# Organización del computador II
# Trabajo Práctico II
# Grupo: Los Marrones

Rodrigo Campos Catelin (561/06)
Francisco Roca (188/06)

May 10, 2011

# Introducción

Este trabajo consiste en un análisis comparativo entre implementaciones en C y en ASM de varios algoritmos con el fin de determinar la conveniencia del procesamiento en paralelo que proveen las instrucciones del tipo SIMD.

Los algoritmos utilizados son algoritmos sobre imágenes. Se utilizaron dos algoritmos de conversión de imágenes a escala de grises y los filtros de detección de bordes de Roberts, Prewitt, Sobel y Frei-Chen.

Para la implementación en ASM se trabajó con la arquitectura IA-32 de Intel.

# Desarrollo

## Implementación en C

La implementación en C de todos los algoritmos es una traducción directa del algoritmo descripto en el enunciado del trabajo práctico. Lo único que consideramos relevante destacar es que en los filtros de detección de bordes que corresponde, despues de correr el filtro, el borde (es decir la primer y ultima fila y primer y ultima columna) de la imágen destino es reemplazado por el borde de la imágen original.

Además, la implementación en C no utiliza ninguna instrucción del tipo SIMD, por lo cual no se hace ningún tipo de procesamiento en paralelo de pixeles.

## Implementación en ASM

### Detección de bordes

Las imágenes en escala de grises que se toman por parámetro (tanto la imágen destino como la fuente) son arreglos de enteros de 8 bits sin signo, donde cada elemento representa un pixel. Aplicando los distintos filtros debemos escribir, en la imágen destino, la imágen fuente filtrada.

Los filtros consisten en modificar un pixel a partir de una mascara aplicada sobre los pixeles vecinos. Debido al tamaño de las máscaras (de 2x2 o 3x3), los coeficientes que utiliza cada filtro y que los registros SSE son de 16 bytes, podemos aplicar más de una máscara en una iteración.

En todos los filtros salvo uno de conversión a escala de grises, para calcular el resultado deseado en cada pixel sin perder correctitud, se decidió extender/convertir los elementos a enteros de 16bits (o a floats de 32bits en el caso del filtro Frei-Chen). Con 16 bits es suficiente ya que ningún filtro utiliza más de 6 sumas y dos multiplicaciones, y considerando que cada suma o multiplicación (como los coeficientes son 2 y $\sqrt{2}$) a lo sumo agrega un bit, esto resulta en peor caso en 16 bits, ya que los números con que se opera son de 8 bits.

### Filtro de Roberts

El filtro consiste en aplicar las siguientes máscaras:

$$\begin{array}{|c|c|}\hline 1 & 0 \\ \hline 0 & -1 \\ \hline \end{array} \text{ y } \begin{array}{|c|c|}\hline 0 & 1 \\ \hline -1 & 0 \\ \hline \end{array}$$

Centradas en el elemento $(1,1)$. Esto implica que no se puede procesar el filtro en la última fila y en la última columna.

En cada iteracion se cargan 8 elementos de cada fila, es decir vamos a poder aplicar el filtro en 7 elementos.

Siendo $i, j$ la fila y columna actual, cada iteración consiste en:

1. Cargar 8 bytes en la parte baja de XMM6 y 8 bytes de la fila debajo en la parte baja de XMM7
   $movq\, XMM6,\, [src_{i,j}]$

*movq* $XMM7$, $[src_{i+1,j}]$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ |

2. Como en la parte baja de XMM6 y XMM7 hay hay enteros de 8 bits sin signo, utilizamos la instrucion PUNPCKLBW para extener los elementos a números de 16 bits con signo
   *pxor* $XMM0$, $XMM0$
   *punpcklbw* $XMM6$, $XMM0$
   *punpcklbw* $XMM7$, $XMM0$

3. Mantener una copia de XMM6 y XMM7 en XMM0, XMM1, XMM2 y XMM3
   $XMM0 \leftarrow XMM6$
   $XMM2 \leftarrow XMM6$
   $XMM1 \leftarrow XMM7$
   $XMM3 \leftarrow XMM7$

4. Shift de 1 elemento en los registros en XMM1 y XMM2:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | 0 |
| $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ | 0 |

5. Calcular la siguiente resta de words empaquetadas (*psubw*):
   $XMM0 \leftarrow XMM0 - XMM1$ (máscara en $x$)
   $XMM2 \leftarrow XMM2 - XMM3$ (máscara en $y$)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $s_{i,j} - s_{i,j+1}$ | $s_{i,j+1} - s_{i,j+2}$ | $s_{i,j+2} - s_{i,j+3}$ | ... | ... | ... | ... | $s_{i,j+7} - 0$ |
| $s_{i+1,j+1} - s_{i+1,j}$ | $s_{i+1,j+2} - s_{i+1,j+1}$ | $s_{i+1,j+3} - s_{i+1,j+2}$ | ... | ... | ... | ... | $0 - s_{i+1,j+7}$ |

   Notar que el octavo elemento lo calculamos pero será reemplazado en la siguiente iteración, ya que no posee el valor deseado.

6. Calculamos el modulo de XMM0 y XMM2 de cada word empaquetada usando *pabsw*

7. Empaquetamos con saturación XMM0 y XMM2 a 8 bits sin signo con *packuswb*

8. Sumamos XMM0 y XMM2 con saturación de a bytes sin signo. Esto corresponde a la suma del modulo del filtro en y y el modulo del filtro en x
   *paddusb* $XMM0$, $XMM2$

| | | | |
|---|---|---|---|
| $\lvert s_{i,j} - s_{i,j+1}\rvert + \lvert s_{i+1,j+1} - s_{i+1,j}\rvert$ | $\lvert s_{i,j+1} - s_{i,j+2}\rvert + \lvert s_{i+1,j+2} - s_{i+1,j+1}\rvert$ | ... | ... |

9. Guardamos la parte baja de XMM0 (64 bits) en la imágen destino. Reiteramos que solo los primeros 7 bytes possen el valor correcto y el octavo byte será sobreescrito en la siguiente iteración

De esta forma se puede ver que en los primeros 7 bytes queda el valor de aplicar la máscara a los primeros 7 pixeles de la imágen, siendo el primer byte $s_{i,j}$

### Filtro de Prewitt

El filtro consiste en aplicar las siguientes máscaras:

$$\begin{array}{|c|c|c|}\hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline -1 & 0 & 1 \\ \hline \end{array} \text{ y } \begin{array}{|c|c|c|}\hline -1 & -1 & -1 \\ \hline 0 & 0 & 0 \\ \hline 1 & 1 & 1 \\ \hline \end{array}$$

Centradas en el elemento $(2, 2)$. Esto implica que no se puede procesar el filtro en ningún borde de la imágen.

En cada iteracion se cargan 8 elementos de cada fila, es decir vamos a poder generar 6 elementos de la imágen destino.

