PONTIFICIA UNIVERSIDAD CATÓLICA DEL PERÚ

FACULTAD DE CIENCIAS E INGENIERÍA

IMPLEMENTACIÓN DEL DETECTOR DE ESQUINAS DE HARRIS EN LA PLATAFORMA JETSON TK1

Tesis para optar el Título de Ingeniero Electrónico, que presenta el bachiller:

Hector Francisco Chahuara Silva

ASESOR: Paul Antonio Rodríguez Valderrama

Lima, 2017

Dedicado a mi familia, por su cariño, apoyo y confianza a lo largo de mi paso por la universidad.

Un agradecimiento especial a mi asesor, Dr. Paul Rodríguez, por su apoyo y recomendaciones.

Asimismo, agradezco a mis amigos del Grupo de Procesamiento Digital de Señales e Imágenes Gustavo Silva y Jorge Quesada y a mis amigos del Grupo de Microelectrónica Stefano Sosa y André Seminario.

Resumen

Las esquinas son puntos invariantes, estructurales y con alto contenido de información en una imagen. Estas son usadas en aplicaciones importantes de Procesamiento de imágenes o video entre las cuales destacan Navegación de UAVs (*Unmanned Aerial Vehicles*) o Detección de objetos que son importantes en distintas áreas y tienen requerimientos de tiempo real. Entre las soluciones para detección de esquinas propuestas destaca el Detector de esquinas de Harris, el cual demuestra ser robusto y eficiente.

El uso de plataformas que permiten realizar procesamiento paralelo permiten implementar métodos de alto costo computacional con un bajo tiempo de procesamiento. Entre ellas destacan los GPU (*Graphics Processing Unit*) que generalmente tienen un alto consumo energético, lo cual es inconveniente en aplicaciones dirigidas a dispositivos móviles como celulares, robots, drones, entre otros. Por ello, plataformas basadas en *mobile* CPU que tienen bajo consumo energético son opciones a tomar en cuenta.

En la presente tesis se propone el diseño e implementación del Detector de esquinas de Harris en la plataforma Jetson TK1 de Nvidia la cual se distingue por su bajo consumo energético y alto rendimiento. El método será implementado en MATLAB, ANSI-C y CUDA. Los resultados muestran que la implementación en CUDA presentada es hasta 32.08 veces aproximadamente más rápida que la implementación en ANSI-C y permite procesar imágenes de resolución full HD (1920 x 1080) en tiempo real. Además, es comparable a implementaciones en software en plataformas con mayores recursos e implementaciones en hardware usando FPGAs (*Field Programmable Gate Array*).

La estructura del presente documento es la siguiente: En el primer capítulo se presenta el estado del arte sobre detección de esquinas y el Detector de esquinas de Harris. En el segundo capítulo se presenta la plataforma Jetson TK1. El diseño del algoritmo paralelo se detalla en el tercer capítulo. Por último, se presenta la implementación y sus resultados en el cuarto capítulo, seguido de las conclusiones y recomendaciones.

Índice general

Introducción

Las esquinas son puntos invariantes, estructurales y con alto contenido de información para una imagen. Debido a ello, estas son usadas en aplicaciones importantes tales como Seguimiento de objetos [1], Reconstrucción 3D [2], Registro de imágenes [3], Navegación de robots [4], entre otras. Por ello, su detección es una tarea importante y varias soluciones como el detector SUSAN [5] o el detector FAST [6], entre otras, han sido propuestas. Entre ellas destaca el detector de esquinas de Harris, el cual tiene alto costo computacional y gran potencial paralelizable, y se caracteriza por su invarianza ante transformaciones de imágenes y cierta robustez ante ruido respecto a otras soluciones. Esto ha motivado a que su uso se extienda a aplicaciones que tienen requerimientos de tiempo real y bajo consumo energético.

Operaciones de uso extendido en procesamiento de imágenes y video como convolución o mutiplicación matriz vector están limitadas por su elevado costo computacional, por ello, su implementación en plataformas de alto rendimiento o el uso de modelos de programación que permitan realizar implementaciones que aprovechen el paralelismo de estos métodos es requerida. El uso de GPUs (*Graphics Processing Unit*), FPGAs (*Field Programmable Arrays*) o de las unidades SIMD (*Single Instruction Multiple Data*) que algunos procesadores poseen es indispensable para implementar métodos costosos con bajo tiempo de procesamiento. Entre ellos destaca CUDA (*Compute Unified Device Architecture*) [7], modelo de programación creado por Nvidia, que permite realizar implementaciones eficientes de algoritmos con gran costo computacional en GPU, lo cual ha motivado su uso para distintas aplicaciones.

En la presente tesis, se propone el diseño e implementación de un algoritmo paralelo para el detector de esquinas de Harris en la plataforma Jetson TK1. La plataforma Jetson TK1 está basada en el SoC (*System on chip*) Tegra K1 que se encuentra integrado por un procesador ARM y un GPU de Nvidia. Esta tiene un rendimiento de hasta 300 GFLOPS y un consumo energético menor a 11 watts, por lo cual es una alternativa adecuada para la implementación eficiente de un método de alto costo computacional y potencial paralelizable a bajo consumo energético.

Capítulo 1

Detección de esquinas

1.1. Importancia

Las esquinas son puntos invariantes, estructurales y con alto contenido de información para una imagen. Por ello, aplicaciones en el área de Procesamiento digital de señales y video como Estabilización de video, Reconocimiento de objetos, Reconstrucción 3D, Registro de imágenes, Seguimiento de objetos, entre otras usan esquinas. Entre estas aplicaciones destacan aquellas que tienen requerimientos de tiempo real. Sin embargo, los diferentes métodos de detección operan usando todos los datos de una imagen lo que resulta en un alto costo computacional.

Una esquina, visualmente, puede ser definida como un punto donde dos o más bordes se intersectan. Por ello, el trabajo de detección de estos puntos puede parecer trivial, sin embargo, la detección de esquinas usando métodos computacionales no lo es. Varios métodos han sido planteados para la detección de esquinas, cada uno usa criterios diferentes de detección y tienen resultados diferentes en precisión, exactitud y costo computacional. Así, la selección de un método de detección depende del análisis del costo computacional de la solución y de la precisión y exactitud requeridas para una aplicación.

1.2. Detector de esquinas de Harris

El detector de esquinas de Harris [8] es un método de detección de esquinas limitado a imágenes en escala de grises [9] que hace uso de la matriz de autocorrelación o tensor estructural. Esta matriz permite determinar una métrica de esquinas dependiente de sus autovalores, los que a su vez, son proporcionales a las magnitudes direccionales de la curvatura de los bordes.

1.2.1. Revisión del detector de esquinas de Moravec

El detector de esquinas de Moravec [10] usa la métrica E que mide las variaciones del cambio de la intensidad de la imagen I respecto a una ventana ω según la ecuación 1.1:

$$E(x,y) = \sum_{u,v} \omega\,(u,v)\,|I\,(x+u, y+v) - I\,(x,y)\,|^2 \tag{1.1}$$

El criterio de detección de esquinas usando la métrica de esquinas de Moravec se verifica analizando las correspondencias entre las regiones de la imagen analizada con los valores de la métrica correspondientes. Así, regiones planas en la imagen analizada corresponderán a valores bajos de la métrica, mientras que las regiones de los bordes acumularán un valor más elevado de métrica debido a la variación de intensidad en una o más direcciones. Finalmente, las esquinas tendrán correspondencia con los valores más altos de la métrica de esquinas.

1.2.2. Formulación matemática del algoritmo de detección de esquinas de Harris

La formulación del detector de esquinas Harris aproxima la métrica E del detector de esquinas de Moravec usando series de Taylor de orden dos. Aplicando esta aproximación, la complejidad de la métrica de esquinas se ve reducida:

$$E(x,y) = \sum_{u,v} \omega\,(u,v) \left(x\frac{\partial}{\partial x} I + y\frac{\partial}{\partial y} I - \mathcal{O}\left(x^2, y^2\right) \right)^2 = \sum_{u,v} \omega\,(u,v)\,(xI_x + yI_y)^2$$

La ventana ω propuesta por Harris es un filtro gaussiano G, luego, la sumatoria anterior puede reformularse como una operación de convolución:

$$E(x,y) = G * (xI_x + yI_y)^2 = x^2\left(G * I_x^2\right) + y^2\left(G * I_y^2\right) + 2xy\,(G * I_x I_y)$$

$$E(x,y) = \begin{bmatrix} x & y \end{bmatrix} \begin{bmatrix} G * I_x^2 & G * I_x I_y \\ G * I_x I_y & G * I_y^2 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

La medida E es una forma cuadrática $E = v^T M v$ donde:

$$M = \begin{bmatrix} G * I_x^2 & G * I_x I_y \\ G * I_x I_y & G * I_y^2 \end{bmatrix} \qquad v = \begin{bmatrix} x \\ y \end{bmatrix}$$

La matriz de autocorrelación o tensor estructural M es una matriz simétrica, por lo tanto diagonalizable. Siendo λ_1 y λ_2 valores propios de M, la métrica E se puede reescribir como:

$$E(x,y) = \begin{bmatrix} k_1 x & k_2 y \end{bmatrix} \begin{bmatrix} \lambda_1 & 0 \\ 0 & \lambda_2 \end{bmatrix} \begin{bmatrix} k_1 x \\ k_2 y \end{bmatrix}$$

Así, la expresión E depende de los valores propios de la matriz de autocorrelación M y se puede establecer un criterio de análisis que los incluya. Luego, los valores propios de M tienen una gran influencia en la determinación del punto, así, Harris propone el uso de una función de métrica de esquinas (*Corner Response Function* o CRF) relacionada a ellos según la ecuación 1.2:

$$CRF = det(M) - k\, tr^2(M) \tag{1.2}$$

Donde $det(.)$ y $tr(.)$ son las operaciones determinante y traza de una matriz respectivamente y k es un factor de sensibilidad. Tanto el determinante como la traza son operaciones que pueden ser expresadas en función de los valores propios de una matriz, entonces, la función de respuesta de esquinas puede expresarse según la ecuación 1.3:

$$CRF = \lambda_1 \lambda_2 - k(\lambda_1 + \lambda_2)^2 \tag{1.3}$$

La métrica de esquinas, entonces, tendrá un comportamiento acorde a los valores propios pero dependiente del factor k, el cual está ilustrado en la figura 1.1. El valor de CRF será pequeño o negativo si los dos valores propios son pequeños (punto plano) o si uno de ellos es muy grande en comparación del otro (borde), puesto que en este caso, la traza será mayor al determinante. En cambio, si ambos valores propios tienen un valor alto, el valor de la CRF será alto también pues el determinante será mayor que la traza y el punto analizado corresponderá a una esquina.

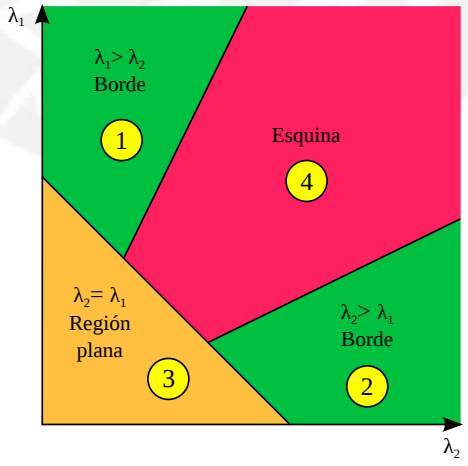


Figura 1.1: Análisis de la CRF respecto a los autovalores de la matriz de autocorrelación [8]. Se distinguen 4 regiones que representan la posible clasificación de un punto en una imagen: regiones 1 y 2 significa que es parte de un borde, la región 4 (los valores propios son ambos altos) significa que es esquina, mientras que la región 1 significa que no es parte de bordes y no es esquina.

Los valores máximos de la métrica CRF son aquellos puntos, según el presente criterio, que pueden ser considerados como esquinas o cercanos a ellas. Debido a ello, Harris propone el uso de umbralización para seleccionar los valores más altos de la métrica, para así, seleccionar esquinas apropiadamente.

1.2.3. Características del algoritmo de detección de Harris

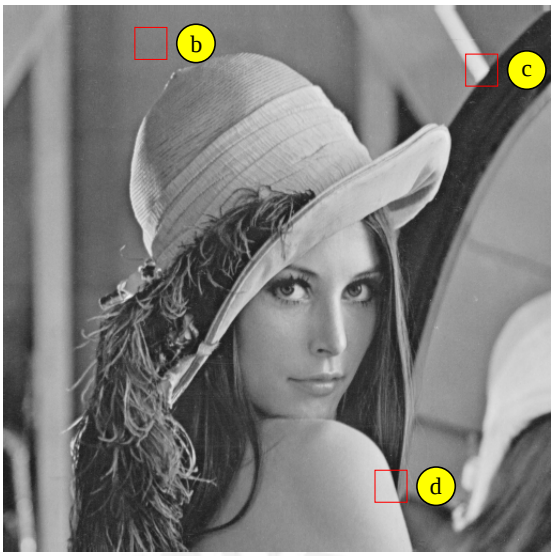
- Propuesto para detección de esquinas y bordes en imágenes en escala de grises.
- Invariante ante rotaciones. Un punto detectado como esquina en una imagen también debe ser detectado como esquina si se aplica el detector a la imagen rotada. Esto se debe al uso del filtro gaussiano, el cual ofrece una respuesta isotrópica debido a que es aproximadamente circular.
- Invariante ante traslaciones. El algoritmo de detección está basado en cambios en intensidad (derivadas direccionales), por lo cual un punto detectado como esquina no debería cambiar su condición ante un desplazamiento o traslación.
- Respuesta dependiente del factor de sensibilidad k. Valores menores de k para detección de bordes, mientras valores mayores de k para lograr una detección más selectiva de esquinas.
- No invariante ante cambios de escala. La vecindad de un punto en una imagen varía ante cambios en la escala, entonces, la respuesta del punto varía acorde con la escala.

1.3. Estado del arte

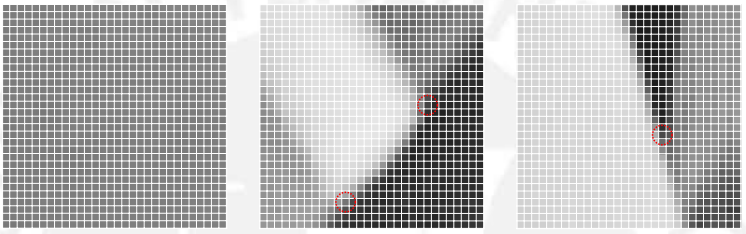
A continuación se exponen algunos métodos presentes en la literatura cuyo impacto se explica tanto por la naturaleza del método empleado como por uso extendido en plataformas de alto rendimiento.

1.3.1. Detector SUSAN

Las esquinas según el detector SUSAN (*Smallest Univalue Segment Assimilating Nucleus*), propuesto en [5], son puntos que deben cumplir con ciertas condiciones. Dado un punto en una imagen, se define una circunferencia centrada en este formada por los píxeles vecinos con intensidad parecida, a esta circunferencia se conoce como USAN y el centro de esta circunferencia se le llama *nucleus*. Entonces, una esquina es aquel punto cuya vecindad forma un USAN de área pequeña, tal y como se puede mostrar en la figura 1.2.



(a)



(b) (c) (d)

Figura 1.2: Ejemplo del criterio de detección de SUSAN. (a) Imagen original: Lena. Se señalan 3 regiones de análisis. (b) No esquinas detectadas (gran cantidad de píxeles con intensidad parecida o región plana). (c) y (d) Dos y una esquinas detectadas respectivamente (circunferencias pequeñas con píxeles de intensidad parecida).

El detector depende de una medida c de comparación entre la intensidad del *nucleus* $\vec{r_0}$ y cada intensidad de los píxeles contenidos $\vec{r}$ por el USAN usando para un umbral t (1.4):

$$c(\vec{r}, \vec{r_0}) = \begin{cases} 1 & \text{si } |I(\vec{r}) - I(\vec{r_0})| \leq t \\ 0 & \text{si } |I(\vec{r}) - I(\vec{r_0})| > t \end{cases} \tag{1.4}$$

Sin embargo, puede ser aproximada según la ecuación 1.5:

$$c(\vec{r}, \vec{r_0}) = e^{-\left(\frac{I(\vec{r}) - I(\vec{r_0})}{t}\right)^6} \tag{1.5}$$

Finalmente, la métrica de esquinas R es calculada la comparación entre un umbral geométrico g y la medida n (1.6), la cual representa la suma de todas las comparaciones c llevadas a cabo para un USAN. Luego, R es usada para la clasificación de puntos como esquinas:

$$R(\vec{r_0}) = \begin{cases} g - n(\vec{r_0}) & \text{si } n(\vec{r_0}) < g \\ 0 & \text{en otro caso} \end{cases} \tag{1.6}$$

1.3.2. LoCoCo

El Detector de esquinas de baja complejidad o Detector LoCoCo (*Low Complexity Corner Detector*) propuesto en [11] es un detector de esquinas que aproxima las operaciones efectuadas por el detector de esquinas de Harris para disminuir el costo computacional. Para ello, usa el concepto de imagen integral ii (ecuación 1.7) de una imagen I:

$$ii(x, y) = \sum_{i \leq x, j \leq y} I(i, j) \tag{1.7}$$

El método plantea el cálculo de las derivadas direccionales g_x y g_y de la imagen g aplicando imágenes integrales, para ello, calcula la imagen integral de g y aplica el filtro *difference box* como aproximación del filtro DoG (*Difference of Gaussians*). Luego, se calcula las componentes del tensor estructural g_x^2, g_y^2 y $g_x g_y$ y las imágenes integrales de cada componente con el objetivo de filtrarlas con un filtro box, el cual es considerado como una aproximación del filtro gaussiano. Finalmente, se calcula la función de métrica tal y como es propuesta por Harris y se ubican los puntos asociados a la mayor intensidad en la métrica. Estos puntos son las esquinas detectadas por este método.

