DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Gastón Máspero	131/17	gaston.maspero@gmail.com
Sebastián Bocaccio	287/18	sebastianbocaccio16@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.

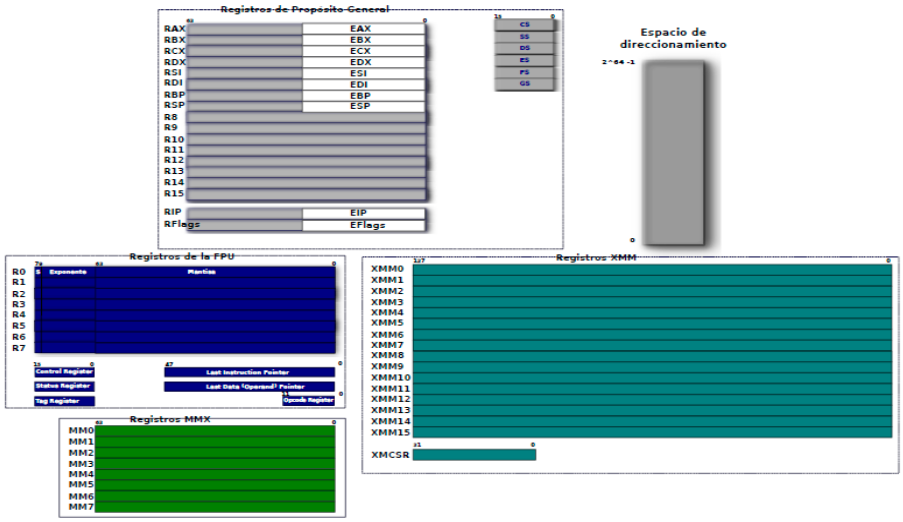
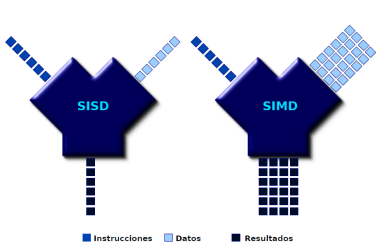
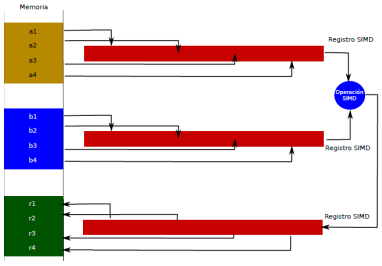


Figura 1: Arquitectura x86 con SSE

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 instrucción 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 vídeo), 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 "colorea 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 rescalados 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 ante última 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 filtros: *Zig-Zag*, *Ocultar* y *Descubrir*. En todos los filtros el byte de transparencia se setea en 255.

1.4.1. Zig-Zag

Este filtro genera un curioso 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	píxel destino dst_{ij}
$i \equiv 0(mod4)$	$(src_{ij-2} + src_{ij-1} + src_{ij} + src_{ij+1} + src_{ij+2})/5$
$i \equiv 1(mod4)$	src_{ij-2}
$i \equiv 2(mod4)$	$(src_{ij-2} + src_{ij-1} + src_{ij} + src_{ij+1} + src_{ij+2})/5$
$i \equiv 3(mod4)$	src_{ij+2}

Este filtro deja un marco blanco de 2 píxeles de espesor.

(a) Imagen 1 original

(b) Imagen 1 con filtro *Zig-Zag*

(c) Imagen 2 original

(d) Imagen 2 con filtro *Zig-Zag*

Figura 4: Filtro *Zig-Zag*

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

```
Para i de 2 a height - 3:
    Para j de 2 a width - 3:
        if( i%4 == 0 || i%4 ==2 )
            dst[i][j].b = (src[i][j-2].b + src[i][j-1].b + src[i][j].b
                           + src[i][j+1].b + src[i][j+2]).b / 5
            dst[i][j].g = (src[i][j-2].g + src[i][j-1].g + src[i][j].g
                           + src[i][j+1].g + src[i][j+2]).g / 5
            dst[i][j].r = (src[i][j-2].r + src[i][j-1].r + src[i][j].r
                           + src[i][j+1].r + src[i][j+2]).r / 5
        else
            if( i%4 == 1)
                dst[i][j].b = src[i][j-2].b
                dst[i][j].g = src[i][j-2].g
                dst[i][j].r = src[i][j-2].r
            else
                dst[i][j].b = src[i][j+2].b
                dst[i][j].g = src[i][j+2].g
                dst[i][j].r = src[i][j+2].r
```

1.4.2. Ocultar

El objetivo de este filtro es jugar un poco con *esteganografía* en imágenes. Este filtro se complementa con el siguiente denominado Descubrir (explicado luego del filtro actual). La idea es ocultar una imagen dentro de otra modificando los bits menos significativos de cada píxel. En principio, cada píxel tiene tres componentes, esto representa tres valores distintos a almacenar, implicando mucha información para ocultar de una imagen en la otra. Para reducir la cantidad de información que vamos a ocultar, convertimos la imagen a escala de grises utilizando la siguiente fórmula:

$$(src.b + 2 \cdot src.g + src.r) = 4$$

Donde src es una imagen, b el componente azul, g el componente verde y r el componente rojo.

