DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Implementación de filtros gráficos

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Alonso, Daiana	682/15	daianalonsok@gmail.com
Caballero, Tomás	628/15	tomycaballero95@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

Índice

1. Introducción

En este trabajo práctico se presentan tres implementaciones de filtros gráficos para el procesamiento de imágenes utilizando el modelo de procesamiento **SIMD (Single Instruction Multiple Data)**:

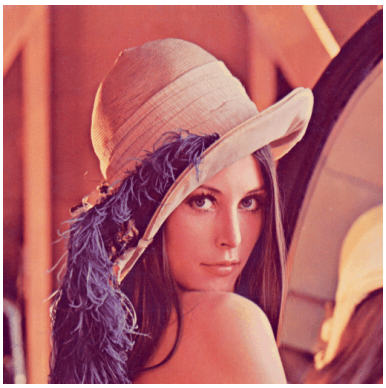
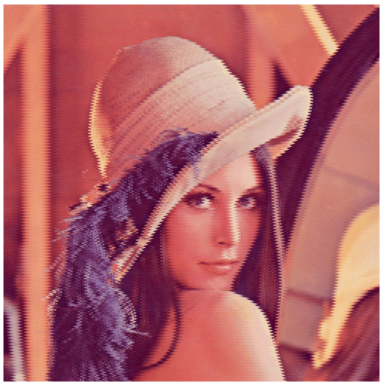



Figura 1: Imágenes de entrada



(a) Filtro **Zig-Zag**

(b) Filtro **Ocultar** (Segunda imagen dentro de la primera)

(c) Filtro **Descubrir**

Al trabajar con imágenes, aplicar un filtro implica procesar un conjunto de pixeles. Por este motivo, podremos computar un mismo conjunto de datos simultáneamente de forma **vectorizada** y lograr una mejora notable en la performance de los filtros. A continuación presentaremos la descripción de estos filtros gráficos, su implementación, y algunas hipótesis y experimentaciones en base a su rendimiento.

2. Desarrollo

2.1. Consideraciones generales

Consideraremos a una imagen como una **matriz de píxeles**, cada uno conformado por cuatro componentes: *Azul (b), Verde (g), Rojo (r) y Transparencia (a)*.
En nuestro caso particular cada una de estas componentes tendrán **8 bits** (1 byte) de profundidad, es decir, estarán representadas por números enteros sin signo en el rango $[0, 255]$, por lo tanto cada píxel ocupará **32 bits** (4 bytes).
Estas matrices de píxeles serán almacenadas por filas, tomando el píxel $[0, 0]$ de la matriz como el píxel arriba a la izquierda de la imagen.

```
typedef struct bgra_t {
    unsigned char b, g, r, a;
} __attribute__((packed)) bgra_t;
```

Figura 3: Estructura de un píxel

2.2. Filtro Ocultar

2.2.1. Descripción

Este filtro recibe dos imágenes I_1 y I_2 y genera una tercer imagen que **contiene oculta la imagen** I_2 **dentro de** I_1. El proceso se logra modificando los bits menos significativos de I_1, previa una conversión a escala de grises de I_2 para reducir la cantidad de información a ocultar.
Para esto tomaremos los seis bits más significativos del valor en escala de grises calculado anteriormente. Estos bits los tomaremos de a pares para poder ocultarlos de a dos en cada una de las componentes de color de la imagen destino.
Además para ocultar aún más la imagen, los bits de la misma no serán colocados directamente, sino que se realizará una operación **XOR** con los bits ya presentes en I_1. Estos bits no se tomaran directamente del píxel con el que se está operando, sino que se buscará el pixel correspondiente a recorrer en espejo la imagen.

