DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Primer Cuatrimestre de 2019

Integrante	LU	Correo electrónico
Giuliana Barbieri	688/15	giulianabarbieriii@gmail.com
Carlos Giudice	694/15	carlosr.giudice@gmail.com
Gastón Máspero	131/17	gaston.maspero@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

1.1. Objetivo

El objetivo del presente trabajo es la implementación en lenguaje assembly para arquitecturas x86 de 4 filtros para imágenes BMP, explotando el uso de instrucciones SIMD, para luego analizar su performance con respecto a implementaciones más convencionales (realizadas en C).

1.2. SIMD

Las siglas SIMD refieren a *Single Instruction Multiple Data* (en contraposición a SISD: *Single Instruction Single Data*) y denotan un modelo de ejecución capaz de computar una misma operación sobre un conjunto de múltiples datos. Esta técnica de *paralelismo a nivel de datos*, es implementada en x86 con la extensión del set de instrucciones llamada SSE, *Streaming SIMD Extensions*, que forma el conjunto de instrucciones utilizado y explorado en este trabajo. Esta extensión incluye su propio set de 16 registros de 128 bits.

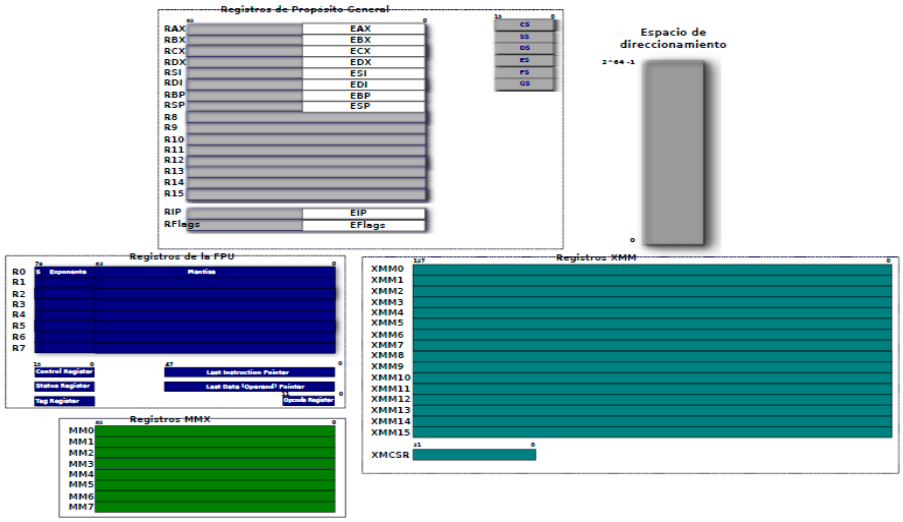
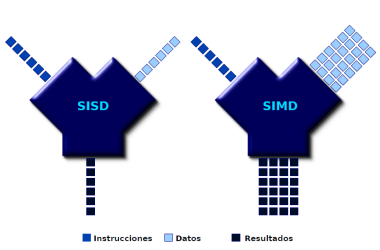
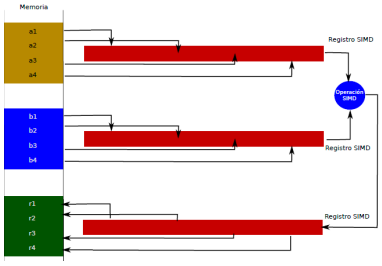


Figura 1: Arquitectura x86 con SSE

La figura 1 muestra un pequeño overview de los registros presentes en la arquitectura con la que estamos trabajando, en particular, los registros xmm de abajo a la derecha, utilizados en instrucciones SIMD.



(a) Concepto de cómputo en SIMD



(b) Ejemplo de cómputo en SIMD

Figura 2: Visualizando SIMD

La figura 2 muestra conceptualmente el procesamiento de datos utilizando el modelo SIMD. En la figura 2b se puede apreciar como con una instruccion podemos computar, por ejemplo, 4 pares de datos de 32 bits.

Si bien SIMD no es aplicable a todo algoritmo, es particularmente útil en tratamiento de imágenes (así como también audio y video), en el que es usual procesar gran cantidad de píxeles siguiendo un patrón común.

1.3. Formato de imagen

Los filtros implementados operan sobre imágenes en formato BMP cuyo ancho es mayor a 16 píxeles y múltiplo de 8 píxeles. Estas consideraciones fueron realizadas (además de ser decisiones de la cátedra) para simplificar la implementación y evitar perder tiempo en casos bordes, ya que lo que se busca aquí no es crear una biblioteca profesional de filtros para imágenes, sino analizar las mejoras en performance que ofrecen los modelos SIMD.
Los archivos BMP (por sus siglas *Bitmap Format*) se componen de direcciones asociadas a códigos de color, uno para cada cuadro en una matriz de píxeles tal como se esquematizaría un dibujo de çolorea los cuadros"para niños pequeños. Normalmente, se caracterizan por ser muy poco eficientes en su uso de espacio en disco, pero pueden mostrar un buen nivel de calidad. A diferencia de los gráficos vectoriales al ser recalados a un tamaño mayor, pierden calidad.

Bytes	Información
0, 1	Tipo de fichero "BM"
2, 3, 4, 5	Tamaño del archivo
6, 7	Reservado
8, 9	Reservado
10, 11, 12, 13	Inicio de los datos de la imagen
14, 15, 16, 17	Tamaño de la cabecera del bitmap
18, 19, 20, 21	Anchura (píxels)
22, 23, 24, 25	Altura (píxels)
26, 27	Número de planos
28, 29	Tamaño de cada punto
30, 31, 32, 33	Compresión (0=no comprimido)
34, 35, 36, 37	Tamaño de la imagen
38, 39, 40, 41	Resolución horizontal
42, 43, 44, 45	Resolución vertical
46, 47, 48, 49	Tamaño de la tabla de color
50, 51, 52, 53	Contador de colores importantes

Figura 3: Header de un archivo BMP

La figura 3 muestra el contenido de los bytes correspondientes a un archivo BMP. Por otra parte, el Bitmap de una imagen BMP comienza a leerse desde abajo a arriba, es decir: en la primera fila de la de la matriz se encuentra la última línea de la imagen, en la segunda fila se encuentra la anteúltima y así sucesivamente. Dentro de cada línea los píxeles se almacenan de izquierda a derecha, y cada píxel en memoria se guarda en el siguiente orden: B, G, R, A, donde cada uno toma valores entre 0 y 255 (ocupan un byte).

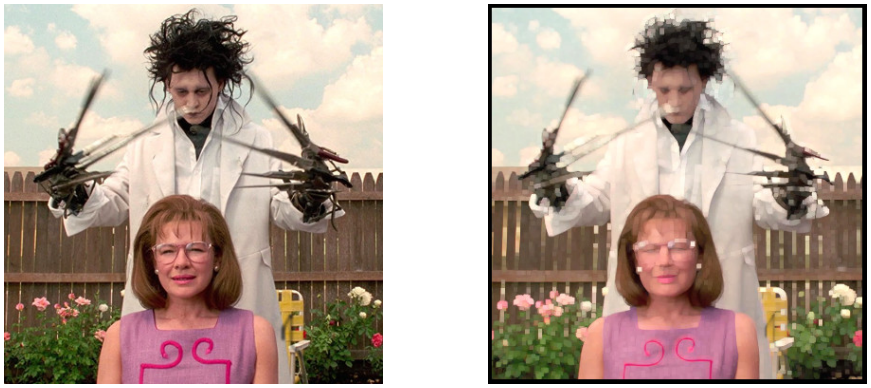
En este trabajo se utiliza una biblioteca y un framework provistos por la cátedra, que simplifican el manejo de estas imágenes, abstrayendo la implementación de los filtros a una función que recibe el array de bytes correspondientes al bitmap de la imagen.

1.4. Filtros

Se implementarán en lenguaje assembly para arquitectura x86 los filtors: Cuadrados, Manchas, Offset y Sharpen.
En todos los filtros el byte de transparencia se setea en 255.

1.4.1. Cuadrados

Este filtro genera un efecto similar al desenfoque, replicando los píxeles de valor mayor alrededor de cada uno. Las operaciones se realizan dejando un marco de 4 píxeles en negro alrededor de toda la imagen, tal como lo muestra la figura 4.



(a) Imagen original (b) Imagen con filtro Cuadrados

Figura 4: Filtro Cuadrados

El pseudocódigo que describe este filtro se incluye en el siguiente snippet:

```
Para i de 4 a height - 5:
    Para j de 4 a width - 5:
        maxB = 0, maxG = 0, maxR = 0
        Para ii de 0 a 3:
            Para jj de 0 a 3:
                maxB = maximo(maxB, src[i + ii][j + jj].b)
                maxG = maximo(maxG, src[i + ii][j + jj].g)
                maxR = maximo(maxR, src[i + ii][j + jj].r)

        dst[i][j].b = maxB
        dst[i][j].g = maxG
        dst[i][j].r = maxR
```

1.4.2. Manchas

Este filtro genera una serie de manchas sobre la imagen modificando los valores de cada píxel en función de una fórmula que calcula el tono. Ésta toma como parámetro el diámetro de las manchas, que en el pseudocódigo a continuación es identificado como n.
Un ejemplo de este filtro se incluye en la figura 5.

