



## DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

# Trabajo Práctico II

SIMD - Filtros

Organización del Computador II  
Primer cuatrimestre de 2021

Integrante	LU	Correo electrónico
Hayon, Gabriel David	701/17	gabrielhayonort@gmail.com
Lopes Perera, Pablo	007/18	plopesperera99@gmail.com
Naftaly, Joaquin	816/17	joaquin.naftaly@gmail.com



**Facultad de Ciencias Exactas y Naturales**  
Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)  
Intendente Güiraldes 2610 - C1428EGA  
Ciudad Autónoma de Buenos Aires - Rep. Argentina  
Tel/Fax: (++54 +11) 4576-3300  
<https://exactas.uba.ar>

## Resumen

En el presente trabajo se desarrollaron implementaciones y experimentos sobre tres filtros de imágenes: *Max* que realiza un difuminado en la imagen; *Broken* que “rompe” la imagen, y *Gamma* que le aumenta el valor de cada color. A lo largo del informe veremos las diferencias de performance entre implementaciones de los mismos en el lenguaje **C** y **ASM** haciendo uso de operaciones **SIMD** con las instrucciones de la extensión **SSE2**, para la arquitectura **x86 64**. El paralelismo que se logra con las instrucciones **SIMD** no se puede alcanzar incluso con optimizaciones al código en C. Se puede ver claramente la diferencia de los mismos en términos de eficiencia temporal, **ASM** performance mucho mas que **C** en todos los casos.

# Índice

<b>1. Introducción</b>	<b>2</b>
<b>2. Filtros</b>	<b>2</b>
2.1. Max . . . . .	2
2.1.1. Implementación . . . . .	2
2.2. Broken . . . . .	5
2.2.1. Implementación . . . . .	6
2.3. Gamma . . . . .	8
2.3.1. Implementación . . . . .	8
<b>3. Comparación</b>	<b>10</b>
<b>4. Experimentación</b>	<b>12</b>
4.1. Matriz vs Vector . . . . .	12
4.2. Limpieza de Cache y recorrido vertical en Broken . . . . .	12
<b>5. Conclusión</b>	<b>13</b>

# 1. Introducción

El objetivo de este trabajo práctico es implementar y analizar tres filtros de imágenes en ASM x86 64. Estos filtros son Max, Broken y Gamma.

**Definición de píxel** Vamos a tomar como píxel en el informe como un espacio en memoria de 4 bytes, donde toma la representación de la Figura 1

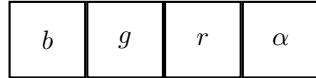


Figura 1: Pixel visualizado en memoria, cada componente es de 1 byte.

El rango de valores que puede tomar cada componente del píxel se encuentra entre  $0, \dots, 255$ .

## 2. Filtros

### 2.1. Max

Este filtro recorre la imagen de a  $4 \times 4$  píxeles, escribiendo en cada iteración una submatriz de  $2 \times 2$  píxeles centrada en la matriz que recorre. El primer par de ciclos anidados en el código calcula el valor a escribir, obteniendo un píxel, tal que la sumatoria de sus componentes sea máxima. Si existe más de un píxel que cumpla la condición, se quedara con el primero según el orden en que recorre la submatriz. Luego, el segundo par de ciclos anidados se encarga de escribir el píxel máximo en los cuatro píxeles de la matriz de  $2 \times 2$ . A continuación se presenta el pseudocódigo de la operación del filtro :

---

```
1: for i ∈ (0, ..., height - 3) as i:=i+2 do
2:   for j ∈ (0, ..., width - 3) as j:=j+2 do
3:     max := 0

4:     for ii ∈ (i, ..., i+3) do
5:       for jj ∈ (j, ..., j+3) do
6:         newMax := srcMatrix[ii][jj].r + srcMatrix[ii][jj].g + srcMatrix[ii][jj].b
7:         if max < newMax then
8:           max := newMax
9:           r = srcMatrix[ii][jj].r
10:          g = srcMatrix[ii][jj].g
11:          b = srcMatrix[ii][jj].b
12:        for ii ∈ (i+1, ..., i+2) do
13:          for jj ∈ (j+1, ..., j+2) do
14:            dstMatrix[ii][jj].r = r
15:            dstMatrix[ii][jj].g = g
16:            dstMatrix[ii][jj].b = b
```

---

Las operaciones se realizan dejando un marco de 1 píxeles alrededor de toda la imagen. Estos píxeles deben ser completados con el color blanco 255 sobre la imagen destino.

#### 2.1.1. Implementación

El filtro se implementó de manera tal que se recorre por filas y luego por columnas, al igual que su par en C.

Para este filtro nos vamos a imaginar que en cada iteración tenemos un cuadrante de 16 pixels de esta forma:

$p_0$	$p_1$	$p_2$	$p_3$
$p_4$	$p_5$	$p_6$	$p_7$
$p_8$	$p_9$	$p_{10}$	$p_{11}$
$p_{12}$	$p_{13}$	$p_{14}$	$p_{15}$

Figura 2: Cuadrante donde los pixels pintados en rojo serán los editados por el máximo valor de los 16 pixels.

