el dominio discreto es la siguiente:

$$F_0(u, v) = \ \Im\{f * h(x, y)\} = \Im\{f(x, y)\}\Im\{h(x, y)\} = F(u, v)H(u, v) \ (2)$$

donde, F y H son dos matrices y n, m son las dimensiones horizontal y vertical de F, que actúa como filtro en el ejemplo. La siguiente figura muestra de manera visual esta operación:



lustración 1. Convolución

2.1.2 Filtrado Gaussiano

Se conoce como filtrado gaussiano al filtro que devuelve como salida una función gaussiana (o una aproximación de ella). Es un filtro de tipo pasa baja, es decir, da más peso a los valores cercanos a cero (o al centro del filtro, que no tiene por qué ser el cero) y desestima los valores más distantes.

Es frecuente en electrónica y proceso de señales el uso la versión de una variable de este filtro. Sin embargo en el campo del procesamiento de imágenes se utiliza la la función gaussiana de dos variables para crear el filtro:

$$G_\sigma(x,y) = \frac{1}{2\pi\sigma^2} e^{-(x^2+y^2)/2\sigma^2} \quad (3)$$

donde σ representa la desviación típica de la función. Cuanto más aumente este valor, mayor será la separación entre las dos vertientes de la función, y por tanto se obtendrá mayor efecto de filtrado. La gráfica del filtro es la siguiente:

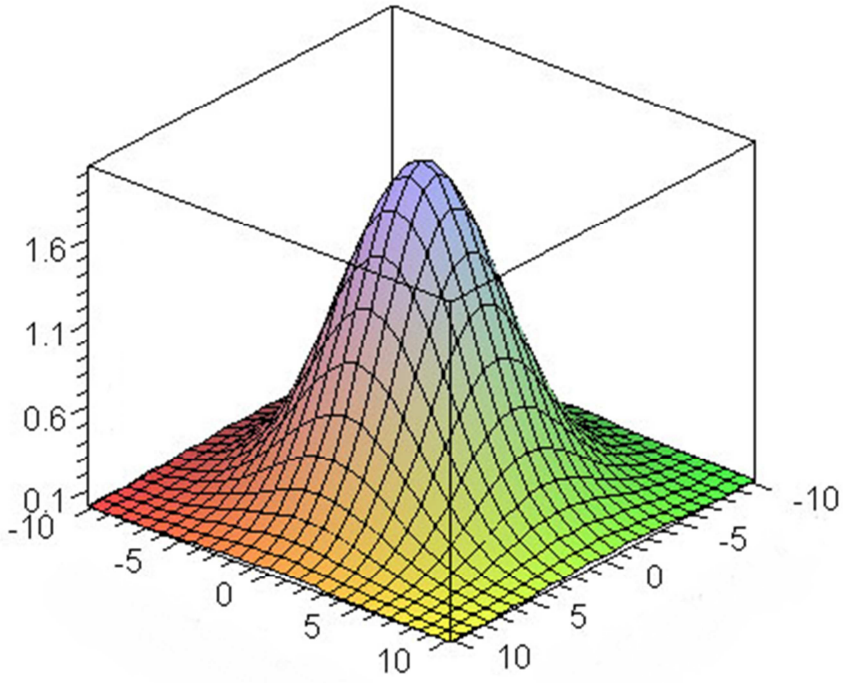


Ilustración 2. Gráfica Gaussiana

Se puede ver en la gráfica que esta función es contínua, es decir, está definida para todos los valores reales. Sin embargo en el caso que nos ocupa no trabajamos en un dominio continuo sino discreto, ya que no hay valores entre el i-ésimo píxel y el (i+1) ésimo. Por este motivo, nuestro sistema representa el kernel del filtro como una matriz de n x n posiciones (siendo n el diámetro del filtro), donde el valor de cada posición es el resultado de sustituir la x y la y de la función gaussiana por un valor entero que va variando entre -n/2 y n/2 (para centrar la función dentro de la matriz), quedando como resultado la siguiente estructura:

0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0000	0.0000	0.0000
0.0000	0.0000	0.0002	0.0011	0.0018	0.0011	0.0002	0.0000	0.0000
0.0000	0.0002	0.0029	0.0131	0.0215	0.0131	0.0029	0.0002	0.0000
0.0000	0.0011	0.0131	0.0586	0.0965	0.0586	0.0131	0.0011	0.0000
0.0001	0.0018	0.0215	0.0965	0.1592	0.0965	0.0215	0.0018	0.0001
0.0000	0.0011	0.0131	0.0586	0.0965	0.0586	0.0131	0.0011	0.0000
0.0000	0.0002	0.0029	0.0131	0.0215	0.0131	0.0029	0.0002	0.0000
0.0000	0.0000	0.0002	0.0011	0.0018	0.0011	0.0002	0.0000	0.0000
0.0000	0.0000	0.0000	0.0000	0.0001	0.0000	0.0000	0.0000	0.0000

Ilustración 3. Matriz Gaussiana

Para aplicar este filtro a la imagen de entrada se aplica una convolución entre ambas. De esta forma lo que se consigue es que el píxel que se está tratando en cada iteración mezcle su información con la de los (n x n) - 1 píxeles vecinos, obteniendo una sensación desenfocado. Este efecto es muy utilizado en procesamiento de imágenes porque permite eliminar ruido y texturas no deseadas, haciendo que se suavice la imagen. Cuanto mayor sea el valor de sigma (o de n), mayor es la sensación de desenfocado, haciendo que la imagen se vea más y más borrosa.

Además, para evitar que exista pérdida o sobreganancia de información, el filtro se debe normalizar a 1, es decir, la suma de todas las componentes debe ser 1. Así conseguimos que los valores máximo y mínimo de cada componente del píxel de salida (R, G y B en este caso), estén dentro del rango posible. En la siguiente figura podemos ver una comparativa de la imagen sin filtrar (a) y la imagen con un filtrado gaussiano, variando su desviación típica σ=4 (b), σ=10 (c), σ=15 (d).

Ilustración 4. Ejemplo Gauss

2.1.3 Filtrado Anisotrópico

Al procesar imágenes, es muy útil realizar filtrados como el gaussiano (que acabamos de ver) u otros tipos de filtros que veremos a continuación. Sin embargo, existe un problema, y es que la mayoría de éstos actúan sobre toda la imagen por igual.

El filtrado anisotrópico sirve para suavizar y desenfocar imágenes, pero respetando los bordes de éstas. De esta forma, se consigue dejar bien nítidos los bordes relevantes, desechando texturas que no sean importantes y ruido que pueda distorsionar la imagen.

Esta característica que lo distingue del resto de filtros de su tipo, lo hace perfecto para ser utilizado en este proyecto, ya que una de las tareas a realizar es un realzado de bordes.

La técnica utilizada consiste en ver en qué partes la imagen tiene cambios bruscos de claro a oscuro, y viceversa. Para ello se calcula el gradiente (variación) de la intensidad de la luz en la imagen. La siguiente fórmula define este proceso:

$$\frac{\partial I_t(x)}{\partial t} = div[c(x)\nabla I_t(x)] \ (4)$$

Donde It(x) es la intensidad de la imagen en el instante t, div representa el operador de divergencia y ∇ representa el gradiente de la imagen. Este proceso lo podemos reducir a un proceso iterativo en el que el numero de pasos sea suficiente como para eliminar el ruido sin perder informacion importante.

$$I^{n+1} = I^n + \Delta t[\nabla(C\nabla I_H) + \nabla(C\nabla I_V) + \nabla(C\nabla I_{D1}) + \nabla(C\nabla I_{D2})] \ (5)$$

Donde n representa el paso n-esimo de la iteracion y I_H,I_V,I_{D1},I_{D2} representan los gradientes de la imagen en las cuatro direcciones, siendo el coeficiente de difusion:

$$C = \frac{1}{1+\sqrt{\nabla I_H^2+\nabla I_V^2+\nabla I_{D1}^2+\nabla I_{D2}^2}} \ (6)$$

Una vez visto todo el proceso matemático que define el filtro anisotrópico explicaremos como se ha implementado dentro de nuestro sistema.

El filtro recorrerá cada pixel de la imagen n veces, siendo n el número de pasos que queremos que se itere.

En la primera iteración, se copiará la información del gradiente de cada píxel en una nueva matriz de tipo float, para ir acumulando información sin perder el valor total acumulado; ya que si usáramos el tipo monocromo, o RGB, limitaríamos el rango de valores a [0..255].

Los siguientes pasos recorren el resto de iteraciones realizando el mismo proceso anterior; pero no sobre la imagen original, sino sobre el valor que vamos acumulando en cada pixel.

Un paso añadido final sería normalizar este valor a un rango de 8 bits, que es como representamos la información de cada pixel.

Es destacable el hecho de que la imagen que se utiliza en este filtro es monocromática (un solo canal), ya que se aplica solamente sobre el canal de luminancia de la imagen. Aun así, el coste computacional del proceso de filtrado es elevado $\Theta(rows * cols * n)$.

2.1.4 Otros tipos de filtrado

Este tipo de filtrado sirve para suavizar imágenes, y es similar al gaussiano. En este, en lugar de asignar valores distintos a cada posición del kernel, se asigna el mismo valor a cada una. De esta manera, al mezclar la información de cada píxel con su vecindad, cada vecino modifica de igual manera el píxel en cuestión; a diferencia del filtrado gaussiano, que pondera la influencia de cada píxel vecino a la distancia al pixel central. Esto es así porque es un filtro pasa-baja.

Como consecuencia, este filtro devuelve una imagen de salida más borrosa que el filtrado gaussiano, ya que cuanto más alejados estén dos píxeles, más probable es que tengan valores dispares.

La siguiente figura muestra la diferencia entre una imagen sin filtrar (a), y la misma imagen al aplicar un filtrado promedio(b), y un filtrado gaussiano (c).

Ilustración 5. Comparación Filtros

2.2 Representación de imágenes

La representacion digital de imagenes es hoy en dia muy diversa, en este apartado presentaremos brevemente los formatos mas representativos.

2.2.1 Formato BMP

BMP es un formato de imagen digital, utilizado para almacenar gráficos rasterizados, que es independiente del dispositivo sobre el que se muestre. Muy conocido por su uso en Microsoft Windows. Las características que nos ofrece son:

- Almacenar matrices de píxeles de 2 dimensiones.
- Pueden ser monocromáticas como de varios canales de color.
- Permite diferentes profundidades de color.

- Elegir si queremos compresión de los datos.
- Almacenar también el canal alfa(luminosidad).
- Perfiles de color.

La estructura de un fichero BMP[11] está compuesta por una zona de cabecera, que contiene una serie de metadatos que nos permiten saber de qué forma están almacenados los datos en la siguiente zona, el cuerpo del fichero. Conocida la información de esta cabecera, deberemos proceder a leer los datos de tal manera que, pixel a pixel, la información sea correcta.

Bitmap File Header BITMAPFILEHEADER	
Signature	
File Size	
Reserved1	Reserved2
File Offset to PixelArray	
DIB Header BITMAPV5HEADER	
DIB Header Size	
Image Width (w)	
Image Height (h)	
Planes	Bits per Pixel
Compression	
Image Size	
X Pixels Per Meter	
Y Pixels Per Meter	
Colors in Color Table	
Important Color Count	
Red channel bitmask	
Green channel bitmask	
Blue channel bitmask	
Alpha channel bitmask	
Color Space Type	
Color Space Endpoints	
Gamma for Red channel	
Gamma for Green channel	
Gamma for Blue channel	
Intent	
ICC Profile Data	
ICC Profile Size	
Reserved	

La lectura de ficheros de este tipo es bastante sencilla, se lee la cabecera para obtener los datos necesarios y a continuacion se lee los datos que iran almacenados en la matriz de pixeles. Es señalable destacar que la imagen esta almacenada de manera que la primera fila que nos encontramos mientras leemos es la ultima fila de la imagen, es decir, la imagen se lee desde su ultima fila hasta la primera y de izquierda a derecha en cada una de ellas en lo que a columnas se refiere.

Image Data PixelArray [x,y]					
Pixel[0,h-1]	Pixel[1,h-1]	Pixel[2,h-1]	...	Pixel[w-1,h-1]	Padding
Pixel[0,h-2]	Pixel[1,h-2]	Pixel[2,h-2]	...	Pixel[w-1,h-2]	Padding
			⋮		
Pixel[0,9]	Pixel[1,9]	Pixel[2,9]	...	Pixel[w-1,9]	Padding
Pixel[0,8]	Pixel[1,8]	Pixel[2,8]	...	Pixel[w-1,8]	Padding
Pixel[0,7]	Pixel[1,7]	Pixel[2,7]	...	Pixel[w-1,7]	Padding
Pixel[0,6]	Pixel[1,6]	Pixel[2,6]	...	Pixel[w-1,6]	Padding
Pixel[0,5]	Pixel[1,5]	Pixel[2,5]	...	Pixel[w-1,5]	Padding
Pixel[0,4]	Pixel[1,4]	Pixel[2,4]	...	Pixel[w-1,4]	Padding
Pixel[0,3]	Pixel[1,3]	Pixel[2,3]	...	Pixel[w-1,3]	Padding
Pixel[0,2]	Pixel[1,2]	Pixel[2,2]	...	Pixel[w-1,2]	Padding
Pixel[0,1]	Pixel[1,1]	Pixel[2,1]	...	Pixel[w-1,1]	Padding
Pixel[0,0]	Pixel[1,0]	Pixel[2,0]	...	Pixel[w-1,0]	Padding

Ilustración 6. Estructura BMP

El caso de la escritura a fichero es un poco mas compleja ya que la cabecera contiene datos que nuestra estructura de datos no maneja. Debemos almacenar el tamaño del fichero en bytes, que tamaño en bytes ocupa la cabecera, profundidad de color,tabla de colores importantes(perfiles de color) y otros datos que son facilmente calculables y que no influyen para nada en el rendimiento de nuestra aplicacion.

2.2.2 Formato JPG

Es un estándar para la compresión de imágenes en color y escala de grises[5]; cuyas siglas son el acrónimo de Joint Photographic Experts Group. El estándar define tres modelos de codificación diferentes: un primer modelo para codificar con pérdidas, que está basado en la transformada discreta del coseno (TDC), un segundo modelo extendido para codificar con más compresión y precisión, y un sistema de codificación

independiente sin pérdidas para compresión reversible. Para que una imagen sea compatible con JPEG debe soportar el modelo de codificación base. Es un sistema de representación de imágenes muy habitual para guardar imágenes, siempre que no es necesario mantener un alto grado de información. No es muy usado para el tratamiento o procesamiento de imágenes, ya que el proceso de compresión introduce pérdidas de calidad. Surge con el desarrollo de internet ante la necesidad de mostrar imágenes en documentos html. Admite los modelos de color RGB, CMYK y escala de grises. La pérdida de información está relacionada con el nivel de compresión del archivo y, en la mayoría de los casos, una menor compresión produce un archivo casi idéntico al original.

2.2.3 Formato PNG

El formato Portable Network Graphics es un formato de codificación de imágenes que fue desarrollado pensando en su uso en internet[6]. Fue diseñado teniendo en cuenta la portabilidad de los ficheros. Es un formato de compresión sin pérdidas que utiliza el algoritmo de compresión *deflate*, el cual se desarrolló originalmente para el formato de compresión de archivos Zip. Admite una profundidad de color de 24 bits, lo que permite un rango de millones de colores diferentes.

Los ficheros en PNG están formadas, además de una serie de cabeceras con metadatos, por uno o varios canales, pudiendo formar:

- imágenes en escala de grises (1 canal),
- imágenes en escala de grises más canal alfa (2 canales),
- imagen con canales rojo, verde y azul (3 canales. Formato RGB),
- imágenes en 3 canales: RGB más canal alfa.

Gracias al canal alfa, que recibe el nombre alternativo de canal de transparencia, se permite la transparencia de fondo.

2.3 Algoritmos de realce.

Los algoritmos de realce son una parte fundamental de la visión por computador. Su principal objetivo es el de realzar o destacar las partes que consideramos más importantes dentro de una imagen.

En este proyecto concreto nos centraremos en algoritmos de realce de bordes, que a grandes rasgos deciden, en cada caso, si el píxel concreto forma parte de un borde dentro de la imagen. Entenderemos como borde de una imagen, no los márgenes del cuadro de la imagen sino, dentro de su propio contenido, la región que limita los objetos que aparecen. Pensemos, por ejemplo, en una imagen, con fondo blanco, que contiene una figura geométrica de color negro. Un algoritmo de realce de bordes aplicado a este ejemplo, nos devolvería el contorno de la figura, es decir, los puntos donde se pasa del color blanco al color negro. La figura que viene a continuación muestra un ejemplo de detección de bordes, marcando esas zonas donde se produce un cambio brusco de color.:

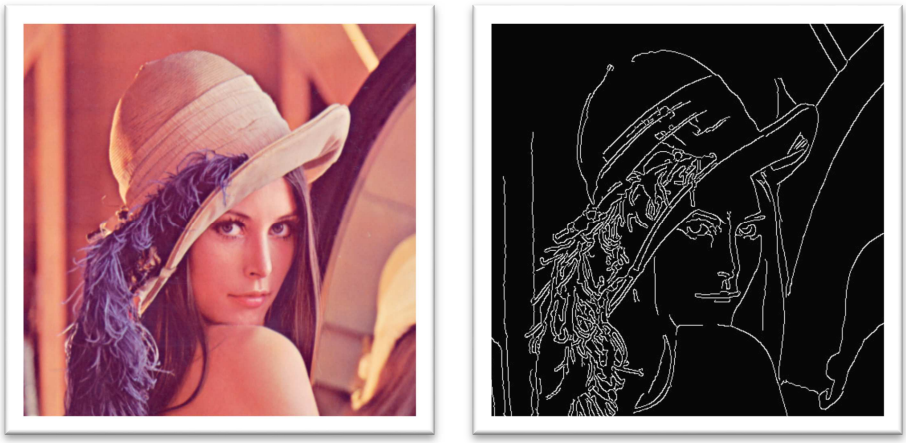


Ilustración 7.Realce

Las imágenes que vamos a tratar son fotografías estáticas tomadas en un momento determinado, que contendrán objetos estáticos en el momento del fotograma. Pero dado que el objetivo de este proyecto pasa por la implementación de este tipo de algoritmos en dispositivos hardware portátiles, que presumiblemente trabajan en movimiento, debemos tener en cuenta el contraste de objetos dentro de una escena en movimiento. Esta escena, por tanto, se puede ver afectada por ruido, entendiendo como tal la distorsión que se produce al observar un objeto mientras nos movemos, un aspecto muy a tener en cuenta. Como ejemplo podemos pensar en cómo percibimos un campo de maíz desde la ventanilla de un coche en movimiento.

2.3.1 Algoritmo de Canny

El algoritmo de Canny está catalogado dentro del campo de la visión artificial, y sirve para efectuar realce de bordes significativos en imágenes digitales. Fue desarrollado por John F. Canny en 1986.

Se utiliza frecuentemente en todo tipo de aplicaciones de procesamiento de imágenes, en las que se desee reducir la cantidad de información a procesar. Como ejemplos encontramos aplicaciones de reconocimiento de formas, búsqueda de patrones, seguimiento de objetos en movimiento o sistemas de reconocimiento OCR.

Los pasos de los que se compone el algorimo de Canny son:

1. Búsqueda de los gradientes en la imagen: A diferencia de otros filtros de primera derivada, como el filtro de Sobel, este algoritmo hace uso de cuatro gradientes (Horizontal, Vertical, y dos diagonales).
2. Supresión de no-máximos: En este paso tomaremos un conjunto discreto de direcciones y elegiremos una dirección posible para el ángulo del gradiente. Los valores posibles a tener en cuenta son: Este-oeste, norte-sur, noroeste-sureste, noreste-suroeste.
3. Mecanismo de doble umbral: En este paso, y teniendo calculados previamente los gradientes y los sectores a los que pertenecen, podemos elegir qué píxeles son candidatos a formar parte de los bordes.
4. Proceso de Histéresis: De todos los posibles candidatos extraídos en el anterior paso, hay que establecer cuáles de ellos son los verdaderos bordes, contrastando variaciones de color con los vecinos.