Siendo $i, j$ la fila y columna actual, cada iteración consiste en:

1. Cargar la fila anterior, actual y siguiente en XMM5, XMM6 y XMM7 respectivamente
   $movq\, XMM5,\ [src_{i-1,j}]$
   $movq\, XMM6,\ [src_{i,j}]$
   $movq\, XMM7,\ [src_{i+1,j}]$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j}$ | $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ |
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ |

2. Como en la parte baja de XMM5, XMM6 y XMM7 hay hay enteros de 8 bits sin signo, utilizamos la instrucion PUNPCKLBW para extener los elementos a números de 16 bits con signo
   $pxor\, XMM0, XMM0$
   $punpcklbw\, XMM5,\ XMM0$
   $punpcklbw\, XMM6,\ XMM0$
   $punpcklbw\, XMM7,\ XMM0$

3. Mantener una copia de memoria de XMM5, XMM6 y XMM7 para trabajar con copias en XMM0, XMM1 y XMM2 respectivamente
   $XMM0 \leftarrow XMM5$
   $XMM1 \leftarrow XMM6$
   $XMM2 \leftarrow XMM7$

4. Calculamos la suma empaquetada de a words de XMM0, XMM1 y XMM2:
   $paddw\, XMM0,\ XMM1$
   $paddw\, XMM0,\ XMM2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $s_{i-1,j} + s_{i,j} + s_{i+1,j}$ | $s_{i-1,j+1} + s_{i,j+1} + src_{i+1,j+1}$ | ... | ... | ... | ... | ... | $s_{i-1,j+7} + s_{i,j+7} + src_{i+1,j+7}$ |

   (máscara en $y$ sin módulo)

5. Copio XMM0 a XMM1

6. Shifteo 2 elementos XMM1 sacando las 2 de posiciones menos significativas

7. Resta empaquetada de a words de XMM1 y XMM0 (máscara en $x$)
   $XMM1 \leftarrow XMM1 - XMM0$

| | | | |
|---|---|---|---|
| $(s_{i-1,j+2} + s_{i,j+2} + s_{i+1,j+2}) - (s_{i-1,j} + s_{i,j} + s_{i+1,j})$ | ... | ... | ... |

8. Copiamos XMM1 a XMM2

| | | | |
|---|---|---|---|
| $(s_{i-1,j+2} + s_{i,j+2} + s_{i+1,j+2}) - (s_{i-1,j} + s_{i,j} + s_{i+1,j})$ | ... | ... | ... |

9. Resta empaquetada de a words de XMM7 y XMM5
   $XMM7 \leftarrow XMM7 - XMM5$

| | | | | |
|---|---|---|---|---|
| $src_{i+1,j} - src_{i-1,j}$ | $src_{i+1,j+1} - src_{i-1,j+1}$ | $src_{i+1,j+2} - src_{i-1,j+2}$ | $src_{i+1,j+3} - src_{i-1,j+3}$ | ... |

10. Copio XMM7 a XMM0 y XMM1

11. Shifteo 1 elemento XMM0, desalojando el elemento de la posición menos significativa

12. Shifteo 2 elementos XMM1, desalojando los 2 elementos de las posiciones menos significativa

13. Suma empaquetada de a words de XMM0, XMM1 y XMM7
    $paddw\, XMM0,\ XMM7$
    $paddw\, XMM0,\ XMM1$

| | | |
|---|---|---|
| $(s_{i+1,j} - s_{i-1,j}) + (s_{i+1,j+1} - s_{i-1,j+1}) + (s_{i+1,j+2} - s_{i-1,j+2})$ | ... | ... |

14. Tomar el modulo empaquetado de a words de XMM0

| | | | |
|---|---|---|---|
| $\lvert(s_{i+1,j} - s_{i-1,j}) + (s_{i+1,j+1} - s_{i-1,j+1}) + (s_{i+1,j+2} - s_{i-1,j+2})\rvert$ | ... | ... | (máscara en $y$) |

15. Tomar el modulo empaquetado de a words de XMM2

| | | | |
|---|---|---|---|
| $\|(s_{i-1,j+2} + s_{i,j+2} + s_{i+1,j+2}) - (s_{i-1,j} + s_{i,j} + s_{i+1,j})\|$ | ... | ... | ... |

(máscara en $x$)

16. Empaquetar XMM0 y XMM2 a enteros de 8 bits sin signo
$packuswb\, XMM0,\, XMM0$
$packuswb\, XMM2,\, XMM2$

17. Suma saturada empaquetada de a byte de XMM0 y XMM2
$paddusb\, XMM0,\, XMM2$

18. Guardamos la parte baja de XMM0 (64 bits) en la imágen destino. Solo los primeros 6 bytes possen el valor correcto y el séptimo y octavo byte serán sobreescritos en la siguiente iteración

De esta forma, se puede ver que las máscaras en $x$ e $y$ en cada elemento se corresponden con el valor teórico de las máscaras de $s_{i,j+1}$ y sus 5 bytes siguientes.

### Filtro de Sobel

Este filtro es similar al de Prewitt, con la diferencia de en algunos de los coeficientes de las máscaras:

| | | |
|---|---|---|
| -1 | 0 | 1 |
| -2 | 0 | 2 |
| -1 | 0 | 1 |

y

| | | |
|---|---|---|
| -1 | -2 | -1 |
| 0 | 0 | 0 |
| 1 | 2 | 1 |

El procedimiento para calcular este filtro es casi idéntico. La primer diferencia es en el cálculo de la máscara en $y$. En este procedimiento, el paso 4 se ve levemente afectado quedando:

1. Calculamos la suma empaquetada de a words de XMM0, XMM1 y XMM2:
$paddw\, XMM0,\, XMM1$
$paddw\, XMM0,\, XMM1$
$paddw\, XMM0,\, XMM2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $s_{i-1,j} + 2s_{i,j} + s_{i+1,j}$ | $s_{i-1,j+1} + 2s_{i,j+1} + src_{i+1,j+1}$ | ... | ... | ... | ... | ... | $s_{i-1,j+7} + 2s_{i,j+7} + src_{i+1,j+7}$ |

(máscara en $y$ sin módulo)

La segunda diferencia con es en el cálculo de la máscara en $x$. El paso 13 se modifica de la forma:

1. Suma empaquetada de a words de XMM0, XMM1 y XMM7
$paddw\, XMM0,\, XMM7$
$paddw\, XMM0,\, XMM7$
$paddw\, XMM0,\, XMM1$

| | | |
|---|---|---|
| $(s_{i+1,j} - s_{i-1,j}) + 2(s_{i+1,j+1} - s_{i-1,j+1}) + (s_{i+1,j+2} - s_{i-1,j+2})$ | ... | ... |

De esta forma se puede ver que si se continúa con el procedimiento utilizado en el filtro de Prewitt, se obtiene el resultado deseado.

### Filtro de Frei-Chen

Las máscaras para son:

| | | |
|---|---|---|
| -1 | 0 | 1 |
| $-\sqrt{2}$ | 0 | $\sqrt{2}$ |
| -1 | 0 | 1 |

y

| | | |
|---|---|---|
| -1 | $-\sqrt{2}$ | -1 |
| 0 | 0 | 0 |
| 1 | $\sqrt{2}$ | 1 |

Este filtro es similar a los dos anteriores con la diferencia de los coeficientes de la $2^{da}$ fila y $2^{da}$columna el coeficiente es $-\sqrt{2}$. Esto implica que hay que transformar algunos elementos a floats para multiplicarlos. Se eligió convertir los enteros a floats de presición simple (32bits). Luego de hacer la multiplicación se vuelven a convertir los datos a su estado anterior.

