



## DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

## SIMD

Organización del Computador II  
Primer Cuatrimestre de 2021

Integrante	LU	Correo electrónico
Federico Ivan Barrena Guzman	236/18	fedebarrena@gmail.com
Camila Rosario Camaño	310/17	camicam26@gmail.com
Tomás Ossa	752/19	ossatomas2@gmail.com



### Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2160 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (54 11) 4576-3359

<http://www.fcen.uba.ar>

## Resumen

En el presente trabajo se hace un análisis de rendimiento entre distintas implementaciones de filtros para imágenes. Uno de los ejes principales es el de analizar la diferencia en rendimiento (tiempo, ciclos de clock) entre implementaciones del mismo filtro con instrucciones de tipo SIMD y su equivalente en C.

# Índice

<b>1. Introducción</b>	<b>4</b>
<b>2. Desarrollo</b>	<b>5</b>
2.1. Filtro Max . . . . .	5
2.1.1. Algoritmo . . . . .	5
2.1.2. maximoSubmatriz . . . . .	5
2.1.3. maximoFilas . . . . .	6
2.1.4. bordeBlanco . . . . .	7
2.2. Filtro Gamma . . . . .	7
2.2.1. Algoritmo . . . . .	8
2.3. Filtro Broken . . . . .	8
2.3.1. Consideraciones previas . . . . .	9
2.3.2. Algoritmo . . . . .	9
<b>3. Comparación de rendimiento</b>	<b>10</b>
3.1. Comparación Max . . . . .	10
3.1.1. Comparación del Código . . . . .	10
3.1.2. Comparación de rendimiento . . . . .	11
3.2. Comparación Gamma . . . . .	13
3.2.1. Comparación del Código . . . . .	13
3.2.2. Comparación de rendimiento . . . . .	13
3.3. Comparación Broken . . . . .	14
3.3.1. Comparación del Código . . . . .	14
3.3.2. Comparación de rendimiento . . . . .	14
<b>4. Experimentos y resultados</b>	<b>15</b>
4.1. Experimento 1: Filtro Gamma con vpgatherdd. . . . .	15
4.1.1. Motivaciones . . . . .	15
4.1.2. Algoritmo Gama_lookup_table . . . . .	15
4.1.3. Comparación Gamma_lookup_table con su versión en C y Gamma original . . . . .	16
4.1.4. Análisis de resultados . . . . .	17
4.1.5. Conclusiones . . . . .	18
4.2. Experimento 2: Alteraciones del Filtro Broken . . . . .	19
4.2.1. Motivaciones . . . . .	19
4.2.2. Algoritmo Broken_alternativo . . . . .	19
4.2.3. Comparaciones Broken_Alternativo ASM vs C . . . . .	19
4.2.4. Algoritmo Broken_real_offset . . . . .	20
4.2.5. Comparaciones Broken_real_offset ASM vs C . . . . .	21
4.2.6. Comparaciones entre los tres filtros para Broken . . . . .	21

## Sección ÍNDICE

---

4.2.7. Conclusiones . . . . .	22
<b>5. Conclusión</b>	<b>22</b>
<b>Appendices</b>	<b>24</b>
<b>A. Imágenes adicionales de las comparaciones de rendimiento entre C y ASM</b>	<b>24</b>
A.1. Filtro Max . . . . .	24
A.2. Filtro Gamma . . . . .	27
A.3. Filtro Broken . . . . .	30
<b>B. Imágenes adicionales para el experimento sobre Gamma</b>	<b>33</b>
B.1. Comparaciones Filtro Gamma_lookup_table ASM vs C . . . . .	33
B.2. Comparaciones de filtros Gamma . . . . .	36
<b>C. Imágenes adicionales para el experimento sobre Broken</b>	<b>39</b>
C.1. Comparaciones Broken_alternativo Assembler vs C . . . . .	39
C.2. Comparaciones de Broken_real_offset en Assembler vs C . . . . .	42
C.3. Comparaciones de los tres filtros Broken . . . . .	45

## 1. Introducción

El objetivo de este Trabajo Práctico consiste en implementar filtros para imágenes utilizando el modelo de procesamiento **SIMD**. En este informe realizaremos comparaciones con filtros implementados en C, y luego desarrollaremos y analizaremos experimentos sobre los filtros en SIMD.

Consideramos a una imagen como una matriz de píxeles, en donde cada píxel está determinado por cuatro componentes: azul (B), verde (G), rojo (R) y transparencia (A). Cada una de estas componentes tendrá 8 bits (1 byte) de profundidad, es decir, sus valores están en el rango [0,255]. En todos los filtros, el valor del componente de transparencia será 255.

La imagen de entrada la denominaremos *src*, y *dst* será nuestra imagen destino. Ambas imágenes son archivos en formato **BMP**: cada línea de píxeles se almacena de izquierda a derecha, y los píxeles se almacenan en memoria en el siguiente orden: B,G,R,A.

En primer lugar, implementamos el filtro **Max**, que consiste en recorrer la imagen buscando el píxel que suma el máximo entre sus componentes RGB en matrices de 4x4 píxeles. El píxel encontrado se replicará en la imagen destino en la matriz de 2x2 en la misma posición que la submatriz que resulta de quitarle a la matriz de 4x4 las filas y columnas en los extremos. Los bordes de la imagen se ponen en blanco.

Luego, para obtener el segundo filtro se cambia la intensidad de los píxeles que componen la imagen a transformar. Este filtro se conoce como **Gamma**, y la transformación de los píxeles es el resultado de dividir cada componente de los píxeles por 255, aplicarle la raíz cuadrada, y finalmente multiplicarlos por 255.

Por último, el filtro **Broken** se obtiene a través de una suerte de shift de los componentes de los pixeles según sea indicado por un arreglo de entrada de 40 números. El arreglo solo contiene los offsets -16, -8, -4, 0, 4, 8, 16 y 32, limitando los casos a considerar para su implementación. Tanto el filtro **Gamma** como el **Broken** operan sobre todos los píxeles de la imagen.

A continuación presentamos ejemplos de los filtros utilizando sus implementaciones en C:



(a) Imagen Original



(b) Filtro Max



(c) Filtro Gamma.



(d) Filtro Broken

El orden en el que este trabajo esta detallado es el siguiente:

- **Sección 2:** desarrollo y explicación de los filtros *Max*, *Gamma* y *Broken*.
- **Sección 3:** comparación entre las versiones de **SIMD** y **C** de cada filtro.
- **Sección 4:** dos experimentos, uno sobre el filtro Gamma y el otro sobre el filtro Broken.
- **Sección 5:** conclusiones finales.
- **Apéndice:** gráficos adicionales.

## 2. Desarrollo

### 2.1. Filtro Max

El filtro Max recorre la imagen de a una matriz de 4x4 píxeles, buscando el píxel que sume el máximo entre sus componentes RGB. Dicho píxel se replica en la submatriz de 2x2 píxeles centrada en la matriz que recorre. Los bordes de la imagen se ponen en color blanco.

#### 2.1.1. Algoritmo

A continuación explicaremos detalladamente el procedimiento que sigue nuestra implementación.

1. El cuerpo principal de la función tiene dos ciclos: uno externo que recorre las filas de la imagen y otro interno que recorre las columnas. Antes de iniciar el ciclo principal, en *ecx* y *r9d* guardamos *height*-2 (cantidad de filas - 2) y *width*-2 (cantidad de píxeles en una fila - 2) respectivamente para utilizarlos como contadores. Les restamos 2 ya que no recorreremos los bordes laterales, inferior y superior (se procesan aparte). Además, cargamos las máscaras en los registros *xmm10*, *xmm11*, *xmm12* *yxmm13*. A su vez guardamos en *esi* el *rowSize* (en bytes).
2. En cada iteración del ciclo interno (columnas) se llama a la función **maximoSubmatriz**. Esta función devolverá en *xmm1* los componentes del píxel que sumen el valor máximo dentro de la submatriz de 4x4 correspondiente, repetidos 4 veces en el registro. Una vez obtenido ese píxel, se escribe su valor en la imagen de destino, utilizando la instrucción *pextrq*. Los píxeles que se deben escribir son 4 en total (matriz de 2x2) pero no podemos escribir todos a la vez porque no son contiguos en memoria. Antes de pasar a la siguiente iteración actualizamos los punteros de la imagen destino y fuente en 8: la próxima submatriz que se tiene que procesar es la que empieza dos píxeles más a la derecha de la iteración actual.
3. Al finalizar el ciclo interno y antes de pasar a la siguiente iteración del ciclo externo, se actualizan los punteros a las imágenes, avanzando dos filas: en el ciclo de las columnas se escribió el resultado en la fila siguiente y en la siguiente a la siguiente.
4. Luego del ciclo principal, una vez que escribimos en toda la imagen menos los bordes el valor correspondiente, se llama a la función **bordeBlanco**, que se encargará de poner en todos los bordes de la imagen el píxel de color blanco.

#### 2.1.2. maximoSubmatriz

- **maximoSubmatriz** recibe en *rdi* el puntero a la imagen *src* y en *rsi* el *rowSize*. Carga en *xmm1* y *xmm2* las dos primeras filas de 4 píxeles apuntadas por *rdi* y llama a **maximoFilas**. Entonces, en *xmm1* obtenemos los componentes del píxel de sumatoria máxima repetidos cuatro veces.
- Se guarda el valor de *xmm1* en el registro *xmm0*.

- Se repite el proceso anterior para obtener el píxel máximo de las últimas dos filas. Una vez obtenido este resultado, utilizando *blends* se pone en la parte baja del registro *xmm0* el píxel máximo de las primeras dos filas y el píxel máximo de las últimas dos filas.
- A continuación, se procede a definir el máximo entre los dos píxeles restantes. Se extienden las componentes de los píxeles (bytes) a words, excepto por la componente A que se anula previamente con una máscara precargada.
- Con sumas horizontales se calcula la suma de las componentes de los dos píxeles: en los primeros 16 bits se tiene la suma del primero (Y0) y en los siguientes 16 la suma del segundo (Y1). Se copia el registro a otro y se realiza un *shift* a izquierda de 16 bits en todo el registro de manera tal que en la primer word se tiene ceros y en la segunda word el valor de Y0. Se compara el registro original (con Y1 y Y0) por mayor contra el nuevo registro con el *shift* aplicado. De esta forma, la segunda word del registro en el que se guarda el resultado de la comparación tendrá unos si Y1 es mayor a Y0 o ceros si no. Con un *shift* aritmético a derecha se replica la máscara en los primeros 16 bits del registro y luego con un *shift* a izquierda se mueve la máscara a la segunda dword del registro.
- La máscara se aplica luego al registro original con los dos píxeles. Como resultado, el segundo píxel se anula si Y1 es menor o igual a Y0 o no se altera si es mayor. Luego, la máscara que se aplicó a la segunda dword original se vuelve a aplicar a la primera dword pero invertida: si Y1 es mayor a Y0, el primer píxel se anula o se mantiene igual en caso contrario.
- Se juntan los resultados con un *blend* de forma tal que en un registro se tiene en algunas de las primeras dos dwords el píxel que suma el máximo entre sus componentes. Aplicando una máscara ya calculada se limpia la parte alta del registro con un *and*. De esta forma, el registro tiene las componentes del píxel correspondiente en alguna de las primeras dos dwords y en el resto tiene ceros.
- Se aplican dos sumas horizontales de dwords sobre el mismo registro del paso anterior: el resultado es que el píxel que se busca está replicado en las cuatro dwords del registro.
- Se realiza un *or* con una máscara cargada en un registro *xmm* para volver a dejar el byte de transparencia en 255.

### 2.1.3. maximoFilas

- **maximoFilas** recibe en *xmm1* la primera fila y en *xmm2* la segunda fila de una matriz. Antes de empezar, cambia el valor del componente de transparencia de la primera fila de píxeles por 0 con la máscara *mascara.rgb*. Luego extiende los componentes de los primeros 4 píxeles de bytes a words con la instrucción **pmovzxbw** en los registros *xmm3* y *xmm4*, y suma horizontalmente con **phaddw** cada registro. Entonces, en *xmm3* almacena las sumatorias (Y0 y Y1) de los primeros 2 píxeles de la primera fila, y en *xmm4* las sumatorias (Y2 y Y3) de los últimos dos píxeles de la primera fila, con los resultados repetidos 4 veces.
- Se repite el paso anterior para los píxeles de la segunda fila. Como resultado, en *xmm5* se obtienen las sumas Y4 y Y5, y en *xmm6* Y6 y Y7.
- Combina los registros que almacenan las sumas con **blends** de manera tal que quede en el registro *xmm3* las sumas de los 8 píxeles (los 4 de la primera fila y los 4 de la segunda). Luego, en *xmm4* guarda una copia de la máscara **max\_word**, una máscara de words con el valor máximo que puede alcanzar una word repetido 8 veces.
- Decidimos calcular el máximo de la siguiente manera: vamos a restarle a las sumas máximas el máximo valor que puede alcanzar una word. De esta forma, el valor máximo pasa a ser el valor mínimo y podemos usar la instrucción **phminoposuw** para encontrarlo horizontalmente. Entonces, la función le resta a *xmm4* los valores de *xmm3* (el registro con las sumas) y almacena el resultado en *xmm4*.

- Aplica **phminposuw** sobre *xmm4* y almacena en la *word* menos significativa de *xmm3* el valor mínimo (que indica la máxima suma entre los *Y<sub>i</sub>*) y su índice en el tercer *byte* del registro, que coincide con el índice del píxel si pensamos los ocho píxeles de entrada como un vector. En *r14* se guarda una copia del valor del índice.
- El primer paso es identificar la fila a la que pertenece el píxel con la suma máxima. Para esta operación, con *shifts* a izquierda se deja el tercer bit (más significativo) de la posición que se obtiene con **phminposuw** en el último bit del registro. Este bit indica efectivamente si el píxel pertenece a la primera (0) o a la segunda (1) fila.  
Aplicamos un *shifts* aritmético a derecha de 31 bits: la dword más significativa tiene ahora ceros si el píxel se encuentra en la primera fila o unos si se encuentra en la segunda fila.
- Con una operación de *shuffle* replicamos la máscara de la fila en todo el registro. Aplicamos la máscara (*and*) al registro con los píxeles de la segunda fila y luego aplicamos la inversa de la máscara al registro con los píxeles de la primera fila. De esta forma, el registro que contiene al píxel máximo permanece intacto y el otro registro se anula.
- Se suman los dos registros (uno es nulo) para tener en un único registro la fila con el píxel máximo.
- Utilizamos los bits menos significativos de la posición del píxel máximo obtenida con **phminoposuw** para acceder a la máscara correspondiente cargada en memoria: para cada una de las cuatro posibilidades (2 bits) tenemos una máscara de 16 bytes preparada. Cada máscara se encarga de mantener el píxel en la posición correspondiente intacta y limpiar el resto de píxeles.
- Se aplica la máscara que se trae de memoria (en el paso anterior) al registro con la fila del píxel máximo con un **pand** para dejar pasar únicamente el píxel que corresponda a *xmm1*.
- Finalmente, en *xmm1* se guardan los componentes del píxel de sumatoria máxima repetidos cuatro veces.

#### 2.1.4. bordeBlanco

- Esta función se encarga de recorrer los bordes de la imagen destino y cambiar los valores de los componentes de los píxeles a 255, mediante el uso de la máscara *blanco*.
- Primero recorre la fila superior y cambia los valores a blanco en el ciclo inicial, procesando los píxeles de cuatro en cuatro.
- Al terminar, retrocede 1 píxel y comienza el ciclo siguiente para cambiar los bordes laterales. Extrae el valor 255 al último píxel de la fila y al siguiente, ya que están contiguos en memoria. Luego, avanza al final de la siguiente fila y repite este paso.
- Al finalizar con los bordes laterales, retrocede 3 píxeles para empezar a recorrer la última fila desde su final y cambiar los valores.

## 2.2. Filtro Gamma

El filtro Gamma consiste en alterar cada componente de los píxeles de la imagen fuente a partir de la siguiente ecuación, considerando el valor del parámetro *Gamma* como 2:

$$\text{Output} = 255 * \left(\frac{\text{Input}}{255}\right)^{1/\text{Gamma}} \quad (1)$$

La idea general de la solución es recorrer la matriz de a cuatro píxeles por ciclo y aplicarles a sus componentes la ecuación en simultáneo y almacenar los resultados en la imagen destino. Ya que no existen las operaciones de raíz o división para enteros, debemos transformar los datos a floats.

A partir de la ecuación (1), fijado *Gamma* = 2 tenemos:

$$\text{Output} = 255 * \left(\frac{\text{Input}}{255}\right)^{1/2} = 255 * \sqrt{\frac{\text{Input}}{255}} = \sqrt{\text{Input} * 255} = \sqrt{\text{Input}} * \sqrt{255} \quad (2)$$

En lugar de aplicar el algoritmo original, simplificamos las cuentas usando los resultados obtenidos en la ecuación (2). Esta versión realiza una menor cantidad de operaciones en punto flotante, evitando la división.

### 2.2.1. Algoritmo

A continuación detallamos el funcionamiento del filtro:

1. Antes de comenzar el ciclo, guardamos en *xmm11* una máscara con el byte de transparencia en 255, y en *xmm12* cuatro veces el valor 255.0 (float) en dwords.
2. Aplicamos la raíz cuadrada al registro *xmm12* para tener calculado  $\sqrt{255}$ .
3. En cada iteración, vamos a mover cada píxel a su propio registro: extendemos con cero las componentes de cada píxel de *bytes* a *dwords* con **pmovzxbd** para luego convertirlas de valores enteros (*ints*) a números de punto flotante de precisión simple (*floats*) con **cvtqd2ps**.
4. Calculamos la raíz cuadrada de cada componente de los píxeles con **sqrtps** y luego multiplicamos cada uno por  $\sqrt{255}$  (calculado en *xmm12*) con **mulps**.
5. Convertimos los componentes transformados de cada píxel con **cvtps2dq**. Después, los empaquetamos: primero de *dwords* a *words* con **packusdw**; y luego de *words* a *bytes* con **packuswb** obteniendo nuevamente los cuatro píxeles de forma contigua con sus componentes en bytes.

Este procedimiento se repite para toda la imagen, tomando de a cuatro píxeles por vez. Si bien los cálculos se realizan de a un píxel por vez dentro del ciclo por la necesidad de tener los componentes en punto flotante para el cálculo de la raíz, se hacen sobre cuatro píxeles en cada iteración para poder aprovechar las instrucciones de empaquetado y realizar la menor cantidad de accesos a memoria para escribir resultados.

## 2.3. Filtro Broken

El filtro Broken consiste en alterar el orden de los componentes de cada píxel con el objetivo de romper la imagen. Utiliza como entrada un arreglo de 40 números y shiftea los componentes de los píxeles según sea indicado el arreglo. El arreglo es el siguiente:

$$a[40] = [0, -4, 4, 8, 4, -4, 4, 8, 0, -4, 4, 8, -4, 0, 4, -4, -4, 4, 16, 32, 4, 0, 4, -4, -8, -16, 0, 8, 0, 4, -4, 0, 0, 4, 0, 16, 32, 16, 8, 4].$$

Los movimientos que realiza para cada componente de un píxel son:

$$jr = (j+8*width + a[(i+10) \bmod 40]) \bmod width,$$

$$jg = (j+8*width + a[(i+20) \bmod 40]) \bmod width,$$

$$jb = (j+8*width + a[(i+30) \bmod 40]) \bmod width,$$

...donde *j* es el *width* e *i* el *height* de la imagen.

Para resolver este filtro de forma más cómoda, decidimos precalcular en una tabla las posiciones  $a[(i+10) \bmod 40]$ ,  $a[(i+20) \bmod 40]$ ,  $a[(i+30) \bmod 40]$  del arreglo de forma tal que al querer acceder a los valores del arreglo *a* para un cierto valor de *i*, se pueda hacer en una única instrucción. A este arreglo lo llamaremos *aModulo*. Es un arreglo de enteros de 3 bytes de 40 posiciones. El primer byte de cada elemento es el correspondiente  $a[(i+10) \bmod 40]$ , el segundo  $a[(i+20) \bmod 40]$  y finalmente el tercero

$a[(i+30) \bmod 40]$ . De esta manera, podemos obtener el elemento del arreglo correspondiente a cada componente a modificar sin tener que calcularlo en el ciclo en sí.

### 2.3.1. Consideraciones previas

Uno de los problemas principales con el que nos encontramos a la hora de implementar el filtro en Assembler es el de garantizar que el índice siga apuntando a un píxel válido de la misma fila que se está procesando. Como vimos, los valores del arreglo  $a$  están acotados. En particular, el menor valor es -16 y el máximo es 32. Cuando sumamos estos valores con  $j$ , es decir la posición del píxel en la fila, podemos obtener un valor negativo (si  $j$  es muy chico) o un valor mayor al número de columnas (si  $j$  es muy grande). En el primer caso, tomar módulo  $width$  es lo mismo que sumarle  $width$  al valor negativo que obtuvimos. En el segundo caso, basta restar por  $width$ . Con estas simplificaciones, se obtiene el mismo resultado y evitamos realizar operaciones más complejas para resolver esta parte del algoritmo.

En el pseudocódigo original, se utiliza una operación de saturación a la componente R, G o B antes de ser copiada al píxel correspondiente. En la implementación en Assembler, no la realizamos porque no es necesario: el byte que se está copiando es una componente válida por ser una componente de otro píxel. El valor de estas componentes no se altera, por lo tanto no es necesario saturar el resultado.