2.3.2 Algoritmo TRON

TRON es otro algoritmo de realce de bordes. Debe su nombre a las siglas de Tinted Reduced Outlined Nature. Este algoritmo fue diseñado para ofrecer ayuda a pacientes con pérdida de visión, para distinguir objetos en movimiento. Se basa en captar el contraste (borde) entre los diferentes objetos que forman la escena (en nuestro caso imágenes fijas), para mezclarlo con la imagen real. Sigue los siguientes pasos:

1. Simplificación de la escena usando un filtrado de tipo anisotrópico.
2. Extracción de los bordes que se consideren significativos.
3. Mezclado de los bordes extraídos con la imagen original para realzar el contenido de ésta.

El primer paso consiste en eliminar el ruido de la imagen con el filtro anisotrópico (ver punto 2.1.3).

El segundo paso consiste en discernir lo que es un borde de lo que no, para lo cual hemos usado el algoritmo de Canny.

El tercer y último paso recoge la imagen original y la máscara resultante de aplicar sobre la primera el algoritmo de Canny, y las mezcla. El resultado mantiene intactas aquellas zonas de la imagen donde no existe borde, para realzar estos últimos.

La siguiente figura muestra una imagen sin transformar (a) y después de haber aplicado el algoritmo TRON (b):

Ilustración 8. Ejemplo TRON

2.3.3 Algoritmo Image Cartoonization

El algoritmo image cartoonization o *cartoonización de imagen* en castellano, es una técnica que se utiliza para transformar imágenes, de forma que sea más sencillo reconocer las formas que contenga. Su nombre viene de la palabra inglesa *cartoon,* que significa dibujo animado.

El objetivo de este algoritmo es hacer que la imagen de salida parezca un dibujo, independientemente de su contenido: objetos, paisajes u otro tipo de escenarios reales. Si la imagen de entrada ya parece un dibujo, o lo es, el algoritmo no surte apenas efecto.

Este proceso se divide en cuatro partes:

1. Simplificación de la escena usando un filtrado de tipo anisotrópico.
2. Extracción de los bordes que se consideren significativos.
3. Cuantización de color.
4. Combinación de mapa de gradientes e imagen binarizada.

Las dos primeras etapas son análogas a las del algoritmo TRON. La primera se repite porque es necesario suavizar texturas y bordes no relevantes de la imagen, pero sin perder información sobre los bordes. La segunda es necesaria para hacer el realce propiamente dicho.

En la tercera etapa, el algoritmo se encarga de limitar el número de colores que hay en la imagen. Para ello nos hemos basado en el método que diseñó Winnemoller [7].

El proceso que define este paso se ejecuta de la siguiente manera: primero se traduce la imagen de entrada a un formato en el que uno de los canales represente a la intensidad lumínica. En este proyecto hemos elegido el formato YCbCr, cuyo canal Y es el deseado. Este canal por sí solo contiene una copia en escala de grises (blanco y

negro) de la imagen original. A continuación, se calcula el ancho de cada *color binario* dividiendo el valor máximo del canal Y entre el número de binarios que queremos obtener (en este caso lo hemos fijado en 8). Por último, se aproxima cada valor de la matriz Y al binario más próximo, para lo cual se divide el valor del píxel entre el ancho del binario, se toma el valor absoluto. Este valor absoluto se multiplica otra vez por el ancho del binario.

Las siguientes ecuaciones definen el funcionamiento de esta etapa:

$$\Delta_q = \frac{Y_{max}}{N}; \quad (7) \qquad X = \frac{1}{\Delta_q} Y(i,j) \; ; \quad (8) \qquad Y_{cartoon} = \Delta_q abs(X) \quad (9)$$

donde Y_{max} representa el valor máximo del canal Y, $Y_{(i,j)}$ representa al valor del píxel actual del canal Y y *abs(X)* representa el valor absoluto de *X*.

El cuarto y último paso consiste en mezclar las derivadas espaciales (mapa de gradientes) obtenidas con el algoritmo de canny con la imagen cartoonizada, de tal forma que si en la máscara hay bordes se dejan los bordes y si no se deja el valor de la cartoonizada.

La siguiente figura ilustra este algoritmo con un ejemplo de imagen sin tratar (a) y tras la ejecución de *image cartoonization*:

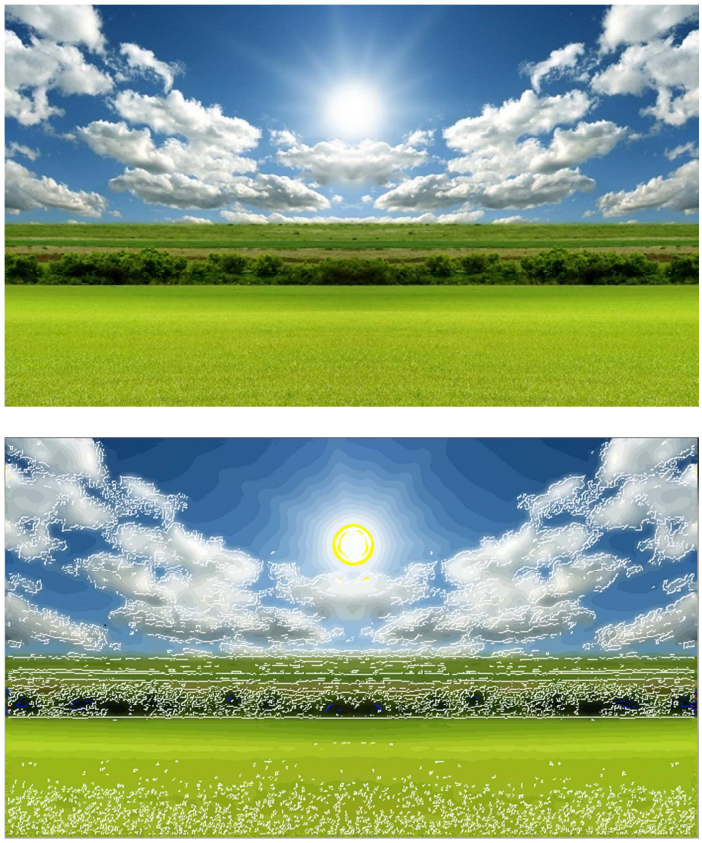


Ilustración 9. Ejemplo Cartoon

2.3.4 Otros métodos de realce

Operadores Prewitt, Sobel y Frei-Chen: Estos operadores sirven para calcular el gradiente en una matriz, es decir, indicar la dirección hacia la cual hay mayor diferencia entre el valor de cada elemento de la matriz de entrada y el elemento en cada dirección. Se puede utilizar en procesamiento de imágenes para calcular la variación de la intensidad para cada píxel de la imagen.

$$\begin{matrix} -1 & -b & -1 \\ 0 & 0 & 0 \\ 1 & b & 1 \end{matrix} \qquad \begin{matrix} -1 & 0 & 1 \\ -b & 0 & b \\ -1 & 0 & 1 \end{matrix} \qquad (10)$$

donde a = 1 en el operador Prewitt, a = 2 en el operador Sobel y $a = \sqrt{2}$ en el operador de Frei-Chen.

Para calcular el gradiente de una imagen, primero se halla la convolución de cada matriz operadora con la matriz intensidad lumínica Y de la imagen de entrada, obteniendo dos resultados G_x y G_y. Estas variables contienen las derivadas direccionales de la imagen $\frac{\delta Y}{\delta x}$y $\frac{\delta Y}{\delta y}$ respectivamente. Cada una representa la variación de la intensidad, es decir, la cantidad de luz en los ejes x e y resp. Así que valores altos en G_x indican bordes en posición horizontal (de arriba a abajo) y valores bajos indican secciones de la imagen uniformes. En G_y sucede de igual manera pero en horizontal (de izquierda a derecha).

Al ser ambas derivadas perpendiculares, podemos obtener el valor final del módulo del gradiente aplicando el teorema de Pitágoras y el ángulo θ resultante de la siguiente forma

$$G = \sqrt{Gx + Gy} \quad (11) \quad \theta = arctan(\frac{Gx}{Gy}) \; (12)$$

En la práctica, el operador de Prewitt detecta mejor los bordes verticales, pero el de Sobel lo hace mejor con los diagonales. Además es menos sensible al ruido, bordes o texturas no relevantes. El operador de Frei-Chen intenta lograr un compromiso entre ambos.

En la figura podemos ver una imagen en escala de grises (blanco y negro), en contraposición con la máscara que genera el operador sobel tras haber unido los mapas de gradientes en los ejes *x* e *y*:

Ilustración 10. Ejemplo Sobel

2.3.5 Operadores Laplacianos

Otra manera de detectar bordes es utilizar operadores basados en la función laplaciana. esta función se define en el dominio continuo como:

$$\nabla f(x,y) = \frac{\delta^2 f(x,y)}{\delta x^2} + \frac{\delta^2 f(x,y)}{\delta y^2} \quad (13)$$

En este caso a diferencia del gradiente, la laplaciana no retorna una información vectorial sino un escalar. Al aplicar la función sobre una región de intensidad homogénea devolverá un valor nulo, y por el contrario si se aplica sobre un borde

aparecerán en sus alrededores valores positivos y negativos. Este comportamiento indicará la regla de actuación para la detección de los bordes.

Por la propiedades de la segunda derivada, este operador es más exacto en la precisión del borde que el uso del gradiente (operadores Sobel, Prewitt...). Para pasar al dominio discreto tendremos que sumar las derivadas parciales de segundo orden, quedando de la siguiente manera para un kernel de 3 x 3:

$$\frac{\delta f(x,y)}{\delta x^2} \simeq \frac{f(x+1,y) - 2f(x,y) + f(x,y-1)}{\Delta x^2}$$

$$\frac{\delta f(x,y)}{\delta y^2} \simeq \frac{f(x,y-1) - 2f(x,y) + f(x,y+1)}{\Delta y^2}$$

Al pasar al dominio discreto, los bordes ya no tendrán valor cero sino valores muy próximos a cero. Éstos, con este operador, estarán formados por el ancho de un píxel a diferencia del método del gradiente. Además, el operador laplaciano es isotrópico a diferencia de las distintas máscaras que se implementan con el operador gradiente.

En la siguiente figura se puede apreciar la diferencia entre una imagen sin procesar (a), un realzado completo (no solo el mapa de gradientes) utilizando el operador Prewitt (b) y un realzado con una técnica laplaciana. Nótese que de nitidez de (b) es significativamente peor que la de (c)

Ilustración 11. Ejemplo Sobel

Capítulo 3. Modelos de retina

3.1 Funcionamiento de la retina humana

Antes de hablar sobre los diferentes modelos de retina que hemos implementado y cómo se han hecho, debemos introducirnos un poco en cómo funciona una retina humana. Este asunto queda fuera del campo de la computación, por lo que haremos un breve esbozo sobre su funcionamiento. Después se aproximará en base a un modelo matemático, totalmente portable a un lenguaje de programación.

La retina es similar a una tela sobre la que luz, al incidir, desencadena una serie de procesos químicos y eléctricos; que son traducidos en impulsos nerviosos y llevados al cerebro a través del nervio óptico [8].

Está formada por varias capas de neuronas interconectadas mediante sinapsis y las unicas celulas sensibles a la luz son los conos y bastones, una retina contiene aproximadamente entre 6.5 millones de conos y 120 millones de bastones. Los bastones trabajan en baja visibilidad y proporcionan visión en blanco y negro, los conos sin embargo operan en condiciones de alta luminosidad y visión en color.

Las distintas capas que forman la retina, de la más superficial a la más interna, son:

- Capa limitante interna: Separa la retina del humor vítreo.
- Capa de fibras del nervio óptico: Formada por los axones que forman el nervio óptico.
- Capa de células ganglionares: Formada por los núcleos de las células ganglionares.
- Capa plexiforme interna: Se establece la conexión sináptica entre las células bipolares, amacrinas y ganglionares.
- Capa granular interna: Contiene los núcleos de células bipolares, horizontales y amacrinas.
- Capa plexiforme externa: Produce la sinapsis entre las células fotorreceptoras y las bipolares.
- Capa Granular externa: Compuesta de los núcleos de las células fotorreceptoras.
- Capa limitante externa: Contiene uniones de tipo zónula adherente entre las fotorreceptores y células de Müller.
- Células fotorreceptoras: Formada por los segmentos más externos de los conos y bastones.
- Epitelio pigmentario: Formada por células que contienen melanina y que le confieren una pigmentación característica.

Partiendo del funcionamiento biológico de la retina y usando un modelo matemático[4], aproximamos los procesos químico/eléctricos mediante una serie de pasos y cálculos.

3.2 Modelo de retina

Existen varios modelos de retina anteriores al que nosotros hemos utilizado. Algunos ejemplos son los trabajos de Hubel y Wiesel [9], Chang PR y Yeh BF [10] entre otros. Estos diseños se centran en el aspecto fisiológico de la retina o están enfocados al diseño de prótesis.

La diferencia con el modelo que hemos utilizado es que tratamos de representar una retina humana con una patología, en lugar de funcionar correctamente. Esto nos permite analizar el problema de la degeneración macular desde el punto de vista del paciente, lo cual es una gran ventaja a la hora de contrastar los resultados obtenidos, para las distintas transformaciones que hagamos a la imagen de entrada.

Nuestro modelo de referencia está dividido en capas, al igual que la retina real. Cada capa lleva a cabo el trabajo de un tipo de células oculares, y toman como entrada la salida de la etapa anterior. La siguiente figura representa esquemáticamente el diseño del modelo:

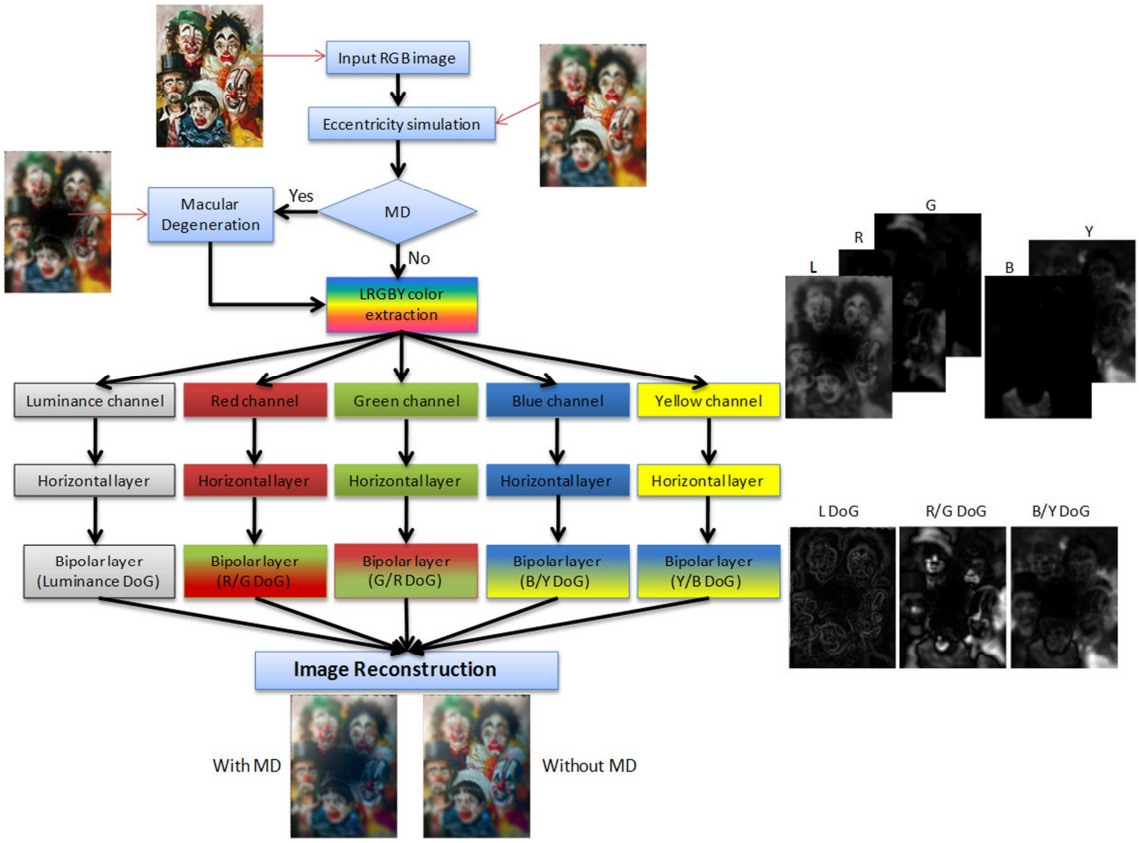


Ilustración 12. Diagrama Retina

3.2.1 Capa Foveal: Simulación Excentricidad

Con el objeto de reducir la cantidad de información que llega a través del nervio óptico al córtex visual, el ojo humano proporciona una imagen de alta resolución centrada en la fóvea y que reduce su resolución a medida que nos alejamos de ese centro. Para poder percibir el entorno de forma correcta el ojo realiza múltiples barridos a través del espacio visual existente, manteniendo una alta calidad en el centro de la imagen y un amplio rango de visión periférica.

Nuestro modelo matemático simula este comportamiento dividiendo la imagen en dos partes claramente diferenciadas: centro foveal y periferia.

- Centro foveal: Contiene una copia exacta datos de la imagen de forma que no se pierde calidad en ese punto. Esta parte ocupa aproximadamente un 2.5% de la imagen.
- Periferia: Contiene una copia de la imagen que va perdiendo calidad a medida que nos alejamos del centro foveal.

Esta pérdida de calidad la simularemos aplicando la convolución de la imagen de entrada y el filtro gaussiano, definiendo círculos concéntricos alrededor de la fóvea; para los que iremos aumentando la desviación típica del filtro, según aumenta la distancia radial al centro foveal. Al aumentar este parámetro, estamos haciendo que la pérdida de calidad sea mayor, para obtener de esta forma el efecto de pérdida progresiva de nitidez en la imagen.

$$I_{foveal}(x,y) = G_\sigma(x,y) * I(x,y) \quad (15)$$

donde: $G_\sigma(x,y)$es un filtro gaussiano para el pixel(x,y), σ es su desviación típica, I(x,y) es el píxel en la posición x, y de la imagen de entrada y $I_{foveal}(x,y)$ es su análogo en la salida de esta capa.

La siguiente figura muestra visualmente el efecto simulado por esta etapa:

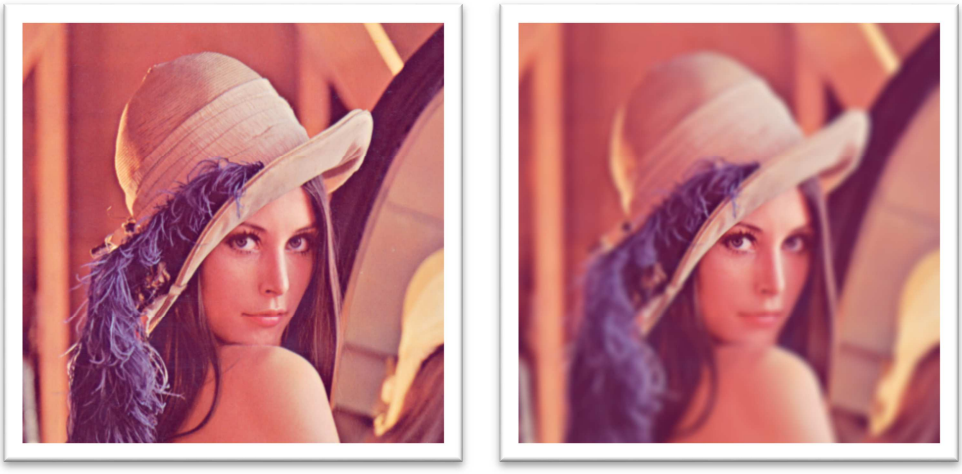


Ilustración 13.Simulación Foveal

3.2.2 Separación de color

El siguiente paso de la retina consiste en separar los distintos canales de color para trabajar con ellos de forma individual al mismo tiempo. Esto se hace ya que la información de color recibida por el córtex visual es codificada en dos canales opuestos: verde-rojo y azul-amarillo. Para ello convertiremos nuestra imagen RGB en una imagen LGRBY.