El pseudocódigo que describe este filtro se incluye en el siguiente snippet:

(a) Imagen original

(b) Imagen con filtro Manchas

Figura 5: Filtro Manchas

```
Para i de 0 a height - 1:
    Para j de 0 a width - 1:
        ii = 2 * PI * (i \% n) / n
        jj = 2 * PI * (jj \% n) / n
        tono = sin(ii) * cos (jj) * 50 -25

        dst[i][j].b = SAT(src[i][j].b +  tono)
        dst[i][j].g = SAT(src[i][j].g +  tono)
        dst[i][j].r = SAT(src[i][j].r +  tono)
```

1.4.3. Offset

El color de cada píxel está dado por sus tres componentes: rojo, verde y azul. En las impresiones por offsets, los colores se forman aplicando una a una cada componente sobre el papel. Cuando la aplicación de los colores se desfasa, se produce un efecto de estela de color alrededor de cada figura. El objetivo de este filtro es generar este efecto para un desfasaje fijo de las componentes de color de las imágenes, respetando el siguiente pseudocódigo:

```
Para i de 0 a height - 9:
    Para j de 0 a width - 9:
        dst[i][j].b = src[i+8][j].b
        dst[i][j].g = src[i][j+8].g
        dst[i][j].r = src[i+8][j+8].r
```

Un ejemplo de este filtro se incluye en la figura 6.

1.4.4. Sharpen

Existe un conjunto de efectos sobre imágenes que se realizan aplicando operaciones matriciales sobre los píxeles. Estas matrices se denominan operadores. Para este filtro se implementa el operador Sharpen o ”realzado”, respetando el siguiente pseudocódigo:

(a) Imagen original

(b) Imagen con filtro Offset

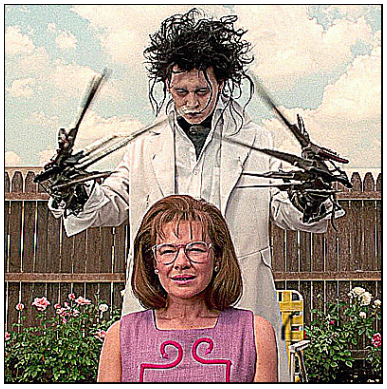
Figura 6: Filtro Offset

```
                        |-1 -1 -1|
float sharpen[3][3] =   |-1  9 -1|
                        |-1 -1 -1|

Para i de 0 a height - 3:
    Para j de 0 a width - 3:
        totalB = 0, totalG = 0, totalR = 0
        Para ii de 0 a 2:
            Para jj de 0 a 2:
                totalB = totalB + sharpen[ii][jj] * src[i+ii][j+jj].b
                totalG = totalG + sharpen[ii][jj] * src[i+ii][j+jj].g
                totalR = totalR + sharpen[ii][jj] * src[i+ii][j+jj].r
        dst[i+1][j+1].b = SAT(totalB)
        dst[i+1][j+1].g = SAT(totalG)
        dst[i+1][j+1].r = SAT(totalR)
```

Un ejemplo de este filtro se incluye en la figura 7.

(a) Imagen original



(b) Imagen con filtro Sharpen

Figura 7: Filtro Sharpen

2. Implementación

En esta sección nos dedicaremos a presentar y explicar nuestras propuestas para las implementaciones de los 4 filtros en lenguaje assembly x86 utilizando SSE.
En la sección de experimentación y análisis se presentarán algunas (pequeñas) modificaciones a las implementaciones que se incluyen a continuación, para mostrar distintos aspectos del funcionamiento del procesador. Decidimos explicar con detalle sólamente el código de las implementaciones que mejor resultaron en términos de tiempo de ejecución, ya que las modificaciones que se introducen para experimentar son pequeñas y no modifican sustancialmente el concepto detrás del ”armado” de cada implementación con SIMD de cada filtro. Para los filtros *Manchas, Cuadrados y Offset* en cada iteración del loop, realizamos 4 iteraciones de procesado, para aprovechar el concepto de *loop unrolling*. Por cuestiones de legibilidad del código y por cómo fue implementado el mismo para el filtro *Sharpen*, ésta técnica no fue aplicada en el código de éste último. Aclaramos entonces, que este detalle técnico no será incluido en la explicación de los filtros que viene a continuación, para evitar repetición en la descripción (que nada aporta).

Implementaciones en C

Las implementaciones en C provistas por la cátedra no serán explicadas ya que son una transcripción casi directa del pseudocódigo antes incluido. El código puede ser consultado en los archivos del repositorio en *src/filters* que terminan con ”*_c.c*”. En general, estas implementaciones procesan cada píxel de manera individual mediante un doble ciclo anidado.

2.1. Filtro Cuadrados

Para cada píxel tomamos el cuadrado 4x4 de ese píxel, con esquina superior izquierda, y calculamos el máximo en cada uno de sus colores. Para ello, tomamos las primeras dos filas y con la instrucción `pmaxub` obtenemos el máximo byte a byte. Con este resultado aplicamos la misma instrucción contra la siguiente fila. Finalmente, se realiza lo mismo con la cuarta fila. Ahora lo que tenemos son 4 máximos en un registro, utilizando shifteos, como se describe la Figura 10 ,calculamos el máximo entre ellos. Por último escribimos el resultado en el píxel destino. Como queremos un marco negro de 4 píxeles, esta iteración lo tiene en cuenta y no realiza el calculo para aquellos píxeles que están en el marco.

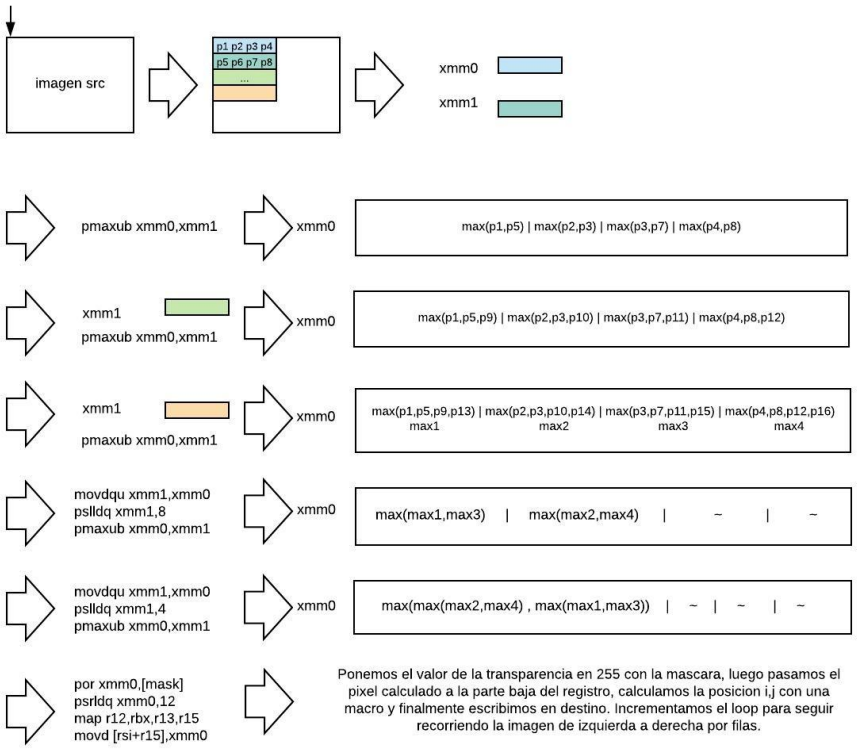


Figura 8: Descripción de cuadrados

2.2. Filtro Manchas

Si bien el pseudocódigo para este filtro es bastante simple y directo, la implementación en assembly aprovechando la extensión SIMD, lleva a un código más extenso y que requiere algo de explicación.

