Captura de movimiento basada en Mean Shift

Organización del Computador II
Trabajo Final

Shai Bianchi

Marzo 2017

1. Introducción

Mean Shift es un método estadístico iterativo cuyo propósito es la ubicación de máximos de una función de densidad. En el contexto de la *visión artificial*, puede utilizarse para construir un algoritmo de captura de movimiento.

En este trabajo se presenta una implementación *SIMD* de un algoritmo tal escrita en *x86 Assembly* con instrucciones *SSE*, y se comparan los tiempos de ejecución resultantes con los de dos implementaciones secuenciales distintas escritas en `C++`.

2. Problema

En general, un algoritmo de captura de movimiento es diseñado para obtener algún tipo de información acerca del movimiento de un objeto en un video, tanto provisto en tiempo real como pregrabado. En el contexto de este trabajo, se busca seguir el movimiento de un objeto arbitrario en una captura a color dada en tiempo real: dada una captura de video, representada como una secuencia de *frames* de la forma $[f_0, f_1, ...]$, y una ventana rectangular V que define en uno de los frames f_m una sección que marca la ubicación en ese instante del objeto a seguir, se desea estimar en cada uno de los frames siguientes f_i con $i > m$ una ubicación para V que corresponda en f_i al nuevo posicionamiento del objeto.

3. Algoritmo

El algoritmo se divide en dos etapas: primero la de *muestreo* y luego la de *seguimiento*.

3.1. Muestreo

Esta primera etapa se ejecuta una sola vez, previo al seguimiento en si. En ella se toma la información de los colores que se encuentran en la región de f_m indicada por V, notada $f_m[V]$. Suponiendo que la ventana contiene de forma "ajustada" a la imagen del objeto, esto es, encerrando la mayor superficie posible del objeo sin incluir píxeles que no le corresponden, estos colores representan al objeto a seguir.

La información de colores extraída se procesa para ser utilizada en la etapa de seguimiento. Dicho procesamiento consiste en construir un histograma de colores H de los píxeles que se encuentran en

$f_m[V]$, produciendo una estructura de datos que, por cada posible valor en el espacio de colores en el que se representan los frames, guarda la cantidad de apariciones de ese color en el objeto a seguir.

3.2. Seguimiento

En la segunda etapa se trabaja frame a frame. En la iteración $i+1$, dada la estimación de V obtenida en el frame inmediatamente anterior f_i dada como las coordenadas del centro geométrico de la ventana, se quiere estimar la posición del centro de V para el nuevo frame f_{i+1}. Para esto, se siguen dos pasos: la construcción a partir de f_{i+1} de un *mapa de confianza* para la nueva ubicación del objeto en ese frame, y la *ubicación* de su posición estimándole aplicando *Mean Shift* al mapa obtenido.

3.2.1. Mapa de confianza

El mapa de confianza M_{i+1} para f_{i+1} dado H es una matriz del mismo tamaño de f_{i+1} que resulta de aplicar un *back projection* de H sobre f_{i+1}. Dicho proceso asigna a la celda $M_{i+1}(x, y)$ del mapa el valor que guarda el histograma para el color del pixel $f_{i+1}(x, y)$, el de la posición x, y de f_{i+1}. Es decir,

$$M_{i+1}(x, y) \leftarrow H(f_{i+1}(x, y))$$

Como los píxeles de los colores del objeto tienen mayores valores, se interpreta al mapa de confianza como una función de densidad de dos parámetros, conteniendo en la posición x, y un valor proporcional a la probabilidad de que el pixel $f_{i+1}(x, y)$ del frame pertenezca al objeto que se quiere ubicar.

3.2.2. Ubicación

En este paso se produce la estimación propiamente dicha de la posición del objeto en f_{i+1}. Dicha estimación, que será el nuevo centro geométrico de V, se calcula como el máximo de densidad en M_{i+1}, local a la posición anterior de V: si el objeto estuvo en una determinada posición en el frame f_i, se estima que en el frame f_{i+1} se encontrará en el máximo de densidad del mapa de confianza cercano a su posición anterior. Para lograr eso, siendo la ubicación de máximos de densidad el propósito de *Mean Shift*, se aplica dicho método con entrada M_{i+1} y las coordenadas del centro geométrico de V actualizado hasta f_i. Tratándose de un algoritmo iterativo, se le pasa también una cantidad de iteraciones a realizar como criterio de terminación.