Canal L representa la luminancia absoluta de la imagen con valores comprendidos entre blanco y negro. Siendo el blanco una total luminosidad y el negro la falta absoluta de luz.

Los otros dos canales serán extraídos de:

- Canal R contiene la información del color rojo.
- Canal G contiene la información del color verde.
- Canal B contiene la información del color azul.
- Canal Y contiene la información del color amarillo que será calculada como: Ychannel=Bchannel-x

Donde: x es el valor máximo que permite cada canal.

3.2.3 Capa Horizontal

Las células que componen la capa horizontal sirven como retroalimentación sobre las células del cono, aumentando la luminosidad. Pero ya que la mayoría de sistemas de representación de imagen no poseen un rango de luminosidad muy alto debido a las limitaciones de representación [11], este modelo considera que la variación de luminosidad es suficientemente pequeña como para despreciarla y no incorporar esta ganancia.

Existen 3 tipos de células horizontales HI que representan el canal acromático (incrementos en luminosidad), HII y HIII que representan el resto de canales cromáticos. Como las células HI representan las variaciones en la intensidad de la luz, y éstas no van a ser tenidas en cuenta, podemos calcular la salida de esta capa como una difusión de la salida de los conos mediante un filtro gaussiano uniforme, es decir, cuya desviación típica sea la misma en todos los puntos de la imagen:

$$Horz_{L,R,G,B,Y}(x,y) = G(x,y) * Cono(x,y) \quad (16)$$

Donde L,R,G,B,Y representan cada uno de los canales en los que se descompuso la imagen en el apartado anterior, *G(x,y)* representa al filtro gaussiano en el píxel *(x,y),* y $Cono_{L,R,G,B,Y}(x,y)$ a las salidas de la separación de canales. Nótese que no se indica ningún parámetro de desviación típica sigma para G ya que en esta capa va a ser constante en toda la imagen.

3.2.4 Capa Bipolar

La entrada de la capa bilateral está compuesta por las salidas de los conos (capa de separación de canales de color) que se han modificado en la capa horizontal.

Hay dos tipos de células bipolares. Las células ON despolarizan la señal (aumentan la sensibilidad a la luz) y las células OFF la hiperpolarizan. Las sinapsis de estas células alrededor de las células ganglionares generan el fenómeno centro-envolvente (centre-surround). Este centro tiene una relación de tamaño 1:10 para los mamíferos.

Este proceso puede ser descrito de forma matemática como la resta de dos filtros gaussianos que simulan la acción de las células del ganglio retinal (centro) y de las células bipolares (periferia):

$$DoG_{Bipolar}(x,y) = \frac{1}{2\pi{\sigma_s}^2} e^{-(x^2+y^2)/2{\sigma_s}^2} - \frac{1}{2\pi{\sigma_c}^2} e^{-(x^2+y^2)/2{\sigma_c}^2} \quad (17)$$

donde $DoG_{Bipolar}(x,y)$ representa la salida de la etapa, σ_s y σ_c representan las desviaciones típicas de los filtros periferia y centro respectivamente. La proporción entre el sigma en el centro y el de la periferia debe ser de 2 a 1 aproximadamente.

Esta capa contrapone las salidas de los conos mezclando los canales mediante este procesamiento centro-envolvente. En este modelo calcularemos 5 señales de salida: el canal ON-OFF acromático y otros 4 contraponiendo los distintos canales de color, y lo haremos de la siguiente forma:

$$DoG_{G/R} = Horz_G - Cone_R$$

$$DoG_{R/G} = Horz_R - Cone_G$$

$$DoG_{B/Y} = Horz_B - Cone_Y \quad (18)$$

$$DoG_{Y/B} = Horz_Y - Cone_B$$

$$DoG_L = Horz_L - Cone_L$$

La penúltima señal no se genera en una retina real pero se mantiene por la simetría de la que dota a todo el proceso y de la cual hablaremos más adelante.

3.2.5 Reconstrucción

La reconstrucción de la imagen es la fase de la retina en la que se fusionan las capas procesadas individualmente en las etapas anteriores. Es un proceso complicado por el cual se aproxima el valor de la imagen mediante un método numérico. La etapa comienza considerando las capas de salida de la etapa anterior, la bipolar. Dichas capas, pueden ser consideradas como derivadas espaciales de los canales de la imagen

Luminancia, R/G y B/Y (nos referiremos a ellas como G). El problema al que nos enfrentamos consiste en encontrar una función I cuyas derivadas, sean muy similares a G; sin embargo, un campo de gradientes modificados, puede no ser integrable. En términos matemáticos, debemos encontrar una función I, buscando en el espacio bidimensional de todas las posibles funciones, cuyas derivadas se parezcan a G. Haciendo uso del operador nabla, esto es equivalente a minimizar la siguiente ecuación:

$$f = \iint \left|\left| \nabla I - G \right|\right| dxdy \quad (19)$$

Sin embargo, de acuerdo al principio de mínima acción, la ecuación que satisface la integral anterior debe satisfacer la ecuación de Euler-Lagrange:

$$\frac{\partial F}{\partial I} - \frac{d}{dx}\frac{\partial F}{\partial I_x} - \frac{d}{dy} - \frac{\partial F}{\partial I_y} = 0 \quad (20)$$

Por lo que podemos reducir el problema a resolver la siguiente ecuación en derivadas parciales.

$$\nabla^2 I = div(G) \quad (21)$$

Resolver esta ecuación mediante el método numérico de las diferencias finitas, puede conducir a resolver un gran sistema de ecuaciones lineales. Existen varios métodos alternativos para la solución numérica de la ecuación, pero por razones de eficiencia computacional, se resuelve siguiendo el método del resolutor rápido de Poisson (*fast Poisson solver*), guiándonos con el trabajo de [12]. El coste del algoritmo descrito en es de O(n log n).

3.2.6 Simulación macular

En esta etapa del procesamiento se modela la pérdida de función visual causada por una degeneración macular.

Para la simulación de esta dolencia, utilizamos un filtro gaussiano de un tamaño equivalente al tamaño de la parte de mácula degenerada. Dicho filtro se construye en virtud de dos parámetros de entrada. El primer parámetro es el área degenerada, que representa el área de la superficie macular dañada, y equivalentemente, el tamaño del filtro gaussiano. El segundo parámetro es el grado de degeneración, que representa la opacidad de la mancha que percibe el paciente, y que sobre los cálculos se hace corresponder con la altura de la función gaussiana. Ambos parámetros vienen dados en tantos por ciento.
Una vez obtenida la función gaussiana para el valor de sigma correspondiente, que debe ser un valor para el cual la gaussiana se anule en los límites del área degenerada, se debe normalizar a 1, invertir, y elevar hasta el valor que indique el grado de degeneración. Con el filtro gaussiano ya calculado, se crea una máscara del mismo

tamaño de la imagen, que será una matriz de 1s, con el objetivo de embeber el filtro en la posición correcta, es decir, haciendo coincidir el centro de la gaussiana con el centro foveal. Esta máscara final será aplicada a cada canal de la imagen, atenuando en las zonas degeneradas la intensidad y el matiz de cada pixel.

Ilustración 14. Simulación Macular

Capítulo 4. Entornos de Experimentación

4.1. Modelo de retina en Octave

Como ya vimos en el capítulo 2, los sistemas digitales de representación de imágenes utilizan matrices de números enteros o reales para simular cada unidad mínima de color en ellas. Debido a la naturaleza de este proyecto y a la cantidad de operaciones que se iban a realizar con estas matices, en una primera fase del proyecto realizó un primera aproximación al desarrollo del modelo matemático de retina utilizando el lenguaje de GNU Octave.

Octave es un programa que forma parte del proyecto GNU, y sirve para realizar todo tipo de cálculos. Permite ejecutar órdenes mediante una consola interactiva y está orientado al análisis numérico. Comenzó como un proyecto para el diseño de reactores químicos, en 1988, y más tarde, en 1994, salió su primera versión. Es el equivalente software libre a MATLAB, siendo este último de carácter privativo, y tiene una sintaxis casi idéntica a la de éste programa. Está escrito en C++ y soporta la mayoría de funciones de la biblioteca estándar de C, lo cual será muy adecuado pensando en desarrollos futuros..

La razón que nos llevó a utilizar Octave, fue disponer de un primer desarrollo de la retina que sirviera no solo de toma de contacto, sino también porque necesitábamos crear un modelo sencillo lo más rápido posible y así comprobar los cálculos que se realizan en cada etapa de la retina. Este primer modelo fue muy útil, aunque muy poco eficiente en cuanto a tiempo de ejecución se refiere, hecho que esperábamos.

Entre las características más destacables de Octave encontramos que permite leer imágenes desde fichero con una simple instrucción, realizando un volcado de los datos de ésta sobre una matriz de $filas\ x\ columnas$, en formato de representación RGB de 8 bits por canal. En consecuencia, cada pixel de la imagen ocupará, por tanto, 3 Bytes de información.

Todos los cálculos que se realizan sobre dicha matriz, incluidas las convoluciones, se realizan sin la necesidad de introducir demasiadas líneas de código; ya que la herramienta manipula las matrices automáticamente. En cualquier caso, todos estos cálculos, aunque correctos, no eran eficientes en tiempo de ejecución, y si nuestro propósito es implementar una retina para ejecutarse en tiempo real, es necesario buscar fórmulas alternativas de programación, que realicen los cálculos de manera eficiente y rápida. La decisión tomada para lograr una mayor eficiencia que la encontrada en Octave, fue desarrollar el código en C, ya que se trata de un lenguaje mucho más eficiente que octave.

4.2. Modelo de retina en C

La búsqueda de una alternativa que nos ofrezca un mayor rendimiento, nos lleva a buscar un lenguaje de programación que sea sencillo y además que sea portable a prácticamente cualquier dispositivo. Esto nos hizo decantarnos por C, que permite programar muy cerca del procesador y, además, facilita la compilación con niveles de optimización.

El lenguaje C es un lenguaje de programación creado por Dennis Ritchie en 1972. Encaja dentro del paradigma imperativo y está orientado a la implementación de Sistemas Operativos ya que trabaja a bajo nivel y generar código eficiente.

4.2.1 Estructura de la aplicacion y flujo de funcionamiento

La aplicación está pensada para ser ejecutada desde consola, no se ha creado ninguna interfaz gráfica ya que lo importante es poder volcarlo a un dispositivo portátil y poder evaluar su rendimiento.

El funcionamiento de la aplicación consiste en la carga de una imagen, un procesado de esa imagen de acuerdo a unos flags de entrada y si se escoge la opción de guardado, salvar los datos que se han generado a partir de la imagen original y las opciones escogidas.Las opciones disponibles son:

- Ejecucion de la retina con todos sus pasos.
- Ejecucion del algoritmo TRON.
- Ejecucion del algoritmo CARTOONIZATION
- Guardar las etapas seleccionadas.

Existe un flag para cada etapa de la retina.

Para saber mas sobre estas opciones consultar la ejecucion del programa con la opcion -h.

IMPORTANTE: Una vez se ha cargado la imagen de entrada se toma una medida de tiempo antes de empezar el procesado, terminado todas las opciones solicitadas se calcula el tiempo transcurrido desde la primera toma y se muestra por pantalla para evaluar el rendimiento.

4.2.2 Estructura que almacena los datos

Para poder contener la informacion de cualquier imagen necesitamos de una estructura que funcione para cualquier tamaño y numero de canales. Por ello que diseñamos un tipo de datos que cumpliera con dichas especificaciones y nos ofreciese un tipo de acceso rapido y sencillo al contenido de cada pixel:

```
struct image_matrix{
     int rows;
     int cols;
     bool tipo;
     RGB* dataRGB;
     monocromo* dataM;
}
```

Con esto permitimos almacenar el tamaño de imagen en filas(rows) x columnas(cols), el tipo de imagen que contiene(RGB o monocromo) y un puntero a cada uno de los posibles arrays de datos. Estos últimos al ser punteros no tienen un tamaño establecido en un principio nunca van a estar ambos poblados de datos al mismo tiempo ya que los controlamos con el tipo de datos que contienen.

Tanto RGB como monocromo son tipos definidos que contienen datos de tamaño 1 Byte, en el caso de monocromo tendremos un 1 Byte/pixel y en RGB 3 Bytes/pixel.

4.2.3 Como se lee y carga la información

Por muy definida que esté la estructura del programa y como guardaremos los datos mientras están siendo procesados, necesitamos poder cargar imágenes para poder procesarlas.

En el mundo de la fotografía digital existen múltiples formatos de imagen que nos permiten tener la misma imagen almacenada con diferentes tipos de compresión, calidad y profundidad de color. Ante tal abrumador panorama de formatos y que para poder cargar por nuestros propios medios cada uno de ellos tardaríamos demasiado, ya que tipo de fichero es distinto, recurrimos a una API para el procesamiento de imágenes digitales llamada openCV.

4.2.4 OpenCV

OpenCV es una biblioteca de libre distribución para procesamiento de imágenes y visión artificial [13].

Fue desarrollada por Intel, en un primer momento, está escrita en C y disponible para múltiples sistemas operativos (Microsoft Windows, GNU Linux y Mac OS X). Durante su desarrollo se hizo hincapié en la eficiencia para aplicaciones en tiempo real, incluyendo optimizaciones para arquitecturas multicore.

Teniendo en cuenta la estructura de datos escogida, estudiamos las distintas formas que tiene openCV para leer imágenes de forma matricial, y acabamos escogiendo el formato CvMat; porque almacena la imagen en un formato muy parecido al nuestro. Así, facilitamos la tarea de migrar los datos a nuestro propio formato de explotación.

El formato CvMat recoge los datos de la imagen haciendo uso de diferentes punteros de acceso. Tiene varios punteros, en función de la profundidad de color escogida; y almacena los píxeles en un formato continuo BGR, en lugar de RGB.

La ventaja de openCV sobre otras librerías de procesamiento de imágenes, es que incorpora multitud de funciones predefinidas para procesos de visión por computador. Se usa en múltiples aplicaciones para cámaras de seguridad, inspección de productos en factorías y, sobre todo, en el campo de la robótica.

Desde el punto de vista de nuestra aplicación, queremos explotar la paralelización a un nivel inferior y no podemos utilizar la mayoría de funciones que incorpora la librería, lo que anula la ventaja antes mencionada. Además, la complicación que requiere su instalación para poder compilar código escrito con ella, también era un factor a tener en cuenta. En un computador de propósito general, la compilación necesaria previa a la instalación se demoraba entre 30 y 40 minutos; en otros dispositivos portátiles, como la Raspberry Pi, lo hacía más de 9 horas.

Ante este problema, y dado que los dispositivos portátiles en los que íbamos a trabajar no eran de nuestra propiedad, se descartó el uso de openCV, y finalmente solo lo usábamos/usamos para cargar y guardar imágenes.

4.2.5 Utilización del formato BMP

Una vez descartado OpenCV, necesitábamos un sistema de lectura que fuese sencillo y rápido de implementar, la mayoria de formatos que hemos comentado previamente disponen de mecansimos de comprensión los cuales hacen que sean mas complejas de leer. Existen multiples librerias para la lecturas de estos formatos(libjpeg, libpng, etc...) pero ante la busqueda de algo portable al 100% y quitarlos el yugo de arrastrar dependencias de librerias externas nos decantamos por el uso de BMP.

La lectura y almacenado de las imagenes se realiza de forma sencilla, bastan un par de metodos para leer y almacenar una imagen, no profundizaremos mas en este aspecto ya que este formato fue previamente expuesto.

4.2.6 Contenido de los ficheros

Este proyecto cuenta con varios ficheros fuente entre los que distribuimos las funciones segun el tipo de trabajo que vayan a realizar. Ademas de esto contamos con un fichero de cabecera que contiene la definicion de la estructura de datos asi como una serie de parametros que son fijos durante la ejecucion.

Los ficheros fuente relacionados con la retina no hacen sino transformar el modelo matematico que se comento previamente a funciones en codigo C que tratan esos

datos sobre la estructura de almacenamiento que hemos definido para nuestras imagenes.

Existen ademas otros ficheros fuente que permiten lanzar la ejecucion de diferentes maneras, escogiendo el tipo de proceso que se requiere y si debemos salvar los datos calculados.

4.3. Optimizaciones de código

Hasta ahora todo lo que hemos visto es sobre el funcionamiento del sistema y qué técnicas hemos utilizado. La mayor parte del código utilizado dentro de este proyecto es bastante complejo en tiempo de ejecución, $\Theta(n^2)$ o superiores en la mayoría de casos. El problema que esto conlleva es que el principal propósito de este sistema es que pueda ser llevado a algún sistema portable y de bajo consumo. Lo ideal sería que los algoritmos de realce trabajasen a tiempo real y para ello hay que hacer que gran parte del código se ejecute de forma óptima, consiguiendo aumentar la productividad y el aprovechamiento de los recursos disponibles.

La ley de Moore expresa que aproximadamente cada dos años se doble el número de transistores que contiene un circuito integrado. Los procesadores actuales siguen esta tendencia y hoy dia en el mercado podemos encontrar que prácticamente cualquier PC, smartphone o dispositivo portátil lleva procesadores multi-core. Estos procesadores son capaces no solo de ejecutar instrucciones a frecuencias muy altas del orden de GHz, sino de ejecutar tareas en paralelo. La mayor parte de los programas existentes no explotan esta característica y es por tanto una pérdida del rendimiento al que podemos someter a estos dispositivos.

En este sistema en particular buscaremos agregar paralelismo a nuestro código como forma de optimización, consiguiendo de esta manera acelerar la ejecución del procesado de las imágenes y aprovechando de forma correcta los recursos que nos brindan los actuales dispositivos.

4.3.1 Cuestiones a tener en cuenta

El poseer un sistema capaz de ejecutar procesos en paralelo no quiere decir que todo programa pueda ser ejecutado de forma paralela ya que los dispositivos lo permiten. Cada programa o algoritmo es distinto y existen una serie de limitaciones por las que no podemos paralelizar todo el código.

Las leyes de Gustafson y Amdahl establecen que debido a una serie de dependencias y otros problemas, no todo el código será paralelizable y por tanto por muchos procesadores que añadamos para trabajar en paralelo no conseguiremos un mayor rendimiento.

Estas dependencias normalmente se deben a la necesidad de tener un dato previamente calculado para el calcular el siguiente. Podemos agregar mecanismos de sincronización y exclusión mutua en el acceso a los datos haciendo que aunque se ejecuten de forma separada mantengan la coherencia necesaria.

4.3.2 Tipos de paralelización

Existen diferentes tipos de paralelización según en qué nivel de la jerarquía del computador nos estemos centrando:

Paralelismo a nivel de bit:

Este tipo de paralelismo existe desde el principio y consiste en acelerar el procesamiento duplicando el tamaño de la palabra 4, 8, 16, 32 y 64 bits, esta última como tendencia actual.

Paralelismo a nivel de instrucción:

Los procesadores multiciclo ya hacen uso de este tipo de paralelismo que consiste básicamente en dividir el procesador en etapas independientes y mediante reordenaciones de código poder ejecutar instrucciones de manera que funcione como una cadena de montaje. Cada etapa del pipeline realiza una tarea determinada dentro del proceso de dicha instrucción. Una vez el pipeline se ha llenado conseguimos ejecutar una instrucción/ciclo.