Ya que hay que procesar todos los píxeles, la idea detrás del código es iterar **por filas**, trayendo de memoria 4 píxeles por iteración. Estando dentro del ciclo, necesitamos calcular los 4 *tonos* necesarios para procesar cada uno de los 4 píxeles. Arrancamos calculando los 4 jj: utilizando un pequeño ciclo, acumulamos en $xmm0$ a $j\,\%n$ convertido a float, para los 4 j de la presente iteración (ya que estamos procesando de a 4 píxeles, para un j dado, necesitaremos $j, j+1, j+2, j+3$ para los correspondientes jj). Las conversiones a float se realizan con la instrucción *cvtdq2ps* sobre el registro $xmm0$ cuyo estado es

$$xmm0 == ||j\,\%n|(j+1)\,\%n|(j+2)\,\%n|(j+3)\,\%n||$$

. El hecho de que el pequeño ciclo nombrado antes itere 3 veces y no 4, se debe a que la cuarta iteración se realiza a mano para evitar un shift demás (cada iteración realiza la operación div, guarda en la parte baja de $xmm0$, y lo shiftea a izquierda 4 bytes). Esta implementación busca utilizar la cantidad mínima de operaciones de conversión ya que son muy costosas en términos temporales. En este punto nos queda dividir por n y multiplicar por 2π. Ésto se realiza con una instrucción *mulps* con el registro $xmm10$, que es "armado" antes del doble ciclo, para que contenga a $2\pi/n$ en cada uno de sus 4 espacios de dword. Ésto nos evita operaciones aritméticas demás. Nos encontramos en el siguient estado:

$$xmm0 = ||(j\,\%n)*(2\pi/n)|((j+1)\,\%n)*(2\pi/n)|((j+2)\,\%n)*(2\pi/n)|((j+3)\,\%n)*(2\pi/n)||$$

Donde cada componente es un float de precisión simple.
Ahora el código procede a aplicar la función coseno a cada componente del registro $xmm0$. Debido a que no contamos con instrucciones SIMD que apliquen funciones trigonométricas de manera "vectorial", utilizamos la instrucción $fcos$ de la fpu individualmente en cada componente de $xmm0$. Esto se realiza con un pequeño ciclo que aplica $fcos$, guarda el resultado en $xmm2$, shiftea el mismo 4 bytes a la izquierda, y shiftea $xmm0$ 4 bytes a la derecha para dejar disponible a la siguiente componente de $xmm0$ para procesar. Igual que antes y para evitar shiftear una vez demás, la última iteración se realiza 'a mano'. Utilizando la instrucción $pshufd$ reordeno a $xmm2$ en $xmm0$, de manera que quede en el siguiente estado:

$$xmm0 = ||cos((j\,\%n)*(2\pi/n))|cos(((j+1)\,\%n)*(2\pi/n))|cos(((j+2)\,\%n)*(2\pi/n))|cos(((j+3)\,\%n)*(2\pi/n))||$$

Así quedan armados los 4 jj en el registro $xmm0$. El código ahora continua calculando los 4 ii en $xmm2$. Ésto se realiza de forma análoga al cálculo anterior, pero es algo más simple debido a que, como estamos dentro de una misma columna, ii vale lo mismo para los 4 píxeles. Ahora se multiplican componente a componente, utilizando la instrucción $mulps$, los registros $xmm0$ y $xmm2$, se multiplica el resultado por 50 y se resta 25, también componente a componente utilizando $mulps$ nuevamente y $subps$. Finalmente convertimos el resultado a enteros de 32 bits en el registro $xmm0$, utilizando la instrucción de conversión $cvtps2dq$. Recordando los nombres utilizados en el pseudocódigo de la sección 1.4.2, el estado de $xmm0$ es ahora

$$xmm0 = ||tono_0|tono_1|tono_2|tono_3||$$

La única tarea que resta ahora es sumar estos 4 tonos calculados a los 4 píxeles que trajimos de memoria. Pero esto no es tan simple, ya que los tonos son enteros signados que ocupan 32 bits, y los píxeles consisten en componentes de 8 bits no signadas. Sabemos por el pseudocódigo y por propiedades del seno y coseno, que los tonos son números comprendidos entre -75 y 25. Por otro lado las componentes de los píxeles van entre 0 y 255. Los resultados intermedios de las cuentas "entran" en words, o sea espacios de 16 bits. La estrategia es entonces "packear" los tonos a tamaño word, y "unpackear" las componentes de los píxeles a tamaño word. El código empieza ésto con la instrucción $packssdw$ para que en el registro $xmm8$ queden los siguientes tonos:

$$xmm8 = ||tono_0|tono_1|tono_2|tono_3|tono_0|tono_1|tono_2|tono_3||$$

Donde todos los tonos son words signados. Utilizando una instrucción de *shuffle*y otra de *blend*, lelgamos a:

$$xmm8 = ||0|tono_1|tono_1|tono_1|0|tono_0|tono_0|tono_0||$$

De manera que la parte más alta tenga los datos necesarios para procesar el segundo píxel, y la parte baja los necesarios para el primer píxel, que es el orden en el que llegarán de memoria. Las posiciones que están en 0 corresponden a las componentes de transparencia. Por otro lado se guardaron en $xmm1$ los 4 píxeles traidos de memoria, de manera que:

$$xmm1 = ||a3|r3|g3|b3||a2|r2|g2|b2||...|g0|b0||$$

Utilizando la instrucción $punpcklbw$ y el registro $xmm12$ que está con todos sus bits en 0, logramos tener en $xmm3$:

$$xmm3 = ||a1|r1|g1|b1||a0|r0|g0|b0||$$

donde todas sus componentes son words no signados. Como sus valores están entre 0 y 255, puedo considerarlos como words signados positivos. Ahora está todo listo para sumar $xmm3$ con $xmm8$ utilizando $paddw$. Lo que quiero hacer ahora es eliminar los valores negativos. El código realiza ésto con la instrucción $pcmpgtw$, que se fija *word* a *word* si son mayores a 0 y guardando el resultado en $xmm11$. Si son mayores a 0, guarda 1's en el correspondiente word de $xmm11$, y 0 sino. Con una instrucción $pand$ guardamos en $xmm3$ los resultados que ahora contienen 0 si fueron iguales o menores que 0, y el resultado normal sino.
Análogamente, el código realiza lo ya explicado para los píxeles 2 y 3, y para los tonos 2 y 3, guardando el resultado en *words* positivos (análogamente a $xmm3$) en el registro $xmm6$. De esta manera, $xmm3$ tiene los píxeles (con sus coordenadas en tamaño word) 0 en la parte baja y 1 en la parte alta, y el registro $xmm6$ los píxeles 2 en la parte baja y 3 en la parte alta. Sus componentes están entre 0 y 280 (255 + 25). Realizamos finalmente un $packuswb$ que realiza el *pack* en modo saturado de *word* a *byte*, dejando en $xmm3$ los píxeles ya procesados, listos para ser mandados al array de píxeles de la imagen destino.

2.2.1. Algunas consideraciones para el filtro Manchas

Al programar la versión SIMD de este filtro se tuvo la precaución de evitar accesos **innecesarios** a memoria, utilizar la menor cantidad posible de *converts*, alinear los datos en la sección *rodata* y de minimizar la cantidad de operaciones aritméticas necesarias por píxel. Ésto se realizó de la siguiente manera:

- Usando la directiva *ALIGN* cuando se declaran e inicializan los datos en *.rodata*.
- Cargando los datos estáticos en registros antes del ciclo, en lugar de leerlos de memoria en cada iteración.
- Utilizamos la menor cantidad de *converts*: exactamente 4 instrucciones de conversión.
- Dejando calculado en un registro el vector cuyas componentes son $2\pi/n$, en lugar de calcularlo en cada iteración.

2.3. Filtro Offset

Este filtro es tal vez el más sencillo de implementar utilizando SIMD. Como ya vimos en el pseudocódigo, este filtro ”desfasa” las componentes `R`, `G` y `B`, produciendo estelas de dichos colores. La idea tras nuestro código en lenguaje assembly es procesar de a 4 píxeles a la vez, realizando 3 lecturas de memoria por iteración: una para traer los píxeles de los cuales se extraerán las componentes azules, otra para traer los píxeles de los cuales se extraerán las componentes verdes y otra para las rojas. En el pseudocódigo, dentro del doble ciclo anidado, estas locaciones son `src[i+8][j]`, `src[i][j+8]` y `src[i+8][j+8]`. Lo único que resta hacer ahora es extraer las componentes correctas de cada registro usado para guardar los píxeles en las direcciones mencionadas. Se utilizaron los registros $xmm1$ (de aquí extraeremos las componentes azules), $xmm2$ (de aquí extraeremos las componentes verdes) y $xmm3$ (de aquí extraeremos las componentes rojas). Mediante dos instrucciones *pblendvb*, utilizando máscaras (a cargarse en $xmm0$, según lo requiere la instrucción) cargadas antes del ciclo en los registros $xmm6$ y $xmm7$, conseguimos guardar en $xmm1$ los píxeles ya procesados.