Luego de convertir la imagen a ocultar ($src2$), se realizará el siguiente procedimiento para ocultarla:

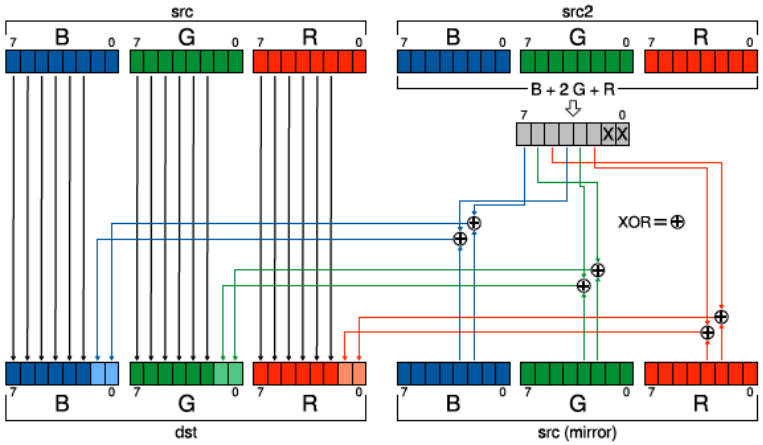


Figura 5: Ocultado de imagen

(a) Imagen 1 original

(b) Imagen 2 original

(c) Imagen 1 oculta en imagen 2

(d) Imagen 2 oculta en imagen 1

Figura 6: Filtro Ocultar

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

```
Para i de 0 a height - 1:
    Para j de 0 a width - 1:
        color = (src2[i][j].b + 2 * src2[i][j].g + src2[i][j].r) / 4

        bitsB = (((color >> 4) & 0x1) << 1) | ((color >> 7) & 0x1)
        bitsG = (((color >> 3) & 0x1) << 1) | ((color >> 6) & 0x1)
        bitsR = (((color >> 2) & 0x1) << 1) | ((color >> 5) & 0x1)

        dst[i][j].b = (src[i][j].b & 0xFC) + ((bitsB & 0x3) ^
                      ((src[(height-1)-i][(width-1)-j].b >> 2) & 0x3))

        dst[i][j].g = (src[i][j].g & 0xFC) + ((bitsG & 0x3) ^
                      ((src[(height-1)-i][(width-1)-j].g >> 2) & 0x3))

        dst[i][j].r = (src[i][j].r & 0xFC) + ((bitsR & 0x3) ^
                      ((src[(height-1)-i][(width-1)-j].r >> 2) & 0x3))
```

1.4.3. Descubrir

Este filtro realiza la operación inversa al filtro *Ocultar*. Realizando todas las operaciones realizadas para ocultar la imagen pero de forma inversa. El destino será una imagen que tenga en los tres componentes el mismo valor, es decir una imagen en escala de grises. La misma será la reconstrucción de la imagen oculta, donde los últimos dos bits (que no son almacenados en la imagen) serán seteados a cero.

(a) Imagen 1 conteniendo a la imagen 2 de manera oculta

(b) Imagen 2 conteniendo a la imagen 1 de manera oculta

(c) Imagen 1 reconstruida

(d) Imagen 2 reconstruida

Figura 7: Filtro Descubrir

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

```
Para i de 0 a height - 1:
    Para j de 0 a width - 1:
        b = ((src[(height-1)-i][(width-1)-j].b >> 2) ^ src[i][j].b) & 0x3
        g = ((src[(height-1)-i][(width-1)-j].g >> 2) ^ src[i][j].g) & 0x3
        r = ((src[(height-1)-i][(width-1)-j].r >> 2) ^ src[i][j].r) & 0x3

        bit2 = (r >> 1) & 0x1
        bit3 = (g >> 1) & 0x1
        bit4 = (b >> 1) & 0x1
        bit5 = r & 0x1
        bit6 = g & 0x1
        bit7 = b & 0x1

        color = (bit7 << 7) | (bit6 << 6) | (bit5 << 5) |
                (bit4 << 4) | (bit3 << 3) | (bit2 << 2)

        dst[i][j].b = color
        dst[i][j].g = color
        dst[i][j].r = color
```

2. Implementación

En esta sección nos dedicaremos a presentar y explicar nuestras propuestas para las implementaciones de los 3 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 solamente 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.