En cada iteración se toman de memoria 4 rows de píxeles cada una de 4 píxeles, teniendo así 16 píxeles en los 4 registros `xmmx`, quedando así:

xmm0:	$\alpha_3$	$r_3$	$g_3$	$b_3$	$\alpha_2$	$r_2$	$g_2$	$b_2$	$\alpha_1$	$r_1$	$g_1$	$b_1$	$\alpha_0$	$r_0$	$g_0$	$b_0$
xmm1:	$\alpha_7$	$r_7$	$g_7$	$b_7$	$\alpha_6$	$r_6$	$g_6$	$b_6$	$\alpha_5$	$r_5$	$g_5$	$b_5$	$\alpha_4$	$r_4$	$g_4$	$b_4$
xmm2:	$\alpha_{11}$	$r_{11}$	$g_{11}$	$b_{11}$	$\alpha_{10}$	$r_{10}$	$g_{10}$	$b_{10}$	$\alpha_9$	$r_9$	$g_9$	$b_9$	$\alpha_8$	$r_8$	$g_8$	$b_8$
xmm3:	$\alpha_{15}$	$r_{15}$	$g_{15}$	$b_{15}$	$\alpha_{14}$	$r_{14}$	$g_{14}$	$b_{14}$	$\alpha_{13}$	$r_{13}$	$g_{13}$	$b_{13}$	$\alpha_{12}$	$r_{12}$	$g_{12}$	$b_{12}$

**Suma ordenada por aparición:** Luego queremos de alguna manera calcular el máximo entre los 16 pixels, manteniendo una relación de orden, es decir que dado dos pixels que su suma de colores sean iguales, nos vamos a quedar con el pixel que haya aparecido primero en la figura 2.

Para ello entonces transponemos la matriz de pixels que tenemos entre los registros `xmmx` y entonces obtenemos:

xmm0:	$p_{12}$	$p_8$	$p_4$	$p_0$
xmm1:	$p_{13}$	$p_9$	$p_5$	$p_1$
xmm2:	$p_{14}$	$p_{10}$	$p_6$	$p_2$
xmm3:	$p_{15}$	$p_{11}$	$p_7$	$p_3$

Figura 3: Registros `xmmx` representando la matriz transpuesta de los pixels a evaluar.

Una vez reordenados los pixels, sumamos sus componentes de colores por pixel. Notemos que la suma a calcular puede exceder el tamaño de representación de un color, en particular con 3 bytes alcanza para representar el resultado de la suma. Teniendo en cuenta esto, entonces, por cada row de pixels, extraemos los colores de cada pixel a 3 registros `xmmx` mediante máscaras y los extendemos a un tamaño de `word` (2 bytes). Ahora haciendo una suma vertical entre los tres registros, obtenemos un registro `xmm` con la suma de  $r$ ,  $g$  y  $b$  de cada píxel.

El resultado de cada row de pixels lo guardamos en los mismos registros `xmm` que estaban, y copiamos los valores de los pixels originales en los registros `xmm12` a `xmm15`.

**Comparaciones:** Ahora tenemos dos matrices, una con los valores de las sumas de los colores de cada pixel, y otra con los valores de los pixels, ambas ordenadas como se puede ver en la figura 3.

Entonces comparamos a word si  $\text{xmm0} < \text{xmm1}$ , es decir que comparamos si las sumas de los colores de un pixel que aparece antes es menor que la suma de otro que aparece después. Obteniendo una máscara para cada comparación, con 1's en los words que registro de la derecha sea mayor o 0's si el de la izquierda era menor o igual.

xmm0:	( $r + g + b$ ) $p_{12}$	( $r + g + b$ ) $p_8$	( $r + g + b$ ) $p_4$	( $r + g + b$ ) $p_0$
	<	<	<	<
xmm1:	( $r + g + b$ ) $p_{13}$	( $r + g + b$ ) $p_9$	( $r + g + b$ ) $p_5$	( $r + g + b$ ) $p_1$
	=	=	=	=
xmm6:	1/0	1/0	1/0	1/0

Entonces este resultado lo usamos para saber con que valor nos queremos quedar, esto para los registros que contienen las sumas y para los registros que contienen los valores originales de los pixels; por lo que hacemos un **and** de la máscara con **xmm1** y un **nand** con **xmm0**,

xmm1:	( $r + g + b$ ) $p_{13}$	( $r + g + b$ ) $p_9$	( $r + g + b$ ) $p_5$	( $r + g + b$ ) $p_1$
	and	and	and	and
xmm6:	1/0	1/0	1/0	1/0
	=	=	=	=
xmm6:	( $r + g + b$ ) $p_{13}/0$	( $r + g + b$ ) $p_9/0$	( $r + g + b$ ) $p_5/0$	( $r + g + b$ ) $p_1/0$
xmm0:	( $r + g + b$ ) $p_{12}$	( $r + g + b$ ) $p_8$	( $r + g + b$ ) $p_4$	( $r + g + b$ ) $p_0$
	nand	nand	nand	nand
xmm6:	1/0	1/0	1/0	1/0
	=	=	=	=
xmm7:	0/( $r + g + b$ ) $p_{12}$	0/( $r + g + b$ ) $p_8$	0( $r + g + b$ ) $p_4$	0/( $r + g + b$ ) $p_0$

luego sumamos los registros **xmm6** y **xmm7**, obteniendo en un registro los valores de sumas máximos ya que son casos disjuntos, y nunca vamos a sumar una posición donde había dos valores. Notar que de esta manera en el registro nos queda el valor de **xmm0** en caso de que los valores sean iguales.