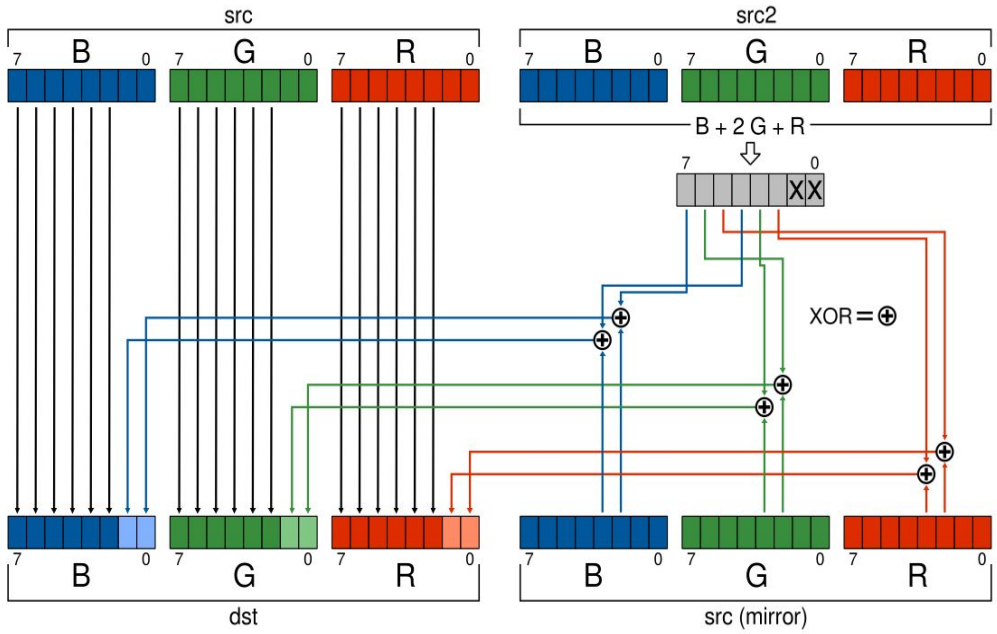


Figura 4

2.2.2. Pseudocódigo

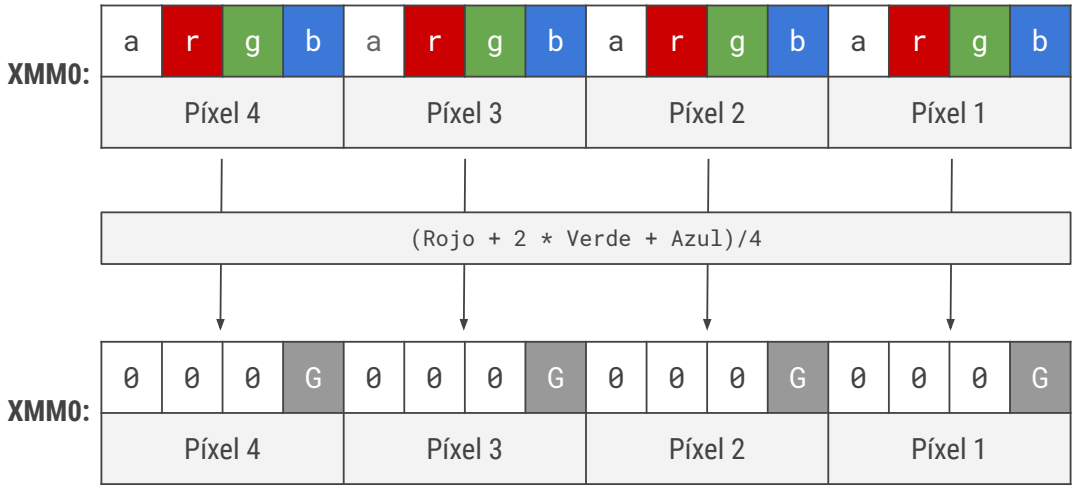
function OCULTAR(src, src_2, dst, $height$, $width$)
  for $i \in \{0 \dots height - 1\}$ **do**
    for $j \in \{0 \dots width - 1\}$ **do**
      $gris \leftarrow$CONVERTIRAGRIS(src_2)
      OCULTARCOMPONENTEAZUL($gris, src, i, j, height, width$)
      OCULTARCOMPONENTEVERDE($gris, src, i, j, height, width$)
      OCULTARCOMPONENTEROJA($gris, src, i, j, height, width$)
      $dst_{i,j}.alpha \leftarrow 255$
    end for
  end for
end function

Las funciones OCULTARCOMPONENTEAZUL, OCULTARCOMPONENTEVERDE y OCULTARCOMPONENTEROJA toman, cada una, dos bits del pixel $gris$ y aplican la operación `XOR` con el segundo y tercer bit de la componente de color correspondiente de $src_{height-i,width-j}$, como se puede observar en la figura 4.

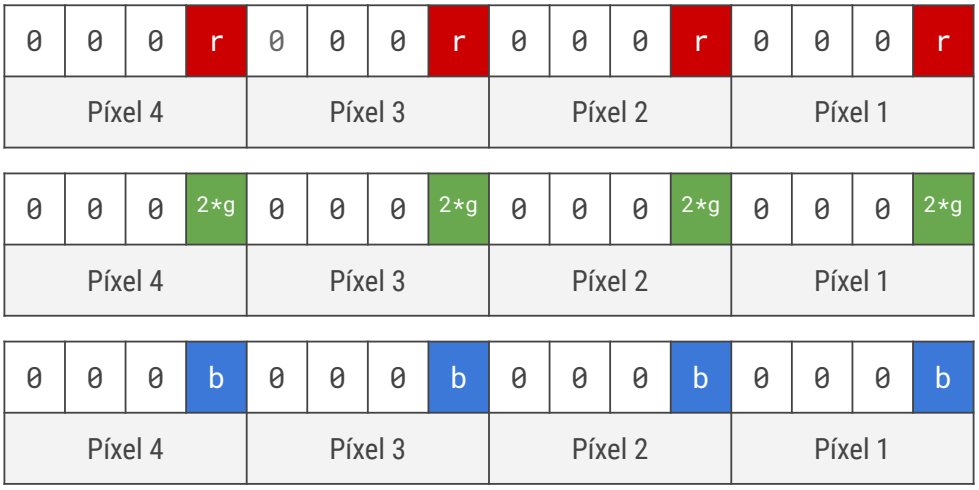
2.2.3. Implementación en ASM

El esquema abordado en la implementación del filtro es recorrer la matriz por filas y en cada una procesar cuatro píxeles en simultáneo. A continuación se presenta una secuencia donde se puede observar de forma gráfica las ideas principales implementadas:

- **Primera etapa:** *Pasar la imagen a ocultar a escala de grises*



Para convertir cada píxel a escala de grises, el objetivo es agrupar las componentes de forma tal que se pueda calcular $\frac{R+2*G+B}{4}$ utilizando operaciones aritméticas verticales. El primer paso consiste en separar cada componente de color en registros `XMM` distintos. Para esto, se aplican máscaras sobre el registro `XMM0` para filtrar cada color y luego desplazamientos para el ordenamiento de los valores. Como resultado se obtiene, almacenado en el byte menos significativo de cada DWORD (32 bits) una componente de color:



El paso siguiente tiene por objetivo aplicar operaciones de suma vertical y shifts para realizar producto y división (ya que son operaciones en base 2) para calcular por cada pixel $\frac{R+2*G+B}{4}$.

- **Segunda etapa:** *Ocultar la imagen*
  Como vimos en la descripción del algoritmo, para ocultar la imagen tomaremos los seis bits menos significativos del valor en escala de grises calculado anteriormente, los píxeles de la imagen I_1 y de sus opuestos en espejo. Para eso utilizaremos otros dos registros `XMM`: en uno levantaremos cuatro píxeles de la imagen fuente y en el otro los cuatro píxeles opuestos, sobre los cuales aplicaremos desplazamientos, `XOR` y `AND`, operando cada componente separada por colores como vimos en la figura anterior. Finalmente moveremos los resultados a la imagen destino.
  De esta forma, se pueden procesar, en simultáneo, cuatro píxeles por iteración y obtener cuatro resultados.

2.3. Filtro Descubrir

2.3.1. Descripción

Este filtro realiza la operación inversa al filtro OCULTAR a partir de aplicar las mismas operaciones de forma inversa. El resultado será una reconstrucción de la imagen oculta en escala de grises, donde los últimos dos bits de cada píxel (que no son almacenados en la imagen) serán seteados a cero.