### 2.3.2. Algoritmo

Detallamos el procedimiento del filtro a continuación:

1. En los registros  $xmm9, xmm10, xmm11, xmm12, xmm13, xmm14$  y  $xmm15$  cargamos las máscaras, en  $r10d$  y  $r9d$  guardamos el  $height$  y  $width$  respectivamente.
2. Armamos en  $xmm4$  un registro que tenga 4 dwords con el valor del  $width$ . En  $r12$  guardamos el puntero al arreglo  $aModulo$  y usamos  $r13$  como offset para obtener los elementos del arreglo.
3. Utilizaremos  $ecx$  y  $edx$  como contadores para las filas y columnas en el ciclo externo e interno respectivamente.
4. Antes de entrar al ciclo interno, vamos a mover a  $xmm3$  los 3 elementos como dwords (y un valor extra al final del registro que no nos interesa) correspondientes de  $aModulo$  utilizando  $r13$  como offset ( $r13$  actuá como  $i \bmod 40$ ).
5. Dentro del ciclo de las columnas, el primer paso es el de analizar si los índices de los píxeles del que se deben tomar las nuevas componentes están en rango. Se realiza una primera comparación por menor a cero  $pcmpgtqd$ . Aplicamos la máscara que resulta de la comparación al registro con  $width$  en las cuatro dwords y luego sumamos al registro de índices este resultado. De esta forma, los índices que eran negativos ahora están nuevamente entre 0 y  $width - 1$ .
6. Análogamente al paso anterior, comparamos el vector de índices por mayor contra  $width$  y utilizamos la máscara con el registro que indica el  $width$ . Luego, le restamos al registro de índices este nuevo registro. Repetimos este paso pero comparando por igualdad. De esta forma, los índices que eran mayores al máximo ahora están nuevamente entre 0 y  $width - 1$ .
7. Una vez que tenemos en  $xmm3$  los índices de los píxeles que tenemos que usar para reemplazar las componentes del primer píxel que estamos procesando, pasamos el índice del píxel del que debemos tomar la componente R a un registro de propósito general con una instrucción de *extract*.
8. Usamos el registro con el índice como *offset* del puntero al principio de la fila actual para traer a un registro  $xmm$  los cuatro píxeles de los que tomamos la componente roja para los píxeles que se están procesando.
9. Aplicamos una máscara al nuevo registro para anular todas las componentes menos R y copiamos el resultado al registro  $xmm0$ .

10. Repetimos los últimos tres pasos para las componentes G y B. Una vez que se aplica la máscara para anular todas las componentes menos la que nos interesa, sumamos el resultado al registro *xmm0*. De esta forma, componemos el registro con el resultado final.
11. Escribimos la componente de transparencia en 255 con un *or*.
12. Movemos el registro *xmm0* roto.<sup>a</sup> la imagen destino.
13. Al finalizar el ciclo interno, incrementamos *r13* en 3 para que el *offset* de *aModulo* sea válido para la próxima iteración. Antes de continuar con la siguiente fila, verificamos que *r13* no se haya pasado de 120 (el máximo de *aModulo*). Si se pasa significa que llegamos al final del arreglo *aModulo*, entonces lo cambiamos a 0.

Este procedimiento se repite para toda la imagen, procesando de a cuatro píxeles por ciclo.

### 3. Comparación de rendimiento

En esta sección vamos a medir y comparar nuestras implementaciones con sus respectivas versiones en C. Con la ayuda de *Compiler Explorer* vamos a transformar las implementaciones en C a código ASM para comparar, y luego realizaremos mediciones de 50 repeticiones para cada filtro en ASM y C, analizando los resultados con boxplots.

Utilizaremos las siguientes optimizaciones para el compilador: *O0*, *O1*, *O2*, *O3*, *Os*, *Ofast*, *Og*<sup>1</sup>.

- *O0*: la optimización por default, reduce el tiempo de compilación.
- *O1*: optimización mínima, reduce el tamaño del código y el tiempo de ejecución. No se realizan optimizaciones que aumenten significativamente el tiempo de compilación.
- *O2*: prende más flags de optimización en comparación con *O1*, aumenta el tiempo de compilación y la performance del código generado.
- *O3*: habilita todas las optimizaciones de *O2* y permite más optimizaciones.
- *Ofast*: realiza todas las optimizaciones de *O3* y también las que no son válidas para programas standard-compliant.
- *Og*: optimización orientada al debugging. Realiza optimizaciones que no interfieren con las opciones de debugging.
- *Os*: optimización orientada a reducir el tamaño del código generado. Realiza todas las optimizaciones de *O2* menos las que incrementan el tamaño del código.

Aplicaremos el filtro a tres imágenes con las siguientes características: una de 32x16 píxeles de tamaño, otra de 800x600 y una última de 1920x1200. Tanto las comparaciones como los experimentos serán corridos en una computadora con un procesador Intel(R) Core(TM) i5-4460 CPU @ 3.20GHz y microarquitectura haswell. Los gráficos de las comparaciones pueden replicarse abriendo el archivo *comparacion-C-ASM.ipynb*. A su vez, algunas las imágenes referenciadas se encuentran en la sección *apéndice*.

#### 3.1. Comparación Max

##### 3.1.1. Comparación del Código

Vamos a analizar la cantidad de instrucciones y accesos a memoria que hace cada código para calcular el píxel de componentes con sumatoria máxima de una matriz de 4x4 : Click Compiler Explorer del Filtro Max sin optimizaciones

---

<sup>1</sup><https://gcc.gnu.org/onlinedocs/gcc-11.1.0/gcc/Optimize-Options.htmlOptimize-Options>

- El código en C calcula píxel a píxel la sumatoria en una instrucción y usa 3 accesos a memoria para conseguir los valores de los componentes. En Compiler Explorer, esto se traduce a 38 instrucciones y 3 accesos a memoria. Entonces, para calcular el máximo de una matriz de 4x4 píxeles(16 píxeles), realiza  $38 \times 16$  instrucciones= 608 y  $3 \times 16 = 48$  accesos a memoria. Contando las instrucciones que tarda para actualizar el contador, toma 37 instrucciones. En el peor caso lo tiene que hacerlo 16 veces. Luego, realiza otros 3 accesos a memoria y 35 instrucciones para cambiar los valores de la imagen dst.
- Mientras tanto, el código de *maximoFilas* en ASM para calcular el máximo de dos filas (8 píxeles) toma 52 instrucciones y no hace ningún acceso a memoria. Luego, en *maximoSubmatriz* realiza 4 accesos a memoria, llama a la función anterior 2 veces (104 instrucciones) y todo el resto de la función toma 28 instrucciones. En total ,132 instrucciones y 4 accesos a memoria para procesar 16 píxeles. Luego en el cuerpo principal del filtro, usa 2 instrucciones y 2 lecturas a memoria para cambiar los valores de la imagen dst.

En resumen, para procesar cuatro píxeles, en C toma **647 instrucciones y 51 accesos a memoria** en el mejor de los casos (solo cambia el contador una vez), mientras que en ASM hace **134 instrucciones y 6 accesos a memoria**.

Teniendo esto en cuenta, vamos a comparar el rendimiento para los filtros aplicándolos sobre imágenes.

### 3.1.2. Comparación de rendimiento

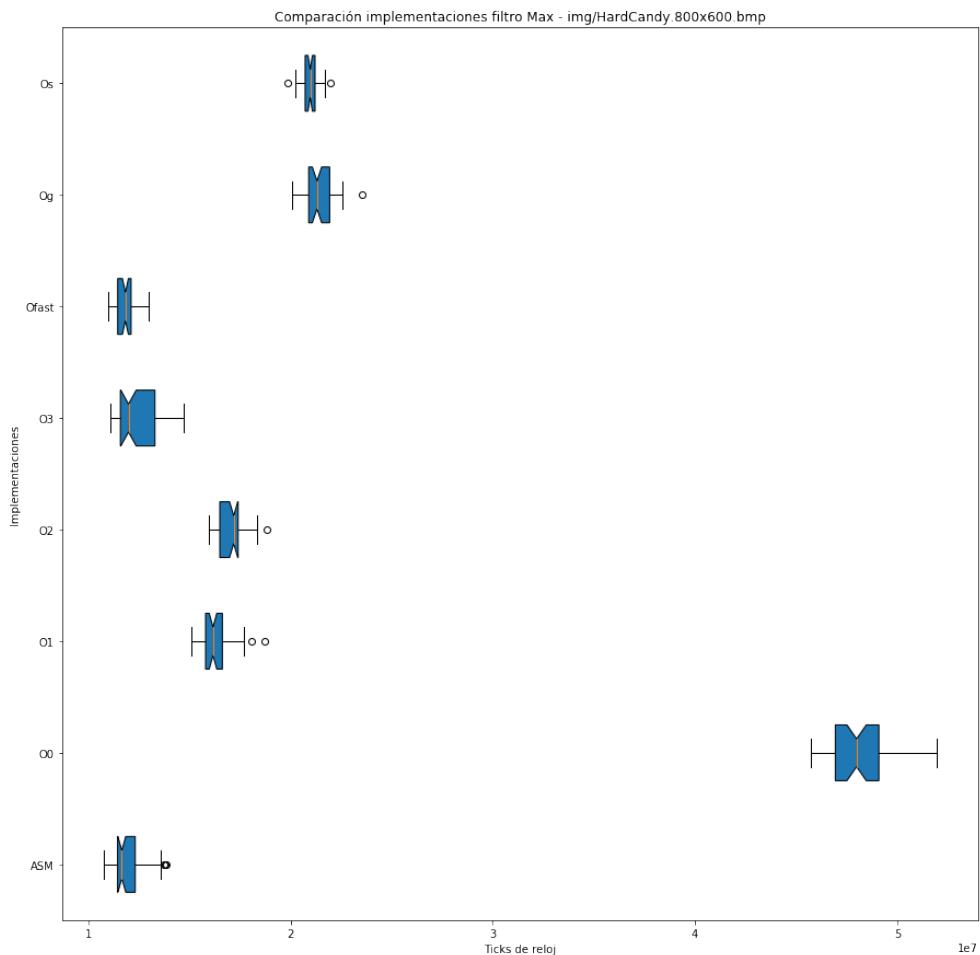


Figura 2: Max ASM vs C en 800x600

## Sección 3.1 Comparación Max

En los gráficos de comparación de rendimiento para el filtro Max (figuras 9, 10 y 11, ver apéndice A) podemos notar un comportamiento similar entre sí. La optimización O0 nunca llega a parecerse a la implementación en Assembler, mientras que el resto tienden a acercarse cada vez más a medida que aumenta el tamaño de la imagen.

En la figura 10 se puede ver que la optimizaciones O3 y Ofast tiene un buen rendimiento comparadas a la implementación en Assembler. Para el caso de Ofast, la dispersión del tiempo de ejecución es menor respecto a la implementación en Assembler, la cantidad de ciclos de reloj se mantiene cercana a la media.

En la figura 11 se puede ver que las optimizaciones O3 y Ofast incluso ganaron en tiempo de ejecución contra la implementación en Assembler.

Realizamos una comparación más para ver en más detalle el comportamiento entre O3, Ofast y Assembler. La imagen tiene tamaño 2560x1920 píxeles.

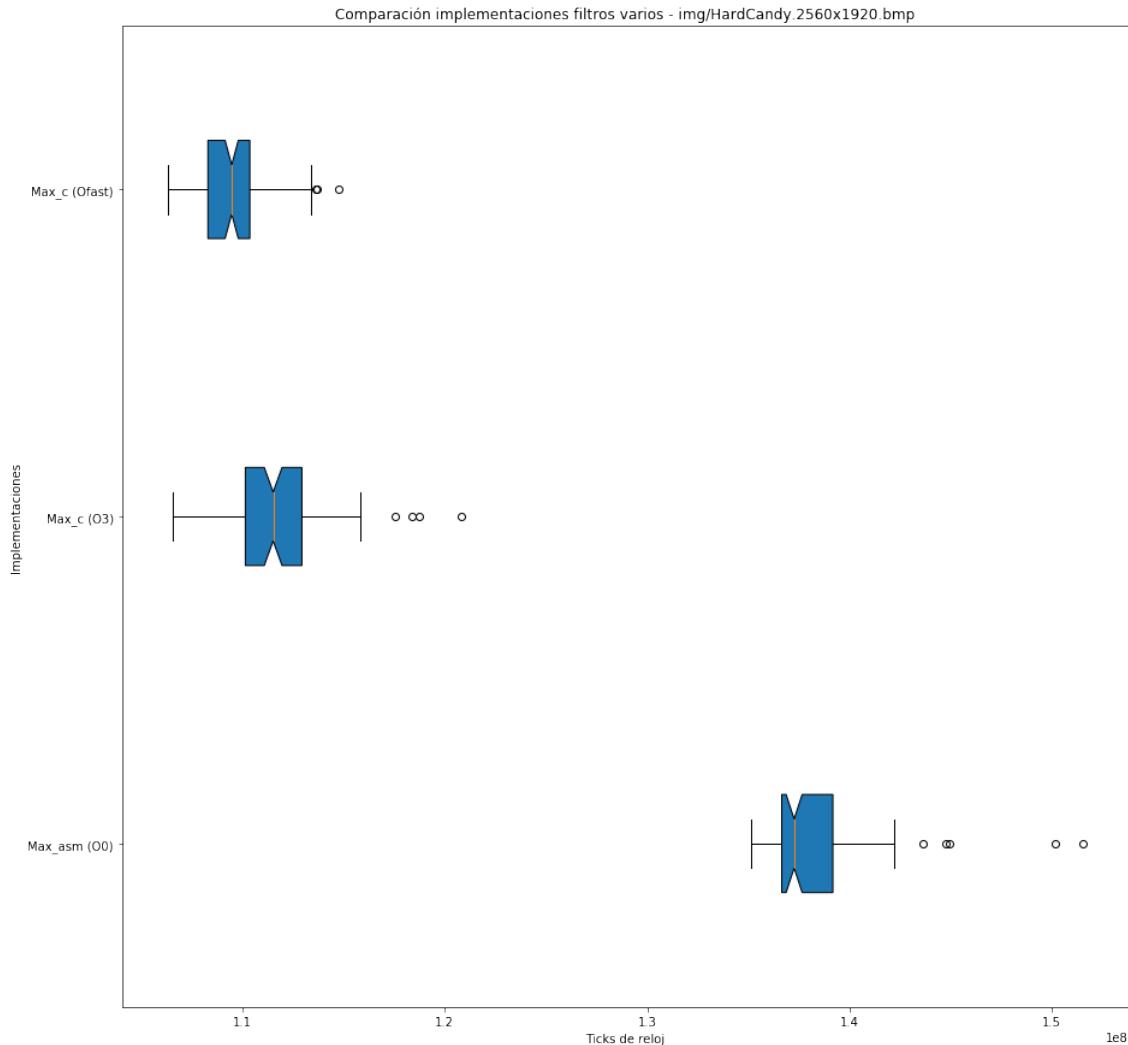


Figura 3: Max ASM vs O3 vs Ofast en 2560x1920

Como se puede ver en la figura 3, las implementaciones en C optimizadas funcionan mejor que la versión en Assembler. La diferencia respecto a lo que se puede ver en la figura 11 es incluso mayor.

## 3.2. Comparación Gamma

### 3.2.1. Comparación del Código

Contaremos las instrucciones del filtro Gamma (ambas implementaciones) para procesar 4 píxeles: Link Compiler Explorer del Filtro Gamma sin optimizaciones

- El código en C calcula y cambia píxel a píxel su nuevo valor, realizando 3 accesos a memoria para la imagen source y otros 3 para la imagen destino. En total, 3 instrucciones y 6 accesos a memoria para un único píxel. Esto se traduce a ASM como un total de 99 instrucciones por píxel. Entonces, para 4 píxeles toma  $99*4=396$  instrucciones y  $6*4=24$  accesos en memoria en calcular y cambiar. Y para calcular usa instrucciones de punto flotante, usa las cvtsi2sd, movsd,divsd, cvtssd2si.
- El filtro en ASM tarda en total 37 instrucciones y 4 accesos a memoria para procesar 4 píxeles de una. Las instrucciones de punto flotante que usa son cvtdq2ps, sqrtsp, mulps, cvtps2dq.

En síntesis, para procesar 4 píxeles, el código en C tarda **396 instrucciones y 24 accesos a memoria** en procesar 4 píxeles y cambiarlos, mientras que el filtro en ASM tarda **37 instrucciones y 4 accesos a memoria** en hacer lo mismo.

### 3.2.2. Comparación de rendimiento

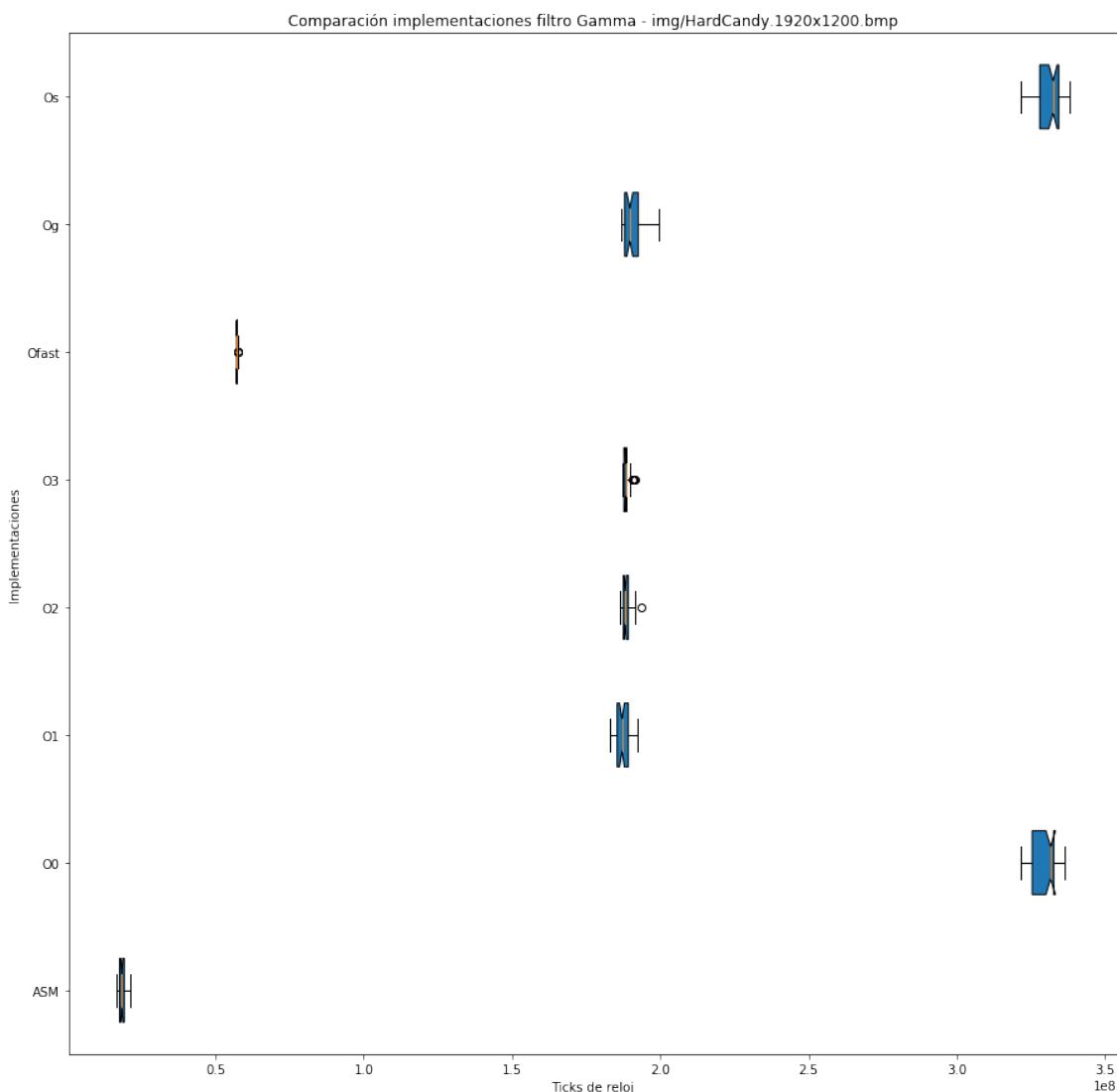


Figura 4: Gamma ASM vs C en 1920x1200

Podemos notar que la implementación en ASM siempre gana en los tres tamaños (figuras 14, 13 y 12), y la única optimización con un comportamiento similar es la Ofast. El resto no logra parecerse.

### 3.3. Comparación Broken

#### 3.3.1. Comparación del Código

Comparamos el procesamiento de 4 píxeles para el filtro Broken: (Click Compiler Explorer del Filtro Broken):

- En C, usa 3 instrucciones para mover los componentes de un píxel, con 3 lecturas a la imagen source y 3 al arreglo  $a$ . En ASM son 149 instrucciones y por píxel. Entonces, para 4 píxeles hará  $149 \times 4 = 596$  instrucciones y 12 lecturas a memoria.
- El filtro en ASM hace 41 instrucciones y 3 lecturas a memoria para procesar 4 píxeles.

Resumiendo, para 4 píxeles el filtro en C tarda **596 instrucciones y 12 accesos a memoria**, mientras que el filtro en ASM tarda **41 instrucciones y 3 lecturas a memoria** para realizar lo mismo.