Los procesadores superescalares añaden otro factor y es poder ejecutar la misma etapa simultáneamente para diferentes instrucciones, permitiendo lanzar varias de ellas al mismo tiempo. Algoritmos como el de Tomasulo permiten este tipo de ejecución fuera de orden.

Paralelismo a nivel de datos:

Este tipo de paralelismo se asocia con programas en los que se va a ejecutar el mismo conjunto de instrucciones para distintos datos, como sucede en cualquier ciclo. Los bucles paralelizables son los for, mientras que los bucles while al no tener una condición de cierre no es posible paralelizarlos.

Paralelismo a nivel de tareas:

Consiste en asignar diferentes tareas independientes a los diferentes sistemas de cómputo que tengamos disponibles.

4.3.3 Sistemas de memoria

Para que un sistema permita que se ejecuten instrucciones de forma paralela no basta con doblar los sistemas de ejecución es también necesario añadir un sistema de memoria que sea capaz de tolerar estos accesos simultáneos, distinguiremos dos tipos:

- Memoria compartida: Cualquier porción de memoria puede ser accedida en todo momento por los procesos o threads que de ella disponen, se basa en mecanismo de sincronización de los datos para que en todo momento

cualquier proceso que necesite de ese dato lo tenga actualizado. Si dos procesos paralelos necesitan acceder a la variable que es compartida y necesitan modificarla sólo uno de ellos podrá acceder, el otro deberá esperar.

- Memoria distribuida: Cada procesador tiene su propia porción privada de memoria, se comunican entre ellos mediante el paso de mensajes, normalmente se utilizan en procesadores superescalares.

4.3.4 Tipos de computadores paralelos

Multinúcleo: Disponen de varias unidades de ejecución en un mismo procesador.

Multiprocesador simétrico: Un sistema SMP es aquel en el cual varios procesadores idénticos acceden al mismo sistema de memoria mediante un bus, también se les conoce como sistemas de tipo UMA(Uniform Memory Access).

Computación cluster: Diferentes máquinas trabajan en estrecha colaboración, son independientes unas de otras y el balance de carga de trabajo de unas a otras es complejo. Se suelen conectar mediante redes de área local ethernet.

Computación distribuida: Hace uso de máquinas totalmente independientes que se comunican entre si a través de internet.

4.3.5 Threads

Un thread o hilo de ejecución es la unidad de ejecución más pequeña que puede ser planificada por un Sistema Operativo.

Dentro de un mismo proceso podemos tener varios threads trabajando de forma concurrente. Comparten las tablas de ficheros y las regiones de memoria, por tanto están dotados de mecanismos de sincronización para permitir una ejecución con datos correctos y evitar problemas. La parte privada de los threads se compone del contador de programa, la pila y los registros de la CPU.

Las ventajas que ofrecen los thread frente a un proceso normal es que su creación, destrucción y cambio entre ellos es mucho más rápida que con procesos normales, los procesos normales implican salvar el estado del BCP(Bloque Control de Proceso). También ofrece una ventaja la comunicación entre ellos, dos procesos que colaboren implican en su comunicación cambios de contexto y llamadas al sistema(kernel) frente a los mecanismos de comunicación de threads que son más rápidos y no implican llamadas al sistema.

4.3.6 Lenguajes paralelos

Los lenguajes paralelos han sido creados para programar en sistemas cuyas unidades de procesamiento permiten ejecutar código de forma concurrente. Se pueden clasificar en base a qué tipo de memoria van dirigidos. En este apartado hablaremos sobre algunos de los más conocidos y que características contienen.

Chapel

Lenguaje desarrollado por Cray Inc. participando en Darpa HPCS(High Productivity Computing Systems) para aumentar el rendimiento de supercomputadores en el año 2010.

Soporta multithreading, paralelismo a nivel de datos, de tareas y anidamiento de paralelismo.

POSIX Threads

Forma parte del estándar POSIX y se encarga del control de hilos(threads). Esta librería contiene múltiples funciones para la creación, manejo y destrucción de hilos, también provee de sistemas de sincronización,mutex y variables de condición.

CUDA

CUDA es el acrónimo de Compute Unified Device Architecture, una estrategia de programación paralela para procesadores gráficos nVidia [14]; pero se utiliza indistintamente para designar tanto a la estrategia, como al conjunto de herramientas creadas para programar en ella. En su uso habitual, hace referencia al entorno de desarrollo usado para paralelizar aplicaciones que aprovechen la arquitectura de las GPU de nVidia. Admite varios lenguajes de programación, pero el más habitual es C.

La estrategia de CUDA permite controlar y dirigir la ejecución de cada uno de los procesadores contenidos en una GPU. También presenta la posibilidad de introducir la estrategia SPMD en nuestros programas convencionales C. Los programas escritos en C, que se ejecutan en CPU tradicionales, en ocasiones tienen que ejecutar cálculos sobre conjuntos extensos de datos, y por ello, pueden admitir un alto grado de paralelización, es decir, tienen una alta granularidad. CUDA permite que los programas en C tradicionales, eventualmente, puedan acceder a las GPU para ejecutar estas secciones de código altamente paralelizable, haciendo uso de las facilidades de CUDA.

En el lenguaje CUDA, al procesador convencional que ejecutará la parte serie, se le llama *host*, y al procesador gráfico, *device.* Las secciones de programa altamente paralelizables se caracterizan por aplicar un determinado tratamiento a multitud de datos, por lo que tiene sentido dedicar un hilo diferente de ejecución a cada instancia de la tarea. En Cuda, a este tipo de tarea se le llama kernel, y será el programa que ejecutará cada procesador de la GPU. Además del kernel, CUDA utiliza los conceptos de threadidx y blockidx, para permitir al programador dirigir la ejecución de los kernels a conjuntos de procesadores del *device*. También utiliza los conceptos de memoria compartida y barreras de sincronización, ya que los distintos cores del *device* comparten memoria y en ocasiones requieren cierta coordinación.

CUDA está disponible para plataformas Microsoft Windows (XP/VISTA/7), para GNU Linux (32/64) y para Mac OS.

OpenCL

Es un framework desarrollado por el grupo Khronos para aplicaciones concurrentes. Genera paralelismo a nivel de datos y de tarea. Soporta la ejecución en diferentes CPU incluidas x86,ARM y PowerPC así como GPU de los principales fabricantes AMD y nVidia. Las arquitecturas soportadas incluyen procesadores multi-core y vectoriales. También soporta paralelismo de grano fino sobre dispositivos FPGA.

La API de OpenCL es una mezcla entre C y C++, teniendo soporte para otros lenguajes tales como Java, Python y .NET. El código que se ejecuta en los dispositivos(devices) normalmente no es el mismo que el que se ejecuta sobre el host, escrito en C.

Podemos dividir OpenCL en cuatro partes, llamadas modelos, que se especifican de la siguiente forma:

- Plataforma: Específica sobre qué procesador se va a ejecutar el proceso host y sobre cuales devices se van a ejecutar los kernels. Se define un modelo hardware en OpenCL C.
- Ejecución: Define cómo se configuran tanto el host como los kernels y provee de los mecanismos necesarios para la comunicación entre el host y los devices así como para mantener la concurrencia entre los diferentes kernels.
- Memoria: Define de forma abstracta el modelo y jerarquía de memoria que van a utilizar los devices, siempre basado en el modelo de memoria que posea la arquitectura sobre la que se va a ejecutar.
- Programación: Define cómo será la concurrencia según la arquitectura de memoria física.

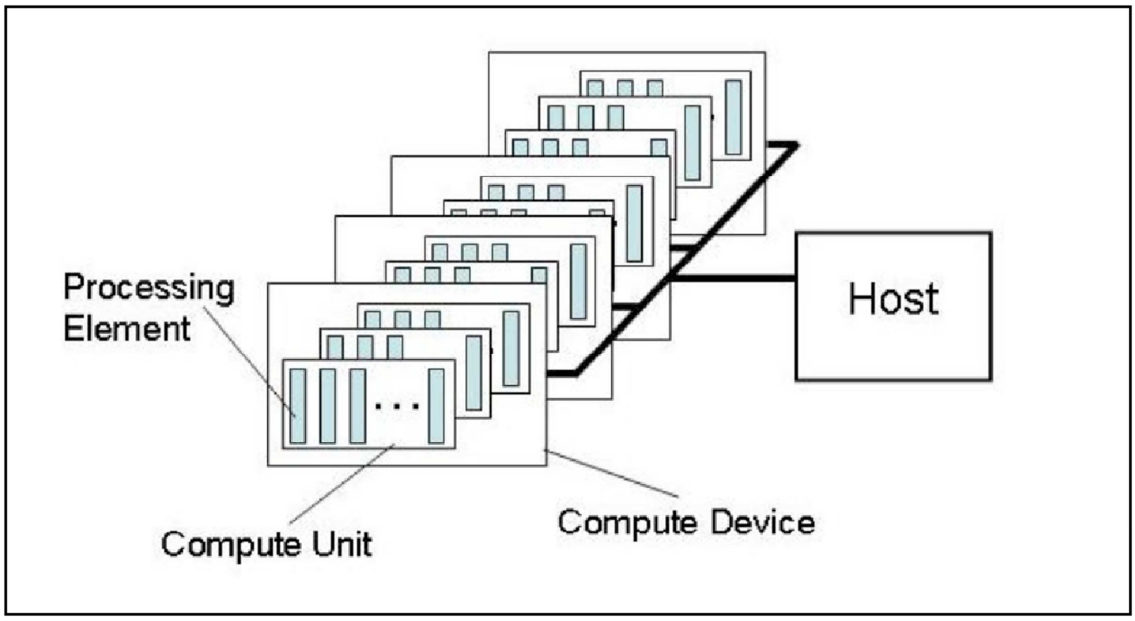


Ilustración 15. Modelo OpenCL

El funcionamiento típico de una aplicación con esta plataforma se compone de host implementado sobre una CPU, que dirige el tráfico de las diferentes tareas sobre una o varias GPU que llamaremos devices. Estos devices poseen unas capacidades de ejecución concurrente muy superior con lo que la ejecución de la tarea se acelera.

OpenMP

OpenMP es una API para la programación multiproceso en sistemas de memoria compartida, tanto UMA como NUMA [15]. Añade concurrencia a programas escritos en C, C++ y Fortran y se encuentra disponible en diferentes arquitecturas incluyendo UNIX y Microsoft Windows.

Su principal ventaja es que permite, de forma muy rápida y sencilla, paralelizar código escrito de forma seria y usarlo en todo tipo de equipos:

- Propósito general.
- Supercomputadores.
- Dispositivos de bajo consumo.

Su funcionamiento se basa en el modelo fork-join, llegado un punto en el cual podemos paralelizar código, se produce un fork de la tarea dividiéndose en varios hilos de ejecución que funcionaran de forma concurrente.

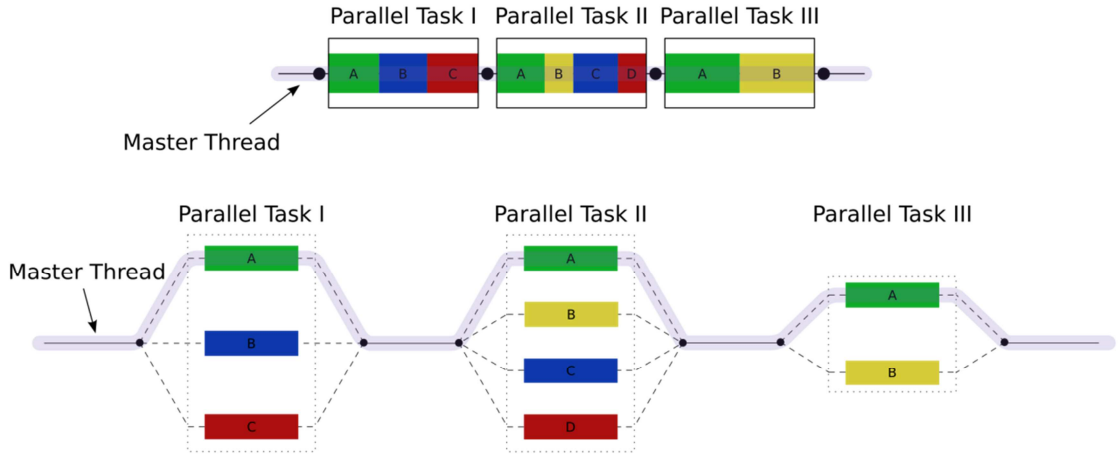


Ilustración 16. Modelo OpenMP

Entre sus características destacadas se incluye:

Entre sus características destacadas se incluye:

Paralelismo anidado: Permite el anidamiento de regiones paralelas pudiendo subdividir tareas que funcionan ya de forma paralela en otras subtareas más pequeñas. Esta característica no siempre está disponible y además puede influir de forma negativa en el rendimiento final.

Threads Dinámicos: Podemos alterar el número de threads en los que se va a subdividir una tarea de forma dinámica y en tiempo de ejecución.

Entrada/Salida: No se maneja de forma automática, es el propio programador el que debe establecer una serie de mecanismos de sincronización para que todo funcione de forma correcta.

La sencillez de la programación que nos brinda OpenMP se basa en que partiendo de un código que sabemos funciona de forma correcta ejecutándose en serie, sin necesidad de modificar nuestro código fuente y agregando unas directivas de compilación, estableceremos como y que queremos paralelizar ya tendremos nuestro código corriendo de forma concurrente.

Estas directivas nos permiten elegir:

- Cómo paralelizar
- Qué variables van a ser parte privada de cada hilo y cuales compartidas.
- Como se van a repartir los datos entre los distintos threads. Este reparto puede ser estatico o dinamico.
- Establecer si queremos que cierta parte del código se ejecute por un hilo en concreto.
- Mecanismos de sincronización tales como barreras, secciones críticas, etc…
- Definir variables de reducción: Si todos los cálculos que se producen en los diferentes hilos están abocados a un fin común, por ejemplo una suma, podemos definir una variable en la que se irá acumulando a medida que cada hilo termina su ejecución.

Capítulo 5. Metodologia

En este capítulo hablaremos de qué forma se han utilizado las técnicas de paralelización descritas en temas anteriores, qué problemas encontramos en su uso, y en qué dispositivos hardware se han implementado.

Este proyecto trabaja con imágenes, realiza una serie de cálculos sobre los datos contenidos en éstas, y devuelve una respuesta en forma de imagen resultante. Partiendo de esta premisa, el tipo de paralelismo escogido fue paralelismo a nivel de datos.

El impulso principal que nos llevó a paralelizar el código, fue el ejecutar los algoritmos de realce de una forma más rápida y óptima, lo más cercana posible al tiempo real, para poder permitir su implantación en dispositivos portátiles.

5.1 Paralelización con OpenMP

Como se ha mencionado antes, esta librería nos permite ejecutar código concurrentemente, de una forma sencilla mediante directivas. La mayor parte de el código de la aplicación, se basa en recorrer matrices de pixeles y realizar operaciones con ellas. Los bucles que utilizamos en el programa en C son del tipo for, recorren las matrices por filas y columnas, y por tanto tienen un tamaño preestablecido. Esto es ideal para un modelo pararlelo, ya que podemos hacer un reparto de las iteraciones entre los distintos threads creados gracias a OpenMP. Así, la paralelización de los bucles se realizó, siempre que fue posible, del más externo al más interno, repartiendo las iteraciones entre el número de hilos creados.

Una de las cuestiones importantes que debíamos aclarar era cuantos hilos se debían crear de forma simultánea. Este número de hilos se fijó en un primer momento, y permanece inamovible a lo largo de toda la ejecución, eliminando la posibilidad de threads dinámicos. Además, este número no superará nunca el número de threads que permita la máquina de forma concurrente; ya que, aunque un mayor número de threads implica un mayor reparto de las iteraciones, también conlleva, en estos casos, que muchos de los hilos permanezcan a la espera de que otros terminen.

El reparto de iteraciones se hizo de forma dinámica, con un tamaño mínimo, o chunk, de 50 iteraciones. Este valor no fue escogido al azar, porque las imágenes que vamos a tratar tienen un tamaño variable, de al menos 200x200 pixeles. La ventaja del reparto dinámico es que si trabajamos, por ejemplo, con una imagen de 512x512 y tenemos la posibilidad de crear 4 threads, obtendremos el reparto siguiente:

- Thread 0: 0..49, 250..299, 500..512
- Thread 1: 50..99, 300..349
- Thread 2: 100..149, 350..399

- Thread 3: 150..199, 400..449
- Thread 4: 200..249, 450..499

Obteniendo una *carga de trabajo equitativa* entre todos los threads.

5.1.1 Pasos realizados en la paralelización:

1 Localizar los bucles paralelizables: Este paso, que en un primer momento puede parecer sencillo, obligó a comprobar, para cada uno de los bucles candidatos, si existía algún tipo de dependencia en alguna de sus variables.

2 Agregar directivas: Estas directivas establecen como deben de coordinarse los threads creados, y qué carga de trabajo recibe cada uno de ellos.

Al principio de este tema mencionamos que las imágenes que se procesan y las imágenes resultantes son distintas. La estructura con la que trabajamos para almacenar los datos se recorre

fila a fila, y dentro de cada una de estas iteraciones, columna a columna. El acceso a cada una de las posiciones se realiza de forma sencilla con un desplazamiento: c=i*cols+j, para obtener la posición efectiva del pixel. Este acceso aparentemente sencillo, podría producir problemas de acceso simultáneo por parte de varios threads si no se manejaba de forma adecuada. Es lo que se conoce en computación como una sección crítica.

Otro problema que resolvimos fueron las sucesivas llamadas en las iteraciones a funciones externas. Cada vez que realizamos una de estas llamadas obligamos a salvar el contexto del procesador y el estado de la pila, además de cambiar e inicializar variables locales. Esto, por simple que parezca, y que lleva un tiempo despreciable si se ejecuta un número escaso de veces, se convierte en un importante factor de sobrecarga cuando tratamos imágenes; ya que estamos multiplicando el pequeño retraso por mil, haciendo que el rendimiento se degrade.

5.1.2 Ejemplos

A continuación comentaremos algunos ejemplos concretos relacionados con nuestro código.

- **Generación Kernel Gauss:**

```
}

void getKernelGauss(int n, int m, double sigma,double kernel[n][m]){
        int mitad_n = n/2;
        int mitad_m = m/2;
        int i,j;
        double sum = 0.0;
        double fact1 = 1/(2*M_PI*pow(sigma,2));
        #pragma omp parallel for default(shared) private(j) reduction(+:sum)
        for(i = -mitad_n; i <= mitad_n; i++)
                for(j = -mitad_m; j <= mitad_m; j++){
                        double potencia = (pow(i,2) + pow(j,2)) / (2*pow(sigma,2));
                        double fact2 = exp(-potencia);
                        kernel[mitad_n+i][mitad_m+j] = fact1*fact2;
                        sum += kernel[mitad_n+i][mitad_m+j];
                }
        #pragma omp parallel for default(shared) private(j)
        for(i = 0; i < n;i++)//normalizacion
                for(j = 0; j < m;j++)
                        kernel[i][j] /= sum;
}
```

Ilustración 17. Código Kernel Gauss

En la imagen anterior se puede observar una función que dado un sigma nos calcula una máscara de convolución para un filtro gaussiano.

En el primero de los bucles se puede observar como se ha extraído el factor1 fuera del bucle ya que permanece constante e implica cálculos matemáticos complejos en número de ciclos/instrucción. La directiva de compilación indica que por defecto las variables serán compartidas ya que las posiciones en las que se escribe son siempre distintas, solamente la variable j del bucle debe ser privada, ya que las iteraciones i son las que se reparten entre los threads. Es también reseñable el uso de una variable de reducción que se usará en el siguiente bucle para normalizar los datos.