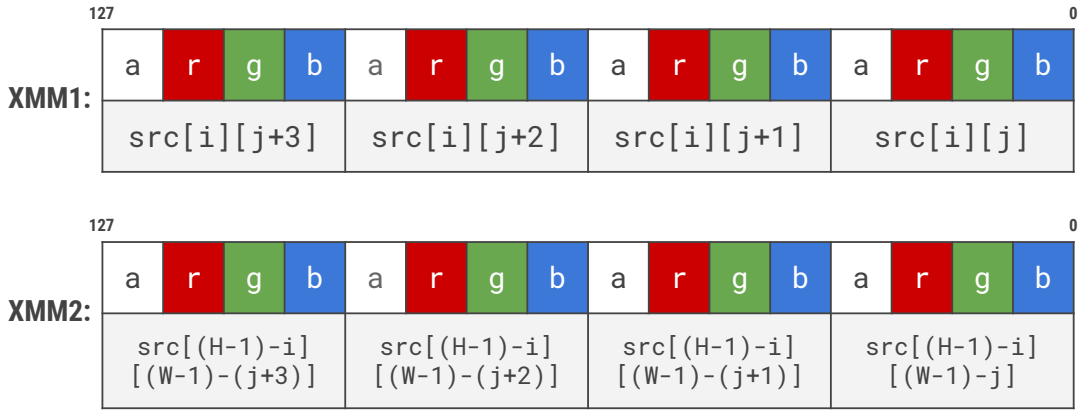
2.3.2. Pseudocódigo

function DESCUBRIR(src, dst, $height$, $width$)
  for $i \in \{0 \dots height - 1\}$ **do**
    for $j \in \{0 \dots width - 1\}$ **do**
      $B \leftarrow$ DESCUBRIRCOMPONENTEAZUL$(src, i, j, height, width)$
      $G \leftarrow$ DESCUBRIRCOMPONENTEVERDE$(src, i, j, height, width)$
      $R \leftarrow$ DESCUBRIRCOMPONENTEROJA$(src, i, j, height, width)$
      $color \leftarrow$ OBTENERCOLOR(R, G, B)
      $dst_{i,j} \leftarrow color$
      $dst_{i,j}.alpha \leftarrow 255$
    end for
  end for
end function

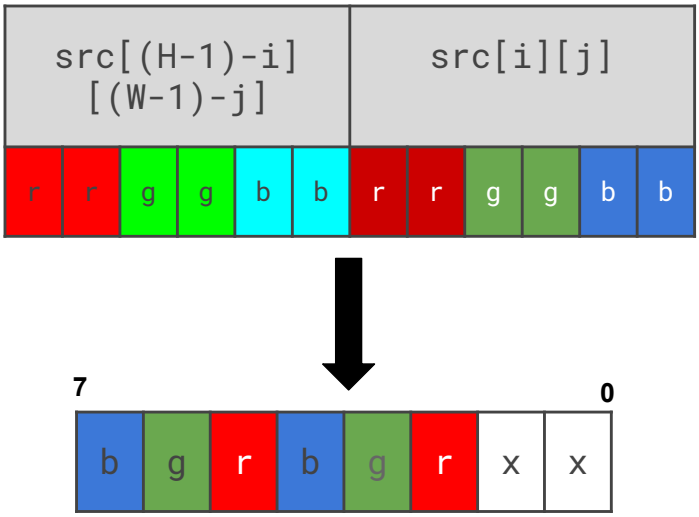
Las funciones DESCUBRIRCOMPONENTEAZUL, DESCUBRIRCOMPONENTEVERDE y DESCUBRIRCOMPONENTEROJA realizan las operaciones inversas de las funciones auxiliares utilizadas en OCULTAR.

2.3.3. Implementación en ASM

Por simplicidad llamaremos H a $height$ (cantidad de filas de la matriz de píxeles) y W a $width$ (cantidad de columnas de la matriz de píxeles).
Tendremos dos ciclos donde recorreremos la imagen por filas y columnas, y se procesan cuatro posiciones en simultáneo levantando los pixeles $src_{i,j}$, $src_{i,j+1}$, $src_{i,j+2}$, $src_{i,j+3}$ en un registro y $src_{(H-1)-i,(W-1)-j}$, $src_{(H-1)-i,(W-1)-(j+1)}$, $src_{(H-1)-i,j(W-1)-(j+2)}$, $src_{(H-1)-i,(W-1)-(j+3)}$ en otro:



Realizaremos la operatoria descripta en el pseudocódigo bit a bit para obtener el color gris de cada pixel que generamos en OCULTAR y almacenaremos el resultado en destino:



2.4. Filtro Zigzag

2.4.1. Descripción

Este filtro genera un efecto de Zig-Zag sobre las imágenes. Para esto separa las operaciones a realizar en cuatro casos, dependiendo la fila de la imagen a procesar:

i es el número de fila	pixel destino $dst_{i,j}$
$i \equiv 0(4)$	$(src_{i,j-2} + src_{i,j-1} + src_{i,j} + src_{i,j+1} + src_{i,j+2})/5$
$i \equiv 1(4)$	$src_{i,j-2}$
$i \equiv 2(4)$	$(src_{i,j-2} + src_{i,j-1} + src_{i,j} + src_{i,j+1} + src_{i,j+2})/5$
$i \equiv 3(4)$	$src_{i,j+2}$

Esta operatoria se realiza sobre todas las componentes de cada uno de los píxeles. El primer y tercer caso son iguales, y realizan un promedio de dos píxeles vecinos del píxel a procesar. Los otros dos casos, copian los píxeles dos posiciones a derecha o a izquierda, respectivamente. Además se va a dejar un marco blanco de dos píxeles alrededor de toda la imagen.

2.4.2. Pseudocódigo

function ZIGZAG(src, dst, $height$, $width$)
  for $i \in \{0, 1\}$ **do**
    for $j \in \{0 \dots width - 1\}$ **do**
      $dst_{i,j} \leftarrow$ PIXELBLANCO
    end for
  end for

  for $i \in \{0 \dots height - 2\}$ **do**
    for $j \in \{0 \dots width - 1\}$ **do**
      if $j \leq 1 \vee j \geq width - 2$ **then**
        $dst_{i,j} \leftarrow$ PIXELBLANCO
      end if
      if $i \equiv 0(4) \vee i \equiv 2(4)$ **then**
        $dst_{i,j} \leftarrow \frac{src_{i,j-2} + src_{i,j-1} + src_{i,j} + src_{i,j+1} + src_{i,j+2}}{5}$
      else
        if $i \equiv 1(4)$ **then**
          $dst_{i,j} \leftarrow src_{i,j-2}$
        else
          $dst_{i,j} \leftarrow src_{i,j+2}$
        end if
      end if
      $dst_{i,j}.alpha \leftarrow 255$
    end for
  end for

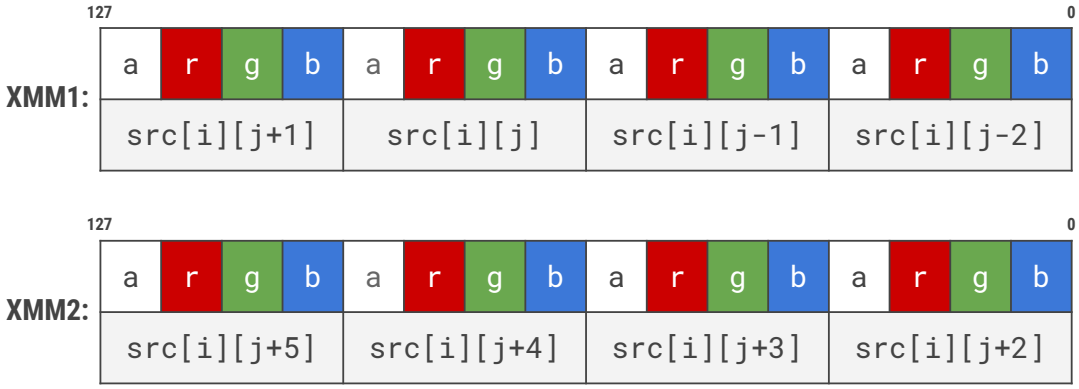
  for $i \in \{height - 2, height - 1\}$ **do**
    for $j \in \{0 \dots width - 1\}$ **do**
      $dst_{i,j} \leftarrow$ PIXELBLANCO
    end for
  end for
end function