#### 3.3.2. Comparación de rendimiento

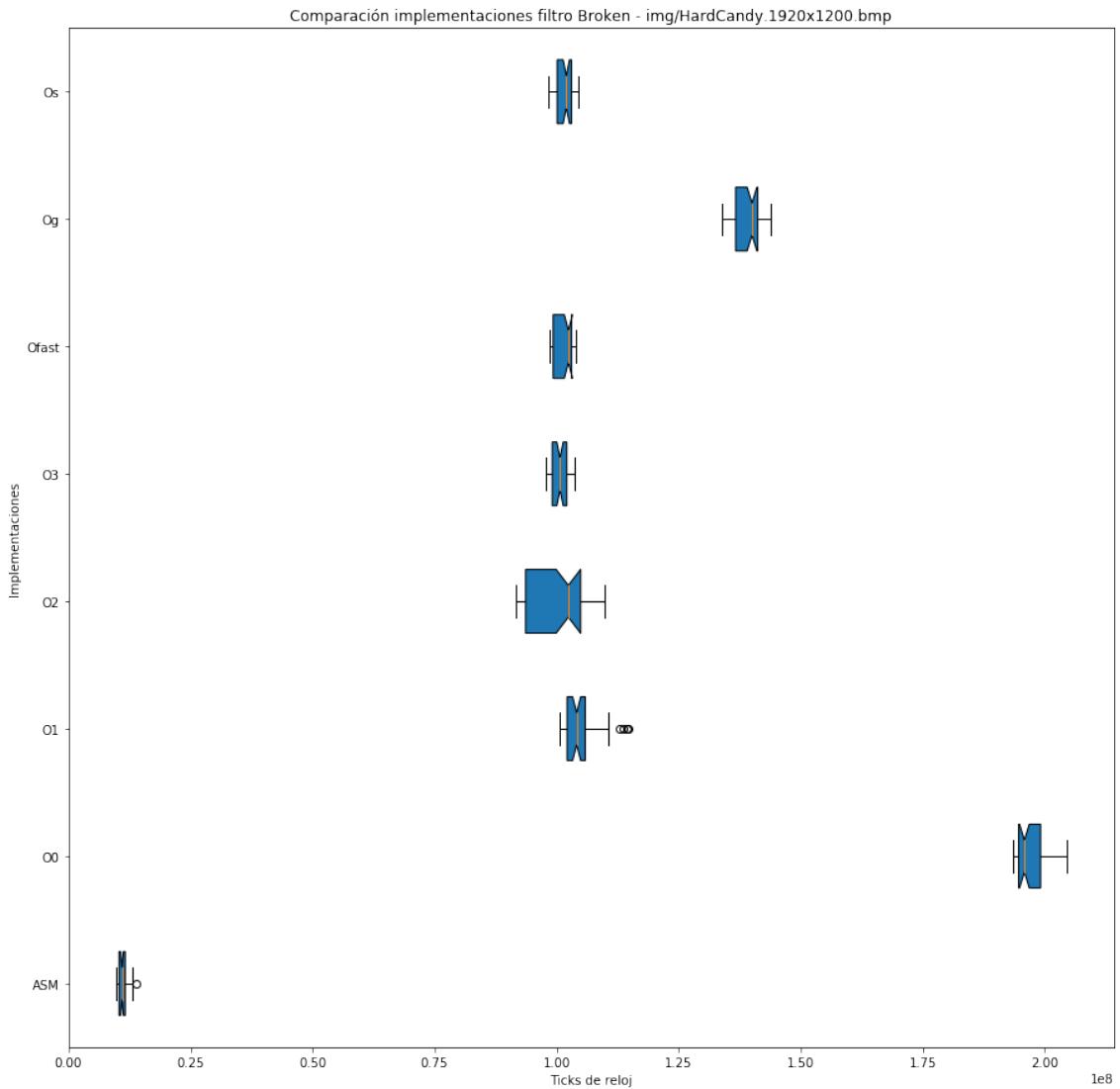


Figura 5: Broken ASM vs C en 1920x1200

Similar al filtro Gamma, en los tres casos (figuras 17, 16 y 15), la implementación en Assembler es superior en rendimiento respecto a la versión en C para todas las optimizaciones. Observando únicamente la figura 17, las mejores versiones optimizadas en términos de rendimiento toman aproximadamente diez veces más ciclos de reloj respecto a la implementación de Assembler.

## 4. Experimentos y resultados

### 4.1. Experimento 1: Filtro Gamma con vpgatherdd.

#### 4.1.1. Motivaciones

Al finalizar con la implementación del filtro Gamma, nos pusimos a pensar si existe alguna manera de evitar las instrucciones con punto flotante y instrucciones de conversión de enteros a punto flotante y viceversa, ya que estas instrucciones son (en general) notoriamente más lentas en comparación con las instrucciones de enteros, y a su vez conseguir el mismo resultado. Entonces, decidimos pensar la ecuación 2 como:

$$\sqrt{Input} * \sqrt{255} = \sqrt{Input * 255} \quad (3)$$

donde el rango de valores de *Input* es de 0 a 255: son todos los valores posibles que puede tomar cada una de las componentes R, G, B de una imagen porque se representan con un byte. En total son 256 valores que se pueden precalcular y guardar en una tabla, la cual llamaremos *lookup\_table*. Luego, para acceder a esos valores desde SIMD, existe la familia de las instrucciones **gather**, en donde en un registro *XMM* se especifican los índices, y la instrucción consigue los elementos de la tabla que corresponden con esos índices. La instrucción que usaremos se llama **vpgatherdd** y es lenta, sin embargo, nuestra hipótesis es que el tiempo que tarda será menor al que toman todas las instrucciones de *floats* que utilizamos en la versión original.

Usamos un simple programa en C para calcular todos los posibles resultados de la ecuación 3 convertido a enteros y cargamos los resultados en un arreglo de 256 posiciones en orden: la posición *i* del arreglo tiene el resultado de  $\sqrt{i * 255}$ . De esta forma, dada una componente de la imagen original, basta acceder al arreglo con el valor de esa componente.

La instrucción **vpgatherdd** puede cargar dwords y utiliza dwords como *offsets*, por lo tanto, en la implementación necesitamos convertir cada componente de cada píxel de byte a dword para poder utilizarlas como índices.

#### 4.1.2. Algoritmo Gama\_lookup\_table

Presentamos el pseudocódigo del algoritmo:

```
for i ← 0; i < height; i + + do
    for j ← 0; j < width; j + + do
        dst_matrix[i][j].r ← lookup_table[src_matrix[i][j].r]
        dst_matrix[i][j].g ← lookup_table[src_matrix[i][j].g]
        dst_matrix[i][j].b ← lookup_table[src_matrix[i][j].b]
    end for
end for
```

Explicamos a continuación la implementación:

1. En un principio, cargamos en *xmm11* y *xmm12* las máscaras que utilizamos para escribir en el byte de transparencia 255 y que necesita la instrucción **vpgatherdd**, y en *r9* almacenamos el puntero al primer elemento de *lookup\_table*. A su vez, dividimos *width* por cuatro porque en cada iteración del ciclo interno se procesarán cuatro píxeles.
2. Utilizaremos *ecx* y *edx* como contadores para los ciclos de las filas y columnas respectivamente.

3. Al iniciar el ciclo de las columnas, vamos a extender los componentes de los píxeles de bytes a *dwords* con **pmovzxbd** ya que la instrucción **vpgatherdd** toma *dwords*. Entonces, tenemos en *xmm1* el primer píxel extendido(P1), en *xmm2* el segundo(P2), *xmm3* el tercero(P3) y en *xmm4* el cuarto(P4).
4. La instrucción **vpgatherdd** funciona de la siguiente manera: usando los índices especificados en un registro *A*, obtiene los valores condicionados por una máscara *B*, y los guarda en un registro *C*. En nuestro caso, *A* serán los índices que indiquen los componentes de cada píxel *P*, *B* la máscara cargada en *xmm12*, y *C* un registro *xmm* en donde almacenamos los valores obtenidos.
5. Antes de utilizar la instrucción, movemos a *xmm13* el valor de la máscara *xmm12*. Debemos realizar este paso previo ya que **vpgatherdd** cambia los valores de la máscara a 0 luego de finalizar.
6. Usamos **vpgatherdd** para los píxeles, repitiendo el paso anterior antes de cada llamada. De esta forma, conseguimos en *xmm5* P1, en *xmm6* P2, *xmm7* P3 y *xmm8* P4.
7. Al terminar el paso anterior, pasamos los píxeles de *dwords* a *words* con **packusdw** a su vez uniendo P1 con P2 y P3 con P4, y finalmente unimos todos con **packuswb**, transformándolos de *words* a bytes.
8. Escribimos 255 en el byte de transparencia y finalmente movemos el resultado a la imagen *dst* apuntada por *rsi*.

#### 4.1.3. Comparación Gamma\_lookup\_table con su versión en C y Gamma original

Primero vamos a comparar esta nueva implementación del filtro contra su versión en C. Todas las comparaciones y gráficos pueden ser reproducidos en el archivo *Experimento\_Gamma.ipynb*.

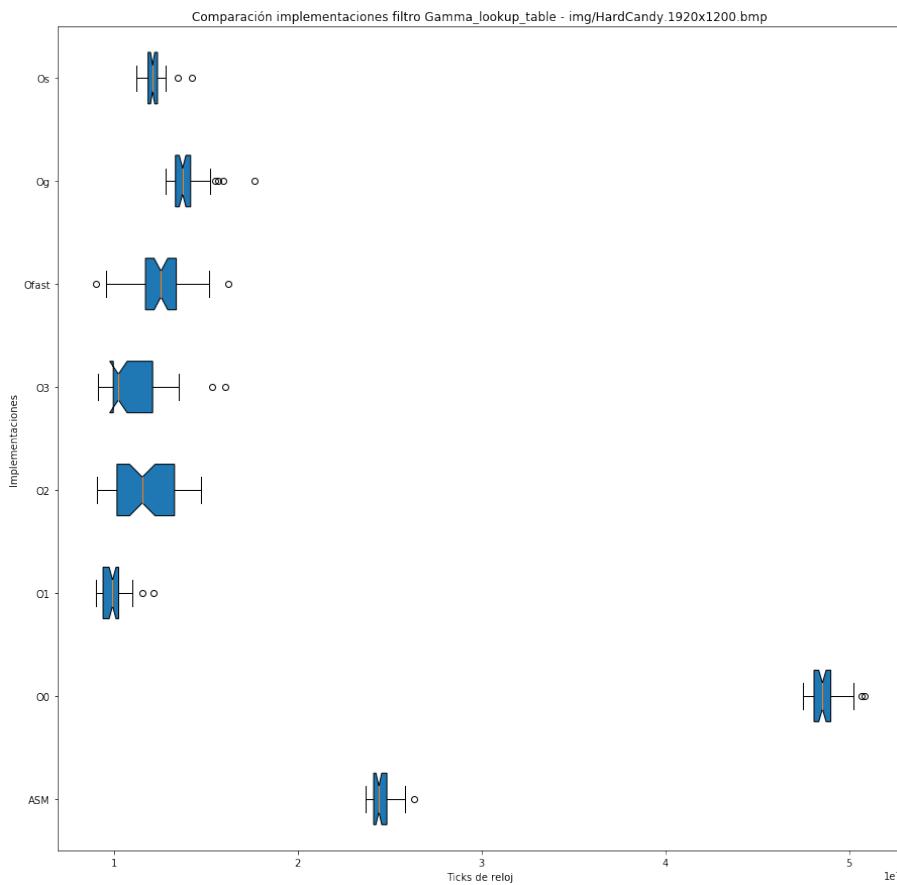


Figura 6: Gamma look\_up\_table ASM vs C en 1920x1200

En la comparación con la imagen de menor tamaño (figura 18) podemos ver que la implementación pierde ante todas las optimizaciones de C.

Luego, para tamaños más grandes (figuras 19 y 6) que la implementación en Assembler le gana a la optimización O0 de C, sin embargo, pierde ante el resto.

Ahora, compararemos `Gamma_lookup_table` en sus implementaciones en Assembler y optimizaciones Ofast y O3 de C contra el filtro Gamma original en Assembler.

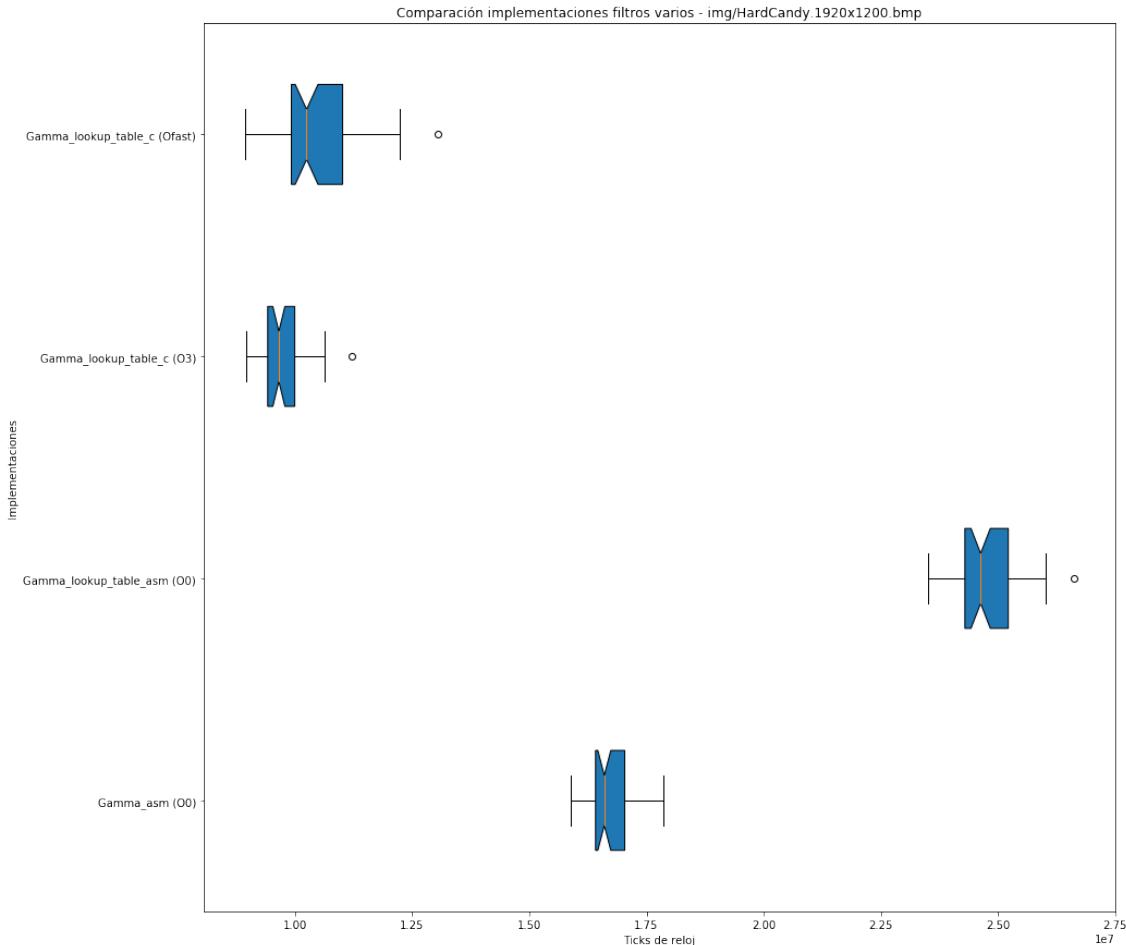


Figura 7: Gamma look\_up\_table vs Gamma original ASM y C

Comparando la implementación de `Gamma.lookup_table` en C contra el filtro `Gamma` original en ASM en la figura 7 podemos ver sorprendentemente que la implementación en C de `Gamma.lookup_table` tiene un mejor rendimiento que `Gamma` original en ASM, que estaba ganando en *performance* respecto a su misma implementación en C.

Intentaremos resolver y darle una explicación a este comportamiento.

#### 4.1.4. Análisis de resultados

Intentaremos identificar el problema con la ayuda de la herramienta LLVM Machine Code Analyzer<sup>2</sup>. Definimos los siguientes términos:

- Se define *RThroughput* como los ciclos de clock en los que tarda en comenzar la segunda instrucción. Por ejemplo, el *RThroughput* de la instrucción `vpgatherdd` es de 9. Esto significa que una nueva instrucción `vpgatherdd` puede empezar luego de 9 ciclos de clock desde que haya empezado la primera (si son independientes).

<sup>2</sup><https://llvm.org/docs/CommandGuide/llvm-mca.html>

- La *latencia* es el *delay* que genera la instrucción en la cadena de dependencias.

Analizamos con esta herramienta los cuerpos de los ciclos principales de cada implementación en Assembler para el caso de la microarquitectura haswell.

Link LLVM Machine Gamma Original, Link LLVM Machine Gamma `lookup` table

A primera vista, notamos que *Gamma.lookup.table* para un total de 100 iteraciones realiza un total de 9500 micro operaciones, mientras que el *Gamma* original para la misma cantidad de iteraciones realiza 2700 micro operaciones.

Un indicador de performance importante es el IPC, el IPC es la cantidad total de instrucciones simuladas dividido la cantidad total de ciclos de reloj. Para el filtro *Gamma.lookup.table* tenemos un IPC de 0.53 mientras que para el caso de *Gamma* el IPC es de 0.81

Luego, observando la información de las instrucciones, vemos que en *Gamma.lookup.table* la instrucción **vpgatherdd** genera una latencia de 16, toma 20 micro instrucciones y el *RThroughput* es de 4. En *Gamma* la latencia más alta es de 11 por la instrucción **sqrtps**, que tiene un *RThroughput* de 7.

La diferencia más notable es en la cantidad de micro operaciones. Para el caso de *Gamma*, todas las instrucciones toman una única micro operación excepto las instrucciones **pmovezbd** que toman dos. Para *Gamma.lookup.table*, las instrucciones de **vpgatherdd** se dividen en 20 micro operaciones.

Otra diferencia notable se encuentra en el *Timeline view*, en donde en *Gamma* puede ejecutar muchas más instrucciones en simultáneo. En *Gamma.lookup.table*, debe esperar a que se termine de ejecutar **vpgatherdd** para continuar con más instrucciones.

Utilizando la misma herramienta con el código generado por gcc en Compiler Explorer, analizamos el cuerpo del ciclo principal. En el código que se genera, los píxeles se procesan de a uno por vez en cada iteración. Link LLVM Machine Gamma `lookup` table C. Además, las únicas instrucciones que se pueden ver son de movimiento desde y a memoria y sumas para incrementar punteros.

Para 100 iteraciones, como en el caso anterior, la cantidad de micro operaciones total es de 1100 y el IPC es de 3.51. Todas las instrucciones se dividen en una micro operación y las instrucciones con un *latency* más alto son las instrucciones **movzx** con 5. Además, el *RThroughput* más alto es de 1, para las instrucciones **mov**.

En los siguientes links se pueden replicar los resultados obtenidos, para cada una de las implementaciones:

- *Gamma.lookup.table* ASM: <https://godbolt.org/z/r6vGEMc35>
- *Gamma.lookup.table* C (compilado con O3): <https://godbolt.org/z/jzcnreWTE>
- *Gamma* ASM: <https://godbolt.org/z/3j93MWMex>

#### 4.1.5. Conclusiones

Los resultados obtenidos parecen servir como uno de los pocos ejemplos en los que la programación vectorial, con instrucciones de tipo SIMD es peor en términos de *performance* respecto a una implementación estándar. Si bien es un caso particular, en donde todos los cálculos que se deben realizar pueden ser precalculados por tener entradas acotadas, el código generado por el compilador gcc de la versión en C arroja mejores resultados en tiempo de ejecución que ambas versiones de los filtros *Gamma*. Hasta este momento, en general las implementaciones de ASM que utilizan programación vectorial resultan superiores a sus versiones en C.

La instrucción **vpgatherdd**, que parecía ser una alternativa viable para acceder a memoria y traer muchos datos a la vez, resulta demasiado lenta como para ser usada en este caso. Las instrucciones de punto flotante del filtro original son lo suficientemente rápidas como para justificar su uso. Sin embargo, como vimos anteriormente, los experimentos indican que este podría ser un caso en el que optar por una implementación sin el uso de instrucciones SIMD es una decisión acertada.

## 4.2. Experimento 2: Alteraciones del Filtro Broken

### 4.2.1. Motivaciones

Al finalizar el filtro Broken, notamos que podríamos conseguir una mejora de rendimiento si evitamos tener que chequear si los índices se van de rango o no. En general, se busca evitar hacer la menor cantidad de operaciones de comparación en SIMD. Con esta idea, analizando el filtro Broken notamos que los únicos píxeles que pueden dar lugar a este problema de índices son los primeros 16 píxeles y los últimos 32 de cada fila: el valor menor en el arreglo  $a$  es -16, por la tanto, si los píxeles que tomamos tienen una posición en la fila mayor o igual a 16, al restarle 16 a la posición nos sigue quedando un valor en rango, y lo mismo pasa para los últimos 32 pero con la posibilidad de caer en una posición mayor al máximo.

En resumen, los píxeles que pueden traer problemas con los cálculos, son solo 48 (en cada fila). En general, se tienen imágenes de ancho mucho mayor a 48 píxeles, lo suficiente como para que un ligero cambio al aplicar el filtro en estos píxeles en los extremos de la imagen no sea visualmente perceptible por el usuario.