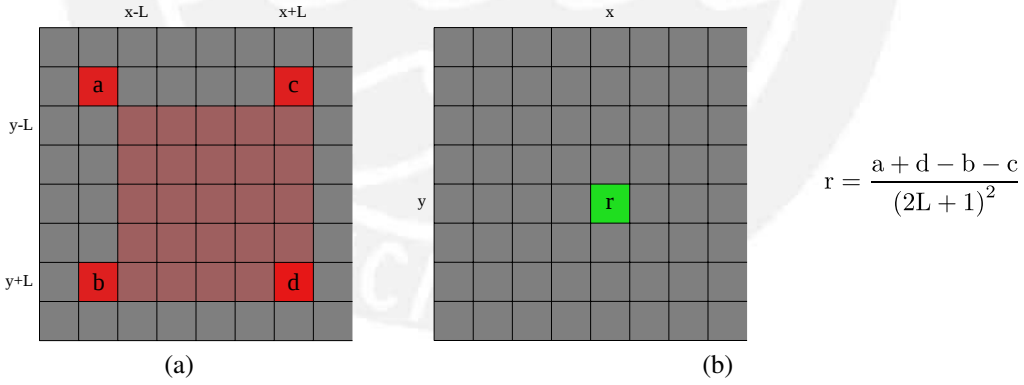


Figura 1.3: Ejemplo de aplicación de un filtro box a una imagen usando imágenes integrales [11]. (a) Imagen original: Se sombrea de rojo oscuro los píxeles que se usarían en caso se usara el algoritmo de filtrado convencional. De rojo se resaltan los píxeles utilizados por el algoritmo usando imágenes integrales. (b) Imagen filtrada: Se resalta en verde el píxel calculado.

Se debe considerar el número de operaciones y accesos a memoria para entender la optimización lograda usando imágenes integrales. El número de accesos a memoria y operaciones cuando se aplica un filtro a una imagen es de M y N accesos a memoria y

operaciones entre sumas y multiplicaciones respectivamente. Esto se reduce a 4 lecturas de memoria y 3 operaciones cuando se usan imágenes integrales considerando la aproximación del filtro gaussiano al filtro *box*, esto se ilustra en la figura 1.3.

1.3.3. Detector Harris-Laplace

El detector Harris-Laplace es un detector de esquinas invariante ante cambios en la escala propuesto en [12] que está basado en en el detector de esquinas de Harris. Para ello propone primero localizar puntos aplicando el método de detección de de esquinas de Harris a diferentes representaciones de una imagen en escala, es decir, diferentes resoluciones. Luego, selecciona aquellos puntos en los cuales se tenga que el LoG (*Laplacian of Gaussian*) represente un máximo a lo largo de la escala.

La primera etapa de este detector está basada en el método de Harris aplicado a la representación *scale-space* de una imagen, es decir, a un conjunto de imágenes a diferente nivel de resolución [13]. Para ello, se propone una adaptación, dentro del esquema del método de Harris, de la matriz de autocorrelación μ ante cambios en la escala para lograr invarianza. Esta adaptación consiste en el uso de dos escalas, una de diferenciación σ_D y una de integración σ_I. Así, las derivadas direccionales L_x y L_y en las direcciones x y y respectivamente son calculadas en función a un filtro gaussiano cuyo tamaño depende de σ_D, y μ se calculará tal y como se muestra en la ecuación 1.8.

$$\mu\left(\mathbf{x}, \sigma_I, \sigma_D\right) = \sigma_D^2 * g\left(\sigma_I\right) \begin{bmatrix} L_x^2\left(\mathbf{x}, \sigma_D\right) & L_x L_y\left(\mathbf{x}, \sigma_D\right) \\ L_x L_y\left(\mathbf{x}, \sigma_D\right) & L_y^2\left(\mathbf{x}, \sigma_D\right) \end{bmatrix} \tag{1.8}$$

Luego se aplica la función de métrica de Harris adaptada a cambios en la escala dada por la ecuación 1.9:

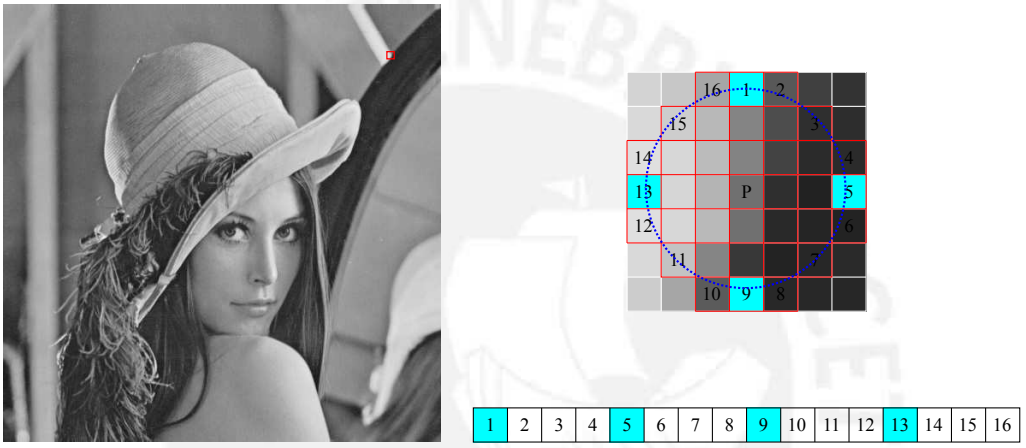
$$\mathbf{C} = det\left(\mu\left(\mathbf{x}, \sigma_I, \sigma_D\right)\right) - \alpha\, tr^2\left(\mu\left(\mathbf{x}, \sigma_I, \sigma_D\right)\right) \tag{1.9}$$

La segunda etapa del detector Harris-Laplace consiste en una selección automática de la escala y para lo cual se buscan aquellos puntos donde una función determinada represente un máximo respecto a la escala. El LoG, expresado en la ecuación (1.10), ya ha demostrado en [14] ser una expresión para la cual, aplicando este criterio, se ha obtenido un alto porcentaje de obtenciones de escalas características, las cuales están relacionadas a estructuras en la imagen. Por ello, los puntos seleccionados son invariantes a cambios en la resolución de la imagen.

$$|LoG\left(\mathbf{x},\sigma_n\right)| = \sigma_n^2|L_{xx}\left(\mathbf{x},\sigma_n\right) + L_{yy}\left(\mathbf{x},\sigma_n\right)| \tag{1.10}$$

1.3.4. Detector FAST

El detector FAST (*Features from Accelerated Segment Test*) propuesto en [6] es un método que evalúa un criterio para determinar si un punto es esquina. Dado un píxel de análisis P en una imagen con intensidad I_P, este punto será clasificado como esquina si al menos n puntos consecutivos de una circunferencia de 16 píxeles centrada en este píxel son más claros que $I_P + t$ o más oscuros que $I_P + t$, donde t es un umbral. El valor de n es usualmente establecido en 12 para detección rápida.



(a) (b)

Figura 1.4: Criterio de segmento para selección de esquinas según FAST. (a) Imagen original Lena. Se señala una región de análisis. (b) Criterio de detección: Se analiza las intensidades de una circunferencia centrada en el punto P. La numeración utlizada por del detector es la señalada en la figura. Para rápida clasificación, se comparan los píxeles 1, 5, 9 y 13.

El criterio, el cual está ilustrado en la figura 1.4, se aplica de la forma siguiente. Primero se evalúan los píxeles 1 y 9, si estos tienen una intensidad alrededor de I_P con una diferencia menor a t, el punto P no puede ser establecido como esquina. En caso cumplan, se evalúan los píxeles 5 y 13 son evaluados de la misma forma. Finalmente, P será clasificado como esquina si el resto de los píxeles restantes cumplen con el criterio de detección establecido.

El criterio señalado tiene deficiencias, entre ellas destacan que el valor de n establecido a 12 no tiene una capacidad de selección suficiente y el rendimiento del detector depende del orden de las "preguntas" que se hagan en la evaluación de un píxel y de la distribución de la imagen. Técnicas de *Machine Learning* se utilizan para solucionar algunas de estas deficiencias. Así, se propone primero particionar los puntos de la circunferencia de 16 píxeles en 3 grupos, *brighter* que agrupa los puntos con intensidad mayor a $I_p + t$, *darker* que agrupa aquellos con intensidad

menor a $I_p - t$ y *similar*, que agrupa los puntos con intensidad entre $I_p - t$ y $I_p + t$. Entonces, el criterio del detector FAST puede "aprenderse" como un árbol de decisión. Luego, para obtener una respuesta más adecuada, se pueden aplicar técnicas de *Non Maximum Suppression* considerando que la métrica de esquinas de FAST para cada punto en la imagen viene a estar dada por el valor máximo del umbral t para el cual el punto es clasificado como esquina.

1.4. Comparación del detector de esquinas de Harris con otros métodos

Los detectores presentados son todos altamente paralelizables, es decir, constan de operaciones cuyos resultados son independientes unos de otros para el conjunto de datos de análisis. El detector SUSAN tiene la desventaja de tener menos robustez ante ruido y tener un factor menor de repetibilidad de esquinas en análisis de vídeo por cuadros [15]. El detector Harris Laplace presenta una mayor robustez frente a ruido, un mayor factor de repetibilidad frente transformaciones *affine* en imágenes que se pueden presentar en vídeos, sin embargo, tiene un costo computacional demasiado alto comparado con el método de Harris. El detector FAST no es robusto frente al ruido [6], el detector LoCoCo es una aproximación del método de detección de Harris que disminuye el costo computacional a costa de tener resultados menos exactos. Por otro lado, el detector de Harris tiene mejores resultados ante cambios en escala e iluminación que otros detectores no mencionados como los detectores de Forstner [16], Horaud [17], Cottier [18] y Heitger [19] tal y como se muestra en [20]. Debido a que el detector de esquinas de Harris presenta cierta robustez frente al ruido, y es invariante frente a tranformaciones de rotación y cambios de escala e iluminación en imágenes, el Detector de esquinas de Harris representa la alternativa más conveniente de implementación en plataformas de alto rendimiento frente a los otros detectores ya mencionados.

Capítulo 2

Plataforma Jetson TK1

2.1. Introducción

Varias plataformas de alto rendimiento y bajo consumo energético han sido usadas para implementar aplicaciones costosas computacionalmente, algunas de ellas pertenecientes al área de Procesamiento de imágenes y video, de forma eficiente. Entre las opciones disponibles se encuentran varios procesadores integrados con unidades SIMD, que permiten realizar operaciones de forma vectorial, FPGAs que permiten realizar implementaciones tanto discretas, con alto costo en hardware, como soluciones que usan la técnica de codiseño hardware-software usando coprocesadores y un *soft processor* como Microblaze.

La plataforma Jetson TK1 creada por Nvidia está basada en el SoC (*System on chip*) Tegra K1 que integra un procesador ARM Cortex A15 de cuatro núcleos y un GPU de alto rendimiento de 192 núcleos. Esta destaca sobre otras plataformas basadas tanto en procesadores como en FPGAs debido a que representa un balance entre costo en el mercado, rendimiento y consumo energético, así, Tegra K1 ha demostrado ser más eficiente que el procesador Apple A7, procesador en el que está basado el Iphone 5S [21], además, la plataforma Jetson TK1 es mucho más barata que varias plataformas basadas en FPGAs de alta gama [22]. Por estos motivos, esta plataforma constituye la alternativa más idónea para la implementación de la presente tesis.

2.2. Capacidades de la plataforma

La plataforma Jetson TK1, basada en Tegra K1, ofrece más de una alternativa para implementar eficientemente un método de computacionalmente costoso. Así, Tegra K1 posee una unidad SIMD, la cual permite implementar métodos altamente paralelizables de modo vectorial, y un GPU de 192 núcleos, el cual usa tecnología de Nvidia comparable a GPUs de mayores recursos dirigidos a PCs.

2.2.1. NEON

NEON es la arquitectura de la tecnología SIMD de los procesadores ARM [23]. Esta tecnología se encuentra disponible para los microcontroladores y procesadores ARM a partir de la versión ARMv5 y permite el control de una unidad SIMD de 128 bits. Esta unidad SIMD dispone de 32 registros de 64 bits llamados desde D0 a D31 o 16 registros de 128 bits etiquetados desde Q0 a Q15 con los cuales se realizan las operaciones vectoriales y se cargan desde la memoria. El uso de esta unidad SIMD de 128 bits permitiría un factor de aceleración teórico de hasta 4 para operaciones en punto flotante de precisión simple y de 2 para operaciones en punto flotante de precisión doble.

2.2.2. CUDA

CUDA es un modelo de de programación creado por Nvidia en 2007 para GPGPU [7], programación de propósito general en Unidades de procesamiento gráfico (GPU). Este modelo de programación permite programar GPU usando extensiones para lenguajes de programación como C, C++, Python, Fortran, entre otros sin tener necesidad de comprender abstracciones referidas a programación gráfica como las utilizadas en DirectX y OpenGL [24]. Debido a que CUDA permite una abstracción apropiada de las arquitecturas del GPU y del CPU y memorias, y el GPU posee una arquitectura adecuada para realizar operaciones intensivas para algorimos altamente paralelizables este modelo es adecuado para la implementación del detector de Harris.

Nvidia, a partir de finales de 2006, ha lanzado varios modelos de GPU y tarjetas gráficas con compatibilidad para CUDA [7]. Cada uno de estos GPU puede ser identificado por un índice llamado capacidad computacional CUDA (*compute capability*), el cual es un indicador de las carácterísticas y funciones que la plataforma permite realizar. En el caso de la plataforma Jetson TK1 el SoC Nvidia Tegra K1 posee *compute capability* 3.2, el cual es alto en comparación con otros GPUs de Nvidia dirigidos a PC.

El uso de CUDA para programación de GPUs ha reportado grandes factores de aceleración, por ejemplo, en [25], donde se presenta un método para multiplicación matriz vector que es usado para resolver sistemas lineales con un factor de aceleración de hasta 30 o en [26] donde se paraleliza el método de *Orthogonal Matching Pursuit* hasta con un factor de aceleración 45 aproximadamente. Debido a ello, implementar un método paralelo en CUDA sería mucho más conveniente que implementarlo usando NEON.

2.3. Arquitectura de un GPU

A continuación, se exponen elementos básicos sobre la arquitectura de un GPU relacionados con CUDA.

2.3.1. Visión general de la arquitectura

Un GPU, es un procesador que consta de varios *Streaming processors*, los cuales poseen a su vez, varios CUDA *cores* o núcleos [27]. Estos núcleos son procesadores de capacidad limitada que ejecutan *threads*, que constan de operaciones aritméticas y lógicas, en paralelo. Estos *threads* son ejecutados de acuerdo al modelo SIMT (*Single Instruction Multiple Thread*), el cual es análogo al modelo SIMD con la diferencia que cada núcleo puede realizar tareas diferentes (*warp divergence*) con penalización en rendimiento. Según el modelo SIMT, los *threads* pueden ser agrupados en *warps* tal que los *threads* pertenecientes a un mismo *warp* ejecuten la misma instrucción con excepción de aquellos en los que explícitamente se encargue tareas diferentes. Realizar esto incluye una penalización en rendimiento debido a que para lograrlo, el *warp scheduler*, unidad encargada de la planificación de las tareas, debe instanciar más de una instrucción por *warp*. El tamaño de un *warp* es generalmente 32 pero es dependiente del *compute capability*.

2.3.2. Jerarquía de memorias

Los GPUs de Nvidia que pueden utilizar CUDA poseen diferentes tipos de memoria que se diferencian en cuanto a visibilidad, latencia y tamaño. Se requiere un adecuado manejo y comprensión del sistema de memorias de un GPU mostrado en la figura 2.1 para implementar eficientemente un algoritmo en CUDA. Entre las diferentes memorias que un GPU puede tener destacan la memoria global, constante y textura, accesibles desde desde el CPU y el GPU, y las memorias compartida, local y los registros que solo son visibles para los *threads* en el GPU.

La memoria global es una memoria visible para los *threads* y el CPU. Esta memoria es usada para almacenar datos que van a ser transferidos en una operación entre el CPU y el GPU y se caracteriza por su alta latencia, gran tamaño y porque los datos almacenados en esta se encuentran en el dispositivo mientras dure la aplicación que la use. Esta memoria posee una línea de caché de cierta longitud que optimiza localidad espacial, es decir, los accesos a memoria global son óptimos si se realizan accesos colaborativos (*coalesced acceses*), es decir, en espacios contiguos de memoria. Las memorias textura y constante son memorias de solo lectura que pueden ser accesadas desde el CPU, se caracterizan por su baja latencia debido a que se encuentran en caché y tamaño

limitado. En particular, la memoria textura se distingue por tener accesos optimizados cuando se quiera indexar un dato usando dos o tres coordenadas.