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 Zig-Zag

Como ya explicamos en la presentación de los filtros, el filtro *Zig-Zag* realiza un procesamiento distinto en cada fila, dependiendo de la congruencia módulo 4 del índice de la misma. Nuestro algoritmo consta de un doble ciclo anidado, que recorre la matriz por filas. Al comienzo del procesado de cada fila, evaluamos la congruencia módulo 4 del índice de la misma, y realizamos un *jump* al ciclo correspondiente que procesará los píxeles recorriendo las columnas.

Caso $i \equiv 0(mod 2)$

De los tres tipos de filas que tenemos que procesar, este caso es el más complejo. El objetivo que buscamos en nuestro algoritmo para este tipo de filas, fue procesar 2 píxeles por iteración. Para lograr ello, realizamos dos lecturas de memoria, utilizando la instrucción `movdqu`, para guardar en dos registros a los píxeles -2, -1, 0 y 1, y a los píxeles 0, 1, 2 y 3 respectivamente, donde dichos números representan el offset con respecto al índice de columna del píxel actualmente siendo procesado en la imagen destino. Con ésto tenemos los 6 píxeles que entran en juego para el procesado de los 2 píxeles que realizaremos por cada iteración.
Describiremos el procedimiento para el procesado del primer píxel en cuestión, ya que el segundo se realiza de manera análoga.
Haciendo uso de la instrucción `pshufb` y una máscara, obtenemos las componentes azul y verde de los píxeles -2, -1, 0 y 1. Las ubicamos de manera contigua, pero intercalando bytes en 0, para garantizar el espacio suficiente en las operaciones de suma que vienen a continuación (el resultado de una suma de dos enteros no signados que ocupan un byte, podría requerir un word para representarse correctamente). De esta manera tenemos en un registro el siguiente contenido:

$$xmm2 = ||0g_1|0g_0|0g_{-1}|0g_{-2}||0b_1|0b_0|0b_{-1}|0b_{-2}||$$

A continuación realizamos dos instrucciones `phaddsw` para tener en el registro, los resultados de la sumatoria de ambas componentes. Debido a que tomamos la precaución de garantizar el espacio necesario para cada componente, alojándolas en un word a cada una, sabemos que no vamos a tener problemas de *overflow*. Con un proceso análogo, realizamos la sumatoria de la componente roja de los mismos píxeles. Una vez realizado ésto, mediante instrucciones de shift y una instrucción `por` combinamos los valores en un solo registro, para que el mismo tenga el siguiente formato:

$$xmm2 = || - - - - - - - - |A|R|G|B||$$

Donde las componentes A, R, G y B, son las resultantes de las operaciones anteriores, y ocupan 1 word. Mediante otra instrucción `pshufb` tomamos el píxel que falta (el píxel 2), y lo posicionamos en `xmm3` de manera que tenga el mismo formato que el inmediatamente antes descripto para `xmm2`. Lo único que debemos hacer ahora es utilizar la instrucción `paddusw` para realizar la suma que nos dejará el siguiente contenido en `xmm2`:

$$xmm2 = || - - - -|A|R|G|B||$$

donde cada componente ocupa 1 word, y representa la suma de las componentes correspondientes de los píxeles -2, -1, 0, 1 y 2.

El proceso se repite para los píxeles -1, 0, 1, 2 y 3, para terminar con el contenido análogo antes logrado en `xmm3`. Mediante un *shift* en `xmm3` y una instrucción `por`, dejamos en `xmm2` el siguiente contenido:

$$xmm2 = ||A|R|G|B|A|R|G|B||$$

Donde las primeras 4 componentes son las correspondientes al primer procesado discutido, y las otras 4 al segundo. Cada una de esas componentes ocupa 1 word.
Lo único que queda realizar es la división de todas las componentes por 5, restaurar las componentes *alpha* (que pudieron ser afectadas en las operaciones), realizar una instrucción `packuswb` para pasar de *word* a *byte* de forma no signada y saturada, y realizar la escritura de ambos píxeles a la matriz destino mediante una instrucción `movq`.

Para realizar la división mencionada realizamos la siguiente operación:

$$x * ((2^8/5) >> 8)$$

El resultado de la división $2^8/5$ fue resuelto fuera del programa y ***hardcodeado*** en una máscara para realizar la multiplicación mediante la instrucción `pmullw`. Luego, mediante la instrucción `psrlw`, realizando un *shift* de 8 bits en cada word, dividimos por 2^8 para el resultado final (y así después poder realizar el pack y movq antes descriptos).
La razón por la cual realizamos este "truco" fue para evitar tener que pasar por dos instrucciones de *convert*, que como bien sabemos no son nada eficientes. Este *workaround* matemático descripto y estudiado en el trabajo de Peter L. Montgomery y Torbjörn Granlund [1], trae aparejado un error, cuyo valor disminuye para valores crecientes del exponente elegido para el 2. Con un pequeño script exhaustivo en python, calculamos cual es el menor exponente que garantiza un error acotado por la cota propuesta por la cátedra para los tests de nuestros filtros, y obtuvimos por resultado 8, que es el valor que utilizamos en nuestro algoritmo.

