



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Sin Filtro

Organización del Computador II
Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Ignacio Alonso Rehor	195/18	arehor.ignacio@gmail.com
Manuel Panichelli	72/18	panicmanu@gmail.com
Vladimir Pomsztein	364/18	blastervla@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

Este paper cubre un análisis de rendimiento y efectividad del uso de instrucciones *SIMD* por sobre instrucciones normales o implementaciones en C.

Índice

1. Introducción	3
2. Desarrollo	3
2.1. Metodología	3
2.2. Filtros	3
2.2.1. Nivel	3
2.2.2. Bordes	4
2.2.3. Rombos	8
3. Experimentos	11
3.1. Sobre la performance del filtro de Nivel	11
3.2. Desempaqueado: Precisión y Rendimiento en el filtro Bordes	12
3.3. Precálculo de valores en Rombos	12
4. Resultados y discusión	13
4.1. Performance del filtro de Nivel	13
5. Desempaqueado: Precisión y Rendimiento en el filtro Bordes	15
5.1. Precálculo de valores en el filtro de Rombos	18
5.2. Posibles mejoras a implementaciones	19
5.2.1. Bordes	19
5.2.2. Rombos	19
6. Conclusiones	19

1. Introducción

El procesamiento de imágenes resulta un interesante e importante foco de estudio en tiempos modernos, teniendo ésta área múltiples aplicaciones en distintos campos de la ciencia y el marketing o entretenimiento; es fundamental para la programación de inteligencias artificiales y también puede ser utilizado como un divertimento cuando aplicamos filtros en nuestra aplicación de celular favorita. Esta experimentación se avoca a analizar 3 filtros distintos que son aplicables sobre imágenes, dos de ellos sobre imágenes de 4 canales (3 de color y un cuarto de *Alpha*, u opacidad) y un tercero que se aplicará sobre imágenes de 1 canal (es decir, en blanco y negro).

De cada filtro se realizarán distintos experimentos, según amerite su comportamiento y complejidad.

2. Desarrollo

2.1. Metodología

Para la implementación de los distintos filtros, se hizo uso del lenguaje Assembly para la arquitectura x86 de procesadores (en particular los de 64 bits). Aprovechando las herramientas dispuestas por la tecnología utilizada, hicimos fuerte uso de las instrucciones **SIMD** (o Single Instruction Multiple Data) para optimizar los filtros y garantizar ejecuciones más rápidas de los mismos. Será uno de nuestros intereses analizar, luego, cuánto mejor funcionan nuestras implementaciones gracias a su uso.

A su vez, contamos con implementaciones de los algoritmos que describiremos a continuación escritas en C. Usaremos estas implementaciones como benchmarks, o puntos de partida sobre los que analizar la performance de nuestras implementaciones en Assembly.

2.2. Filtros

A continuación, se presentan los 3 filtros implementados, con una breve explicación de cada uno y algunas aclaraciones, relevantes según el caso, acerca de la implementación en particular.

Como fue mencionado en la introducción, dos de estos filtros trabajan con imágenes de 4 canales mientras que uno de ellos utiliza imágenes de canal único.

Las imágenes multi-canal utilizan los canales Rojo, Verde, Azul y Alpha (de 1 Byte cada uno, representando números enteros no signados), en ese orden. Las imágenes de canal único poseen simplemente el canal de Brillo (de 1 Byte representando un entero sin signo también).

2.2.1. Nivel

El más sencillo de los 3 filtros, la idea del filtro de Nivel es sencilla; detectar qué bits están en 1 en la representación binaria de cada canal, para cada píxel. Así, el objetivo es proveer un número i que actuará de índice, y saturar (para cada píxel) todos los canales en los que el valor del bit i sea 1, y llevarlos a 0 en caso contrario.

Puesto que es el filtro más sencillo, no hay demasiadas cosas para destacar de él. Su implementación con instrucciones SIMD realiza los siguientes pasos:

1. Se crea una máscara de 128 bits en un registro xmm donde todos los bits se encuentran en 0, exceptuando el bit del nivel indicado por parámetro (para cada byte del registro).
Por ejemplo, para el nivel 3, la máscara quedaría: 0x0808080808080808 (00001000 para cada byte).
2. Se procede a levantar 4 píxeles (con sus respectivos 4 canales de 1 Byte cada uno) en simultáneo.
3. Se realiza un **AND** con la máscara creada.
4. Luego, se compara cada byte para saber si es igual a 0. De esta manera, negando el resultado de la comparación obtenemos (casi) los bytes a escribir.
5. Se sanitiza los canales Alpha con una operación **OR**, consiguiendo como resultado dejar intactos todos los otros canales y configurar el Alpha en 255

6. Finalmente, se escribe el resultado descripto en la imagen destino.

A fines de demostrar su funcionamiento, podemos apreciar la siguiente imagen:



Figura 1: Imagen original (izq.) vs imagen con filtro de nivel 6 aplicado (med.) vs imagen con filtro de nivel 7 aplicado (der.)

2.2.2. Bordes

El siguiente es el filtro de “Bordes”, cuyo objetivo es remarcar (valga la redundancia) los bordes de los objetos presentes en la imagen original. Este filtro resulta particularmente interesante dados sus campos de aplicación; es frecuentemente utilizado para “Computer Vision”, por lo que este filtro forma parte del cinturón de herramientas básico de cualquier programador de sistemas inteligentes de reconocimiento de características en imágenes.

Una característica particular de este filtro es el hecho de que las imágenes que procesa poseen un único canal, es decir que son imágenes en escala de grises. Para la implementación de este filtro se utilizó el “Operador Sobel”, que consiste en calcular las diferencias de los cambios horizontales y verticales de los puntos que se encuentran alrededor del píxel a procesar.

La manera mediante la que obtenemos estas diferencias en la práctica es calculando los resultados de las siguientes matrices para cada píxel de la imagen. La idea sería, mediante el uso de estas matrices, reemplazar cada píxel por el valor de la suma de todos sus vecinos multiplicados por un factor (especial para cada posición, que es justamente dado por la matriz).

$$G_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} \quad G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix}$$

Notemos que, dada la forma de las matrices, se le estará dando mayor importancia a los píxeles vecinos adyacentes, por sobre los diagonales. Es decir que su valor ponderado será mayor.

Sean $height$ y $width$ el alto y ancho de la imagen respectivamente, y sean src y dst las matrices de las imágenes fuente y destino respectivamente, entonces el algoritmo para la aplicación del filtro resulta ser el siguiente:

```

1: for  $i$  in  $1 \dots height - 2$  do
2:   for  $j$  in  $1 \dots width - 2$  do
3:      $totalG_x \leftarrow 0$ 
4:      $totalG_y \leftarrow 0$ 

5:     for  $ii$  in  $0 \dots 2$  do
6:       for  $jj$  in  $0 \dots 2$  do
7:          $totalG_x \leftarrow totalG_x + G_x[ii][jj] \times src[i + ii - 1][j + jj - 1]$ 
8:          $totalG_y \leftarrow totalG_y + G_y[ii][jj] \times src[i + ii - 1][j + jj - 1]$ 
9:       end for
10:      end for

11:       $dst[i][j] \leftarrow SAT(|totalG_x|, |totalG_y|)$ 
12:    end for
13:  end for

```

Al implementarlo con instrucciones *SIMD*, el objetivo será procesar más de un píxel en simultáneo. Luego, es importante notar que si dos píxeles son consecutivos, entonces todos sus vecinos también lo son entre sí; y más aún, sus posiciones de memoria también serán contiguas entre sí. (ver Figura 2).