La memoria local es una memoria de tamaño limitado y alta latencia privativa para un *thread*, es decir, se define un espacio de memoria local por cada *thread* y visible solo para este. La memoria compartida es una memoria privativa para un grupo de *threads* llamado bloque (*block*), así, se tendrá un espacio de memoria compartida por cada bloque de *threads*. Se caracteriza por su tamaño pequeño, ser configurable por el programador y su baja latencia. Los registros son un espacio de memoria privativo por *thread* y se caracteriza por ser bastante pequeño y por tener la latencia más baja en toda la jerarquía. Este conjunto de memorias se distingue porque los datos almacenados en ellas existen solo durante la ejecución de la tarea en el GPU que haya instanciado el uso de estos recursos.

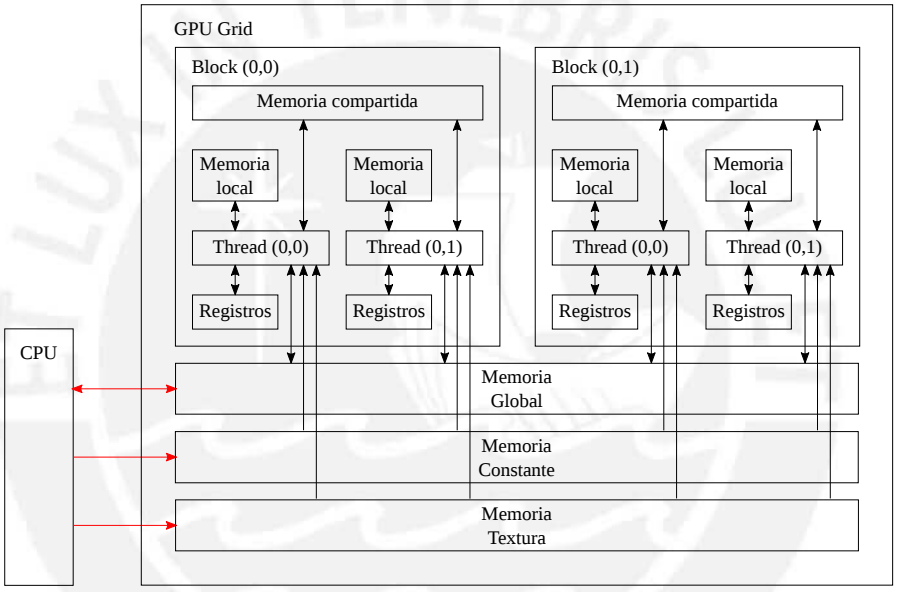


Figura 2.1: Jerarquía de memorias en un GPU [7]. Las puntas de flechas indican qué tipo de accesos se encuentran disponibles.

2.4. Modelo de programación CUDA

CUDA considera dos actores principales, el *host* (CPU) y el *device* (GPU), los cuales no comparten un espacio único de memoria y están comunicados a través de un bus PCIE (*PCI Express*) [7]. Las abstracciones de CUDA permiten tener control de la configuración de tareas que se ejecutan en los CUDA *cores*, de transferencias de memoria, entre otras.

2.4.1. Interacción entre el GPU y el CPU

El *device* es considerado según el modelo como un coprocesador del *host* especializado en operaciones con alto potencial de paralelismo. Así, para procesar datos usando el *device*, se deben realizar transferencias desde el *host* al *device*, y para recuperar los resultados se deben efectuar transferencias en sentido contrario. Esto se puede apreciar en el contexto de programación CUDA C donde se puede apreciar el flujo de sentencias en C para realizar procesamiento en el GPU. Sin embargo, a partir de la versión 6 de CUDA [28], Nvidia ha desarrollado el modelo de memoria unificada donde las transferencias entre CPU y GPU son "invisibles" para el programador disminuyendo la complejidad del código y alterando el rendimiento computacional, en cuanto a transferencias de memoria. Este modelo de memoria unificada es soportado por dispositivos de Nvidia con *compute capability* mayor a 3.2. El cambio de perpectiva del programador respecto al modelo de memoria usado se muestra en el esquema gráfico 2.2.

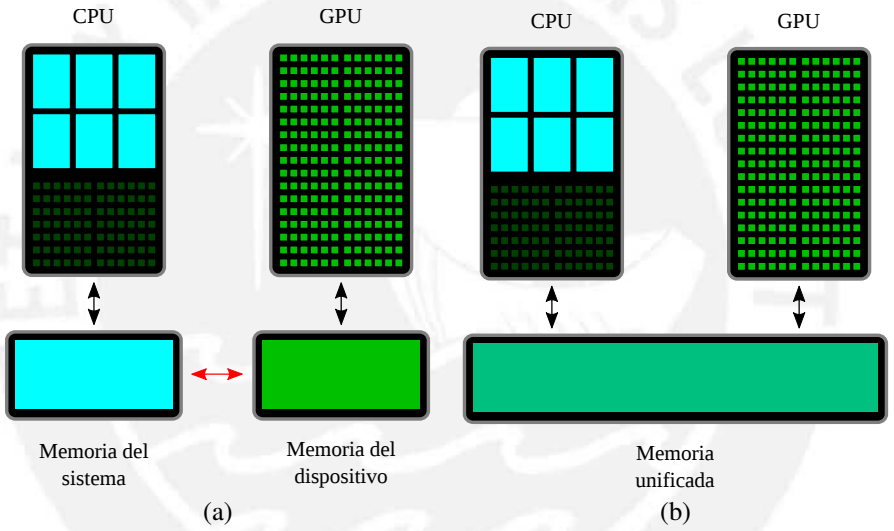


Figura 2.2: Perspectiva del programador [28] en: (a) Modelo de memoria estándar. (b) Modelo de memoria unificada.

2.4.2. Jerarquía de *threads*

Una aplicación en CUDA está compuesta por uno o más *kernels*, que son las funciones que un GPU ejecuta y definen operaciones que los *threads* ejecutan en modo paralelo. Cuando el programador instancia un *kernel* en su programa, es importante fijar la configuración de los *threads* que van a ejecutarse. En tal sentido se ha definido una jerarquía de *threads* expresada mediante una geometría que permite al programador realizar una configuración adecuada a un *kernel* determinado. Entre los elementos que definen esta geometría se encuentran los *threads*, *blocks* (bloques) y el *grid* (grilla).

Un *thread* es un elemento de programación que está asociado a un CUDA *core*, el cual se ejecuta de forma paralela por *warps* según el modelo SIMT ya explicado. Un bloque es un conjunto de *threads* asociados a un mismo *Streaming processor* que comparten un mismo espacio de memoria compartida y en donde se pueden realizar operaciones especiales (según el *compute capability* del dispositivo estudiado). Finalmente, una grilla es el conjunto de todos los bloques instanciados para un *kernel*. Tanto los bloques como las grillas pueden tener una geometría de hasta tres dimensiones y tienen un tamaño máximo según el dispositivo usado. Esta jerarquía de *threads* es mostrada en la figura 2.3

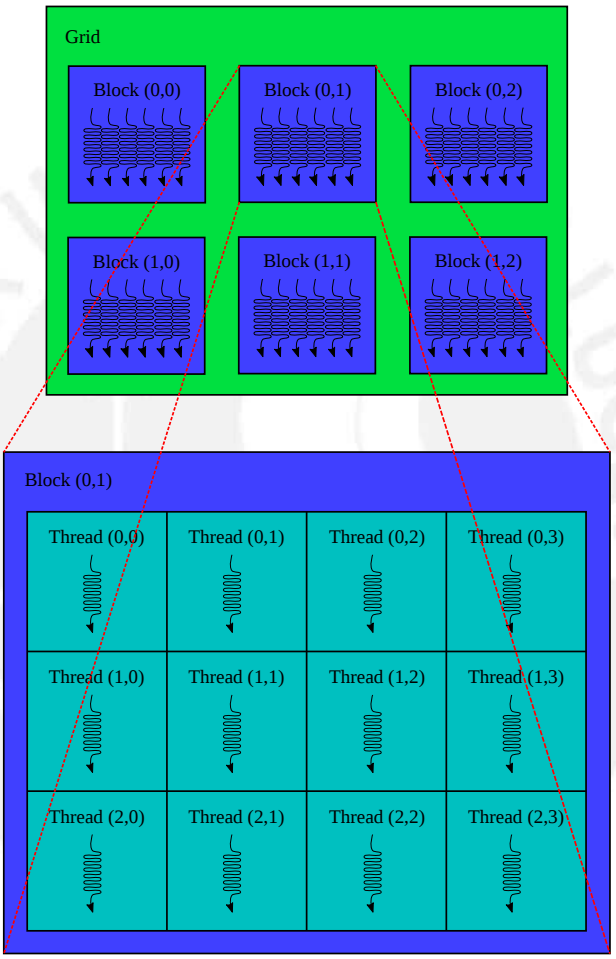


Figura 2.3: Jerarquía de *threads* en el modelo de programación CUDA [7]. Se asume una configuración geométrica en dos dimensiones para este ejemplo gráfico.

2.5. Detalles técnicos de Tegra K1

Este es un SoC basado en los procesadores ARM Cortex A15 y un GPU de Nvidia [21]. Este procesador gráfico posee un *compute capability* 3.2, lo cual lo identifica como un procesador basado en la arquitectura Kepler. Posee además una un sistema de memorias caché de niveles L1

y L2, de las cuales el nivel L2 es de solo lectura. Cabe mencionar también, que este procesador cuenta con una unidad *warp scheduler* que permite instanciar 2 instrucciones por cada 4 *warps*.

La tecnología Kepler optimiza accesos a memoria si se cumplen una serie de condiciones [29] que permiten tener mayor rendimiento que con su antecesor, la arquitectura Fermi. Uno de los más importantes consiste en que los accesos a memoria global se hacen usando una pequeña memoria caché que carga segmentos de 128 bytes alineados, es decir, deben tener direcciones de inicio múltiplo de 64. Debido a ello, todas las peticiones de acceso a memoria global en la tecnología Kepler se hacen del tal forma que cargan la caché 128 bytes de datos de la memoria global. Por otro lado, la memoria compartida en arquitectura Kepler tiene 2 modos de acceso, uno de 32 bytes y otro de 64 bytes. Ambos se modelan como 32 bancos de memoria, los cuales deben ser accesados de forma alineada para su uso óptimo, y, además, permiten accesos de tipo a paso (*strided acceses*) y de tipo *scatter* y *gather* sin generar conflictos de banco (serialización de accesos a memoria). En la tabla 2.1 se especifican más detalles técnicos de Tegra K1 que pueden ser encontrados ejecutando el programa *deviceQuery* que se provee en los Nvidia CUDA *samples*.

Nombre	GK20A
Capacidad computacional	3.2
Memoria global	1894 MB
Número de multiprocesadores	1
Número de núcleos	192
Frecuencia de operación	852 MHz
Frecuencia de operación de memoria	924 MHz
Ancho de bus	64 bits
Caché L2	131072 B
Memoria constante	65536 B
Memoria compartida por bloque	49152 B
Registros por bloque	32768 B
Tamaño de warp	32
Máximo número de threads por Multiprocesador	2048
Máximo número de threads por bloque	1024
Dimensiones máximas de bloque	1024 x 1024 x 64
Dimensiones máximas de grid	2147483647 x 65535 x 65535

Tabla 2.1: Detalles técnicos de Tegra K1.

2.6. Objetivos

El principal objetivo de este trabajo consiste en el diseño e implementación del algoritmo paralelo del detector de esquinas de Harris en la plataforma Jetson TK1 usando el modelo de programación CUDA. Este resulta ser el más conveniente en esta plataforma tomando en cuenta que puede alcanzar un rendimiento mayor a la unidad SIMD del SoC en que está basada. Para ello e implementará el algoritmo en MATLAB con el objeto de conocer la precisión de la solución propuesta. Luego se implementará una versión del método en ANSI-C con el motivo de realizar comparaciones y establecer métricas de rendimiento del método paralelo a implementar.

Capítulo 3

Diseño del algoritmo propuesto

3.1. Revisión del algoritmo de Harris

El Detector de esquinas de Harris propone el cálculo de una métrica de esquinas basada en la matriz de autocorrelación y una umbralización. A continuación se presenta el método propuesto por Harris en [8]:

Algoritmo 1 Detector de esquinas de Harris

Entrada:

Imagen **I** en escala de grises,
Filtro gaussiano **G**,
Factor **k** de sensibilidad,
Umbral **th**.

Resultado:

Lista de esquinas **Co**.

Etapa 1: Cálculo de la métrica de esquinas

1: $I_x \leftarrow \frac{d}{dx} I$
2: $I_y \leftarrow \frac{d}{dy} I$
3: $M \leftarrow G * \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$
4: $CRF \leftarrow \det(M) - k\ tr(M)^2$

Etapa 2: Umbralización y generación de la lista de esquinas

5: $Co \leftarrow \{(x,y) \ / \ CRF(x,y) > th\}$

La precisión del algoritmo de Harris es altamente dependiente del umbral seleccionado, así, no es posible establecer un umbral adecuado para tener una correcta detección de esquinas para varios tipos de imágenes. Además, el algoritmo anterior solo permite trabajar con imágenes a escala de grises.

3.2. Descripción y análisis del método propuesto

Debido a las limitaciones mencionadas, se propone modificar el detector de Harris diseñando etapas que permitan realizar detección adecuada de esquinas en imágenes de escala de grises y en color. El método propuesto consta de cuatro etapas y se muestra a continuación:

Algoritmo 2 Método propuesto

Entrada:

Imagen **I** en escala de grises o color,
Filtro gaussiano **G**,
Factor **k** de sensibilidad,
Tamaño **L** de ventana,
Número **nCo** de esquinas.

Resultado:

Lista de esquinas **Co**.

Etapa 1: Cálculo de la métrica de esquinas

1: **if** Imagen I está en escala de grises **then**
2: $N \leftarrow 1$
3: **else**
4: $N \leftarrow 3$
5: **end if**
6: **for** k=1 to N **do**
7: $I_{x,k} \leftarrow \frac{d}{dx} I_k$
8: $I_{y,k} \leftarrow \frac{d}{dy} I_k$
9: **end for**
10: $A \leftarrow \sum_{k=1}^{N} I_{x,k}^2$
11: $B \leftarrow \sum_{k=1}^{N} I_{y,k}^2$
12: $C \leftarrow \sum_{k=1}^{N} I_{x,k} I_{y,k}$
13: $M \leftarrow G * \begin{bmatrix} A & C \\ C & B \end{bmatrix}$
14: $CRF \leftarrow \det(M) - k\ \mathrm{tr}(M)^2$

Etapa 2: Cálculo automático del umbral

15: $h \leftarrow \text{histograma}\,(CRF)$
16: $th \leftarrow \text{umbralRosin}\,(h)$

Etapa 3: Supresión del no máximo

17: $Mp \leftarrow \text{MáximoPorBloque}\,(M\,(M > th))$
18: $Mm \leftarrow \text{MáximosLocales}\,(Mp)$

Etapa 4: Selección de la lista de esquinas

19: $Ci \leftarrow \{(x,y)\ /\ Mm(x,y) > 0\}$
20: $Ci \leftarrow \text{ordenamientoDescendente}\,(M(Ci), Ci)$
21: $Co \leftarrow \text{selección}\,(Ci, nCo)$

Las cuatro etapas del método son: Cálculo de la métrica de esquinas, Selección automática del umbral, Supresión del no máximo y Selección de la lista de esquinas. En el método propuesto se

extiende el cálculo de la métrica de esquinas usando el método presentado en [9], donde se usan diferenciales invariantes para calcular un tensor estructural aplicable a imágenes a color. Por otro lado, la métrica de esquinas obtenida usando la función de Harris tiene la particularidad de tener, generalmente, distribución unimodal. Por ello, se propone usar una etapa de Selección de umbral automático basada en la técnica de umbralización unimodal propuesta en [30]. Adicionalmente, se propone usar una etapa de Supresión del no máximo (NMS) para eliminar los falsos positivos generados alrededor de las esquinas detectadas en la función de métrica. El método de NMS usado está basado en el método presentado en [31] y comprende el cálculo de máximos por bloque en toda la matriz y luego de máximos locales alrededor de ventanas centradas en los puntos máximos por bloque. Finalmente, una etapa de Selección de lista de esquinas es propuesta para tener control sobre el número de puntos que se necesitan extraer. Esta última etapa se encuentra basada en la generación de una lista de esquinas representada por dos listas, una de coordenadas y una de valores, y un algoritmo de ordenamiento, el cual debe ordenar en orden descendente la lista de valores y reproducir las permutaciones realizadas en esa lista en la lista de coordenadas, luego, los N valores máximos seleccionados deben corresponder a las N mejores esquinas detectadas.

3.2.1. Cálculo de la métrica de esquinas

Consiste en el cálculo de la matriz de autocorrelación y en la aplicación de la función de métrica propuesta por Harris en [8]. La extensión del cálculo del tensor estructural propuesto en [9] consiste en la descomposición de la imagen en capas de dos dimensiones, si la imagen se encuentra en escala de grises, constará de una capa, y, si la imagen es a color, constará de tres. Así, se calculan las derivadas direccionales de cada una de las N capas de la imagen, las cuales son aproximadas usando diferencias finitas a una convolución con un filtro que calcula la diferencia entre dos componentes de los datos:

$$I_{x,k} = I_k * [1, 0, -1] \qquad I_{y,k} = I_k * [1, 0, -1]^T$$

Los componentes de la matriz de autocorrelación presentada en el primer capítulo se calculan a partir de las derivadas direccionales usando operaciones punto a punto (multiplicaciones y sumas). Luego, se procede a calcular la convolución de cada componente del tensor estructural con un filtro gaussiano:

$$A = h * \sum_{k=1}^{N} I_{x,k}^2 \qquad B = h * \sum_{k=1}^{N} I_{x,k} I_{y,k} \qquad C = h * \sum_{k=1}^{N} I_{y,k}^2$$

Finalmente, la métrica de esquinas puede ser calculada con operaciones punto a punto a partir

de la fórmula planteada por Harris:

$$CRF = AC - B^2 - k(A + B)^2$$

Las operaciones involucradas en esta etapa son principalmente operaciones punto a punto y convoluciones, las cuales involucran operaciones aritméticas intensivas y producen resultados independientes por píxel, por ello esta etapa es altamente paralelizable. Respecto al cálculo de convolución, se puede notar que el filtro gaussiano es un filtro separable, lo cual es conveniente debido a que el algoritmo de cálculo de convolución separable involucra menos operaciones por píxel que el algoritmo de convolución no separable, tal y como se muestra en el esquema gráfico 3.1 y la tabla 3.1.