En cada iteración, *Mean Shift* calcula el *centro de masa* c^{i+1} de la sección de $M_{i+1}[V]$ y pasa a ubicar V de modo que su nuevo centro geométrico sea ese.

El centro de masa c^{i+1} de coordenadas c_x^{i+1} y c_y^{i+1} se calcula como

$$c_x^{i+1} = m_{10}/m_{00}$$
$$c_y^{i+1} = m_{01}/m_{00}$$

donde

$$m_{00} = \sum_{(x,y) \in V} M_{i+1}(x, y)$$

es el momento de orden 0 de $M_{i+1}[V]$, y

$$m_{10} = \sum_{(x,y) \in V} x\, M_{i+1}(x,y)$$
$$m_{01} = \sum_{(x,y) \in V} y\, M_{i+1}(x,y)$$

son los momentos de segundo orden de $M_{i+1}[V]$.

El método eventualmente converge a una determinada posición; en este trabajo se interrumpe luego de la cantidad de iteraciones que se le indica, determinada empíricamente en el contexto del trabajo como suficiente para la convergencia.

4. Implementación

La implementación del algoritmo de captura de movimiento se divide en tres rutinas: una para el muestreo, otra para la generación del mapa de confianza y una tercera que implementa *Mean Shift* para la ubicación del objeto en su nueva posición. Todas las implementaciones utilizan recursos de la librería de visión artificial *OpenCV*[1] para el manejo de entrada y salida de frames y de estructuras de datos para el almacenamiento de las matrices que representan frames, mapas de confianza e histogramas.

Para las tres rutinas se implementan tres versiones distintas. Una es la vectorial que se basa en instrucciones *SSE* en *x86 Assembly*, y las otras dos son versiones secuenciales hechas en `C++`: una escrita de manera idiomática utilizando facilidades de `C++` y de *OpenCV*, como por ejemplo construcciones de la STL o iteradores a las imágenes, con la intención de contar con una versión como la que probablemente se obtendría programando en un contexto real en el que se prioriza la mantenibilidad y legibilidad del código sobre una optimización especialmente marcada de sus tiempos de ejecución; la segunda escrita *à la C*, utilizando aritmética de punteros y sin valerse de ningún recurso de *OpenCV*. La versión vectorial, al igual que la anterior, no utiliza ningún servicio de *OpenCV* o la STL de `C++`.

Se plantea como hipótesis que, entre las versiones secuenciales, la escrita idiomáticamente será la de tiempos de ejecución menos favorables, mientras que la versión vectorial será la mejor de las tres.

4.1. Detalles generales

Los frames se representan en memoria como matrices de píxeles, almacenados como una secuencia contigua de filas siendo cada fila una secuencia contigua de píxeles, con posible padding entre filas. El espacio de colores utilizado es RGB, con 8 bits de profundidad. En el muestro y la estimación posterior de la ubicación del objeto se utilizan recortes de los frames correspondientes a la ventana de seguimiento, y estos recortes se representan como imágenes al igual que los frames, con alta probabilidad de padding entre filas.

Por su parte, el histograma se representa con una matriz tridimensional, ya que que el espacio de colores RGB define a cada color con tres números y la matriz debe guardar un escalar por cada color posible. De esta forma, cada valor RGB se interpreta como una terna de coordenadas a la

[1] http://opencv.org/

matriz tridimensional que es el histograma. Se almacena en memoria como una secuencia contigua de planos, donde cada plano es almacenado fila por fila al igual que los frames, en este caso sin padding.

Por último, la estructura usada para el mapa de confianza es una matriz bidimensional de escalares de iguales dimensiones que los frames, y se almacena de la misma manera, sin padding.