El objetivo principal del experimento es el de modificar la forma en la que se procesan los píxeles de los bordes laterales, buscando evitar comparaciones en cada iteración del ciclo para todos los píxeles de la imagen, cuando las comparaciones únicamente sirven para salvar casos borde.

Se quiere llegar a un filtro cuyo resultado sea visualmente indistinguible a efectos prácticos de una aplicación del filtro Broken original, pero con una ganancia en términos de tiempo de ejecución y cantidad de operaciones aprovechando técnicas de programación SIMD.

### 4.2.2. Algoritmo Broken alternativo

Una de las primeras ideas, fue la de definir arreglos alternativos, especiales para cada caso: para los píxeles centrales mantener el mismo arreglo  $a$ , para los primeros 16 píxeles de cada fila mantener los valores de  $a$  que sean positivos y anular los valores negativos: de esta forma, los valores que traen problemas ya no existen y se pueden procesar los píxeles sin comparaciones. Análogamente, para los últimos 32 píxeles definimos otro arreglo que mantiene los valores negativos y anula los valores positivos.

El filtro Broken alternativo procesa la imagen  $src$  de la siguiente manera:

- Para los primeros 4 píxeles de cada fila, utiliza la tabla  $aMenores$ , la cual toma los valores originales de la tabla  $aModulo$  pero con los valores negativos cambiados a 0.
- Para los últimos 32 píxeles de cada fila, utiliza la tabla  $aMayores$ , en donde cambian los valores positivos de la tabla  $aModulo$  a 0.
- Para los píxeles mayores a 4 y menores a 32, utiliza la tabla original  $aModulo$ .

Como resultado, la implementación realiza 3 ciclos para cada fila de píxeles: el primer ciclo  $cicloPrimerasColumnas$  en donde procesa los primeros 4 píxeles,  $cicloColumnasCentrales$  donde procesa los del "medio" y  $cicloColumnasFinales$ , donde procesa los últimos 32. Trabaja de forma similar al filtro original en cuanto al procesamiento de los píxeles, con la diferencia de que utiliza las tablas mencionadas anteriormente para cada caso por lo que evita verificaciones si los índices se van de rango o no.

### 4.2.3. Comparaciones Broken Alternativo ASM vs C

La implementación en C de este nuevo filtro agrega dos arreglos más:

$$\begin{aligned}aMenores[40] &= \{0,0,4,8,4,0,4,8,0,0,4,8,0,0,4,0,0,4,16,32,4,0,4,0,0,0,0,8,0,4,0,0,0,4,0,16,32,16,8,4\}; \\aMayores[40] &= \{0,-4,0,0,0,-4,0,0,-4,0,0,-4,0,0,-4,-4,0,0,0,0,0,-4,-8,-16,0,0,0,-4,0,0,0,0,0,0,0,0\};\end{aligned}$$

El pseudocódigo del algoritmo es el siguiente:

```
for i ← 0; i < height; i + + do
```

```
for j ← 0; j < width; j ++ do
    if j < 16 then
        dst_matrix[i][j].r ← src_matrix[i][j + aMenores[(i + 10) %40]].r
        dst_matrix[i][j].g ← src_matrix[i][j + aMenores[(i + 20) %40]].g
        dst_matrix[i][j].b ← src_matrix[i][j + aMenores[(i + 30) %40]].b
    else
        if j >= width - 32 then
            dst_matrix[i][j].r ← src_matrix[i][j + aMayores[(i + 10) %40]].r
            dst_matrix[i][j].g ← src_matrix[i][j + aMayores[(i + 20) %40]].g
            dst_matrix[i][j].b ← src_matrix[i][j + aMayores[(i + 30) %40]].b
        else
            dst_matrix[i][j].r ← src_matrix[i][j + a[(i + 10) %40]].r
            dst_matrix[i][j].g ← src_matrix[i][j + a[(i + 20) %40]].g
            dst_matrix[i][j].b ← src_matrix[i][j + a[(i + 30) %40]].b
        end if
    end if
end for
end for
```

Compararemos el rendimiento de filtro Broken\_alternativo contra su versión en C. Todas las comparaciones y gráficos pueden ser reproducidos en el archivo *Experimento\_Broken.ipynb*.

En las figuras 24, 25, 26 podemos ver que en general supera a las optimizaciones de C. Para la imagen más pequeña, todas tienen una media similar. A medida que crece el tamaño de la imagen, las optimizaciones *O0* y *Og* crecen en tiempo de ejecución mientras que el resto se mantienen cercanas al tiempo que toma la implementación de Assembler.

#### 4.2.4. Algoritmo Broken\_real\_offset

Una segunda idea, fue la de permitir que los *offsets* no se tomen en módulo *width*. En el filtro Broken, tomar módulo *width* asegura que los píxeles de los que se toman las componentes están en la misma fila de los píxeles que se están procesando. Como vimos anteriormente, tomar módulo únicamente es necesario para los píxeles en los bordes laterales.

Si pensamos en lo que sucede cuando por ejemplo estamos procesando el primer píxel de una fila ( $j = 0$ ) y el valor que indica *a* para cierta componente es negativa, si tomáramos ese *offset* como válido, estaríamos tomando un píxel al final de la fila anterior. Si el elemento de *a* que tenemos fuera -4, estaríamos buscando el elemento en la posición *width* - 4 de la fila anterior.

En el filtro original, tomar módulo sería lo mismo que sumar *width* al *offset* - 4: eso nos da el píxel en la misma fila en la posición *width* - 4. Entonces, sin tomar módulo estaríamos tomando el píxel que se encuentra una posición más arriba del píxel que tomáramos originalmente. Es decir, los píxeles vecinos.

Lo mismo sucede para el otro extremo, pero en lugar de tomar el vecino superior estaríamos tomando el vecino inferior.

Parece razonable pensar que los píxeles que están cercanos entre sí se parecen bastante, si pensamos en imágenes de tamaño no muy chico. En general tenemos imágenes donde los cambios de color no son drásticos píxel a píxel.

Bajo este concepto, decidimos realizar este filtro adicional de nombre *Broken\_real\_offset*. Como en el caso anterior, se busca una mejora en *performance* (aprovechando las ventajas de SIMD) con resultados visuales prácticamente iguales.

El filtro *Broken\_real\_offset* realiza lo siguiente:

- No modifica los primeros 4 píxeles de la primera fila (los copia en la imagen destino). Entonces comienza el ciclo principal desde el píxel 4 (contando los píxeles 0 a 3 como los primeros 4).
- Procesa la primera fila, luego el resto de las filas. Utilizamos el mismo arreglo *a* del filtro original, sin tomas módulo *width* como explicado.
- Procesa la última fila, dejando los últimos 32 píxeles de la imagen iguales.

La razón por la que tratamos de forma distinta los primeros 4 píxeles de la primera fila y los últimos 32 de las últimas es porque no tenemos fila anterior o siguiente a la que podemos recurrir para salvar los casos borde. Son 36 píxeles en total y se encuentran en los extremos de la imagen.

#### 4.2.5. Comparaciones Broken\_real\_offset ASM vs C

El código en C utiliza únicamente el arreglo *a* original. El pseudocódigo es el siguiente:

```

for i  $\leftarrow$  0; i < height; i + + do
    for j  $\leftarrow$  0; j < width; j + + do
        jr  $\leftarrow$  j + a[(i + 10) %40]
        dg  $\leftarrow$  j + a[(i + 20) %40]
        jb  $\leftarrow$  j + a[(i + 30) %40]
        if i == 0 and j < 4 then
            dst_matrix[i][j].r  $\leftarrow$  src_matrix[i][j].r
            dst_matrix[i][j].g  $\leftarrow$  src_matrix[i][j].g
            dst_matrix[i][j].b  $\leftarrow$  src_matrix[i][j].b
        else
            if i == height - 1 and j >= width - 32 then
                dst_matrix[i][j].r  $\leftarrow$  src_matrix[i][j].r
                dst_matrix[i][j].g  $\leftarrow$  src_matrix[i][j].g
                dst_matrix[i][j].b  $\leftarrow$  src_matrix[i][j].b
            else
                dst_matrix[i][j].r  $\leftarrow$  src_matrix[i][jr].r
                dst_matrix[i][j].g  $\leftarrow$  src_matrix[i][dg].g
                dst_matrix[i][j].b  $\leftarrow$  src_matrix[i][jb].b
            end if
        end if
    end for
end for
```

Compararemos el rendimiento de la versión en Assembler con su versión en C. Las figuras se encuentran en el apéndice. Si las observamos, las figuras 27, 28 y 29 podemos ver un comportamiento similar al filtro Broken\_alternativo.

#### 4.2.6. Comparaciones entre los tres filtros para Broken

En esta instancia, vamos a comparar los tres filtros implementados en Assembler y analizar los resultados. Haremos comparaciones de rendimiento y de imágenes resultantes.

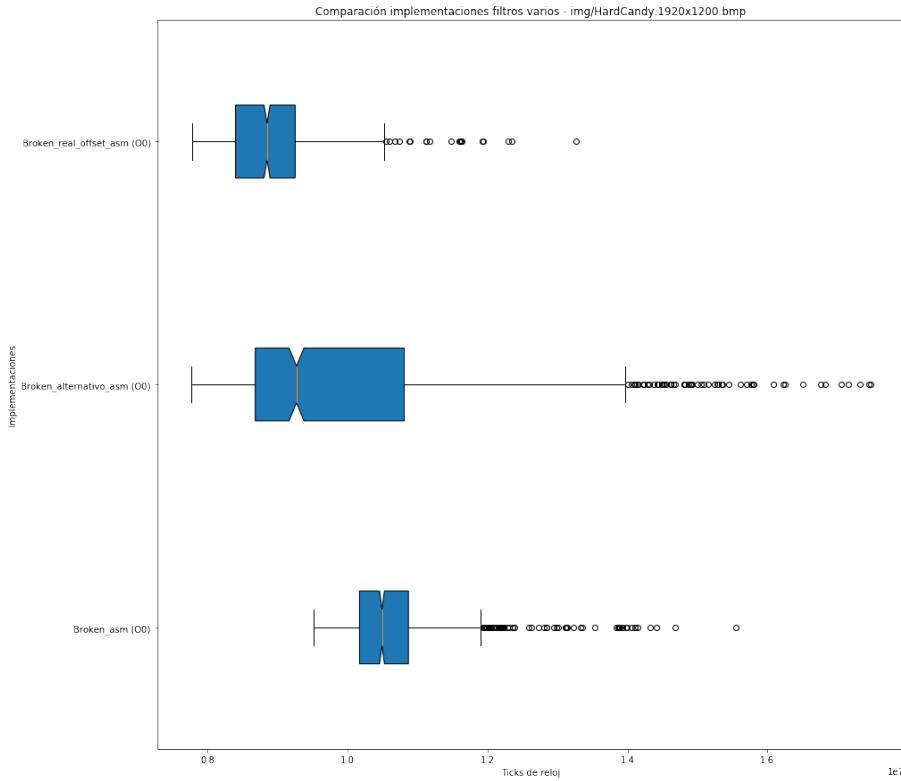


Figura 8: Comparaciones tres filtros broken 1920x1200

Observando la figura 32 se puede apreciar la mejora general en rendimiento del filtro *Broken\_real\_offset*, respecto de *Broken*. El filtro *Broken\_alternativo* en media también tiene una mejor performance que *Broken* pero tiene muchos *outliers*, la dispersión es mucho mayor a cualquiera de los otros dos filtros. En síntesis, podemos asumir que el filtro *Broken\_real\_offset* provee mejores resultados en términos de rendimiento.

Ahora vamos a comparar las imágenes resultantes. Aplicaremos los filtros a la imagen "Labyrinth.bmp" los resultados se encuentran en el apéndice.

Podemos observar que existen diferencias visuales en los bordes de la imagen entre el filtro original y el *Broken\_alternativo*. Mientras tanto, con el *Broken\_real\_offset*, apenas pueden verse diferencias. Por ello, nuevamente *Broken\_real\_offset* ofrece el mejor resultado.

#### 4.2.7. Conclusiones

A partir de los resultados obtenidos, el filtro *Broken\_real\_offset* parece una alternativa aceptable al filtro *Broken*. A partir de la figura 33, podemos decir que a efectos prácticos las imágenes resultantes son idénticas. En ámbitos donde el tiempo de ejecución es uno de los principales factores al momento de optar por una implementación sobre otra y en donde es aceptable obtener un resultado no del todo idéntico al esperado, *Broken\_real\_offset* es una implementación razonable.

El filtro *Broken\_alternativo* tiene en promedio mejor *performance* que *Broken* pero en general es más lento que *Broken\_real\_offset* y los resultados no son tan buenos. En la figura 33 se puede apreciar claramente la diferencia en los bordes de la imagen.

## 5. Conclusión

En el desarrollo del trabajo práctico encontramos tanto ventajas como desventajas con la programación vectorial en SIMD.

A la hora de comparar versiones implementadas en Assembler y C, encontramos resultados en su mayoría favorables hacia la programación vectorial en casos tales que se necesita realizar gran cantidad

de procesamiento de datos. Por ejemplo, en la versión original del filtro Gamma, en donde ambas implementaciones deben realizar cálculos en punto flotante a toda una imagen, su versión en Assembler permite realizar mayor cantidad de cálculos y procesamiento de píxeles por ciclo.

Una de las desventajas que encontramos es tal vez la complejidad agregada a la hora de programar. En casos donde se debe prestar especial atención en el análisis de casos especiales (como tener que resolver el hecho de que puede haber varios píxeles que suman el máximo entre sus componentes para el filtro Max), la programación vectorial puede generar problemas nuevos respecto a la programación a la que estamos acostumbrados.

Otra desventaja no menor es la claridad del código. No siempre resulta fácil entender las operaciones que se realizan sobre los datos. Por ejemplo, en general estamos acostumbrados a ver comparaciones y saltos condicionales para procesar datos con características distintas, mientras que una de las técnicas que se quiere evitar programando en SIMD es esta, que se puede reemplazar (a veces) generando máscaras en tiempo de ejecución.

En general, a pesar de las desventajas, podemos ver que la programación con instrucciones SIMD tiene una gran potencia y en muchos casos se puede ganar una diferencia muy grande en *performance* respecto a implementaciones que no las utilizan.