- **Aplicación filtro Gauss:**

```
int gaussianFilter(const struct image_matrix* imageIn,struct image_matrix* imageOut,int n,int m,double sigma){
        int mitad_n = n/2;
        int mitad_m = m/2;
        int cols = imageIn->cols;
        int rows = imageIn->rows;
        double kernel[n][m];
        getKernelGauss(n,m,sigma,kernel);
        //*imageOut = malloc(sizeof(struct matrix_rgb));
        imageOut->rows = rows;
        imageOut->cols = cols;
        imageOut->tipo = imageIn->tipo;
        if(imageOut->tipo == false){
                imageOut->dataRGB = malloc(rows*cols*sizeof(rgb));En la EE
                int i,j;
                #pragma omp parallel for default(shared) private(j)
                for(i = 0; i < rows;i++)
                        for(j = 0; j < cols;j++){
                                if( i < mitad_n || i > rows-mitad_n-1 || j < mitad_m || j > cols-mitad_m-1){//relleno de bordes
                                        int k = i*cols + j;
                                        imageOut->dataRGB[k][0] = imageIn->dataRGB[k][0];
                                        imageOut->dataRGB[k][1] = imageIn->dataRGB[k][1];
                                        imageOut->dataRGB[k][2] = imageIn->dataRGB[k][2];
                                }
                                else
                                        convolutionPixel(imageIn,n,m,kernel,imageOut,i,j);
                        }
        }else{
                imageOut->dataM = malloc(rows*cols*sizeof(monocromo));
```

Ilustración 18. Código Filtro Gauss

En este trozo de código se puede observar cómo se aplica la máscara de convolución que hemos calculado antes, por cada pixel se calcula su nuevo valor en función de la máscara invocando a la función convolutionPixel(...).

El principal problema que encontramos en este punto fue que las llamadas a funciones externas implican el salvado del estado de la pila y ralentizan el rendimiento. Además se estudió el paralelizar el bucle que posee esa función pero lo único que conseguimos fue ralentizar aún más el problema. Esto se debe a que una vez hecho el reparto de hilos tal como se explicó antes y no permitiendo modificar su número en diferentes puntos, lo único que conseguimos es penalizar: Pensemos un thread posee la iteración 10 y otro que posee la iteración 9 ha llegado antes a dicha función y por tanto indica a OpenMP que sus hilos tienen que trabajar para realizar las iteraciones de esa convolución. Esto implica que el hilo que ejecuta la iteración 10 deja su trabajo actual para pasar a ejecutar otra sentencia y esto al final perjudica en gran parte el rendimiento. Es por tanto que se decidió prescindir de esta función que aunque funcionaba de forma correcta, perjudicaba el rendimiento del sistema.

- **Algoritmo de Canny:**

```
G =(float*) malloc(rows*cols*sizeof(float));
phi =(float*) malloc(rows*cols*sizeof(float));
pedge =(int*) malloc(rows*cols*sizeof(int));
int i,j,c;
print_verbose("***Calculo de Gradientes***\n");
#pragma omp parallel for default(shared) private(j,c) schedule(dynamic,50)
for(i = 0; i < rows;i++){//CALCULO DE LOS GRADIENTES
        print_verboseI(omp_get_thread_num());
        for(j = 0; j < cols;j++){
                if( i < 1 || i > rows-2 || j < 1 || j > cols-2){//relleno de bordes
                        c = i*cols + j;
                        imageOut->dataM[c]= 0;
                }
                else{
                        c = i*cols + j;
                        imageOut->dataM[c]= 0;
                        monocromo z1,z2,z3,z4,z5,z6,z7,z8,z9;
                        z1 = imageIn->dataM[(i-1)*cols+(j-1)]; z2 = imageIn->dataM[(i-1)*cols+j];
                        z3 = imageIn->dataM[(i-1)*cols+(j+1)]; z4 = imageIn->dataM[i*cols+(j-1)];
                        z5 = imageIn->dataM[i*cols+j];
                        z6 = imageIn->dataM[i*cols+(j+1)]; z7 = imageIn->dataM[(i+1)*cols+(j-1)];
                        z8 = imageIn->dataM[(i+1)*cols+j]; z9 = imageIn->dataM[(i+1)*cols+(j+1)];
                        Gx[c] = (z1 + 2*z4 + z7) - (z3 + 2*z6 + z9);//convoluciones
                        Gy[c] = (z7 + 2*z8 + z9) - (z1 + 2*z2 + z3);
                        G[c] = sqrtf((Gx[c]*Gx[c])+(Gy[c]*Gy[c]));
                        phi[c] = atan2f(fabs(Gy[c]),fabs(Gx[c]));
                        //DIRECCION DEL GRADIENTE
                        if(fabs(phi[c])<=PI/8 )
                                phi[c] = 0;
                        else if (fabs(phi[c])<= 3*(PI/8))
                                phi[c] = 45;
                        else if (fabs(phi[c]) <= 5*(PI/8))
                                phi[c] = 90;
                        else if (fabs(phi[c]) <= 7*(PI/8))
                                phi[c] = 135;
                        else
                                phi[c] = 0;
                }
        }
```

Ilustración 19. Código Canny

Este algoritmo de resaltado de bordes implica cálculos complejos en cada iteración: raices cuadradas, arco-tangentes. Todos ellos sobre números en coma flotante aumentado el número de ciclos/instrucción.

En el código expuesto se puede observar como se ha quitado la función que invoca a la convolución, solucionando el anterior problema.

- **Simulación Foveal:**

```
void fovealSimulation(struct image_matrix *image, struct image_matrix *imageOut, double hFov, double wFov, int nCircles){

        imageCopy(image, imageOut);

        if(nCircles > 0){

                int imgHeight = imageOut->rows;
                int imgWidth = imageOut->cols;
                int shortSide = MIN(imgHeight, imgWidth);

                if((hFov < 0) || (hFov > imgHeight))    hFov = imgHeight / 2;
                if((wFov < 0) || (wFov > imgWidth))             wFov = imgWidth / 2;

                double fovRadius = (1/42) * shortSide / 2;
                double maxRadius = MAX(MAX(hFov, (imgHeight - hFov)), MAX(wFov, (imgWidth - hFov)));
                double radiusGap = maxRadius / nCircles;
                double nextRadius = fovRadius;
                double maxSigma = MAX_SIGMA;
                double sigmaGap = maxSigma / nCircles;
                double sigma = sigmaGap;
                int kernelSize = getKernelSize(shortSide);
                double kernels[nCircles][kernelSize][kernelSize];
                int i;
                int j;

                struct image_matrix *circlesMask = malloc(sizeof(struct image_matrix));
                init(imgHeight, imgWidth, 0, circlesMask);

                for(i=0; i<nCircles; i++){

                        struct image_matrix *circleAux = malloc(sizeof(struct image_matrix));
                        createCircle(nextRadius, hFov, wFov, imgHeight, imgWidth, circleAux);
                        masIgualMatrix(circlesMask, circleAux);

                        getKernelGauss(kernelSize, kernelSize, sigma, kernels[i]);
                        sigma += sigmaGap;
                        nextRadius += radiusGap;
                }
                #pragma omp parallel for default(shared) private(j)
                for(i=kernelSize/2; i<imgHeight-kernelSize/2; i++){
                        for(j=kernelSize/2; j<imgWidth-kernelSize/2; j++){

                                if(circlesMask->dataM[i*imgWidth + j] != 0){
                                        int index = circlesMask->dataM[i*image->cols + j];
                                        convolutionPixel(image ,kernelSize ,kernelSize ,kernels[index-1], imageOut, i, j);
                                }
```

Ilustración 20. Código Foveal

Una de las partes que mayor complejidad ofrecen en este proyecto es la simulación de la fóvea. Recordemos que se crean n círculos que aplican un filtrado gaussiano. En un primer momento se desarrolló esta idea aplicando n gaussianas a la imagen entera, lo cual era muy ineficiente ya que hacía cálculos sobre píxeles que no iban a tener ese valor. Una mejora que se hizo fue el recorrer pixel a pixel y asignar la matriz de convolución correcta a cada uno de ellos.

5.2 Paralelización OpenCL

La paralelización mediante OpenCL no se llegó a terminar debido a varios problemas.

El desarrollo de OpenCL implica el uso de dispositivos de tecnología reciente, ya que gran parte de sus funciones se realizan con el conjunto de instrucciones SSE4, y nuestros equipos no poseen dicha tecnología.

Otro aspecto que debíamos tener en cuenta es la curva de aprendizaje de OpenCL. Se trata de una tecnología complicada y posee una curva más grande que la curva de aprendizaje de OpenMP. Aunque se abordó el desarrollo en un primer momento, ralentizó demasiado la obtención de código correcto, y nos vimos obligados a dejarlo fuera de los límites de esta investigación.

Por todos estos motivos decidimos descartar OpenCL, permitiéndonos concentrar la atención en la paralelización con OpenMP. Una vez terminada, y obtenidos los

resultados que produce este nivel de paralelización, podremos considerar la necesidad, y viabilidad, de realizar una paralelización más sofisticada con tecnología openCL.

5.3 Implantación en dispositivos

Una vez finalizada la paralelización del código, y con la idea de comprobar su efectividad, se ha probado en diferentes máquinas con diferentes arquitecturas.

5.3.1 Equipos Escritorio:

El primer paso fue probarlo en equipos de ámbito doméstico, así como en los entornos de los laboratorios de la Facultad de Informática.

Buscamos diferentes equipos, para obtener resultados lo más heterogéneos posible, que nos permitieran vislumbrar las cifras de rendimiento alcanzables en cada tipo de sistema. De esta forma, podemos comprobar cómo el número de hilos disponibles, y la tecnología del procesador, influye en el tiempo de ejecución.

5.3.2 Dispositivos móviles:

Con el fin de incorporar los algoritmos de realce en dispositivos móviles, lo cuales pueden ser accedidos fácilmente por personas con deficiencias visuales, buscamos un sistema portátil y eficiente, que pueda trabajar con OpenMP.

En el mercado, existen multitud de placas y sistemas empotrados que pueden dar cabida a nuestro código, y ejecutarlo de forma eficiente.

Samsung Exynos 5 Octa 5420

Este SoC(System on a Chip) contiene un procesador ARM-Cortex A15 de 28nm capaz de ejecutar 4 threads de forma simultánea a 1.7GHz. Está pensado para su uso en sistemas embebidos tales como cámaras digitales fijas y de procesamiento de video, sistemas de automoción, etc…

Entre sus características técnicas encontramos:

- 2 Procesadores: ARM Cortex A-15 y ARM Cortex A7.
- Caché de nivel 1 unificada para datos e instrucciones.
- Arquitectura tipo Harvard.
- Conjunto de instrucciones RISC.
- GPU ARM Mali-T628 MP6
- Unidad de gestión de memoria MMU.
- Bus de datos de 32 Bits.

El sistema cuenta con dos procesadores [16], de 4 cores cada uno. El ARM Cortex A-15, llamado “BIG core”, proporciona un mayor rendimiento, y opera a 1.8 GHz con una

caché de nivel 2 de 2MB. El segundo procesador ARM Cortex A-7, llamado “LITTLE core”, opera a 1.2 GHz con caché L2 de 512KB. La función principal de este procesador es servir de apoyo al primero, en tareas que exijan un incremento de la potencia de cálculo del sistema..

La gestión de la energía es realmente eficiente, pudiendo desconectar partes desocupadas para evitar consumos innecesarios, y reactivándolas cuando sean requeridas de nuevo. Además de alimentar los componentes de forma selectiva, permite disminuir la frecuencia del reloj del sistema cuando éste se encuentre desocupado.

La GPU ARM Mali-T628 MP6, está pensada para el procesado de gráficos que vayan destinados a una pantalla. Esta capacidad la consigue con sus 8 cores, que trabajan gracias a su unidad interna de gestión de la carga, la *Inter-Core Task Management*. Además de doblar el número de cores que su predecesor, el Mali-T624 GPU, también incorpora optimización del pipeline y soporte hardware para operaciones de 64 bits, ya sea con enteros, vectores o números en punto flotante. El manejo de este tipo de operandos, le permite acelerar notablemente las aplicaciones intensivas en datos, por lo que puede ser utilizado también en procesadores de imágenes. Una característica importante de los procesadores gráficos de la serie T600, es que permiten liberar la ocupación de la CPU principal, si esta se encuentra sometida a una carga excesiva.

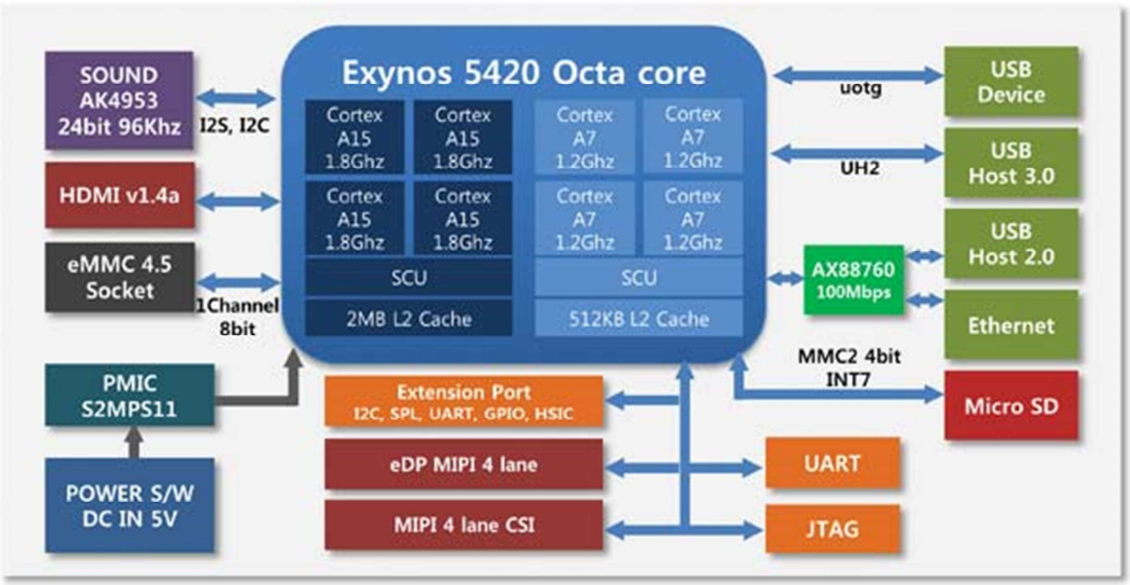


Ilustración 21. Arquitectura Exynos

Otras características contenidas en la placa son:

- Soporte OpenGL ES1.1/2.0/3.0, OpenCL1.1, OpenVG1.0.1, DirectX 11 y Google Renderscript
- Soporte HDMI 1.4a.
- Salida de un canal eDP Single WQXGA
- Sonido MIPI DSI y MIPI CSI

- Conexiones USB
- Conexión 10/100 Mbps Ethernet
- Memoria RAM eMMC 4.41 DDR en 8 bit 4GB
- Soporte SD
- Conexiones UART
- Conexiones GPIOs
- Botones y leds configurables
- Mantenimiento de dispositivos hasta 2.3A
- Corriente de entrada DC5V.

Capítulo 6. Resultados

En este capítulo analizaremos los tiempos de ejecución en distintas máquinas con diferentes prestaciones para ver cómo influye la paralelización realizada sobre el código en el aumento de productividad.

Todos los datos obtenidos en los diferentes dispositivos pueden verse en las **tablas y gráficos al final de este tema.**

6.1 Realización de las pruebas

Con el fin de obtener los resultados que se analizarán en los siguientes apartados, se realizó una batería de pruebas lo mas diversa posible en cuanto a tamaño de las imágenes, ya que este factor es el mas influyente en el tiempo de ejecución.

Estas pruebas dependiendo del entorno y de la imagen concreta con la que se realizará podían tardar bastante, por este motivo se realizó un banco de pruebas que leyera las imágenes de un directorio concreto, realizara las distintas simulaciones sobre ellas y guardase los datos obtenidos en un fichero de resultados para su posterior análisis.

6.2 Performance(Rendimiento)

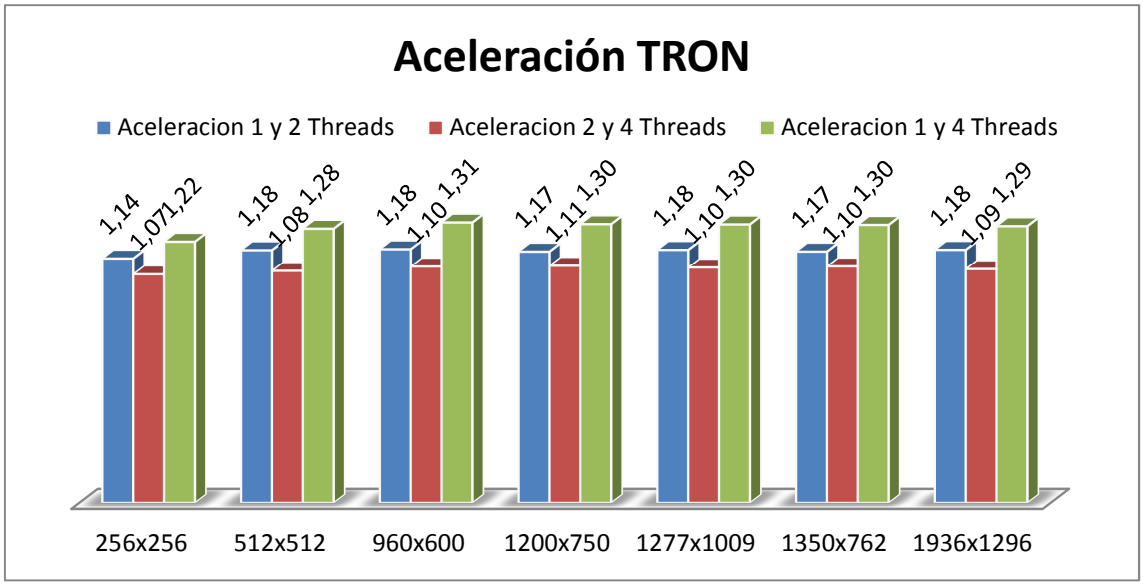
En el mercado existen diferentes programas para comprobar el rendimiento obtenido por un cierto sistema, en nuestro caso estos benchmark no sirven ya que no necesitamos calcular cómo de bueno es el sistema sino cómo de eficiente es nuestro código en base al número de hilos que se ejecuten de forma paralela.

Dada la diversidad de procesadores en los que ha testeado el sistema, limitaremos las mediciones de rendimiento al sistema portatil Exynos. El resto de medidas de tiempo están disponibles en las tablas del final del capítulo.