El tipo de datos numérico del histograma se decide buscando un balance entre el tamaño que ocuparían las estructuras y las limitaciones que impondrían las distintas posibilidades: un tipo de mayor rango de representación ocuparía más espacio, y uno de menor rango podría imponer una cuota superior demasiado chica al tamaño posible de la ventana de seguimiento. Al mismo tiempo, el de menor tamaño puede permitir sacarle mayor ventaja a la vectorización, pudiendo posiblemente procesar más elementos por iteración. En el peor caso teórico en el que todos los píxeles delimitados por la ventana V en el frame de muestreo f_m son del mismo color, se tendría en el histograma un solo valor no-nulo igual al área de V. Entonces, el máximo valor representable en los escalares el histograma impone un límite al tamaño de la ventana. Con esto en mente, entre los tipos básicos de datos enteros de `C`, se decide utilizar `unsigned short` para el histograma, lo cual permitiría un área máximo para la ventana de $2^16 - 1 = 65535 \simeq 255^2$. Si se quiere usar una ventana cuadrada, esta podría tener como máximo un tamaño de 255×256 que se considera más que razonable para el uso esperado del algoritmo en este trabajo.

Como el mapa de confianza almacena valores leídos del histograma, el tipo de datos aritmético usado para esa estructura es `unsigned short` también.

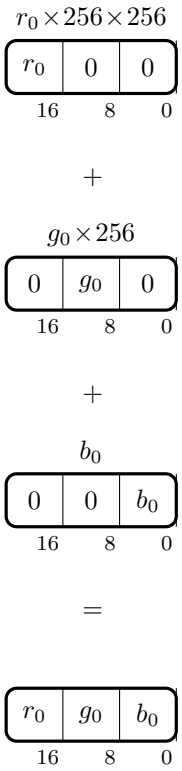
En todas las implementaciones vectoriales se trabaja de a 4 o 5 elementos por iteración, recorriendo los elementos en el orden que se encuentran en memoria. Para soportar matrices cuya cantidad de filas no es múltiplo del número de elementos que se procesa por iteración, lo que sobra en cada fila, que serán a lo sumo 3 o 4 elementos según corresponda, se itera secuencialmente.

4.2. Muestreo

En este paso se quieren leer píxeles de $f_m[V]$, y por cada uno incrementar el contador inicializado en 0 en la posición correspondiente de H.

En una matriz bidimensional, la posición i, j suele interpretarse como la del j-ésimo elemento de la i-ésima fila. De forma similar, en una matriz tridimensional, entendemos la posición de coordenadas i, j, k como la del k-ésimo elemento de la j-ésima fila del i-ésimo plano. Ahora, observando que el histograma tiene 256 planos de 256×256 elementos cada uno, podemos concluir que el offset en memoria del elemento i, j, k es (suponiendo que el tamaño de cada uno es 1) $i \times 256 \times 256 + j \times 256 + k$, donde el primer sumando direcciona el plano en el que se encuentra el elemento, el segundo direcciona la fila en ese plano y el tercero al elemento en si.

Ahora, consideremos el píxel $[r_0, g_0, b_0]$, que en memoria es una secuencia de 24 bits almacenados de modo que b_0 es el byte bajo, e interpretemos sus valores de cada canal RGB como coordenadas al histograma. Procediendo de manera análoga a lo explicado arriba, la posición del píxel en H podría ser $r_0 \times 256 \times 256 + g_0 \times 256 + b_0$, que es exactamente el número resultante de interpretar $[r_0, g_0, b_0]$ como un entero de 24 bits:



Entonces, el offset en memoria de esa posición es el resultado de multiplicar ese número por `sizeof(short)`. Aprovechando esta observación, el único procesamiento que debe hacerse a los píxeles para obtener el offset en H de cada uno es multiplicarlos el número que representan como entero de 24 bits por 2.

En la implementación vectorial se traen 5 píxeles de f_m en cada iteración, se extrae la terna correspondiente a cada píxel, se multiplica por 2 el valor que representa como entero de 24 bits y se utiliza como offset respecto del puntero a los datos de H para incrementar el valor correspondiente en 1.

4.3. Mapa de confianza

Dado un frame, su mapa de confianza se construye consultando el valor asociado al color del píxel i, j en H y escribiéndolo en la posición i, j del mapa. La forma de calcular el offset del valor de cada píxel en H es idéntica a la del muestreo. En la versión vectorial, se leen 5 píxeles del frame en cuestión por iteración y se escriben en el mapa de confianza los 5 valores leídos de H.