Para la mascara $x$ se hace un calculo similar al de los filtros anteriores, salvo porque convertimos a floats una de las filas, pero para la máscara $y$ cambiamos la forma de calcularlo. Considerando que cargamos la 1era y 3ra fila en registros y hacemos shifts para hacer los calculos en paralelo, hacemos el siguiente calculo:

$filtro_y = fila3 + fila3_{shift2} - fila1 - fila1_{shift2} + \left[(fila3_{shift1} - fila1_{shift1}) * \sqrt{2}\right]$

El procedimiento de cada iteración es el siguiente:

1. Cargar la fila anterior, actual y siguiente en XMM5, XMM6 y XMM7 respectivamente
   $movq\, XMM5,\ [src_{i-1,j}]$
   $movq\, XMM6,\ [src_{i,j}]$
   $movq\, XMM7,\ [src_{i+1,j}]$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $src_{i-1,j}$ | $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ | ... |
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | ... |
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ | ... |

2. Como en la parte baja de XMM5, XMM6 y XMM7 hay hay enteros de 8 bits sin signo, utilizamos la instrucion PUNPCKLBW para extener los elementos a números de 16 bits con signo
   $pxor\, XMM0, XMM0$
   $punpcklbw\, XMM5,\ XMM0$
   $punpcklbw\, XMM6,\ XMM0$
   $punpcklbw\, XMM7,\ XMM0$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j}$ | $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ |
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ |

3. Para empezar con la mascara en $x$ hay obtener el producto de la segunda fila (alojada en XMM6) con $\sqrt{2}$. Para esto expandimos cada entero de 16bits con signo a enteros de 32bits y luego lo convertimos a float. Despues sacamos el producto y reconvertimos a enteros de 16 bits para hacer el calculo.

   (a) Hacemos dos copias ya que en un registro no entra ($8 * 32 = 256$):
   $movdqa\, xmm1,\ xmm6$
   $movdqa\, xmm2,\ xmm6$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |

   (b) Extendemos con 0s la parte baja de la segunda fila en XMM1 y la parte alta en XMM2 ya que son numeros positivos, ahora son 4 enteros de 32 bits en cada uno:
   $punpcklwd\, xmm1,\ xmm0$
   $punpckhwd\, xmm2,\ xmm0$

| | | | |
|---|---|---|---|
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ |
| $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |

   (c) Cargamos en XMM0 el valor 2 en forma entera de 32bits empaquetada
   $movdqu\, xmm0,\ [dos32b]$

| | | | |
|---|---|---|---|
| 2 | 2 | 2 | 2 |

   (d) Convertimos a float los 3 registros, dos donde esta la segunda fila y uno con 2 empaquetado.
   $CVTDQ2PS\, xmm1,\ xmm1$
   $CVTDQ2PS\, xmm2,\ xmm2$
   $CVTDQ2PS\, xmm0,\ xmm0$

   (e) Sacamos la raiz cuadrada empaquetada de presición simple
   $sqrtps\, xmm0,\ xmm0$

| | | | |
|---|---|---|---|
| $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ |

(f) Multiplicamos por $\sqrt{2}$
$mulps\, xmm1,\ xmm0$
$mulps\, xmm2,\ xmm0$

| | | | |
|---|---|---|---|
| $src_{i,j} * \sqrt{2}$ | $src_{i,j+1} * \sqrt{2}$ | $src_{i,j+2} * \sqrt{2}$ | $src_{i,j+3} * \sqrt{2}$ |
| $src_{i,j+4} * \sqrt{2}$ | $src_{i,j+5} * \sqrt{2}$ | $src_{i,j+6} * \sqrt{2}$ | $src_{i,j+7} * \sqrt{2}$ |

(g) Reconvertimos a enteros de 32bits
$cvtps2dq\, xmm1,\ xmm1$
$cvtps2dq\, xmm2,\ xmm2$

| | | | |
|---|---|---|---|
| $[src_{i,j} * \sqrt{2}]$ | $[src_{i,j+1} * \sqrt{2}]$ | $[src_{i,j+2} * \sqrt{2}]$ | $[src_{i,j+3} * \sqrt{2}]$ |
| $[src_{i,j+4} * \sqrt{2}]$ | $[src_{i,j+5} * \sqrt{2}]$ | $[src_{i,j+6} * \sqrt{2}]$ | $[src_{i,j+7} * \sqrt{2}]$ |

(h) Empaquetamos a 16bits con saturacion con signo.
$packssdw\, xmm1,\ xmm2$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $[src_{i,j} * \sqrt{2}]$ | $[src_{i,j+1} * \sqrt{2}]$ | . | . | . | . | . | $[src_{i,j+7} * \sqrt{2}]$ |

4. Sumo las 3 filas empaquetadas. Luego hago una copia de la suma, la shifteo dos lugares y resto. Esto deja como resultado la mascara de $x$ en XMM5.
$paddw\, xmm5,\ xmm1$
XMM5 =

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j} + [src_{i,j} * \sqrt{2}]$ | $src_{i-1,j+1} + [src_{i,j+1} * \sqrt{2}]$ | . | . | . | . | . | ... |

$paddw\, xmm5,\ xmm7$
XMM5 =

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j} + [src_{i,j} * \sqrt{2}] + src_{i+1,j}$ | $src_{i-1,j+1} + [src_{i,j+1} * \sqrt{2}] + src_{i+1,j+1}$ | . | . | . | . | . | ... |

$movdqa\, xmm1,\ xmm5$
$psrldq\, xmm1,\ 4$
$psubw\, xmm1,\ xmm5$

| | | | | | | |
|---|---|---|---|---|---|---|
| $-[src_{i-1,j} + [src_{i,j} * \sqrt{2}] + src_{i+1,j}] + src_{i-1,j+2} + [src_{i,j+2} * \sqrt{2}] + src_{i+1,j+2}$ | . | . | . | . | . | ... |

5. Comienzo a procesar la mascara $y$. Cargo la 1era fila y la 3ra en XMM6 y XMM7 y lo extiendo a enteros de 16bits.
$movq\, xmm6,\ [ebx]$
$movq\, xmm7,\ [esi + edx]$
$pxor\, xmm0,\ xmm0$
$punpcklbw\, xmm6,\ xmm0$
$punpcklbw\, xmm7,\ xmm0$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j}$ | $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ |
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ |

6. Comienzo a hacer el calculo de $filtro_y$ descripto arriba.

(a) Cargo $fila1$ en XMM1 y $fila1_{shift2}$ en XMM2
$movdqa\, xmm1,\ xmm6$
$movdqa\, xmm2,\ xmm6$
$psrldq\, xmm2,\ 4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i-1,j}$ | $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ |
| $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ | $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ | 0 | 0 |

(b) Cargo $fila3$ en XMM3 y $fila3_{shift2}$ en XMM4
$movdqa\, xmm3,\ xmm7$
$movdqa\, xmm4,\ xmm7$
$psrldq\, xmm4,\ 4$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i+1,j}$ | $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ |
| $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ | $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ | 0 | 0 |

(c) Hago $fila3 + fila3_{shift2} - fila1 - fila1_{shift2}$ y lo guardo en XMM1
$paddw\, xmm3,\ xmm4$

$psubw\, xmm3,\, xmm1$
$psubw\, xmm3,\, xmm2$
$movdqa\, xmm1,\, xmm3$

XMM1=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i+1,j} + src_{i+1,j+2} - src_{i-1,j} - src_{i-1,j+2}$ | . | . | . | . | . | .. | .. |

7. Hago el calculo de $[(fila3_{shift1} - fila1_{shift1}) * \sqrt{2}]$

   (a) Extiendo las filas en dos registros para tener enteros de 32 bits, preparados para convertirlos a floats de presición simple.
   $movdqa\, xmm2,\, xmm6$
   $psrldq\, xmm2,\, 2$
   $movdqa\, xmm6,\, xmm2$
   $punpcklwd\, xmm2,\, xmm0$ (xmm2 = parte baja de $fila1_{shift1}$)
   $punpckhwd\, xmm6,\, xmm0$ (xmm6 = parte alta de $fila1_{shift1}$)
   $movdqa\, xmm3,\, xmm7$
   $psrldq\, xmm3,\, 2$
   $movdqa\, xmm7,\, xmm3$
   $punpcklwd\, xmm3,\, xmm0$ (xmm3 = parte baja de $fila3_{shift1}$)
   $punpckhwd\, xmm7,\, xmm0$ (xmm7 = parte alta de $fila3_{shift1}$)

| | | | |
|---|---|---|---|
| $src_{i-1,j+1}$ | $src_{i-1,j+2}$ | $src_{i-1,j+3}$ | $src_{i-1,j+4}$ |
| $src_{i-1,j+5}$ | $src_{i-1,j+6}$ | $src_{i-1,j+7}$ | 0 |
| $src_{i+1,j+1}$ | $src_{i+1,j+2}$ | $src_{i+1,j+3}$ | $src_{i+1,j+4}$ |
| $src_{i+1,j+5}$ | $src_{i+1,j+6}$ | $src_{i+1,j+7}$ | 0 |