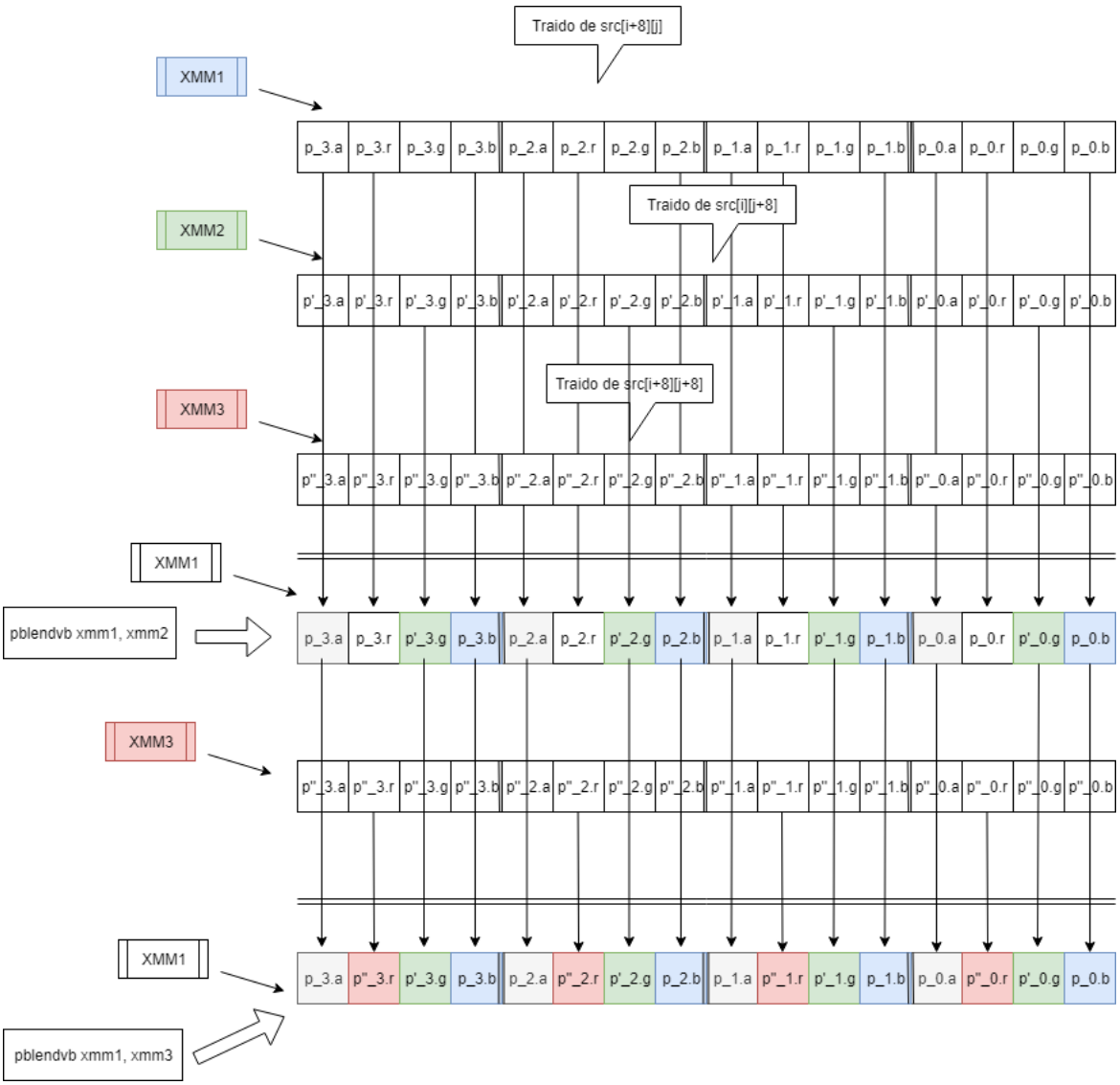


Figura 9: Implementación asm Offset

El diagrama de la figura 9 ilustra el uso de la instrucción $pblendvb$ para armar en $xmm1$ los 4 píxeles ya procesados, listos para devolver al array de píxeles dst.
El filtro requiere además un borde negro de 8 píxeles de espesor, en los 4 lados de la imagen. Ésto es logrado con 2 pequeños ciclos previos al ciclo principal del filtro offset propiamente dicho: el primero

”pinta” los bordes superior e inferior en forma simultánea. El segundo se encarga de pintar los bordes izquierdo y derecho de forma simultánea. Cuando implementamos este filtro, consideramos la posibilidad de escribir todo el código -filtro + bordes- en el mismo ciclo, pero decidimos hacerlo en forma dividida (como acabamos de explicar) para que el código resulte más legible, y además para evitar una cantidad no despreciable de saltos condicionales en cada iteración, para decidir qué hacer con los píxeles correspondientes.
Igual que en los otros filtros, aprovechamos la directiva `ALIGN 16`, para traer las máscaras con la instrucción $movdqa$ desde memoria a registros xmm de forma alineada y por lo tanto más rápidamente.

2.4. Filtro Sharpen

El filtro sharpen tiene el propósito de realzar las diferencias entre los pixeles, acentuando todo cambio de color en la imagen. Es por esto que para calcular cada pixel de la imagen destino usaremos el pixel correspondiente en la imagen original y los pixeles que lo rodean. Diremos entonces que el pixel correspondiente es el pixel central y que los ocho pixeles que lo rodean son el resto. Para cada color el pixel destino cumplira la siguiente propiedad: $pixelDestino = pixelCentral * 9 - pixelesPerifericos$. Este cálculo no se puede realizar para calcular el valor de los píxeles de los bordes de la imágen destino. Es por esto que los dejaremos pintados de negro. En todos los píxeles, queremos que la cuarta componente, el alfa o coeficiente de transparencia, siempre tenga un valor de 255, o sea el máximo.

Como cada pixel está compuesto por cuatro componentes de un byte cada una, el píxel en si pesa 32 bytes y por lo tanto, podemos almacenar hasta cuatro píxeles en un registro de 128 bits.

En nuestra implementación del filtro sharpen nos propusimos obtener el resultado para dos píxeles de la imagen destino al mismo tiempo. Dadas las coordenadas (i, j) e (i, j+1) de un par de píxeles destino que queremos calcular, para tres filas diferentes levantamos cuatro píxeles contiguos obteniendo así doce píxeles de la imagen original con los siguientes índices:

Cuadro 1: Píxeles a procesar de la imágen fuente en la iteración i, j

Registro	**Píxel 1**	**Píxel 2**	**Píxel 3**	**Píxel 4**
xmm0	(i - 1, j - 1)	(i - 1, j)	(i - 1, j + 1)	(i - 1, j + 2)
xmm1	(i, j - 1)	(i, j)	(i, j + 1)	(i, j + 2)
xmm2	(i + 1, j - 1)	(i + 1, j + 2)	(i + 1, j + 2)	(i + 1, j + 2)

Nótese que los 9 píxeles izquierdos son los píxeles necesarios para calcular el píxel destino izquierdo y los 9 píxeles derechos son los píxeles necesarios para calcular el pixel destino derecho.

Habiendo cargado los valores de los píxeles en registros de 128 bits, nuestro objetivo es empezar a hacer sumas y restas entre los registros para lograr el resultado deseado. Quisiéramos poder obtener los valores finales para cada componente. Pero como en los resultados intermedios podrían crecer mucho, tendremos a bien aumentar la precisión con la cual representamos los números enteros. Subiremos de 8 bits a 16 bits, pasando de representar cada componente con un byte a hacerlo con un word. Para esto usamos la operación PUNPCKLBW (packed unpack low byte to word) y la operación PUNPCKHBW (packed unpack high byte to word). Entonces nuestra matriz de pixeles queda repartida en seis registros de la siguiente manera:

Cuadro 2: Píxeles desempaquetados

	Columna 1	**Columna 2**	**Columna 3**	**Columna 4**
	Low	**High**	**Low**	**High**
Fila i-1	xmm0		xmm3	
Fila i	xmm1		xmm5	
Fila i+1	xmm2		xmm6	

Cada registro contiene dos píxeles almacenados con precisión de words en cada componente. Al haber representado nuestros datos de esta manera, nuestra implementación difiere de la implementación de la cátedra, que castea cada byte a float. Dado que no existe representación exacta para los enteros al usar floats, y que usando words tenemos asegurado que no haya overflow, consideramos que nuestra implementación es adecuada y nos evita problemas numéricos.

Para calcular el valor del píxel destino izquierdo, es necesario obtener el resultado de la cuenta $pixelDestino = pixelCentral * 9 - pixelesPerifericos$. Comenzaremos realizando las operaciones necesarias para poder obtener el valor de la suma de los píxeles periféricos, luego obtendremos el valor del píxel central multiplicado por nueve y finalmente realizaremos la resta entre ambos valores, obteniendo así el valor deseado.

A la hora de realizar sumas y restas de píxeles aprovecharemos que sus componentes siempre siguen el mismo orden y nuestra estrategia estará basada en utilizar este "alineamiento" de los valores, eligiendo con qué registros operar en base a qué píxeles contengan estos registros.

Para calcular la suma de los píxeles periféricos, comenzaremos observando que los ocho píxeles que nos interesan están distribuidos en 6 registros diferentes.

Los registros $xmm0$ y $xmm2$ contienen píxeles que nos interesan tanto en el high como en el low. Sumamos ambos almacenando el resultado en el registro $xmm7$. Digamos que $xmm7$ contiene ahora píxeles llamados [A B]. Copiamos el contenido de $xmm7$ a $xmm8$, realizamos un right shift de 8 bytes en $xmm8$, de manera que ahora contiene [0 A] y almacenamos la suma de $xmm7$ y $xmm8$ en $xmm7$ el cual pasa a contener [A A+B]. Nótese que A+B es un píxel equivalente al total de la suma de todos los píxeles originalmente contenidos por $xmm0$ y $xmm2$.