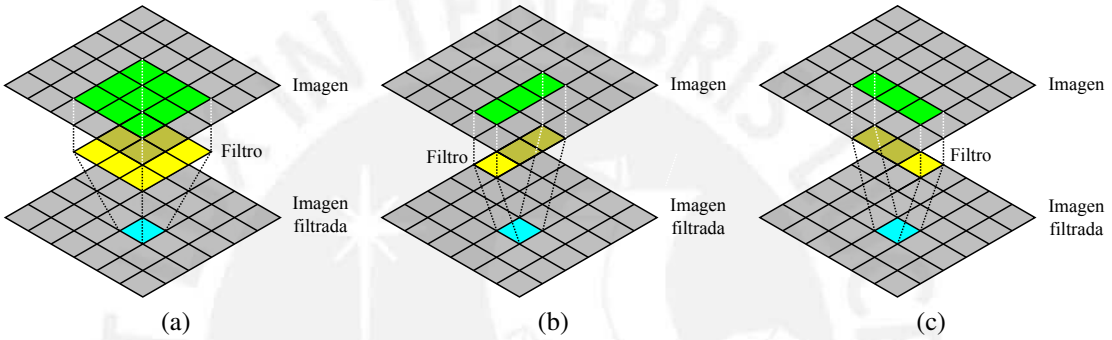


Figura 3.1: (a) Esquema gráfico de convolución no separable para un filtro de tamaño 3 x 3. (b) y (c) Esquema gráfico de convolución separable. Derecha: Convolución con filtro fila 3 x 1. Izquierda: Convolución con filtro columna 1 x 3.

Algoritmo		**Sumas**	**Multiplicaciones**	**Número total de operaciones por píxel**
Convolución no separable		MN-1	MN	2MN-1
Convolución separable	Filtro fila	M-1	M	2(M+N-1)
	Filtro columna	N-1	N	

Tabla 3.1: Número de operaciones por píxel para los algoritmos de convolución separable y no separable para un filtro de tamaño MxN. Se muestra que el algoritmo de convolución separable es más eficiente que el algoritmo de convolución no separable mientras el filtro tenga mayor tamaño.

3.2.2. Selección automática del umbral

En esta etapa se propone usar el método planteado en [30], el cual requiere el cálculo del histograma de una matriz. El histograma es una acumulación frecuencial de los valores presentes en una matriz, su cálculo involucra recorrer la matriz y leer cada elemento para luego aumentar un contador asociado al bin donde se ubica el valor del elemento leído.

El histograma de la métrica de esquinas de Harris, es bastante particular, puesto que generalmente tiene distribución unimodal, lo cual permite el uso del siguiente método. El método

de umbralización de Rosin está planteado para datos con distribución unimodal y utiliza el histograma para el cálculo del umbral. El umbral según Rosin viene a estar dado por el punto del histograma que tiene la mayor distancia a la recta que pasa por el punto más alto y el último punto no nulo del histograma. Este proceso se muestra gráficamente en la figura 3.2.

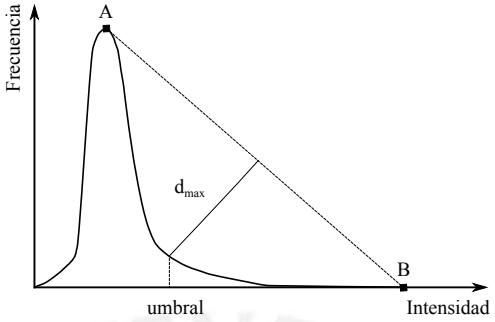


Figura 3.2: Cálculo del umbral según el método de Rosin: La recta que une los puntos A y B, puntos máximo y el último no nulo respectivamente, tiene una distancia máxima d_{max} con el histograma que corresponde al umbral buscado.

El cálculo del histograma es una operación, que hecha de manera secuencial no presenta variaciones significativas en desempeño, sin embargo, su implementación en paralelo está limitada al ancho de bus y será altamente dependiente de la distribución de la métrica. Además, operaciones de tipo reducción a gran escala como la búsqueda del mínimo y máximo de la métrica están involucradas para asegurar la pertenencia del valor de un punto de la matriz de métrica a un bin determinado, estas operaciones son paralelizables. El método de Rosin para el cálculo del umbral, sin embargo, involucra operaciones punto a punto como el cálculo de la distancia de un punto a una recta, y operaciones de tipo reducción como la búsqueda del mínimo y el máximo, las cuales son paralelizables. Sin embargo, este método opera sobre un arreglo pequeño de datos, el histograma, por lo cual la paralelización de la parte final de esta etapa no tiene un efecto significativo en el rendimiento general del método propuesto.

3.2.3. Supresión del no máximo

Esta etapa, basada en [31], tiene la utilidad de eliminar los falsos positivos que quedan luego de la umbralización debido a que los valores en la función de métrica se incrementan en dirección a los valores máximos locales, que vedrían a ser idealmente las esquinas detectadas, y que no necesariamente pueden ser eliminados mediante umbralización. Esta etapa consta de dos partes: Cálculo de máximos por bloque y Cálculo de máximos locales. El Cálculo de máximos por bloque se realiza separando la matriz en bloques de tamaño L x L y calculando el máximo valor de cada bloque. Los puntos con valores menores al máximo por bloque son descartados, si el valor máximo

del bloque es cero, entonces, no hay máximo por bloque. Se tendrá entonces al menos un punto por bloque, los cuales serán analizados en la parte de búsqueda de máximos locales. El Cálculo de máximos locales consiste en la búsqueda de los máximos en ventanas de tamaño $2L$+1 x $2L$+1 centradas en puntos con valor distinto a cero.

El Cálculo del máximo por bloque es una operación donde el resultado de un bloque es independiente de otro, por lo tanto es paralelizable. Por otro lado, el cálculo de máximos locales también es paralelizable debido a que las búsquedas de máximo por ventana son independientes entre sí. Se muestran un ejemplo numérico de los procesos de Cálculo de máximo por bloque y el cálculo de máximos locales en las figuras 3.3 y 3.4.

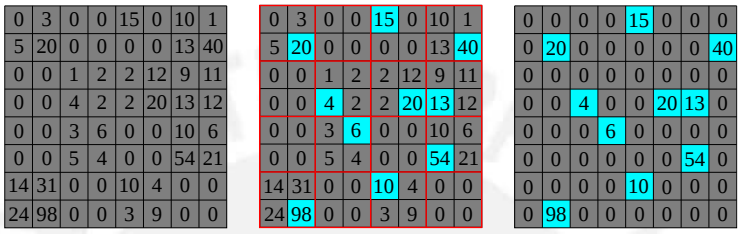


Figura 3.3: Ejemplo numérico del proceso de búsqueda del máximo por bloque. (a) Matriz de entrada. (b) Proceso de búsqueda para máximo por bloque para un bloque de tamaño 2 x 2. (c) Matriz resultado.

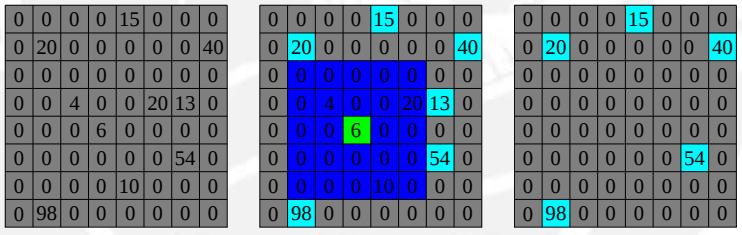


Figura 3.4: Ejemplo numérico del proceso de búsqueda del máximo por vecindades. (a) Matriz de entrada. (b) Proceso de búsqueda para máximo por vecindades. Se muestra la búsqueda del máximo en la vecindad resaltada de azul centrada en el elemento resaltado de verde. (c) Matriz resultado.

3.2.4. Selección de la lista de esquinas

La etapa de selección de la lista de esquinas consta de tres partes: Generación de lista de esquinas, Ordenamiento descendiente por valor y Selección. La Generación de lista de esquinas es una operación que consiste en identificar todos los puntos con valores no nulos y hacer una lista de valores de los puntos y una lista de las coordenadas asociadas a esos puntos. El Ordenamiento descendiente por valor tiene como objetivo ordenar la lista de valores de los puntos y reproducir las permutaciones hechas en el ordenamiento en la lista de coordenadas de tal forma

que la lista comience con las mejores esquinas, aquellas que tengan los valores asociados más altos. Finalmente, la Selección es la creación de una lista con los N primeros datos de las listas anteriores, es decir, las N mejores esquinas, las cuales serían el resultado de aplicar el método propuesto.

La Generación de lista de esquinas es una operación difícil de paralelizar debido a que no se tiene un patrón de los puntos depurados, sino que estos tienen una distribución aleatoria. Por otra parte, existen varios algoritmos de ordenamiento, uno de los más conocidos es el ordenamiento Burbuja *Bubblesort*, el cual resulta ineficiente. En contraste, uno de los algoritmos más eficientes es el ordenamiento rápido *Quicksort*, propuesto en [32], que tiene una baja complejidad computacional, sin embargo, su implementación en paralelo sería complicada debido a que implica el cálculo de pivotes y una alta inestabilidad en cuanto al número y al patrón de accesos a memoria. Por ello, se plantea usar redes de ordenamiento, algoritmos de ordenamiento cuya secuencia de comparaciones no es dependiente de la distribución de datos a ordenar [33], debido a que los accesos a memoria que requieren siguen un patrón establecido dado por esa secuencia. En particular, se propone usar la red de ordenamiento bitónico *Bitonicsort*, propuesto en [34], la cual tiene un bajo costo computacional y puede ser eficientemente implementada en paralelo. Finalmente, la Selección es una etapa paralelizable y equivalente a una copia en memoria (*memcpy*). A continuación, se muestra en la figura 3.5 un elemento de una red de ordenamiento y una red de ordenamiento bitónica de 8 elementos.

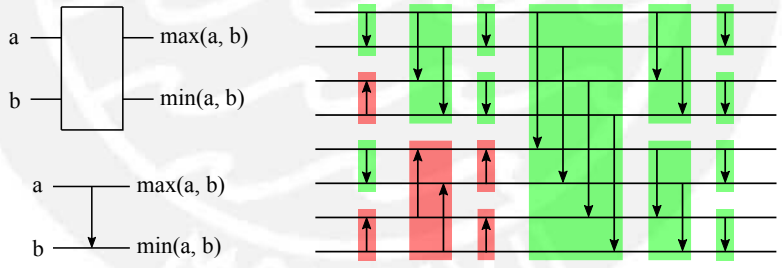


Figura 3.5: (a) Representaciones del elemento de comparación en una red de ordenamiento. (b) Red de ordenamiento bitónico para 8 elementos. Se resaltan de un mismo color aquellos elementos de ordenamiento que ordenen en la misma dirección.

3.3. Consideraciones de diseño del algoritmo paralelo

- La memoria global es la memoria de latencia más alta, tiempo de acceso más largo, por lo cual la minimización de accesos a esta es una prioridad en el diseño. La minimización de accesos a memoria global se realizará asignando tareas y operaciones a cada *kernel* de forma conveniente, minimizando el número de *kernels* a usar para la paralelización del método y

utilizando la memoria compartida en caso se necesite compartir datos.

- Balance entre paralelismo a nivel de *threads* (TLP) y paralelismo a nivel de instrucciones (ILP). En [35] se estudia el diseño de *kernels* explotando ILP, asignando a un *thread* el cálculo de más de un dato en el resultado del *kernel* en comparación con el diseño basado en TLP, donde se propone lanzar *kernels* con la mayor cantidad de *threads* posible. El resultado indica que el diseño basado en ILP es más eficiente que el basado en TLP, por lo cual, en el diseño del algoritmo paralelo, se buscará un balance entre ambos.

- Diseño de *kernels* minimizando divergencia de *warps*, aprovechamiento máximo de la linea de caché de accesos a memoria global (colaborativos). Divergencia es el problema que ocurre cuando los *threads* de un mismo *warp* ejecutan instrucciones diferentes. El *warp* que presente divergencia es más lento. La presencia de divergencia en el diseño es, generalmente, no deseada pero inevitable, sin embargo, se debe tener en cuenta su minimización para no afectar el rendimiento.

- Diseño del algoritmo debe priorizar el uso de memorias con menos latencia, en ese sentido, se tendrá la prioridad siguiente: registros, memoria compartida, memoria global. La prioridad viene dada por la jerarquía de memorias de CUDA y se usará memorias de mayor latencia solo en casos en donde se necesite compartir datos. Así se minimizan las latencias y se maximiza el rendimiento.

- Consideración de las capacidades de la plataforma, *compute capability* 3.2 tiene ciertas características que se debe tomar en cuenta para el diseño del algoritmo paralelo. Asimismo, se debe tener en cuenta que Tegra K1 tiene un número limitado de núcleos, 192 CUDA *cores*, agrupados en un *Streaming Multiprocessor*.

3.4. Diseño del método paralelo en CUDA

Se detalla a continuación el diseño del algoritmo en CUDA por cada etapa.

3.4.1. Cálculo de la métrica de esquinas

La presente etapa se desarrolla usando dos *kernels*. El primero calcula el tensor estructural y realiza la convolución en torno a columnas de cada componente usando un filtro, este ha sido diseñado para funcionar con bloques de tamaño 32 x 3 x 1. Los dos primeros *warps* leen datos de la memoria global correspondientes a la imagen de manera que calculan las derivadas direccionales

de la imagen y almacenan en memoria compartida los datos correspondientes a los componentes del tensor estructural considerando que se van a necesitar datos adicionales de este para realizar la convolución. El último *warp* lee los datos correspondientes a los coeficientes del filtro y los almacena en memoria compartida. Luego, cada *warp* realiza la operación de convolución y escribe en memoria global una componente del tensor estructural. Este conjunto de tareas se repite cuatro veces, de manera que cada *thread* calcule cuatro elementos por componente del tensor estructural. Esto se puede visualizar en la figura 3.6.

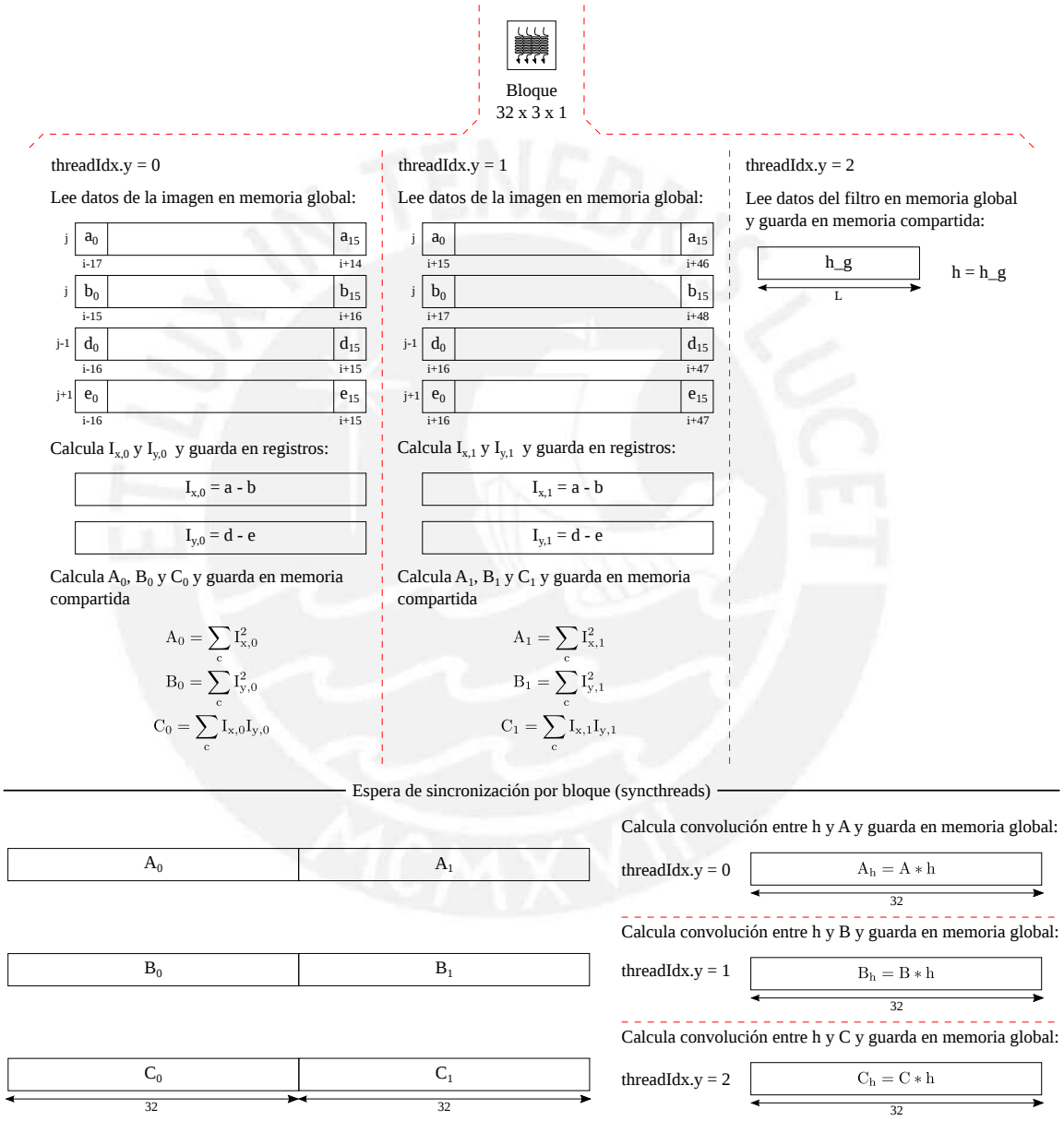


Figura 3.6: Idea básica del cálculo del tensor estructural y convolución con filtro fila usando CUDA para un *thread* que calcula 1 elemento por componente del tensor. Las sumatorias en el cálculo de A, B y C hacen referencia al método propuesto en [9].