xmm6:	$(r + g + b)p_{13}/0$	$(r + g + b)p_9/0$	$(r + g + b)p_5/0$	$(r + g + b)p_1/0$
	+	+	+	+
xmm7:	$0/(r + g + b)p_{12}$	$0/(r + g + b)p_8$	$0(r + g + b)p_4$	$0/(r + g + b)p_0$
	=	=	=	=
xmm0:	$\max((r + g + b)p_{13}, (r + g + b)p_{12})$	$\max((r + g + b)p_{10}, (r + g + b)p_8)$		
	$\max((r + g + b)p_5, (r + g + b)p_4)$	$\max((r + g + b)p_1, (r + g + b)p_0)$		

Figura 4: El registro **xmm0** se muestra dividido en parte alta y parte baja por el tamaño de hoja

por otro lado, hacemos lo mismo para los registros **xmm12** y **xmm13** que contienen los valores de los pixels **xmm0** y **xmm1**, haciendo un **nand** para el caso de **xmm12** y un **and** para el caso de **xmm13**. Notar que para los **and** y **nand** con los registros que contienen los valores de los pixels, primero se debe extender a **dword** la máscara.

Todo este proceso se repite con el otro par de rows de pixels **xmm2** y **xmm3**, y para todo par de resultados que se generan a partir de las comparaciones (siempre teniendo a la izquierda del menor, los pixels que aparecen antes que los de la derecha), llegando así a un resultado en un registro donde tenemos el valor máximo de la suma y un registro con el pixel que representa ese valor máximo.

**Dibujar resultado:** Ahora con este resultado lo que hacemos es reemplazar los colores de los pixels en el cuadrante interno (ver imagen 2) y hardcodear el **alpha** en 0xFF.

Todo lo explicado arriba se ejecuta en dos loops recorriendo las columnas por cada fila, incrementando el contador de los mismos en 2 para la siguiente iteración.

**Dibujar bordes:** Por último, para dibujar los bordes se recorren los bordes de la imagen (sin SIMD) y se hardcodean los valores de todos los pixels en 0xFFFFFFFF

**Ejemplo:** A continuación se muestran dos imágenes, a la izquierda la original, y a la derecha la imagen con el filtro aplicado.



(a) Imagen original

(b) Filtro aplicado

## 2.2. Broken

La operatoria de este filtro consiste en romper la imagen, alterando el orden de los componentes de cada píxel. El filtro utiliza como entrada un arreglo de 40 números y realiza una suerte de shift de los componentes de los píxeles según sea indicado el arreglo. El arreglo solo contiene los offsets -16, -8, -4, 0, 4, 8, 16 y 32, limitando los casos a considerar para su implementación. A continuación se presenta el pseudocódigo de la operatoria del filtro:

---

```

1: 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 }

2: for i ∈ (0, . . . , height − 1) do
3:   for j ∈ (0, . . . , width − 1) do
4:     jr = (j + 8 * width + a[(i+10)%40]) %width
5:     jg = (j + 8 * width + a[(i+20)%40]) %width
6:     jb = (j + 8 * width + a[(i+30)%40]) %width
7:     dstMatrix[i][j].r = SATURAR( srcMatrix[i][jr].r )
8:     dstMatrix[i][j].g = SATURAR( srcMatrix[i][jg].g )
9:     dstMatrix[i][j].b = SATURAR( srcMatrix[i][jb].b )

```

---

### 2.2.1. Implementación

En esta sección se detallará el procedimiento realizado para obtener, calcular y guardar los píxeles de la imagen junto con las instrucciones SIMD utilizadas en cada paso.

El cálculo RGB del píxel  $P_{i,j}$  necesita 3 píxeles  $P_{i,jr}, P_{i,jg}, P_{i,jb}$  que se obtienen realizando las siguientes operaciones:

$$jr = (j + 8 * width + a[(i + 10) \% 40]) \% width \quad (1)$$

$$jg = (j + 8 * width + a[(i + 20) \% 40]) \% width \quad (2)$$

$$jb = (j + 8 * width + a[(i + 30) \% 40]) \% width \quad (3)$$

Notar que  $preindex_X = 8 * width + a[(i + X) \% 40]$ , con  $X \in [10, 20, 30]$  no depende de  $j$ . Por ende se puede calcular fuera del ciclo de columnas utilizando instrucciones normales. Además, como el  $width$  es múltiplo de 8, se puede afirmar que 4 píxeles consecutivos  $P_{i,j}, P_{i,j+1}, P_{i,j+2}, P_{i,j+3}$  se calculan con 3 pares de píxeles consecutivos. Es decir:

$$P_{i,j+k} \leftarrow (P_{i,jr+k}, P_{i,jg+k}, P_{i,jb+k}) \text{ con } 0 \leq k \leq 3. \quad (4)$$

Una vez calculados los  $preindex_X$  en la iteración de filas se itera por columnas de 4 en 4 ya que se procesan 4 píxeles en cada iteración. Dentro del ciclo de columnas se terminan de calcular los  $jr, jg, jb$  utilizando instrucciones normales. Con esos índices ya calculados se obtienen los 12 píxeles necesarios para calcular los  $P_{i,j}, P_{i,j+1}, P_{i,j+2}, P_{i,j+3}$ . Observación: Para mayor simplicidad en el informe se remueve el índice  $i$  ya que todos los píxeles de los que estemos hablando pertenecen a la misma fila. Las máscaras y los píxeles se obtienen usando la instrucción `movdqu`.