Caso $i \equiv 1(mod4)$

En este caso la imagen destino tendrá los mismos píxeles que la imagen origen, pero desfasados en 2 hacia atrás. Es decir, dado un píxel $p *_{i\ j}$ de la imagen de la imagen destino, este coincidirá con p_{ij-2} de la imagen origen. La solución en `SIMD` para este problema es bastante directa, ya solo necesitamos hacer una lectura de la imagen origen mediante la instrucción movdqu atrasada en 8 bytes (8 bytes = 2 píxeles, ya que cada píxel requiere 1 byte por componente), hacia un registro `xmm`. Luego simplemente realizamos la escritura a la matriz destino mediante el uso de la instrucción `movdqu`. En cada iteración de este ciclo que recorre las columnas de las filas que cumplen con el criterio que estamos describiendo, se procesan 4 píxeles.

Caso $i \equiv 3(mod4)$

Este caso es casi idéntico al anterior:
En este caso la imagen destino tendrá los mismos píxeles que la imagen origen, pero desfasados en 2 hacia adelante. Es decir, dado un píxel $p *_{i\ j}$ de la imagen de la imagen destino, este coincidirá con p_{ij-2} de la imagen origen. La solución en `SIMD` para este problema es bastante directa, ya solo necesitamos hacer una lectura de la imagen origen mediante la instrucción movdqu adelantada en 8 bytes (8 bytes = 2 píxeles, ya que cada píxel requiere 1 byte por componente), hacia un registro `xmm`. Luego simplemente realizamos la escritura a la matriz destino mediante el uso de la instrucción `movdqu`. En cada iteración de este ciclo que recorre las columnas de las filas que cumplen con el criterio que estamos describiendo, se procesan 4 píxeles.

2.2. Filtro Ocultar

Para el caso de Descubrir también procesaremos de a 4 píxeles simultáneamente. Utilizamos dos punteros para movernos dentro de las imágenes y un contador de columnas procesadas para saber hasta cuando iterar. Para hacer el calculo de color ***(src.b + 2 · src.g + src.r)/4*** lo que hacemos es filtrar a cada píxel por su color en un registro determinado poniendo para cada píxel el valor de la componente en la parte menos significativa de el píxel. Luego sumamos las distintas componentes de cada píxel que ahora nos quedaron alineadas y dividimos por 4.Para poder filtrar los colores hacemos gran uso de mascaras que nos permiten alinear datos de la forma que deseamos y llenar con 0 en donde sea necesario.

Luego para conseguir los bits necesarios para BitsB,BitsG y BitsR lo que hacemos es tener una mascara que nos conserve el bit menos significativo y lo que hacemos es llevar al bit deseado a la posición menos significativa del píxel para poder filtrarlo.Este proceso hay que hacerlo dos veces por cada componente pues tenemos que correr 2 bits diferentes, procurándonos de no perder la información conseguida en la primera iteración luego de la segunda iteración. Almacenados los resultados lo que hacemos es desplazarlos a su ubicación correspondiente nuevamente. Por ejemplo para la componente red hay que correrlo a la parte menos significativa de el componente red, de igual manera se desplaza el green. Ya corridos ponemos los resultados de todas las componentes en un solo registro.

Una vez hecho esto lo que hacemos es levantar de memoria los píxeles espejos. La problemática al levantarlos es que debemos invertir el orden en que aparecen los píxeles para que queden alineados a los cuales acabamos de procesar, sino hacemos esto no estaríamos trabajando con sus espejos. Para invertirlos usamos una mascara que 'divide' el registro en 4 *double word* y aplicando un `pshufb` ponemos en la parte menos significativa la *double word* mas significativa y viceversa , en la siguiente menos significativa pone la anterior a la mas significativa y viceversa. pasando de un registro de este estilo : pi,pi+1,pi+2,pi+3 a pi+3,pi+2,pi+1,pi. Con los píxeles invertidos, como nos interesa utilizar los bits 2 y 3 de cada componente los *shifteamos* a los dos bits menos significativos y usando una mascara que conserva los últimos dos bits por componente nos quedamos con la información. Aplicamos el `xor` entre el registro que tiene los resultados de BitsB,BitsG y BitsR y el registro en donde procesamos espejo.

Por ultimo, a la foto que esconde a la imagen en blanco y negro le aplicamos una mascara que a cada componentes RGB les pones en 0 sus 2 bits menos significativos. Notemos que deja alpha intacta y para terminar hacemos un `or` entre el resultado de esta imagen y el registro donde tenemos la información para los últimos 2 bits de cada componente y guardamos en memoria los píxeles.