A continuación, nuestro objetivo es obtener el valor de la suma de los cuatro píxeles restantes. En este paso aprovecharemos que todos estos píxeles están en el low de un registro. Limpiamos $xmm8$ y le sumamos consecutivamente los registros $xmm3$, $xmm5$, $xmm6$ y $xmm1$. Siendo los primeros tres los registros cuyo low almacena los píxeles periféricos del borde derecho y el low de $xmm1$ siendo el píxel del borde izquierdo y la fila central.

Almacenamos en $xmm7$ la suma de $xmm7$ y $xmm8$ con lo cual el low de $xmm7$ es equivalente a la suma de los píxeles periféricos. Hemos logrado así obtener uno de los operandos para nuestro cálculo principal.

A continuación cargamos en $xmm10$ una máscara, definida en words, que contiene nueves en el high, exceptuando por el número más significante, que está seteado en cero. Utilizando $pmullw\ \ xmm1,\ xmm10$ obtenemos en el high de $xmm1$ el valor del píxel central multiplicado por nueve, que es el segundo operando necesario para nuestro cálculo principal.

Realizamos la resta correspondiente entre ambos operandos con psubusw y almacenamos nuestro resultado, el valor del pixel destino izquierdo, en el low del registro $xmm1$.

Para calcular el valor del píxel central derecho se realizan exactamente las mismas operaciones, pero intercambiando low por high en cada caso y almacenando el valor del píxel destino derecho en el high del registro $xmm5$.

Para escribir en memoria ambos píxeles, primero necesitamos que ambos estén en un mismo registro con precisión de byte en cada componente. Logramos ambas cosas con un solo comando: $packusb\, xmm1,\ xmm5$. Como este comando nos deja $xmm1$ = [0 pi pd 0], realizamos un shift derecho de 4 bytes para mandar ambos píxeles resultado al low de $xmm1$. Con esto estamos listos para escribir los píxeles resultado en memoria

3. Resultados y Análisis

3.1. Introducción

3.1.1. Objetivos

Como ya mencionamos al principio del presente informe, el objetivo de este trabajo incluye, además de la implementación de los filtros en lenguaje assembly usando SIMD propiamente dicha, un análisis que permita dar evidencia acerca de la mejora (o no) en performance de este tipo de implementaciones con respecto a otras un poco más *naive* que no aprovechen la parte de la arquitectura que implementa el modelo SIMD.

3.1.2. Overview de la experimentación

Durante la etapa de programación, hemos implementado los filtros aprovechando el modelo SIMD, y teniendo en cuenta los aspectos de la arquitectura que pueden hacer que un código se ejecute más rápidamente. Con el fin de verificar que ésto sea así, hemos realizado variantes puntuales a este código original, llegando así a 4 variantes por filtro:

- Iteración por columnas en lugar de filas, para ver el impacto de la mal utilización de la caché en el filtro.
- Utilización de máscaras no alineadas.
- Leer las máscaras desde memoria en cada iteración, en lugar de guardarlas al principio en un registro para evitar lecturas a memoria innecesarias.
- No utilizar *loop unrolling*.

Por otra parte hemos realizado una 4 variante por filtro, que modifica el código original para explorar el *loop unrolling* y ver si éste logra un código con mejor performance.
Por último contamos con la implementación en `C` provista por la cátedra.

3.1.3. Set de prueba

Las pruebas serán realizadas sobre la siguiente imagen:

Figura 10: Imagen de prueba para las experimentaciones.

Parte de la experimentación a realizar, implica ver como cambian las métricas utilizando distintos tamaños de imagen, por lo cual el set de pruebas se completa con otras 16 versiones de la imagen de la figura, con diferentes resoluciones. Las resolucones van de 64x64 píxeles a 1024x1024 píxeles, aumentando de a 64 píxeles (en ancho y alto).

3.1.4. Métricas

Antes de la experimentación debemos establecer métricas que guien nuestro análisis para determinar si una implementación es mejor que otra. Utilizaremos las siguientes:

- Cantidad de ticks de clock por aplicación de cada filtro.
- Error cuadrático medio de la imagen resultante con respecto a la imagen resultante de la aplicación de la implementación en C de la cátedra (es decir, tomaremos a ella como *source of truth*).

La métrica que refiere al tiempo, es decir la cantidad de ticks de clock por aplicación de filtro, necesita un poco más de justificación y algunos cuidados extra (que la métrica del error no necesita).
La forma de medir los ticks tomados en una aplicación de filtro, se realizan mediante la instrucción de assembler `rdtsc`, que permite obtener el valor del Time Stamp Counter (TSC) del procesador. Éste es un registro que se autoincremente en uno con cada ciclo del procesador. La diferencia entre el valor del `TSC` al fin de la aplicación y al comienzo, nos dará el número buscado. Debemos tener en cuenta que esta diferencia no será siempre igual para todas las invocaciones debido a algunas problemáticas que pueden presentarse. Las principales son:

- La ejecución puede ser interrumpida por el *scheduler* para realizar un cambio de contexto, lo que implica que el valor obtenido resulta mucho mayor al que se hubiera obtenido si no hubiera habido interrupciones.
- Los procesadores modernos varían su frecuencia de reloj, por lo que la forma de medir ciclos cambia dependiendo del estado del procesador.
- El comienzo y fin de la medición deben realizarse con la suficiente exactitud como para que se mida solamente la ejecución de los filtros, sin ser afectada por ruidos como la carga o el guardado de las imágenes.

Proponemos al promedio muestral como un estimador razonable dado que la distribución de tiempos de corrida no manifiesta un patrón claro. Con lo cual el promedio nos dará una buena aproximación para todos los casos.

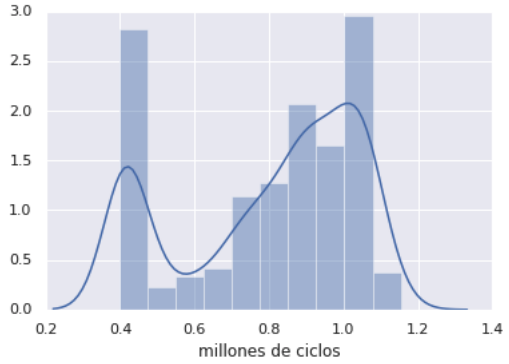


Figura 11: Distribución de las muestras tomadas.

El framework provisto por la cátedra provee, en el archivo `tp2.c` las funciones `MEDIR_TIEMPO_START` y `MEDIR_TIEMPO_END`, para poder realizar las mediciones de tiempo. Éstas mediciones se realizan justo antes y justo después de la invocación de cada filtro, para cubrir la problemática mencionada en el tercer item anterior. Para cubrir las otras dos, decidimos realizar cada medición un número ”grande” de veces, que asegure, bajo el amparo de la ley de los grandes números, que la media muestral converja a la media real. Para elegir la cantidad de iteraciones correcta, que en efecto nos brinde una media representativa (y por lo tanto habilitada para realizar comparaciones), realizamos un pequeño experimento para ver a partir de que número se estabilizan la media muestral y la desviación estandar.

En la figura 11 puede observarse el experimento mencionado. Se aplicó el filtro *Sharpen*, para la imagen de prueba, en resolución 128x128 píxeles, con repeticiones que fueron desde 0 a 1000. Para cada

Figura 12: Convergencia de la media muestral.

Figura 13: Convergencia de la desviación estándar.

conjunto de iteraciones, se calculó la media del tiempo medido en *ticks*, y esos valores fueron ploteados contra los números de cantidades de iteraciones. Como fue explicado antes, esperábamos ver que la media muestral se estabilizara para valores altos en la cantidad de iteraciones. El gráfico muestra que en efecto, ésto sucede. A partir de 200 iteraciones, la media muestral comienza a estabilizarse, y ésto se sigue acentuando conforme crece la cantidad de repeticiones. Con este criterio, decidimos realizar los experimentos que siguen, con 500 iteraciones por aplicación. Este número nos da un *tradeoff* razonable entre tiempo de cómputo de los experimentos, y fiabilidad de los resultados.

3.1.5. Equipo utilizado

Las mediciones que llevaremos a cabo en los siguientes experimentos serán realizadas sobre equipos que constan de las siguientes prestaciones:

- Procesador Intel(R) Xeon(R) CPU E5-2670 v3 a 2.30GHz, con 251 Gb de RAM y utilizando Ubuntu 14.04.1 LTS.

3.2. Experimentación

3.2.1. Diferentes experimentaciones en ASM

La idea de este primer experimento es la de quedarnos con la mejor experimentación en ASM, para luego compararla con la versión de la cátedra en C. Debido a que éstas solo introducen cambios en la manera en la que se leen datos de memoria, y no en la que se realizan los cálculos numéricos, solo mediremos la métrica relacionada con el tiempo, para quedarnos con la "ganadora". Realizaremos el experimento para cada uno de los filtros, ploteando tiempos con respecto a tamaño de la imagen de prueba, realizando 500 iteraciones por aplicación de filtro en cada imagen del set de prueba. Los resultados se mostrarán y analizarán para las 3 máquinas con las cuales estamos trabajando (las mencionadas en la sección 3.1.5).

Repasamos una vez más las características alternativas a la implementación principal (explicada en la sección 2) que serán testeadas contra esta última. Todas las variantes "remueven" la feature mencionada a la implementación original, dejando las otras intactas.