Primero en `xmm2` se levanta una máscara que tiene  $\alpha = 255$  y R,G,B en 0 en donde se van a ir guardando los resultados de cada componente. Posteriormente se levantan los 4 píxeles consecutivos desde el índice  $jb$  en `xmm0` y se levanta en `xmm1` una máscara con 1s en la componente *blue* y 0s en las demás.

xmm2:	FF	00	00	00												
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

xmm0:	$\alpha_{jb+3}$	$r_{jb+3}$	$g_{jb+3}$	$b_{jb+3}$	$\alpha_{jb+2}$	$r_{jb+2}$	$g_{jb+2}$	$b_{jb+2}$	$\alpha_{jb+1}$	$r_{jb+1}$	$g_{jb+1}$	$b_{jb+1}$	$\alpha_{jb}$	$r_{jb}$	$g_{jb}$	$b_{jb}$
-------	-----------------	------------	------------	------------	-----------------	------------	------------	------------	-----------------	------------	------------	------------	---------------	----------	----------	----------

xmm1:	00	00	00	FF												
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Para obtener la componente correspondiente al *blue* y 0s en las demás se aplica la máscara usando la instrucción `pand xmm0, xmm1`.

xmm0:	00	00	00	$b_{jb+3}$	00	00	00	$b_{jb+2}$	00	00	00	$b_{jb+1}$	00	00	00	$b_{jb}$
-------	----	----	----	------------	----	----	----	------------	----	----	----	------------	----	----	----	----------

Para mantener ese resultado en `xmm2`, que es donde se acumulan los resultados de cada componente, se realiza la instrucción `por xmm2, xmm0`.

xmm2:	FF	00	00	$b_{jb+3}$	FF	00	00	$b_{jb+2}$	FF	00	00	$b_{jb+1}$	FF	00	00	$b_{jb}$
-------	----	----	----	------------	----	----	----	------------	----	----	----	------------	----	----	----	----------

Luego se repite el mismo procedimiento para los 4 píxeles consecutivos de  $jk$  pero la mascara de `xmm1` se shiftea 8 a la izquierda dejando 1s en la componente *green* y 0s en las demás. Para eso se utiliza la instrucción `pslld xmm1,8`.

xmm1:	00	00	FF	00												
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

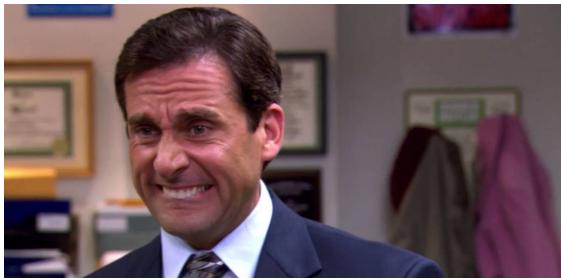
xmm2:	FF	00	$r_{jr+3}$	$b_{jb+3}$	FF	00	$r_{jr+2}$	$b_{jb+2}$	FF	00	$r_{jr+1}$	$b_{jb+1}$	FF	00	$r_{jr}$	$b_{jb}$
-------	----	----	------------	------------	----	----	------------	------------	----	----	------------	------------	----	----	----------	----------

Finalmente se vuelve a repetir el mismo procedimiento levantando los 4 píxeles consecutivos a partir del índice  $jr$  y shifteando otra vez la mascara de `xmm1` 8 a la izquierda. Dejando en `xmm2` el resultado final del píxel  $P_{i,j}$  para hacer `movdqu` y mover los 4 píxeles a la imagen destino en las posiciones a partir del índice  $i, j$ .

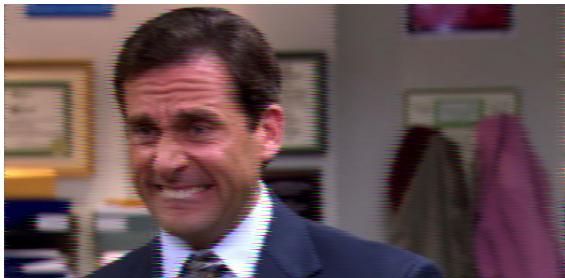
xmm1:	00	FF	00	00												
-------	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

xmm2:	FF	$r_{jr+3}$	$g_{jg+3}$	$b_{jb+3}$	FF	$r_{jr+2}$	$g_{jg+2}$	$b_{jb+2}$	FF	$r_{jr+1}$	$g_{jg+1}$	$b_{jb+1}$	FF	$r_{jr}$	$g_{jg}$	$b_{jb}$
-------	----	------------	------------	------------	----	------------	------------	------------	----	------------	------------	------------	----	----------	----------	----------

**Ejemplo** A continuación se muestran dos imágenes, a la izquierda la original, y a la derecha la imagen con el filtro Broken aplicado.



