INSTITUTO TECNOLÓGICO DE BUENOS AIRES – ITBA

ESCUELA DE INGENIERÍA Y GESTIÓN

Detección Automática y Rastreo en Tiempo Real de Objetos

AUTORES: Saqués, M. Alejo (Leg. N° 56047)

Marcantonio, Nicolás (Leg. N° 56288)

Benítez, Julián (Leg. N° 56283)

TUTORES: Dr. Rodrigo Ramele, Dra. María Juliana Gambini

TRABAJO FINAL PRESENTADO PARA LA OBTENCIÓN DEL TÍTULO DE INGENIERO EN INFORMÁTICA

Lugar: Buenos Aires, Argentina
Fecha: 14/08/2019

Índice

Índice de figuras

Detección Automática y Rastreo en Tiempo Real de Objetos

Autores: M. Alejo Saqués, Nicolás Marcantonio, Julián Benítez

Tutores: Dr. Rodrigo Ramele, Dra. María Juliana Gambini

Instituto Tecnológico Buenos Aires (ITBA)

Resumen

Shi *et al.* propusieron en 2005[1] una novedosa técnica para rastreo de múltiples objetos en tramas de vídeo basada en el concepto de conjuntos de nivel. Sin embargo, este algoritmo presenta falencias en lo que respecta a la expansión incontrolada del contorno de un objeto al entrar en contacto con una región no rastreada de similares características. Este trabajo propone una técnica basada en el análisis del centroide y radio de los objetos, a los efectos de fijar una cota superior a la expansión del contorno de las regiones.

Adicionalmente, los autores ratifican que su algoritmo es capaz de seguir en tiempo real a múltiples objetos. Se han realizado implementaciones en **Java 1.8** y **CUDA/C++** a los efectos de verificar dicha aseveración. Se ha comprobado la veracidad de dicho planteo, determinándose que la implementación en **Java 1.8** es superior a la segunda únicamente al rastrearse un bajo número de objetos, advirtiéndose la mayor escalabilidad de la implementación en GPU.

Por último, se propone una novedosa técnica para detectar automáticamente la posición inicial de objetos en movimiento, incluso en entornos con fondo cambiante basada en [5], [6] para la sustracción del fondo y [7] para obtener los contornos de los potenciales objetos.

Palabras clave: *level sets*, centroide, GPGPU

1. Introducción

La temática de la detección de bordes es de profundo interés en el campo de la visión artificial. Tan pronto como en 1968[2], Sobel y Feldman presentaron en *Stanford* su solución al problema, la cual se basa en estimaciones del gradiente del color alrededor de los bordes. Esta solución, conocida más tarde como **Operador de Sobel**, realiza convoluciones de dos máscaras 3×3 contra una imagen, para conseguir estimaciones de la derivada del color tanto en una dirección horizontal como vertical. Por ende, la conjetura es que, de no existir un borde, la magnitud de dicho gradiente debería tender a 0. Caso contrario, debe ser máximo.

Sin embargo, este operador (y otros basados exclusivamente en el análisis del gradiente) posee severas falencias, entre las cuales se encuentra su susceptibilidad al ruido en la imagen, a las pobres condiciones de iluminación y a la falta de contraste en los objetos. Al analizarse una imagen con este tipo de características bajo el operador de Sobel, a menudo se obtienen bordes indeseados inducidos por el ruido, otros poco definidos y entrecortados, o bien se los marca duplicados (esto es, bordes de más de 1 píxel de grosor).

Para confrontar algunos de estos problemas, Canny presentó en 1986 su *detector*[3], el cual busca mitigar algunas de estas falencias. Dicho detector incorpora las ideas del operador de Sobel, pero previamente a su aplicación, procura eliminar el ruido mediante un filtro gaussiano. Posteriormente, aplica técnicas para disminuir el grosor de los bordes, tales como la supresión de no máximos y el doble umbral, que trabajan sobre el valor del gradiente y su dirección. Esto logra una mejor aproximación a la solución, evitando el surgimiento de bordes espurios y el *aliasing* de los mismos.

No obstante, no soluciona todos los problemas que presentan los métodos clásicos basados en el gradiente. Por un lado, el detector de Canny no logra una completa segmentación de la imagen: tal como el resto de los métodos de este tipo, produce un conjunto de píxeles de borde y no un contorno definido y cerrado. Por otro lado, sigue siendo susceptible a problemas de contraste, a texturas rugosas y a variaciones de intensidad poco definidas. Más notablemente, trabajan con la imagen *como un todo*: no disciernen entre diferentes objetos dentro de una misma imagen. Dado el costo computacional de estos algoritmos, el no segmentar los cuadros dificulta utilizarlos en aplicaciones de tiempo real.

Existen técnicas de rastreo en vídeo basadas en métodos clásicos, tales como el *algoritmo de cuenca*[4]. Este algoritmo formula al problema como uno de minimización, basándose en una función invertida de la distancia de un mapa de bordes. Si bien este algoritmo es capaz de rastrear objetos en vídeos, no garantiza que los bordes sean cerrados, y es susceptible a capturar bordes espurios donde el gradiente es de gran magnitud.

La técnica de contornos de nivel propuesta por Shi *et al.* en 2005[1] confronta la tarea de detección de bordes mediante la idea de los *contornos activos*: la idea que se parte desde una curva inicial definida por el usuario, haciéndosela evolucionar iterativamente hasta haberse logrado abarcar todo el contorno del objeto con la curva, o bien al alcanzar un máximo de iteraciones. Por un lado, este tipo de técnicas logra generar curvas cerradas. Por otro, permite segmentar el análisis de la imagen, sin tener que analizarla por completo para detectar los bordes de un objeto. Además, ofrece versatilidad en la forma de definir el criterio para aceptar un píxel como parte de un objeto, y el rastreo de múltiples objetos es de fácil implementación.

Sin embargo, el algoritmo definido por los autores no presenta un mecanismo de detección automática de objetos. Precisa que un usuario especifique dónde se encuentra aproximadamente el objeto para luego expandirse hasta abarcar todo su borde. Uno de los objetivos de este trabajo es explorar técnicas a los efectos de proveer tal mecanismo automático, para aplicarse al seguimiento de objetos en una trama de vídeo.

Por otro lado, el algoritmo presentado no confronta una problemática que se evidencia notoriamente en el caso de análisis de tramas de vídeo. En este escenario, un objeto siendo rastreado se aproxima hasta rozar por un lapso de tiempo infinitesimal con otro objeto de similar color sin ser rastreado. En el momento exacto del roce, según el algoritmo presentado el borde debería extenderse lo más posible, así abarcando todo el contorno del objeto aledaño. Esto no solo puede que no sea deseado en una aplicación real, sino que tiene un profundo impacto en la *performance* del algoritmo. A los efectos de solucionar esta problemática, se explorarán alternativas a los efectos de impedir esta explosión de crecimiento del contorno, a los efectos de mitigar sus consecuencias negativas.

Por último, uno de los objetivos centrales de cualquier aplicación de rastreo de objetos en tramas de vídeo será que mantenga su tiempo de procesamiento por debajo del tiempo de aparición de nuevos cuadros, es decir, que sea en tiempo real. Se ha notado que los autores de [1] hacen pocas menciones a resultados numéricos de la ejecución del algoritmo, que respalden que el algoritmo pueda aplicarse en este escenario. Por ende, se buscará

comprobar dicha afirmación, explorando diversas implementaciones en CPU y GPU para evaluar su desempeño.

Este trabajo está compuesto de la siguiente forma:

- **Sección** 2: se realizará una breve descripción del algoritmo original, explicando sus fundamentos y el método para evitar colisiones entre objetos rastreados.
- **Sección** 3: se introducirá una mejora que busca limitar las modificaciones en la topología de los objetos rastreados.
- **Sección** 4: se presentará el algoritmo de detección automática de regiones iniciales.
- **Sección** 5: se discutirá sobre las diferentes implementaciones realizadas.
- **Sección** 6: se presentará una serie de pruebas realizadas, a los efectos de verificar el rendimiento de las implementaciones y la mejora inducida por la limitación del radio.
- Finalmente, en la sección 7 se cerrará con una serie de conclusiones y propuestas de continuación.

2. *Rastreo en Tiempo Real Utilizando Conjuntos de Nivel*

En líneas generales, se ha seguido el algoritmo propuesto en [1] para el seguimiento de múltiples objetos en un vídeo.

Para el caso del seguimiento de un único objeto, el algoritmo parte de una escena únicamente compuesta por el fondo (*background*) Ω_0 y el objeto Ω_1.