4.4. Ubicación

Conseguido el mapa de confianza M_{i+1} para un frame determinado f_{i+1}, se aplica Mean Shift para actualizar V. Para ello se requiere realizar las cuentas exhibidas en la sección anterior:

$$c_x^{i+1} = m_{10}/m_{00}$$
$$c_y^{i+1} = m_{01}/m_{00}$$

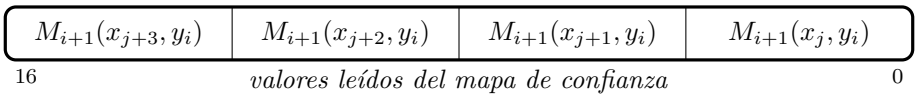
donde

$$m_{00} = \sum_{(x,y) \in V} M_{i+1}(x,y)$$
$$m_{10} = \sum_{(x,y) \in V} x\, M_{i+1}(x,y)$$
$$m_{01} = \sum_{(x,y) \in V} y\, M_{i+1}(x,y)$$

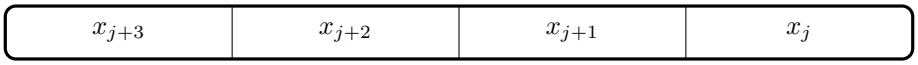
y actualizar la posición de V de modo que su nuevo centro geométrico sea $c^{i+1} = (c_x^{i+1}, c_y^{i+1})$.

Para realizar estos cálculos, se inicializan tres registros `xmm` en 0 que sirven como acumuladores para el cálculo de los momentos de $M_{i+1}[V]$. Luego se recorren los elementos de esa región del mapa de confianza. En cada iteración, como los escalares de M_{i+1} son de tipo `unsigned short`, se desempaquetan de manera de tener 4 enteros sin signo de 32 bits empaquetados en un registro `xmm`, paso necesario para poder realizar las cuentas planteadas que rápidamente se salen del rango representable por enteros de 16 bits. Estos cálculos se realizan con enteros, aprovechando que los sumandos de las sumatorias exhibidas multiplican índices enteros por valores leídos de los acumuladores de H que son enteros también, evitando así operaciones innecesarias de punto flotante.

Suponiendo que estamos en la iteración que procesa los escalares de M_{i+1} de posiciones de coordenada $y = y_i$ y coordenadas $x = x_j, x_{j+1}, x_{j+2}, x_{j+3}$, tenemos los siguientes enteros empaquetados de 32 bits:



Para poder realizar las multiplicaciones por los índices de fila y columna de las fórmulas de los momentos de segundo orden m_{10} y m_{01}, se mantienen registros `xmm` con esos índices enteros de x e y que se van actualizando iteración a iteración. En la iteración actual, tienen los índices:



Las multiplicaciones se realizan de manera empaquetada, logrando en tres registros `xmm` distintos los resultados:

$M_{i+1}(x_{j+3}, y_i)$	$M_{i+1}(x_{j+2}, y_i)$	$M_{i+1}(x_{j+1}, y_i)$	$M_{i+1}(x_j, y_i)$

16 0

$x_{j+3} M_{i+1}(x_{j+3}, y_i)$	$x_{j+2} M_{i+1}(x_{j+2}, y_i)$	$x_{j+1} M_{i+1}(x_{j+1}, y_i)$	$x_j\, M_{i+1}(x_j, y_i)$

16 0

$y_i\, M_{i+1}(x_{j+3}, y_i)$	$y_i\, M_{i+1}(x_{j+2}, y_i)$	$y_i\, M_{i+1}(x_{j+1}, y_i)$	$y_i\, M_{i+1}(x_j, y_i)$

16 0

que corresponden a los momentos m_{00}, m_{10} y m_{01} respectivamente. Previo a sumarlos a los acumuladores de momentos, se convierten a `float`. Este paso se realiza en cada iteración ya que los valores resultantes de estas sumatorias exceden el rango representable por enteros de 32 bits en el peor caso.