(a) Imagen original



(b) Filtro aplicado

## 2.3. Gamma

Alterar la gama de una imagen corresponde a cambiar la intensidad de los píxeles que la componen. Esta transformación se conoce también como Power Law Transform y la ecuación que aplica a cada componente de los píxeles de la imagen es:  $\text{Output} = 255 \cdot (\text{Input}/255)^{(1/\gamma)}$  ( $\gamma$  es el parámetro Gamma). Este filtro busca implementar una simplificación del filtro general, considerando el parámetro  $\gamma = 2$ . Es decir, calculando la raíz cuadrada de cada componente de cada píxel. A continuación se presenta el pseudocódigo de la operatoria del filtro presentado:

---

```

1: for i ∈ (0, . . . , height − 1) do
2:   for j ∈ (0, . . . , width − 1) do
3:     dstMatrix[i][j].r = sqrt( (srcMatrix[i][j].r) * 255.0 )
4:     dstMatrix[i][j].g = sqrt( (srcMatrix[i][j].g) * 255.0 )
5:     dstMatrix[i][j].b = sqrt( (srcMatrix[i][j].b) * 255.0 )

```

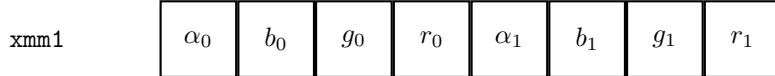
---

Ya que en la implementación de este trabajo práctico utilizamos  $\gamma = 2$ , podemos simplificar la ecuación del algoritmo a  $\text{Output} = \sqrt{\text{Input} \cdot 255}$ . Por lo tanto tenemos unas instrucciones menos en la implementación en ASM.

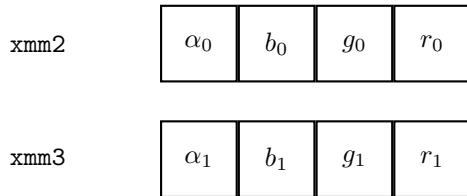
### 2.3.1. Implementación

A continuación se realizara un paso a paso del algoritmo del filtro Gamma que fue implementado en ASM. El algoritmo comienza con un proceso de inicialización de variables, estas incluyen las direcciones de las imágenes original y destino, ancho y alto de la imagen, etc. También inicializamos contadores de filas y columnas para mas adelante iterar.

Al comenzar la ejecución del filtro, luego de crear y guardar las variables temporales, entramos al primer ciclo, que es el de filas. Allí se inicializa en 0 el contador de columnas, que es el que utilizaremos en el ciclo de columnas. Una vez dentro del ciclo de columnas, se traen 2 píxeles de la imagen `src`. Los cuales son extendidos con ceros.



Luego el registro `xmm1` es copiado a los registros `xmm2/3`, en donde se desempaquetan los píxeles,  $p_0$  en `xmm2` y  $p_1$  en `xmm3`.



Luego, procede a realizar la multiplicación (`pmulld`) de cada componente del pixel por 255.

xmm2	$\alpha_0 * 0xff$	$b_0 * 0xff$	$g_0 * 0xff$	$r_0 * 0xff$
------	-------------------	--------------	--------------	--------------

xmm3	$\alpha_1 * 0xff$	$b_1 * 0xff$	$g_1 * 0xff$	$r_1 * 0xff$
------	-------------------	--------------	--------------	--------------

Se convierten a double (`cvtdq2ps`) ambos registros para aplicarles la raíz cuadrada (`sqrtps`).

xmm2	$\sqrt{\alpha_0 * 0xff}$	$\sqrt{b_0 * 0xff}$	$\sqrt{g_0 * 0xff}$	$\sqrt{r_0 * 0xff}$
------	--------------------------	---------------------	---------------------	---------------------

xmm3	$\sqrt{\alpha_1 * 0xff}$	$\sqrt{b_1 * 0xff}$	$\sqrt{g_1 * 0xff}$	$\sqrt{r_1 * 0xff}$
------	--------------------------	---------------------	---------------------	---------------------

Renombrando cada componente del pixel a  $c'_i$ , se convierten los pixeles a entero, y se empaquetan para que queden ambos pixeles en el mismo registro.

xmm3	$\alpha'_0$	$b'_0$	$g'_0$	$r'_0$	$\alpha'_0$	$b'_0$	$g'_0$	$r'_0$
------	-------------	--------	--------	--------	-------------	--------	--------	--------

Finalmente se almacenan los píxeles en la imagen destino, se incrementan los contadores, hasta que cumplan las condiciones y el programa termine.

**Ejemplo** A continuación se muestran dos imágenes, a la izquierda la original, y a la derecha la imagen con el filtro aplicado.