2.4.3. Implementación en ASM

Como dijimos anteriormente, cada píxel ocupa 4 bytes, por lo tanto en un registro `XMM` podemos levantar de la matriz de a cuatro píxeles por registro.
Tendremos cuatro ciclos:

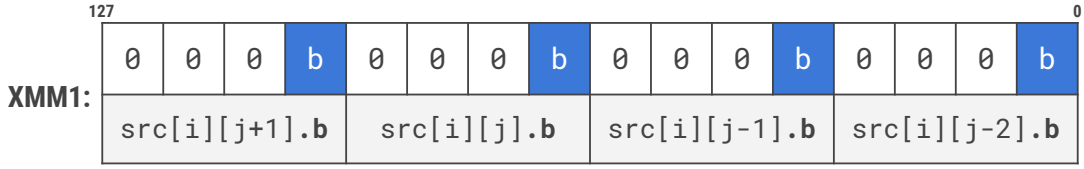
- En el primer y último ciclo levantaremos todos los píxeles pertenecientes a las dos primeras y dos últimas filas tomados de a cuatro y los pondremos en color blanco (255).
- En el segundo ciclo recorreremos las filas i y en el tercero las columnas j. Para los valores de $j \in \{0, 1, width - 2, width - 1\}$ pondremos esos píxeles en blanco para dejar ese marco de dos píxeles de final y principio de fila.

Vamos a querer tomar los dos píxeles a la izquierda y los píxeles a la derecha del de la posición (i, j), por lo tanto, vamos a utilizar dos registros: en uno vamos a levantar los píxeles $src_{i,j-2}$, $src_{i,j-1}$, $src_{i,j}$, $src_{i,j+1}$, y en el otro registro `XMM` vamos a levantar de $src_{i,j+2}$ en adelante:

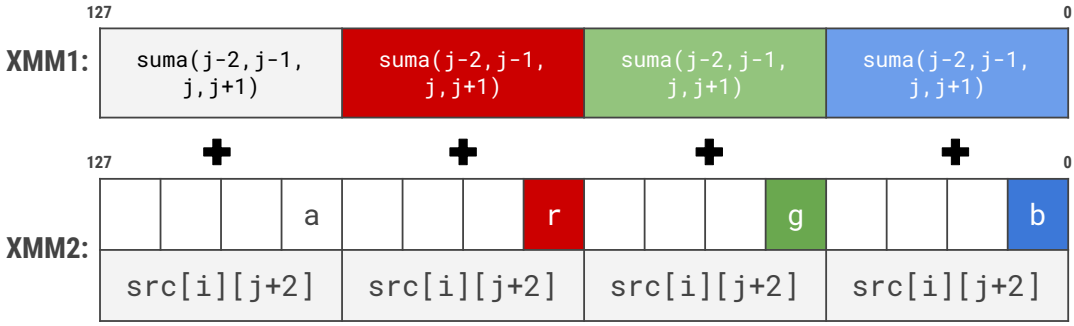


En cada fila, calcularemos el módulo de i y procederemos de la siguiente forma:

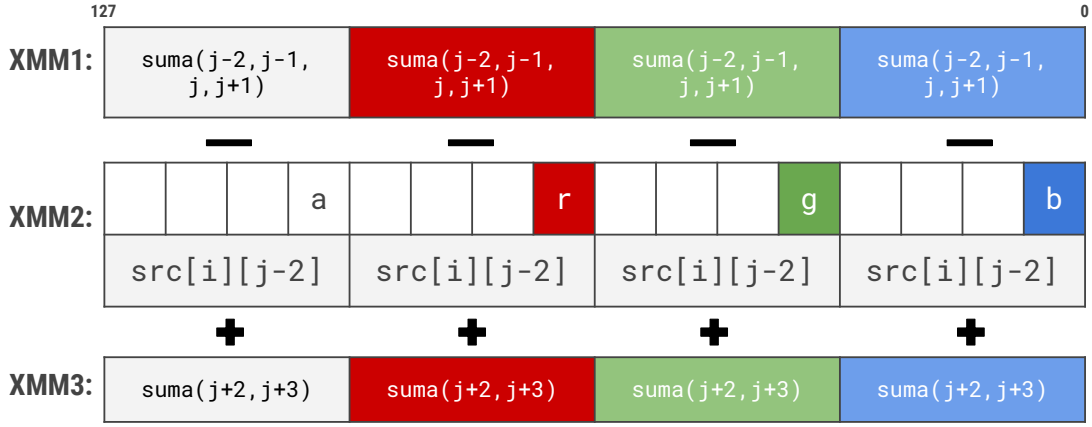
- Caso $i \equiv 0(4) \vee i \equiv 2(4)$: Luego de levantar esos píxeles, separaremos y reordenaremos sus componentes para poder operar con cada valor de color de un byte por separado.



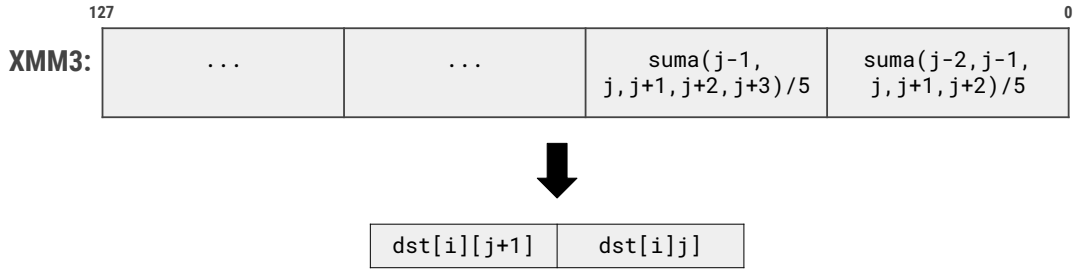
Para calcular $dst_{i,j} = \frac{suma(j-2,j-1,j,j+1,j+2)}{5}$ y $dst_{i,j+1} = \frac{suma(j-1,j,j+1,j+2,j+3)}{5}$ realizaremos una suma horizontal dos veces para obtener la suma entre los valores de cada componente de los cuatros primeros píxeles y luego agregaremos en una suma vertical $src_{i,j+2}$ como muestra la figura:



Después para obtener $suma(j-1, j, j+1, j+2, j+3)$ le restamos $src_{i,j-2}$ mediante una resta vertical. Luego realizamos una suma vertical entre $src_{i,j+2}$ y $src_{i,j+3}$ y lo sumamos a la cuenta anterior mediante otra suma vertical:



Realizaremos una conversión a $float$ en los dos casos para poder dividir por 5. Después de eso reconvertiremos los valores a $enteros$ y reorganizaremos el resultado con instrucciones $pack$ para poder almacenar el resultado en $dst_{i,j}$ y $dst_{i,j+1}$.