A_1	B_1	C_1	D_1	E_1
A_2	B_2	C_2	D_2	E_2
A_3	B_3	C_3	D_3	E_3
A_4	B_4	C_4	D_4	E_4
A_5	B_5	C_5	D_5	E_5

Cuadro 1: Vecinos de un píxel (C_3 , resaltado en rojo).

...	A_2	B_2	C_2	D_2	E_2	A_3	B_3	C_3	D_3	E_3	A_4	B_4	C_4	D_4	E_4	...
-----	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-----

Cuadro 2: Disposición en memoria de un píxel y sus vecinos.

...	A_2	B_2	C_2	D_2	E_2	A_3	B_3	C_3	D_3	E_3	A_4	B_4	C_4	D_4	E_4	...
-----	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-------	-----

Cuadro 3: Disposición en memoria del píxel consecutivo al anterior y sus vecinos.

Teniendo en cuenta cómo se representa una imagen en memoria, podemos explotar esta propiedad para la implementación del filtro utilizando *SIMD*. La idea principal de la implementación sería la de ir recorriendo la imagen almacenando en registros “XMM” los vecinos de los píxeles a procesar. Esto resulta beneficioso para nosotros puesto que, por lo visto anteriormente, podemos así asegurar que en cada uno de estos registros se encuentran los mismos tipos de vecinos para todos los píxeles a procesar. Es decir, un registro “XMM” tendrá toda la tira de vecinos superiores derechos, otro guardará todos los vecinos superiores del medio, etc.

Para la implementación decidimos procesar 8 píxeles en simultáneo, pues esto simplificará el algoritmo por motivos que veremos más adelante.

Para recorrer la imagen, utilizaremos 3 punteros distintos que apunten a las filas anterior, actual y siguiente de los píxeles a procesar. Luego, para obtener los vecinos simplemente le sumaremos el valor correspondiente (0, 1 o 2) a estos punteros.

Por último, al final de cada iteración incrementaremos cada uno de estos punteros en 8, ya que (como ya expusimos anteriormente) esa es la cantidad de píxeles que procesaremos en simultáneo. Estaremos sobreescribiendo los límites de la imagen, pero los corregiremos más adelante.

Ahora bien, haremos este recorrido de píxeles hasta llegar a la última fila de la imagen, pues no podríamos aplicar el filtro sobre píxeles que no poseen un conjunto completo de vecinos. Para obtener la

cantidad de píxeles (o bytes, recordemos que la imagen posee un único canal de un byte lo que implica que un píxel es equivalente a un byte) hasta la última fila de la imagen basta con:

```
; Suponiendo que ecx = height y edx = width

mov ecx, ecx          ; rcx = 0x00000000 | height
mov rax, rcx          ; rax = 0x00000000 | height
mov edx, edx          ; rdx = 0x00000000 | width
dec rax               ; rax = height - 1

; Como width y height ambos ocupan 32 bits, podemos asegurar que el
; resultado de la siguiente multiplicación ocupa como máximo 64 bits,
; luego este estará almacenado en su totalidad en rax.

mul rdx               ; rax = width * (height - 1) = width * height - width
```

Ahora bien, plantear este como nuestro límite nos genera un problema al acercarnos al final de la imagen: al procesar los últimos 8 píxeles de estaríamos realizando una lectura inválida, pues estaríamos tratando de leer fuera del rango de la imagen. Supongamos que, con el siguiente esquema de memoria intentáramos procesar el filtro para el píxel A_3 (y por consiguiente también los píxeles $B_3 \dots H_3$, dado el procesamiento en simultáneo). En ese caso, al levantar los píxeles (o bytes, en este caso son lo equivalentes) de la esquina inferior derecha (en este ejemplo B_4), estaríamos pasándonos de los límites de memoria de la imagen, haciendo así una lectura inválida (ver tabla a continuación, que ilustra este problema).

...	A_1	B_1	C_1	D_1	E_1	F_1	G_1	H_1
...	A_2	B_2	C_2	D_2	E_2	F_2	G_2	H_2
...	A_3	B_3	C_3	D_3	E_3	F_3	G_3	H_3
...	A_4	B_4	C_4	D_4	E_4	F_4	G_4	H_4

Una solución para esto es procesar los últimos 8 píxeles de la imagen por separado. Veremos esto con más en detalle en breve.

Con estas nuevas consideraciones, debemos cambiar nuestro límite para detenernos cuando el puntero de la fila actual llegue a los 8 píxeles anteriores a la ultima fila.

Sean $r14$ el puntero a la última fila, $r11$ el puntero a la fila actual, rdi el puntero al comienzo de la imagen en memoria y rax la cantidad de filas - 1:

```
lea r14, [rdi + rax - 8]
cmp r11, r14
je .fin
```

Finalmente, llegamos a la aplicación del filtro. Para hacer esto, una vez que tengamos los valores de los vecinos de cada píxel en registros XMM , comenzaremos restando los registros que deberían ser multiplicados por -1 contra aquellos que serán multiplicados por su contraparte positiva ($+1$). Notemos que hay que tener cuidado y desempaquetar antes de restar, pues al restar dos enteros no signados debemos poder interpretar el resultado como un entero signado. Luego si teníamos 8 bits para representar el entero no signado, perderíamos precisión si el resultado de la resta esta fuera del rango $[-2^7, 2^7]$.

Luego debemos desempaquetar cada uno de los registros XMM .

```
punpck2bw xmm1, xmm15
```

Donde $xmm15$ es un registro XMM seteado en 0. El resultado de desempaquetar los registros es el siguiente:

```
xmm1 = ... | 0x00 | A3 | 0x00 | A2 | 0x00 | A1 | 0x00 | A0
```

Notemos que no resulta necesario que copiar el contenido de xmm1 para desempaquetar pues solo vamos estar trabajando con los 8 bytes del registro que almacenamos en la parte baja del mismo. Por esto, no perderemos ninguna información al desempaquetar.

Restamos de a pares (xmm3 - xmm1, xmm5 - xmm4 y xmm8 - xmm6). Luego para multiplicar por 2 simplemente shifteamos cada word 1 bit a la izquierda.

```
psllw xmm5, 1
```

Una vez que sumamos todos los registros, le calculamos el valor absoluto. Para esto podríamos utilizar máscaras para ver cuales de los bytes son negativos y luego invertirlos usando una operación XOR, pero es más fácil utilizar la instrucción Intel nos provee para hacer esto:

```
pabs xmm3, xmm3
```

Donde xmm3 es el registro donde almacenamos el resultado de la suma entre todos los otros registros. Para la segunda matriz el procedimiento resulta análogo.

Por ultimo guardamos el resultado de la saturación de la suma entre los resultados de las operaciones para ambas matrices y lo escribimos en la posición de memoria correspondiente.