# Appendices

## A. Imágenes adicionales de las comparaciones de rendimiento entre C y ASM

### A.1. Filtro Max

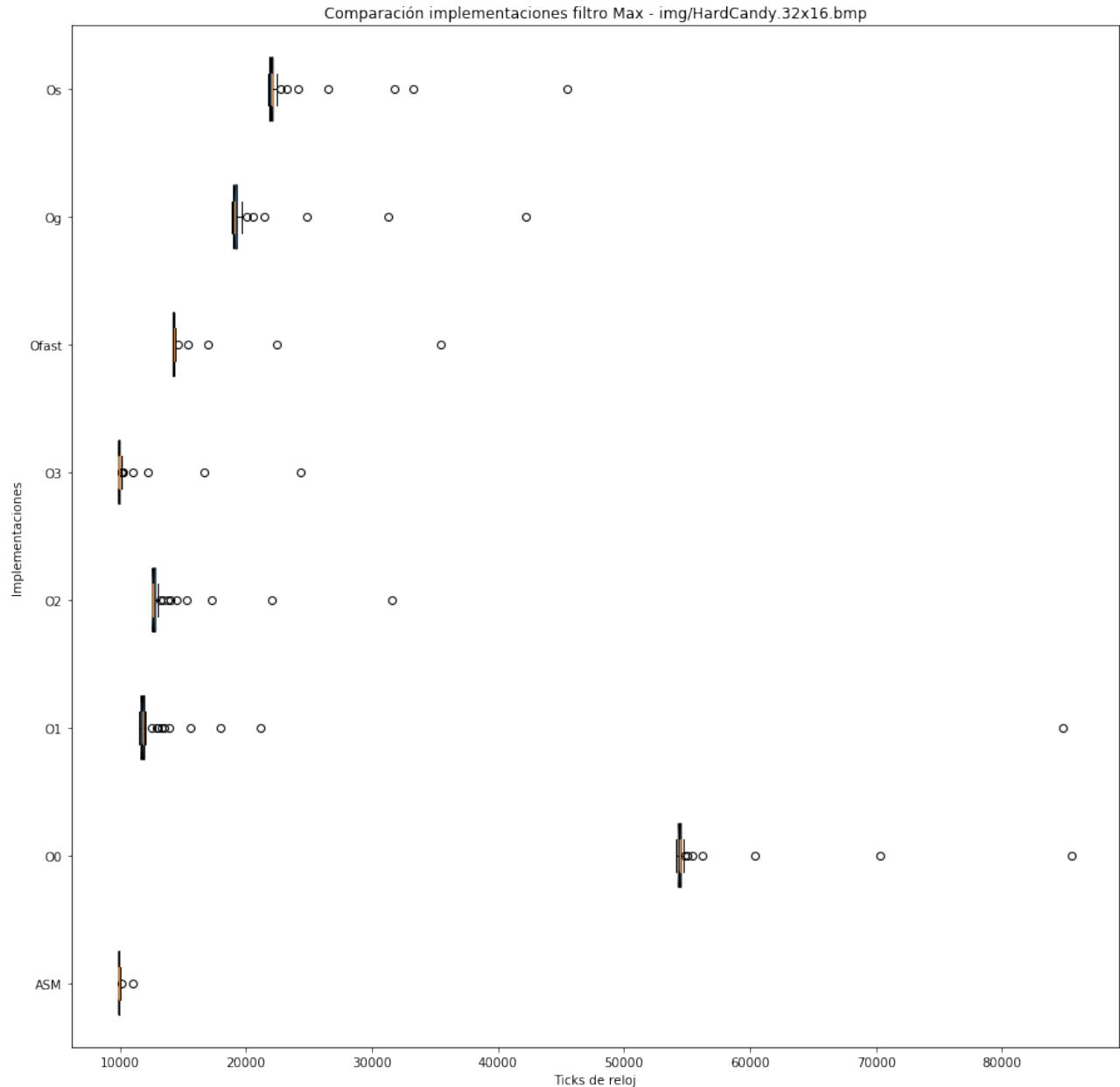


Figura 9: Max ASM vs C en 32x16

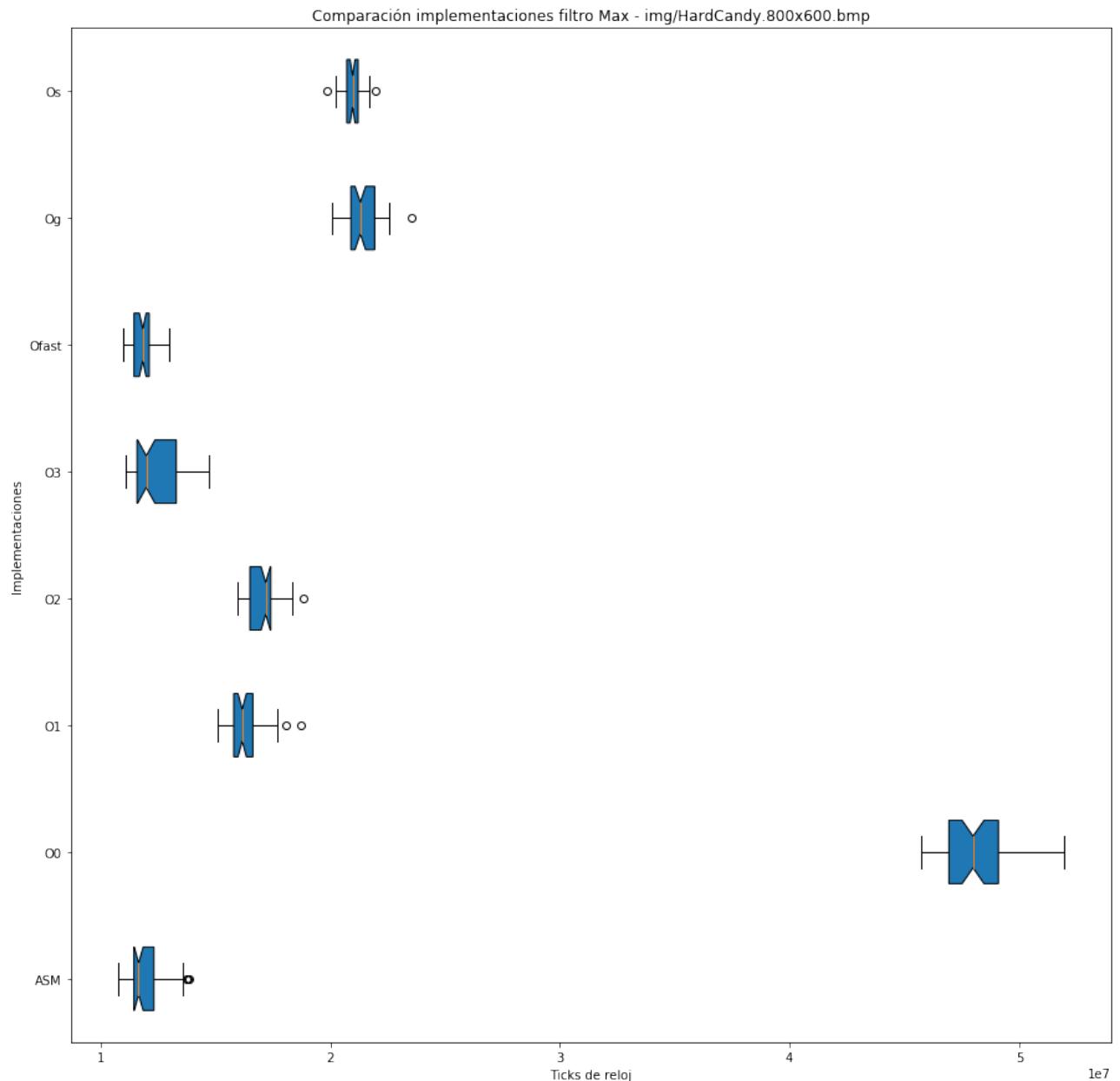


Figura 10: Max ASM vs C en 800x600

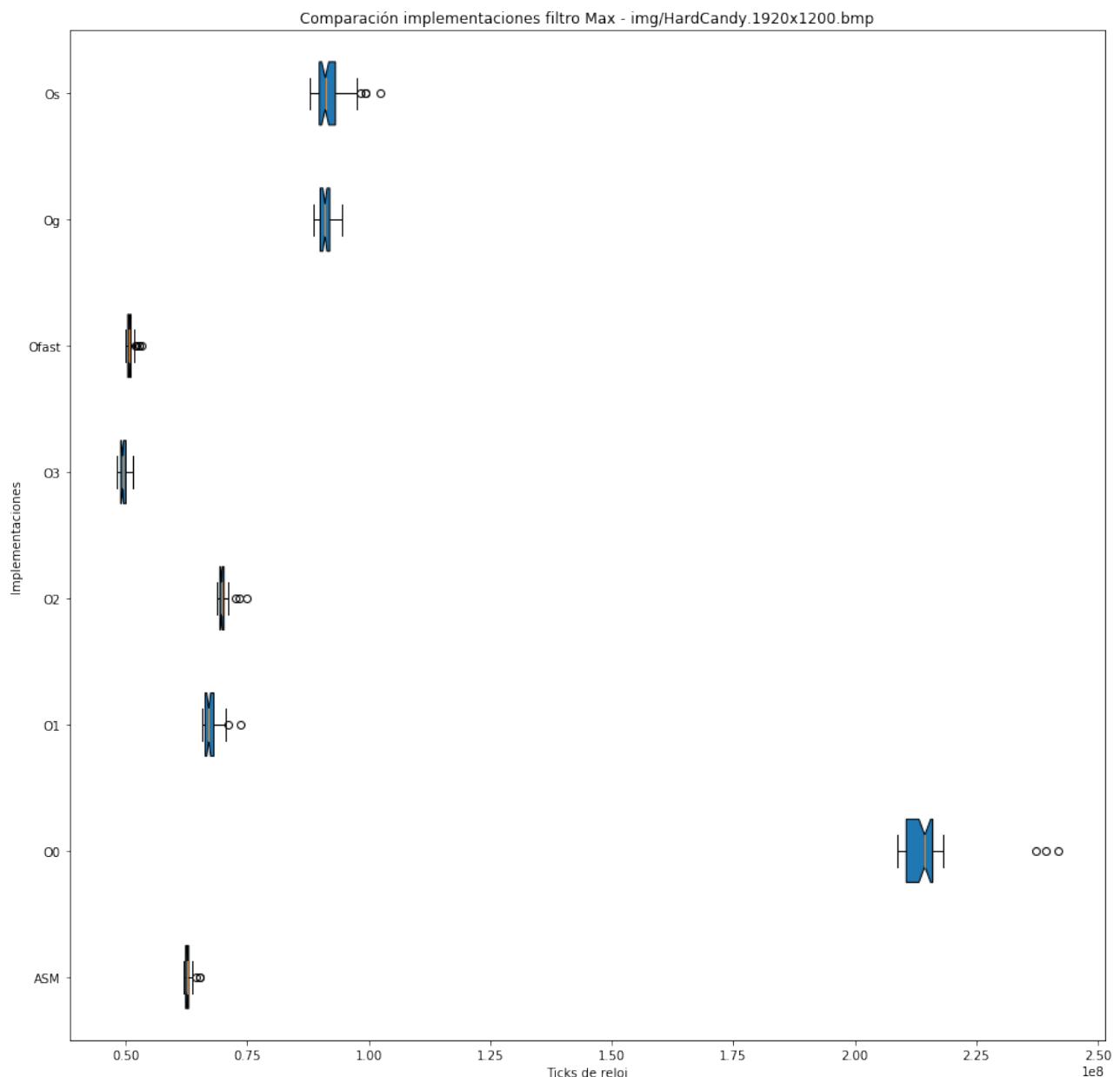


Figura 11: Max ASM vs C en 1920x1200

## A.2. Filtro Gamma

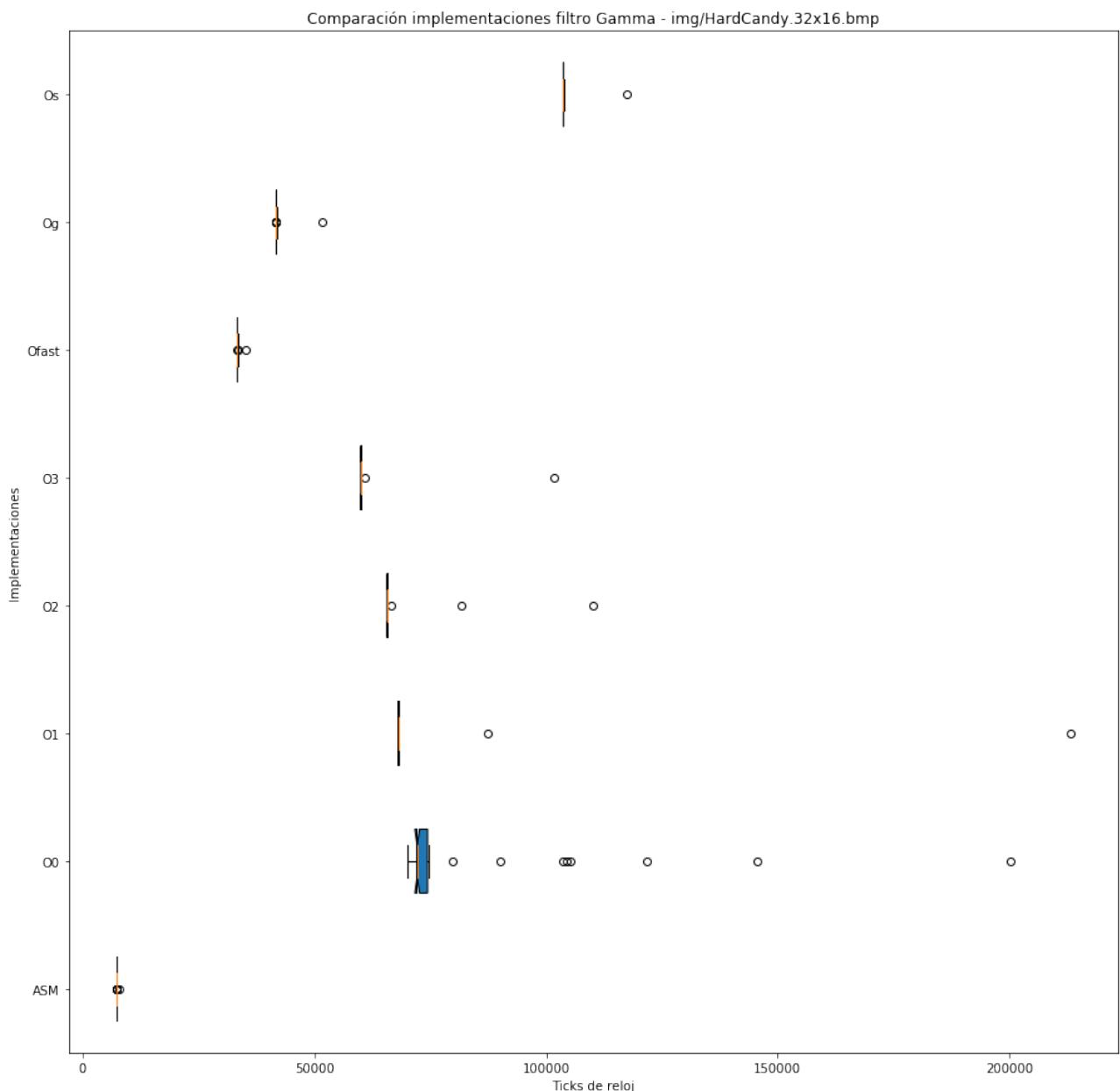


Figura 12: Gamma ASM vs C en 32x16

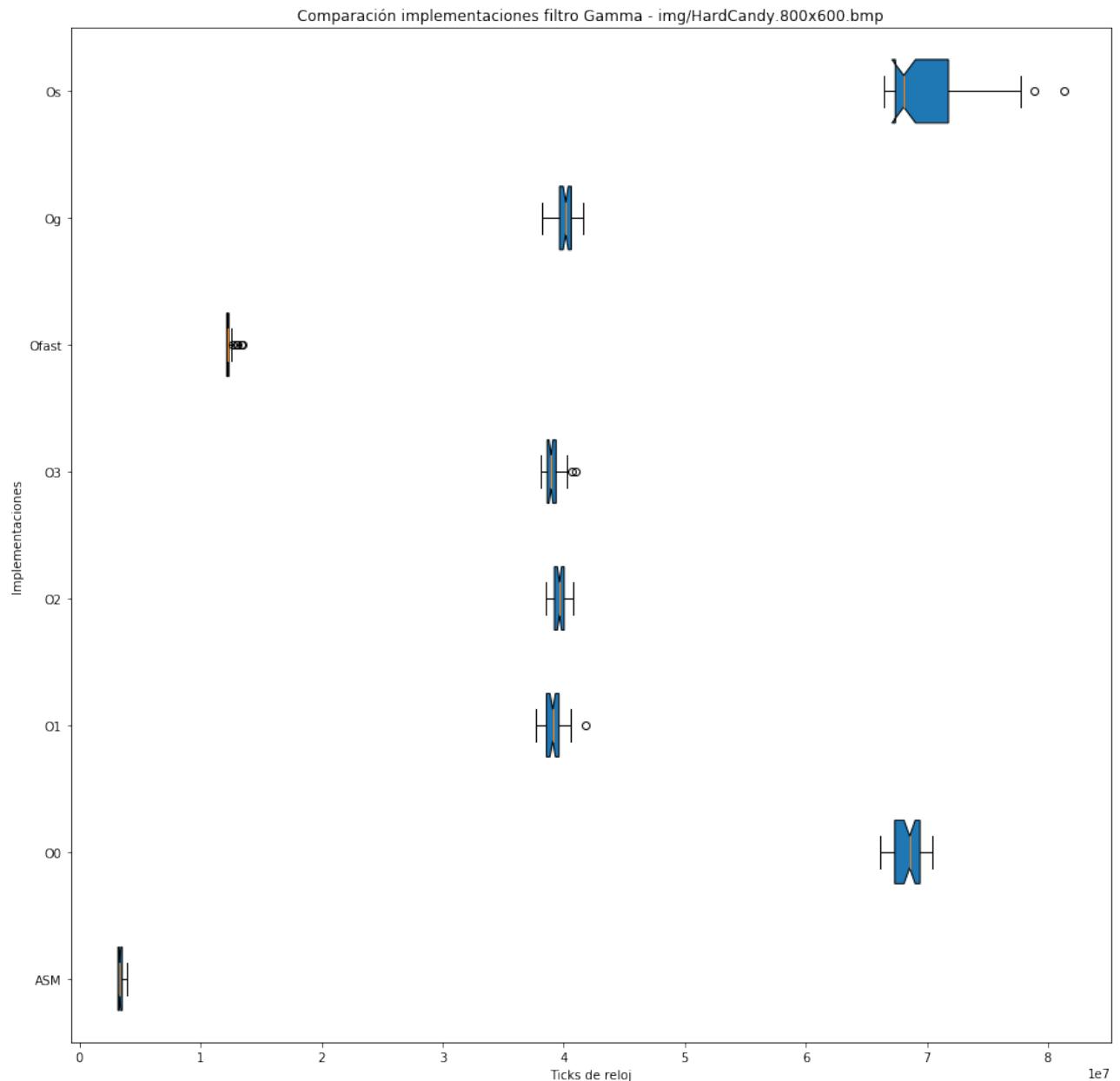


Figura 13: Gamma ASM vs C en 800x600

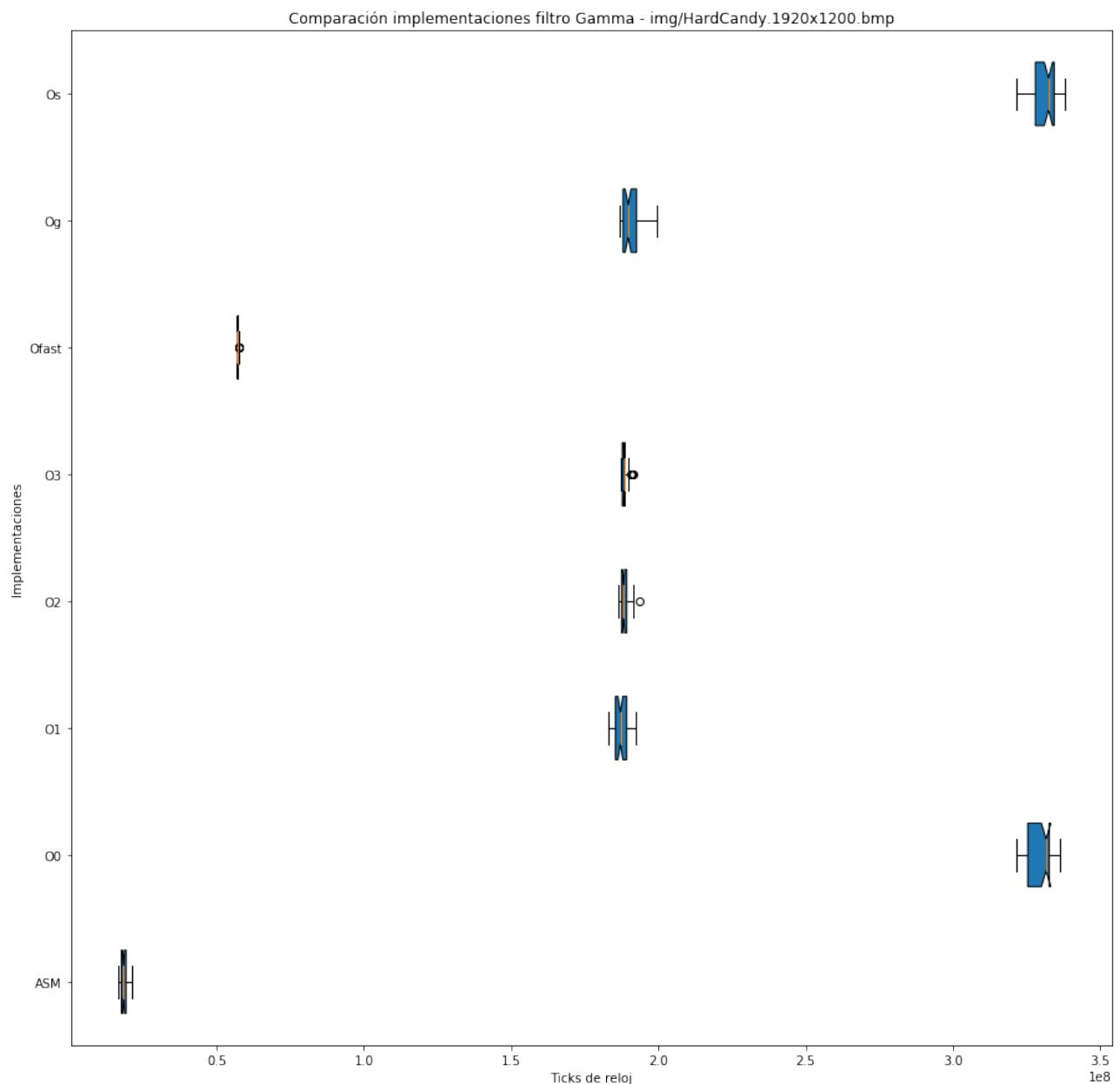


Figura 14: Gamma ASM vs C en 1920x1200

### A.3. Filtro Broken

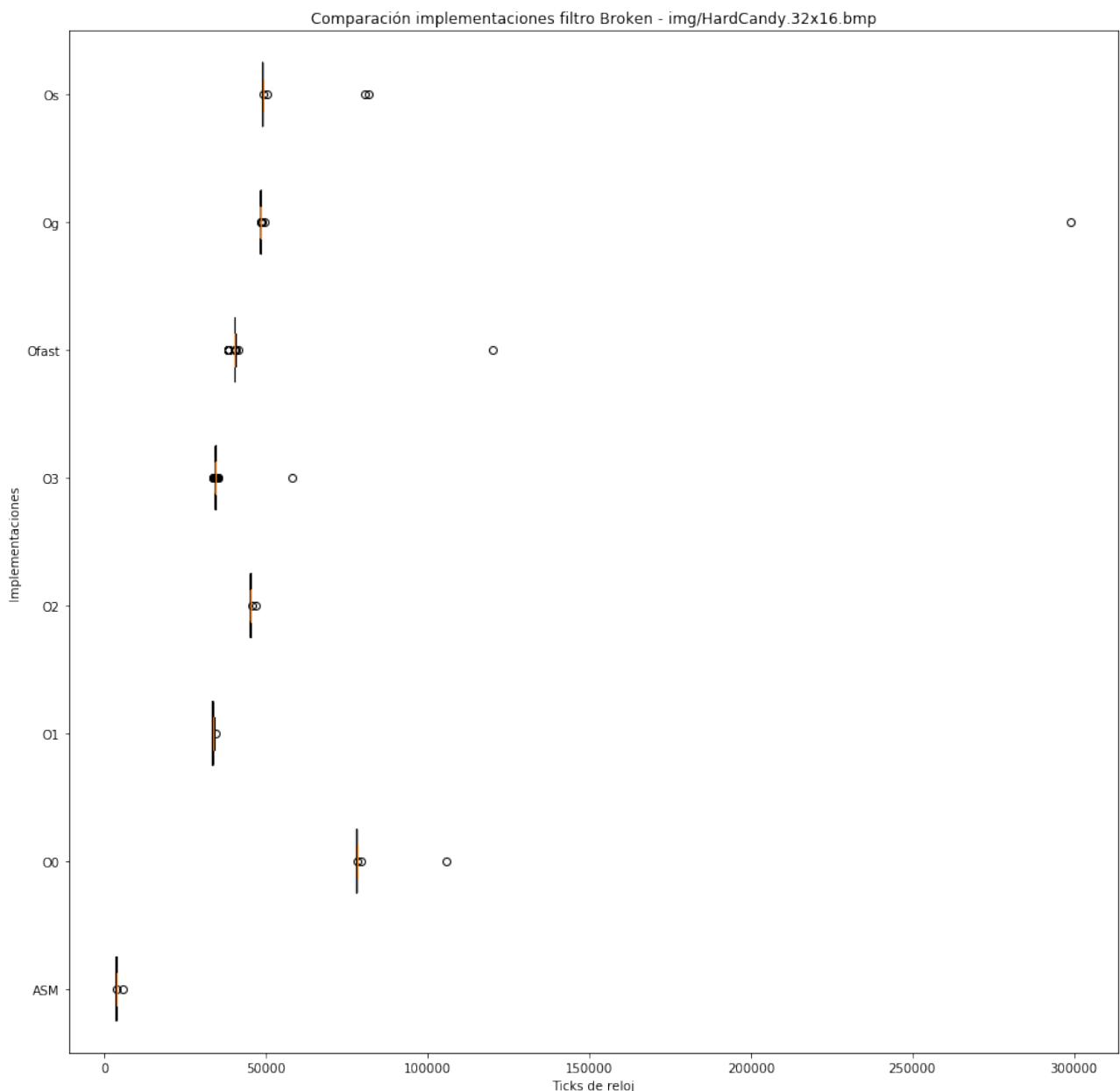


Figura 15: Broken ASM vs C en 32x16

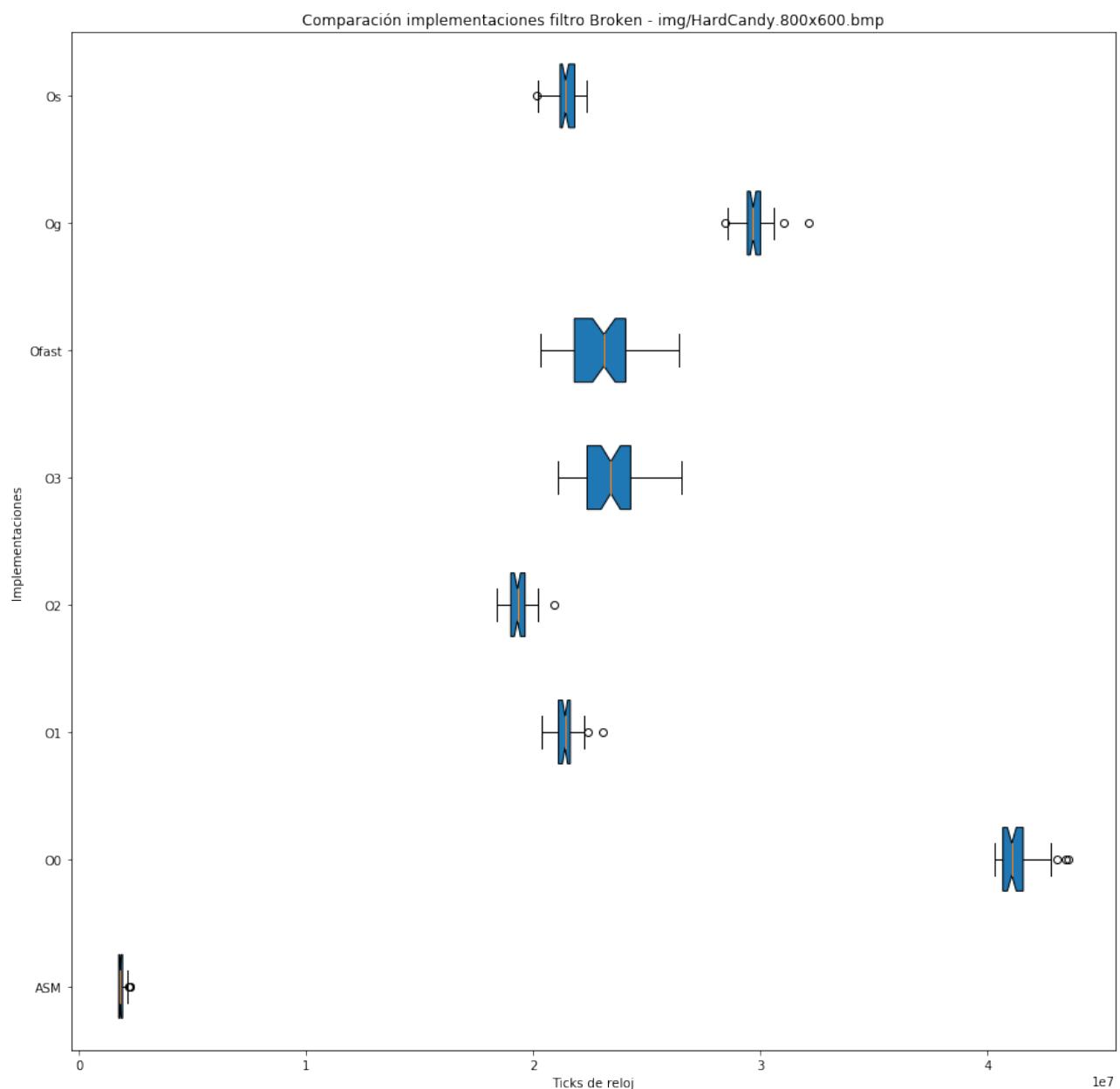