Luego de recorrer $M_{i+1}[V]$ y acumular los resultados en los registros `xmm` de acumulación mencionados, tenemos en cada uno de ellos 4 enteros cuya suma da m_{00}, m_{10} y m_{01}:

m_{00}^3	m_{00}^2	m_{00}^1	m_{00}^0

16 $m_{00} = \sum_{i=0}^{3} m_{00}^i$ 0

m_{10}^3	m_{10}^2	m_{10}^1	m_{10}^0

16 $m_{10} = \sum_{i=0}^{3} m_{10}^i$ 0

m_{01}^3	m_{01}^2	m_{01}^1	m_{01}^0

16 $m_{01} = \sum_{i=0}^{3} m_{01}^i$ 0

Con el objetivo de realizar las operaciones de división necesarias para calcular el centro de masa c^{i+1}, se calcula el momento de primer orden realizando sumas horizontales en el registro correspondiente, obteniendo:

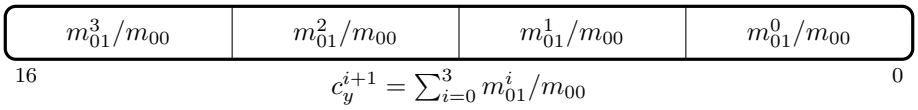
m_{00}	m_{00}	m_{00}	m_{00}

16 0

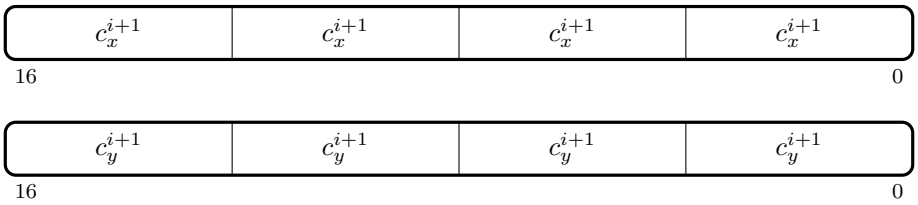
Seguido de eso, aprovechando la distributividad de la división, se dividen los acumuladores de los momentos de segundo orden previo a sumar sus componentes con el fin de evitar un mayor potencial de pérdida de precisión debida a la magnitud del resultado de la suma:

m_{10}^3/m_{00}	m_{10}^2/m_{00}	m_{10}^1/m_{00}	m_{10}^0/m_{00}

16 $c_x^{i+1} = \sum_{i=0}^{3} m_{10}^i/m_{00}$ 0



y luego se realizan sumas horizontales para obtener finalmente

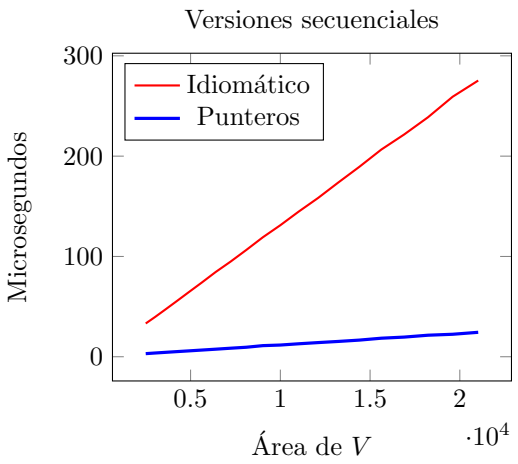


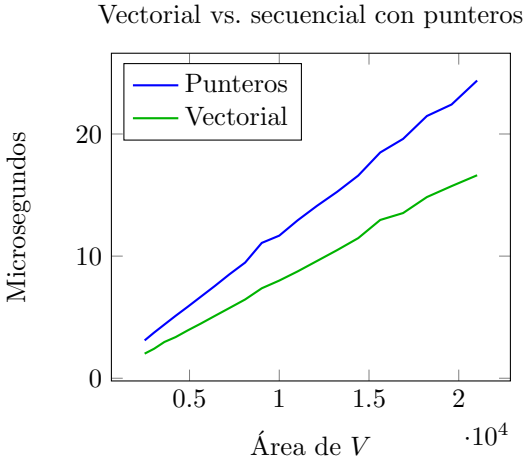
Notar que de esta forma en ningún momento se calcula el valor de los momentos de segundo orden, sino que se logra pasar de los acumuladores directamente a las coordenadas que se busca calcular de c^{i+1}. Estos valores se convierten a enteros y se utilizan para actualizar V. Luego se procede a la iteración siguiente, tantas veces como se indique como criterio de terminación de Mean Shift.