En este caso procesamos dos píxeles por iteración.

- Caso $i \equiv 1(4)$:
  Tomaremos los primeros valores que levantamos que corresponden a $src_{i,j-2}$, $src_{i,j-1}$, $src_{i,j}$, $src_{i,j+1}$ y los moveremos a $dst_{i,j}$, $dst_{i,j+1}$, $dst_{i,j+2}$, $dst_{i,j+3}$
- Caso $i \equiv 3(4)$:
  Tomaremos los valores que levantamos en el segundo registro que corresponden a $src_{i,j+2}$, $src_{i,j+3}$, $src_{i,j+4}$, $src_{i,j+5}$ y los moveremos a $dst_{i,j}$, $dst_{i,j+1}$, $dst_{i,j+2}$, $dst_{i,j+3}$.

En los ultimos dos casos procesamos cuatro píxeles por iteración.

3. Experimentos

3.1. Consideraciones generales

En esta sección se presentan los resultados de dos experimentos aplicando los filtros desarrollados sobre una serie de imágenes. En todos los casos se midió el tiempo de ejecución en **ciclos de clock**, utilizando la instrucción de assembler `rdtsc`. Cada ejecución fue repetida **50 veces** para mejorar la calidad de los resultados, donde luego se calculó el **tiempo promedio por iteración**.

Las imágenes utilizadas como entrada son `color.bpm` y `pico.bmp`, tomadas en diferentes dimensiones: 1856×928 (1722368 pixeles), 1600×800 (1280000 pixeles), 1280×640 (819200 pixeles), 1024×512 (524288 pixeles), 800×400 (320000 pixeles), 512×256 (131072 pixeles), 400×200 (80000 pixeles), 256×128 (32768 pixeles), 200×100 (20000 pixeles), 128×64 (8192 pixeles), 64×32 (2048 pixeles), 32×16 (512 pixeles).

(a) `color.bmp`

(b) `pico.bmp`

3.2. Comparación C vs ASM

Llevaremos a cabo una comparación sobre los tres filtros desarrollados, evaluará la performance de dos implementaciones en **C** dadas por la cátedra (una versión compilada con el flag `-O0` y la otra con el flag `-O3`) y nuestra implementación en **Assembler**. Se medirá el tiempo de cómputo obtenido luego de aplicarlo a cada una de las imágenes mencionadas, en cada una de sus dimensiones: imagen en color vs imagen que contiene muchas componentes en blanco (valores grandes en cada componente del píxel).

Es importante notar que además se compilaron y corrieron versiones de los filtros en C utilizando los flags de optimización `-O1` y `-O2` pero al ser resultados muy similares a los obtenidos con el flag `-O3`, descartaremos por simplicidad los tiempos obtenidos en los gráficos a continuación.

Hipótesis

Se espera observar que las implementaciones desarrolladas en **Assembler** sean más eficientes, independientemente del tamaño de imagen considerado. Estos resultados deberían reflejar las ventajas de utilizar el modelo **SIMD** por sobre las implementaciones en C, donde se procesan los pixeles de forma individual.