El segundo *kernel* realiza la convolución con filtro columna de cada componente del tensor

estructural y calcula la métrica de esquinas. Este ha sido diseñado para funcionar usando bloques de tamaño 32 x N x 1. Cada *thread* lee datos de la memoria global correspondientes a cada componente, de tal forma que cada bloque almacena primero los coeficientes del filtro y 32 x $6N$ datos de cada componente a memoria compartida, donde los 32 x $4N$ datos centrales corresponden a los datos a filtrar y 2 regiones de 32 x N superior e inferior son datos necesarios para realizar la operación de convolución por componente. Así, cada *thread* se encargará del cálculo de 4 componentes filtrados por componente, entonces, 4 datos de la métrica serán calculados por *thread*.

3.4.2. Selección automática del umbral

La presente etapa se desarrolla usando cinco *kernels*. Los primeros dos realizan el cálculo de los valores mínimo, máximo y suma total de la métrica las cuales son operaciones de reducción. El primer *kernel* está diseñado tal que cada *thread* lee varios datos en memoria y calcula de forma secuencial el valor mínimo, máximo y la suma de estos datos. Luego, se efectúa una reducción dentro de cada bloque, así, se tendrá el mínimo, máximo y suma de cada valor asociado a un *thread* por bloque, luego, se escribirán tres valores por bloque en la memoria global.

El segundo opera de la misma manera que el anterior, sin embargo, en este caso, se lanza un solo bloque, entonces, se obtendrá un solo resultado por reducción, es decir, tres valores en total: máximo, mínimo y suma. En este caso serán los resultados finales, los cuales se escribirán en variables visibles desde *device*.

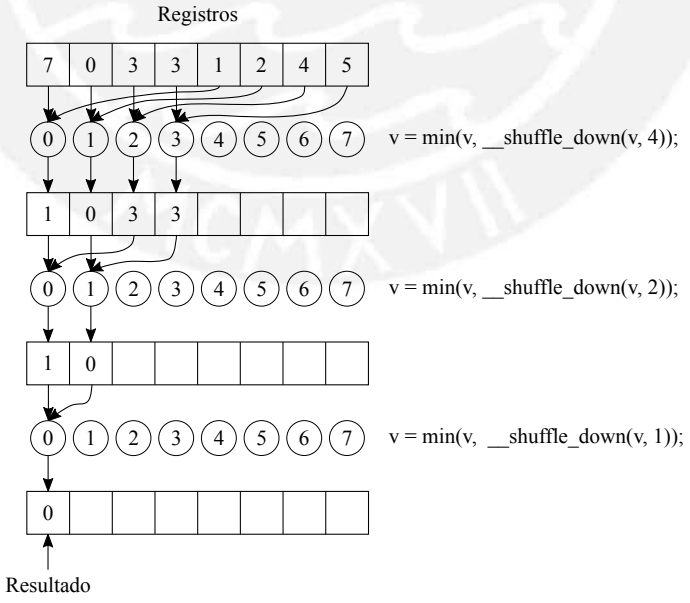


Figura 3.7: Ejemplo del cálculo de reducción usando la instrucción *__shuffle_down* para reducción dentro de un *warp* asumiendo que este tiene tamaño 8.

El método de reducción por bloque está basado en reducciones por *warp*, descritas en [36], las cuales hacen uso de la función *__shuffle_down* que permite a un *thread* leer variables de otro *thread* en su mismo *warp* sin necesidad de hacer uso de una memoria intermediaria. El método de reducción dentro de un *warp* es descrito gráficamente mediante un ejemplo en la figura 3.7.

El tercer y cuarto *kernel* realizan el cálculo del histograma. El histograma es un método difícil de paralelizar debido a que los accesos a memoria que involucra no tienen un patrón sino que dependen de la distribución de los datos analizados. Por otro lado, el conjunto de datos a analizar por esta etapa es la métrica de esquinas, la cual tiene distribución unimodal. Tomando en cuenta estas consideraciones se propone un algoritmo para el cálculo del histograma basado en el método conocido como *Histogram Per Block*, propuesto en [37], el cual plantea el cálculo de histogramas parciales por bloque, procedimiento realizado por el tercer *kernel*, y una suma de los histogramas parciales, inicializados y almacenados inicialmente en memoria compartida, para obtener el histograma total, realizado por el cuarto *kernel*. Debido a la distribución unimodal de la métrica, es posible que el algoritmo propuesto no reduzca colisiones en memoria, que ocurrirán cuando dos o más *threads* incrementen la cuenta de un mismo bin. Para evitar esto se propone usar el promedio de los datos para aproximar la moda, puesto que ambas son medidas de tendencia central, de manera que la cuenta del bin asociado a esta aproximación no sea incrementado, sino que un registro por *thread* sea usado para acumular una cuenta, luego la cuenta del bin aproximado puede calcularse sumando los valores de los registros, lo cual es una operación de reducción.

Es posible que el promedio no corresponda a algún bin cercano a la moda, por ello se propone evaluar el crecimiento de los histogramas parciales y verificar el bin que registre mayor aumento, el cual será considerado como el bin donde ocurra el mayor número de colisiones en memoria. Luego, la cuenta asociada a este bin será calculada de la misma forma que la cuenta del bin aproximado y la cuenta del bin aproximado se calculará accesando a la memoria donde se almacena el histograma parcial. Finalmente, cada bloque traslada desde la memoria compartida a la memoria global su histograma parcial.

El cuarto *kernel* calcula la suma de todos los histogramas, para ello, extrae los datos de los histogramas parciales y efectúa una reducción por bin usando registros como acumuladores. Para ello, solo se necesita un bloque, y un solo histograma es cargado a la memoria global como resultado.

Finalmente, el último *kernel* realiza el método del cálculo del umbral de Rosin utilizando el histograma. Para ello, solo utiliza un bloque de *threads* el cual realiza primero una operación de reducción para buscar el punto máximo del histograma y calcula las distancias entre la recta

especificada por Rosin y los puntos del histograma de forma paralela. Finalmente, realiza una operación de reducción para la búsqueda de la máxima distancia para calcular el umbral.

3.4.3. Supresión del no máximo

Esta etapa está desarrollada usando dos *kernels*. El primero realiza el Cálculo de máximos por bloque. Este está diseñado de forma que los *threads* lean datos desde la memoria global de forma colaborativa tal que cada *thread* lea en memoria L datos por columna, calcule el máximo y lo almacene en memoria compartida. Luego, este proceso sigue por L veces recorriendo los datos en torno a las columnas. Finalmente, cada *thread* calcula el máximo de L datos almacenados por cada L espacios contiguos en la memoria compartida, resultando en que cada *thread* tenga el valor del máximo por bloque el cual se escribe en memoria global. En este *kernel* se crea la lista inicial de coordenadas.

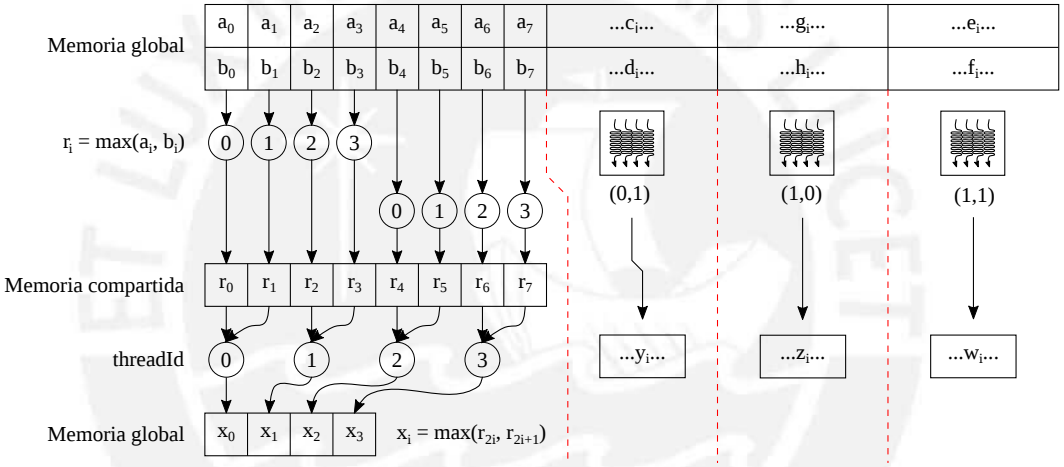


Figura 3.8: Ejemplo de operación búsqueda de máximo por bloque en CUDA para un bloque CUDA de 4 *threads* calculando máximos en ventanas de tamaño 2 x 2.

El segundo *kernel* realiza el Cálculo de máximos locales. Este está diseñado de tal forma que usa los máximos por bloque, de forma que una ventana cuadrada de 9 datos es leída por cada *thread*. En esta ventana se evaluará si el píxel central es máximo, es decir, se compararán los máximos por bloques. Si el píxel central es máximo, conserva su valor, caso contrario, su intensidad se establece a cero.

3.4.4. Selección de la lista de esquinas

Esta etapa se encuentra desarrollada por tres *kernels*. El primer *kernel* realiza eliminación de ceros en un arreglo de datos, opera sobre la lista de máximos locales y sobre la lista de coordenadas, y está diseñado de forma que se instancia un único bloque donde los *threads* leen de

forma colaborativa el arreglo de datos. Luego, cada *thread* evalúa si el dato leído es diferente a cero, si no lo es, se guardará el valor uno en un registro, caso contrario, se guadará un cero. Luego, se procede a realizar una operación de *Parallel scan* que acumula progresivamente los valores de los registros y los almacena en memoria compartida. Este valor almacenado en memoria compartida es el número de datos distintos a cero que encuentran los *threads* del bloque. Utilizando el valor del registro, se determinan los *threads* que escriben en memoria, los cuales se escriben en las direcciones de memoria indicadas por el valor obtenido de la operación *Parallel scan*. Esta operación se repite hasta terminar de recorrer el arreglo de datos de entrada en memoria global.

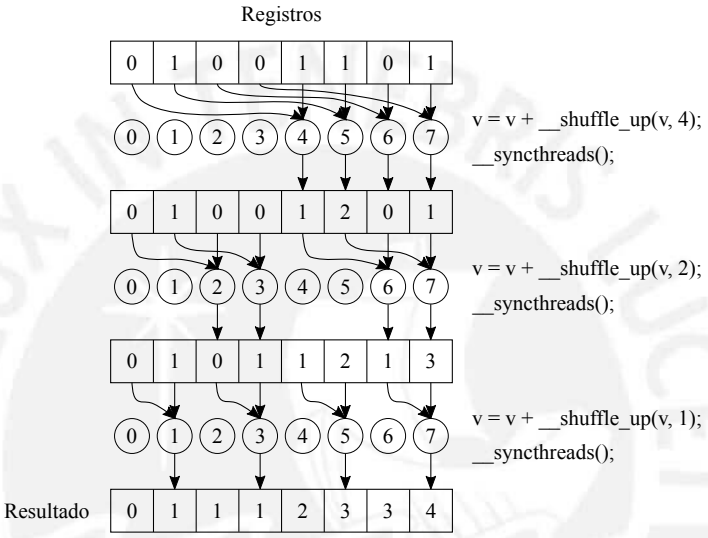


Figura 3.9: Ejemplo de operación *Parallel scan* o *Parallel prefix sum* para un *warp* de 8 *threads* basado en instrucciones *__shuffle_up*.

El segundo *kernel* está diseñado de forma que se instancia un único bloque donde cada *thread* lee dos datos de memoria global siguiendo el patrón dado por el método *Bitonicsort*, los compara y cambia su posición si es necesario. Esto se repite hasta que el patrón permita realizar las operaciones dentro de la memoria compartida, reduciendo los accesos a memoria global. El conjunto de permutaciones hecho es reproducido en la lista de coordenadas en cada comparación hecha por este ordenamiento.

Finalmente, el último *kernel* realiza la generación de la lista de columnas y filas a partir de la lista de coordenadas para lo cual se utilizan las dimensiones de la imagen de entrada. Así, se obtienen las coordenadas de los puntos identificados como esquinas.

Capítulo 4

Implementación y resultados computacionales

4.1. Consideraciones de la implementación

1. **Plataforma:** Plataforma Jetson TK1 de Nvidia, la cual está basada en el procesador ARM Cortex A15 y el GPU GK20A de arquitectura Kepler, y opera usando el sistema operativo L4T, *Linux for Tegra*.

2. **Software y herramientas de desarrollo:** MATLAB R2014b para realizar la implementación en MATLAB del método propuesto, las herramientas provistas por el proyecto GNU tales como el compilador GCC versión 4.0 fue utilizado para realizar la implementación ANSI-C, finalmente, las herramientas de Nvidia, el CUDA *Toolkit* versión 6.5 que incluyen el compilador NVCC y el *profiler* NVPROF, fueron utilizadas para realizar la implementación en CUDA. Las implementaciones ANSI-C y CUDA fueron realizadas utilizando y actualizando la biblioteca Yupana [38] utilizada por el Grupo de Procesamiento Digital de Señales e Imágenes dirigido por el Dr. Paul Rodríguez.

3. **Parámetros del detector:** Longitud del filtro, desviación estándar del fitro, factor de sensibilidad, número de bins, tamaño de ventana para Supresión del no máximo y longitud de lista de esquinas. En caso no se ingresen los parámetros mencionados, los valores por defecto son respectivamente: 5, 1.5, 0.04, 256, 4 y 200.

4. **Formatos de imagen:** El método propuesto es aplicable a imágenes en escala de grises y color. Los formatos admitidos cambian según la implementación utilizada. El método puede procesar imágenes con resolución de hasta 4096 x 4096.

4.2. Descripción de la implementación

4.2.1. Implementación MATLAB

Se ha implementado el método propuesto en MATLAB con motivos de verificación del método propuesto en cuanto a la calidad de esquinas detectadas. La implementación MATLAB comprende todas las etapas propuestas en el capítulo 3 y acepta los parámetros de configuración para el método propuesto ya expuestos, los cuales son ingresables como parámetros de una función en MATLAB. En caso no se ingrese alguno de estos parámetros se utilizan los parámetros por defecto de la implementación. La imagen analizada por la implementación puede ser ingresada como una matriz en MATLAB o como la ruta de la imagen en el sistema de archivos y puede estar en cualquier formato admitido por MATLAB. El resultado obtenido de aplicar la implementación es una matriz que contiene las coordenadas de los puntos de las esquinas detectadas.

4.2.2. Implementación ANSI-C

Se ha implementado el método propuesto en ANSI-C dirigido a la plataforma Jetson TK1, el cual está basado en el procesador ARM Cortex A15 integrado al SoC Tegra K1 de Nvidia y ha sido propuesta con motivos de comparación en cuanto a desempeño computacional (tiempo de procesamiento). La implementación ANSI-C comprende todas las etapas propuestas en el capítulo 3 y puede aceptar los mismos parámetros de configuración para el método propuesto que se pueden a la implementación MATLAB vía línea de comandos. La imagen analizada por la implementación solo pueder ser ingresada vía línea de comandos como la ruta de la imagen en el sistema de archivos y solo pueden analizarse imágenes en formato PGM (escala de grises) y PPM (color). El resultado de la implementación presentada son dos archivos en formato RAW que contienen las listas de coordenadas en columnas y filas.

4.2.3. Implementación CUDA

Se ha implementado el método propuesto en CUDA dirigido a la plataforma Jetson TK1. El GPU usado tiene el código GK20A, el cual está integrado al SoC Tegra K1. Esta implementación es la que se va a analizar y tiene las mismas características que la implementación ANSI-C en cuanto a ingreso de parámetros de configuración, ingreso de la imagen y obtención de resultados. La implementación CUDA tiene una restricción en cuanto a los tamaños de filtro y de ventana para Supresión del No Máximo, siendo los valores máximos 9 y 32 respectivamente.

4.3. Consideraciones del método de evaluación de la implementación

La métrica de esquinas calculada por la función propuesta Harris varía dependiendo de la imagen analizada, entonces, el número de puntos detectados y la distribución de estas es variable. Procedimientos como el cálculo del histograma tienen un rendimiento dependiente de la distribución de los datos que se están analizando, además, la lista de puntos detectados tendrá una longitud diferente para imágenes diferentes, por lo que el ordenamiento tendrá un rendimiento variable por cada imagen. Por ello, se espera que el rendimiento computacional del método propuesto sea variable ante imágenes de la misma resolución porque sería dependiente también del contenido de la imagen analizada.