Ahora, como habíamos anticipado anteriormente, necesitamos procesar los últimos 8 píxeles por separado. Para esto lo que haremos es levantar los últimos 8 píxeles de cada fila apuntada por un puntero (fila anterior, actual y siguiente de los píxeles a procesar). Es decir que todos los vecinos superiores tendrán por el momento los mismos 8 píxeles almacenados en sus registros correspondientes. Luego lo que haremos para mantener la información consistente en estos registros será limpiar los bytes que corresponden a píxeles que no son vecinos correctos de los píxeles a procesar. Por ejemplo, al levantar en xmm3, los últimos 8 píxeles de la fila anterior, los primeros dos píxeles no serán vecinos superiores izquierdos de los píxeles a procesar, luego lo que hacemos para estos registros es shiftearlos 1 o 2 bytes a la derecha, según corresponda.

```
movq xmm1, [r10]      ; r10 -> fila n-2 : ____ | ____ | ... | ____  
movq xmm4, [r11]      ; r11 -> fila n-1 : ____ | ____ | ... | ____  
movq xmm6, [r12]      ; r12 -> fila n   : ____ | ____ | ... | ____  
  
movdqu xmm2, xmm1    ; xmm2[63:0] = p_7 | p_6 | p_5 | p_4 | ... | p_0  
psrlqd xmm2, 1       ; xmm2[63:0] = ___ | p_7 | p_6 | p_5 | ... | p_1  
movdqu xmm3, xmm1    ; xmm3[63:0] = ___ | ___ | p_7 | p_6 | ... | p_2  
psrlqd xmm3, 2       ; xmm3[63:0] = ___ | ___ | p_7 | p_6 | ... | p_2
```

Llegando a la última parte del proceso del filtro, debemos dejar un marco de 1 píxel de color blanco alrededor de la imagen. Para ello, vamos a separar el “rellenado” de los bordes verticales con el de los bordes superior e inferior.

Para “rellenar” estos últimos, vamos a recorrer la primera utilizando un puntero, y en simultáneo recorremos la última fila sumándole a este puntero el offset correspondiente. Este offset ya lo calculamos previamente al multiplicar la cantidad de filas por la cantidad de columnas decrementada en 1. Este dato lo habíamos almacenado en *rax*.

A su vez, necesitamos en memoria especificar el color blanco. Y, como las filas son múltiplos de 8, iremos llenando de a 8 píxeles.

```
section .rodata:  
edges: TIMES 8 db 255
```

```

section .text:
    movq xmm0, [edges]

    lea r11, [r10 + rax]
    movq [rsi + r10], xmm0
    movq [rsi + r11], xmm0
    add r10, 8

```

Por último, para los bordes laterales recorreremos la última columna de la imagen, abusando del hecho de que al posicionarnos sobre el borde derecho de una fila, la siguiente posición de memoria corresponderá a la del borde lateral izquierdo de la fila siguiente.

Luego si solamente escribimos 2 píxeles posicionándonos sobre esta columna, estaríamos llenando todos los píxeles de los bordes horizontales salvo 2: el píxel del borde izquierdo de la primer fila y el píxel del borde derecho de la última. Pero, resulta fácil notar que estos pertenecen a su vez a los bordes superior en inferior, por lo que ya tenemos el caso cubierto de antemano.

2.2.3. Rombos

Finalmente, se encuentra el filtro de Rombos, cuyo objetivo es distorsionar levemente la imagen de manera tal que aparezcan unos tenues rombos de tamaño específico sobre la imagen.

Para lograr esto, parte la imagen en sectores cuadrados del tamaño de rombo especificado (64 en este caso), y calcula un valor a sumar al píxel que depende exclusivamente de la posición *relativa* del píxel dentro de su respectivo cuadrado *padre*.

Como lo calculado no depende del valor del píxel en sí, todos los píxeles que estén en la misma posición relativa pero en diferentes rombos tendrán sumados el mismo valor. Luego, podría ser precalculado. Sin embargo, la implementación *inicial* realizada para este filtro se limita a realizar el cálculo en cada iteración.

```

1: size ← 64
2: for i in 0...height - 1 do
3:   for j in 0...width - 1 do
4:     ii ← ( $\frac{\text{size}}{2} - (i \% \text{size})$ ) > 0? ( $\frac{\text{size}}{2} - (i \% \text{size})$ ) : - $\frac{\text{size}}{2} - (i \% \text{size})$ 
5:     jj ← ( $\frac{\text{size}}{2} - (j \% \text{size})$ ) > 0? ( $\frac{\text{size}}{2} - (j \% \text{size})$ ) : - $\frac{\text{size}}{2} - (j \% \text{size})$ 
6:     x ← ( $ii + jj - \frac{\text{size}}{2}$ ) >  $\frac{\text{size}}{16}$ ? 0 :  $2 \times (ii + jj - \frac{\text{size}}{2})$ 
7:     dst[i][j].b ← SAT(src[i][j].b + x)
8:     dst[i][j].g ← SAT(src[i][j].g + x)
9:     dst[i][j].r ← SAT(src[i][j].r + x)
10:    dst[i][j].a ← 255
11:  end for
12: end for

```

El pseudocódigo anterior, a pesar de no hacer uso de **SIMD**, es útil para entender el funcionamiento de la implementación en assembler, cuya idea general es levantar los índices relativos en un registro *xmm*, para luego de varios cálculos llegar al *x* que es sumado a los píxeles.

Dado que los valores calculados siempre entran en un *short* (16 bytes) se podrían levantar hasta 8 índices (i.e 4 pares (i, j)) en un registro *xmm*, con lo cual se podrían obtener los *x*-es para 4 píxeles a la vez. Por simpleza, se realizan para solo 2.

Teniendo eso en cuenta, los pasos que realiza la implementación son los siguientes:

1. Levanta las máscaras utilizadas en los ciclos en registros *xmm* para evitar hacerlo reiteradas veces.
2. Inicializa los índices y los offsets.
3. Calcula los *ii*-es y *jj*-es.

```

ii = ((s/2) - (i % s)) > 0 ? ((s/2) - (i % s)) : -((s/2) - (i % s))
= abs(s/2 - i % s)
jj = abs(s/2 - j % s)

```

Donde `abs` es el valor absoluto. Para ello:

- Levanta `size/2` en un registro e `i % s`, `j % s` en otro

```

xmm2 = s/2 | s/2 | s/2 | s/2
xmm1 = i_0 | j_0 | i_1 | j_1

```

Donde `_0` indica que corresponde al primer pixel, y `_1` al segundo.

- Los resta verticalmente (con `psubd`)

```
xmm2 = s/2 - i_0 | s/2 - j_0 | s/2 - i_1 | s/2 - j_1
```

- Toma el valor absoluto del resultado (con `pabsd`)

```

xmm2 = abs(s/2 - i_0) | abs(s/2 - j_0) |
       abs(s/2 - i_1) | abs(s/2 - j_1) |

```

- Finalmente, tiene lo que buscado en el registro

```
xmm2 = ii_0 | jj_0 | ii_1 | jj_1
```

4. Calcula los x-es.

```

x = (ii+jj - s/2) > s/16 ? 0 : 2*(ii+jj - s/2)
= 2*(ii+jj - s/2) > s/8 ? 0 : 2*(ii+jj - s/2)

```

Para ello:

- Notemos que ya están `ii`-es y `jj`-es en un registro
- Los suma horizontalmente (con `phaddd`)

```
xmm2 = ii_0 + jj_0 | ii_1 + jj_1 | ii_0 + jj_0 | ii_1 + jj_1
```

- Les resta `size/2` y multiplica por dos

```

xmm2 = 2*(ii_0+jj_0 - s/2) | 2*(ii_1+jj_1 - s/2) |
       2*(ii_0+jj_0 - s/2) | 2*(ii_1+jj_1 - s/2) |

```

- Compara los resultados con `size/8` (con `pcmpgtd`)

```

xmm2[i] = FFFFFFFF    si 2*(ii_x+jj_x-s/2) > s/8
          00000000    si no

```

- Lleva a 0 los que sean mayores (pandn con el resultado de la comparación)

```

xmm2[i] = 0            si era mayor a s/8
          lo mismo      si no

```

- Finalmente, están los x -es en el registro

$$\text{xmm2} = \text{x_0} \mid \text{x_1} \mid \text{x_0} \mid \text{x_1}$$

5. Los suma a los canales de los píxeles.

Al levantarlos, cada canal es de 1 byte y son enteros sin signo, pero los x -es son signados. Es necesario pasarlos a la misma representación. Para eso, empaqueta los x -es de *doubleword* a *word*, y desempaquetar los píxeles de *byte* a *word*.