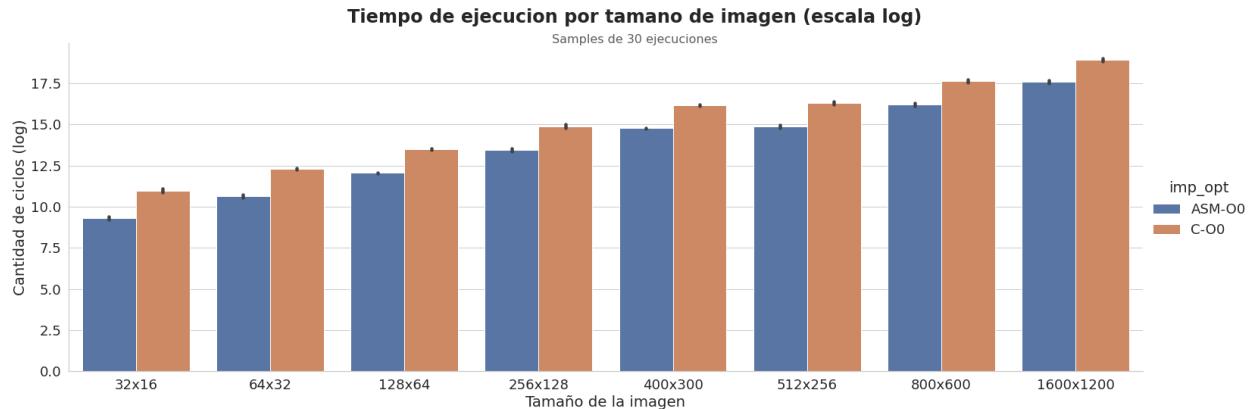
(a) Imagen original

(b) Filtro aplicado

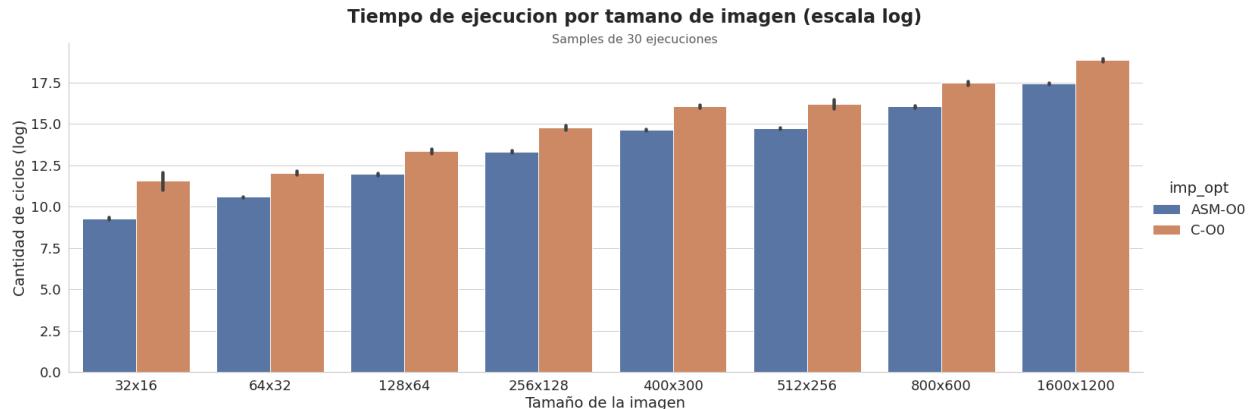
### 3. Comparación

Primero vamos a ver como ambas implementaciones varían según el tamaño de la imagen de entrada.

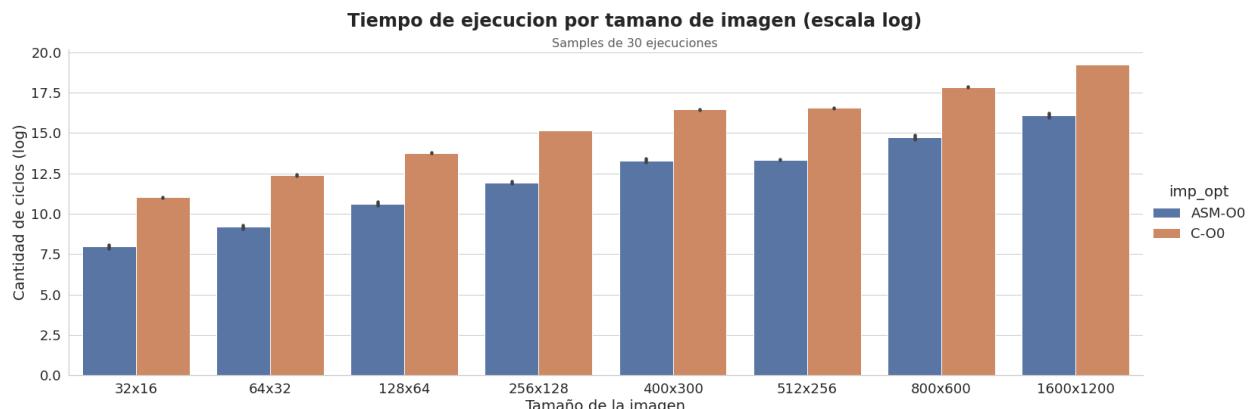
**Max** Comparaciones para el filtro *Max*



**Broken** Comparaciones para el filtro *Broken*



**Gamma** Comparaciones para el filtro *Gamma*



Al estar en escala log, pequeñas diferencias en números grandes implican que la diferencia real es grande. Por lo tanto podemos ver que la implementación en **ASM** es mucho mas rápida que la implementación en **C** en todas las imágenes analizadas.

A continuación se mostraran gráficos comparando los tiempos de ejecución para la implementación en **ASM** y **C**, junto con sus distintas optimizaciones (Para la versión en **C**), siendo ejecutados en la misma imagen de tamaño 1280x720.