5. Tiempos de ejecución y análisis

Se generan casos de prueba para medir los tiempos de ejecución de cada etapa del algoritmo, aumentando las dimensiones de los parámetros que correspondan para cada una: en la de muestreo se va aumentando el tamaño de V, en la del mapa de confianza se aumenta el tamaño del frame y en la de la ubicación se aumenta el tamaño de V dejando fijo el del mapa de confianza. Los tiempos exhibidos son el resultado de un promedio de 1000 ejecuciones. Las secuenciales, escritas en `C++`, se compilan siempre con el flag de mayor optimización `-O3` del compilador `g++`.

5.1. Etapa de muestreo

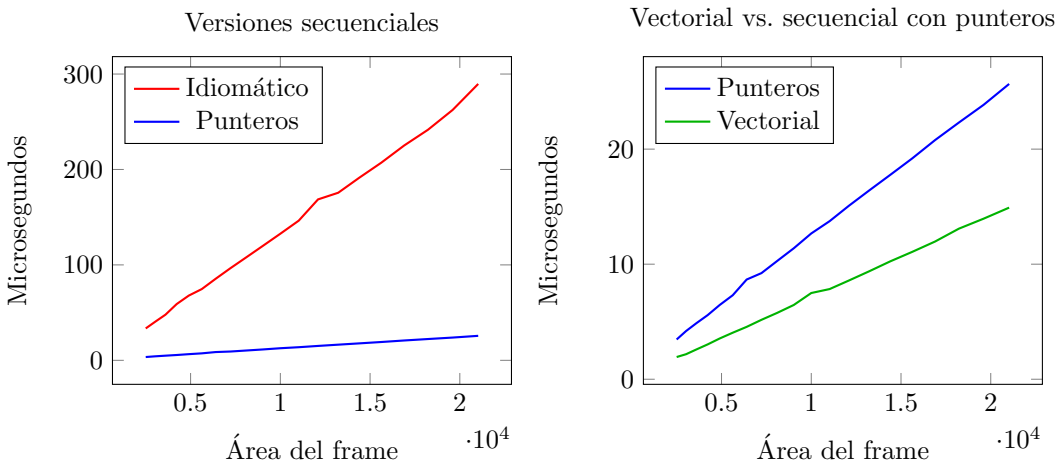




La implementación con punteros corre en tiempos de entre 8.6 % y 9.4 % de los de la versión idiomática. Esta marcada diferencia se puede asociar al uso de iteradores en la versión idiomática, que facilita y simplifica mucho el código, pero evidentemente agrega un costo de computación considerable. Por su parte, la vectorial corre tiempos de entre el 64.6 % y el 70.2 % de la versión de punteros. Comparando la vectorial con la versión secuencial idiomática, la primera corre en tiempo de alrededor del 6 % de los tiempos de la segunda.

Las tres versiones realizan operaciones aritméticas sencillas de sumas y multiplicaciones por 2 para calcular direcciones de celdas del histograma. Por ello y analizando el desensamblado del ejecutable generado por `g++ -O3`, tiene sentido asociar la mejora notable de la versión vectorial sobre las secuenciales a la reducción significativa de la cantidad de accesos a memoria obtenida al procesar 5 píxeles de f_m en cada iteración mientras que el código de máquina de `g++ -O3` recorre el frame en cuestión píxel por píxel.

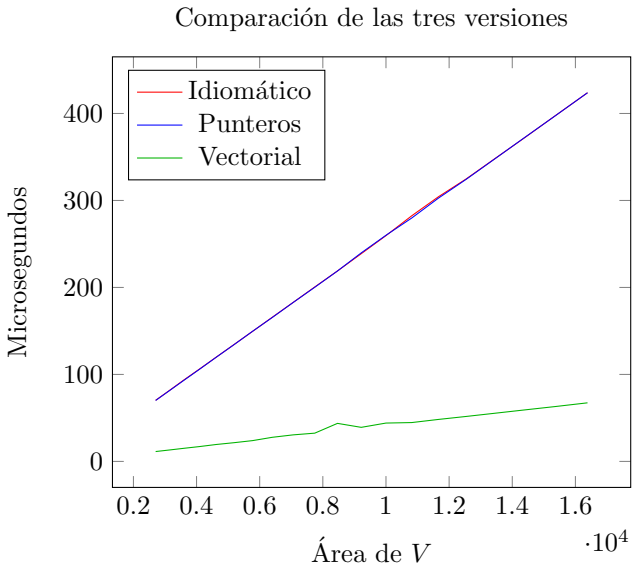
5.2. Generación del mapa de confianza



Comparando las implementaciones secuenciales, la de punteros tiene tiempos entre el 8.9 % y el 10.4 % de la idiomática. La versión vectorial a su vez muestra tiempo de entre el 51.9 % y el 59.1 % de la que utiliza aritmética de punteros. Comparando la vectorial con la secuencial más lenta, la primera corre en tiempos de entre el 5 % y el 5.8 % de la segunda.