Figura 16: Broken ASM vs C en 800x600

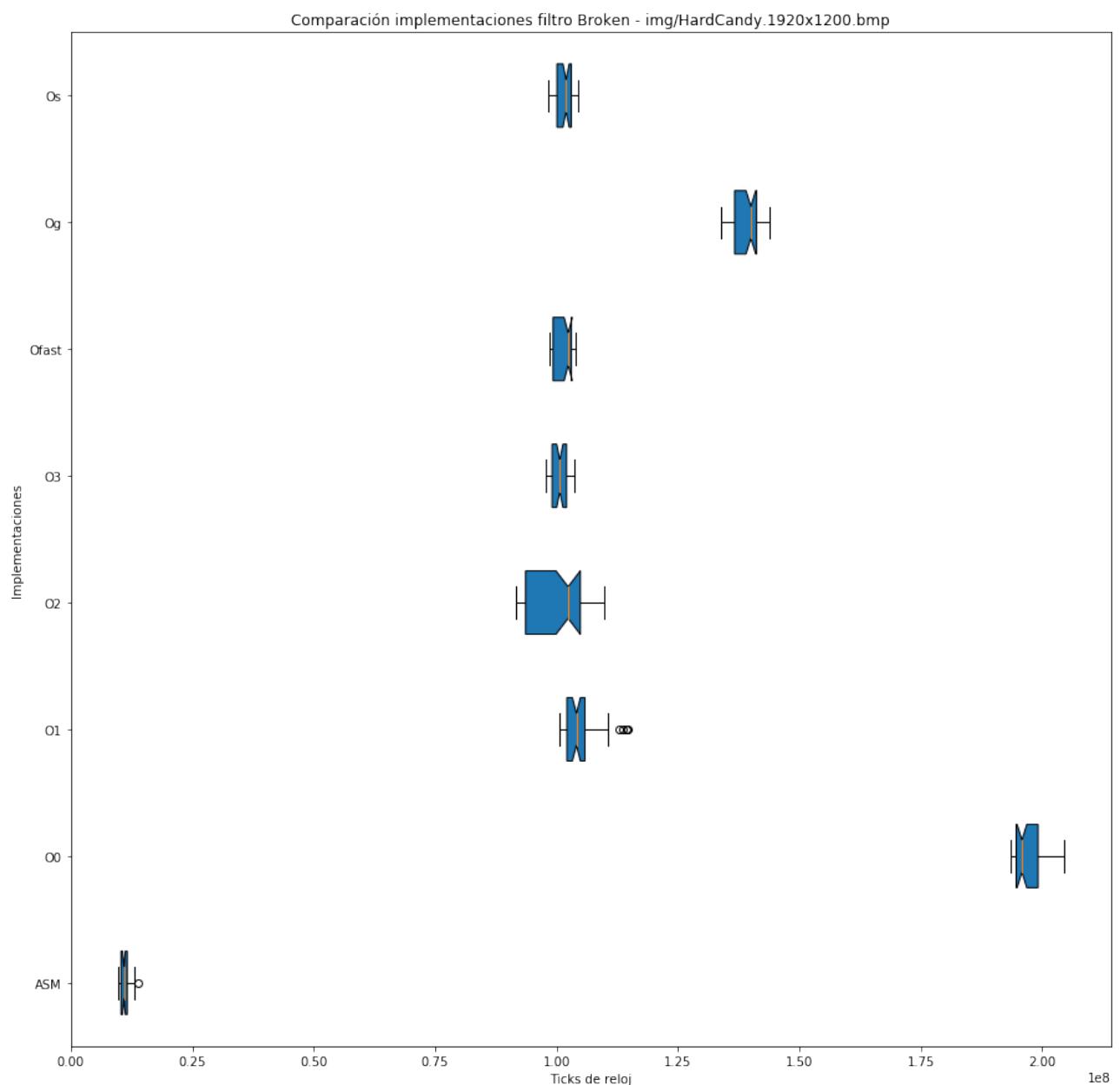


Figura 17: Broken ASM vs C en 1920x1200

## B. Imágenes adicionales para el experimento sobre Gamma

### B.1. Comparaciones Filtro Gamma\_lookup\_table ASM vs C

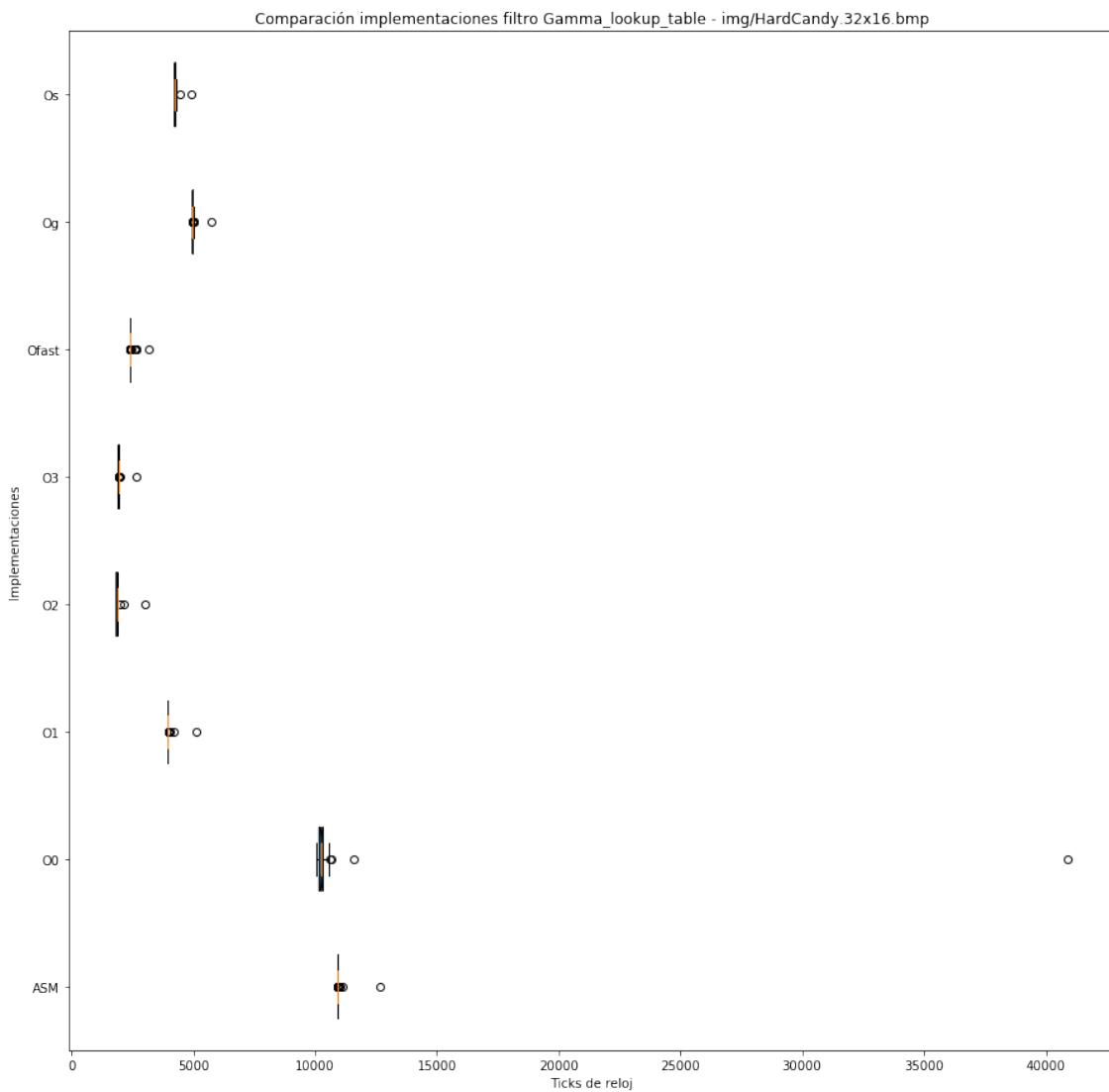


Figura 18: Gamma\_lookup\_table ASM vs C en 32x16

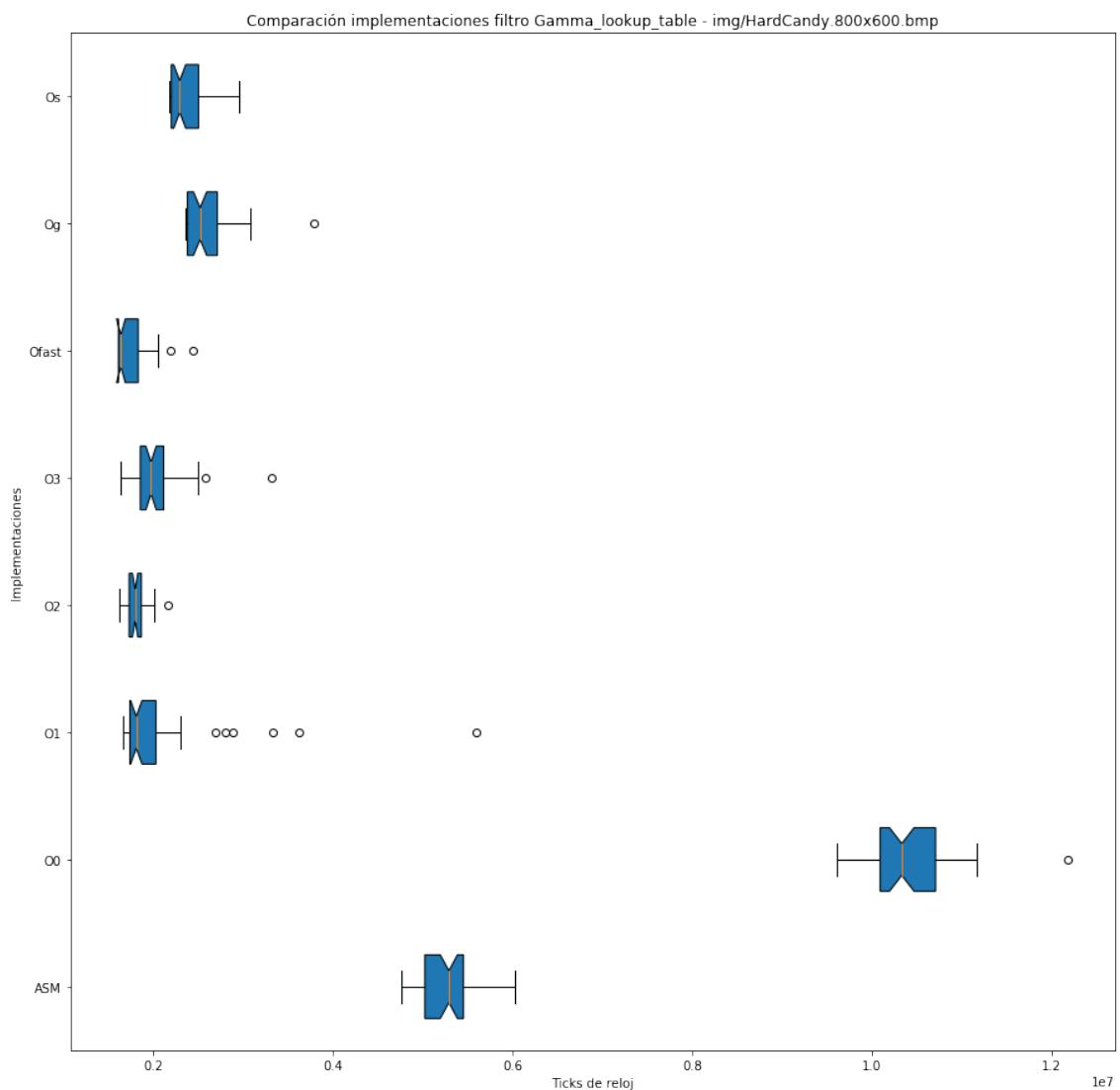


Figura 19: Gamma\_lookup\_table ASM vs C en 800x600

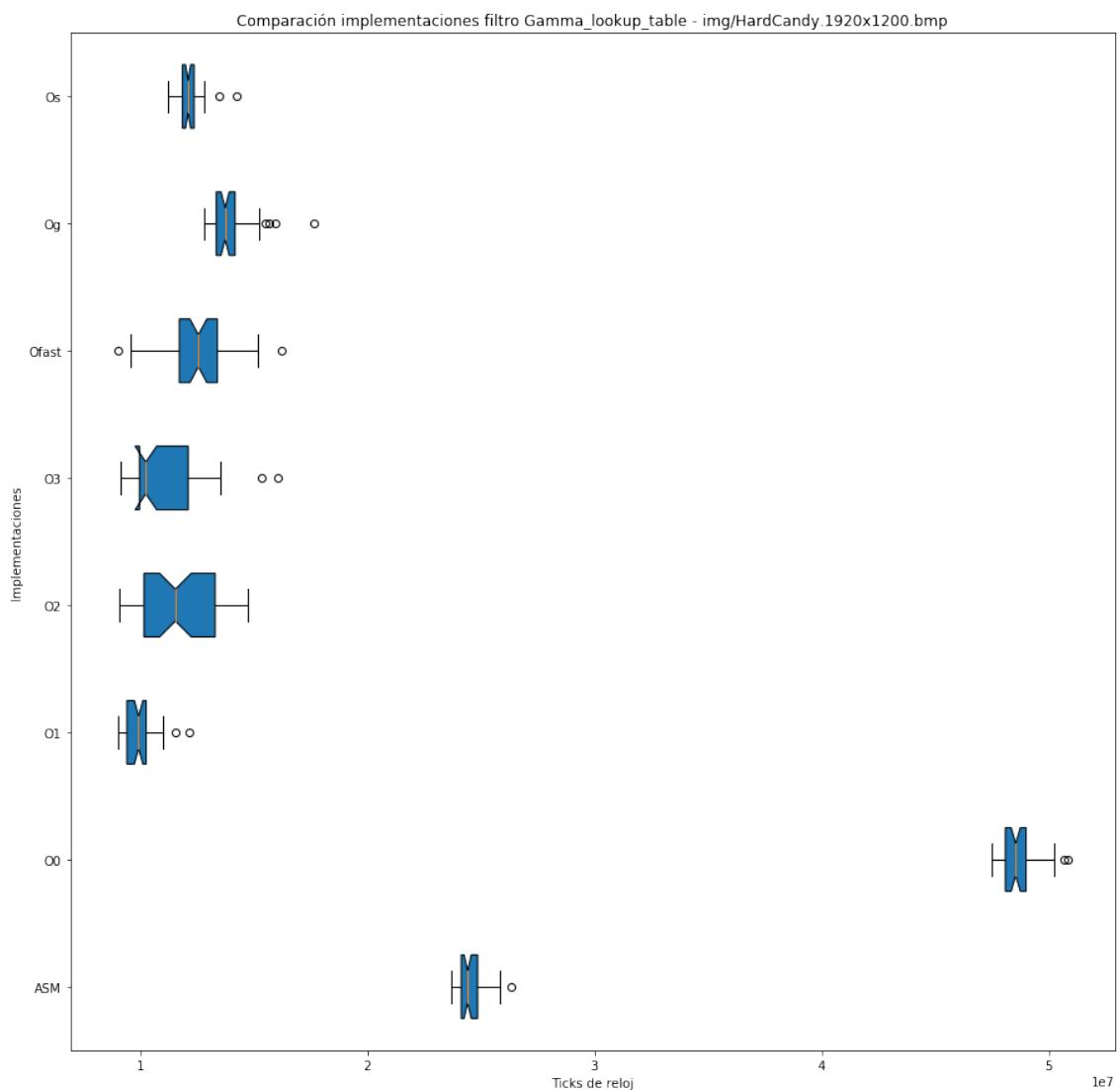


Figura 20: Gamma\_lookup\_table ASM vs C en 1920x1200

## B.2. Comparaciones de filtros Gamma

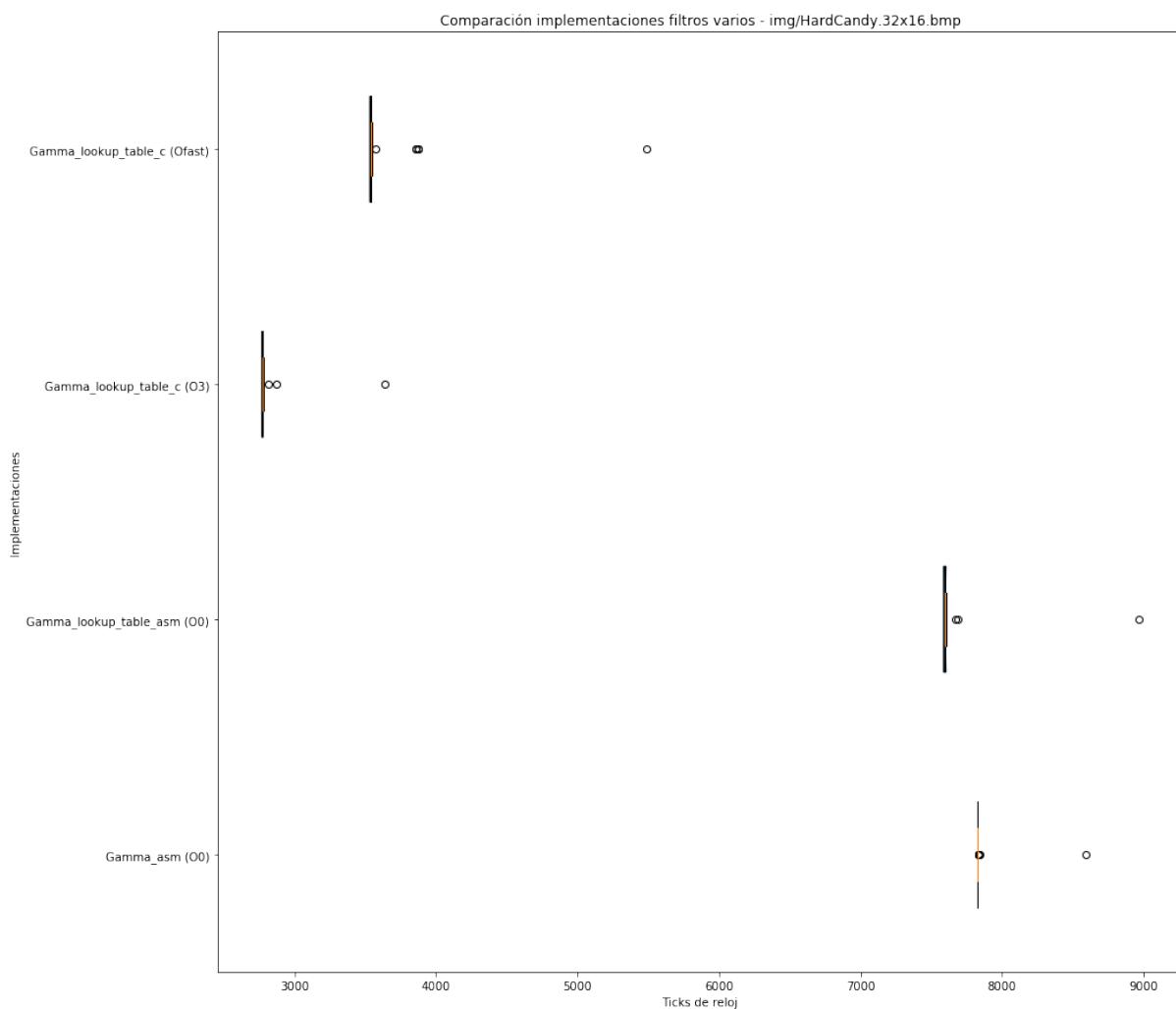


Figura 21: Comparación de Gammas en 32x16

## Sección B.2 Comparaciones de filtros Gamma

---

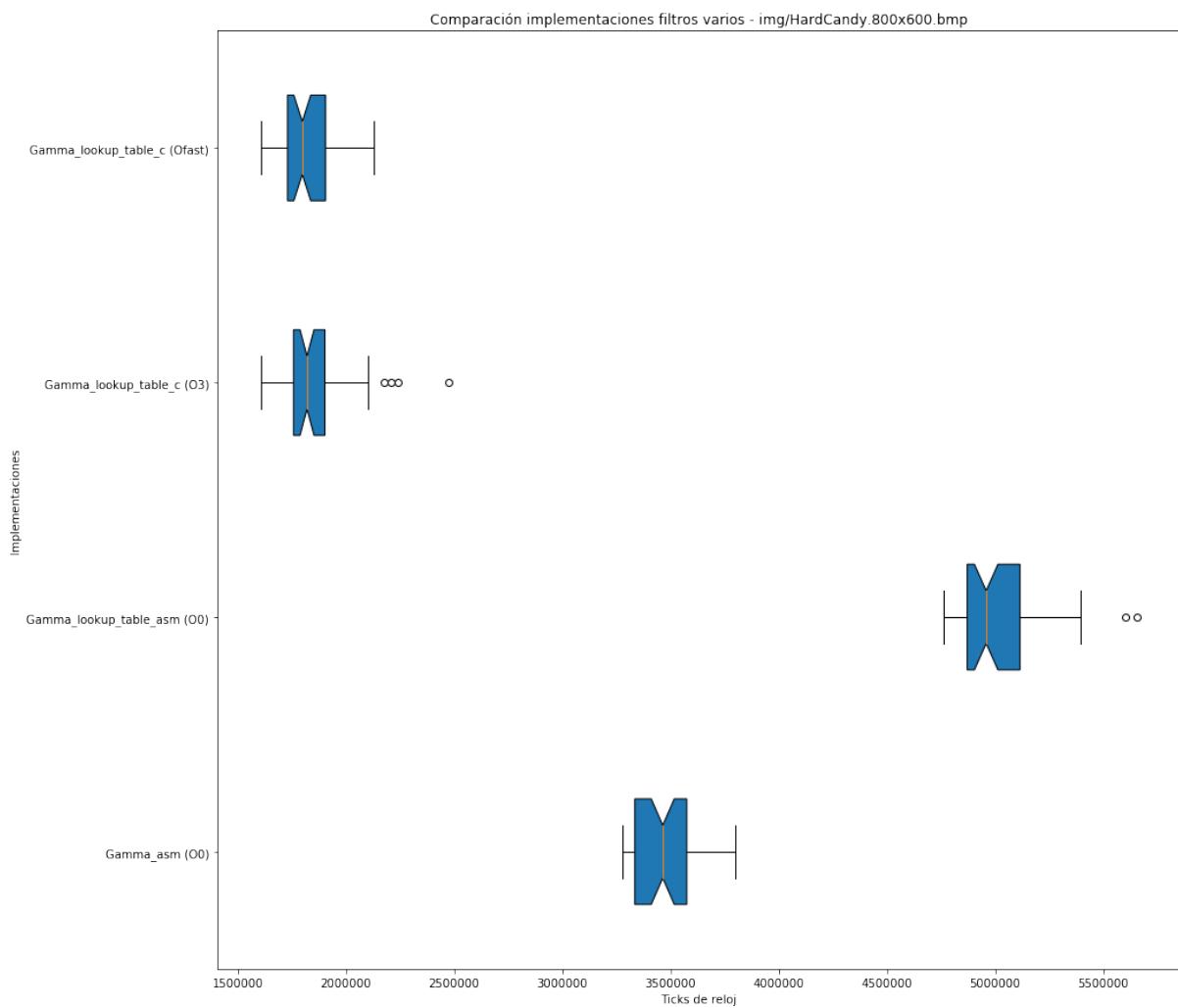