La verificación de la calidad de esquinas detectadas por la implementación presentada se realiza usando imágenes de prueba para las cuales se debe tener, para cada una, un *groundtruth*, es decir, una lista de coordenadas de las esquinas reales. Para determinar la calidad de esquinas detectadas se comparan las implementaciones presentadas y una referencia, la función corner de MATLAB que también es una implementación del detector de esquinas de Harris. Los parámetros de configuración de prueba son los parámetros por defecto ya definidos en el caso de las implementaciones presentadas y los parámetros por defecto de la función corner pero sin aplicar un limitante de esquinas. Utilizando estas listas se podrán definir métricas para la evaluación de la calidad de las esquinas detectadas.

Para efectuar una prueba adecuada del desempeño computacional del método presentado, se han generado tres grupos de imágenes LOW, MED y FULL, las cuales tienen forma de tableros de ajedrez y se diferencian en el número de esquinas presentes, esto con el objetivo de comprobar si el método presenta variaciones significativas en rendimiento computacional para imágenes de igual resolución pero diferente contenido. El primer grupo, LOW, tiene imágenes donde hay 1 esquina, el grupo MED, imágenes donde hay 49 esquinas , y el grupo FULL, imágenes que tienen 961 esquinas presentes. Los tres grupos están conformados por imágenes de resoluciones 128 x 128, 256 x 256, 512 x 512, 1024 x 1024, 2048 x 2048 y 4096 x 4096, lo cual permitirá realizar un estudio y aplicar métricas de desempeño a la implementación presentada.

Finalmente, para tener resultados con bajo margen de error respecto a tiempo computacional se procederá a realizar 100 ejecuciones del programa por imagen de análisis. Luego, la media de los tiempos de ejecución para una imagen de resolución determinada para cada grupo de imágenes será el resultado buscado.

4.4. Resultados computacionales

El método propuesto se implementó en la plataforma Jetson TK1 funcionando con frecuencias de reloj de CPU y GPU de 2.33 GHz y a 852 MHz respectivamente, frecuencia de reloj de memoria de 952 MHz. El sistema operativo de la plataforma es *Linux for Tegra* (L4T) Ubuntu 14.04 de 32 bits, Kernel versión 3.10.40.

4.4.1. Evaluación de las implementaciones en MATLAB, ANSI-C y CUDA

Se muestran en las figuras los resultados obtenidos por las implementaciones MATLAB, ANSI-C y CUDA para cuatro diferentes imágenes de prueba y su *groundtruth* respectivo. Se puede evaluar por simple inspección que todas las implementaciones detectan esquinas en las imágenes de prueba, sin embargo, se presenta a continuación una comprobación formal de ello. Para evaluar las implementaciones mencionadas se calculan las coordenadas de las esquinas detectadas por las implementaciones propuestas y se comparan con el *groundtruth*. Se considera un acierto si entre las coordenadas evaluadas y las coordenadas del *groundtruth* no hay una diferencia mayor a cuatro pixeles en distancia.

Para la evaluación de la calidad de esquinas detectadas se han utilizado dos parámetros: *precision* y *recall* siendo definidos como el porcentaje de esquinas detectadas por la implementación analizada que corresponden a un acierto y el porcentaje de aciertos obtenidos respectivamente tal y como se muestra en las ecuaciones 4.1 y 4.2 respectivamente:

$$\text{Precision} = 100\,\% \left[\frac{\text{Número de aciertos}}{\text{Número total de esquinas detectadas}} \right] \tag{4.1}$$

$$\text{Recall} = 100\,\% \left[\frac{\text{Número de aciertos}}{\text{Número total de esquinas verdaderas}} \right] \tag{4.2}$$

En la tabla 4.1 se muestran los resultados de aplicar las implementaciones evaluadas frente a cuatros imágenes de prueba: Mold, Building, Toyball y Origami, de las cuales las primeras tres fueron extraídas de [39] donde se proporciona también un *groundtruth* obtenido de forma manual para cada imagen, la última imagen de prueba se ha obtenido de forma independiente y se ha obtenido el *groundtruth* correspondiente de forma manual. Se muestra que la referencia de MATLAB (función corner) detecta un gran número de puntos, entre los cuales se tiene un gran número de aciertos pero que no representan en general un porcentaje signficativo respecto del número total de puntos detectados por lo cual se puede afirmar que el detector provisto por MATLAB es impreciso pero detecta una mayor cantidad de esquinas. En contraste, las

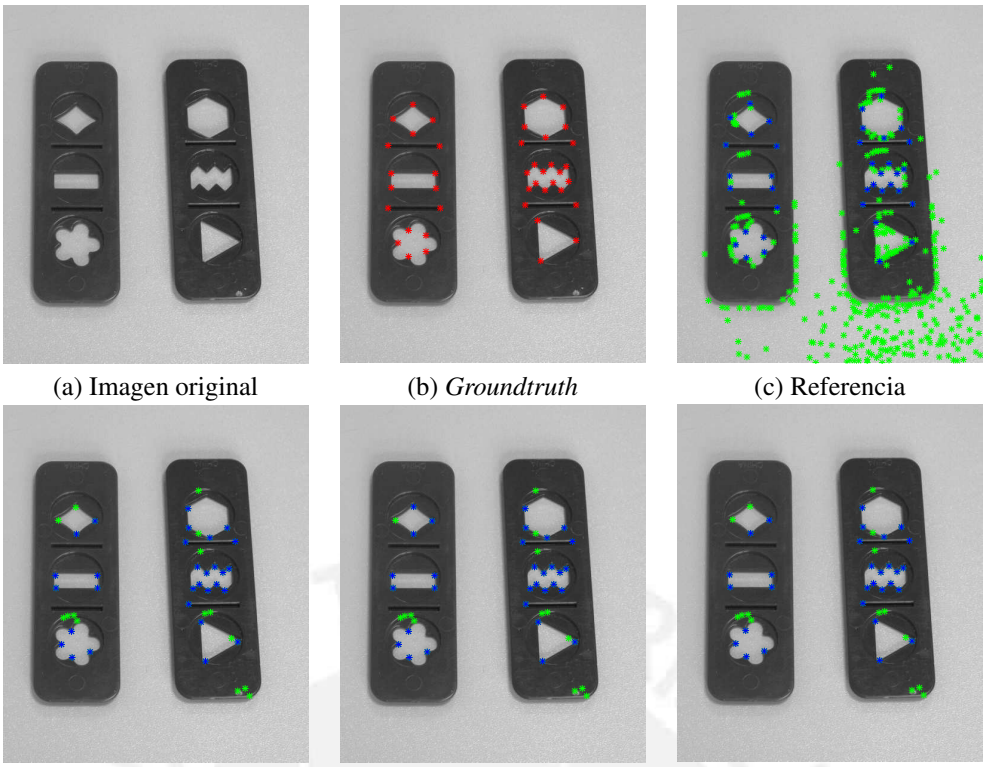
implementaciones presentadas detectan un número de puntos menor en la mayoría de casos donde el porcentaje de aciertos es significativo tanto respecto al número total de esquinas verdaderas como al total de puntos detectados. Por ello, las implementaciones presentadas representan un balance entre *precision* y *recall*, siendo bastante más precisas que esta pero con un poco menor número de esquinas verdaderas detectado.

Los resultados indican también que las etapas de selección automática del umbral y Supresión del No Máximo cumplieron con su objetivo exitosamente pues se redujo el número de falsos positivos, puntos detectados como esquinas que no guardan correspondencia con el *groundtruth*. Además se puede observar que los resultados obtenidos entre la implementaciones MATLAB, ANSI-C y CUDA son los mismos (parecidos para los resultados de Mold). Es decir, la implementación CUDA, a pesar de realizar operaciones que usan las unidades de hardware especializadas en operaciones aritméticas que permiten un menor tiempo de procesamiento pero generan un margen de error, es una paralelización adecuada.

Para una visualización gráfica de los resultados, estos se presentan en las figuras 4.1, 4.2, 4.3 y 4.4 donde se muestran la imagen original, el *groundtruth* y las esquinas detectadas por las implementaciones evaluadas. Los puntos rojos son las esquinas verdaderas y solo se muestran en el *groundtruth*, los puntos azules y verdes representan los verdaderos y falsos positivos respectivamente.

Imagen	**Esquinas reales**	**Implementación**	**Esquinas detectadas**	**Verdaderos positivos**	**Falsos positivos**	***Precision***	***Recall***
Mold	42	Referencia MATLAB	376	38	338	10.64 %	95.24 %
		MATLAB	45	30	15	66.67 %	71.43 %
		ANSI-C	45	31	14	68.89 %	73.81 %
		CUDA	45	30	15	66.67 %	71.43 %
Building	234	Referencia MATLAB	340	217	123	63.82 %	92.74 %
		MATLAB	210	193	17	91.9 %	82.48 %
		ANSI-C	210	193	17	91.9 %	82.48 %
		CUDA	210	193	17	91.9 %	82.48 %
Toyball	22	Referencia MATLAB	47	19	28	40.43 %	86.36 %
		MATLAB	32	18	14	56.25 %	81.82 %
		ANSI-C	32	18	14	56.25 %	81.82 %
		CUDA	32	18	14	56.25 %	81.82 %
Origami	10	Referencia MATLAB	9	7	2	77.78 %	70 %
		MATLAB	11	9	2	81.82 %	90 %
		ANSI-C	11	9	2	81.82 %	90 %
		CUDA	11	9	2	81.82 %	90 %

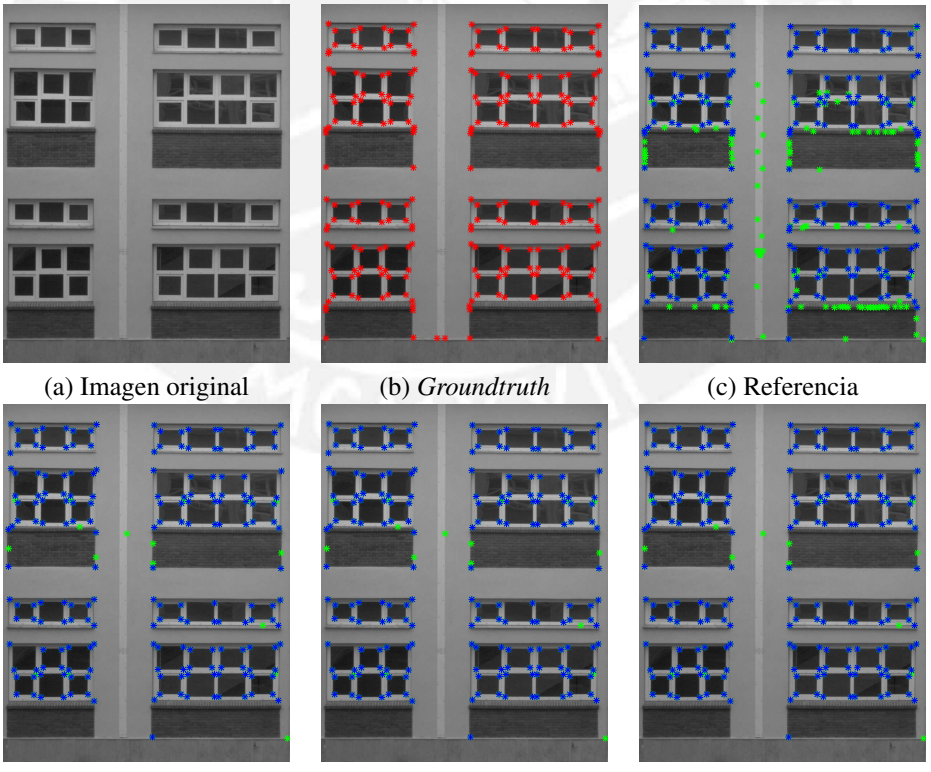
Tabla 4.1: Desempeño de las implementaciones presentadas comparadas con la Referencia (función corner de MATLAB) y el *groundtruth*.



(a) Imagen original (b) *Groundtruth* (c) Referencia

(d) MATLAB (e) ANSI-C (f) CUDA

Figura 4.1: Resultados de desempeño del detector para la imagen Mold.

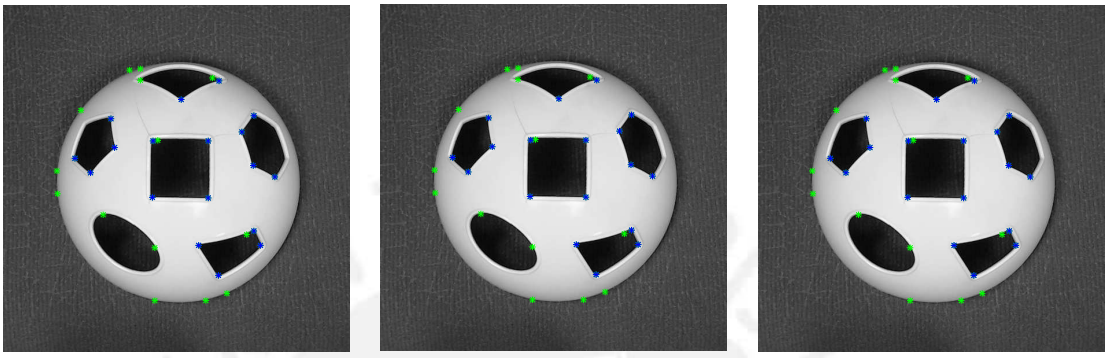


(a) Imagen original (b) *Groundtruth* (c) Referencia

(d) MATLAB (e) ANSI-C (f) CUDA

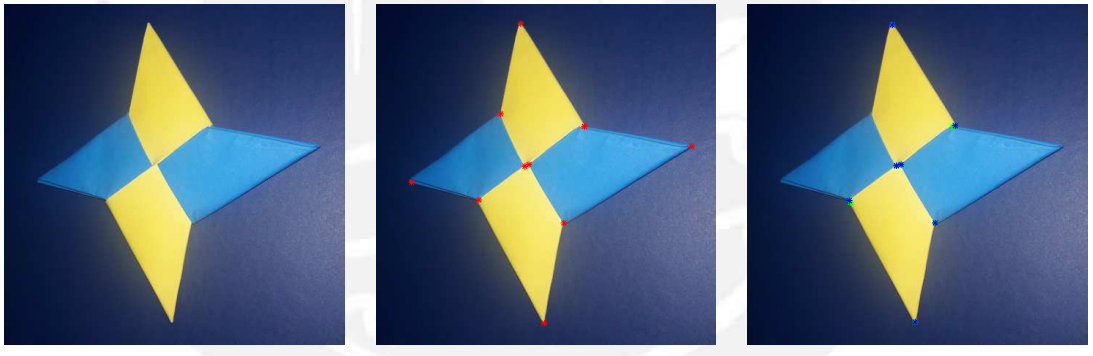
Figura 4.2: Resultados de desempeño del detector para la imagen Building.

(a) Imagen original (b) *Groundtruth* (c) Referencia

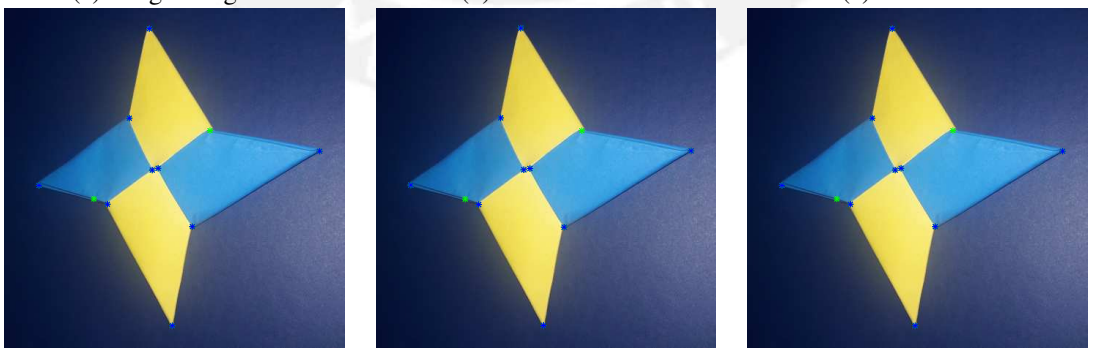


(d) MATLAB (e) ANSI-C (f) CUDA

Figura 4.3: Resultados de desempeño del detector para la imagen Toyball.



(a) Imagen original (b) *Groundtruth* (c) Referencia



(d) MATLAB (e) ANSI-C (f) CUDA

Figura 4.4: Resultados de desempeño del detector para la imagen Origami.

4.4.2. Evaluación de desempeño computacional de la implementación CUDA

Se presenta en esta sección los resultados de las implementaciones ANSI-C y CUDA se compararán. Se utilizarán como medidas de rendimiento el tiempo de ejecución (incluyendo transferencias entre *host* y *device*), el *speedup* [27] y cuadros por segundo. El *speedup* es una medida de rendimiento que está definida como:

$$S_p = \frac{T(n)}{T_p(n)}$$

donde $T(n)$ y $T_p(n)$ son los tiempos de ejecución para la implementación secuencial más rápida y la implementación en paralelo para un problema de tamaño n. Mientras que cuadros por segundo se define como la cantidad de cuadros de vídeo de cierta resolución que pueden ser procesados por segundo. Se evalúan las implementaciones para un filtro de tamaño 5 x 5, ventana de 4 x 4 y 256 bins.