Para representar ambas regiones, se utiliza una función de conjunto de nivel ϕ, la cual es negativa en Ω_1 y positiva en Ω_0. Basándose en esta representación, los autores definen dos conjuntos de píxeles vecinos L_{in} y L_{out} del contorno C, tales que:

$$\begin{aligned} L_{out} = \{\, x \mid \phi(x) > 0 \text{ y} \\ \exists\, y \in N_4(x) \text{ tal que } \phi(y) < 0\} \end{aligned} \tag{1}$$

$$\begin{aligned} L_{in} = \{\, x \mid \phi(x) < 0 \text{ y} \\ \exists\, y \in N_4(x) \text{ tal que } \phi(y) > 0\} \end{aligned} \tag{2}$$

Donde $N_4(x)$ es el conjunto de 4-vecinos del píxel x.

A los fines de la implementación del algoritmo, esta representación prevé la utilización de una serie de estructuras de datos:

- Un arreglo para la función de conjunto de nivel ϕ;
- Dos arreglos para los conjuntos de píxeles: L_{in} y L_{out}.

Los valores del arreglo del conjunto de nivel son definidos de la siguiente forma:

$$\phi(x) = \begin{cases} +3 & \text{si } x \text{ es un píxel exterior} \\ +1 & \text{si } x \in L_{out} \\ -1 & \text{si } x \in L_{lin} \\ -3 & \text{si } x \text{ es un píxel interior} \end{cases} \tag{3}$$

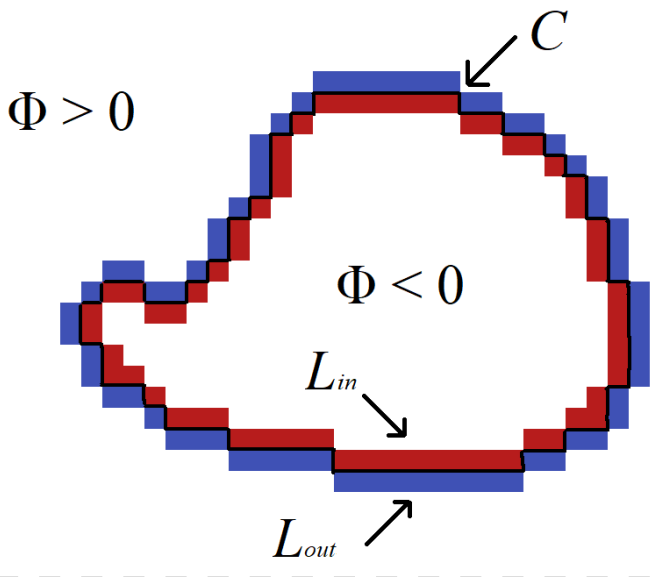


Figura 1: Representación de la curva C y de los conjuntos L_{in} y L_{out}.

Los autores en [1] utilizan un arreglo F de *velocidades* de los píxeles en L_{in} y L_{out}, precomputado en cada iteración de los ciclos, a los efectos de determinar la región a la pertenece en la iteración actual el píxel en cuestión. Si el píxel $p \in L_{out}$, y $F(p) = +1$, p debe pertenecer a L_{in}, tras lo cual existirán píxeles de L_{in} que se convirtieron en interiores. De manera similar, si $p \in L_{in}$ y $F(p) = -1$, el píxel debe ser trasladado a L_{out}, quedando píxeles previamente contenidos en L_{out} como exteriores.

En la versión del algoritmo implementada, una función invocada bajo demanda hace las veces del arreglo utilizado por los autores. Esto posibilita una reducción en la complejidad en memoria del algoritmo del orden de $O(Width \times Height)$, de considerarse que se utiliza un arreglo del tamaño de la imagen a los efectos de evitar la utilización de primitivas de sincronización. La función F utilizada es la que sigue:

$$f_{\Omega_i}(p) = \frac{\|\Theta_{\Omega_i} - \Theta_p\|}{MAX\_VALUE * \sqrt{3}} \tag{4}$$

$$F_{\Omega_i}(p, \mu) = \begin{cases} +1 & \text{si } f_{\Omega_i}(p) > \mu \\ -1 & \text{si } f_{\Omega_i}(p) < \mu \\ 0 & \text{si } f_{\Omega_i}(p) = \mu \end{cases} \tag{5}$$

Donde Θ_{Ω_i} es un vector promedio de las componentes de color de los píxeles de la zona inicial, Θ_p es el color del píxel p y MAX_VALUE es el máximo valor posible de una componente de color (255 en el caso de considerar valores en el rango [0, 255]). μ

es un umbral utilizado para variar la sensibilidad del algoritmo a variaciones del color. Nótese que f_{Ω_i} es máxima cuando Θ_p es nulo y Θ_{Ω_i} es máximo (sus tres componentes son MAX_VALUE).

Luego de esta iteración, se evalúa una condición de corte. Si es satisfecha, se finaliza el ciclo, caso contrario, se continúa el proceso iterativo. La condición de corte se define de la siguiente forma:

Condición de corte. *El algoritmo de evolución de la curva corta si cualquiera de las siguientes condiciones es satisfecha:*

(a) *La velocidad de cada píxel de la curva satisface:*

$$\begin{aligned} &F(x) \leq 0 \,\forall\, x \in L_{out}; \\ &F(x) \geq 0 \,\forall\, x \in L_{in}. \end{aligned} \tag{6}$$

(b) *Un número máximo de iteraciones especificado es alcanzado.*

Lo que la condición en (6) implica es que si todos los píxeles de L_{out} están en el fondo, y todos los píxeles de L_{in} están en el objeto, entonces se ha alcanzado un estado de convergencia. La segunda condición puede ser útil en casos en los que existe ruido en la imagen, y es necesario detener la ejecución para detener la evolución del contorno.

Este primer ciclo del algoritmo puede producir curvas de naturaleza áspera. A los efectos de evitar esto, y el potencial impacto en la eficiencia del algoritmo de una curva de mayor perímetro, los autores proponen un ciclo de alisamiento de la curva generada en el primer ciclo, utilizando filtros Gaussianos.

Se denota un filtro Gaussiano isotrópico de dimensiones $N_g \times N_g$ como G. Para alisar el contorno obtenido en el conjuntos de nivel del paso anterior, se evalúa la respuesta de la función ϕ frente a G, únicamente en los píxeles de L_{in} y L_{out}. Únicamente cuando la respuesta es diferente al valor de ϕ, se mueve un píxel de un conjunto a otro o viceversa, así alterando el valor de $\phi(x)$. Caso contrario, los valores de dicha matriz permanecen inalterados.

A modo de resumen, se presenta el algoritmo en dos pasos:

- **Paso 1:** Inicializar ϕ, L_{in} y L_{out} dada una región inicial.
- **Paso 2:** Primer ciclo. Para un número máximo de iteraciones N_a, hacer:
  - Para cada píxel $x \in L_{out}$, si $F(x) > 0$ luego mover x a L_{in} e intercambiar los valores de ϕ según (3). Luego, $\forall\, y \in N_4(x)$ tal que $\phi(y) = 3$, mover y a L_{out} y fijar $\phi(y) = 1$.
  - Para cada píxel $x \in L_{in}$, si $\forall\, y \in N(x)$, $\phi(y) < 0$, borrar x de L_{in} y fijar $\phi(x) = -3$. x es interior.
  - Para cada píxel $x \in L_{in}$, si $F(x) < 0$ luego mover x a L_{out} e intercambiar los valores de ϕ según (3). Luego, $\forall\, y \in N_4(x)$ tal que $\phi(y) = -3$, mover y a L_{in} y fijar $\phi(y) = -1$.
  - Para cada píxel $x \in L_{out}$, si $\forall\, y \in N(x)$, $\phi(y) > 0$, borrar x de L_{out} y fijar $\phi(x) = 3$. x es exterior.

- Verificar la condición de corte. Si es satisfecha, proceder al paso 3. Caso contrario, volver a ejecutar el presente ciclo.

- **Paso 3:** Segundo ciclo. Para un número máximo de iteraciones N_g, hacer:

  - Para cada píxel $x \in L_{out}$, calcular $G \otimes \phi(x)$. Si $G \otimes \phi(x) < 0$, mover x a L_{in} e intercambiar los valores de ϕ según (3). Luego, $\forall\, y \in N_4(x)$ tal que $\phi(y) = 3$, mover y a L_{out} y fijar $\phi(y) = 1$.
  - Para cada píxel $x \in L_{in}$, si $\forall\, y \in N(x)$, $\phi(y) < 0$, borrar x de L_{in} y fijar $\phi(x) = -3$. x es interior.
  - Para cada píxel $x \in L_{in}$, calcular $G \otimes \phi(x)$. Si $G \otimes \phi(x) > 0$, mover x a L_{out} e intercambiar los valores de ϕ según (3). Luego, $\forall\, y \in N_4(x)$ tal que $\phi(y) = -3$, mover y a L_{in} y fijar $\phi(y) = -1$.
  - Para cada píxel $x \in L_{out}$, si $\forall\, y \in N(x)$, $\phi(y) > 0$, borrar x de L_{out} y fijar $\phi(x) = 3$. x es exterior.