Figura 22: Comparación de Gammas en 800x600

## Sección B.2 Comparaciones de filtros Gamma

---

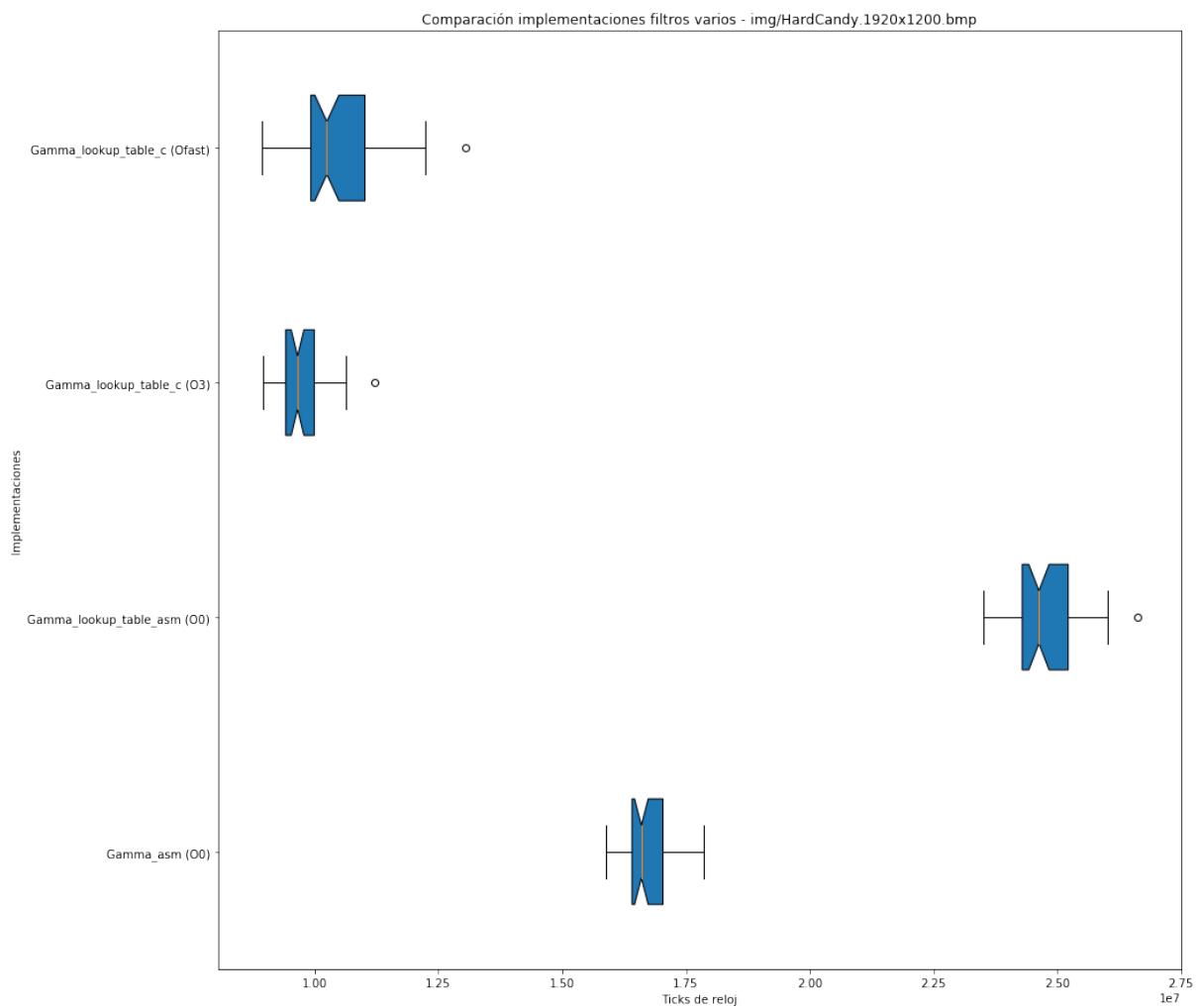


Figura 23: Comparación de Gammas en 1920x1200

## C. Imágenes adicionales para el experimento sobre Broken

### C.1. Comparaciones Broken\_alternativo Assembler vs C

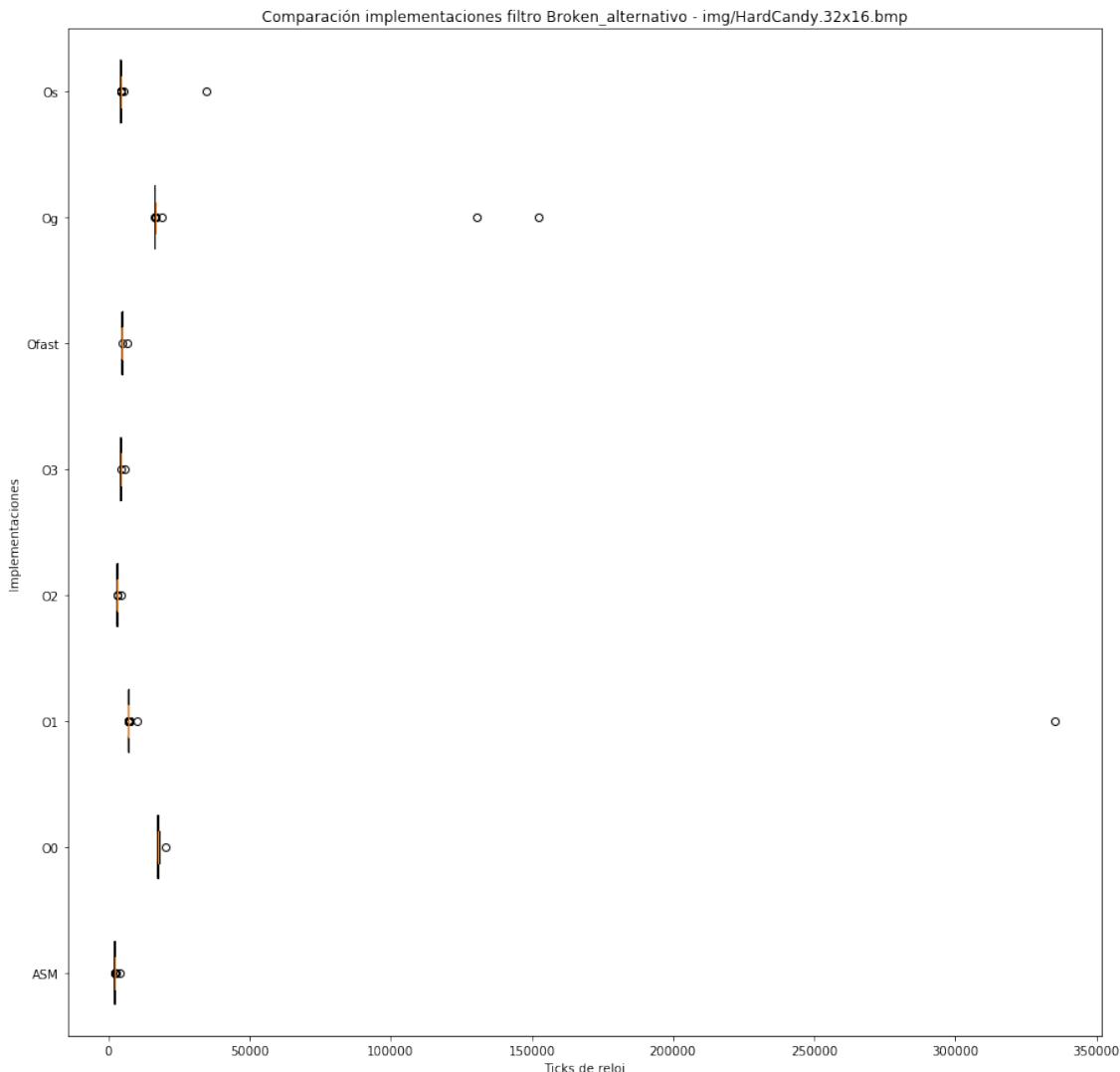


Figura 24: Broken alternativo ASM vs C en 32x16

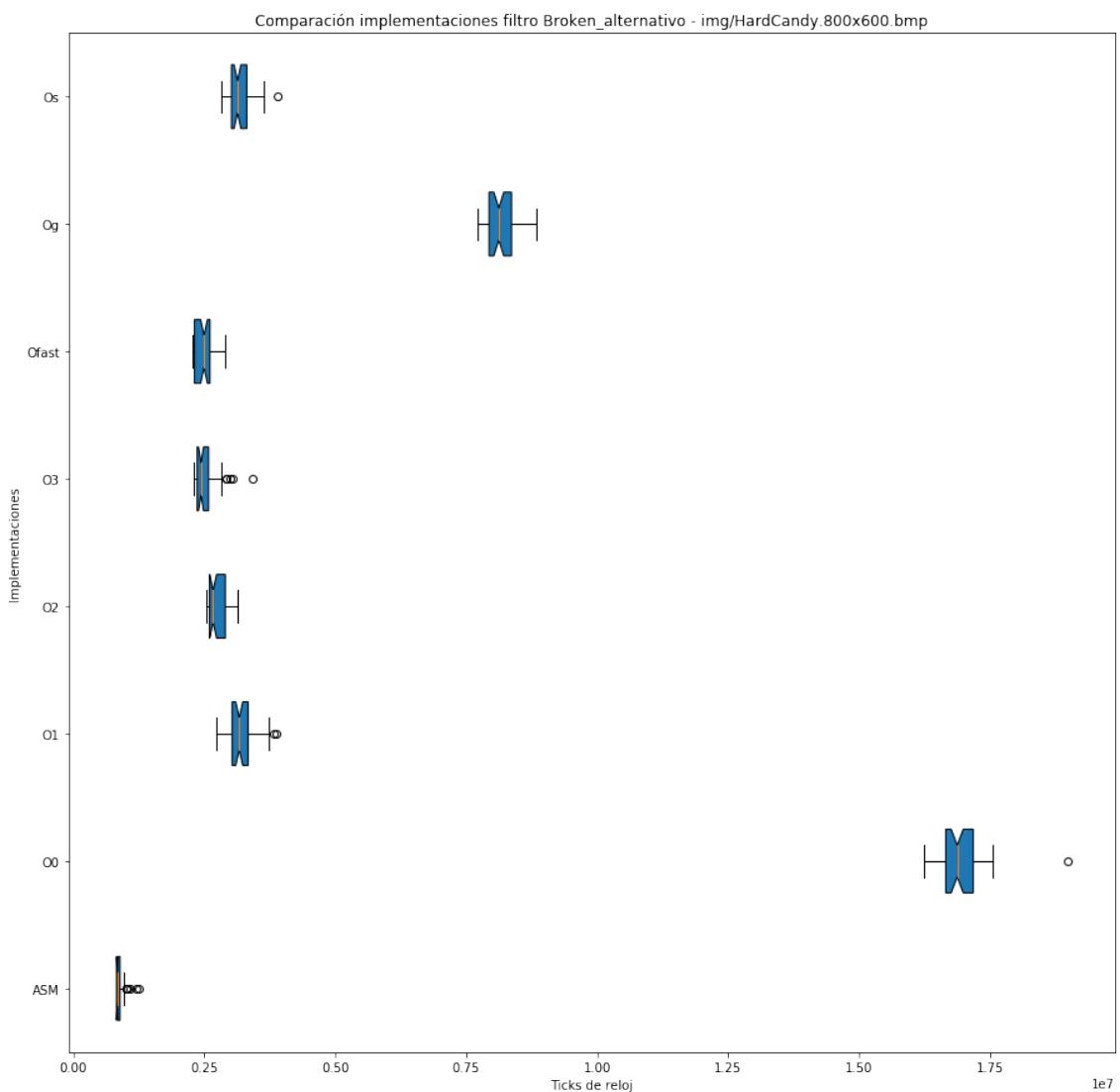


Figura 25: Broken alternativo ASM vs C en 800x600

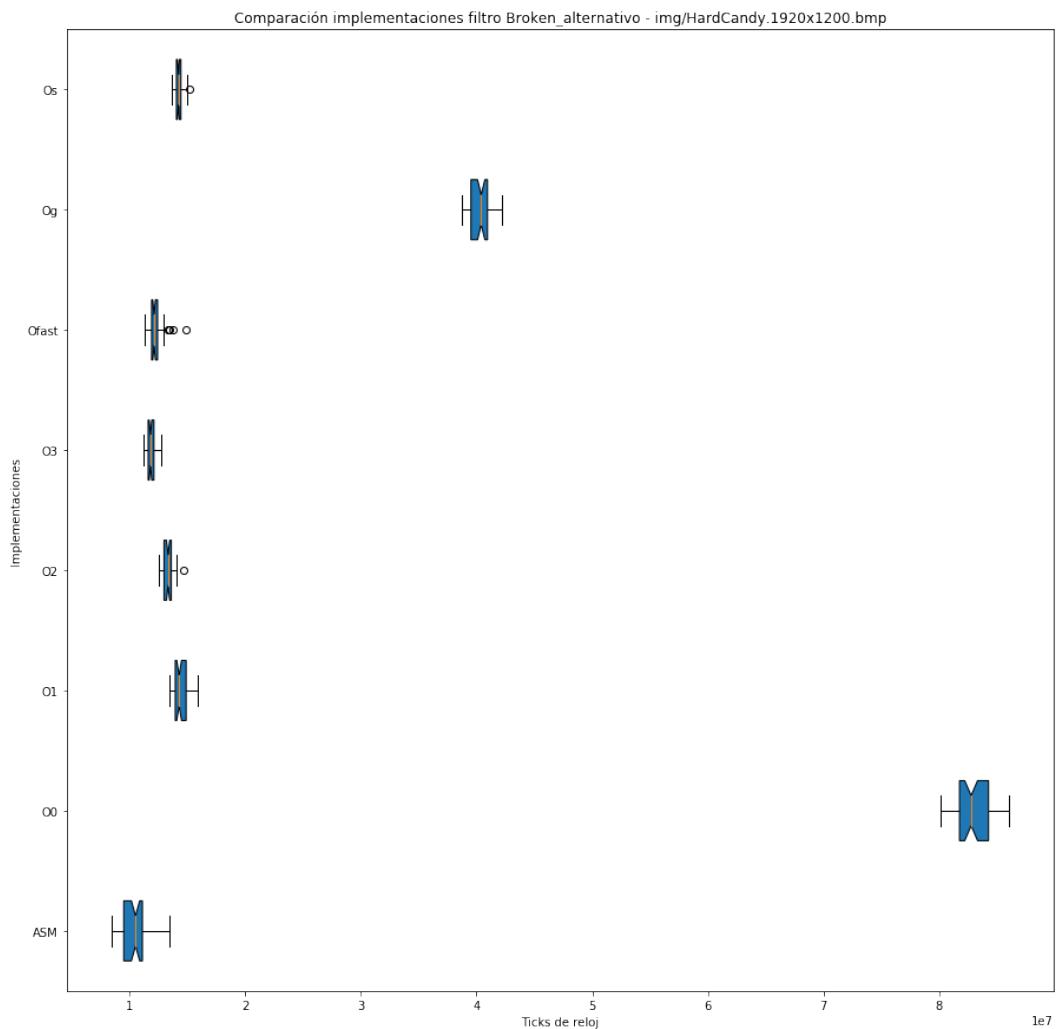


Figura 26: Filtro Broken\_alternativo en 1920x1200

## C.2. Comparaciones de Broken\_real\_offset en Assembler vs C

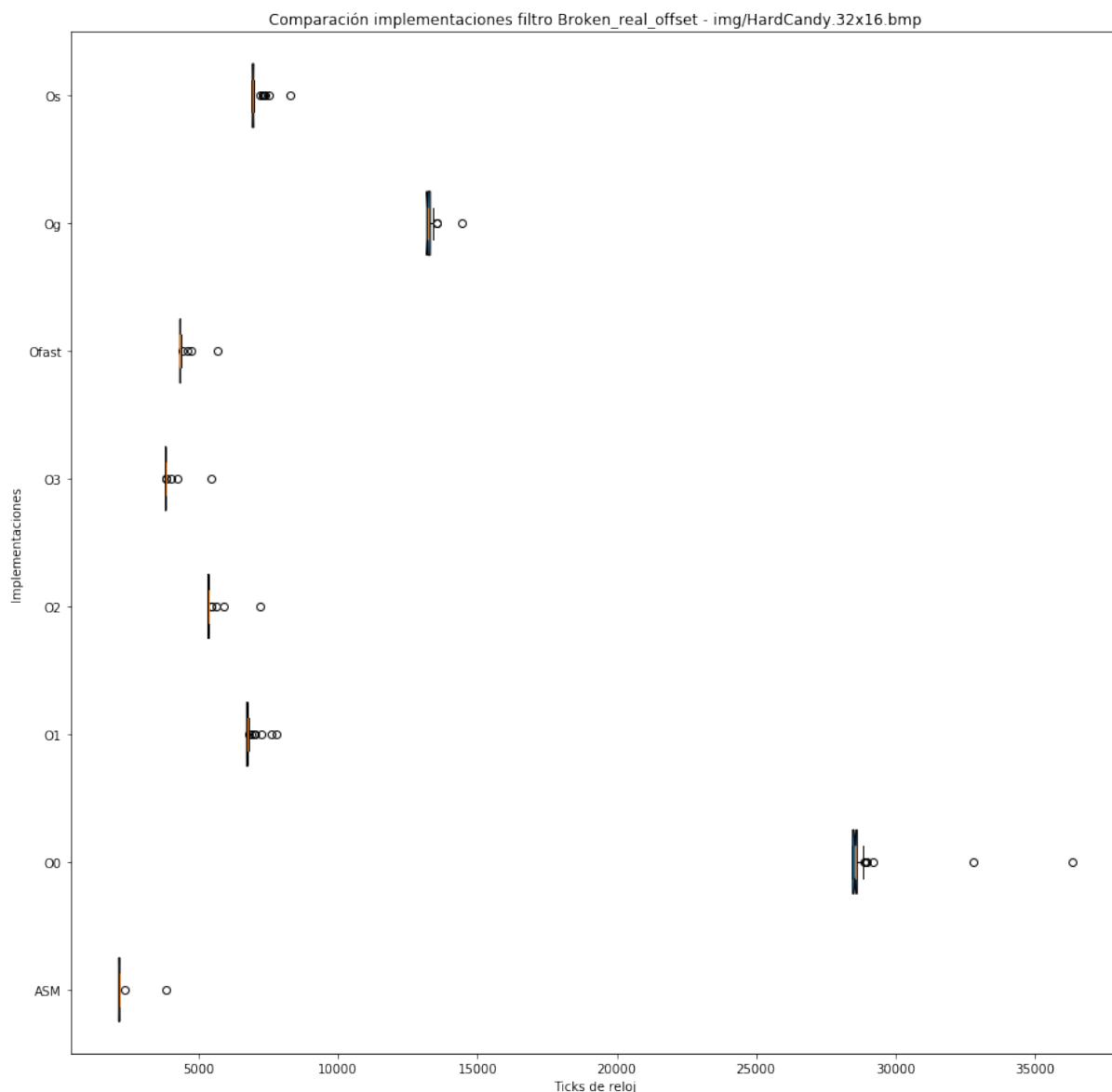


Figura 27: Broken real\_offset ASM vs C en 32x16

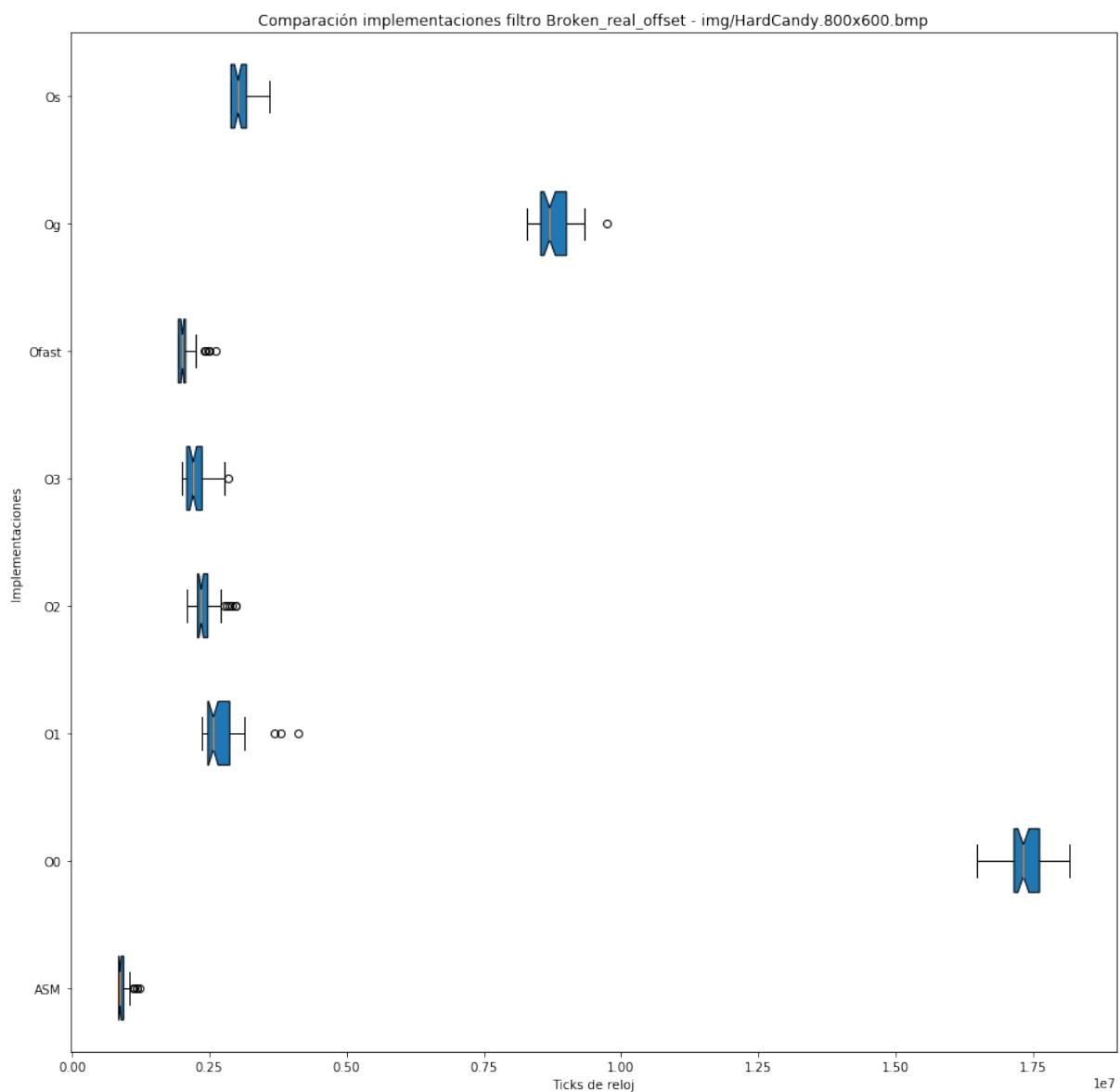