Adicionalmente, se propone comparar el rendimiento de las implementaciones ya mencionadas con una implementación del detector de esquinas de Harris basada en la biblioteca OpenCV (*Open Source Computer Vision*), ampliamente usada en proyectos de Visión por Computadora. Para realizar una comparación justa, se está comparando la implementación que usa el uso módulo GPU de OpenCV, el cual permite usar el procesador gráfico incorporado en la plataforma. Al respecto, se ha relizado una implementación del método básico de detección de Harris (Cálculo de la métrica de esquinas y Umbralización), para ello se usan las funciones cornerHarris, normalize y thresholding que OpenCV provee y utilizan el módulo GPU. Cabe mencionar que no hay un método de OpenCV de conversión RGB a escala de grises que use dicho módulo por lo que esencialmente el tiempo de detección para imágenes a color y escala de grises es el mismo para imágenes de igual resolución.

Las tablas 4.2 y 4.3 muestran los tiempo de procesamiento en milisegundos y cuadros por segundo alcanzados por las implementaciones ANSI-C, OpenCV y CUDA para imágenes de los grupos LOW, MED y FULL en escala de grises y color respectivamente. Cabe mencionar que estas mediciones de tiempo incluyen los tiempos de transferencia de memoria entre *device* y *host* en el caso de la implementación CUDA. Adicionalmente, se muestran los factores de mejora que logran las implementaciones CUDA y OpenCV respecto de la implementación ANSI-C . Para una vizualización más apropiada de los resultados, estos se muestran en las figuras 4.5 y 4.6 para imágenes de resolución desde 512 x 512 en los distintos grupos analizados.

Grupo	Resolución	Tiempo de procesamiento (ms)			Cuadros por segundo (FPS)			Speedup	
		ANSI-C	OpenCV	CUDA	ANSI-C	OpenCV	CUDA	CUDA vs. ANSI-C	OpenCV vs. ANSI-C
LOW	128 x 128	1.88	3.64	0.57	531.98	274.73	1745.41	3.28	0.52
	256 x 256	8.61	5.56	0.88	116.12	179.86	1139.37	9.81	1.55
	512 x 512	48.4	12.18	2.08	20.66	82.1	479.85	23.22	3.97
	1024 x 1024	215.53	33.57	7.73	4.64	29.79	129.37	27.88	6.42
	2048 x 2048	891.55	118.24	30.13	1.12	8.46	33.19	29.59	7.54
	4096 x 4096	3901.13	478.71	121.59	0.26	2.09	8.22	32.08	8.15
MED	128 x 128	1.87	3.67	0.63	536.15	272.48	1594.44	2.97	0.51
	256 x 256	8.64	5.64	1.05	115.74	177.3	949.13	8.2	1.53
	512 x 512	47.81	12.06	2.43	20.92	82.92	411.24	19.66	3.96
	1024 x 1024	215.5	33.41	8.37	4.64	29.93	119.42	25.73	6.45
	2048 x 2048	893.48	118.53	31.1	1.12	8.44	32.15	28.73	7.54
	4096 x 4096	3942.02	477.89	123.44	0.25	2.09	8.1	31.93	8.25
FULL	128 x 128	2.09	3.89	0.56	478.76	257.07	1792.05	3.74	0.54
	256 x 256	8.81	5.69	1.12	113.52	175.75	893.99	7.88	1.55
	512 x 512	48.18	11.94	2.91	20.75	83.75	343.85	16.57	4.04
	1024 x 1024	214.01	33.56	9.6	4.67	29.8	104.16	22.29	6.38
	2048 x 2048	885.9	119.7	34.2	1.13	8.35	29.24	25.9	7.4
	4096 x 4096	3969.12	480.62	129.4	0.25	2.08	7.73	30.67	8.26

Tabla 4.2: Resultados de rendimiento computacional de la implementación presentada para imágenes en escala de grises.

Grupo	Resolución	Tiempo de procesamiento (ms)			Cuadros por segundo (FPS)			Speedup	
		ANSI-C	OpenCV	CUDA	ANSI-C	OpenCV	CUDA	CUDA vs. ANSI-C	OpenCV vs. ANSI-C
LOW	128 x 128	2.21	3.64	0.63	452.19	274.73	1576.22	3.49	0.61
	256 x 256	12.17	5.56	1.07	82.18	179.86	937.48	11.41	2.19
	512 x 512	67.36	12.18	2.97	14.84	82.1	337.01	22.7	5.53
	1024 x 1024	286.27	33.57	12.15	3.49	29.79	82.32	23.57	8.53
	2048 x 2048	1160.59	118.24	46.69	0.86	8.46	21.42	24.86	9.82
	4096 x 4096	5016.03	478.71	189.29	0.2	2.09	5.28	26.5	10.48
MED	128 x 128	2.24	3.67	0.68	446.54	272.48	1463.94	3.28	0.61
	256 x 256	12.21	5.64	1.27	81.9	177.3	788.56	9.63	2.16
	512 x 512	66.82	12.06	3.13	14.96	82.92	319.67	21.36	5.54
	1024 x 1024	289.78	33.41	12.69	3.45	29.93	78.89	22.84	8.67
	2048 x 2048	1157.5	118.53	46.94	0.86	8.44	21.3	24.66	9.77
	4096 x 4096	5030.31	477.89	189.18	0.2	2.09	5.29	26.59	10.53
FULL	128 x 128	2.5	3.89	0.65	399.39	257.07	1540.17	3.86	0.64
	256 x 256	12.47	5.69	1.21	80.19	175.75	827.34	10.32	2.19
	512 x 512	65.87	11.94	3.71	15.18	83.75	269.56	17.76	5.52
	1024 x 1024	284.17	33.56	13.73	3.52	29.8	72.84	20.7	8.47
	2048 x 2048	1158.31	119.7	49.27	0.86	8.35	20.29	23.51	9.68
	4096 x 4096	5046.66	480.62	193.74	0.2	2.08	5.16	26.05	10.5

Tabla 4.3: Resultados de rendimiento computacional de la implementación presentada para imágenes a color.

Se muestra en las tablas que la implementación CUDA tiene el mejor rendimiento computacional comparado con las implementaciones ANSI-C y OpenCV para todos los tamaños presentados. En particular, se observa que la implementación CUDA presenta variaciones pequeñas en tiempo de procesamiento cuando se compara el rendimiento para imágenes a color o en escala de grises para los grupos LOW, MED y FULL lo cual indica que se ha minimizado el efecto del cálculo del histograma y generación de lista de esquinas, métodos que inducen diferencia en rendimiento para imágenes de igual resolución. Se observa también que el factor de

mejora de la implementación CUDA respecto a la implementación ANSI-C es estrictamente creciente respecto de la resolución de la imagen analizada. Esto se debe a que el tiempo que se necesita para ejecutar las operaciones que realiza el método es mayor al tiempo de transferencia de memoria entre el GPU y el CPU. Además, el rendimiento de la implementación CUDA es menor cuando se analizan imágenes a color debido a que el número de operaciones aumenta según el método propuesto en [9]. Los FPS alcanzados por la implementación CUDA varían entre 7.73 y 1792.05 para imágenes en escala de grises y entre 5.16 y 1576.22 para imágenes a color para resoluciones de imagen desde 128 x 128 hasta 4096 x 4096.

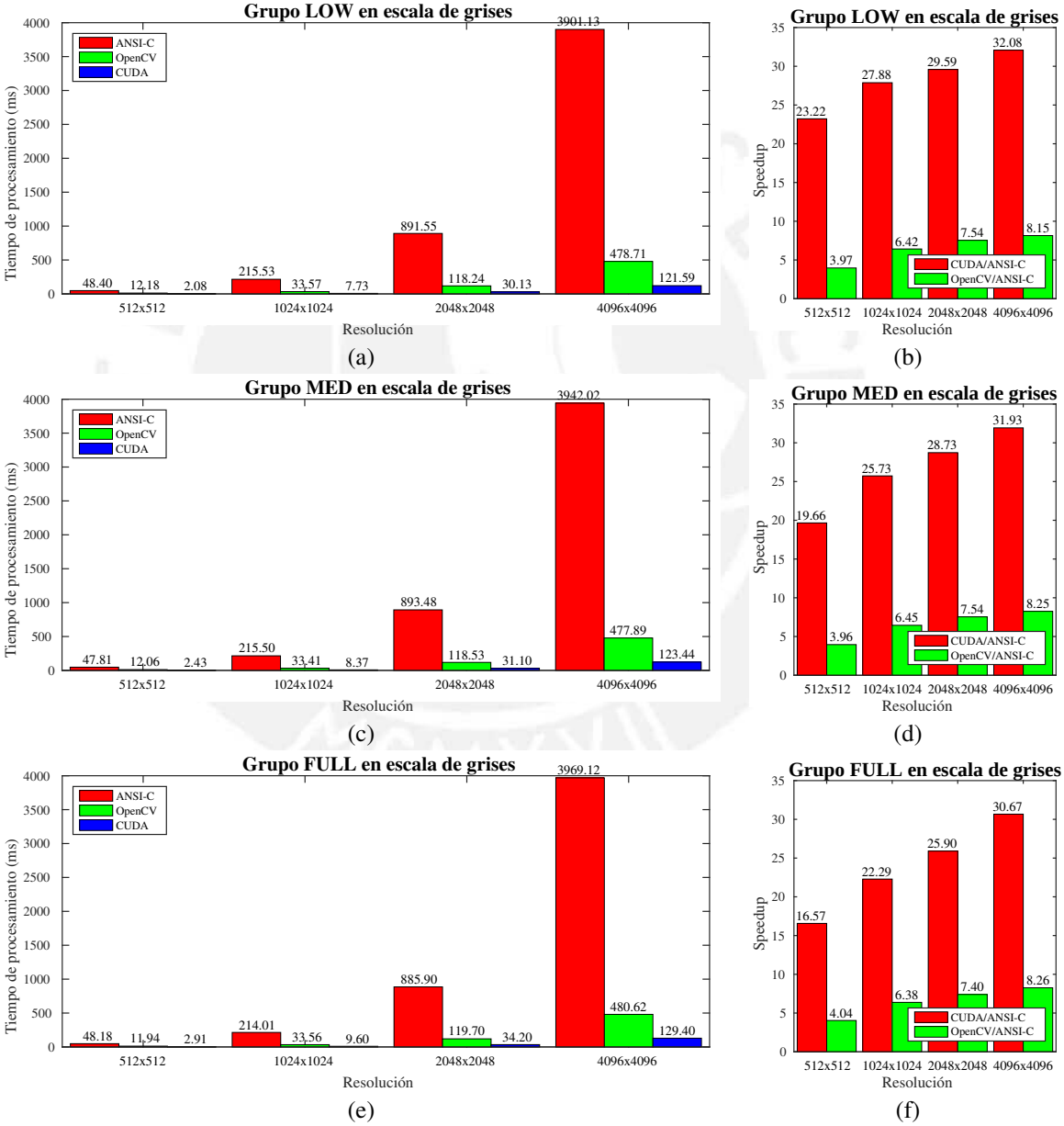


Figura 4.5: (a), (c) y (e) Tiempos de procesamiento para imágenes en escala de grises. (b), (d) y (f) Speedups presentados respecto de la implementación CUDA para imágenes en escala de grises.

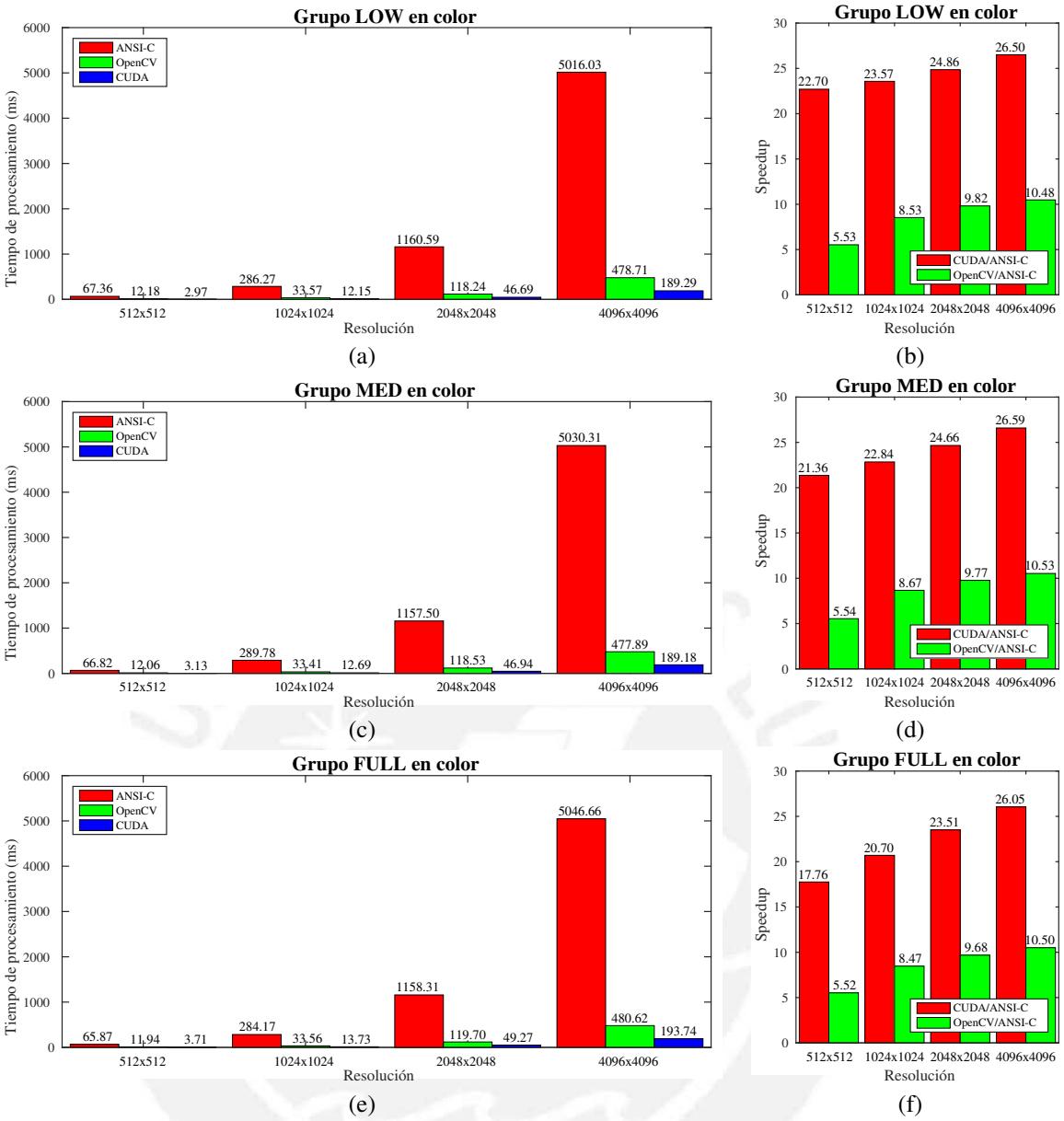


Figura 4.6: (a), (c) y (e) Tiempos de procesamiento para imágenes a color. (b), (d) y (f) Speedups presentados respecto de la implementación CUDA para imágenes a color.

4.5. Comparación de la implementación CUDA con el estado del arte

En esta sección se compararán los resultados de la implementación presentada con implementaciones presentadas en la literatura que utilizan plataformas que realizan procesamiento paralelo. Cabe mencionar que los métodos enunciados no presentan sus imágenes de prueba así que se ha optado por realizar una comparación solo de desempeño computacional utilizando imágenes reales.

Algoritmo	Plataforma	Tipo de imagen	Tamaño de filtro	Tamaño de ventana	Resolución	Tiempo de procesamiento (ms)	Cuadros por segundo (FPS)
LoCoCo [40]	Nvidia Geforce GTX 280 @ 1296 MHz	Escala de grises	-	3 x 3	640 x 480	2.4	416.67
Harris [41]	Nvidia Geforce GTX 650 @ 1058 MHz	Color	5 x 5	3 x 3	162 x 122	0.86	1162.8
					400 x 400	1.31	763.36
Bouguet [42]	Nvidia GTX 580 @ 1544 MHz	Color	-	3 x 3	512 x 512	2.32	431.03
					1024 x 1024	4.49	222.71
Harris [42]	Altera Arria V @ 232 MHz	Color	5 x 5	3 x 3	512 x 512	1.15	869.57
					1024 x 1024	4.56	220.26
Harris [43]	Xilinx Spartan 6 @ 153 MHz	Escala de grises	3 x 3	3 x 3	1024 x 1024	6.85	146
					1280 x 720	6.02	166
	Xilinx Spartan 6 @ 138 MHz			7 x 7	1024 x 1024	7.69	130
					1280 x 720	7.46	134
	Xilinx Virtex 5 @ 225 MHz			3 x 3	1024 x 1024	4.67	214
					1280 x 720	4.09	244
	Xilinx Virtex 5 @ 225 MHz			7 x 7	1024 x 1024	4.69	213
					1280 x 720	4.15	241
SUSAN [44]	Xilinx Virtex II Pro @ 100 MHz	Escala de grises	-	-	640 x 480	3.142	318.26
Harris [45]	Clearspeed CSX700 @ 250 MHz	Escala de grises	3 x 3	-	512 x 512	1.74	574
					640 x 480	2.15	465
					1280 x 720	7.04	142
			5 x 5		512 x 512	2.63	380
					640 x 480	3.28	304
					1280 x 720	10.89	91
Harris [46]	Nvidia Geforce 310m	Escala de grises	9 x 9	*	1024 x 1024	121.95	13.8
	Nvidia Geforce 520m					72.46	8.2
	Nvidia Geforce GTS 450					13.01	74
	Nvidia Geforce GTX 480					5.08	197
LoCoCo [46]	Nvidia Geforce 310m	Escala de grises	-	*	1024 x 1024	133.33	7.5
	Nvidia Geforce 520m					76.43	13.1
	Nvidia Geforce GTS 450					14.71	68
	Nvidia Geforce GTX 480					5.18	193
Harris [47]	Avnet Zedboard	Escala de grises	5 x 5	-	640 x 480	6.94	144
						10.2	93
Harris [48]	Altera Cyclone IV @ 208 MHz	Escala de grises	5 x 5	7 x 7	1024 x 1024	5	200
Harris (propuesto)	Nvidia Tegra K1 (GK20A) @ 852 MHz	Escala de grises	3 x 3	4 x 4	512 x 512	2.15	465.12
					640 x 480	2.36	423.33
					1280 x 720	7.56	132.28
					1024 x 1024	8.02	124.69
					1920 x 1080	16.29	61.39
			5 x 5		512 x 512	2.47	404.86
					640 x 480	2.54	393.7
					1280 x 720	8.06	124.07
					1024 x 1024	8.42	118.76
					1920 x 1080	17.49	57.18
		Color	3 x 3		512 x 512	3.02	331.13
					640 x 480	3.75	266.67
					1280 x 720	10.76	92.93
					1024 x 1024	12.72	78.62
					1920 x 1080	24.56	40.72
			5 x 5		512 x 512	3.24	308.64
					640 x 480	3.67	272.48
					1280 x 720	12.38	80.77
					1024 x 1024	12.69	78.8
					1920 x 1080	27.09	36.91

Tabla 4.4: Comparación con estado del arte. Se indican los tamaños del filtro gaussiano y ventana para Supresión del no máximo utilizadas en cada trabajo. Los resultados presentados se encuentran bajo la etiqueta Harris (propuesto), * quiere decir que el parámetro no fue especificado.