2.1. Rastreo simultáneo de múltiples objetos

En aplicaciones prácticas, se puede prever la necesidad de rastrear simultáneamente multiplicidad de objetos, tanto de diferentes colores como del mismo color. El algoritmo tal como se lo ha presentado hasta el momento es, en efecto, capaz de rastrear múltiples objetos simultáneamente − será solo cuestión de realizar n ciclos independientes uno del otro por cada objeto que se desee seguir en una trama. El problema ocurre en los casos en los que dichos objetos entran en contacto, y se exacerba cuando los mismos son de similar o idéntico color.

Para el caso que los objetos sean de colores suficientemente diferentes ($\Theta_{\Omega_i} \neq \Theta_{\Omega_j}$), y para un μ apropiado, $f_{\Omega_i}(p) \neq f_{\Omega_j}(p)$. Esto es, esencialmente, lo mismo que sucede para el caso en el que se tiene que distinguir un píxel dentro del objeto de otro en el fondo de la imagen.

Sin embargo, para el caso que los colores de los objetos sean similares, o que μ sea excesivamente permisivo, puede ocurrir que $f_{\Omega_i}(p) = f_{\Omega_j}(p)$. Esto provocará que los contornos de los respectivos objetos se expandan hasta alcanzar los bordes del otro. Esta situación perdurará incluso luego de que dichos objetos se distancien, efectivamente duplicándose el contorno rastreado por cada objeto.

Para evitar este tipo de situaciones, se define una función para determinar si un píxel evaluado pertenece a la región interna de otro objeto siendo rastreado:

$$T_{\Omega_i}(p) = \begin{cases} 1 & \text{si } \exists\, o \in N_8(p)\,, \exists\, \Omega_j \mid \phi_{\Omega_j}(o) < 0 \\ 0 & \text{si no} \end{cases} \tag{7}$$

De esta forma, cada vez que se intente mover un píxel p a L_{in} se debe verificar que $T_{\Omega_i}(p) = 0$. Caso contrario, se debe abortar dicha operación.

3. Prevención del contagio a objetos aledaños mediante limitación del radio

En la sección 2, se ha presentado el algoritmo propuesto por los autores en [1], con algunas sutiles diferencias en lo que respecta a la notación y a su implementación, con resultados similares a los expuestos en el trabajo de referencia.

Sin embargo, en dicho trabajo los autores no confrontan una de las situaciones en las que su algoritmo presenta serias falencias. Dicha situación se presenta a continuación:

Figura 2: Secuencia del contagio de un objeto de similar color no rastreado

En la figura 2 se puede observar la secuencia completa de la situación que se busca evitar. En un cuadro z, una única región Ω está siendo rastreada por el algoritmo, mientras un objeto de similar (o idéntico) color se encuentra en su proximidad. Podría suceder que uno de los objetos se solape con el otro por un lapso de tiempo infinitesimal (del orden de los 2 a 5 cuadros), generando una modificación en la topología del objeto origina.

El algoritmo presentado no tiene ningún tipo de restricción respecto de cuánto puede crecer en tamaño el contorno rastreado, lo que puede conducir a situaciones en las que el contorno crece de manera ilimitada. En el cuadro $z+1$, al ser ambos objetos del mismo color, la función $f_{\Omega(p)}$ no tiene más que aceptar a un píxel negro p como un *píxel* de la Ω inicial, generando una masiva expansión de los conjuntos de niveles L_{in} y L_{out} en el lapso de un único cuadro. Esto puede tener un impacto imperceptible de ser pequeñas las regiones involucradas, pero de producirse el caso en que el objeto espurio es de ordenes de magnitud más grande que el rastreado hasta el momento, los tiempos de procesamiento para los cuadros subsiguientes pueden verse severamente afectados.

En el cuadro $z+2$, el objeto indeseado se aleja de la región rastreada inicialmente, pero ahora en efecto forma parte de la misma región. Esta propiedad del algoritmo presentado en 2 constituye una modificación en la topología del objeto. En ciertas aplicaciones, esta propiedad puede ser de interés, pero al seguir un mismo objeto en un vídeo es preciso eliminarla.

A continuación, se introduce un mecanismo diseñado a los efectos de mitigar el impacto en la *performance* de tales situaciones. Una manera de limitar el máximo tamaño que una región siendo rastreada puede alcanzar en los sucesivos cuadros de la trama es acotar el radio máximo de expansión de la curva, tal como se presentará a continuación.

3.1. Limitación del radio máximo desde el centroide

El objetivo de esta mejora es limitar mediante un parámetro qué tanto puede crecer el radio de una región Ω_i en un cuadro z, respecto de su centroide en el cuadro anterior $(z-1)$ y de su radio. Para ello, se redefine la función F_{Ω_i} definida en 5:

$$F^I_{\Omega_i}(p, \mu, c, r, \rho) = \begin{cases} 0 & \text{si } \|c - p\| > r(1+\rho) \\ F_{\Omega_i}(p, \mu) & \text{sino} \end{cases} \tag{8}$$

Donde c es el centroide calculado a partir de la curva L_{in} en el cuadro $z-1$, r es el radio del objeto y ρ es el *ratio de expansión* del radio.

Lo que fundamentalmente hace la función $F^I_{\Omega_i}$ es definir una región circular de radio r centrada en el centroide de Ω donde F_{Ω_i} puede retornar valores distintos a 0. Fuera de dicha región, su único valor de retorno será el nulo. Luego se redefinen los ciclos presentados en la sección 2, intercambiando a F por su redefinición F^I. La importancia de que F^I devuelva 0 en estos casos es que dicho valor es el único que evitará que un *píxel* p no sea ni desde o hacia L_{in} o L_{out} o viceversa.

Hasta el momento se han presentado los valores de r el radio y c el centroide sin mayores especificidades. Sin embargo, es preciso realizar una serie de aclaraciones sobre su cómputo, abordadas en la sección que sigue.

3.1.1. Cómputo del centroide y del radio

Al redefinir la función F, se ha mencionado que su redefinición F^I toma como parámetros dos variables c y r, representando al centroide y al radio de la región Ω respectivamente. Sin embargo, no se han provisto precisiones respecto de cuándo y cómo se computan dichos valores. Para ello, es preciso definir los conceptos de *etapa de preparación* e *etapa de procesamiento*.

La *etapa de preparación* para una región Ω_i está comprendida por aquellos cuadros tales que $z_i < P_z$. De idéntica forma, la *etapa de procesamiento* está comprendida por todos aquellos cuadros tales que $z_i \geq P_z$.

De esta forma, el cálculo del centroide se realiza de la siguiente forma:

$$c(L_{in}) = \sum_{p \in L_{in}} \frac{p}{size(L_{in})} \tag{9}$$

$$C(L_{in}, z, c_{z-1}, \xi) = \begin{cases} c(L_{in}) & \text{si } z < P_z \\ c(L_{in})(1-\xi) + c_{z-1}\,\xi & \text{sino} \end{cases} \tag{10}$$

Donde z es el cuadro correspondiente al L_{in} parámetro, c_{z-1} son las coordenadas del último centroide computado, y ξ define una proporción del centroide previo a considerar en el nuevo valor computado. Este valor ξ, idealmente del orden del 5 %, es una de las aproximaciones a evitar que el objeto varíe de radio en demasía, al generar una cierta *resistencia* del centroide a cambiar de posición exageradamente. Esto se complementa con

la limitación del radio de la región: de haberse producido un cambio indeseado producto del procesamiento en un cuadro, este parámetro ξ permite compensar parcialmente dichas pérdidas de fidelidad.

Por otro lado, el cálculo del radio únicamente se realiza en la *etapa de preparación*, de esta forma estableciendo efectivamente un radio máximo a alcanzar por el objeto. De realizarse este cálculo en todo cuadro, el incremento del radio sería exponencial, sin poderse evitar esta situación incluso con valores de ρ pequeños. A modo de ejemplo, si $\rho = 1\,\%$, luego de 30 cuadros (o 1 segundo), el máximo incremento posible es del orden del 35 %. En 2 segundos, del 82 % y en 3 segundos, del 245 %.

Así pues, el radio r se computa de la siguiente forma:

$$\begin{aligned} & r(c, r_{z-1}, L_{in}) \\ = \text{máx}(r_{z-1}, & \max_{r}(\| p - c \| \mid p \in L_{in})) \end{aligned} \tag{11}$$

Donde c es el centroide recientemente computado, r_{z-1} es el radio computado en la iteración anterior y L_{in} el conjunto de nivel interior de dicha iteración. Esencialmente, el resultado de realizar estas comparaciones durante toda la *etapa de preparación* será que el radio a utilizar en la etapa siguiente sea efectivamente el máximo radio alcanzado durante la etapa de preparación.