El comportamiento de esta rutina en cuanto a tiempos de ejecución es parecido al de la rutina anterior, hecho reflejado en la descripción de los algoritmos que ambas implementan. El patrón de acceso a memoria en ambas rutinas es similar, y en ambas el procesamiento de los datos en si es relativamente sencillo, de modo que la ventaja conseguida por la versión vectorial sobre las secuenciales se asocia principalmente al hecho de reducir la cantidad de accesos a memoria: la vectorial lee de a 5 píxeles por iteración, mientras que el código generado por `g++ -O3` para ambas versiones secuenciales recorre las matrices de a un elemento. Pero además, a diferencia de la rutina anterior, la de esta etapa tiene la tarea de escribir datos en el mapa de confianza, paso que la implementación vectorial de esta rutina hace de a 5 elementos y que el código de `g++ -O3` también hace de a uno. Lo último explica por qué la mejora respecto de las versiones secuenciales es mayor aún en esta etapa que en la de muestreo.

5.3. Ubicación



En este caso, las dos versiones secuenciales presentan tiempos prácticamente iguales. Este hecho se explica observando que, por la naturaleza del algoritmo y a diferencia de los de las dos rutinas anteriores, no resulta natural implementarlo utilizando iteradores. Por lo tanto, si bien la versión idiomática utiliza facilidades de *OpenCV* para acceder a las matrices en vez de realizar aritmética de punteros, la forma de recorrer las matrices en memoria es muy parecida. Junto al hecho de computar las fórmulas de momentos de manera idéntica, los tiempos dan muy parecido.

Sin embargo, la versión SIMD presenta una mejora muy marcada respecto de las versiones secuenciales, mucho más que en las rutinas anteriores: comparando la vectorial contra la secuencial con punteros, las mejoras son de entre el 80 % y el 84.1 %, mientras que las mejoras en las rutinas anteriores están en el rango de 29.8 %-48.1 %. Analizando el desensamblado del ejecutable generado para las versiones secuenciales de `g++ -O3`, se aprecia que el compilador, al igual que con las rutinas anteriores, realiza el recorrido de los elementos en memoria de a uno mientras que la implementación vectorial lo hace de a 4. Pero más aún, a diferencia de los algoritmos de las etapas anteriores, en este se realizan varios cómputos y conversiones de punto flotante en cada iteración; al ser estas operaciones relativamente costosas, el hecho de poder hacerlas de manera empaquedata en la versión escrita con instrucciones SSE marca una gran diferencia de performance con un código que opera únicamente de manera escalar. Además, vale notar que el código generado por `g++ -O3` realiza accesos a la pila en sus iteraciones, mientras que en la versión vectorial escrita para este trabajo se logran evitar por completo en los ciclos los accesos a memoria adicionales al necesario para leer los datos requeridos para los cálculos de momentos.

5.4. Conclusión

En conclusión, de manera compatible con la hipótesis planteada, se nota una clara ventaja para la versión vectorial, especialmente cuando se la compara con una implementación secuencial idiomática.

Este hecho es útil a la hora de implementar una aplicación de esta naturaleza en la que poder procesar rápidamente es vital para su funcionamiento en tiempo real. A medida que se aumente el *FPS* del feed de video será de mayor importancia la performance de las rutinas, y más aún si se agregan cálculos adicionales al algoritmo como el filtrado de ruido con una convolución gaussiana sobre los mapas de confianza para mejorar la precisión de la etapa de ubicación. Tal mejora al algoritmo puede ser una mayor motivación aún para optar por una implementación vectorial como la utilizada para estos experimentos, ya que demostró una clara ventaja ante la tarea de llevar a cabo un procesamiento matricial intensivo en cálculos de punto flotante.