- Empaque los x -es a *word*

$$\text{xmm2} = \text{x_0} \mid \text{x_1} \mid \text{x_0} \mid \text{x_1} \mid \text{x_0} \mid \text{x_1} \mid \text{x_0} \mid \text{x_1}$$

- Como quedaron desordenados, los agrupa según a qué píxel pertenecen con dos *pshuflw* (uno para la parte alta, y otro para la baja)

$$\text{xmm2} = \text{x_0} \mid \text{x_0} \mid \text{x_0} \mid \text{x_0} \mid \text{x_1} \mid \text{x_1} \mid \text{x_1} \mid \text{x_1}$$

- Levanta los píxeles

$$\begin{aligned} \text{xmm3}[0:63] &= \quad \quad \quad \text{p}_0 \quad \quad \quad \mid \mid \quad \quad \quad \text{p}_1 \\ &= \text{b}_0 \mid \text{g}_0 \mid \text{r}_0 \mid \text{a}_0 \mid \mid \text{b}_1 \mid \text{g}_1 \mid \text{r}_1 \mid \text{a}_1 \end{aligned}$$

- Los desempaquetan de *byte* a *word*

$$\text{xmm3} = \text{b}_0 \mid \text{g}_0 \mid \text{r}_0 \mid \text{a}_0 \mid \mid \text{b}_1 \mid \text{g}_1 \mid \text{r}_1 \mid \text{a}_1$$

- Los suman verticalmente con los x -es

$$\begin{aligned} \text{xmm3} = \text{b}_0 + \text{x}_0 \mid \text{g}_0 + \text{x}_0 \mid \text{r}_0 + \text{x}_0 \mid _ \mid \\ \text{b}_1 + \text{x}_1 \mid \text{g}_1 + \text{x}_1 \mid \text{r}_1 + \text{x}_1 \mid _ \mid \end{aligned}$$

- Pone el canal *alpha* en 255.

$$\text{xmm3} = _ \mid _ \mid _ \mid \text{FF} \mid _ \mid _ \mid _ \mid \text{FF}$$

- Empaque con saturación los resultados de *word* a *byte*

- Escribe el resultado en memoria

6. Avanza los índices (% size) y los offsets

7. Si no terminó, loopea.

Ejemplo del filtro en acción:



(a) Imagen original



(b) Imagen con el filtro rombos aplicado

3. Experimentos

A continuación, se presentan los experimentos planteados para cada filtro, cada uno con su correspondiente hipótesis y motivación.

3.1. Sobre la performance del filtro de Nivel

Con el objetivo de obtener una idea de cuánto aumenta el rendimiento de las implementaciones al utilizar SIMD (en sus distintos exponentes, operando con más o menos datos en simultáneo) y en comparación con los distintos niveles de optimización del compilador GCC (O0, O1, O2 y O3), decidimos aprovechar el filtro más sencillo para realizar un *benchmarking* de estas mejoras de rendimiento. El objetivo es entender cuánto aumenta la performance cuando aumentamos la cantidad de bytes procesados en simultáneo.

Así, proponemos ejecutar 100 veces cada implementación (con el objetivo de tener varias mediciones que nos permitan ver las diferencias de performance independientemente del ruido generado por el sistema operativo) y luego comparar resultados. Se ejecutarán la implementación de *C*, compilada con los 4 distintos niveles de optimización mencionados previamente (O0, O1, O2, O3) y 3 implementaciones distintas en *ASM*, donde se variará la cantidad de píxeles trabajados en simultáneo con SIMD (tendremos de 1, 2 y 4 píxeles a la vez).

Esperamos que la peor implementación en *C* y la mejor en *ASM* estén a dos niveles distintos, siendo la de *ASM* más performante. Sin embargo, consideramos razonable que la peor implementación de *ASM* esté razonablemente cerca en rendimiento que la mejor de *C*.

Luego, nos gustaría notar que la implementación planteada para el filtro de nivel hace fuerte uso de la instrucción *pcmpeqb*, utilizada para generar el resultado del filtro (se utiliza para generar el resultado negado utilizando la información de cada píxel y la máscara creada previamente, como se detalló en 2.2.1). Sin embargo, es destacable que todo esto bien podría haber sido implementado sencillamente con shifts aritméticos y lógicos. Es por esto, que proponemos una variante de implementación con **shifts**, que funciona de la siguiente manera.

1. Creamos la máscara de 128 bits de la misma manera que la implementación original (ver 2.2.1)
2. Nos guardamos en algún registro *xmm* el resultado de la resta $15 - nivel$, siendo *nivel* el nivel seleccionado por el usuario para el filtro.
3. Acto seguido, hacemos la **AND** con la máscara.
4. Luego, levantaremos de a 4 píxeles a la vez (16 bytes), y desempaquetamos de byte a word, completando con 0's (de esta manera, tenemos dos píxeles en un registro *xmm* y dos en otro, empaquetados ahora de a words).
5. Ahora vamos a shiftear a izquierda $15 - nivel$. Así, el bit de nivel queda en el bit más significativo de cada word empaquetado
6. Inmediatamente después, hacemos un shift aritmético a derecha de 15 bits. De esta manera, si el bit de nivel era un 1, ahora los 15 bits del word serán 1's. Si en cambio era 0, el word entero valdrá 0. Este es exactamente el resultado que buscábamos.
7. Se empaqueta nuevamente en bytes, saturando con signo y se sanitizan los canales Alpha con una operación **OR**, dejándolos en 255.
8. Finalmente, se escribe el resultado en la imagen destino.

Ahora bien, puesto que los shifts son operaciones que consideramos elementales (son componentes usualmente sencillas desde el punto de vista del hardware), esperamos que esta variante de implementación sea al menos un poco más rápida que la que utiliza comparaciones. Más aún, cabe destacar que esta variante no requiere de la negación de ningún resultado, por lo que estaríamos ahorrándonos esa tarea.

Sin embargo, notaremos que la instrucción de shift de SIMD que nos permitiría hacer esto corresponde a la extensión AVX y no a ninguna de las SSE. Es por eso que, además, decidimos proponer otra

variante más, que tenga a su vez 8 implementaciones internas distintas, una para cada nivel. De esta manera, podremos shiftear con inmediatos siendo que cada implementación se utilizará cuando corresponda según el nivel pedido. Lamentablemente, esto significa tener un código muy similar repetido 8 veces, pero podría presentar interesantes mejoras. A esta tercera variante la llamaremos “ASM (branch shifts)”, puesto que con branches decidiremos cuál de las 8 subimplementaciones de shift utilizar.

Nos gustaría, luego, verificar qué implementación resulta mejor (si la de *ASM* con comparaciones, la de shifts o la de branches con shifts), comparando sus rendimientos.

3.2. Desempaquetado: Precisión y Rendimiento en el filtro Bordes

Una de las particularidades de la implementación del filtro bordes es la utilización de instrucciones de desempaquetado. Al procesar simultáneamente distintos píxeles, estas instrucciones nos permiten almacenar el resultado parcial de operaciones aritméticas entre los distintos píxeles, sin perder precisión. Para la implementación del filtro en cuestión, desempaquetamos el valor del píxel de Byte a Word (es decir, de 1 Byte a 2 Bytes). Esto implica que para cada píxel a procesar, necesitaremos el doble de espacio para almacenar los resultados parciales sin perder precisión.