4. Detección automática de regiones iniciales

El procedimiento para señalar las posiciones de las regiones iniciales puede ser configurado para detectar automáticamente objetos en un cuadro, o bien tener regiones iniciales fijas. Existen dos pasos cruciales a la hora de detectar automáticamente las regiones iniciales: la sustracción del fondo y el procesamiento de las diferencia.

4.1. Sustracción del fondo

Aquí el objetivo es marcar los píxeles del cuadro actual que cambiaron su color de una manera significativa. Para ello se utiliza un algoritmo de OpenCV llamado MOG2 que modela cada píxel como una combinación de filtros gaussianos, representando el peso de cada filtro la proporción del tiempo que esos colores están en escena. La cantidad de filtros gaussianos es determinada automáticamente por el algoritmo. Esto causa que MOG2 se pueda adaptar a cambios del fondo, y en menor medida, a cambios de iluminación, pero lo que caracteriza a los cambios en el fondo es que son lentos en comparación a los cambios producidos por el ingreso de nuevos objetos. Por lo tanto, un nuevo objeto entrando en escena se mueve demasiado rápido para que MOG2 se adapte a el, y sera marcado como una diferencia con el fondo. MOG2 fue implementado por OpenCV basándose en [5] y [6].

Al algoritmo MOG2 se le configura con dos parámetros, la *sensibilidad* y el *tamaño del historial*. La sensibilidad determina la susceptibilidad del algoritmo a los cambios en el fondo. Una sensibilidad muy baja no se adapta bien a cambios de fondo o iluminación, y con una sensibilidad alta, habría objetos que, al moverse muy lento, no serian detectados como cambios, dado que MOG2 se adapta al color del objeto antes que sea detectado.

En la siguiente sección se discuten los pasos utilizados para detectar automáticamente las regiones iniciales.

4.2. Pasos del algoritmo

1. Input:
   - *areaMinima* Área mínima que tiene que tener un objeto nuevo para ser incorporado al algoritmo principal.
   - *distanciaMinima* Distancia mínima entre el centroide del objeto y el objeto ya rastreado mas cercano.
   - *listaCentroides* Lista de los centroides de objetos ya siendo rastreados.
   - *objMaximos* La cantidad máxima de objetos a retornar
2. Pasar el cuadro a HSV y tomar el canal H.
3. Pasar un filtro de mediana de 5x5
4. Aplicar la sustracción de fondo como fue descripta anteriormente y obtener los píxeles con cambios relevantes
5. Aplicar otra vez un filtro de mediana de 5x5
6. Aplicar el algoritmo para hallar contornos según [7] y obtener la lista de contornos, denominados **objetos potenciales**.
7. Para cada uno de los objetos potenciales, calcular el área y el centroide.
8. Filtrar los objetos potenciales:
   - Descartar los objetos potenciales que tienen área menor a un tamaño predefinido por el usuario.
   - Descartar los objetos potenciales que tienen el centroide muy cerca a un centroide de un objeto ya existente.
   - Ordenar la lista de objetos potenciales en manera descendiente según su área y quedarse solo con los *objMaximos* con mas área.
9. Centrado en el centroide del objeto, agregar un nuevo objeto en el algoritmo de contornos activos con tamaño proporcional al área del objeto detectado y agregar al objeto en la *listaCentroides*.

5. Implementaciones

En esta sección, se discutirán las diferentes implementaciones del algoritmo realizadas, haciendo énfasis en las mejoras realizadas a los efectos de lograr un mejor rendimiento.

En primer lugar, se elaborará brevemente sobre la implementación en Java 1.8, la cual más adelante servirá como muestra de control frente a la implementación en unidades de procesamiento gráfico (GPUs, por sus siglas en inglés).

Se hará especial énfasis sobre el último punto, señalando los aspectos arquitecturales de las GPUs, ejemplos de utilización del lenguaje **CUDA/C++**, como así también una explicación pormenorizada de los detalles de la implementación producida.

5.1. Java 1.8

Como se ha señalado más arriba, la primera implementación fue realizada en la plataforma Java 1.8, es decir, ejecutando el algoritmo utilizando la CPU. Esta versión, al servir únicamente de control para evaluar la *performance* de la implementación en GPU, no provee detección automática de objetos, verificación de colisiones ni limitación del radio del objeto según el algoritmo descripto mas arriba.

5.1.1. Implementación

La clase `Image` concentra toda la lógica del algoritmo descripto más arriba, siendo su punto de entrada la función:

```
public void activeContours(boolean parallel, RegionFeatures
    features, double threshold, BiFunction<Integer, Integer,
    BooleanMatrix2D> matrixProvider)
```

Donde:

- `parallel` es un *flag* que permite activar/desactivar la ejeución en paralelo
- `features` es una clase que describe a la región
- `threshold` es el umbral definido en la fórmula 5
- `matrixProvider` es una función que permite parametrizar la creación de la estructura para almacenar los elementos a incorporar o remover de los conjuntos de nivel.

`RegionFeatures` es una clase utilizada tanto en el modelado en **Java 1.8** como en su contraparte de **CUDA/C++**. Permite una simple extensión del algoritmo de rastreo de un objeto a uno con N objetos en simultáneo. Esencialmente, esta clase contiene los siguientes atributos y métodos:

```
public class RegionFeatures{

    private int[] phi;
    private Vector3D objAvg;
    private int width, height;
    private BooleanMatrix2D lin, lout;

    public static RegionFeatures buildRegionFeatures(Image img,
       int x0, int y0, int x1, int y1){
        ...
    }
}
```

Que incluyen:

- `int[] phi` el arreglo de tamaño $Width \times Height$ descripto en 3
- `Vector3D objAvg` el promedio del color del objeto rastreado en sus tres canales (R, G, B)

- `BooleanMatrix2D lin, lout` los conjuntos de nivel
- `buildRegionFeatures` la función de inicialización, que toma la imagen y los extremos superior izquierdo e inferior derecho y computa los campos descriptos.

La clase `BooleanMatrix2D` es esencialmente una *façade* de una estructura subyacente, que implementa métodos de carácter booleano tales como *and* y *or*, y un método *set* para fijar un valor en una coordenada dada. Esta estructura, única para la implementación en **Java 1.8**, ha sido útil a la hora de evaluar diferentes tipos de datos según su utilidad en un entorno concurrente. La implementación utilizada ha sido la siguiente:

```
public class BitSetMatrixPool implements BooleanMatrix2D {

    private int width, height;

    private BitSet[] mat;
    private ReentrantLock[] locks;
    private int size = java.util.concurrent.ForkJoinPool.
       getCommonPoolParallelism();

    ...
}
```

Esencialmente, esta implementación genera un par de `BitSet` y `ReentrantLock` por cada *working thread* dentro del *pool* utilizado por `BaseStream.parallel()`. La operación *set* intenta secuencialmente adquirir un *lock* y fija el valor del primer `BitSet` al que logra entrar. Luego, las operaciones *booleanas* descriptas más arriba se realizan por cada `BitSet` de esta estructura.

Luego, cada uno de los pasos de los ciclos presentados en 2 es esencialmente una función ejecutada paralelamente para cada elemento del conjunto de nivel correspondiente L_{in} o L_{out}. Estas funciones reciben `BooleanMatrix2D`, donde establecen las coordenadas de los píxeles a agregar o remover de dichos conjuntos de nivel de una manera *thread-safe*. Luego, operaciones sincrónicas remueven o agregan de los conjuntos de nivel contenidos en `RegionFeatures` los píxeles indicados, para luego proceder al siguiente paso. Este proceso se repite hasta cumplirse la condición de corte definida más arriba, o, en su defecto, hasta alcanzarse un máximo de iteraciones predefinido.

5.2. CUDA/C++

La implementación final del algoritmo ha sido realizada en **CUDA/C++**, utilizando el concepto GPGPU, que prevé la utilización de la arquitectura masivamente paralela de las unidades de procesamiento gráfico para aplicaciones de índole general.

El concepto GPGPU se basa en la implementación de funciones conocidas como *kernels*. Estos *kernels* son unidades de código ejecutadas paralelamente por unidades de procesamiento de alto *throughput* como las GPUs y que, a discreción del programador, pueden ser aplicadas a unidades de datos tales como a los píxeles de una imagen, partículas en un sistema de masas, entre otros casos.

Es preciso señalar que este tipo de implementaciones es ideal en casos en los que se debe realizar un pequeño número de cómputos en una gran cantidad de datos mutuamente independientes. Esto se debe a que las unidades de procesamiento gráficas generalmente

priorizan una cantidad masiva de núcleos de procesamiento en detrimento de la velocidad de *clock* de cada uno. Además, ciertas optimizaciones presentes en unidades de cómputo tradicionales, tales como la predicción de saltos (*branch prediction*), suelen no ser implementadas en los núcleos de la GPU. De esta forma, se deberá procurar evitar programas con una alta complejidad ciclomática.