Se puede apreciar en la tabla 4.4 que el método implementado tiene un buen rendimiento,

pues puede procesar imágenes en resolución full HD (1920 x 1080) en tiempo real (procesa a más de 30 cuadros por segundo). También se puede observar que presenta mayor rendimiento frente a [44], [46] (para todas las plataformas excepto para Nvidia Geforce GTX 480) y [47] (las dos variantes de la implementación presentada se diferencian en el uso de recursos del FPGA de la plataforma que se utiliza en dicho trabajo) mientras que es competitiva con [45] y la implementación hecha en Spartan 6 del detector de esquinas de Harris (arquitectura híbrida que permite ventana de Supresión del no máximo de tamaño 7 x 7) presentada en [43], siendo la implementación presentada en algunos casos más eficiente que las nombradas. Cabe mencionar que las implementaciones realizadas en el estado del arte realizan solo el Cálculo de la métrica de esquinas, umbralización y una Supresión del no máximo básica (máximos por bloque), por ello, no garantizan una detección adecuada y controlada a diferencia del método propuesto pero exhiben un menor costo computacional, lo cual explica un tiempo de procesamiento más corto que la implementación CUDA.

Por otro lado, la mayor parte de implementaciones se han hecho en plataformas con grandes recursos. En [42] y en [43] se usan FPGAs de la familia Arria V de Altera y de la familia Virtex 5 de Xilinx, usadas por su gran desempeño y potencial computacional. Ambas cuentan con un número muy alto de celdas lógicas y unidades dedicadas a operaciones para DSP (*Digital Signal Processing*). También en [42] se propone la implementación del detector de Bouguet, que solo necesita del cálculo de los elementos de la matriz de autocorrelación prescindiendo de las operaciones de convolución, en un GPU de mayores prestaciones (mayor número de CUDA *cores*, mayores frecuencias de operación del procesador y memoria, entre otros). También se puede mencionar que los GPUs donde se han implementado los trabajos presentados en [41] y [40] tienen mayores capacidades que Tegra K1, entre ellas, número de CUDA *cores* y frecuencias de operación del procesador y memoria.

Conclusiones

La implementación del método propuesto en la plataforma Jetson TK1 usando CUDA presenta un factor de mejora desde un factor desde 2.97 hasta 32.08 y desde 3.28 hasta 26.59 respecto de la implementación ANSI-C en el procesador ARM Cortex A15 para imágenes de resolución desde 128 x 128 hasta 4096 x 4096 en escala de grises y color respectivamente, donde este crece a medida que se analiza una imagen de mayor resolución. Esto responde al hecho de que las transferencias de memoria con efectivamente compensadas por el método desarrollado, es decir, estas no limitan el desempeño computacional del método de forma signifcativa a pesar de que se incrementen con la resolución.

El método propuesto tiene un desempeño dependiente de las características de la imagen analizada, lo cual puede comprobarse al utilizar el método para detección de esquinas en imágenes de la misma resolución pero de diferente naturaleza. Esto se debe a que el histograma es un método dependiente de la distribución de datos y la selección de la lista de esquinas opera sobre listas de longitud variable por imagen. Estos métodos, sin embargo, representan una fracción pequeña respecto al tiempo total de procesamiento. Otra característica de la imagen que afecta el rendimiento del método es si la imagen está en escala de grises o a color, esto es debido a que el método usado requiere un mayor número de operaciones y accesos a memoria para imágenes a color en el cálculo de la métrica de esquinas. Este factor altera el rendimiento de forma significativa lo cual se puede observar en los resultados presentados.

El método presentado logra seleccionar un umbral adecuado y filtrar falsos positivos de forma eficiente para la métrica dada por el detector de esquinas Harris y su extensión para imágenes a color. Así, este permite calcular esquinas en imágenes de hasta resolución full HD (1920 x 1080) a una tasa mayor a 30 cuadros por segundo y de hasta 4096 x 4096 hasta a 8.22 cuadros por segundo con resultados aceptables y superiores al detector de referencia (MATLAB). En conclusión, la implementación usando CUDA en la plataforma Jetson TK1 mostrada puede procesar imágenes de alta resolución en tiempo real hasta para imágenes en resolución full HD y a una alta tasa de cuadros por segundos para resoluciones mayores con precisión adecuada.

Recomendaciones

El método propuesto fue implementado en la plataforma Jetson TK1, la cual posee una capacidad de cómputo 3.2 y tiene pocos recursos en comparación de otros GPUs con capacidad CUDA. En tal sentido, se recomienda implementar el método en una plataforma con capacidad de cómputo mayor como las arquitecturas Kepler (3.5) o Maxwell (5.x), en las cuales se puede tener mejor manejo en accesos de memoria y funcionalidades como Paralelismo dinámico que permitirían implementaciones posiblemente más eficientes. En tal sentido, una alternativa en el marco de *mobile* CPU sería la plataforma Jetson TX1 [49] de capacidad de cómputo 5.2 y mayores prestaciones.

Por otro lado, se recomienda analizar el código PTX (*Parallel Thread Execution*) producido por el compilador e identificar las partes del código donde se produce divergencia, analizar el paralelismo a nivel de instrucciones (ILP) y reescribir ciertas partes del código CUDA en PTX de ser necesario. Además, se recomienda optimizar los accesos a memoria global, pues los datos de la imagen se encuentran en formato char (8 bits), por lo cual no se aprovecha la longitud total del bus de lectura de 64 bits. En tal sentido, usar datos vectorizados podría ser beneficioso.

Finalmente, se recomienda analizar la función de métrica de Harris y realizar comparaciones con otros métodos basados en la matriz de autocorrelación tales como [50] o [51] con el fin de determinar la métrica más adecuada para una obtención más robusta de esquinas. Además, se recomienda utilizar otros métodos de Supresión del No Máximo para reducir el número de comparaciones por píxel tales como el presentado en [52] y extender el método de ordenamiento *Bitonicsort* para operar sobre listas de cualquier longitud.

Bibliografía

[1] L. Huang and M. Barth, "Real-time Multi-vehicle Tracking Based on Feature Detection and Color Probability Model," in *2010 IEEE Intelligent Vehicles Symposium*, pp. 981–986, June 2010.

[2] K. Peng, X. Chen, D. Zhou, and Y. Liu, "3D Reconstruction Based on SIFT and Harris Feature Points," in *2009 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, pp. 960–964, Dec 2009.

[3] Y. Pei, H. Wu, J. Yu, and G. Cai, "Effective Image Registration based on Improved Harris Corner Detection," in *2010 International Conference on Information, Networking and Automation (ICINA)*, vol. 1, pp. V1–93–V1–96, Oct 2010.

[4] S. Kim, I. S. Kweon, and W. H. Lee, "Orientation based multi-scale corner detection for mobile robot application," *Control, Automation and Systems (ICCAS), 2012 12th International Conference on*, pp. 466–468, Oct 2012.

[5] S. M. Smith and J. M. Brady, "SUSAN - A New Approach to Low Level Image Processing," *International Journal of Computer Vision*, vol. 23, pp. 45–78, 1995.

[6] E. Rosten, R. Porter, and T. Drummond, "Faster and Better: A Machine Learning Approach to Corner Detection," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 32, pp. 105 –119, jan. 2010.

[7] NVIDIA Corporation, *NVIDIA CUDA C Programming Guide*, August 2014.

[8] C. Harris and M. Stephens, "A Combined Corner and Edge Detector," in *Proceedings of the 4th Alvey Vision Conference*, pp. 147–151, September 1988.

[9] P. Montesinos, V. Gouet, and R. Deriche, "Differential invariants for color images," in *Proceedings. Fourteenth International Conference on Pattern Recognition (Cat. No.98EX170)*, vol. 1, pp. 838–840 vol.1, Aug 1998.

[10] H. P. Moravec, *Obstacle Avoidance and Navigation in the Real World by a Seeing Robot Rover*. PhD thesis, Carnegie-Mellon University, september 1980.

[11] P. Mainali, Q. Yang, G. Lafruit, R. Lauwereins, and L. V. Gool, "Lococo: low complexity corner detector," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 810–813, March 2010.

[12] K. Mikolajczyk and C. Schmid, "Scale and affine invariant interest point detectors," *International Journal of Computer Vision*, vol. 60, no. 1, pp. 63–86, 2004.

[13] A. P. Witkin, "Scale-space Filtering," in *Proceedings of the Eighth International Joint Conference on Artificial Intelligence - Volume 2*, IJCAI'83, (San Francisco, CA, USA), pp. 1019–1022, Morgan Kaufmann Publishers Inc., 1983.

[14] T. Lindeberg, "Feature Detection with Automatic Scale Selection," *Int. J. Comput. Vision*, vol. 30, pp. 79–116, Nov. 1998.

[15] L. hui Zou, J. Chen, J. Zhang, and L. hua Dou, "The Comparison of Two Typical Corner Detection Algorithms," in *Intelligent Information Technology Application, 2008. IITA '08. Second International Symposium on*, vol. 2, pp. 211–215, Dec 2008.

[16] W. Förstner, "A Framework for Low Level Feature Extraction.," in *ECCV (2)* (J.-O. Eklundh, ed.), vol. 801 of *Lecture Notes in Computer Science*, pp. 383–394, Springer, 1994.

[17] R. Horaud, F. Veillon, and T. Skordas, *Computer Vision — ECCV 90: First European Conference on Computer Vision Antibes, France, April 23–27, 1990 Proceedings*, ch. Finding geometric and relational structures in an image, pp. 374–384. Berlin, Heidelberg: Springer Berlin Heidelberg, 1990.

[18] J. C. Cottier, "Extraction et appariements robustes des points d'interest dedeux images non etalonne," tech. rep., Rhone-Alpes, 1994.

[19] F. Heitger, L. Rosenthaler, R. V. D. Heydt, E. Peterhans, and O. Kübler, "Simulation of neural contour mechanisms: from simple to end-stopped cells," *Vision Research*, vol. 32, no. 5, pp. 963 – 981, 1992.

[20] C. Schmid, R. Mohr, and C. Bauckhage, "Evaluation of Interest Point Detectors," *Int. J. Comput. Vision*, vol. 37, pp. 151–172, June 2000.

[21] NVIDIA Corporation, *NVIDIA Jetson TK1 Development Kit Bringing GPU-accelerated computing to Embedded Systems*, April 2014.

[22] Altera, "All Development Kits." https://www.altera.com/products/boards_and_kits/all-development-kits.html.

[23] ARM Corporation, "Introducing NEON." https://software.intel.com/sites/default/files/m/b/4/c/DHT0002A_introducing_neon.pdf.

[24] D. B. Kirk and W. Mei W. Hwu, *Programming Massively Parallel Processors - A Hands-on Approach.* Morgan Kaufmann, 2010.

[25] N. Fujimoto, "Faster matrix-vector multiplication on Geforce 8800GTX," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, April 2008.

[26] Y. Fang, L. Chen, J. Wu, and B. Huang, "GPU Implementation of Orthogonal Matching Pursuit for Compressive Sensing," in *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, pp. 1044–1047, Dec 2011.

[27] T. Rauber and G. Rünger, *Parallel Programming*. Springer, 2010.

[28] M. Harris, "Unified Memory in CUDA 6." https://devblogs.nvidia.com/parallelforall/unified-memory-in-cuda-6/.

[29] NVIDIA Corporation, *Tunning Cuda Aplications for Kepler*, August 2014.

[30] P. L. Rosin, "Unimodal thresholding," *Pattern Recognition*, pp. 2083–2096, 2001.

[31] A. Neubeck and L. V. Gool, "Efficient Non-Maximum Suppression," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3, pp. 850–855, August 2006.

[32] C. A. R. Hoare, "Algorithm 64: Quicksort," *Commun. ACM*, vol. 4, pp. 321–, July 1961.

[33] D. E. Knuth, *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[34] K. E. Batcher, "Sorting Networks and Their Applications," in *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), (New York, NY, USA), pp. 307–314, ACM, 1968.

[35] V. Volkov, "Better performance at lower occupancy," in *GPU Technology Conference*, 2010.

[36] J. Luitjens, "Faster Parallel Reductions on Kepler." https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/.

[37] NVIDIA Corporation, "Histogram Calculation in CUDA." http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/histogram64/doc/histogram.pdf.

[38] P. Rodriguez, *Yupana*. Grupo de Procesamiento Digital de Señales e Imágenes (GPDSI-PUCP), 2014.

[39] C. Topal, K. Özkan, B. Benligiray, and C. Akinlar, "A robust CSS corner detector based on the turning angle curvature of image gradients." http://ceng.anadolu.edu.tr/CV/CornerDetection/default.htm.

[40] R. Phull, P. Mainali, Q. Yang, P. R. Alface, and H. Sips, "Low Complexity Corner Detector Using CUDA for Multimedia Applications," in *MMEDIA 2011, The Third International Conferences on Advances in Multimedia*, pp. 7–11, 2011.

[41] S. Luo and J. Zhang, "Accelerating Harris Algorithm with GPU for Corner Detection," in *Image and Graphics (ICIG), 2013 Seventh International Conference on*, pp. 149–153, July 2013.

[42] P. R. Possa, S. A. Mahmoudi, N. Harb, C. Valderrama, and P. Manneback, "A Multi-Resolution FPGA-Based Architecture for Real-Time Edge and Corner Detection.," *IEEE Trans. Computers*, vol. 63, no. 10, pp. 2376–2388, 2014.

[43] A. Amaricai, C. E. Gavriliu, and O. Boncalo, "An FPGA sliding window-based architecture harris corner detector," in *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pp. 1–4, Sept 2014.

[44] C. Claus, R. Huitl, J. Rausch, and W. Stechele, "Optimizing the SUSAN corner detection algorithm for a high speed FPGA implementation," in *Field Programmable Logic and Applications, 2009. FPL 2009. International Conference on*, pp. 138–145, Aug 2009.

[45] F. Hosseini, A. Fijany, and J.-G. Fontaine, "Highly Parallel Implementation of Harris Corner Detector on CSX SIMD Architecture," in *Proceedings of the 2010 Conference on Parallel Processing*, Euro-Par 2010, (Berlin, Heidelberg), pp. 137–144, Springer-Verlag, 2011.

[46] A. Glenis and S. Petridis, "Performance and energy characterization of high-performance low-cost cornerness detection on GPUs and multicores," in *IISA 2014, The 5th International Conference on Information, Intelligence, Systems and Applications*, pp. 181–186, July 2014.

[47] T. L. Chao and K. H. Wong, "An efficient FPGA implementation of the Harris corner feature detector," in *2015 14th IAPR International Conference on Machine Vision Applications (MVA)*, pp. 89–93, May 2015.

[48] H. Orabi, N. Shaikh-Husin, and U. U. Sheikh, "Low cost pipelined FPGA architecture of Harris Corner Detector for real-time applications," in *2015 Tenth International Conference on Digital Information Management (ICDIM)*, pp. 164–168, Oct 2015.

[49] Nvidia, "Jetson TX1 Developer Kit." `http://www.nvidia.com/object/jetson-tx1-dev-kit.html`.

[50] J. Shi and C. Tomasi, "Good features to track," tech. rep., Ithaca, NY, USA, 1993.

[51] J. A. Noble, "Finding corners," *Image and Vision Computing Journal*, pp. 2–121, 1988.

[52] T. Q. Pham, *Advanced Concepts for Intelligent Vision Systems: 12th International Conference, ACIVS 2010, Sydney, Australia, December 13-16, 2010, Proceedings, Part I*, ch. Non-maximum Suppression Using Fewer than Two Comparisons per Pixel, pp. 438–451. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.