2.3. Filtro Descubrir

Nuestro algoritmo para la implementación de este filtro procesa 4 píxeles por iteración. Mantenemos dos punteros: uno para recorrer la máscara origen, y otro para recorrer la imagen de atrás para adelante (mirror). Mediante dos máscaras e instrucciones `pand`, nos quedamos con los bits 0 y 1, y 2 y 3 de las componentes de los píxeles de origen y mirror, respectivamente. Con una instrucción `pxor` obtenemos el resultado intermedio necesario según el pseudocódigo, es decir los dos bits asociados a b, los dos bits asociados a g y los dos asociados a r para cada uno de los 4 píxeles que estamos procesando. En esta instancia, tenemos todos los bits que vamos a necesitar en un registro, así que solo queda reordenarlos. El reordenado se realiza mediante el uso de una máscara con un 1 en el comienzo de cada *double word*, que es *shifteada* según el bit que se necesita extraer. Una vez *shifteada*, se realiza la instrucción `pand` con el registro en el que guardamos los datos antes obtenidos, para así extraer el bit deseado de cada *double word*, y lo re-acomodamos mediante otro *shift*. Una vez realizado ésto, acumulamos el resultado en otro registro.
Cuando ya obtuvimos los 6 bits en el orden que requiere el pseudocódigo para cada uno de los cuatro *double words*, realizamos el *broadcast* del color gris que obtuvimos a todas las componentes de cada uno de los cuatro píxeles que vamos a escribir, mediante una instrucción `pshufb`.
Finalmente restauramos las componentes de transparencia, y realizamos la escritura mediante un `movdqu` a la imagen destino.

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. Para explorar más en detalles estos aspectos, y tener mejor criterio a la hora de programar nuestra mejor implementación en `SIMD`, realizamos 4 implementaciones alternativas para el filtro *Zig-Zag* que los dejan en evidencia:

- 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.
- Utilizar instrucciones *convert* para la división, en lugar del algoritmo alternativo antes explicado.
- Disminución en la cantidad de saltos, para disminuir *branch penalties*.

La razón por la cual estas variantes fueron implementadas solamente para el filtro *Zig-Zag* radica en cuestiones de tiempo y de la naturaleza del filtro, que hizo estas variantes más fáciles de realizar que en los otros dos.

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 8: 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 resoluciones 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 guíen nuestro análisis para determinar si una implementación es mejor que otra. Utilizaremos la siguiente:

- Cantidad de ticks de clock por aplicación de cada filtro.

Esta métrica es referida al tiempo, es decir la cantidad de ticks de clock por aplicación de filtro, necesita un poco más de justificación y algunos cuidados extra.
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. Junto con el promedio muestral, expondremos el desvío estándar asociado.

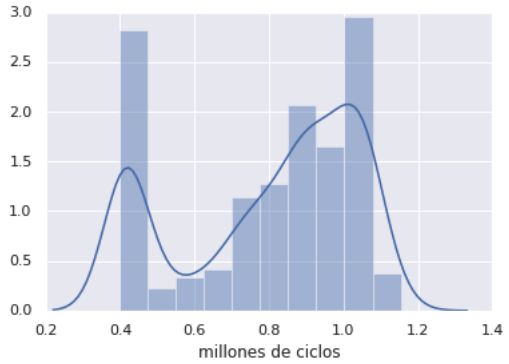


Figura 9: Distribución de las muestras tomadas para el filtro Zig-Zag con la imagen del set de pruebas correspondiente a la resolución 512x512 píxeles.

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 estándar.

Figura 10: Convergencia de la media muestral.

Figura 11: Convergencia de la desviación estándar.

En la figura 10 y 11 puede observarse el experimento mencionado. Se aplicó el filtro *ZigZag*, para la imagen de prueba, en resolución 128x128 píxeles, con repeticiones que fueron desde 0 a 1000. Para cada 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, esto sucede. A partir de 100 iteraciones, la media muestral comienza a estabilizarse, y esto se sigue acentuando conforme crece la cantidad de repeticiones. Con este criterio, decidimos realizar los experimentos que siguen, con 100 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 Core i7 de cuatro núcleos 1,7GHz, con 16 Gb de RAM y utilizando Ubuntu 18.04.4 LTS.

3.2. Experimentación

3.2.1. C vs ASM

El objetivo de este experimento es comparar, para cada filtro, nuestra implementación en lenguaje `assembly` explotando las instrucciones `SIMD`, contra la implementación en `C` provista por la cátedra con las siguientes opciones de optimización en tiempo de compilación:

- **O0**: Este nivel de optimización apaga por completo la optimización provista por el compilador. Esto reduce al mínimo el tiempo de compilación, y puede mejorar la información de debug.
- **O1**: El nivel más básico de optimización. El compilador tratará de producir código más rápido y más pequeño, sin tomar tanto tiempo de compilación.
- **O2**: A menos que el sistema tenga necesidades especiales, éste es el nivel de optimización recomendado. El compilador tratará de mejorar la performance aún más, sin comprometer el tamaño del binario ni el tiempo de compilación. En este nivel, el compilador puede utilizar SSE o AVX, pero no utilizará registros YMM.
- **O3**: Es el nivel más alto de optimización. Habilita optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. En muchos casos, puede hacer más lento un sistema debido a binarios generados más grandes, y mayor uso de memoria. De ser posible, los ciclos serán vectorizados utilizando registros YMM.