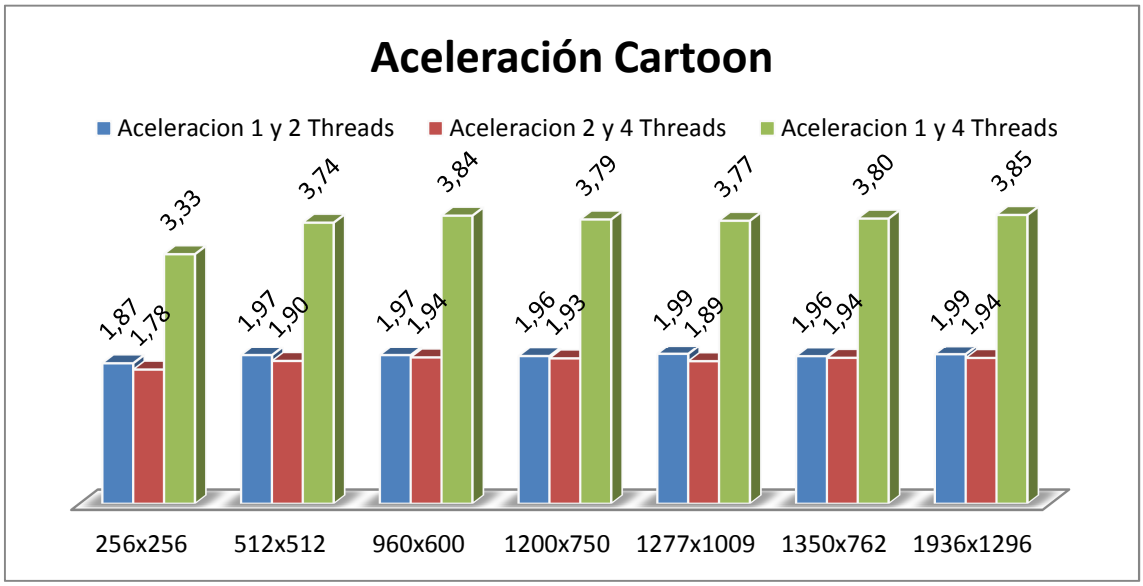
En los siguientes gráficos se calculará la aceleración del sistema respecto de la siguiente medida:

$$aceleración = TiempoEjecución_Y \,/\, TiempoEjecución_X$$

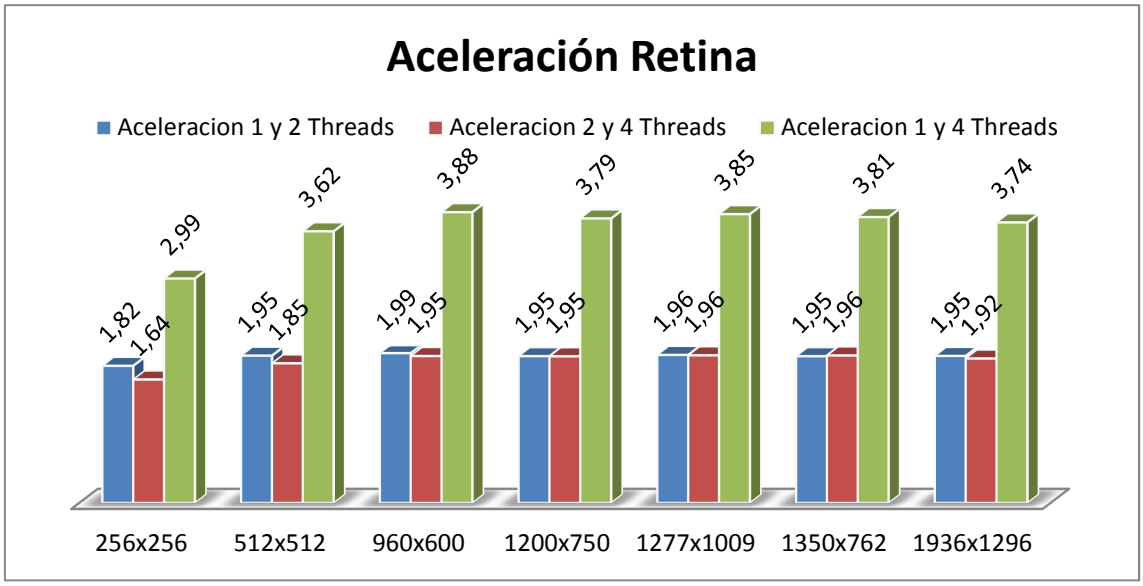
La cual nos indica que la configuración X es A veces más rápido que la configuración Y.



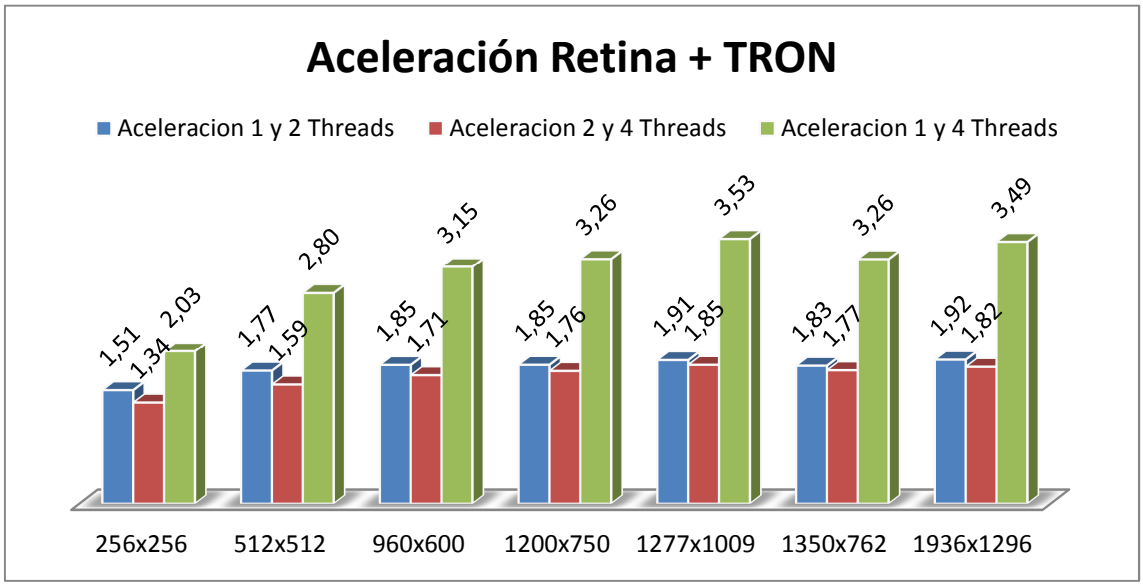
Gráfica 1



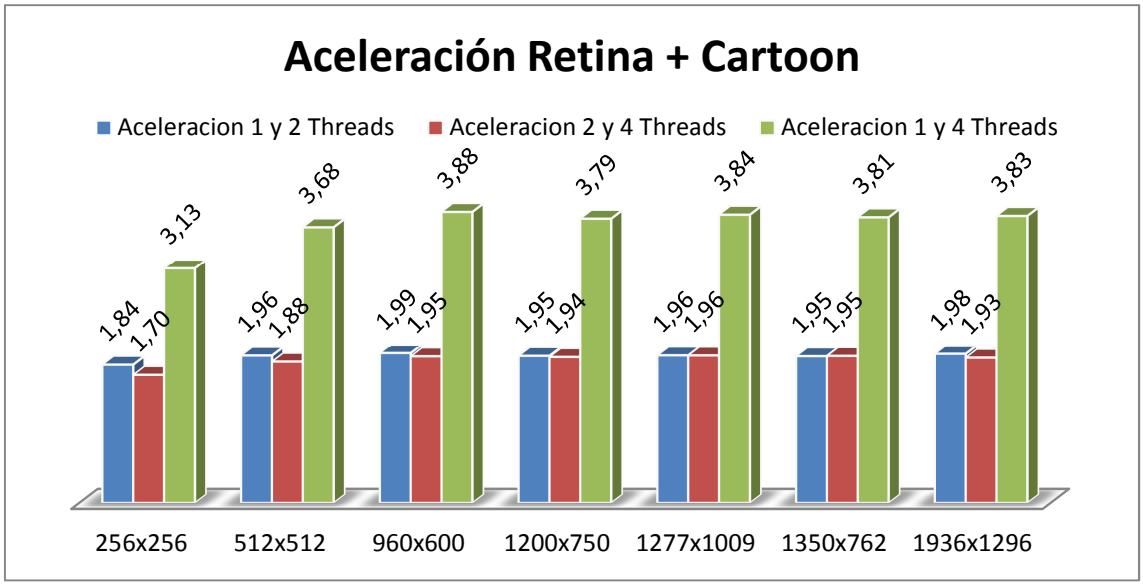
Gráfica 2



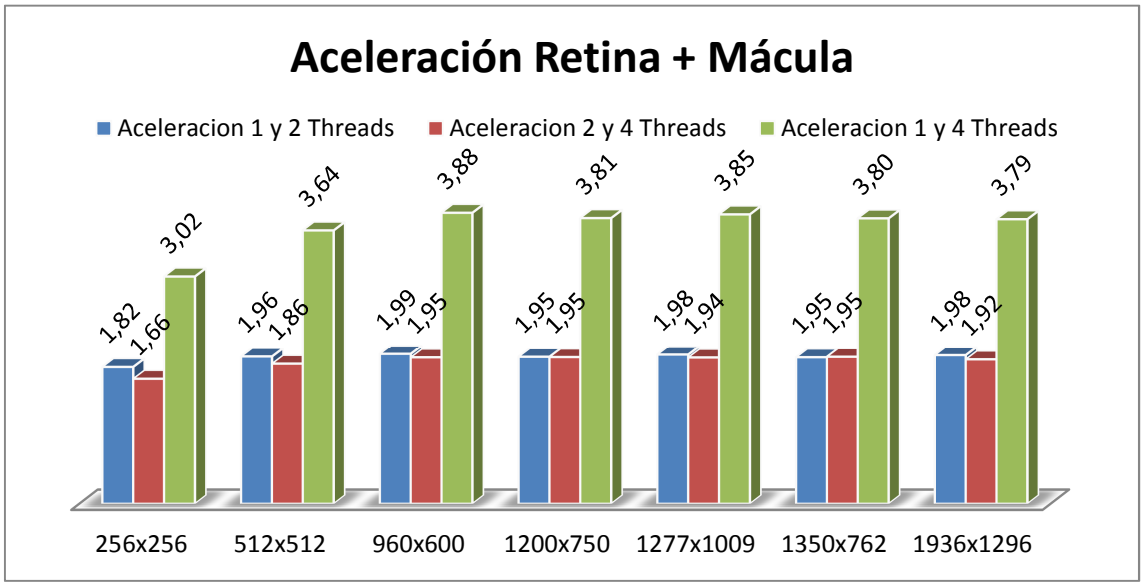
Gráfica 3



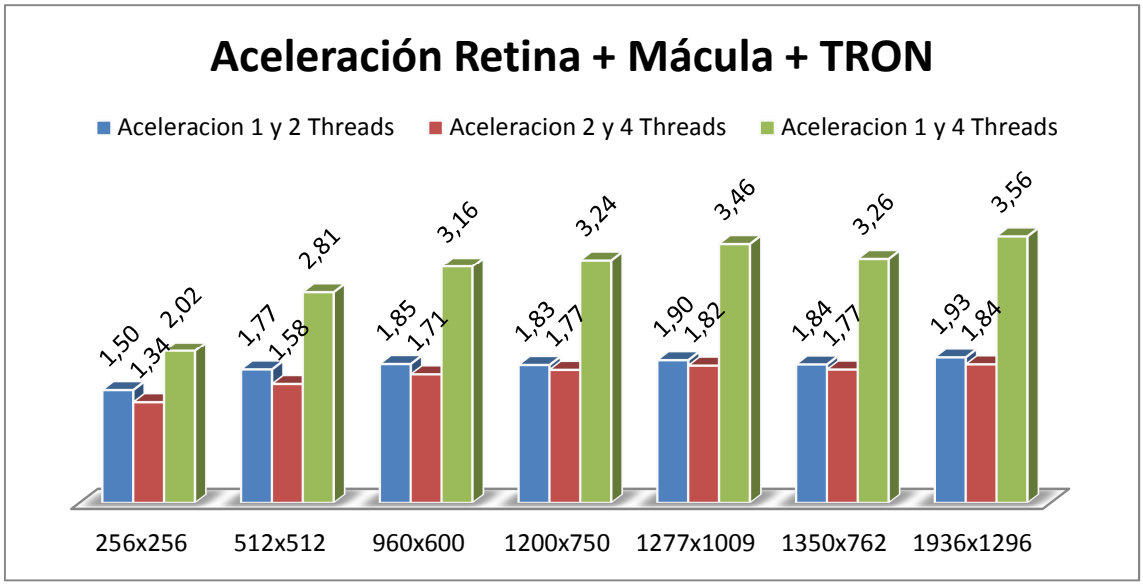
Gráfica 4



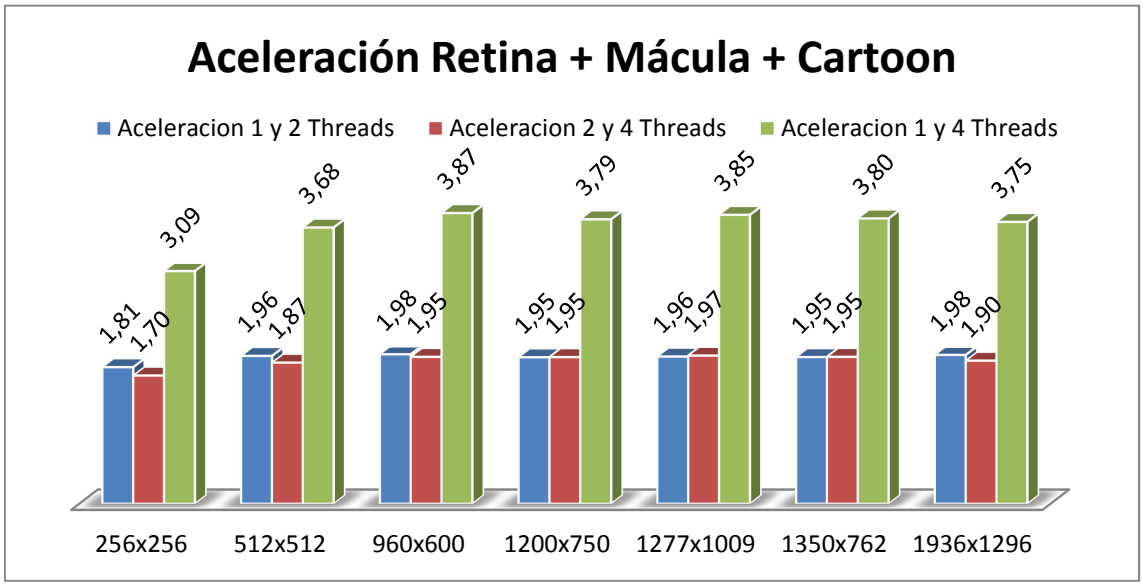
Gráfica 5



Gráfica 6



Gráfica 7



Gráfica 8

Una vez vistas las gráficas que recogen estos cálculos, podemos observar como el incremento del número de threads que participan de la ejecución mejora siempre el rendimiento obtenido.

Un dato que puede chocar en un principio es que el paso de 2 a 4 threads es menor que el de 1 a 2, esto no significa que estemos obteniendo una ejecución más lenta, sino que la aceleración obtenida en este salto es menor.

El salto más significativo se produce si observamos la ejecución pasando de 1 a 4 hilos. Es lógico ya que pasamos de ejecutar iteración una a una a dividir por cuatro la carga de trabajo.

También es destacable la comparación de rendimiento entre los distintos tamaños de imagen, a medida que crece el número de píxeles que contiene aumentamos su productividad, es decir notamos como es más eficiente. Esto se debe a que la carga de trabajo que soportan los threads es similar,cargando de forma dinámica las iteraciones, obligamos a todos a trabajar forma similar.

6.3 Calidad

En este apartado hablaremos de la calidad del producto final desde diferentes puntos de vista.

Desde el punto de vista funcional, nuestro proyecto cumple con creces los requisitos necesarios, siendo capaz de simular una retina humana para realizar pruebas sobre ella y los algoritmos de realce aumentan la calidad de la información que se recibe por parte de la imagen.

La portabilidad es posible como bien reflejan los resultados en la placa Exynos, el código que hemos desarrollado en C, no contiene dependencias de librerías que evitan arrastramos dependencias si exportamos a otros sistemas.

La eficiencia actualmente solo esta probada con imágenes estáticas pero a corto plazo seria posible aumentar su rendimiento y trabajar con imágenes en movimiento.

6.4 Escalabilidad

La escalabilidad de la paralelización se define como el nivel máximo de procesadores que trabajando en paralelo aumentan el rendimiento de un cierto algoritmo.

En las pruebas realizadas hemos sido capaces de ejecutar el sistema hasta en 8 threads de forma simultánea, obteniendo a medida que aumentamos el número de threads un mejor rendimiento.

Actualmente el sistema podría ejecutarse de forma paralela hasta un máximo del número de filas que contiene la imagen. Las pruebas realizadas se han ejecutado usando OpenMP y paralelizamos los recorridos de las imágenes según su fila, un número mayor no mejoraría el rendimiento ya que no tendríamos más iteraciones disponibles, desaprovechando recursos.

6.5 Análisis Crítico

El desarrollo de este proyecto, en especial la última parte realizada, no es sino un ejemplo de cómo hoy en día la programación en paralelo ayuda a resolver los problemas de una forma eficiente y más rápida. Este ha sido un objetivo básico en la concepción de la aplicación, presente en todas las fases por las que pasó su desarrollo. La razón por la cual es un aspecto básico es porque aspira a ser un sistema en tiempo real.

El procesamiento de imágenes es en general costoso e implica un gran número de cálculos, las resoluciones actuales hacen que estos cálculos se disparen, obligando en

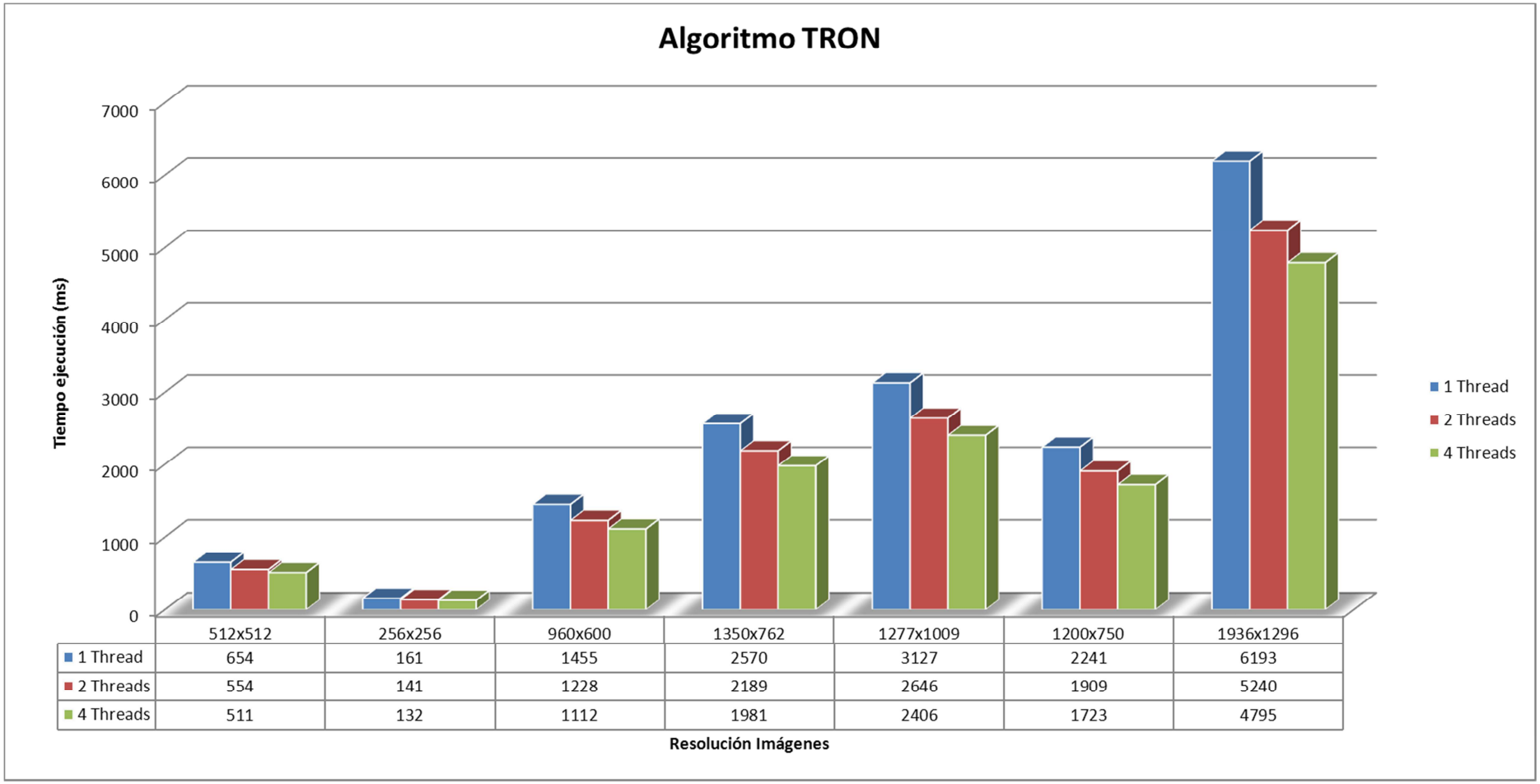
la medida que sea posible a paralelizar el código para poder reducir los tiempos de procesado.