   (b) Genero un registro empaquetado con $\sqrt{2}$ y convierto XMM2, XMM6, XMM3, XMM7 a floats empaquetados de presición simple
   $movdquxmm0, [dos32b]$
   $CVTDQ2PS\, xmm0,\, xmm0$
   $sqrtps\, xmm0,\, xmm0$

| | | | |
|---|---|---|---|
| $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ | $\sqrt{2}$ |

   $CVTDQ2PS\, xmm2,\, xmm2$
   $CVTDQ2PS\, xmm6,\, xmm6$
   $CVTDQ2PS\, xmm3,\, xmm3$
   $CVTDQ2PS\, xmm7,\, xmm7$

   (c) Hago las multiplicaciones y la resta del calculo. En XMM3 y XMM7 me queda el resultado (parte alta y baja)
   $mulps\, xmm2,\, xmm0$
   $mulps\, xmm6,\, xmm0$
   $mulps\, xmm3,\, xmm0$
   $mulps\, xmm7,\, xmm0$

| | | | |
|---|---|---|---|
| $src_{i-1,j+1} * \sqrt{2}$ | $src_{i-1,j+2} * \sqrt{2}$ | $src_{i-1,j+3} * \sqrt{2}$ | $src_{i-1,j+4} * \sqrt{2}$ |
| $src_{i-1,j+5} * \sqrt{2}$ | $src_{i-1,j+6} * \sqrt{2}$ | $src_{i-1,j+7} * \sqrt{2}$ | 0 |
| $src_{i+1,j+1} * \sqrt{2}$ | $src_{i+1,j+2} * \sqrt{2}$ | $src_{i+1,j+3} * \sqrt{2}$ | $src_{i+1,j+4} * \sqrt{2}$ |
| $src_{i+1,j+5} * \sqrt{2}$ | $src_{i+1,j+6} * \sqrt{2}$ | $src_{i+1,j+7} * \sqrt{2}$ | 0 |

   $subps\, xmm3,\, xmm2$
   $subps\, xmm7,\, xmm6$

| | | | |
|---|---|---|---|
| $src_{i+1,j+1} * \sqrt{2} - src_{i-1,j+1} * \sqrt{2}$ | $src_{i+1,j+2} * \sqrt{2} - src_{i-1,j+2} * \sqrt{2}$ | .. | .. |
| $src_{i+1,j+5} * \sqrt{2} - src_{i-1,j+5} * \sqrt{2}$ | $src_{i+1,j+6} * \sqrt{2} - src_{i-1,j+6} * \sqrt{2}$ | .. | 0 |

   (d) Convierto el resultado del calculo a enteros de 32 bits y luego lo empaqueto a enteros de 16bits. En XMM3 queda el resultado empaquetado en enteros de 16 bits.
   $cvtps2dq\, xmm3,\, xmm3$

$cvtps2dq\, xmm7,\ xmm7$
$packssdw\, xmm3,\ xmm7$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\left[src_{i+1,j+1} * \sqrt{2} - src_{i-1,j+1} * \sqrt{2}\right]$ | . | . | . | $\left[src_{i+1,j+5} * \sqrt{2} - src_{i-1,j+5} * \sqrt{2}\right]$ | . | . | 0 |

8. Hago la suma final para obtener $filtro_y = fila3 + fila3_{shift2} - fila1 - fila1_{shift2} + \left[(fila3_{shift1} - fila1_{shift1}) * \sqrt{2}\right]$ en XMM1
$paddw\, xmm1,\ xmm3$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i+1,j} + src_{i+1,j+2} - src_{i-1,j} - src_{i-1,j+2} - \left[src_{i+1,j+1} * \sqrt{2} - src_{i-1,j+1} * \sqrt{2}\right]$ | . | . | . | . | . | .. | .. |

9. Obtengo el modulo de ambas máscaras, $filtro_x$en XMM5 y $filtro_y$en XMM1
$pabsw\, xmm1,\ xmm1$
$pabsw\, xmm5,\ xmm5$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $\text{abs}(src_{i+1,j} + src_{i+1,j+2} - src_{i-1,j} - src_{i-1,j+2} - \left[src_{i+1,j+1} * \sqrt{2} - src_{i-1,j+1} * \sqrt{2}\right])$ | . | . | . | . | . | . | . |
| $\text{abs}(- \left[src_{i-1,j} + \left[src_{i,j} * \sqrt{2}\right] + src_{i+1,j}\right] + src_{i-1,j+2} + \left[src_{i,j+2} * \sqrt{2}\right] + src_{i+1,j+2})$ | . | . | . | . | . | . | . |

10. Empaqueto a enteros de 8 bits sin signo con saturación (ya que son positivos luego del modulo en el paso anterior).
$paddusb\, xmm1,\ xmm5$
En XMM1 tengo el resultado final del operador aplicado a 6 elementos.

### Conversión a escala de grises

A diferencia de los filtros de detección de bordes, al convertir una imágen en escala de grises se debe recorrer la matriz entera, incluyendo los bordes. Cada pixel de la imágen fuente se corresponde con 3 bytes utilizados para representar la intensidad de rojo, verde y azúl de ese pixel. Es decir, la imágen fuente se puede ver como un arreglo de elementos de 24 bits sin signo.

Como función de monocromatización se utilizaron dos casos particulares de la siguiente función que toma un pixel $p$ y donde $R$, $G$ y $B$ son sus componentes de intensidad roja, verde y azul:

$$f(p) = \sqrt[\epsilon]{\alpha R^\epsilon + \beta G^\epsilon + \gamma B^\epsilon}$$

Es decir, si aplicamos la función a un pixel con componentes $R$, $G$ y $B$ obtenemos el valor correspondiente a su representación en escala de grises. Si aplicamos la función a todos los pixeles de una imágen y generamos una imágen del mismo tamaño que la original, donde cada pixel es el resultado de aplicarle la función al pixel de esa posición en la imágen original, obtenemos la conversión de la imágen original a escala de grises.

Los casos particulares utilizados son por un lado tomando $\alpha = \frac{1}{4}$, $\beta = \frac{1}{2}$, $\gamma = \frac{1}{4}$ y $\epsilon = 1$ y por otro lado, esos mismos parámetros pero cuando $\epsilon$ tiende a infinito.

### Conversión a escala de grises con $\epsilon = 1$

La función de monocromatización cuando fijamos los parámetros $\alpha = \frac{1}{4}$, $\beta = \frac{1}{2}$, $\gamma = \frac{1}{4}$ y $\epsilon = 1$ resulta:

$$f_{uno}(p) = \frac{R + 2G + B}{4}$$

Al ser $R$, $G$ y $B$ números de 8 bits, el numerador no podrá ser mayor que $255 + 2 \times 255 + 255 = 1020$ por lo que la fracción no podrá ser mayor que $1020/4 = 255$. Es decir, si se trabaja con 16 bits el numerador nunca podrá ocasionar overflow y el resultado siempre entrará en un entero de 8 bits sin signo.