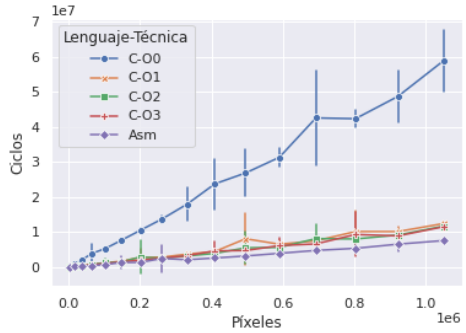
Para cada filtro, correremos las implementaciones para todo el set de imágenes de pruebas antes descripto, con la cantidad de iteraciones por corrida antes descripta. También incluiremos diferencias porcentuales en cantidad de ciclos de clock para todas las implementaciones con respecto a la implementación en `ASM`, para todos los filtros, para la imagen de resolución 1024x1024 píxeles. Ésto nos da alguna evidencia acerca de cuánto más lenta son las otras implementaciones contra la nuestra.

Hipótesis

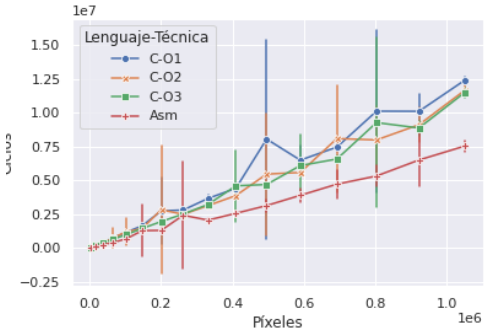
Nuestra hipótesis para este experimento, es que nuestra implementación en `Assembly` debería ser más rápida que la implementación en `C` con optimización O0 y O1. Además, nuestras implementaciones deberían ser n veces más rápidas que las recién nombradas, donde n es la cantidad de píxeles procesados por iteración. Por ejemplo, nuestra implementación del filtro *Descubrir*, procesa 4 píxeles por iteración. Así que los ciclos necesarios para la aplicación del filtro deberían ser 4 veces menores para nuestra implementación. Para flags de optimización O2 y O3, nuestra implementación deberías ser más rápida, pero la diferencia probablemente sea menor.

Resultados del experimento

Zig-Zag



(a) Ciclos de clock para asm, C-O0, C-O1, C-O2 y C-O3 para el filtro Zig-Zag



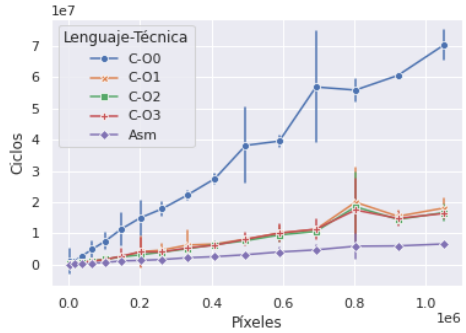
(b) Ciclos de clock para asm, C-O1, C-O2 y C-O3 para el filtro Zig-Zag

Figura 12: C vs ASM - Filtro Zig-Zag

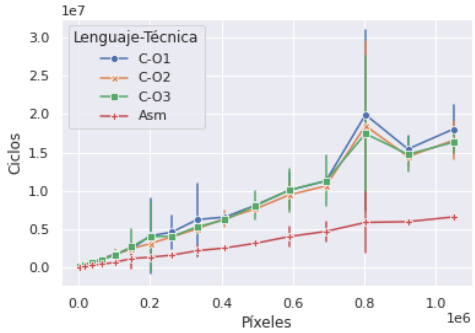
	C-O0	C-O1	C-O2	C-O3
Diferencia Porcentual contra Asm	635 %	64 %	54 %	52 %

Cuadro 1: Diferencias porcentuales en ciclos de reloj de las implementaciones en C contra la implementación en ASM, para el filtro Zig-Zag, para la imagen de prueba de 1024x1024 píxeles.