- **Variante 1**. Recorrer la imagen por columnas, para no aprovechar la memoria *cache*.
- **Variante 2**.Cargar las máscaras sin utilizar la directiva `ALIGN 16`.
- **Variante 3**.Leer cada máscara en cada iteración, en lugar de cargarlas en registros antes del loop principal.
- **Variante 4**.No realizar *loop unrolling*.

3.2.1.1. Hipótesis

Ya que el tamaño de la imagen va a ser ploteado en el eje horizontal con la cantidad de píxeles correspondiente a la resolución, la progresión de tiempo de cómputo (medido en ticks) de cada filtro en cada implementación, debería ser aproximadamente lineal (Esta aclaración se debe a que otros gráficos de otros experimentadores pueden verse con tendencia cuadrática, probablemente debido a que plotearon contra la resolución, en lugar de cantidad de píxeles). Por otro lado, esperamos que la implementación principal sea la más rápida, ya que es la que más explota las correspondientes características de la arquitectura de los equipos. Ésto se debe a varios factores:

- Al usar la directiva `ALIGN 16`, el ensamblador agregará suficiente padding para que las máscaras caigan en memoria de manera alineada. De esta manera los accesos a los mismos serán más rápidos que si estuvieran desalineados. Ésto se aprovecha en los filtros (en general) con la instrucción `movdqa` en lugar de `movdqu`.
- Al cargar las máscaras a registros antes del loop principal, nos ahorramos una cantidad considerable de accesos a memoria, que crecen con el tamaño de la imagen. Debido a que el hardware de los registros suele estar implementado con memorias SRAM (en contraposición a DRAM en la memoria principal), las lecturas son mucho más rápidas. Ésto puede consultarse en cualquier libro de organización de computadoras, en secciones que hablen de "Jerarquía de Memoria".
- El recorrido de píxeles se realiza por filas y **no** por columnas, para aprovechar la utilización de memoria *cache*. Debido al funcionamiento de la misma, cada vez que realizamos una lectura de memoria (para leer los píxeles), se traen a memoria no solo los píxeles pedidos, sino también los que siguen a continuación en memoria (recordar que los mismos están guardados en memoria, según la codificación BMP, como un arreglo de dos dimensiones, es decir primero la primer fila, a continuación la segunda, y así hasta la última). Ésto nos asegura *hits* en las lecturas a *cache* de los píxeles subsiguientes al último píxel leído. Cuando no haya más píxeles en la memoria *cache* por ser aprovechados, se producirá uun *miss*, y se volverán a cargar en *cache* un nuevo tramo de píxeles que podrán ser aprovechados de la misma manera. Todo ésto se basa en que las lecturas a memoria *cache* son más rápidos que las lecturas a memoria principal debido a cómo están implementadas electrónicamente ambas memorias. Si el procesado de píxeles se hiciera por columnas, en

cada iteración realizaríamos un *miss*, desaprovechando los píxeles presentes en la memoria *cache*. Cuando terminemos de recorrer la columna, ya no tenemos asegurado que los píxeles siguientes al ya procesado en la presente fila, sigan estando disponibles en la memoria *cache*.

- *Loop unrolling*: El objetivo de aplicar esta técnica es el de reducir las instrucciones de control y las instrucciones de salto para reducir *branch penalties* y reducir latencias asociadas a *delays* en accesos a memoria. Si bien una desventaja de la técnica es el crecimiento del tamaño del código, en las implementaciones de los filtros, éste no fue significativo (en el filtro offset, ésto se tradue a 113 líneas en la alternativa sin *loop unrolling* contra 150 en implementación principal).

La implementación que omite el uso de la directiva `ALIGN 16`, no debería tener un tiempo de cómputo mucho mayor a la implementación principal, ya que de todas maneras las máscaras son cargadas después en registros (previo al loop) y leídas desde ahí en accesos posteriores.

La implementación que no carga las máscaras en registros, debería tener un tiempo apreciablemente mayor al de la implementación principal, porque está agregando una cantidad proporcional al tamaño de la imagen de accesos a memoria.

La implementación que no utiliza *loop unrolling*, debería ser más lenta, por lo ya explicado, pero creemos que ésto no va a ser suficiente como para ser apreciado o que valga la pena la escritura de código extra que requirió.

Finalmente, la implementación que recorre la imagen por columnas, debería ser considerablemente más lenta, por lo ya explicado acerca del funcionamiento de la memoria *cache*.

3.2.1.2. Resultados del experimento

A continuación exponemos los resultados del experimento presentado, para cada uno de los filtros.

Cuadrados y offset

Observamos que existe una clara distinción entre la implementación que recorre por columnas y el resto de las implementaciones. A continuación podemos ver gráficos de comparación de las distintas implementaciones:

Nótese que al mirar los tiempos de corrida sin tener en cuenta la implementación por columnas no es posible diferenciar los tiempos de corrida de las otras implementaciones. Concluimos a partir de estos resultados que el verdadero problema a resolver para mejorar la performance es el aprovechamiento de la memoria caché.

En esta misma sección de resultados mostraremos los análisis temporales de las distintas implementaciones de offset, dado que hemos observado el mismo patrón en los tiempos de corrida.

El poder replicar resultados con distintos experimentos le da sustento empírico a las observaciones realizadas.

Sharpen

En el caso del filtro sharpen, observamos la misma tendencia que en los filtros anteriores. Hemos decidido mostrarlo en una sección diferente a causa de que realizamos un subconjunto de las experimentaciones realizadas para los otros filtros. Esto es debido a que, por razones implementativas, los cambios que planteaban las implementaciones no afectaban realmente el funcionamiento del filtro sharpen, con lo cual estas implementaciones no hubieran aportado datos nuevos en términos experimentales.

A continuación se encuentran los gráficos de costos temporales de las distintas implementaciones del filtro sharpen:

Una vez más, observamos que el costo temporal de las implementaciones se ve dominado por el comportamiento de la caché.

Manchas

Las figuras 21 y 22 muestran los resultados por los tiempos de corrida del filtro manchas. Contrario a lo que venimos viendo (y prediciendo) en otros filtros, no encontramos diferencias marcadas entre las

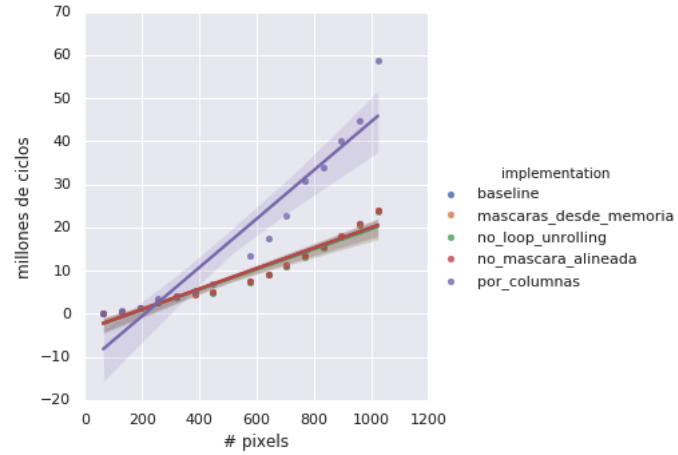


Figura 14: Cuadrados: tiempos de corrida.

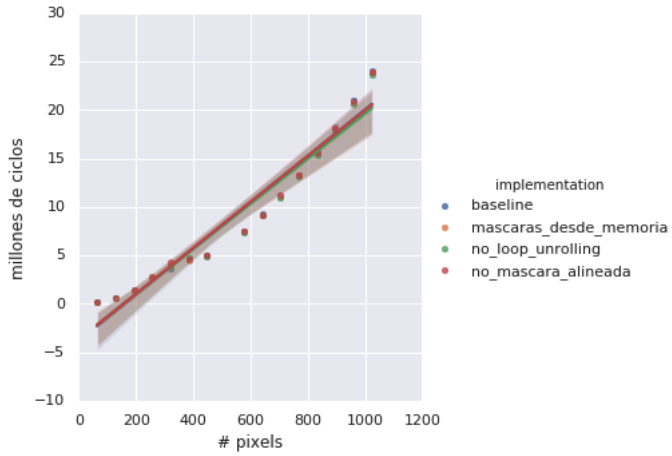


Figura 15: Cuadrados: tiempos de corrida sin implementación por columnas.

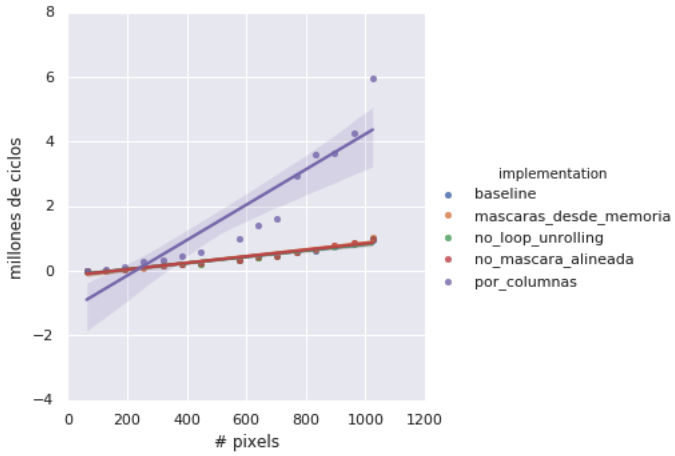


Figura 16: Offset: tiempos de corrida.