El algoritmo utilizado para su cálculo es el siguiente:

1. Cargar 8 bytes en la parte baja de XMM6
$movq\, XMM6,\ [src_{i,j}]$

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ |

2. Como en la parte baja de XMM6 hay enteros de 8 bits sin signo, utilizamos la instrucion PUNPCKLBW para extener los elementos a números de 16 bits con signo
   $pxor\, XMM0, XMM0$
   $punpcklbw\, XMM6, XMM0$

3. Mantener una copia de XMM6 en XMM0, XMM1 y XMM2
   $XMM0 \leftarrow XMM6$
   $XMM1 \leftarrow XMM6$
   $XMM2 \leftarrow XMM6$

4. Shift de 1 elemento en XMM1 y de 2 elementos en XMM2:
   $psrldq\, xmm1$, 2
   $psrldq\, xmm2$, 4

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | 0 |
| $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | 0 | 0 |

5. Suma empaquetada de a words de XMM0 y XMM1

| | | | | | |
|---|---|---|---|---|---|
| $src_{i,j} + src_{i,j+1}$ | $src_{i,j+1} + src_{i,j+2}$ | $src_{i,j+2} + src_{i,j+3}$ | $src_{i,j+3} + src_{i,j+4}$ | $src_{i,j+4} + src_{i,j+5}$ | ... |

6. Suma empaquetada de a words de XMM0 y XMM1

| | | | | | |
|---|---|---|---|---|---|
| $src_{i,j} + 2src_{i,j+1}$ | $src_{i,j+1} + 2src_{i,j+2}$ | $src_{i,j+2} + 2src_{i,j+3}$ | $src_{i,j+3} + 2src_{i,j+4}$ | $src_{i,j+4} + 2src_{i,j+5}$ | ... |

7. Suma empaquetada de a words de XMM0 y XMM2

| | | | | |
|---|---|---|---|---|
| $src_{i,j} + 2src_{i,j+1} + src_{i,j+2}$ | $src_{i,j+1} + 2src_{i,j+2} + src_{i,j+3}$ | ... | $src_{i,j+3} + 2src_{i,j+4} + src_{i,j+5}$ | ... |

8. Pongo el cuarto elemento como segundo y mantengo el primero en el primer lugar
   $pshuflw\, xmm0$, $xmm0$, $0x30$

| | | | | |
|---|---|---|---|---|
| $src_{i,j} + 2src_{i,j+1} + src_{i,j+2}$ | $src_{i,j+3} + 2src_{i,j+4} + src_{i,j+5}$ | ... | ... | ... |

9. Empaqueto XMM0 como enteros de 8 bits sin signo
   $packuswb\, xmm0$, $xmm0$

10. Escribo los primeros 16 bytes de XMM0 en la imágen destino
    $movd\, edx$, $xmm0$
    $mov\, [dst]$, $dx$

De esta forma, se puede ver que el valor calculado en las primeros dos elementos de XMM0 se corresponde con el valor de aplicar la función a los pixeles $src_{i,j}$ y el siguiente.

### Conversión a escala de grises con $\epsilon = \infty$

La función de monocromatización cuando fijamos los parámetros $\alpha = \frac{1}{4}$, $\beta = \frac{1}{2}$, $\gamma = \frac{1}{4}$ y $\epsilon$ tendiendo a infinito, resulta:

$$f_\infty(p) = max(R, G, B)$$

Como la función no implica hacer ninguna cuenta con los números, solo fijarse el mayor, no los extenderemos a 16 bits. Cargaremos 16 bytes y generaremos 5 pixeles de la imágen destino en cada iteración.

Una iteración del algoritmo utilizado es el siguiente:

1. Cargar 16 bytes en XMM6
   $movdqu\, XMM6$, $[src_{i,j}]$

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $src_{i,j}$ | $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | ... |

2. Mantener una copia de XMM6 en XMM0, XMM1 y XMM2
   $XMM0 \leftarrow XMM6$
   $XMM1 \leftarrow XMM6$
   $XMM2 \leftarrow XMM6$

3. Shift de 1 elemento en XMM1 y de 2 elementos en XMM2:
   $psrldq\, xmm1, 1$
   $psrldq\, xmm2, 2$

| | | | | | | |
|---|---|---|---|---|---|---|
| $src_{i,j+1}$ | $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | ... |
| $src_{i,j+2}$ | $src_{i,j+3}$ | $src_{i,j+4}$ | $src_{i,j+5}$ | $src_{i,j+6}$ | $src_{i,j+7}$ | ... |

4. Calcular el maximo empaquetado de a bytes de XMM0 y XMM1

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $max(s_{i,j}, s_{i,j+1})$ | ... | ... | $max(s_{i,j+3}, s_{i,j+4})$ | ... | ... | $max(s_{i,j+6}, s_{i,j+7})$ | ... |

5. Calcular el maximo empaquetado de a bytes de XMM0 y XMM2

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| $max(s_{i,j}, s_{i,j+1}, s_{i,j+2})$ | ... | ... | $max(s_{i,j+3}, s_{i,j+4}, s_{i,j+5})$ | ... | ... | $max(s_{i,j+6}, s_{i,j+7}, s_{i,j+8})$ | ... |

6. Poner los bytes de las posiciones 3k (empezando de 0) en los primeros 5 lugares
   $pshufb\, xmm0, xmm3$ con XMM3 apropiado

| | | | |
|---|---|---|---|
| $max(s_{i,j}, s_{i,j+1}, s_{i,j+2})$ | $max(s_{i,j+3}, s_{i,j+4}, s_{i,j+5})$ | $max(s_{i,j+6}, s_{i,j+7}, s_{i,j+8})$ | ... |

7. Escribir los primeros 5 bytes en la imágen destino apuntada por $dst$
   $movd\, edx, xmm0$
   $mov\, [dst], edx$
   $psrldq\, xmm0, 4$
   $movd\, edx, xmm0$
   $mov\, [dst + 4], dl$

De esta forma, se puede ver que el valor calculado en las primeros 5 elementos de XMM0 se corresponde con el valor de aplicar la función a los pixeles $s_{i,j}$ y los 4 siguientes.

# Resultados

Para comparar las implementaciones en ASM y en C se usó la cantidad de ciclos de clock que consume cada una. Pero como no se cuenta con una forma precisa de saber esto, ya que realmente lo que se calcula es los ciclos de clock que transcurrieron desde que se inició la función hasta que terminó (contando también los ciclos que el sistema operativo le puede haber dado a otro proceso), se decidió correr las pruebas con la computadora mayormente en idle y varias veces cada una, utilizando como valor final el promedo de éstas.

Los gráficos a continuación muestran la cantidad de ciclos insumidos por cada implementación en función del tamño de la imágen. Como imágenes de entrada se decidió usar la imágen edificio provista por la cátedra en resoluciones cuadradas.