Siguiendo en las líneas de la temática GPGPU, se puede comenzar por introducir algunos aspectos arquitecturales básicos del lenguaje CUDA y de la declaración e invocación de *kernels* en dicho lenguaje. A continuación se presenta una declaración de ejemplo (un *kernel* que dibuja en un cuadro los contornos de un objeto siendo seguido):

```
__global__ void drawFeaturesKernel(RegionFeatures * features, cv
    ::cudev::GpuMat dst, int bpp, dim3 color);
```

En esta función, la única palabra clave proveniente de CUDA es la palabra `__global__`, que indica que la función puede únicamente ser invocada desde el *host*, y que ejecutará en el *device*. Es decir, la función será invocada desde el CPU (*host*) para ejecutar en la GPU (*device*). Debe notarse que, en la función señalada, el puntero a `RegionFeatures` debe residir en la memoria global de la GPU (*device*), por lo cual el programador deberá encargarse de copiar información de estructuras entre *host* y *device* y viceversa.

Ahora bien, el lector podrá preguntarse cómo se realiza esta invocación desde el CPU. Un *kernel* en CUDA se ejecuta de la siguiente forma:

```
drawFeaturesKernel<<<Dg,Db,Ns,S>>>(features_d, dst, bpp, color);
```

Donde:

- `Dg (dim3)` especifica las dimensiones de la **grilla (grid)**
- `Db (dim3)` especifica las dimensiones del **bloque (block)**
- `Ns (size_t)` especifica la cantidad de *bytes* a asignar dinamicamente en la memoria compartida *por bloque*.
- `S (cudaStream_t)` especifica el *stream* asociado

Para este caso en particular, los argumentos relevantes son los primeros dos, ya que no se está realizando una invocación mediante *streams*, ni se está utilizando memoria compartida a nivel del *block*. Como se detallará más adelante, el tercer argumento ha sido extensamente utilizado a los efectos de optimizar los accesos a la memoria global de la GPU.

A continuación sigue una gráfica que ejemplifica los conceptos elaborados más arriba:

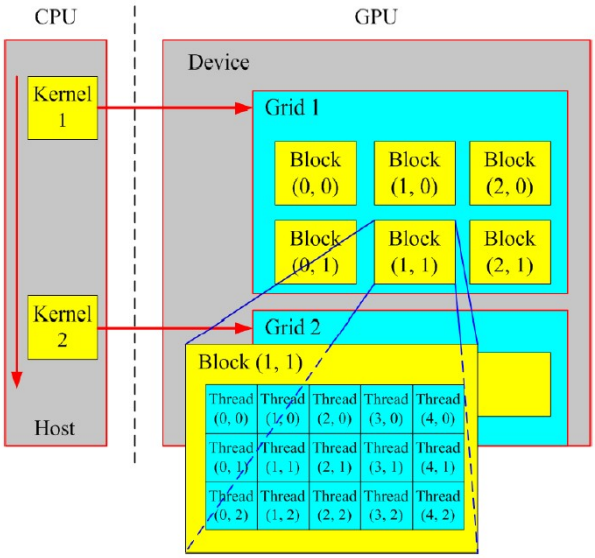


Figura 3: Esquema de invocación de *kernels*

Como se puede ver en la Figura 3, desde el CPU se invoca un *kernel*, el cual invoca de manera bloqueante en el GPU. En el GPU, dicho *kernel* es ejecutado por todos los **threads** contenidos en todos los **blocks** de la **grid**, y cada **thread** tiene información distintiva que le permite ejecutar sobre una unidad de información:

- `blockDim (dim3)` especifica las dimensiones del **block**
- `blockIdx (dim3)` especifica la posición del **block** dentro de la **grid**
- `threadIdx (dim3)` especifica la posición del **thread** dentro del **block**

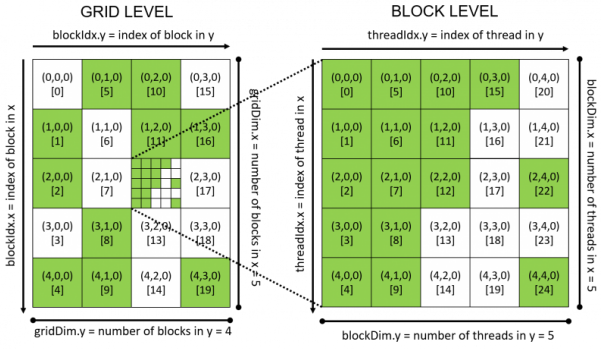


Figura 4: Ejemplo de dimensiones

Con esta información, es posible calcular los índices `x` e `y` sobre los cuales ejecutará un *kernel* en un thread dado:

```
int x = blockIdx.x*blockDim.x + threadIdx.x;
int y = blockIdx.y*blockDim.y + threadIdx.y;
```

Luego será solo cuestión de aplicar las operaciones deseadas. En la siguiente sección, se elaborará sobre algunos detalles de la implementación del algoritmo en este lenguaje.

Se invita al lector a consultar la documentación de **CUDA/C++** para mayores detalles sobre el modelo de programación y buenas prácticas de uso [1].

5.2.1. Implementación

Uno de los mayores desafíos a la hora de realizar la implementación de este algoritmo en un ambiente masivamente paralelo es garantizar la integridad de los datos, minimizando la cantidad de operaciones bloqueantes utilizadas a tal efecto. Para ello, se ha explorado una serie de caminos, detallados a continuación.

Como se ha detallado en el apartado 5.1, el problema central de una implementación concurrente de este algoritmo es el diseño de un mecanismo mediante el cual los diferentes *threads* puedan indicar los elementos de los conjuntos de nivel a añadir o remover en un paso dado. En dicho caso, se diseñó un *pool* de `BitSet`, de tal forma que existiese uno de dichos elementos por cada *thread* del que dispusiese la CPU. Esto aprovecha la eficiencia de las operaciones booleanas para realizar las inserciones o remociones de los conjuntos de nivel. En el caso de la GPU, considerándose que existen de cientos a miles de unidades de procesamiento, esta alternativa no pudo ser considerada.

Escritura simultánea y reducción La primera aproximación consistió en disponer de dos arreglos de dimensiones presumidas infinitas (acotados por $Width \times Height$) por cada conjunto de nivel. Uno de ellos, el primario, efectivamente contiene a los píxeles del conjunto de nivel, mientras que el otro (secundario) actúa de reserva para copiar los valores válidos del conjunto primario y tomar su lugar luego de la ejecución de alguno de los pasos del algoritmo en 2. El procedimiento sigue como se detalla a continuación:

```
#define BLOCK_SIZE 128

...

Pixel * lin = ref1;
Pixel * reserveLin = ref2;
Pixel * lout = ref3;
Pixel * reserveLout = ref2;

kernel<<<effectiveSize(lout)/BLOCK_SIZE, BLOCK_SIZE>>>(lout, lin
   );

fillZeroes(reserveLin);
compact(lin, reserveLin, compactionPredicate);
swap(lin, reserveLin);

...
```

En el ejemplo, el `kernel` toma los píxeles de L_{out} para luego depositarlos en *algún lugar* de L_{in} de cumplirse alguna condición (como sucede en el primer paso del ciclo definido en 2). Luego de ejecutarse, se invoca una función `fillZeroes`, que prepara el arreglo de reserva para recibir los datos compactados. Una función `compact` toma el par de arreglos para un conjunto de nivel y un predicado, y luego de realizada dicha operación, se intercambian las referencias. Nótese que los procedimientos `fillZeroes`

[1] `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html`

y `compact` han sido implementados con una librería de algoritmos paralelos en CUDA llamada **Thrust**.

Toda esta lógica se basa sobre la definición de una serie de **segmentos** en los arreglos, es decir, rangos de memoria en los cuales se puede depositar determinados elementos. Siendo L alguno de los conjuntos de nivel de N píxeles, se definen los siguientes **segmentos**:

1. **Segmento primario:** $[0,\ N-1]$. Donde se encuentran los píxeles propios de L.
2. **Segmento secundario:** $[N, N*(1+MAX\_PER\_PIXEL)-1]$. Para los pasos 1 y 3 de los ciclos definidos en 2, este segmento se utiliza cuando se debe escudriñar los cuatro vecinos de un píxel x para determinar a cuáles de ellos se los debe agregar a L.
3. **Segmento terciario:** $[N*(1+MAX\_PER\_PIXEL), +\infty]$. Segmento en donde agregar píxeles a L según la primera verificación de los pasos 1 y 2 del algoritmo.