Ocultar



(a) Ciclos de clock para asm, C-O0, C-O1, C-O2 y C-O3 para el filtro Ocultar



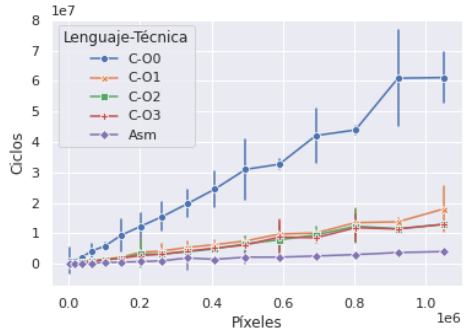
(b) Ciclos de clock para asm, C-O1, C-O2 y C-O3 para el filtro Ocultar

Figura 13: C vs ASM - Filtro Ocultar

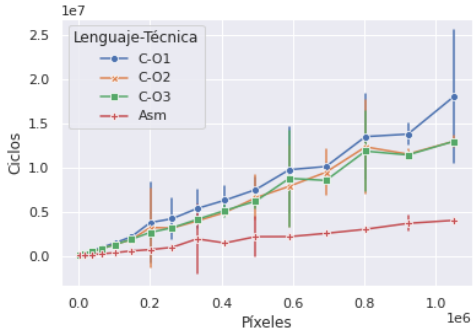
	C-O0	C-O1	C-O2	C-O3
Diferencia Porcentual contra Asm	965 %	161 %	148 %	149 %

Cuadro 2: Diferencias porcentuales en ciclos de reloj de las implementaciones en C contra la implementación en ASM, para el filtro Ocultar, para la imagen de prueba de 1024x1024 píxeles.

Descubrir



(a) Ciclos de clock para asm, C-O0, C-O1, C-O2 y C-O3 para el filtro Descubrir



(b) Ciclos de clock para asm, C-O1, C-O2 y C-O3 para el filtro Descubrir

Figura 14: C vs ASM - Filtro Descubrir

	C-O0	C-O1	C-O2	C-O3
Diferencia Porcentual contra Asm	1360 %	294 %	224 %	228 %

Cuadro 3: Diferencias porcentuales en ciclos de reloj de las implementaciones en C contra la implementación en ASM, para el filtro Descubrir, para la imagen de prueba de 1024x1024 píxeles.

En este experimento, y para todos los filtros, incluimos un gráfico que no incluye la corrida de la implementación en `C` con O0 de optimización, ya que la misma es considerablemente más lenta, y hace poco legibles los demás datos. En general, los resultados se condicen bastante bien con lo planteado en nuestras hipótesis. En particular, observamos:

- Para todos los filtros, la implementación en `ASM` es la más rápida.
- La corrida sin optimización (O0), es mucho más lenta (ver diferencias porcentuales en las tablas correspondientes) que las demás corridas en `C`. Su diferencia porcentual con la implementación en `ASM` tampoco corresponde con la predicha.
- Las corridas en O3 fue la más rápida en `C` en el filtro Zig-Zag, pero fue apenas más lenta que O2 en Descubrir y Ocultar. Ésto puede tener que ver con lo explicado en la descripción de los niveles de optimización y el comportamiento de O3.
- Si miramos la diferencia porcentual para C-O1 para todos los filtros, ella se corresponde bastante bien con lo predicho. Para el filtro descubrir, la diferencia fue 294 %, lo que indica que resultó aproximadamente 4 veces más lenta que la implementación en `ASM`. Ésto tiene sentido, pues la implementación en `ASM` de descubrir, procesa 4 píxeles por vez. Un análisis análogo en el filtro Zig-Zag, muestra que la corrida con C-O1, resultó casi 2 veces más lenta, lo que refleja el hecho de que el filtro procesa de a dos píxeles por vez la mitad de sus filas. Teniendo en cuenta que el filtro Ocultar procesa de a 4 píxeles, el análisis da también muy similar.

3.2.2. Recorrido por filas vs Recorrido por columnas

El objetivo de este experimento es analizar como impacta el correcto aprovechamiento de la memoria cache, comparando una implementación que procesa la imagen recorriéndola por filas, contra otra que la recorre por columnas. El experimento será realizado para el filtro *Zig-Zag*, para todas las imágenes del set de pruebas, y se adjuntará, al igual que antes, una tabla con la diferencia porcentual de la corrida por columnas con respecto a la corrida por filas, para la imagen de resolución 1024x1024.

Hipótesis

El recorrido de píxeles en nuestra implementación estándar 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, 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 un *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*.

Resultados del experimento

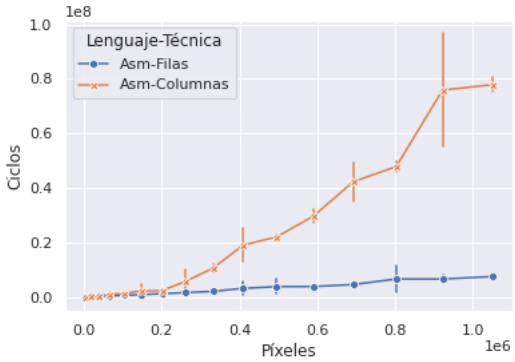


Figura 15: Recorrida por filas vs recorrida por columnas para el filtro ZigZag.

	Recorrido por columnas
Diferencia Porcentual contra Asm por filas	934 %

Cuadro 4: Diferencia porcentual en ciclos de reloj de la implementación en ASM por columnas contra la implementación en ASM por filas, para la imagen de prueba de 1024x1204 píxeles.