Con esto en mente, la idea detrás de este experimento será medir dos factores: el efecto generado en la precisión de la imagen final sin el uso de instrucciones de desempaquetado, y qué consecuencias produce el uso del doble de almacenamiento sobre el rendimiento del filtro.

Para poder efectuar mediciones sobre esto vamos a contar con una implementación de referencia contra la cual contrastar los resultados de la versión sin instrucciones de desempaquetado. A su vez, vamos a evaluar qué impacto tienen distintos tipos de imágenes a la hora de medir la precisión entre ambas implementaciones.

Para medir la diferencia de precisión entre las dos implementaciones vamos a calcular el promedio de las diferencias de todos los píxeles entre las dos imágenes producidas por las mismas.

Ahora bien, respecto del efecto en el rendimiento, esperamos que la implementación sin instrucciones de desempaquetado sea más eficiente por el hecho de poder procesar más elementos en simultáneo.

Por otro lado, en cuanto a la precisión, esperamos que para imágenes más uniformes con pocos bordes la perdida de precisión no sea mucha pero que, conforme vaya apareciendo más ruido en la imagen, la brecha se vaya ampliando.

Finalmente, esperamos que la mayor diferencia entre las implementaciones se produzca con imágenes donde el valor de cada píxel es aleatorio (a.k.a: white noise).

3.3. Precalculo de valores en Rombos

La implementación inicial de este filtro realiza los cálculos para cada píxel en cada iteración. Esto implica que se repite muchas veces el mismo cálculo, innecesariamente.

El experimento que proponemos consiste en *precalcular* previo al ciclo principal los valores que podrá tener cada posición de cada píxel y almacenarlos, por ejemplo, en una matriz. De esta forma, puede ser obtenido simplemente indexando la matriz desde el ciclo principal. Esta se almacena en el *heap* con memoria solicitada con *malloc*, aunque también podría haberse alojado en la *pila*.

Uno tendería a suponer que con estos cambios la implementación funcionaría mucho más rápido, ya que se están ahorrando muchos cálculos. Pero, esto se hace bajo el costo de tener que ir a memoria una vez más (recordemos que ya tenemos que levantar el píxel) por iteración. Además, los cálculos que se realizan son relativamente simples; no hay operaciones costosas como, por ejemplo, la división.

No obstante, intuimos que será más rápido de todas formas, puesto que la porción precalculada es de pequeño tamaño (32KB) y por lo tanto entraría en una línea de la memoria caché, por lo que la penalización por ir a buscarlo debería ser poco relevante.

Por último, destacamos que por la naturaleza de la mejora, sus efectos deberían hacerse más evidentes mientras más grande sea la imagen de entrada, ya que de esta manera mayor será la cantidad de cálculos que se están precalculando.

Además, para imágenes chicas, debería ser mucho más notoria la penalidad de la ida a memoria adicional, a tal punto que seguramente sea peor.

4. Resultados y discusión

4.1. Performance del filtro de Nivel

En la figura 3, se presenta una imagen en el que se podrá apreciar gráficamente las diferencias de rendimiento para cada implementación, donde cada una de ellas fue ejecutada 100 veces sobre la misma imagen y con el mismo nivel (7). Dejar de lado, de momento, la variación ASM(1 byte), hablaremos de ella en breve.

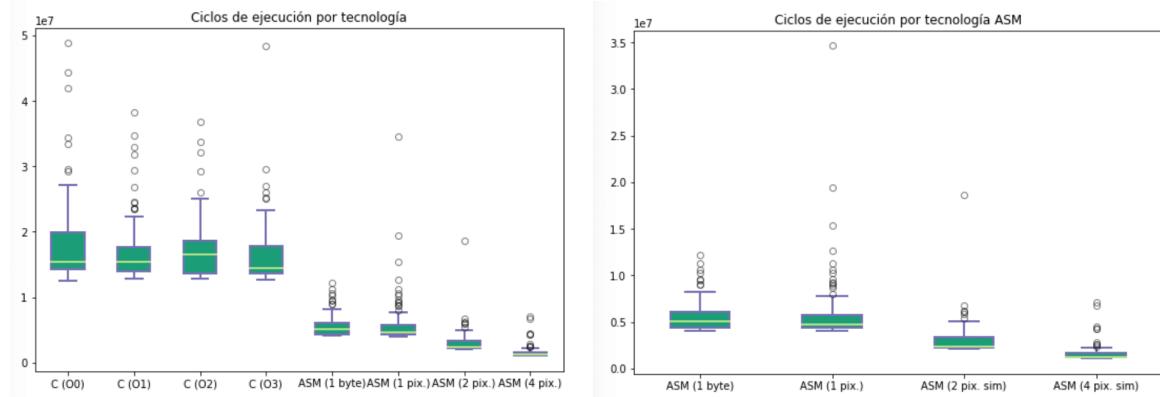


Figura 3: Ciclos de ejecución para cada implementación en *C* y *ASM* con implementación de *ASM* procesado de a 1 byte (izq.) y ciclos de ejecución para implementaciones de *ASM* (der.). Notar diferencias en escalas del eje coordenado

Como puede apreciarse en el gráfico, la diferencia de rendimiento entre las distintas implementaciones en *C* y las de *ASM* es altamente notoria. A esta gran diferencia adjudicamos dos posibles explicaciones:

- La implementación de *C* utiliza dos ciclos, por lo que se nos ocurre que los saltos podrían estar influenciando
- La implementación de *C* levanta un byte a la vez, mientras que la de *ASM* más lenta levanta 1 píxel entero a la vez (por lo tanto, 4 bytes).

Para comprobar esta última hipótesis, decidimos crear una cuarta implementación de *ASM* que levante de a byte, tal y como la implementación en *C* lo hace. Sin embargo, los resultados no fueron los esperados. Esta es la variación *ASM* (1 byte) que mencionábamos previamente y que puede observarse en la figura 3.

Como podemos ver en la figura 3, apenas si hay diferencia entre la implementación de *ASM* con SIMD de 1 píxel en comparación con la que levanta de a bytes. Intuimos que la memoria caché debe ser la responsable de esta falta de diferencia. A su vez, notamos que aún hay una gran diferencia entre la mejor implementación de *C* y esta última, por lo que proponemos utilizar branches (if's) en lugar de shifts aritméticos, como venimos haciendo en la implementación actual. Como veremos en el gráfico 4, esto hace que la implementación empeore levemente, pero no lo suficiente como para acercarse considerablemente a la implementación en *C*.

Además, observando los gráficos de la figura 4, resulta muy notoria la mejora de rendimiento de la implementación con SIMD que levanta de a 4 píxeles. En términos de la mediana, esta implementación corre en $\frac{1}{4}$ del tiempo que la *ASM* (1 byte) y en $\frac{1}{10}$ del tiempo que cualquier implementación de *C*. Podremos ver todas estas comparaciones en la figura 4.

Cabe destacar, por otro lado, que los resultados de rendimiento del filtro para otros niveles fueron los mismos, por lo que el su performance resulta independiente del nivel que se utilice.