Resultados obtenidos

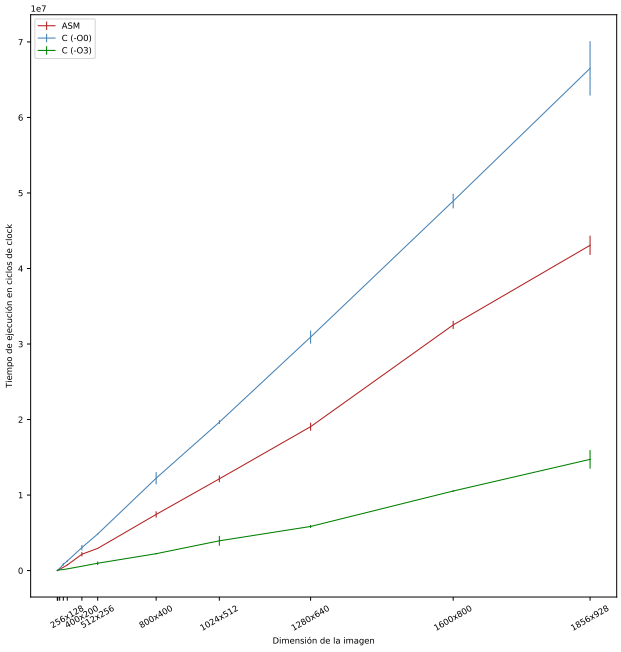


Figura 6: Filtro **Zigzag** sobre `color.bmp`

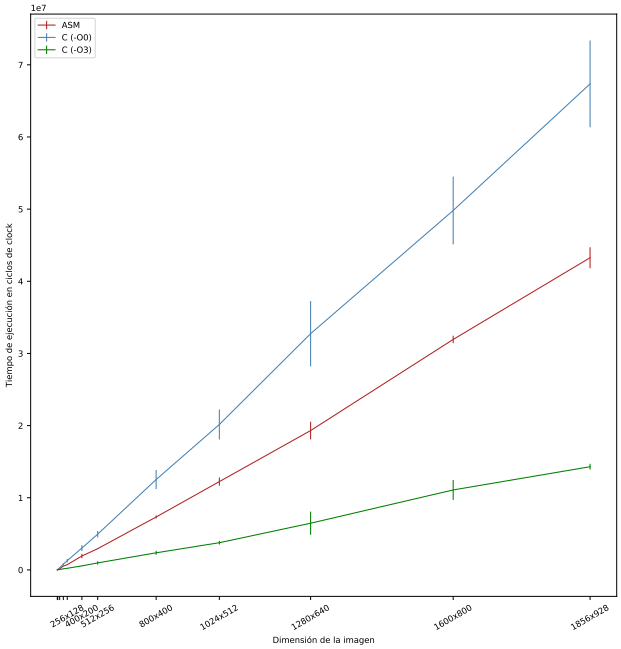


Figura 7: Filtro **Zigzag** sobre `pico.bmp`

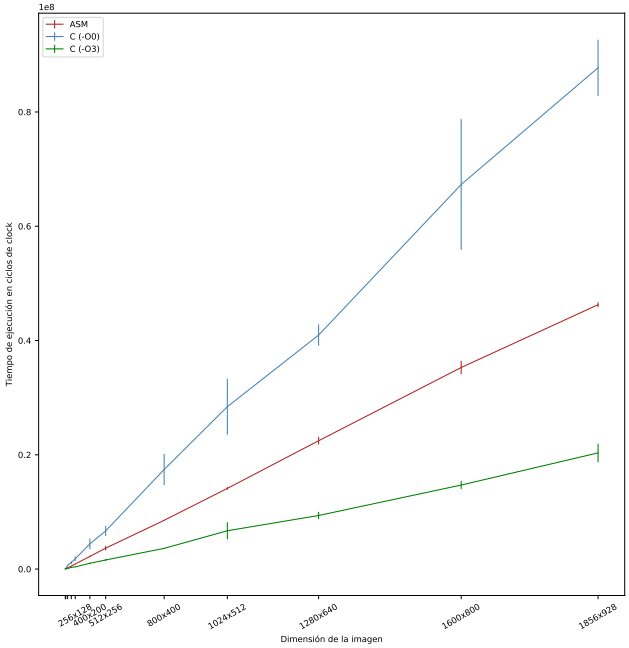


Figura 8: Filtro **Ocultar** (`pico.bmp` oculta en `color.bmp`)

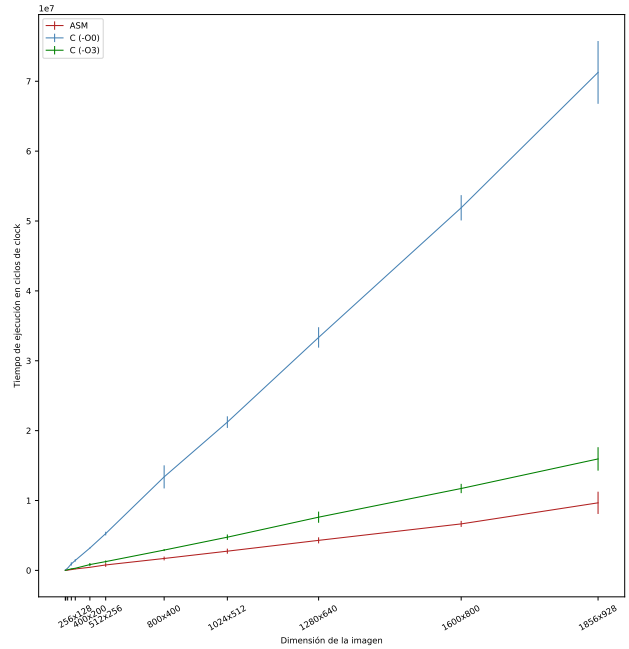


Figura 9: Filtro **Descubrir** sobre `pico.bmp` oculta en `color.bmp`

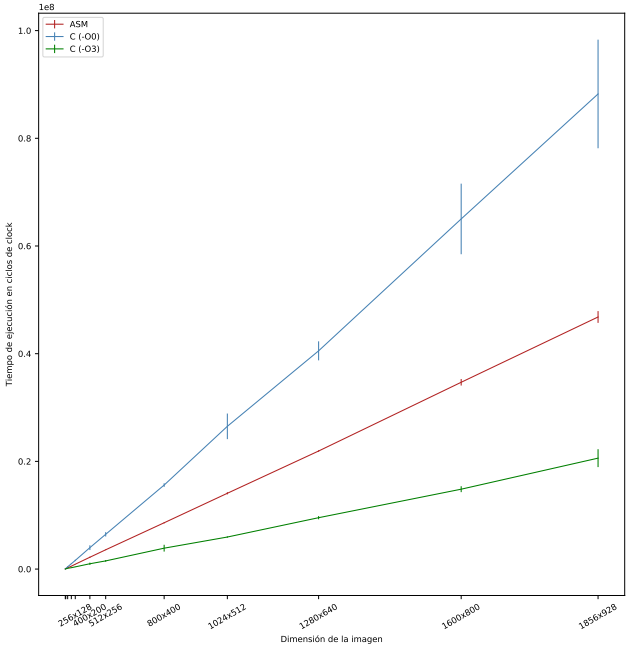


Figura 10: Filtro **Ocultar** (`color.bmp` oculta en `pico.bmp`)

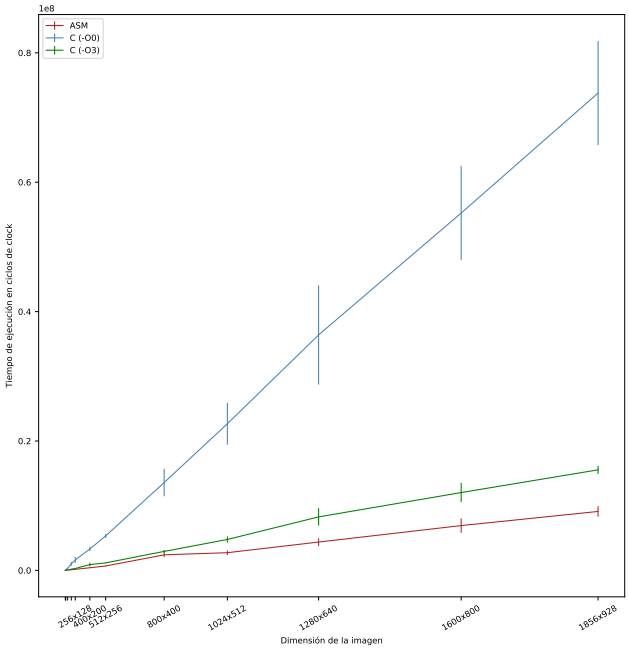


Figura 11: Filtro **Descubrir** sobre `color.bmp` oculta en `pico.bmp`

A partir de los resultados obtenidos se puede confirmar parcialmente la hipótesis planteada: la implementación en lenguaje ensamblador demoró menos ciclos de clock que la implementación en C compilada con el flag `-O0` (sin optimizaciones) en todos los casos e independientemente del tamaño de imagen procesado. Además, se puede observar que el rendimiento de los algoritmos no difiere si tomamos imágenes

con características distintas en términos de los valores de sus píxeles.
En el caso de los filtros ZIGZAG y OCULTAR, la compilación en C con el flag `-O3` mostró un rendimiento mejor en términos de ciclos de clock, y ocurrió lo contrario en la implementación del filtro DESCUBRIR.
En el caso del filtro ZIGZAG, podemos mencionar algunos motivos por los cuales no se obtuvieron amplias diferencias de performance contra la implementación en C sin optimizaciones, como en los otros dos filtros. En la implementación en lenguaje ensamblador, aquellos casos donde solo realizamos movimientos a registros, avanzamos cuatro píxeles por iteración y, en los casos donde calculamos promedios, procesamos dos píxeles por iteración. Cambiando la implementación en el segundo caso mencionado, seguramente se hubiesen obtenido mejores resultados o muy similares a los vistos en DESCUBRIR y OCULTAR.