Este mecanismo es, en teoría, adecuado para las necesidades de la implementación. Sin embargo, la necesidad de utilizar la función `fillZeroes` para purgar el arreglo de valores indeseados repercutió fuertemente en la *performance* de esta implementación. De no eliminarse los valores preexistentes, la compactación puede incluir valores no deseados (por ejemplo, píxeles repetidos), lo que incrementa exponencialmente el tamaño de L y, en consecuencia, el tiempo de cómputo. Por ende, se han explorado otras alternativas.

Escritura a *caché* y *lock* a nivel de bloque La alternativa utilizada para la versión final del algoritmo consistió de una utilización extensiva de la memoria *caché* a nivel de bloque como almacenamiento temporal de los píxeles a mover a la memoria global. El procedimiento se describe a continuación:

```
__global__ void kernel(RegionFeatures * features, int * linAdd,
    ...) {

    extern __shared__ Pix s[];

    int x = blockIdx.x*blockDim.x + threadIdx.x;

    if (...) {
        Pix p = features->lout[x];

        s[threadIdx.x].x = p.x;
        s[threadIdx.x].y = p.y;
        features->setPhi(p.x, p.y, -1);

        loopUnroll(features, p, 3, 1, loutAdd, features->lout);
    }

    __syncthreads();
    if (threadIdx.x == 0)
        sendToGlobalMemory(s, features, features->lin, features
            ->sizeLout, blockDim.x, blockIdx.x, threadIdx.x,
            linAdd, -1);
```

```
}
```

En este extracto (una simplificación del código para el primer paso del ciclo), se define un `extern __shared__ Pix s[]`. La palabra clave `__shared__` indica que esta memoria es *caché*, y `extern` indica que es asignada dinámicamente (mediante el parámetro de invocación de *kernels* `Ns (size_t)` indicado más arriba). Luego, `x` es el índice sobre el que está ejecutando el *kernel*.

Dentro del bloque del `if` se realiza toda la lógica propia del algoritmo de contornos activos, asignando los píxeles a mover de un conjunto de nivel al otro en la memoria *caché*. Al completar este procedimiento, se coloca a todos los *threads* del bloque a la par mediante la instrucción `__syncthreads()`. Luego, únicamente el *thread* con índice 0 en el bloque es el encargado de copiar la memoria *caché* a la global, así utilizándose un único conjunto de *locks* por bloque.

Esta alternativa probó ser la de mejor *performance* de las exploradas, destacándose el uso extensivo de memoria *caché*. Debe notarse que los accesos a dicha memoria son de órdenes de magnitud más veloces que los accesos a memoria global, por lo que su uso debe ser priorizado. Otras optimizaciones incluyen el desenrollado de ciclos, notablmente en los casos en los que se debe explorar los vecinos de un píxel x.

6. Resultados

En esta sección, se exhibirán los resultados de las pruebas realizadas, las cuales buscan evaluar el rendimiento de tanto las implementaciones del algoritmo como de las mejoras introducidas.

La primera prueba buscará evaluar el rendimiento de la implementación en **Java 1.8** contra su contraparte en **CUDA/C++**. El primero de los objetivos de esta prueba será determinar la viabilidad de este algoritmo para aplicaciones en tiempo real. En segunda instancia, se buscará probar si efectivamente la implementación en GPU supera en todo caso a la implementación en GPU. Como nota adicional, ciertas funcionalidades fueron desactivadas en la implementación en GPU (tales como la detección automática de objetos y la limitación del radio), a los efectos de equiparar ambas implementaciones.

Una segunda prueba consistió en analizar el desempeño de la funcionalidad para limitar el radio máximo desde el centroide. Para esto, se simuló un escenario en el cuál a un objeto fijo siendo rastreado se le aproxima y aleja rápidamente otro objeto de igual color y similares dimensiones. Se ejecutó un número de pruebas con la funcionalidad activada y desactivada para evaluar su impacto en el desempeño.

6.1. Verificación del tiempo real y comparación del rendimiento de las implementaciones

Se plantearon 3 casos de prueba a analizar, variando la cantidad de objetos a seguir, procurando que el fondo fuese uniformemente iluminado y altamente contrastante con los objetos rastreados. El primer caso, que puede observarse en la figura 5, es sólo un objeto a seguir, en el segundo caso, que muestra la figura 6, son 3 objetos a seguir y finalmente el último caso, que se observa en la figura 7, son 5 objetos a seguir.

Algo que vale la pena aclarar es que se trató de llevar al límite ambas implementaciones en función al número de objetos a seguir. Por ende, cada caso sigue un patrón diferente

de movimientos, siendo algunos más bruscos que otros. Por ende, no será posible una comparación *vis-à-vis* entre los diferentes casos de prueba.

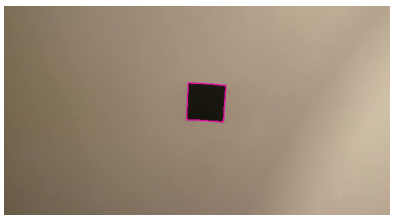


Figura 5: Primer caso: 1 objeto

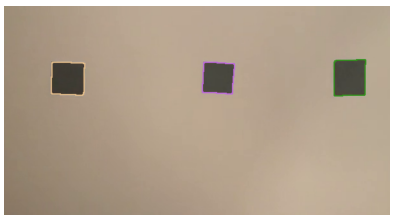


Figura 6: Segundo caso: 3 objetos

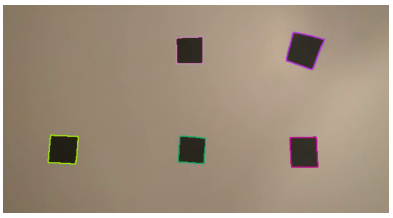


Figura 7: Tercer caso: 5 objetos

Para realizar esta comparación se propuso medir el tiempo de procesamiento por cuadro en ambas implementaciones, y contrastarlos en un mismo gráfico, graficando asimismo la función constante:

$$F(x) = \frac{1000}{24}$$

Que representa el umbral de tiempo real (24 cuadros por segundo).
En cuanto al *hardware*, se utilizó la siguiente configuración:

- 16GB RAM DDR4
- CPU Intel Core i7-6700HQ 5.6GHz
- GPU NVIDIA GeForce GTX 960M (4GB)

Para cada caso de prueba se filmó un vídeo y se realizaron 3 ejecuciones para cada implementación, por lo que los gráficos muestran el tiempo de procesamiento promedio por cuadro con sus respectivas barras de error.

A continuación se muestran los gráficos correspondientes a los 3 casos de prueba que se ven en las figuras 8, 9 y 10 respectivamente y en la figura 11 se grafica el tiempo promedio de procesamiento por cuadro para todos los casos.

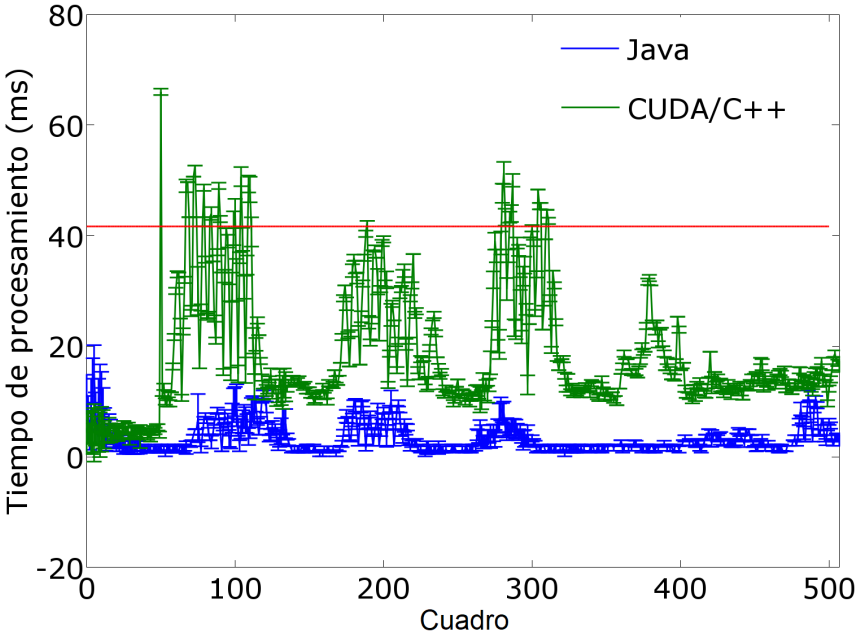


Figura 8: Tiempo de procesamiento para todos los objetos en un cuadro del primer caso de prueba.