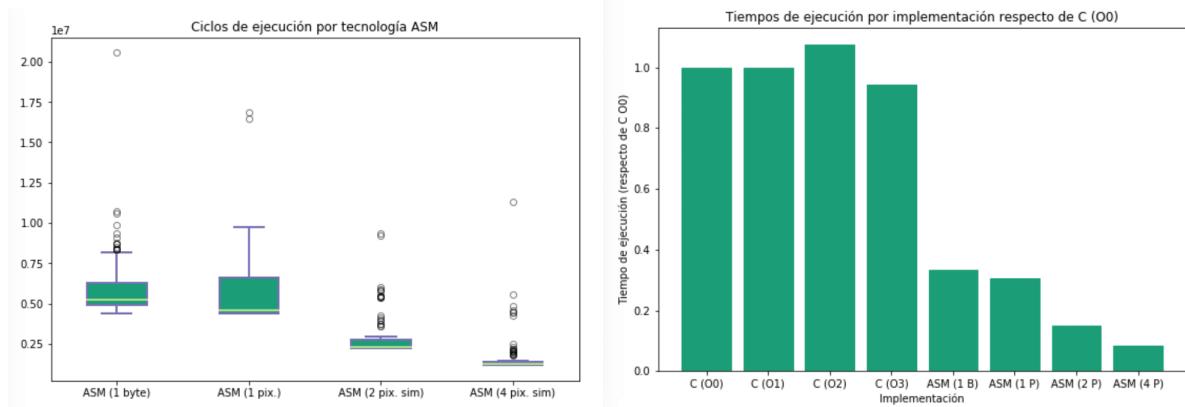


Figura 4: Ciclos de ejecución para implementaciones de *ASM* utilizando branches para la implementación que trabaja de a byte (izquierda) y Tiempo de ejecución para cada implementación respecto de C (00) (derecha). Resultados según la mediana de cada medición

Concluimos, luego, que las mejoras en rendimiento obtenidas al comenzar a trabajar con *SIMD* son excepcionales, como pudimos ver. Por lo tanto, resulta súmamente conveniente utilizarlo de ser posible. Más aún, pudimos comprobar que el uso de *ASM* por sobre *C* tiene sus beneficios, incluso sin aprovechar tecnologías como *SIMD*.

Finalmente, nos queda por realizar el análisis de la implementación de *ASM* con comparaciones en oposición a la que utiliza shifts. Podremos apreciar, en la figura 5, los resultados de este experimento.

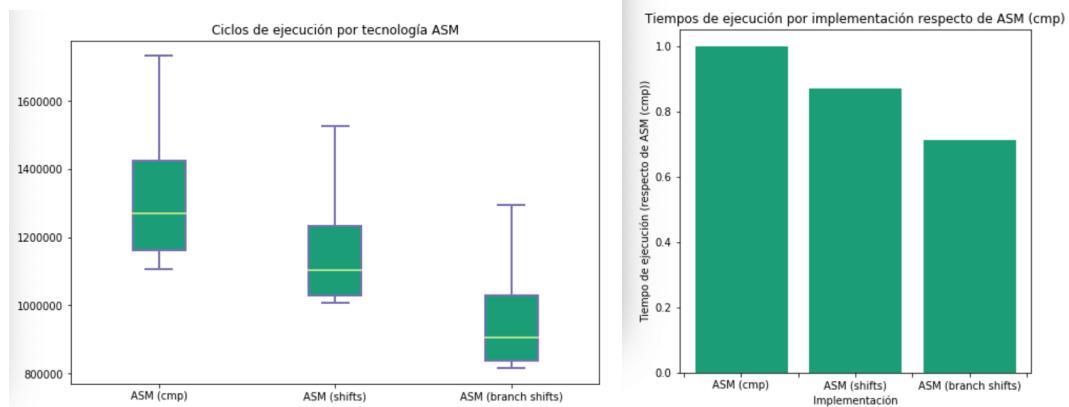


Figura 5: Ciclos de ejecución para implementaciones de *ASM* utilizando *SIMD* con 4 píxeles a la vez y compares vs. shifts vs. shifts con branches (izquierda) y Tiempo de ejecución para cada implementación respecto de *ASM* (compares) (derecha). Resultados según la mediana de cada medición

Como se puede ver, las mejoras son claras. La implementación que utiliza shifts corre $\frac{1}{10}$ más rápido que la implementación sobre comparaciones (es decir, en $\frac{9}{10}$ del tiempo que requiere esta última). Este es un número interesante, sobre todo teniendo en cuenta lo pequeño que resultó este cambio en cuanto a la implementación se refiere. A su vez, la implementación de branch shifts corre $\frac{3}{10}$ más rápido que la de compares (es decir, en $\frac{7}{10}$ del tiempo que requiere esta última). Queda así justificada para bien en términos de performance la repetición del código 8 veces.

De esta manera, podemos concluir que las operaciones de shifteo resultan menos costosas para la máquina que operaciones de comparación, y deberían ser priorizadas por sobre esta última alternativa cuando así pudiera hacerse. Más aún, notamos que la instrucción de shifteo SIMD por inmediato (extensión SSE) resulta bastante más rápida que la de shifteo SIMD por registro XMM (extensión AVX).

5. Desempaquetado: Precisión y Rendimiento en el filtro Bordes

Para encarar este experimento, lo primero que debimos conseguir fue la generación de imágenes con niveles de ruido incrementales. Para ello, decidimos basarnos en el modelo de ruido AWGN¹ (Additive White Gaussian Noise), que trabaja basándose en valores aleatorios bajo la distribución Normal.

Para implementar este modelo, utilizamos el conocido Box Muller Transform, un algoritmo que permite, dado una variable aleatoria con distribución uniforme (presente en todos los lenguajes a través de la función `rand()`), obtener una nueva variable aleatoria con distribución Normal Estándar (es decir, $\mu = 0$ (esperanza) y $\sigma = 1$ (desvío)).

Ahora bien, para conseguir variar el nivel de ruido, nos fue necesario desestandarizar esta muestra para aumentar su desvío, obteniendo de esto mayor rango de valores posibles. Acto seguido, simplemente nos dedicaremos a sumar con saturación las muestras obtenidas a cada canal de cada píxel, obteniendo así una diferencia de su valor original, determinada por el desvío.

Con el fin de determinar el desvío que posee cada nivel de ruido, determinamos uno máximo (σ_m) y pensamos en la cantidad de niveles deseados. Luego, quedará determinada por la fórmula

$$\sigma = i \times \frac{\sigma_m}{n}$$

Donde i es el i -ésimo nivel de ruido y n la cantidad total de niveles. Así $\frac{\sigma_m}{n}$ representa, en cierta forma, el incremento necesario (o step) para pasar de un nivel al siguiente.

La cantidad de niveles que decidimos utilizar es de 28, arbitrariamente, pero bien podría haber sido un número distinto.



Figura 6: Output de las imágenes con ruido para distintos niveles (0, 5, 10 y 20)

Ahora, para generar las imágenes en cuestión, aplicamos las distintas implementaciones del filtro de Bordes a las imágenes (con distintos niveles de ruido).

¹https://en.wikipedia.org/wiki/Additive_white_Gaussian_noise

Para poder medir la precisión, calculamos el promedio de las diferencias de todos los píxeles entre ambas imágenes (la correcta y la obtenida).

Como habíamos esperado, para imágenes con más ruido observamos mayor pérdida de precisión. Los valores de píxeles cercanos se relacionan de alguna manera por su proximidad, es decir, píxeles cercanos tienden a tener valores parecidos, a menos que se trate de un borde. Al agregar ruido, lo que efectivamente estamos generando es mayor diferencia entre un píxel y sus vecinos, que finalmente termina resultando, por la naturaleza de las operaciones realizadas para los cálculos del filtro, en la pérdida de precisión.

Esto es porque, al tener valores muy distintos aumenta la probabilidad de que los resultados parciales de las operaciones se vayan de la representación.