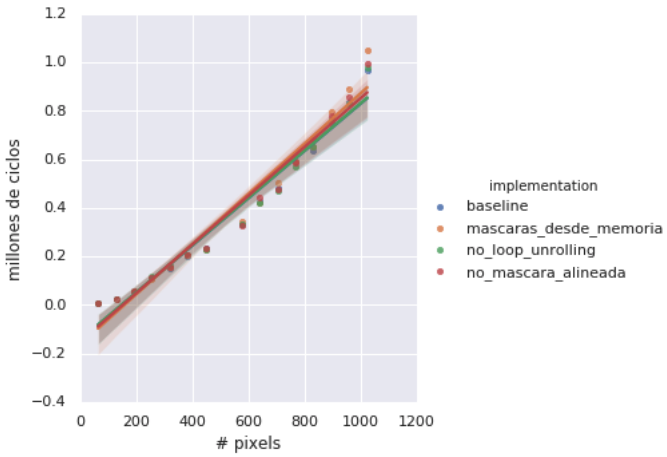


Figura 17: Offset: tiempos de corrida sin implementación por columnas.

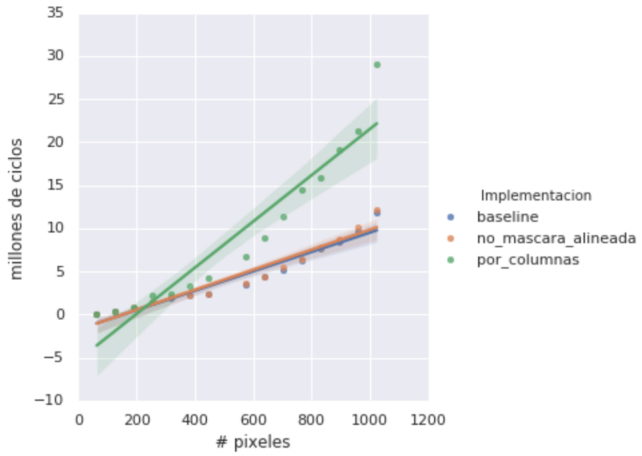


Figura 18: Sharpen: tiempos de corrida.

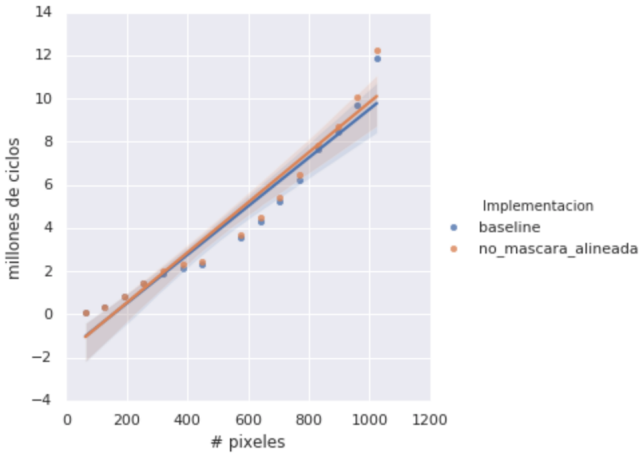


Figura 19: Sharpen: tiempos de corrida sin implementación por columnas.

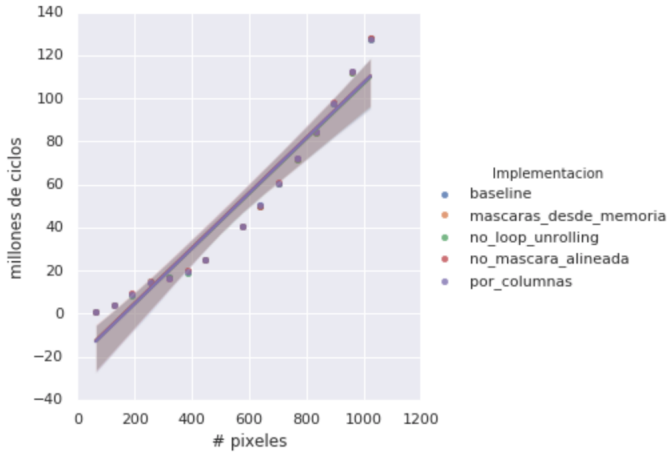


Figura 20: Manchas: tiempos de corrida.

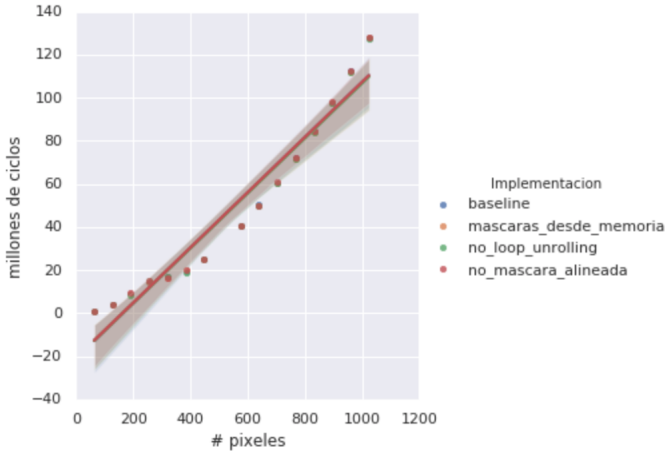


Figura 21: Manchas: tiempos de corrida sin implementación por columnas.

diferentes implementaciones. Como dijimos en nuestras hipótesis, esperábamos no encontrar diferencias tan significativas entre la implementación baseline, la implementación que carga las máscaras desde memoria, la que no utiliza la directiva ALIGN, y la que no realiza loop unrolling, y ésto en efecto fue así. Lo que llama la atención acá, es que la implementación que recorre la imagen por columnas, no muestra tiempos mucho mayores a sus competidoras. También se puede notar que en comparación con los demás filtros, el filtro manchas es el que más tarda, necesitando aproximadamente 130 millones de ciclos para la imagen de 1024 píxeles, cuando los demás filtros tardaron entre 2 y 30 millones de ciclos para sus mejores implementaciones para la imagen de 1024 píxeles. El hecho de que no haya diferencia entre la implementación por columnas que la implementación por filas, nos hace sospechar que aún cuando estamos recorriendo por filas, algo en el código está rompiendo la forma de utilizar la caché.

3.2.2. ASM vs C

Ahora que tenemos evidencia para argumentar que la implementación principal es la que brinda mejor performance en términos de tiempo de cómputo, procederemos a compararla con la implementación en código `C` provista por la cátedra. Ésta es compilada con máxima optimización (`o3`), y según puede verse en el código (disponible en la página de la materia), el procesado de píxeles se realiza de a uno por iteración.

Para este experimento, además de medir tiempos de ejecución, mediremos el error cuadrático medio presentado en la sección 3.1.4., especialmente debido al filtro Manchas, que utiliza cómputos en punto flotante y utilizando las funciones `fsin` y `fcos` de la FPU, que **no** son utilizados por la biblioteca `math.h` de la implementación en lenguaje `C`.

3.2.2.1. Hipótesis

Nuestra hipótesis para este experimento es que el tiempo de cómputo para la implementación en lenguaje assembly utilizando el modelo SIMD será considerablemente menor a la implementación en código `C`. En general, en las implementaciones utilizando SIMD, se procesan 4 píxeles por iteración (recordar que para el formato de imagen utilizado, un píxel ocupa 1 byte: 8 bits la transparencia, 8 para el rojo, 8 para el verde y 8 para el azul), ésto quiere decir que, en principio, nuestra implementación debería ser **al menos** cuatro veces más rápida que la implementación en lenguaje `C`. En general, si llamamos n a la cantidad de píxeles que entran en un registro SIMD, ésta implementación debería ser al menos n veces más rápida que aquella que procesa los píxeles de a uno por vez. Ésta aclaración la hacemos porque, en futuros experimentos propuestos (al final de este informe), proponemos realizar los filtros utilizando AVX-512 y AVX, que proveen registros de 512 bits y 256 bits, respectivamente (ésto equivale a procesar de a 16 y 8 píxeles por vez, respectivamente).

3.2.2.2. Resultados del experimento

A continuación exponemos los resultados del experimento presentado, para cada uno de los filtros.

Cuadrados, Sharpen y Offset

Puede observarse en las figuras 22, 23, 24 que las pendientes de las rectas de costos computacionales son marcadamente diferentes. Este es un patrón que se repite para todos los filtros.

Manchas

Este filtro, a diferencia de los otros, tiene la particularidad de tomar un parámetro n. Para verificar el impacto de este parámetro en el costo temporal de corrida comparamos los promedios muestrales para distintos valores de n y no encontramos relevancia estadística. Es por esto que para evaluar el costo temporal de las distintas implementaciones de este filtro dejaremos fijo un n y el procedimiento será idéntico al utilizado con los otros filtros.