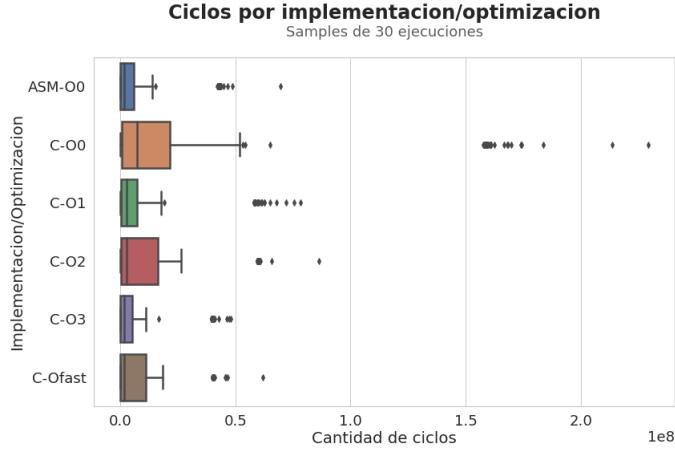


Figura 11: *Max*: Iteraciones promedio por ejecución

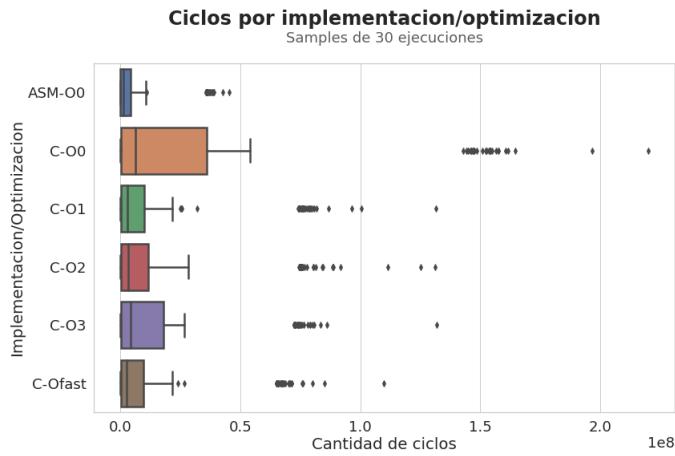


Figura 12: *Broken*: Iteraciones promedio por ejecución

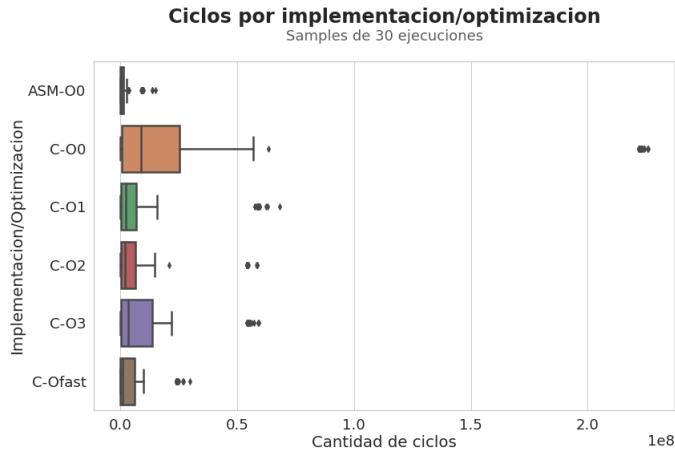


Figura 13: *Gamma*: Iteraciones promedio por ejecución

Se puede ver como las optimizaciones en C impactan directamente en su tiempo de ejecución, mejorando mucho en comparación a la optimización 00. De todas formas, la implementaciones en **ASM** son mucho más veloces. Se puede ver que la optimización **Ofast** en el filtro *Max* es más lenta que la optimización 03. Notar el crecimiento de ciclos requeridos en la optimización 03 en comparación a las optimizaciones 01 y 02<sup>1</sup> en el caso del filtro *Gamma*. También podemos ver como **Ofast** performance similar a 01 y 02. Podemos concluir que las optimizaciones disminuyen el tiempo de ejecución de los filtros, pero las diferencias entre 01, 02 y

<sup>1</sup><https://stackoverflow.com/a/43941854/5813798>

`Ofast` no son tan notables.

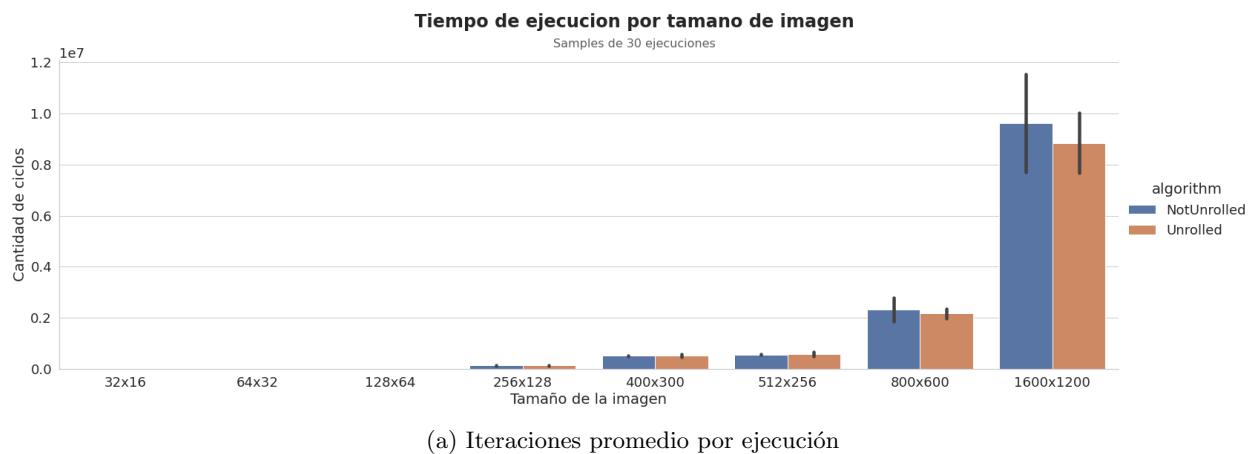
También podemos ver como en el caso del filtro `Max` si vemos el caso `O3` vs `ASM`, se ve que el filtro implementado en `Assembly` performance peor que su contra parte optimizada en `C`, esto se puede deber a que en la implementación se usaron muchas instrucciones para reordenar los pixels y crear una transposición de la matriz, cuando en realidad pudieron haber sido instrucciones como `Insert` y `Blend`, las cuales reducirían los tiempos del filtro.