La paralelización realizada consigue aumentar el rendimiento y disminuir los tiempos de procesado de una imagen en particular, antes de la paralelización del código se estudio la conexión de una webcam para capturar imágenes en tiempo real pero distaba mucho de aproximarse todavía a una ejecución de 24fps.

Dejando de lado la simulación de la retina y la DMAE que nos han servido de objeto de estudio, la ejecución de los algoritmos de realce TRON y Cartoonization solo se aproximaron a esta tasa de imágenes/seg el procesador i7.

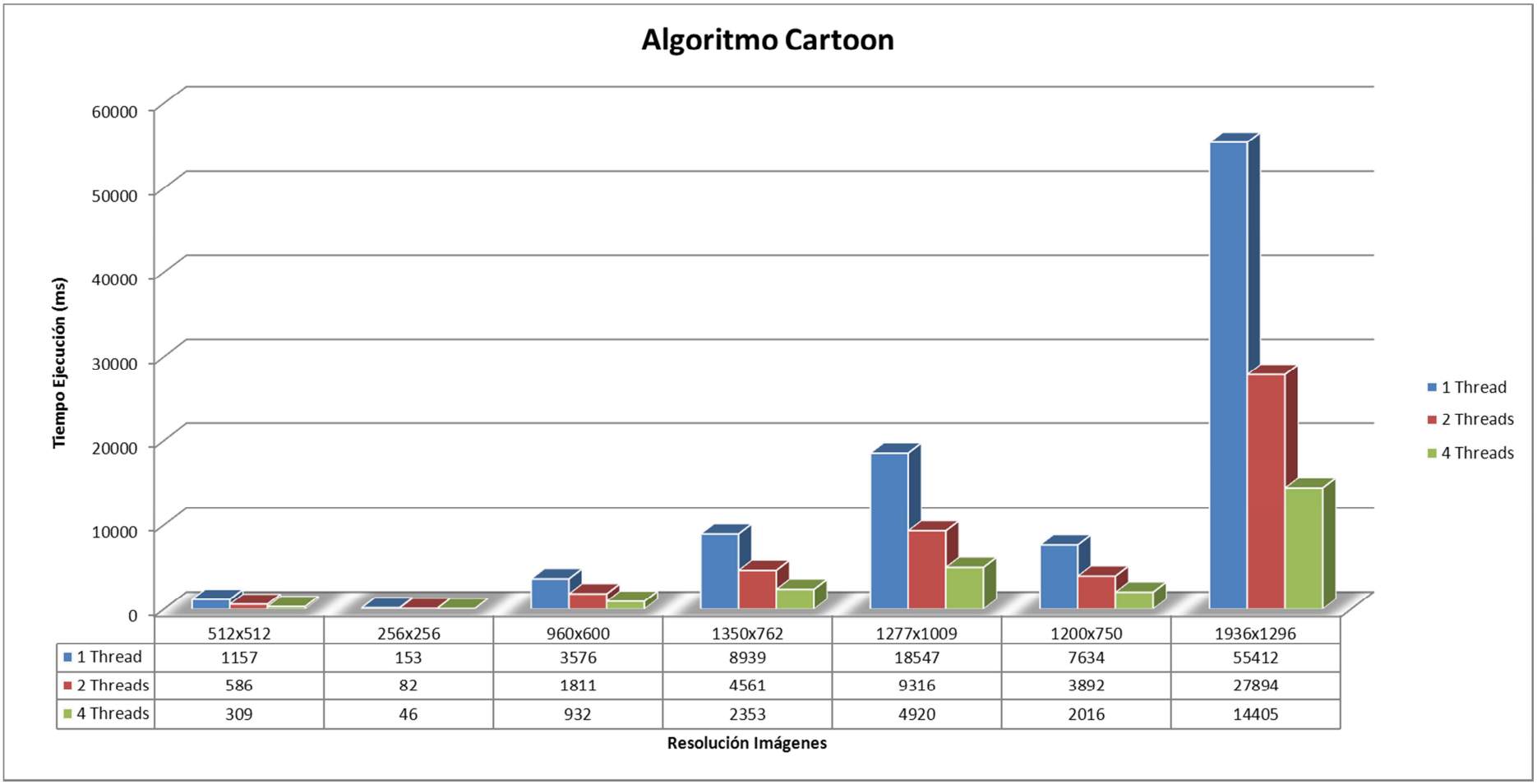
El tamaño de imagen necesario para que el ojo humano reciba información útil de una pantalla de un dispositivo de realidad aumentada es pequeño, 640x360 pixeles, con lo que nuestro trabajo se aproxima bastante a poder ser utilizado por personas con deficiencias visuales.

6.6 Graficas con tiempos de ejecución obtenidos en Samsung Exynos



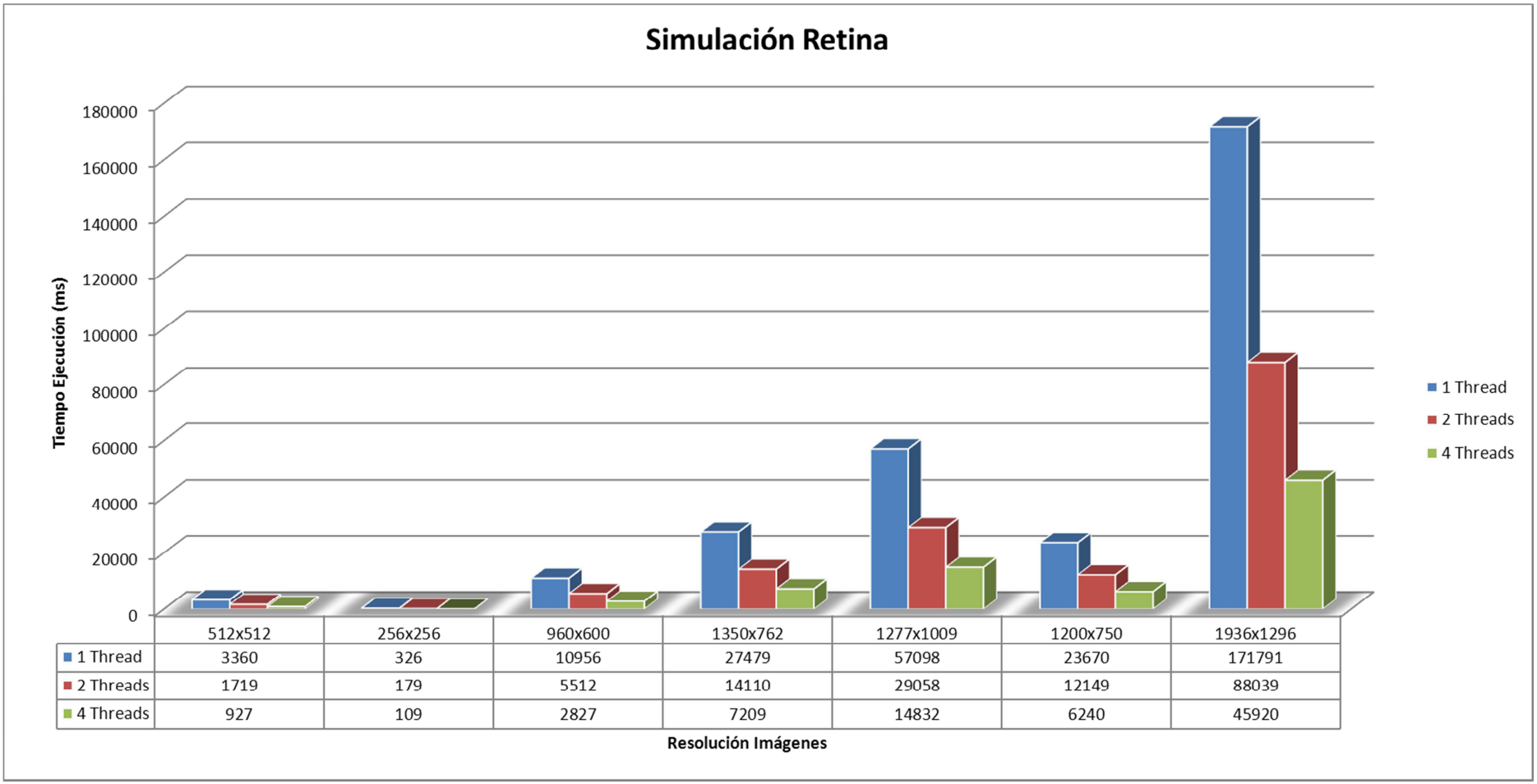
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	654	161	1455	2570	3127	2241	6193
2 Threads	554	141	1228	2189	2646	1909	5240
4 Threads	511	132	1112	1981	2406	1723	4795

Gráfica 9



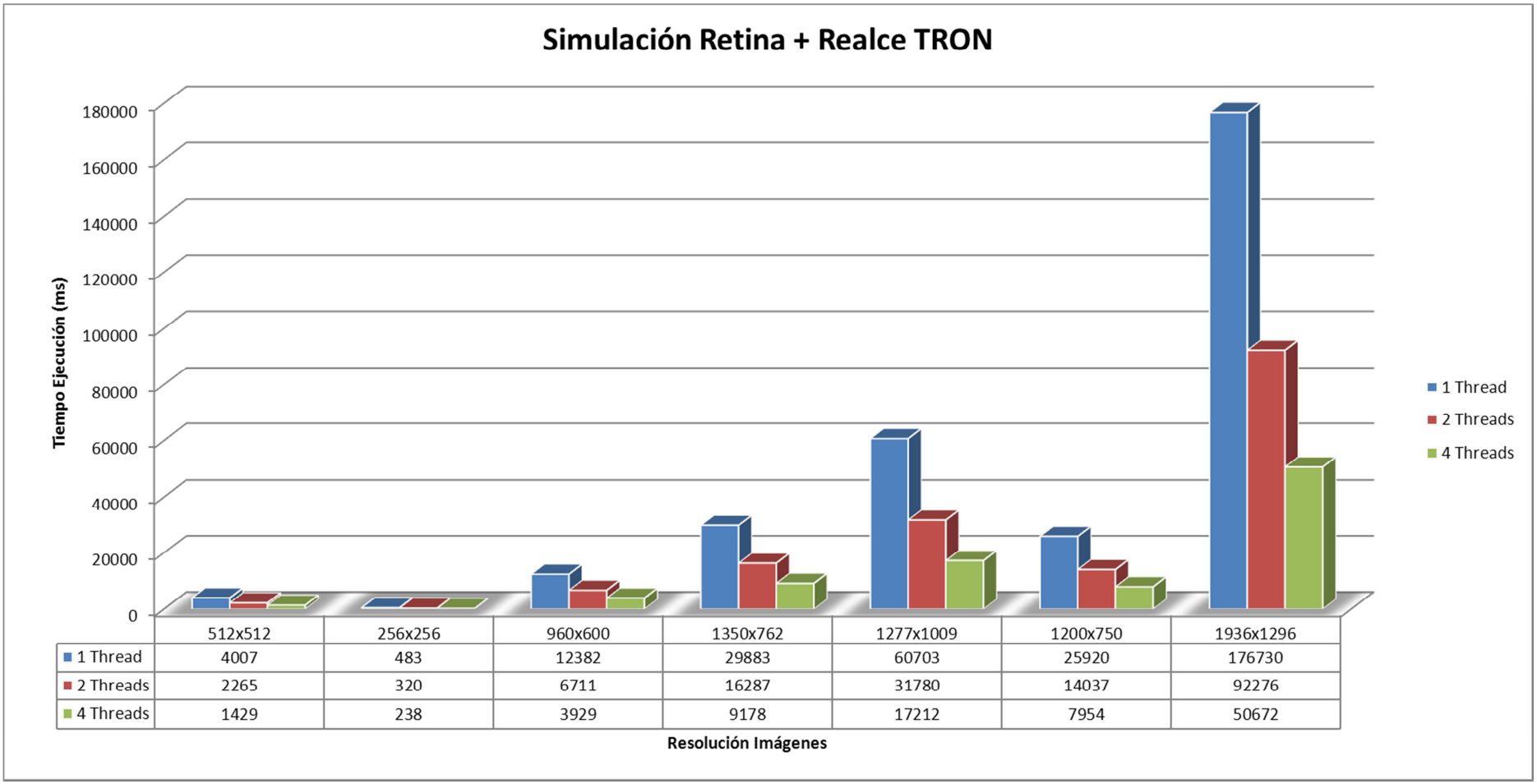
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	1157	153	3576	8939	18547	7634	55412
2 Threads	586	82	1811	4561	9316	3892	27894
4 Threads	309	46	932	2353	4920	2016	14405

Gráfica 10



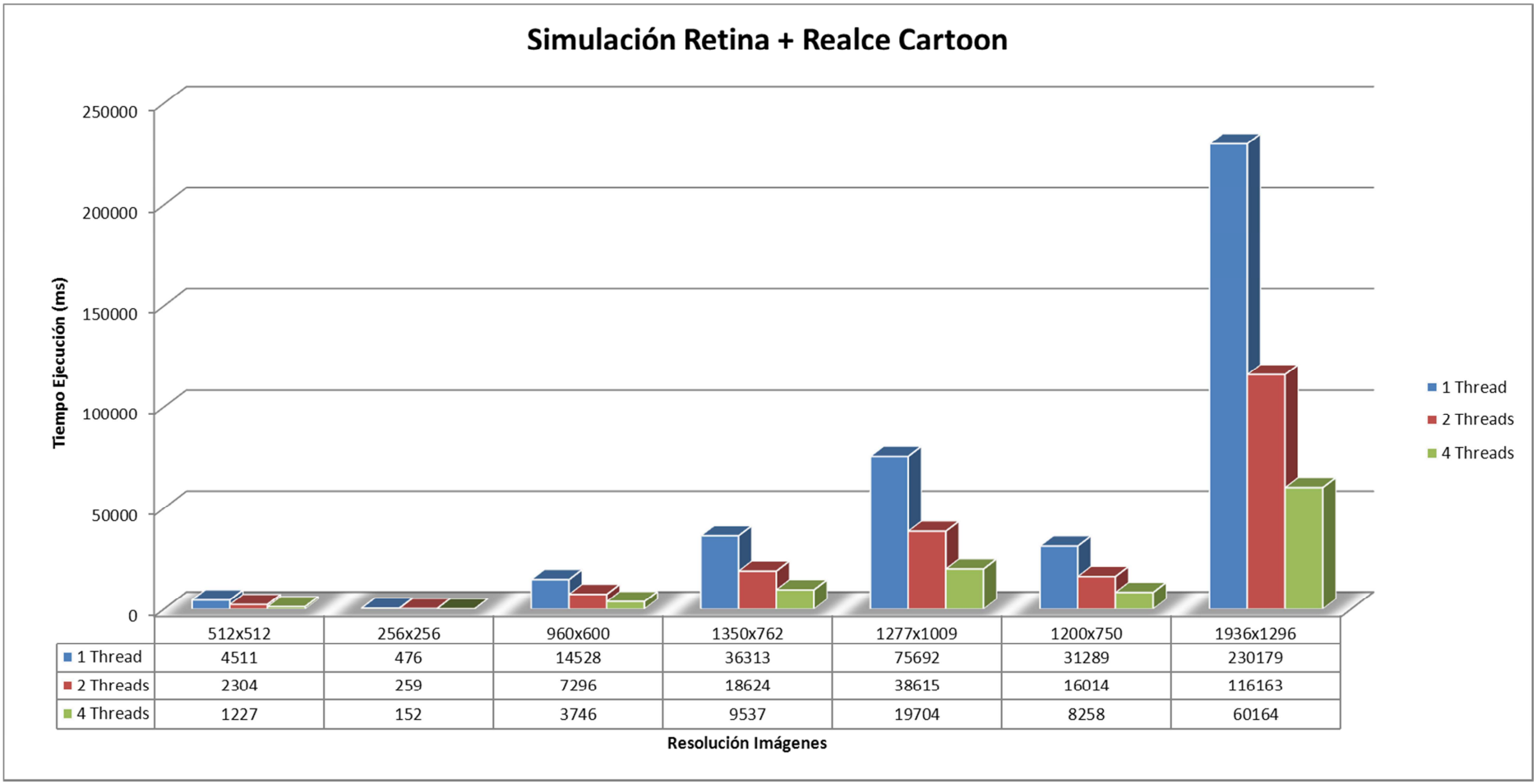
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	3360	326	10956	27479	57098	23670	171791
2 Threads	1719	179	5512	14110	29058	12149	88039
4 Threads	927	109	2827	7209	14832	6240	45920

Gráfica 11



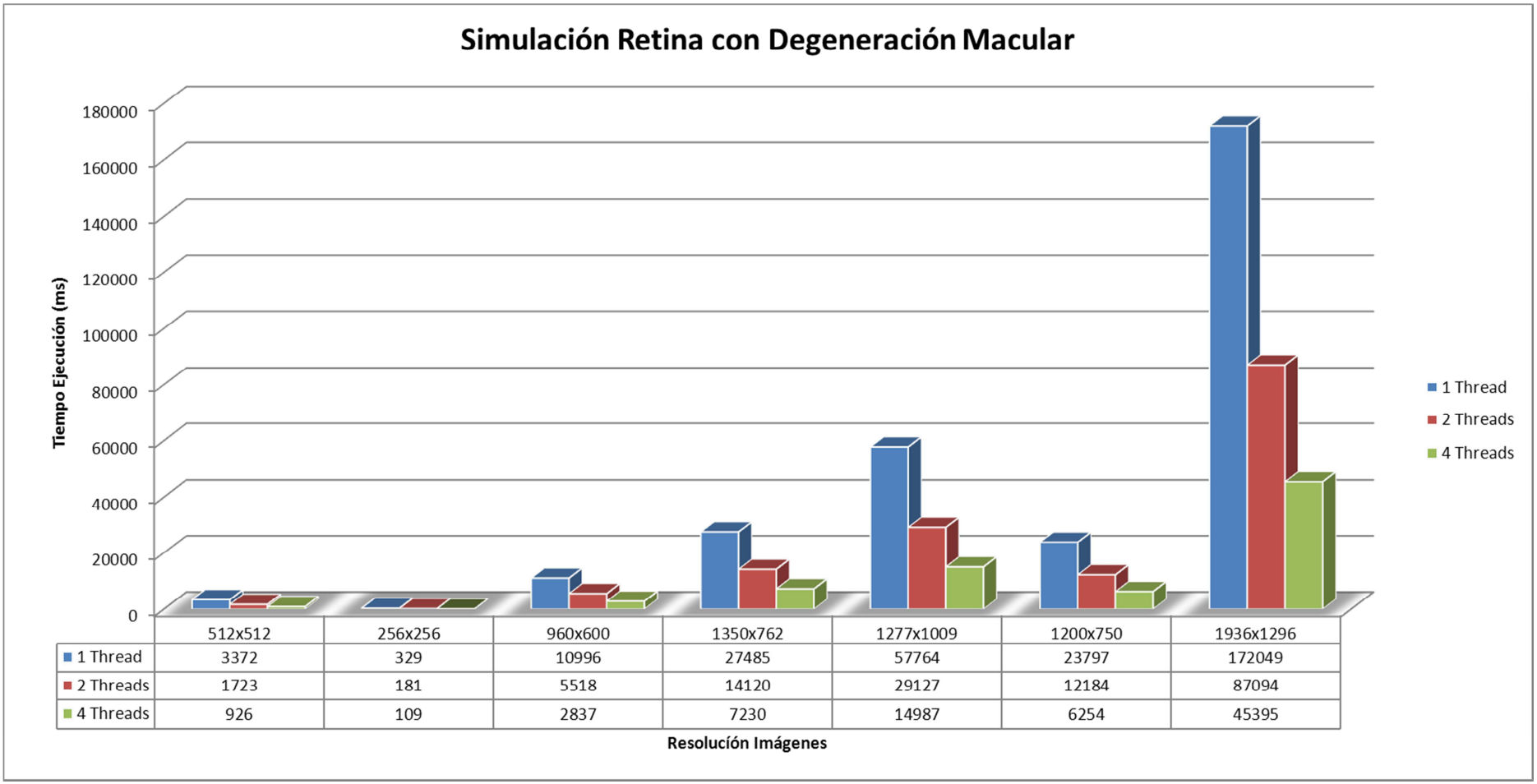
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	4007	483	12382	29883	60703	25920	176730
2 Threads	2265	320	6711	16287	31780	14037	92276
4 Threads	1429	238	3929	9178	17212	7954	50672