En la figura 26 pueden verse los resultados de corrida del filtro manchas en asm con respecto a C. Coincidente con la hipótesis, los tiempos son menores. Sin embargo, los cálculos dicen que la implementación en C resultó ser 1,4 veces más lenta que la de asm, que es mucho menor a lo previsto (habíamos predicho una mejor de 4x). Creemos que ésto tiene que ver con el uso de la FPU para las operaciones de punto flotante de seno y coseno, y sus repetidos accesos a memoria para transportar datos al stack de la FPU.

3.2.2.3. *

Análisis de resultados asm vs c A partir de los tiempos de cómputo registrados para implementaciones de ambos lenguajes concluimos que el costo de los algoritmos estudiados es lineal en función de la cantidad de píxeles totales que contiene la imagen y que, por lo tanto, depende fuertemente del costo asociado al procesamiento de cada píxel.

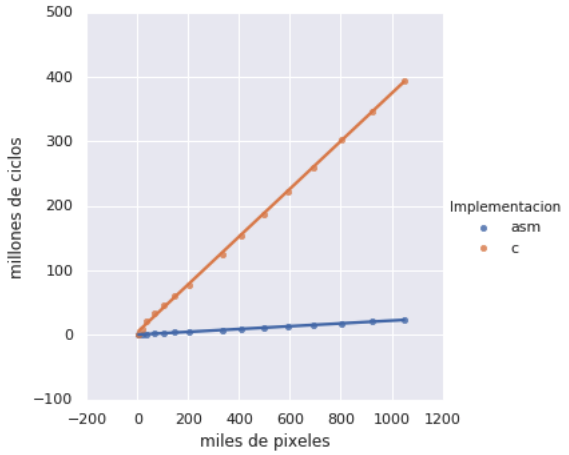


Figura 22: Cuadrados: tiempos de corrida asm vs c.

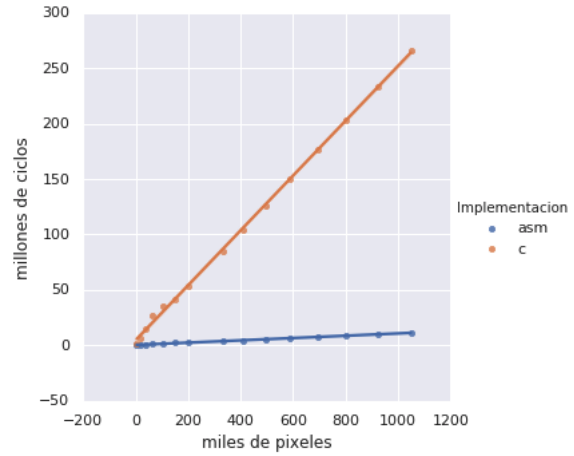


Figura 23: Sharpen: tiempos de corrida asm vs c.

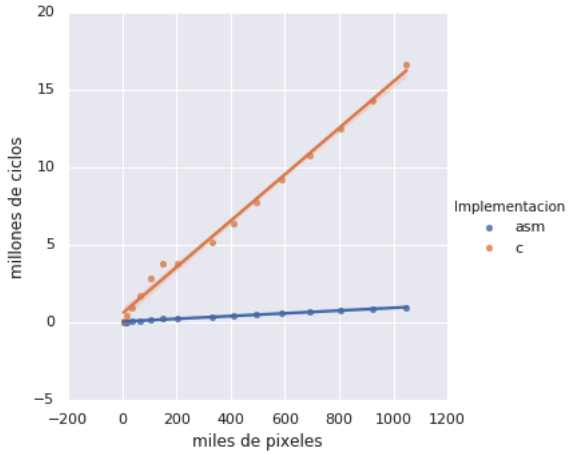


Figura 24: Offset: tiempos de corrida asm vs c.

3.2.3. Análisis de calidad de las imágenes

Para entender si nuestras implementaciones en assembler con SIMD son válidas, proponemos comparar las imágenes post filtro utilizando la métrica del error cuadrático medio. Ésta consiste en restar píxel a píxel, componente a componente, cada valor numérico para obtener un vector de errores. Cada elemento de este vector de errores es potenciado al cuadrado, con lo cual se garantiza signo positivo, se suman todos estos valores y finalmente se calcula la raíz cuadrada de esta suma.

3.2.4. Resultados de experimentaciones con error cuadrático medio

Los filtros Sharpen, Cuadrados y Offset realizan operaciones con enteros. Por lo tanto las imágenes resultado han manifestado valores idénticos y, equivalentemente, un ECM igual a cero.

En la figura 26 mostramos los resultados de las mediciones de ECM realizadas para el filtro Manchas, variando el tamaño de las manchas, parámetro al cual nos referiremos como manchas n:

Puede observarse que, dado un tamaño de manchas, el ECM es casi idéntico para distintos tamaños de imagen. En la figura 27 presentamos un gráfico en el cual se muestra el ECM promedio para un tamaño de manchas:

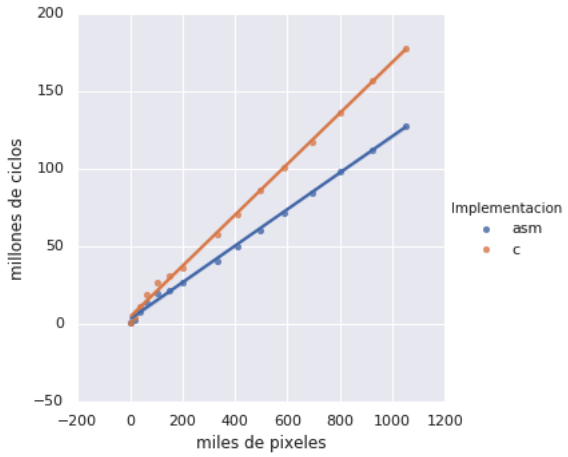


Figura 25: Manchas: tiempos de corrida asm vs c.

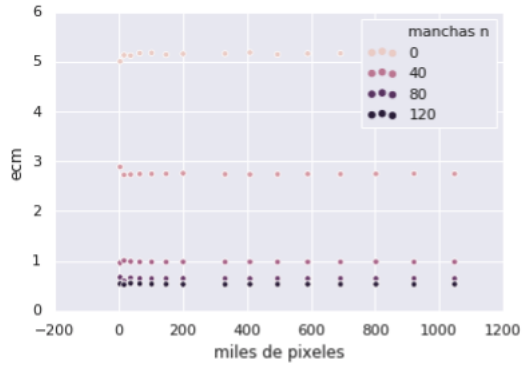


Figura 26: Manchas: ECM para distintos tamaños de imagen y distintos tamaños de mancha

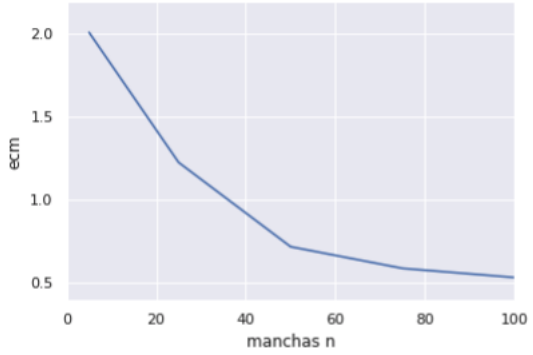


Figura 27: Manchas: ECM vs distintos tamaños de mancha

Lo que buscamos estudiar acá fue cuán ”malos”son los cómputos de las funciones fsin y fcos que provee la FPU con respecto a las implementaciones de la biblioteca math.h de la biblioteca standard de C. Era de esperar, según la documentación y diferentes fuentes, que la biblioteca de C iba a ser mucho más exacta. Así que ese resultado no fue sorpresa. Por otro lado, si es interesante notar que el error disminuye en forma cuadrática con respecto al tamaño de la mancha. Para tamaños grandes de mancha, los valores ii y jj tienden a 0, por lo que eso deja ver que la implementación electrónica de estas funciones tiene poco error para valores cercanos a 0.

4. Conclusiones y trabajo futuro

Los experimentos realizados brindan evidencia suficiente para afirmar que la utilización del modelo SIMD en la implementación de filtros para fotos, reduce drásticamente los tiempos de cómputo, en comparación con procesados *naive* que procesan un píxel por iteración. En general, nuestro informe muestra que es beneficioso utilizar las extensiones SIMD de la arquitectura x86 para procesamientos de archivos multimedia.

Por otro lado, quedó en evidencia el impacto en tiempo de cómputo que tienen accesos innecesarios a memoria, accesos desalineados, y en especial el mal uso de la memoria *cache*.

Finalmente, proponemos como trabajo futuro, implementar los filtros utilizando las extensiones AVX y AVX-512, para seguir explorando la relación numérica entre tiempo de cómputo utilizando SIMD y tiempo de cómputo sin utilizarlo, de acuerdo a la cantidad de píxeles que entran en un registro SIMD.