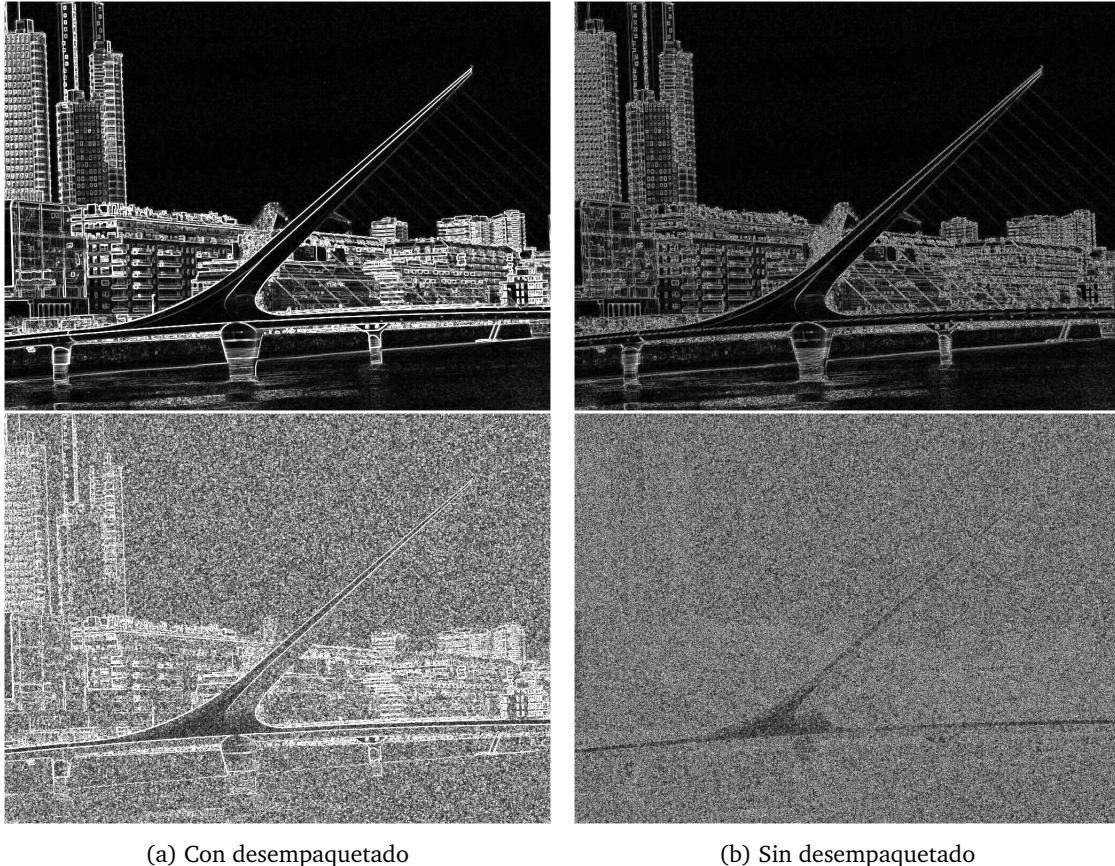


Figura 7: Output de las imágenes generadas por la implementación con desempaquetado (izquierda) vs sin desempaquetado (derecha). Con nivel de ruido 0 (arriba) vs 5 (abajo)

En la figura 8 se puede observar una gráfico cualitativo de las mediciones de precisión para los distintos niveles de ruido. Cabe destacar que, como máximo, obtuvimos una pérdida de precisión cercana al 30 % (es decir, que en promedio los píxeles “distan” entre sí en un 30 % de su valor, en el peor caso).

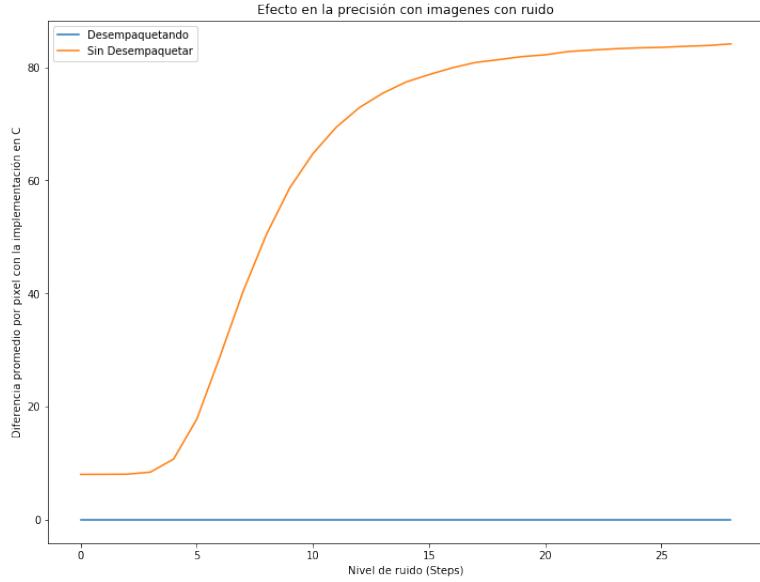


Figura 8: Precisión del filtro de bordes por nivel de ruido (vs versión con desempaquetado)

Además, la versión del filtro que no desempaquetó (y luego, procesa más píxeles por ciclo) resultó ser más eficiente, es decir, tomó menor cantidad de ciclos (como ya esperábamos), como podemos ver en la figura 9.