En conclusión, la hipótesis planteada de que la implementación en lenguaje ensamblador es más eficiente que la realizada en lenguaje C no ocurre en la mayoría de los casos. En la comparación contra el filtro en C optimizado con el flag `-O3`, solo pudimos ver que se cumple en la implementación del filtro DESCUBRIR. No conocemos como están implementadas esas optimizaciones que realiza el compilador, pero podemos conjeturar, en base a nuestro desarrollo de los algoritmos en lenguaje ensamblador, que este resultado desfavorable puede deberse a la cantidad de operaciones aritméticas utilizadas, los saltos condicionales, o la cantidad de valores que estemos procesando por iteración en esos algoritmos.

3.3. Experimento 1: Performance ASM evaluando cantidad de conversiones

Para este experimento modificamos el código realizado en ASM en el filtro ZIGZAG. El objetivo es utilizar instrucciones costosas lo menos posible, a fines de poder comparar las dos implementaciones y ver la variación entre las mismas.
Como estuvimos viendo en la materia una de las instrucciones que resulta ser costosa es la de conversión. En nuestro caso nos vimos obligados a utilizarla para poder realizar la división utilizando `divps`, que es una operación sobre valores de precisión simple de 32 bits. Por lo tanto cuando calculamos la suma de los valores, realizamos todas las operaciones posibles con enteros para reducir la cantidad de conversiones aplicadas. Luego, utilizamos la instrucción `cvtdq2ps` para pasar de valores enteros de 32 bits a valores de precisión simple de 32 bits, los dividimos por *5.0* y el resultado lo convertimos a entero nuevamente con la instrucción `cvtps2dq`.

Entonces nos preguntamos, *¿Hay una diferencia notoria de performance si minimizamos la cantidad de conversiones?*. Nuestra hipótesis es que *minimizar la cantidad de instrucciones de conversión mejora el rendimiento del algoritmo*. Para eso modificamos el código de ZIGZAG y en lugar de realizar la conversión a punto flotante, luego de calcular las sumas y unir las componentes de cada color, lo realizamos para cada componente. Por lo tanto, en el código original donde teníamos 4 conversiones (dos para pasar de entero a punto flotante y dos para volver de punto flotante a entero), ahora pasamos a tener 12 conversiones: tres de entero a punto flotante para cada componente de la primer suma, tres de entero a punto flotante para cada componente de la segunda suma, y luego las seis correspondientes para pasar de punto flotante nuevamente a entero.

Resultados obtenidos

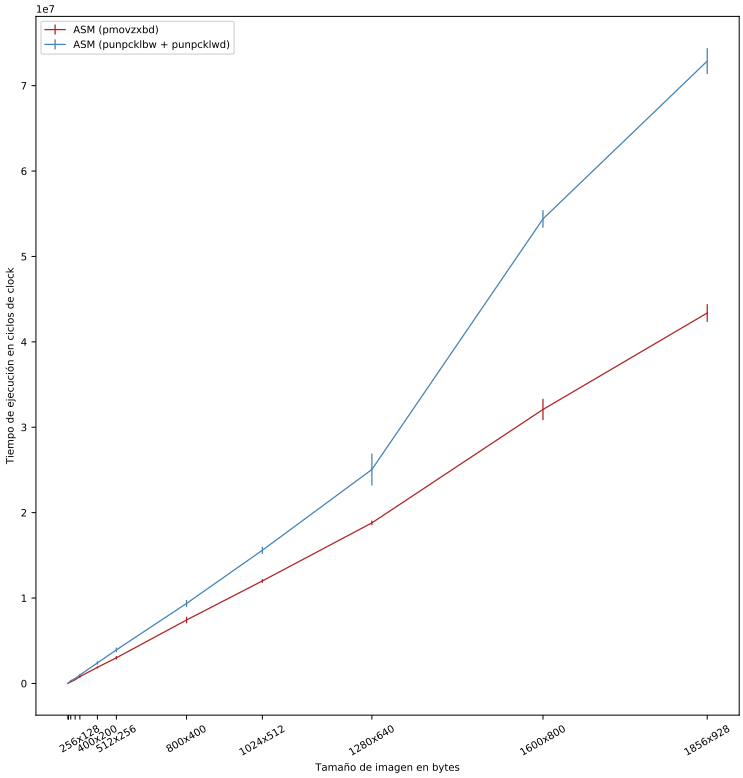


Figura 12: Filtro **Zigzag**

El resultado de este experimento concuerda con la hipótesis que planteamos inicialmente acerca del costo de complejidad de utilizar más operaciones de conversión. Podemos ver que en los casos en los que procesamos imágenes más pequeñas no hay tanta diferencia, pero a medida que aumenta el tamaño de entrada la diferencia entre ambas implementaciones es más notoria.
Nuestra primera versión código realizaba más conversiones que la versión final, y logramos ver, que repensando la solución y utilizando menos operaciones de conversión, sí hay una diferencia notoria entre ambas implementaciones. Esto nos deja en claro que para poder optimizar un programa en lenguaje ensamblador que requiera conversiones y que no incremente el costo de ejecución, debemos minimizar la cantidad de veces que utilizamos estas instrucciones.