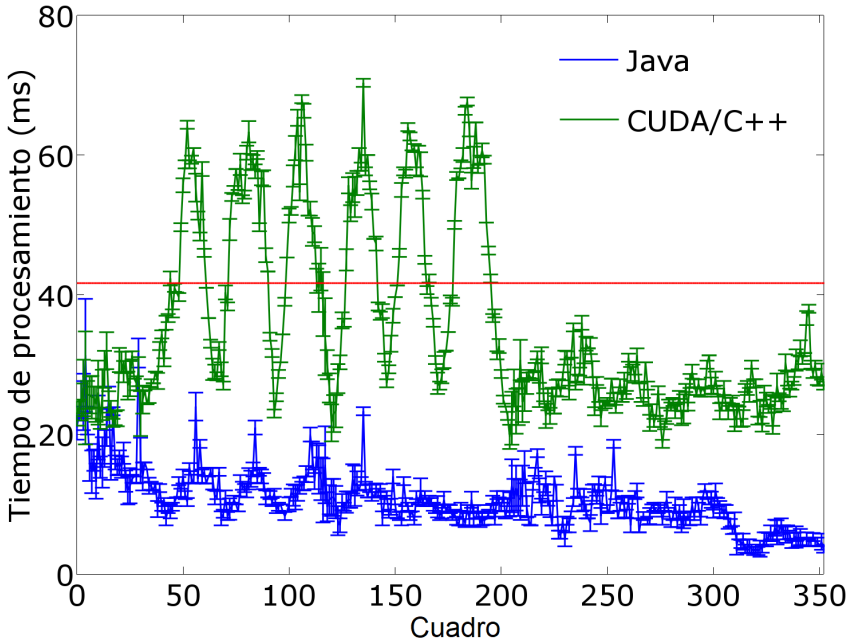


Figura 9: Tiempo de procesamiento para todos los objetos en un cuadro del segundo caso de prueba.

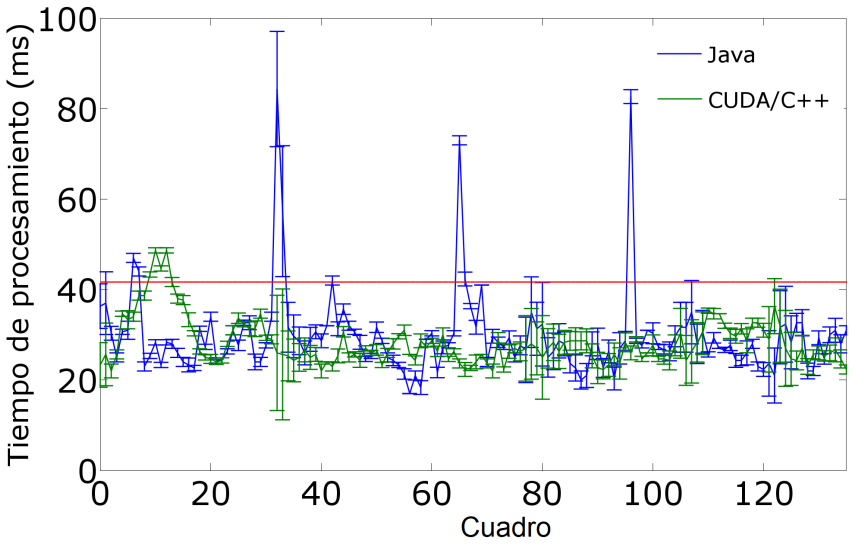


Figura 10: Tiempo de procesamiento para todos los objetos en un cuadro del tercer caso de prueba.

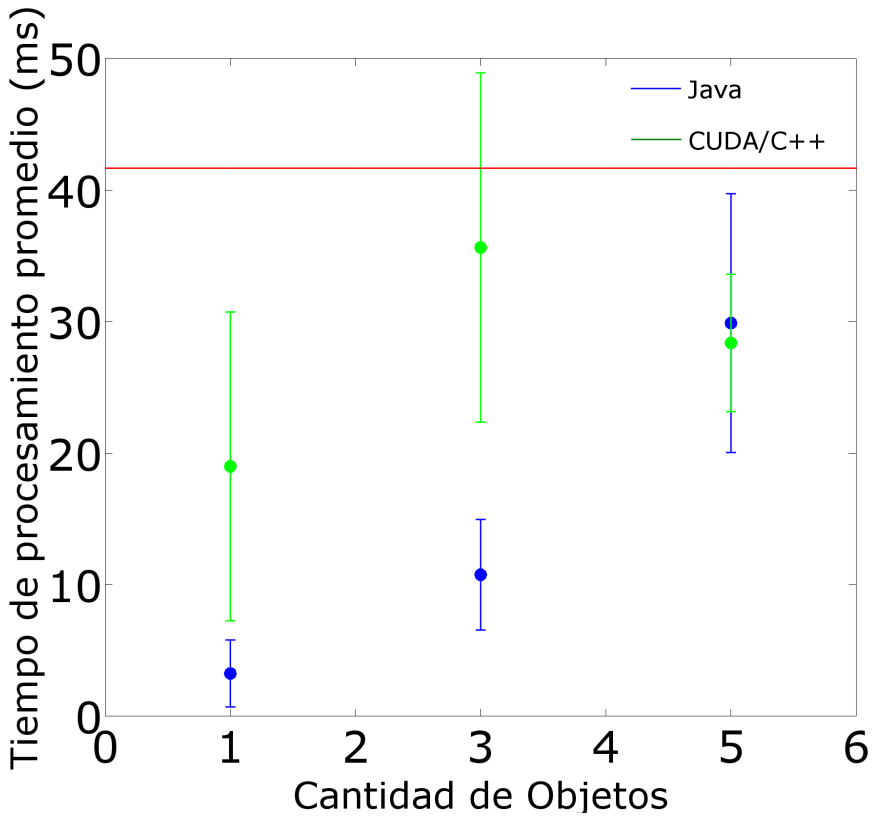


Figura 11: Tiempo de procesamiento promedio por cuadro para los tres casos.

6.1.1. Análisis de los resultados

Algo que puede generar confusión en la figura 11 es que para el tercer caso el tiempo promedio de procesamiento es menor que para el segundo. Como se mencionó anteriormente, debido a las limitaciones del *hardware* con el que se dispuso para las pruebas, los

vídeos son diferentes en cuanto a la brusquedad de sus movimientos. Esto se realizó de esta forma para evitar la pérdida de objetos rastreados por cualquiera de las implementaciones, manteniéndose válida la comparación entre las ejecuciones de un mismo número de objetos rastreados.

Como se observa en la figura 11, para los primeros dos casos la implementación en **Java 1.8** tiene un menor tiempo promedio de procesamiento por cuadro que la implementación en **CUDA/C++**, y su desvío estándar también es menor, por lo que se puede decir que es más estable.

Si bien se esperaba que la implementación en **CUDA/C++** fuese superior en rendimiento a la implementación en Java, ambas lograron un nivel de procesamiento adecuado con respecto al umbral de tiempo real.

En cuanto a lo esperado por la implementación en **CUDA/C++**, su inferior rendimiento en los primeros casos puede atribuirse a una serie de factores elaborados al discutir de las ventajas y desventajas de las unidades de procesamiento gráfico. Por un lado, debe notarse que la velocidad de reloj en los núcleos de las CPU es notablemente mayor a su contraparte en GPU. Al realizar un seguimiento de un reducido número de objetos, se puede asumir que los recursos de la CPU son suficientes para realizar el procesamiento en simultáneo. Al crecer el número de objetos, la GPU escala de mejor manera, aprovechándose de esta forma el mayor número de núcleos. Como referencia, la placa de vídeo utilizada posee 640 núcleos de procesamiento, mientras que placas más modernas, como la **NVIDIA GeForce RTX 2080 Ti**, tienen 4352 núcleos, alrededor de 7 veces más.

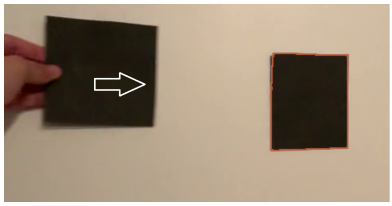
Finalmente, es posible que las primitivas de sincronización utilizadas para asegurar consistencia en los datos hayan repercutido negativamente en el rendimiento, junto con los pasajes de memoria de CPU a GPU involucrados en el algoritmo.

Si bien todas las razones mencionadas anteriormente pueden haber repercutido en el rendimiento obtenido, esto parece percibirse únicamente al rastrearse un número reducido de objetos (primeros dos casos). En el tercer caso, la implementación en **CUDA/C++** tiene tiende a exhibir un mejor rendimiento que la implementación en **Java 1.8** y su desvío estándar también es menor. Luego, podría asumirse que, al menos en el *hardware* utilizado, la implementación en **CUDA/C++** muestra una tendencia sublineal contra la lineal que exhibiría la implementación en CPU. Esto puede atribuirse a la mejor escalabilidad de la implementación en GPU, dado que aunque la velocidad de reloj sea menor en la GPU, el tener cientos de núcleos operando al mismo tiempo resulta más relevante que dicha diferencia de velocidad, o que el tiempo de procesamiento usado en pasajes de memoria y primitivas de sincronización.