## 4. Experimentación

### 4.1. Matriz vs Vector

El código del filtro `Gamma` fue modificado para desarmar los ciclos, es decir, en vez de recorrer por filas y columnas (matriz), en dos ciclos, se toma la imagen como un vector continuo, de tamaño `srcWidth · srcHeight`. De esta forma se ejecutan menos instrucciones de comparación, suma, etc.

**Hipótesis** La ejecución debería mejorar (marginalmente, en términos de tiempo), ya que se ejecutan una (pequeña) menor cantidad de instrucciones.



Efectivamente vemos que se reduce el tiempo de ejecución, pero de forma casi despreciable, en particular viendo que esto sucede en las imágenes de “gran” tamaño. Lo que si es interesante notar, es como disminuye la varianza en los tiempos de ejecución para el algoritmo *Unrolled* en los casos mas grandes. Luego en casos de imágenes pequeñas es despreciable la diferencia. Podemos decir que es valida la hipótesis para imágenes de tamaño mayor.

### 4.2. Limpieza de Cache y recorrido vertical en Broken

Por ultimo, vamos a experimentar sobre el filtro `Broken` el recorrer la imagen en sentido vertical en contrario al recorrido horizontal que se hizo en la implementación del filtro.

**Hipótesis** El recorrido vertical afectará la performance del filtro ya que va a generar una mayor cantidad de `miss` al levantar píxeles de la cache. Pero no va a ser peor que la implementación sin cache (con `CLFLUSH`) ya que igualmente se pueden generar algunos `hits`.

Si vemos la Figura 15 podemos ver como la hipótesis se cumple. La implementación por columnas se comporta peor que por filas pero no tan mal como con `clflush`. Pensemos que esto está muy ligado a los tamaños de la caché, el recorrido horizontal del algoritmo original logra tener un porcentaje de `hits` alto en la caché ya que pide píxeles que por lo general están cerca en memoria y es muy probable que estén cacheadas. En el caso de recorrido vertical, la totalidad de la imagen no entra en caché, o en pocas líneas de caché, lo que causa que cada pedido de memoria sea muy probable a tener un `miss` en caché por cambiar de línea constantemente. Concluyendo que, por esto, el algoritmo genera un porcentaje alto de `miss` en caché. Pero igualmente logra generar algunos `hits`, por eso sigue siendo mejor que la implementación de `clflush` en la que el 100 % son `miss`.

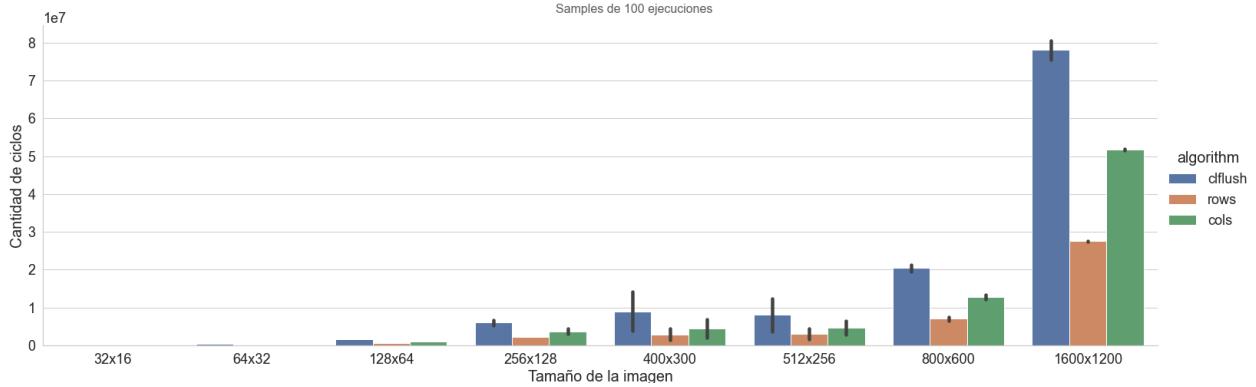


Figura 15: Comparación CLFLUSH vs Recorrido Vertical vs Recorrido Horizontal

## 5. Conclusión

La conclusión general mas importante que podemos sacar de este trabajo es la diferencia en tiempo de ejecución de aquellas implementaciones con instrucciones/tecnología SIMD, versus aquellas que no la utilizan. El paralelismo que se logra con las instrucciones SIMD no se puede alcanzar incluso con optimizaciones al código en C. Se puede ver claramente la diferencia de los mismos.

También pudimos ver la importancia de usar las instrucciones correctas a la hora de implementar en **Assembly** un filtro, ya que en el caso de **Max** por no haberlo hecho, causó que la implementación no performe como lo esperado y una optimización del código hecho en **C** lo hace mejor. De acá se puede ver como una oportunidad de mejora, reimplementar el mismo usando las instrucciones del tipo **Insert** y **Blend**.

Otra conclusión muy importante que podemos extraer de este trabajo es la importancia de la memoria caché en los algoritmos y de implementaciones que aprovechen el uso de la misma. Notamos como la memoria caché, al ser aprovechada, reduce significativamente el tiempo de ejecución en las funciones de Assembler.