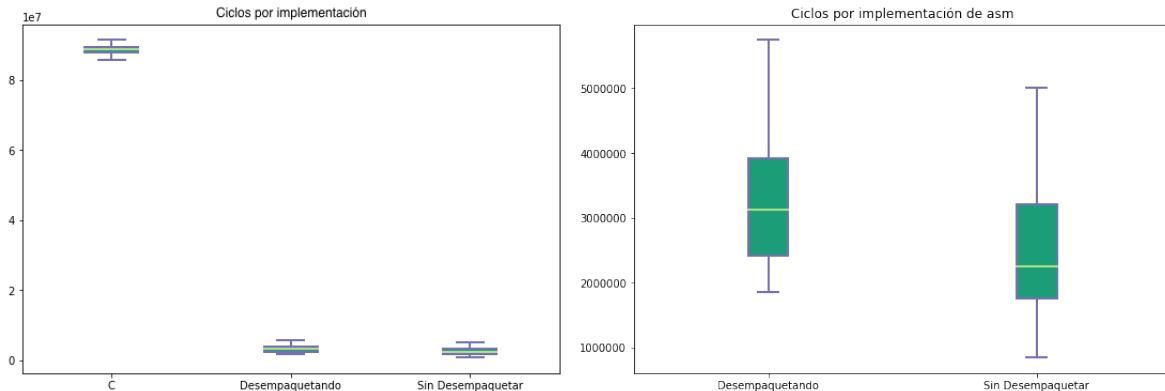


Figura 9: Ciclos ejecución para cada implementación en *C* y *ASM*

5.1. Precálculo de valores en el filtro de Rombos

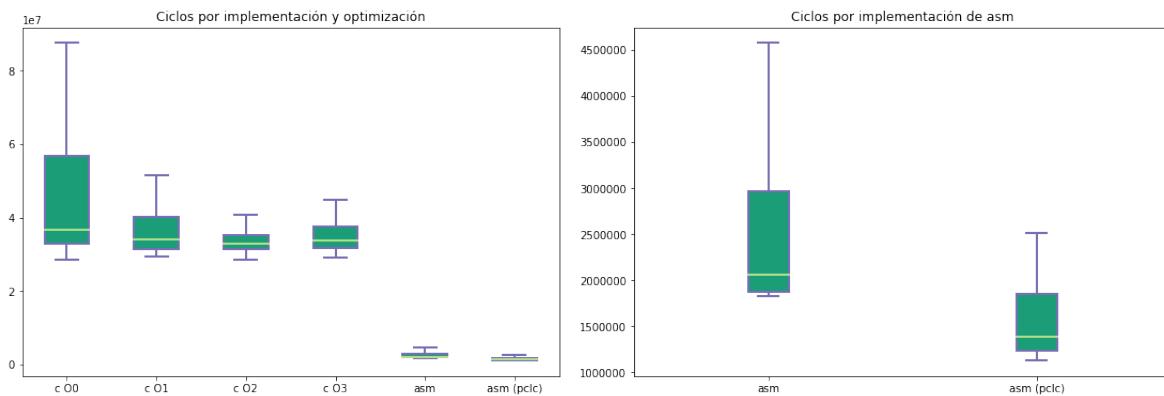


Figura 10: Ciclos ejecución para cada implementación / optimización en *C* y *ASM*

En estas figuras podremos apreciar varios aspectos sobre la implementación de rombos. La metodología usada en este caso fue iterar 100 veces una corrida del filtro sobre una imagen de 1024x600 para cada implementación. Destacamos, luego, las siguientes observaciones:

- La implementación en *ASM* resulta más veloz que cualquier optimización de *C*, como ya era de esperarse luego de los resultados de las experimentaciones del filtro de Nivel.
- Notemos que la implementación que realiza el precálculo resulta más performante en términos de tiempo hasta para una imagen relativamente chica como con la que se realizó la prueba. Y, ¿qué pasará con imágenes de distintos tamaños?

Para responder esa última pregunta, corrimos cada implementación en *ASM* 150 veces para diversos tamaños de imagen.

Los resultados pueden verse en la figura 11.

- Para imágenes chicas, la implementación que no realiza el precálculo es más veloz que la que sí, lo cual se ajusta a lo predecido anteriormente.
- A medida que las imágenes se hacen cada vez mas grande, empieza a notarse más la diferencia en velocidad, y importa cada vez menos el overhead generado por el precálculo y la ida a memoria.

De esta manera, comprobamos que el hecho de tener que ir una vez más por iteración a memoria no produjo mayores impactos en la eficiencia temporal. Por lo tanto, podríamos pensar que la caché está ayudando fuertemente a esta implementación en ese aspecto.

Para mejorar la performance en imágenes más chicas, y que así la versión experimental sea mejor en todos los casos, una alternativa sería tener *hardcodeada* la matriz en memoria para un tamaño dado, y así no hay necesidad de calcularla al principio.

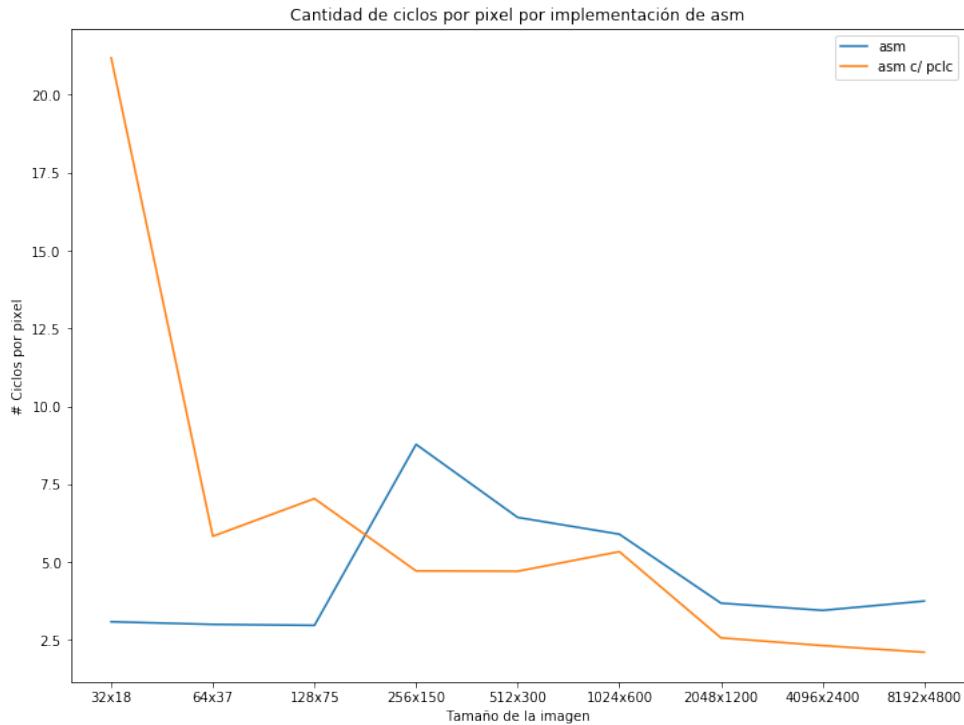


Figura 11: Ciclos de ejecución para cada implementación de ASM

5.2. Posibles mejoras a implementaciones

5.2.1. Bordes

Para la implementación de este filtro, se planteó procesar 8 píxeles en simultáneo para disminuir la cantidad de instrucciones de desempaquetado. La elección fue motivada por aumentar la legibilidad del código y facilitar su escritura; pero si no tenemos en cuenta esos factores, podíamos procesar 16 píxeles en simultáneo, así aumentando el rendimiento de la implementación.

5.2.2. Rombos

Independientemente de la mejora propuesta durante la experimentación, se puede pensar trivialmente en una oportunidad de mejora levantando 4 píxeles a la vez en lugar de 2.

Ya que se levantan valores como enteros (32 bits) cuando en realidad son shorts (16 bits) se podrían almacenar en la mitad de espacio, y procesar el doble a la vez, lo cual permitiría efectivamente levantar 4 píxeles.

6. Conclusiones

Notamos como conclusiones:

El hecho de que el uso de *SIMD* supone una gran mejora en la performance de un algoritmo al ser utilizado, pudiendo mejorar al menos diez (10) veces la velocidad de ejecución del mismo en comparación a otras tecnologías (como *C*). Además, resulta muy notoria la mejora de rendimiento entre implementaciones en *C* y en *ASM*, incluso sin el uso de *SIMD*, y observamos que la memoria caché es una gran aliada cuando este último conjunto de instrucciones no se aprovecha.

A su vez, notamos que la utilización de shifts por sobre comparaciones supone una pequeña mejora en el rendimiento del programa, por lo que debería optarse por usar dichas operaciones en lugar de comparaciones cuando esto fuera posible (como es el caso de la implementación de niveles).

Por otra parte, notamos que evitar desempaquetar puede ser una técnica útil para mejorar la performance de una implementación, pero al costo de pérdida de precisión. Más aún, notamos que esta pérdida de precisión puede depender fuertemente de la imagen de entrada (dependiendo de las operaciones a realizar, por supuesto). Esto supone un buen motivo para considerar seriamente el contexto del problema a resolver antes de utilizar la técnica en cuestión.

En cuanto a lo que a precálculos y su almacenamiento en memoria se refiere, concluimos que esto puede contribuir a notorias mejoras en el rendimiento del algoritmo, siempre y cuando se aproveche la tecnología de memoria caché, ya sea teniendo un precálculo suficientemente pequeño de forma tal que entre en una sola línea de la misma, o leyéndolo de forma inteligente para que no se vaya desalojando.