Figura 28: Broken real\_offset ASM vs C en 800x600

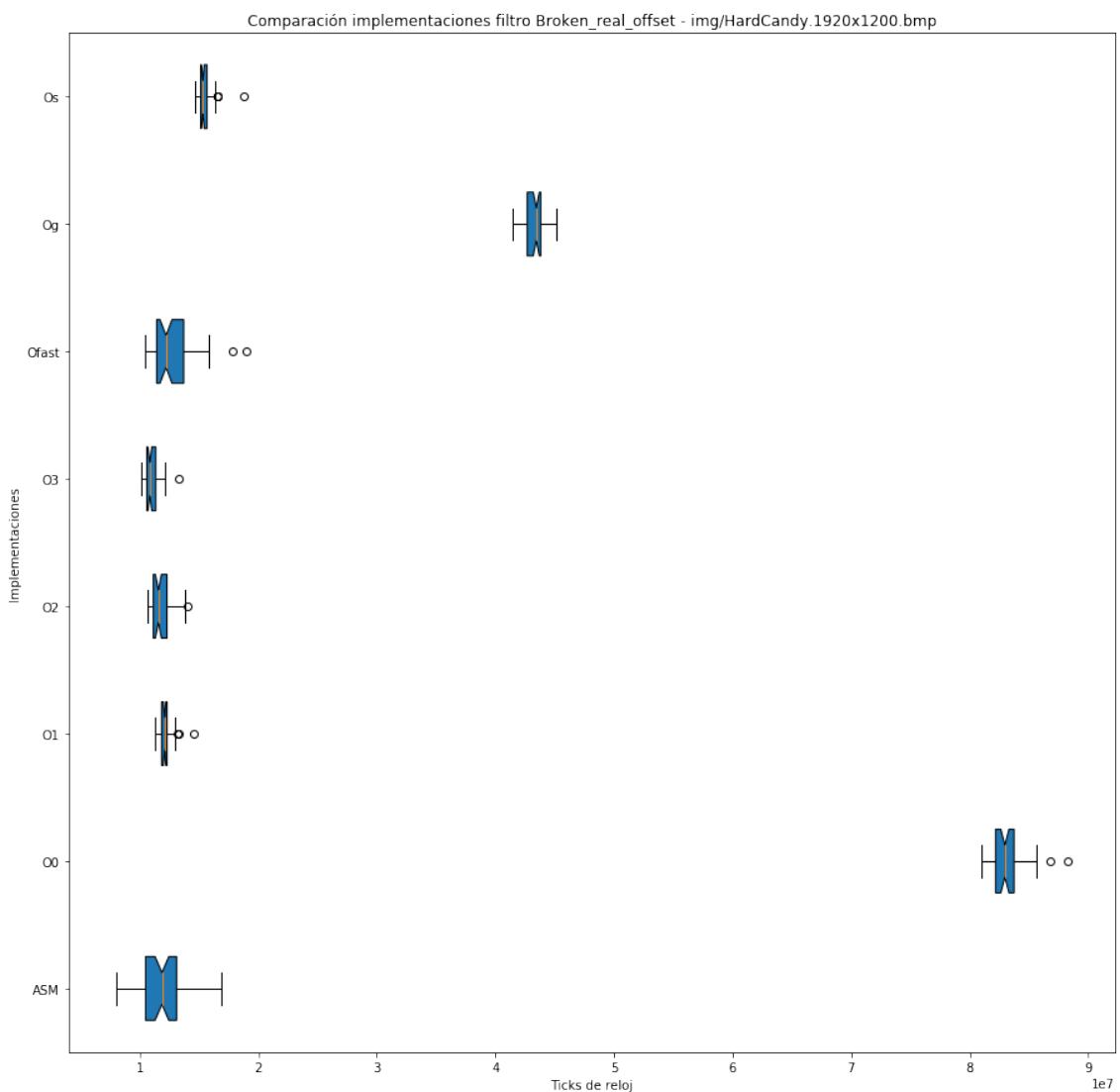


Figura 29: Filtro Broken\_real\_offset en 1920x1200

### C.3. Comparaciones de los tres filtros Broken

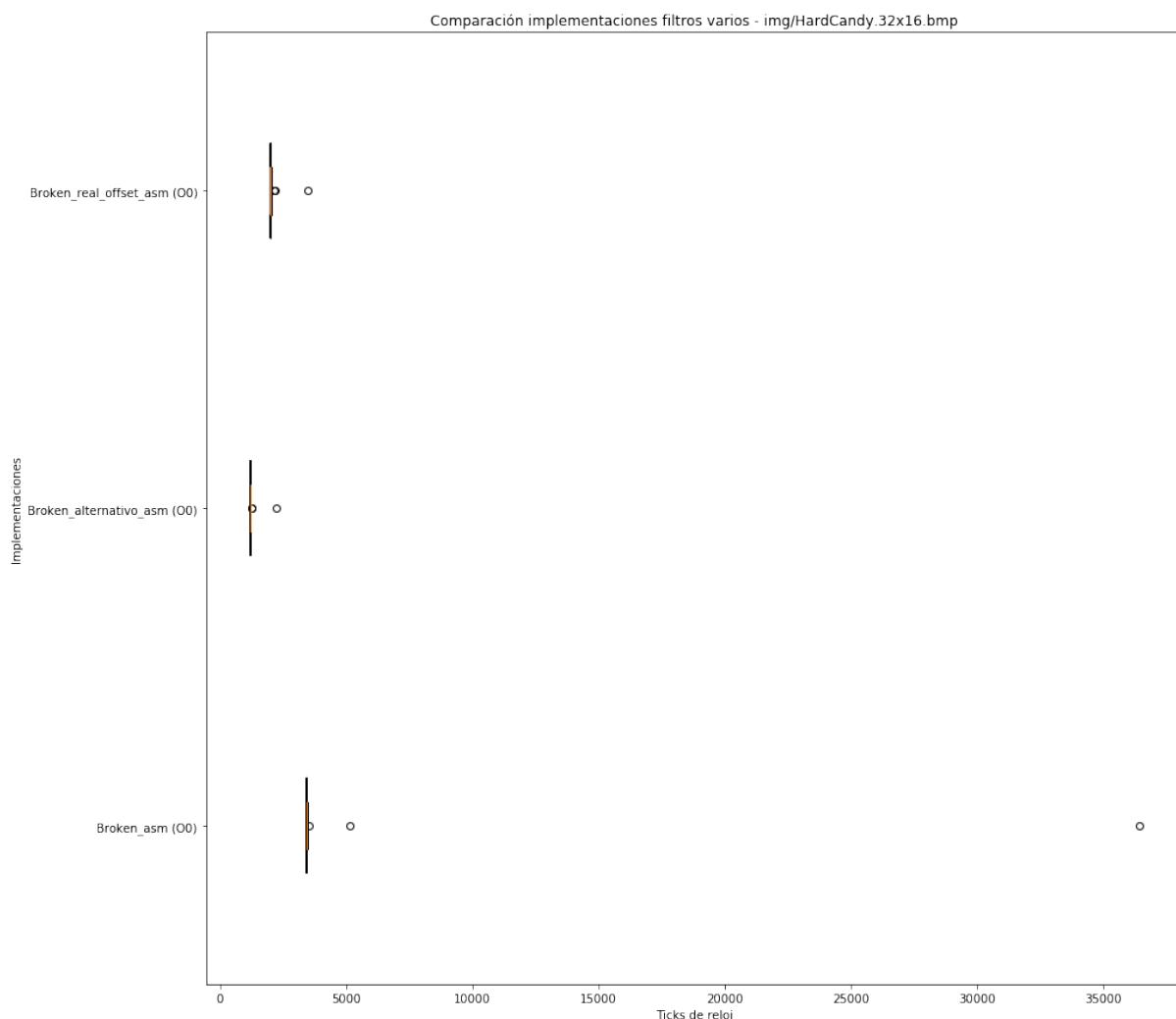


Figura 30: Comparaciones tres filtros Broken en ASM en 32x16

### Sección C.3 Comparaciones de los tres filtros Broken

---

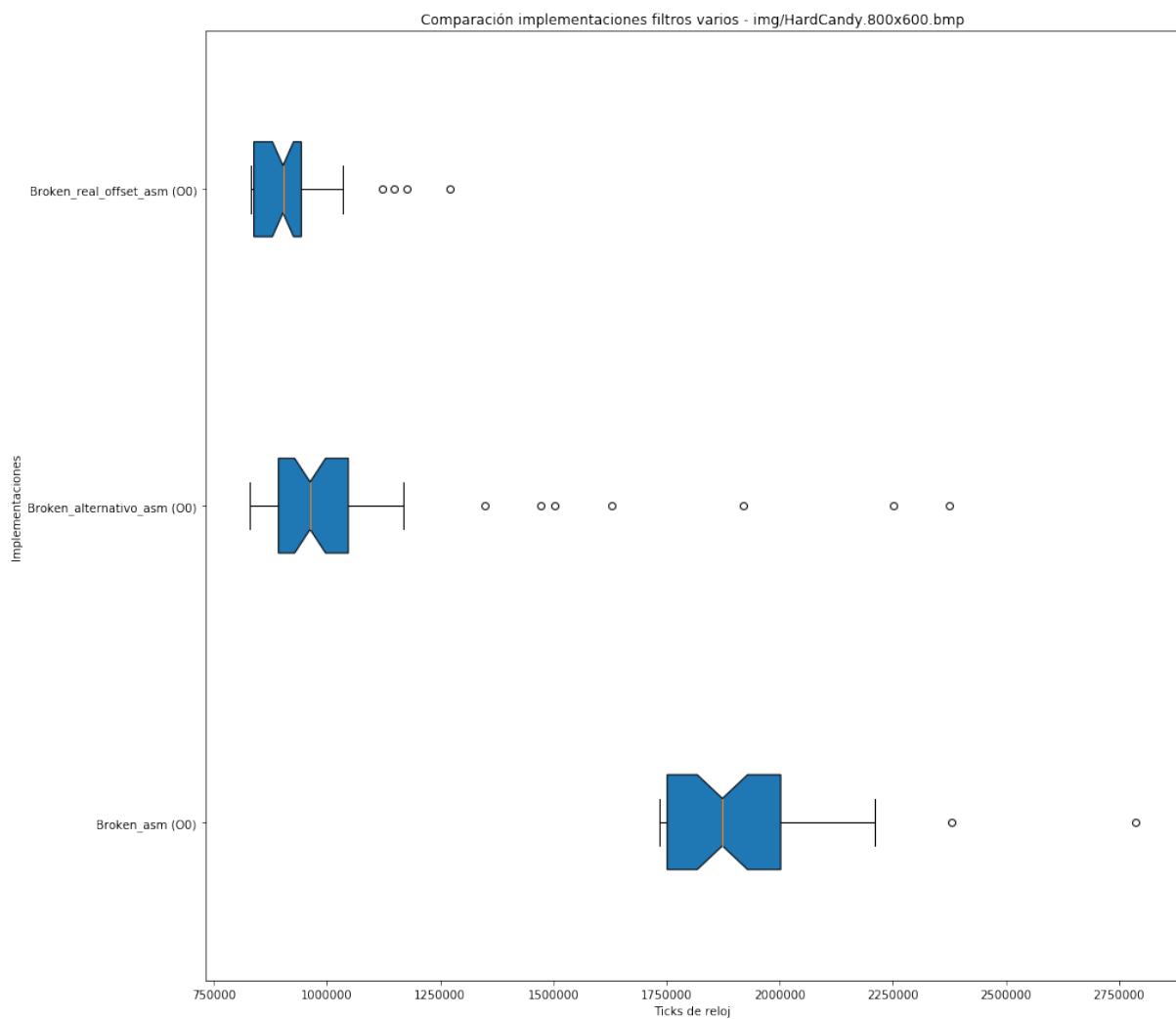


Figura 31: Comparaciones tres filtros Broken en ASM en 800x600

### Sección C.3 Comparaciones de los tres filtros Broken

---

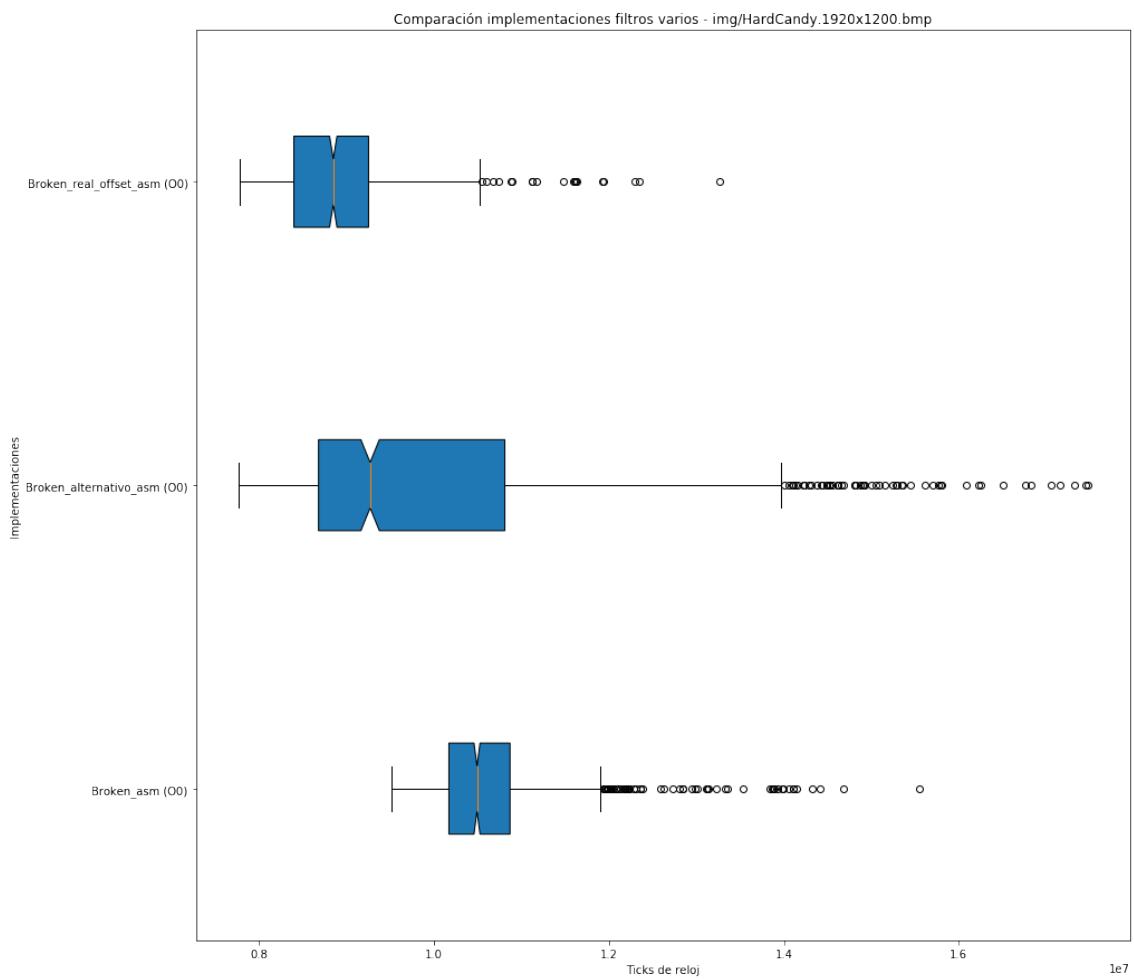


Figura 32: Comparaciones tres filtros Broken 1920x1200



Figura 33: Filtro Broken Original



Figura 34: Filtro Broken\_alternativo



Figura 35: Filtro Broken\_real\_offset.