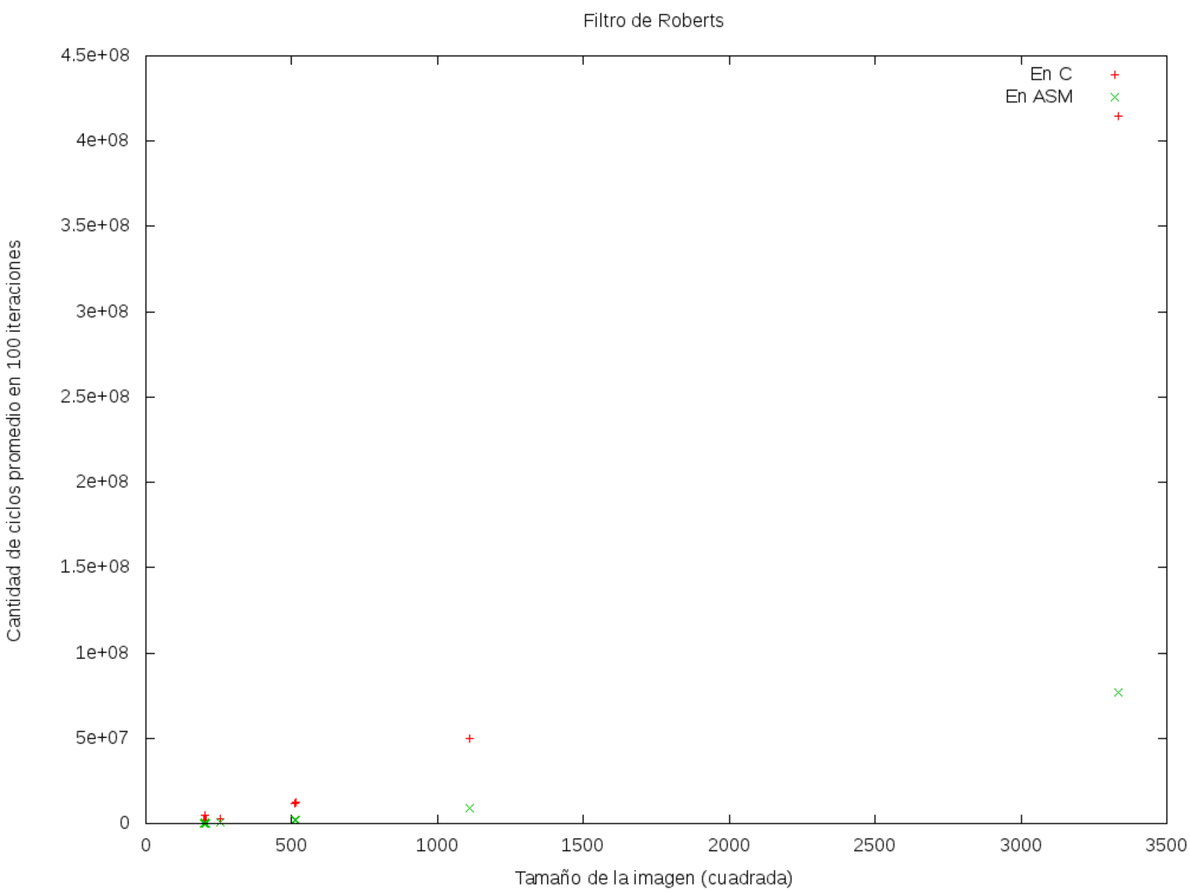


Figure 1: Comparativa C vs ASM, filtro de Roberts

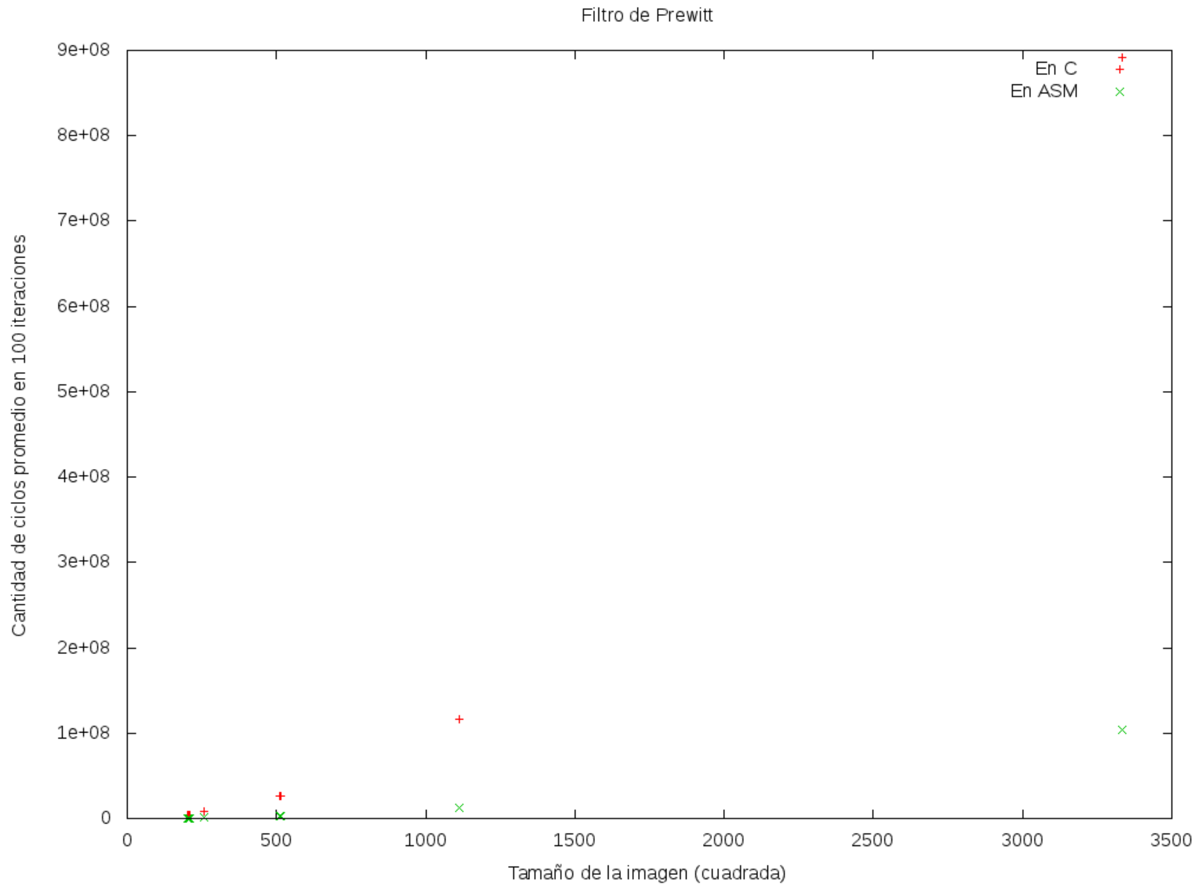


Figure 2: Comparativa C vs ASM, filtro de Prewitt

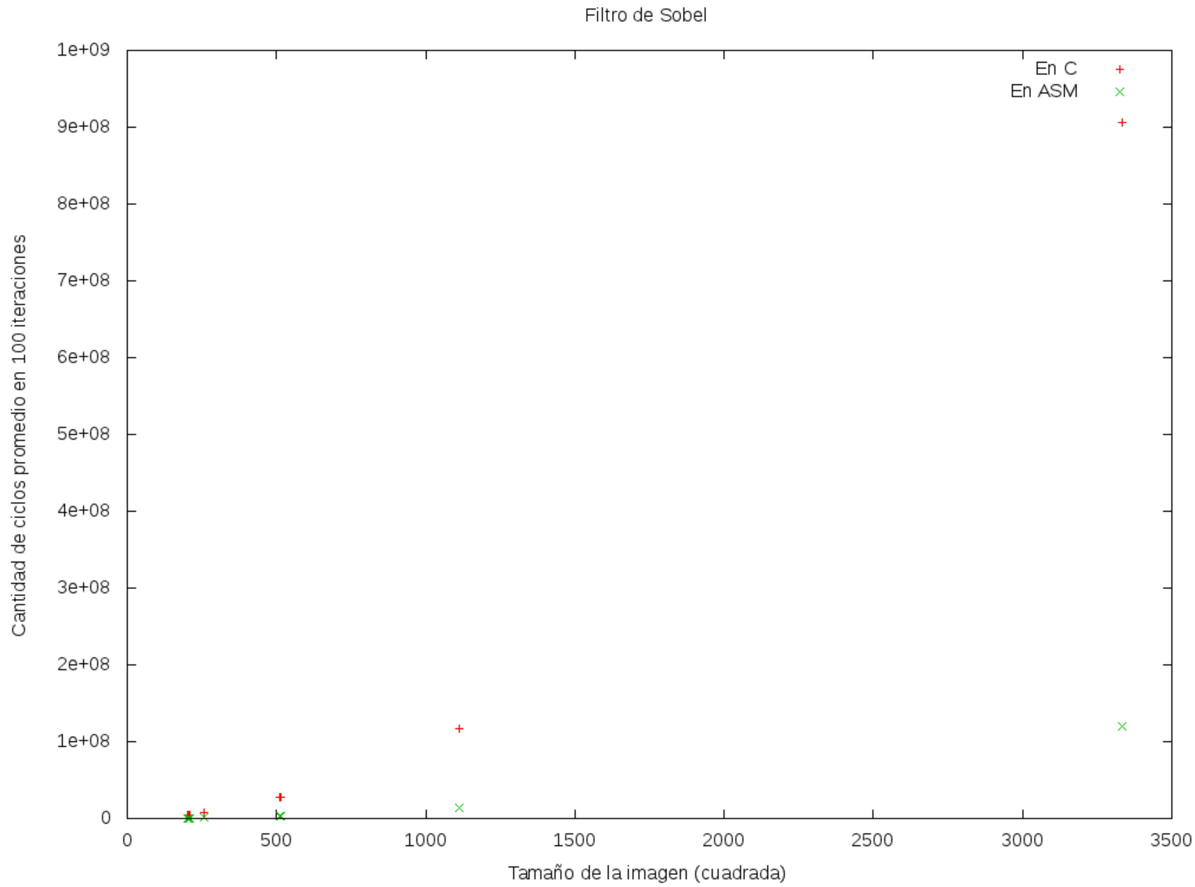


Figure 3: Comparativa C vs ASM, filtro de Sobel

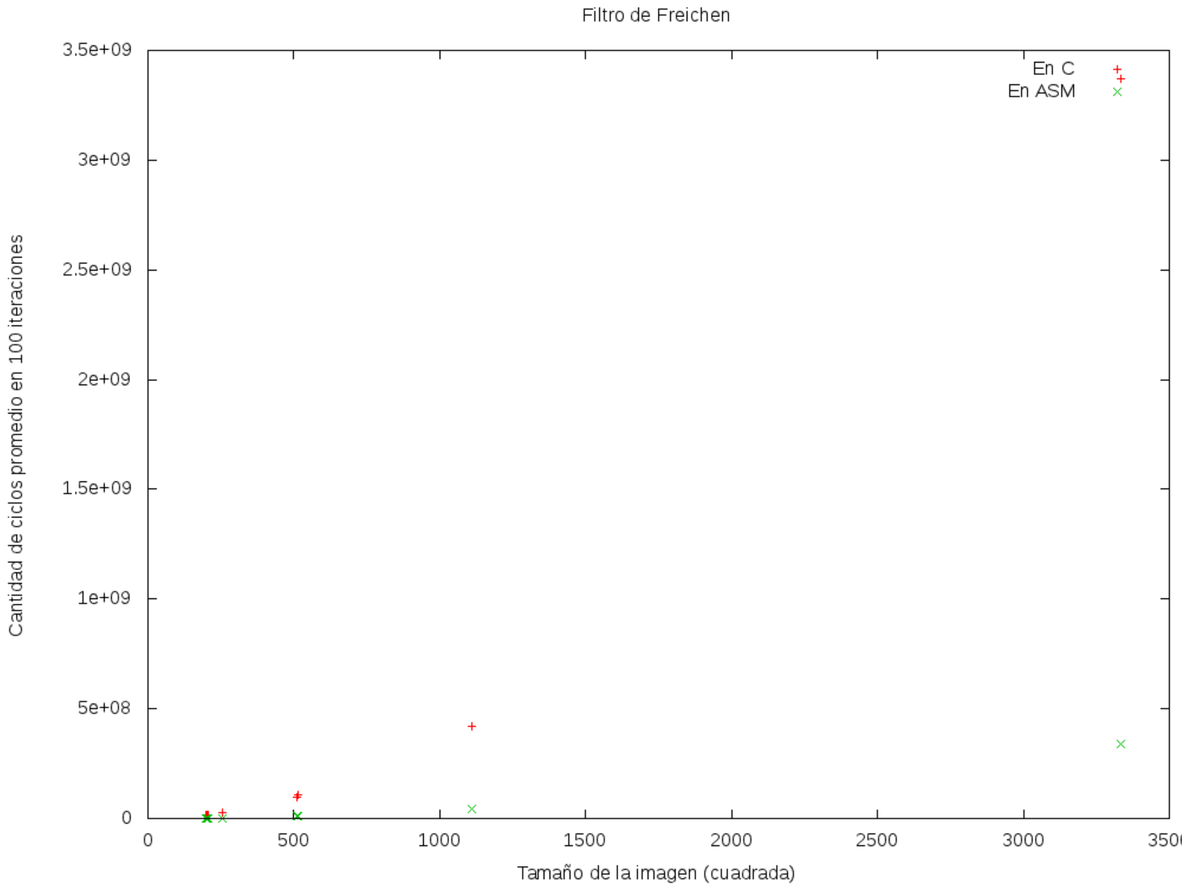


Figure 4: Comparativa C vs ASM, filtro de Frei-Chen

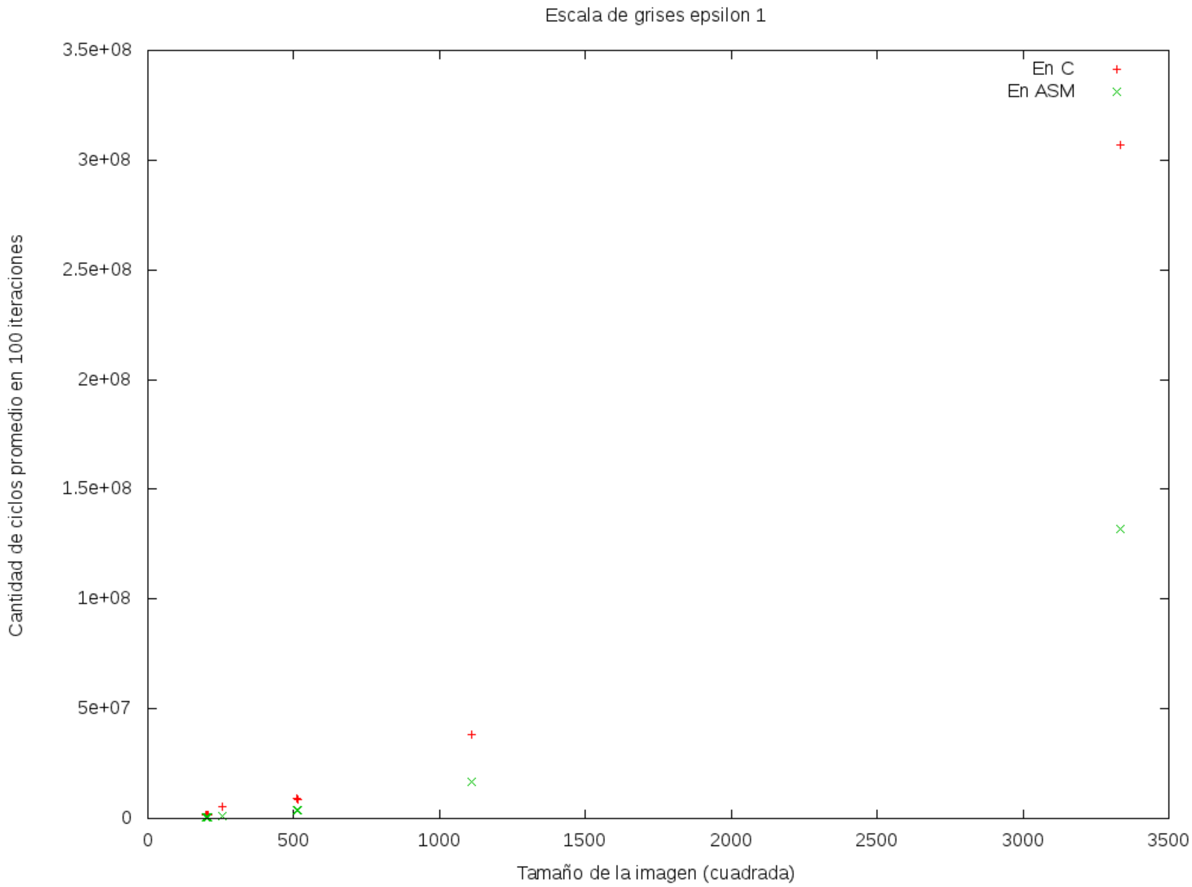


Figure 5: Comparativa C vs ASM, epsilon uno

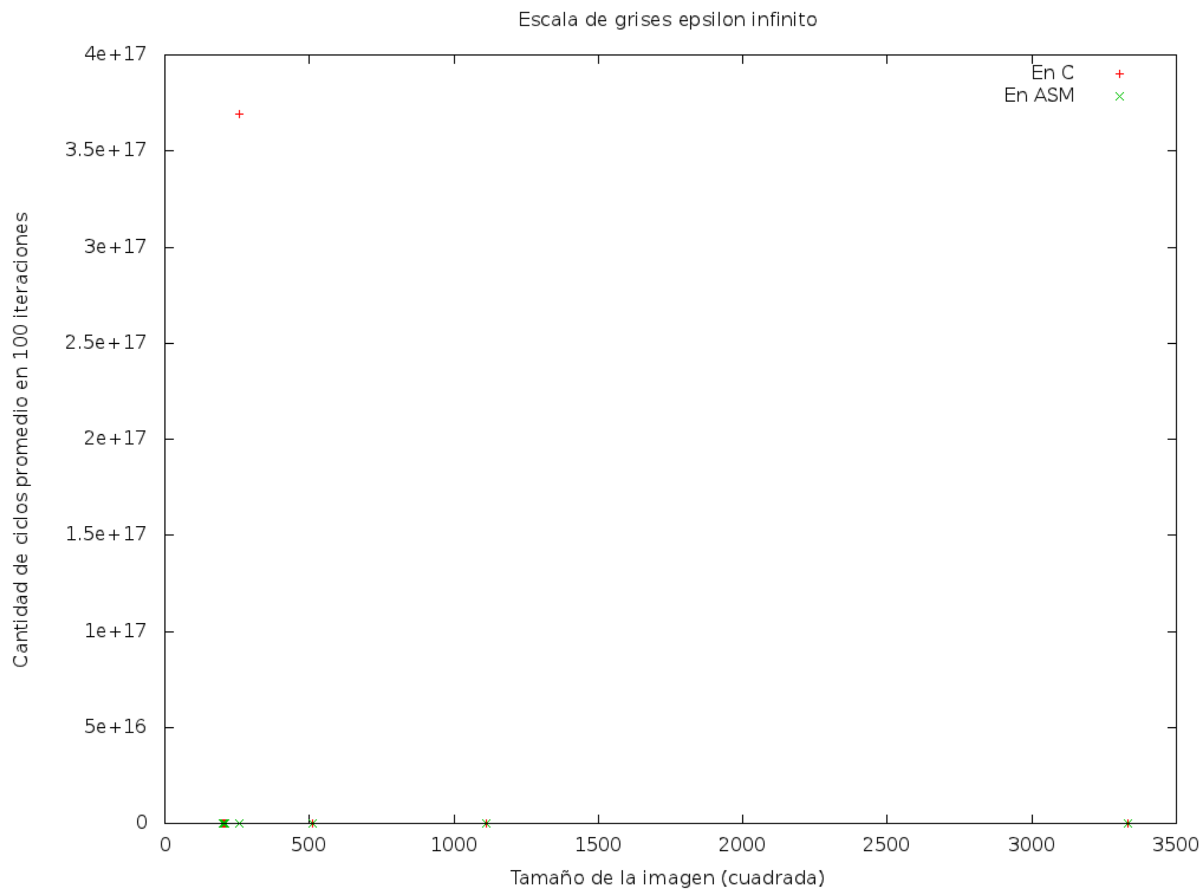


Figure 6: Comparativa C vs ASM, epsilon infinito

En todos los gráficos se puede ver la notable diferencia entre las implementaciones de C y ASM. En particular, cuanto más grande es el tamaño de la imágen, más notoria es esta diferencia. De hecho, la diferencia es tal que en todos los filtros insume una cantidad de ciclos similar aplicarlo sobre una imágen de más de 3000 x 3000 en ASM que aplicarlo en una de 1000x1000 en C. Sin embargo, esto no ocurre con la conversión a escala de grises con $\epsilon = 1$: si bien lleva una cantidad no despreciable de ciclos menos procesar una imágen de 3000x3000 en ASM que en C, ésta no es comparable con la cantidad de ciclos que lleva procesar una de 1000x1000 en