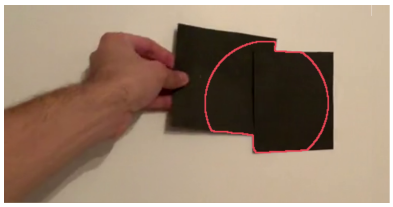
6.2. Desempeño de la limitación del radio

Al igual que la prueba anterior, para realizar esta comparación se propuso medir el tiempo de procesamiento por cuadro en ambos casos, y contrastarlos en un mismo gráfico superponiéndole la función constante que representa el umbral de tiempo real.

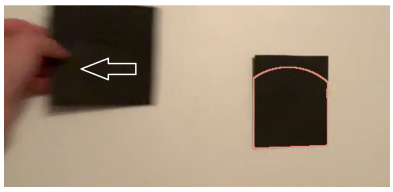
Las figuras 12 y 13 muestran las dos secuencias observadas en los casos planteados.



Acercamiento

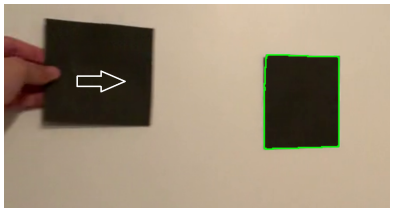


Solapamiento

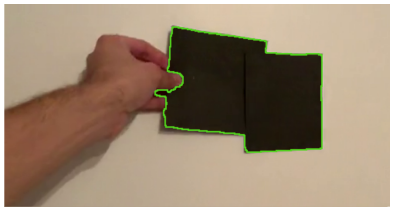


Alejamiento

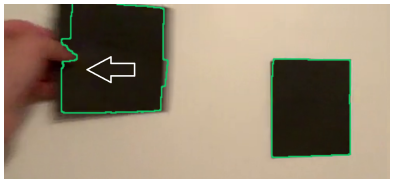
Figura 12: Secuencia observada con la funcionalidad para limitar el radio del centroide encendida



Acercamiento



Solapamiento



Alejamiento

Figura 13: Secuencia observada con la funcionalidad para limitar el radio del centroide apagada

Se realizaron 3 ejecuciones para cada caso, por lo que el gráfico muestra el tiempo de procesamiento promedio por cuadro con sus respectivas barras de error. En la figura 14 se exhiben los resultados del experimento.

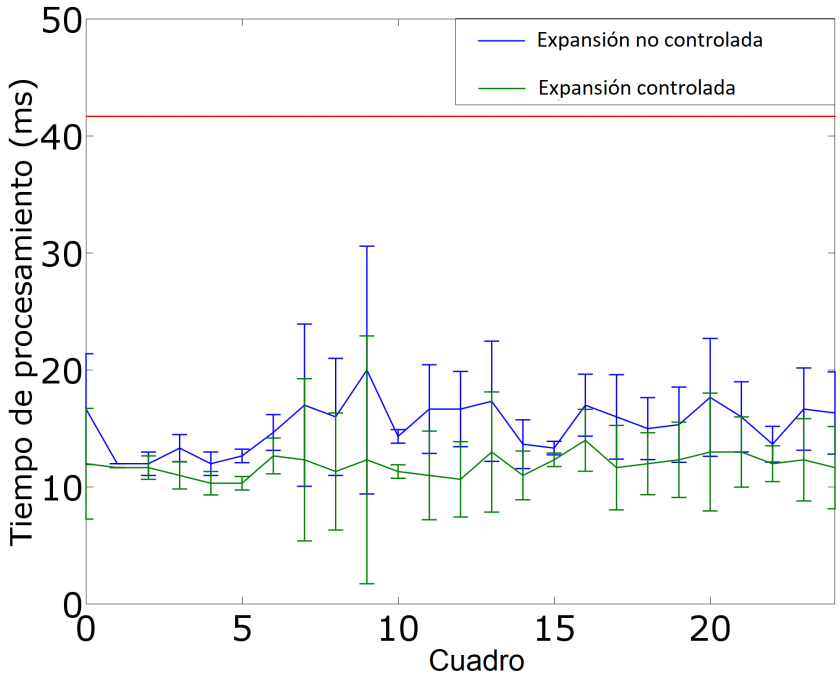


Figura 14: Tiempo de procesamiento para todos los objetos en un cuadro.

6.2.1. Análisis de los resultados

A diferencia de la comparación entre la implementación **CUDA/C++** contra **Java 1.8**, en esta prueba no se notaron discrepancias respecto de los resultados esperados. Al controlar la expansión, el sistema ignora exitosamente al objeto que se superpone al que está siendo seguido, mientras que en el otro caso lo comienza a rastrear.

También se destaca la mayor eficiencia y correctitud del caso de expansión controlada contra el de expansión no controlada, que puede observarse en la figura 14, dado que en el primer caso el tiempo de ejecución promedio por cuadro es menor que en el segundo caso.

7. Conclusiones

La detección de bordes en imágenes o en tramas de vídeo es de sumo interés en el campo de la visión artificial, destacándose su multiplicidad de aplicaciones en los campos de seguridad y robótica. Existen numerosas técnicas clásicas basadas en estimaciones del gradiente del color, y otras más recientes basadas en el concepto de contornos activos, tal como Shi *et al.* propusieron en 2005[1]. Sin embargo, este último trabajo no presenta un mecanismo automático para la detección de regiones iniciales. Asimismo, no toma en consideración el caso en que un objeto es rastreado en la vecindad de objetos de idéntico color, lo que induce una pérdida en el rendimiento a partir de una expansión incontrolada de los contornos del mismo. Para suplir estas falencias, en este trabajo se propuso una serie de adiciones al algoritmo presentado, las cuales han demostrado cumplir con los objetivos propuestos.

Por un lado, se ha propuesto una técnica para la detección automática de objetos en tramas de vídeo. Dicha técnica, basada en el algoritmo MOG2[5][6], detecta píxeles en donde la imagen cambió de manera significativa durante un lapso lo suficientemente prolongado de tiempo, de acuerdo a un parámetro de sensibilidad.

Por otro lado, se ha logrado añadir una restricción al tamaño máximo alcanzado por los objetos rastreados mediante un análisis del centroide y radio de los mismos. De esta forma, se previenen situaciones en las cuales la dimensión de los contornos se incremente de manera incontrolada en un lapso infinitesimal de tiempo, así evitando repercusiones en el rendimiento del algoritmo.

Por último, se ha determinado que el algoritmo presentado en [1] es capaz de rastrear múltiples objetos en tiempo real. A tal propósito, se han realizado implementaciones tanto en CPU como en GPU de dicho algoritmo, notando que en ambas arquitecturas se logra el acometido. Para valores pequeños del número de objetos rastreado, se ha logrado mejores tiempos de ejecución en CPU, observándose una tendencia opuesta en favor a la GPU para $N \geq 5$. Se atribuyó esto a un comportamiento sublineal de la última implementación.

Al discutir acerca de las ventajas y falencias de las unidades de procesamiento gráfico, se ha hecho especial énfasis en la dificultad de implementar métodos de exclusión mutua en entornos masivamente paralelos *sin* repercutir en el rendimiento del algoritmo. Adicionalmente, algunas de las características comunes a las CPU como los predictores de saltos no son habitualmente encontradas en las GPU. Estos elementos hacen que haya algoritmos más y menos aptos para implementarse este tipo de *hardware*. En el caso en particular del algoritmo expuesto en este trabajo, si bien pudo ser implementado en GPU con resultados positivos, es posible que las primitivas de sincronización utilizadas para asegurar consistencia en los datos repercutan negativamente en su rendimiento. Por ende, se propone explorar la posibilidad de realizar una implementación que combine las ventajas de ambas plataformas, a los efectos de lograr una implementación en tiempo real que minimice los recursos necesarios.

Referencias

[1] Y. Shi and W.C. Karl "Real-time tracking using level sets", IEEE Conf. on Computer Vision and Pattern Recognition, pp. 34-41, 2005.

[2] I. Sobel and G. Feldman "Presentation at Stanford A.I. Project", 1968

[3] Canny, J. "A Computational Approach To Edge Detection", IEEE Transactions on Pattern Analysis and Machine Intelligence, 8(6):679–698, 1986

[4] Hieu Tat Nguyen *et al.* "Tracking Nonparameterized Object Contours in Video", IEEE Transactions On Image Processing, Vol. 11, No. 9, 2002

[5] Zoran Zivkovic "Improved Adaptive Gaussian Mixture Model for BackgroundSubtraction", 2004.

[6] Zoran Zivkovic, Ferdinand van der Heijden "Efficient adaptive density estimation per image pixelfor the task of background subtraction", 2006.

[7] Suzuki, S. and Abe, K. "Topological Structural Analysis of Digitized Binary Images by Border Following", CVGIP 30 1, pp 32-46 (1985)