Gráfica 12



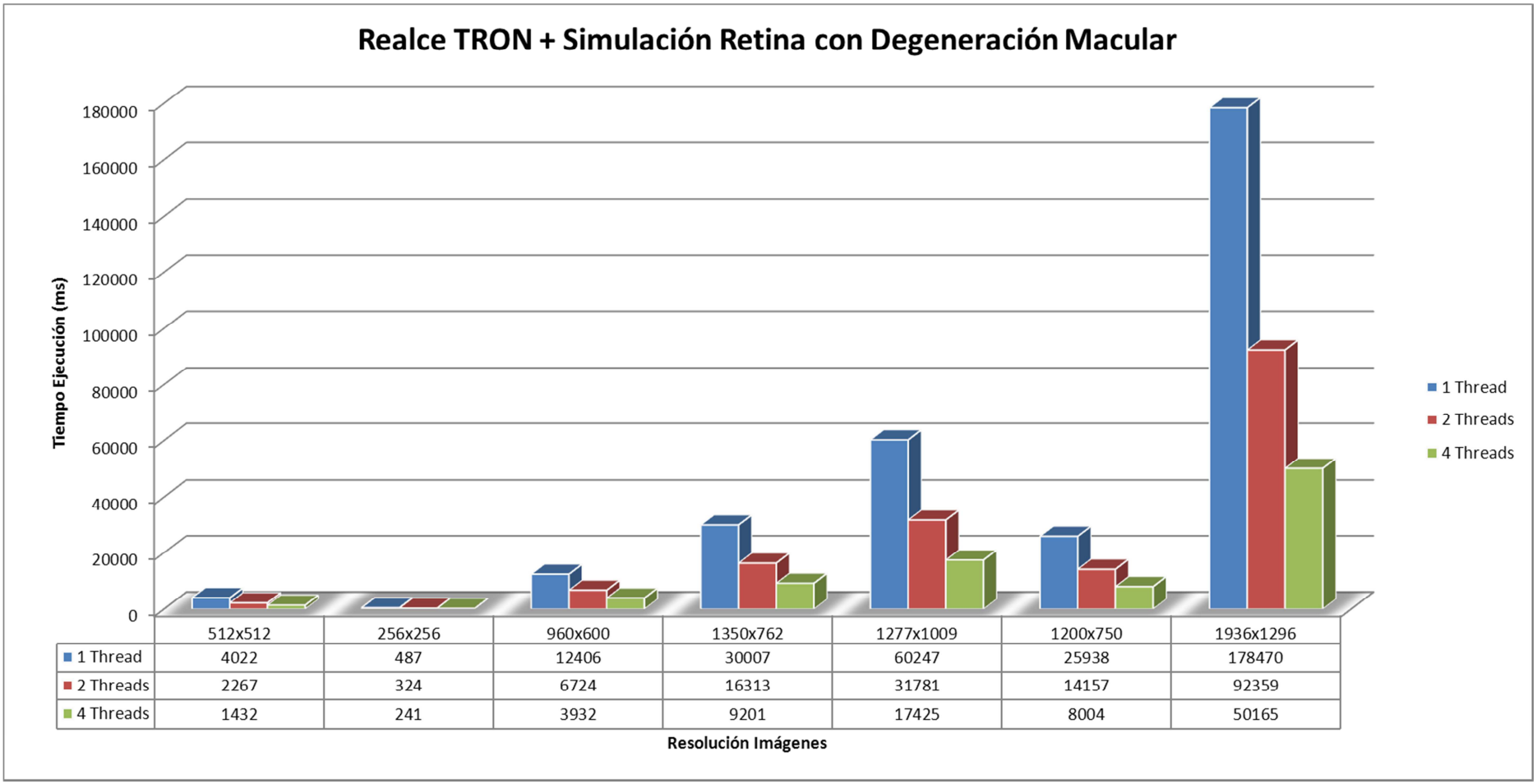
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	4511	476	14528	36313	75692	31289	230179
2 Threads	2304	259	7296	18624	38615	16014	116163
4 Threads	1227	152	3746	9537	19704	8258	60164

Gráfica 13



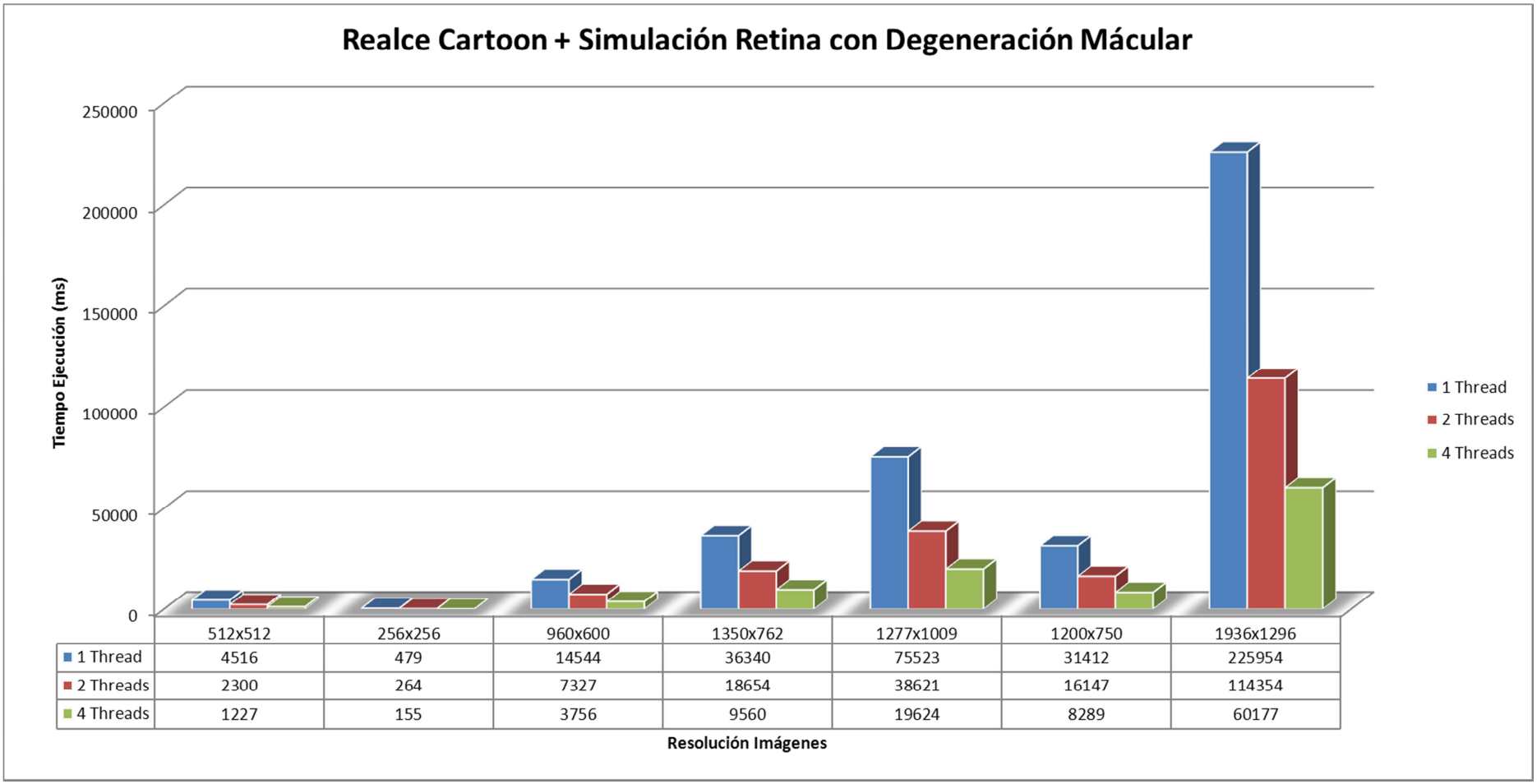
	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	3372	329	10996	27485	57764	23797	172049
2 Threads	1723	181	5518	14120	29127	12184	87094
4 Threads	926	109	2837	7230	14987	6254	45395

Gráfica 14



	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	4022	487	12406	30007	60247	25938	178470
2 Threads	2267	324	6724	16313	31781	14157	92359
4 Threads	1432	241	3932	9201	17425	8004	50165

Gráfica 15



	512x512	256x256	960x600	1350x762	1277x1009	1200x750	1936x1296
1 Thread	4516	479	14544	36340	75523	31412	225954
2 Threads	2300	264	7327	18654	38621	16147	114354
4 Threads	1227	155	3756	9560	19624	8289	60177

Gráfica 16

6.7 Tabla de Resultados 1

CPU	Velocidad GHz	#Threads	Tamaño imagen	Tiempos Ejecución(ms)							
				TRON	Cartoon	Retina					
							TRON	Cartoon	Macular	TRON + Macular	Cartoon + Macular
Samsung Exynos Octa 5420	1.9 GHz	1	512x512	654	1157	3360	4007	4511	3372	4022	4516
			256x256	161	153	326	483	476	329	487	479
			960x600	1455	3576	10956	12382	14528	10996	12406	14544
			1350x762	2570	8939	27479	29883	36313	27485	30007	36340
			1277x1009	3127	18547	57098	60703	75692	57764	60247	75523
			1200x750	2241	7634	23670	25920	31289	23797	25938	31412
			1936x1296	6193	55412	171791	176730	230179	172049	178470	225954
		2	512x512	554	586	1719	2265	2304	1723	2267	2300
			256x256	141	82	179	320	259	181	324	264
			960x600	1228	1811	5512	6711	7296	5518	6724	7327
			1350x762	2189	4561	14110	16287	18624	14120	16313	18654
			1277x1009	2646	9316	29058	31780	38615	29127	31781	38621
			1200x750	1909	3892	12149	14037	16014	12184	14157	16147
			1936x1296	5240	27894	88039	92276	116163	87094	92359	114354
		4	512x512	511	309	927	1429	1227	926	1432	1227
			256x256	132	46	109	238	152	109	241	155
			960x600	1112	932	2827	3929	3746	2837	3932	3756
			1350x762	1981	2353	7209	9178	9537	7230	9201	9560
			1277x1009	2406	4920	14832	17212	19704	14987	17425	19624
			1200x750	1723	2016	6240	7954	8258	6254	8004	8289
			1936x1296	4795	14405	45920	50672	60164	45395	50165	60177

Tabla 1

6.8 Tabla de Resultados 2

CPU	Velocidad GHz	#Threads	Tamaño imagen	Tiempos Ejecución(ms)							
				TRON	Cartoon	Retina					
							TRON	Cartoon	Macular	TRON + Macular	Cartoon + Macular
Intel Core2Duo T5800	2 GHz	2	512x512	434	1346	3606	3969	4880	3738	4079	4921
			256x256	109	144	365	479	515	398	462	528
			960x600	934	4514	12519	13613	16729	12972	13686	16999
			1350x762	1790	11373	32366	33981	43262	33638	34586	43890
			1277x1009	2106	25693	72503	74222	96902	75030	76051	97140
			1200x750	1453	10486	28341	29777	38240	29320	30343	38398
			1936x1296	4162	66727	223563	226707	297191	227337	232327	299497
Intel Core2Duo T2400	1.83 GHz	2	512x512	468	1040	3120	3446	3460	3220	3546	3871
			256x256	119	117	331	433	430	348	465	445
			960x600	1041	3488	11108	11339	11382	11640	11539	11781
			1350x762	1844	8948	30559	28922	30305	30703	29312	30508
			1277x1009	2247	19548	68323	63651	65254	68431	64651	65284
			1200x750	1611	7809	28032	25697	26241	28300	25912	26541
			1936x1296	4438	59660	185278	195554	199730	185478	195654	201810
Intel Core i7-2600	3.46 GHz	8	512x512	132	161	533	681	685	533	694	686
			256x256	33	18	58	92	78	59	92	80
			960x600	284	559	1749	1992	2249	1715	1985	2265
			1350x762	494	1377	4386	4796	5710	4285	4809	5661
			1277x1009	611	3020	9611	10175	12591	9559	10235	12565
			1200x750	426	1208	3826	4215	4974	3782	4244	4956
			1936x1296	1220	9254	30477	31614	39690	30565	31625	39640

Tabla 2

Capítulo 7. Desarrollo Futuro

En este proyecto se ha desarrollado un modelo de retina artificial sobre el que realizar simulaciones con visión normal y con degeneración macular. En un futuro se podrían agregar otro tipo de enfermedades visuales al modelo de retina actual, con el fin de obtener resultados de cómo mejora el reconocimiento de formas existentes

Se han desarrollado también un par de algoritmos de realce, pero existen otro tipo de algoritmos que podrían implementarse que en combinación con la DMAE y otros tipos de enfermedades visuales ampliar el estudio de cuál algoritmo funciona mejor con una determinada enfermedad.

En el apartado de paralelización podría concluirse el estudio de OpenCL así como usar otras tecnologías alternativas que explotan la paralelización en GPU, como por ejemplo CUDA.

Dejando de lado el desarrollo, la implementación en dispositivos de realidad aumentada del estilo Google Glass darían el paso final al proyecto, pudiendo observar su funcionamiento en pacientes reales más allá de la mera simulación.

En otro ámbito el realce de bordes en tiempo real podría aplicarse en vehículos, de forma que ayude a detectar obstáculos presentes en el camino.

Bibliografía

[1] W. H. Organization, *Clasificación Internacional del Funcionamiento, de la Discapacidad y de la Salud: CIF*. Ministerio de Trabajo y Asuntos Sociales, Secretaría General de Asuntos Sociales, Instituto de Migraciones y Servicios Sociales (IMSERSO), 2001.

[2] F. Gómez-Ulla y S. Ondategui-Parra, «Informe sobre la ceguera en España». Ernst & Young, Fundación Retinaplus+, 2012.

[3] «Mácula lútea», *Wikipedia, la enciclopedia libre*. 07-ene-2014.

[4] W. I. Al-Atabany, M. A. Memon, S. M. Downes, y P. A. Degenaar, «Designing and testing scene enhancement», *Biomed. Eng. Online*, vol. 9, pp. 1–25, 2010.

[5] G. P. Martinsanz y J. M. de la C. García, *Visión por computador: imágenes digitales y aplicaciones*. Ra-Ma, 2001.

[6] «Portable Network Graphics», *Wikipedia, la enciclopedia libre*. 14-may-2015.

[7] H. Winnemöller, S. C. Olsen, y B. Gooch, «Real-time video abstraction», en *ACM Transactions On Graphics (TOG)*, 2006, vol. 25, pp. 1221–1226.

[8] A. L. Kierszenbaum y L. Tres, *Histology and cell biology: an introduction to pathology*. Elsevier Health Sciences, 2015.

[9] D. H. Hubel y T. N. Wiesel, «Receptive fields of single neurones in the cat's striate cortex», *The Journal of physiology*, vol. 148, n.º 3, pp. 574–591, 1959.

[10] P.-R. Chang y B.-F. Yeh, «Retina-like image acquisition system with wide-range light adaptation», en *Visual Communications,'91, Boston, MA*, 1991, pp. 456–469.

[11] J. D. Murray y W. vanRyper, *Encyclopedia of Graphics File Formats (2Nd Ed.)*. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 1996.

[12] A. Averbuch, M. Israeli, y L. Vozovoi, «A fast Poisson solver of arbitrary order accuracy in rectangular regions», *SIAM Journal on Scientific Computing*, vol. 19, n.º 3, pp. 933–952, 1998.

[13] G. Bradski y A. Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Inc., 2008.

[14] C. Nvidia, «Compute unified device architecture programming guide», 2007.

[15] R. Chandra, *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[16] P. Greenhalgh, «Big. little processing with arm cortex-a15 & cortex-a7», *ARM White paper*, pp. 1–8, 2011.

Glosario

Chunk: en procesamiento paralelo, se denomina chunk a cada uno de los subproblemas en que se divide el procesamiento sobre un conjunto extenso de datos.

Filtros ópticos: Son lentes especiales que se encargan de filtrar las luces luces azules, de menor longitud de onda, causantes de los deslumbramientos. Suelen ser utilizados por pacientes con síntomas relacionados con una hipersensibilidad a la luz, como por ejemplo, en pacientes con fotofobia.

Sección crítica: una sección crítica es un recurso compartidos por threads o procesos, que no debe ser accedido simultáneamente por más de una entidad. Se suele controlar su acceso con mecanismos de sincronización de procesos y threads. Algunos ejemplos de regiones críticas pueden ser conjuntos de datos o dispositivos.

Telemicroscopios: Un telemicroscopio es un dispositivo de ayuda a la baja visión que, mediante lentes, asiste la distinción de objetos a media distancia.

Telescopios: Un telescopio es un dispositivo similar al telemicroscopio para la asistencia de deficiencias visuales; pero diseñado para facilitar la visión lejana.

Test de Landolt: el test de Landolt se utiliza para medir la agudeza visual en niños de una determinada edad, y en personas adultas que no sepan identificar las letras del alfabeto latino; bien porque no sepan leer, o porque sufran alguna patología, como por ejemplo, la dislexia. El test utiliza exclusivamente caracteres con forma de anillo circular; pero introduciendo una discontinuidad en el trazo, que apunta en alguna dirección. Las personas que se someten a este test tienen que identificar, para cada tamaño de carácter, la orientación de la discontinuidad.

Test de Lea: el test de Lea es utilizado para medir la agudeza visual en niños que aún no son capaces de identificar, ni siquiera, los caracteres más básicos del alfabeto latino. En este test, el ojo debe identificar figuras básicas con forma de cuadrado, círculo, casa o manzana.

Test de Snellen: el test de Snellen es un test para medir la agudeza visual en personas que sepan identificar las letras del alfabeto latino. Lleva el nombre de su descubridor, el oftalmólogo holandés Herman Snellen, y consiste en distinguir letras de diferentes tamaños en orden decreciente, a una determinada distancia. Cada tamaño de letra representa un nivel de agudeza visual, que será mayor, cuanto más pequeño sea el tamaño de la letra. Es el más usado en Europa.

Arquitectura tipo Harvard Es una arquitectura de computadores que se caracteriza principalmente por introducir dos memorias, una para almacenar datos y otra para almacenar instrucciones; a diferencia de la arquitectura Von Neumann, en la que esta característica no está disponible por existir, únicamente, un espacio de memoria.

Conjunto de instrucciones RISC: Reduced instruction set computer, es una filosofía para el diseño de CPU en la que el objetivo es construir procesadores que ejecuten un reducido número de instrucciones, de formato fijo, y más rápidas de procesar que las que incorporan los procesadores diseñados en base al paradigma opuesto, CISC (complex instruction set computer) .