C. Esto, creemos, se debe a que esa función de ASM genera solo dos pixeles de la imágen destino en cada iteración. En cambio, los filtros, generan al menos 6 pixeles de la imágen destino en cada iteración.

La función que menos pixeles de la imágen destino genera en una iteración es la función de conversión a escala de grises con $\epsilon = 1$, que genera 2 pixeles por iteración. Sin embargo, es notable la diferencia en la cantidad de ciclos de clock con su implementación en C, que genera un pixel de la imágen destino en cada iteración. Nosotros dudábamos si en este caso se iba a producir una diferencia tan notable, pero evidentemente sí.

También se notó que en el caso del filtro de Frei-Chen utilizando la implementación de ASM se llegó a consumir hasta 10 veces menos en cantidad de ciclos de clock.

# Conclusiones

El uso de las instrucciones SSE realmente provee una forma de mejorar varias veces la performance de un algoritmo, y reduciendo hasta 10 veces la cantidad de ciclos de clock según nuestros ejemplos. Esta diferencia es tan grande que creemos que debe haber aplicaciones en las cuales, si no se las utiliza, algunos tamaños de entrada resultarían intratables.

Sin embargo, implementar los algoritmos en C nos llevó en el orden de horas e implementarlos en ASM en el orden de días. Las implementaciones de algoritmos tan triviales en C pueden ser bastante complicadas en ASM usando las extensiones SSE del procesador. La implementación en ASM es bastante delicada, ya que es muy de bajo nivel y cualquier error/omisión puede ser difícil de debugguear. El recorrido de la matriz, cómo procesar los elementos en paralelo, cómo leer/escribir los bytes en memoria, etc. son preguntas que para implementar en C uno casi no necesita hacerse, porque son muy directas. En cambio, en ASM, hay que tomarse un tiempo para responder todas esas preguntas, pensando bien la respuesta. Esto hace que implementarlas en ASM sea bastante más complejo y propenso a errores.

Es decir, si bien implementando en ASM usando instrucciones SIMD se puede aumentar notablemente la performance comparado con una implementación en C, esto tiene un costo: mayor complejidad del código y es más propenso a errores. Nos parece razonable entonces el uso de ASM para optimizar ciertas funciones de una aplicación, si previamente se hizo un análisis de performance, se las encontró como cuello de botella y no se pudo mejorar su performance en C.