3.4. Experimento 2: Performance ASM operaciones desempaquetado vs. operaciones load/store

Para este experimento nuevamente modificamos el código realizado en ASM en el filtro ZIGZAG. En el código original utilizamos la operación de load/store `pmovzxbd` para tomar el píxel de la parte baja del registro y extender cada componente con ceros (ya que los valores son enteros sin signo) de 8 bits a 32 bits:

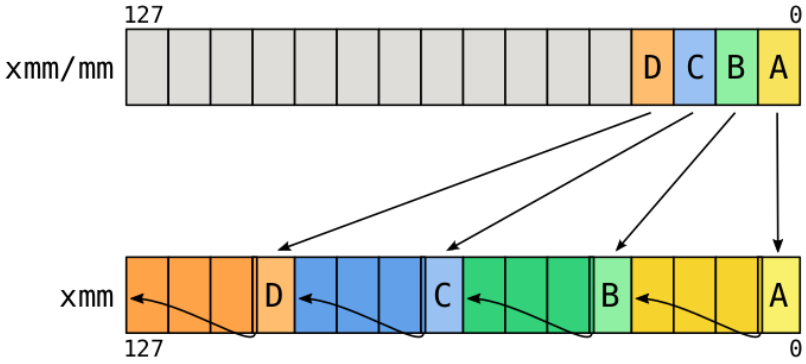


Figura 13: `PMOVZXBD` extensión con ceros de 8 bits a 32 bits

Pero esta instrucción puede ser reemplazada por dos operaciones de desempaquetado, donde src sea un registro `XMM` con valores en cero para que extienda el signo de los valores de dst con ceros. Para eso reemplazaremos `pmovzxbd` por `punpcklbw` y `punpcklwd`. La primera extenderá las componentes de 8 bits a 16 bits y la segunda de 16 bits a 32 bits.

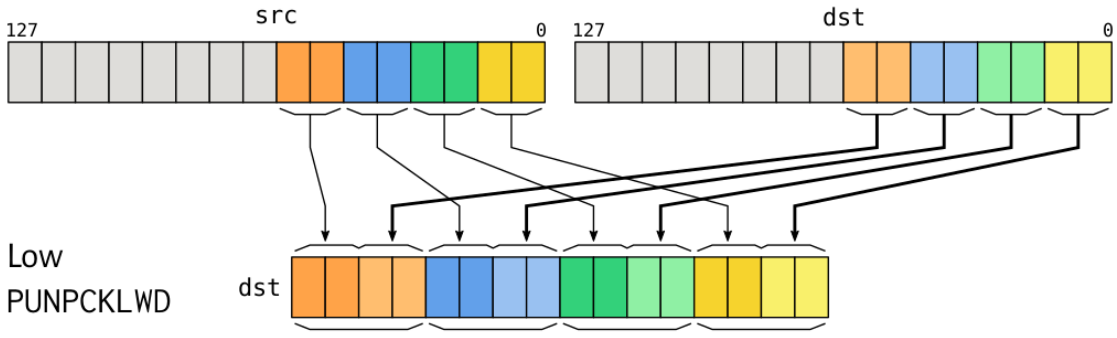


Figura 14: `PUNPCKLWD` desempaqueta cuatro enteros de la parte baja de 16 bits en 32 bits

Comparando las dos implementaciones, bajo los mismos parámetros de entrada, nos preguntamos si afecta a la performance del algoritmo utilizar un tipo de operación o la otra. Nuestra hipótesis será que *utilizar movimientos con extensión de signo, en lugar de desempaquetados, mejora el tiempo de ejecución de nuestro programa.*

Resultados obtenidos

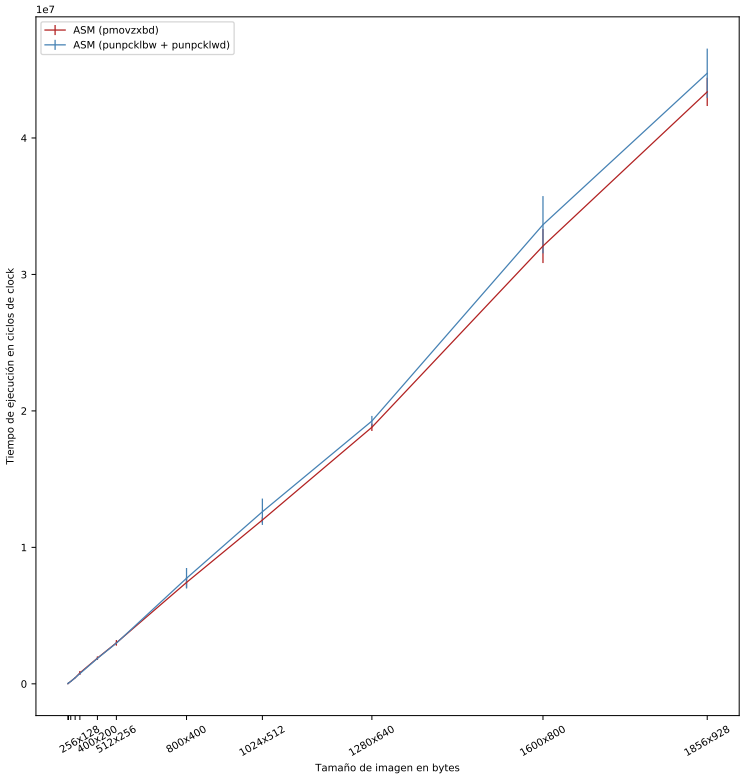


Figura 15: Filtro **Zigzag**

Mediante el experimento realizado entre las dos implementaciones, podemos ver que se cumple la hipótesis planteada. En tamaños de imagen pequeños casi no hay diferencia, y a medida que el tamaño de entrada aumenta podemos ver que la implementacion con movimiento de extensión de signo es más eficiente, aunque no hay diferencias muy significativas. A grandes rasgos, elegir entre un tipo de operación u otra no afectaría demasiado a la performance final de nuestro programa en lenguaje ensamblador.

4. Conclusión

En este trabajo práctico pudimos ver las ventajas significativas que puede brindarnos el paradigma de SIMD a la hora de implementar programas que puedan realizar operaciones en paralelo. Esto se ve reflejado notablemente en el rendimiento de las implementaciones que utilizan este modelo y las de alto nivel, como las implementadas en lenguaje C.
Debemos contraponer también todo lo que conlleva una implementación en lenguaje ensamblador: un código mucho más extenso, propenso a errores y confusiones, menos legible. Por eso debemos tener en consideración estas características para poder decidir si este esfuerzo adicional, de hacer un trabajo más minucioso, lo vale.
Como trabajo pendiente nos quedaría repensar la implementación en el filtro ZIGZAG para poder obtener

cuatro resultados por iteración en lugar de dos, en el caso donde se calcula el promedio, y poder mejorar aún más su rendimiento. Por otro lado, queda pendiente analizar en profundidad las causas por las cuales las optimizaciones en lenguaje C resultaron más eficientes que las propuestas en los casos vistos, cómo fueron implementadas esas optimizaciones, y cómo podemos mejorar los algoritmos desarrollados en lenguaje ensamblador para obtener, si es posible, resultados más eficientes que los anteriores y lo suficientemente óptimos para superar las implementaciones optimizadas en lenguaje C.