Los resultados coinciden con lo planteado en las hipótesis, marcando una clara diferencia en cuanto a eficiencia temporal, a favor de la recorrida por filas. Para la imagen de prueba de 1024x1024 píxeles, el recorrido por columnas es 10 veces más lento que el de filas.

3.2.3. Accesos innecesarios a memoria

El objetivo de este experimento es analizar el impacto de realizar lecturas innecesarias a memoria al momento de utilizar máscaras. Nuestra implementación estándar en `ASM` mueve las máscaras a registros xmm, para no tener que leer la máscara de memoria en cada iteración. Las dos implementaciones alternativas que usaremos para realizar el experimento, no realizan el guardado de la máscara previo al ciclo, y leen la máscara de memoria en cada iteración. La particularidad que diferencia a estas dos implementaciones alternativas, es que una de ellas tiene las máscaras alineadas a 16 en memoria, y la otra no.

Correremos este experimento para las tres implementaciones mencionadas, para todo el set de imágenes de prueba, para el filtro *Zig-Zag*.

Hipótesis

Planteamos como hipótesis que se debería apreciar una pequeña diferencia en ciclos de clock, dejando ver que la implementación más rápida es la que no lee innecesariamente de memoria, y a su vez que la que lee de memoria alineada es más rápida que la que lee de memoria desalineada. Entendemos, de todas maneras, que la existencia de la memoria cache puede hacer que esta diferencia sea bastante pequeña.

Resultados del experimento

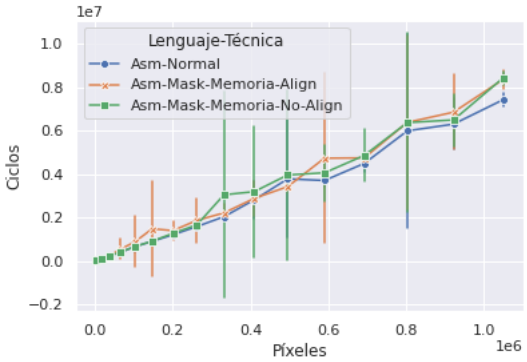


Figura 16: Análisis de implementaciones con accesos innecesarios a memoria.

Lamentablemente este no es un experimento concluyente. Si bien se puede ver que la implementación que no realiza lecturas innecesarias se mantiene por debajo de las curvas que si lo hacen, no hay un claro comportamiento en cuanto a la diferencia entra las lecturas alineadas contra las lecturas sin alinear. Además, el desvío estándar es bastante grande en casi todas las imágenes. Teniendo en cuenta eso, y que las diferencias son pequeñas, este experimento no está dando un resultado tan marcado como en los demás experimentos del presente trabajo.

3.2.4. Disminución en la cantidad de saltos

El objetivo de este experimento es analizar si se puede reducir el tiempo de ejecución del filtro *Zig-Zag*, disminuyendo la cantidad de saltos condicionales dentro de las iteraciones del ciclo, para evitar que se rompa el pipeline. La implementación que incluimos, que disminuye la cantidad de saltos, elimina la "pregunta" en cada iteración de fila correspondiente a analizar a qué caso corresponde la fila. Para evitar eso, hacemos tres ciclos de fila independientes: uno analiza solamente las filas del caso 1 y 3, otro las del caso 2, y otro las del caso 4. Con esa lógica de procesado, nos evitamos los saltos condicionales cada vez que cambiamos de fila.

El experimento se realizará para el filtro *Zig-Zag*, para todo el set de imágenes de prueba, para las dos implementaciones mencionadas.

Hipótesis

Esperamos ver una diferencia que se marque más a medida que crece la cantidad de filas, ya que la cantidad de "roturas de pipeline" debido a los saltos condicionales mencionados, aparecen en cada cambio de fila. La disminución de saltos condicionales debería hacer más rápida la implementación.

Resultados del experimento

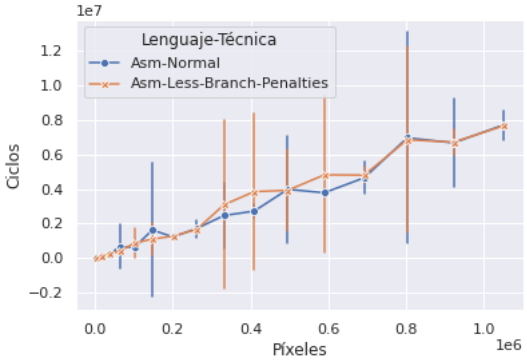


Figura 17: Análisis de implementación con menos saltos condicionales para evitar branch penalties.

El resultado del experimento es totalmente no concluyente. Las diferencias en ciclos de clock para las dos implementaciones son muy pequeñas, y los desvíos estándar asociados son muy grandes. Incluimos en futuras propuestas, al final de este informe, ideas para seguir investigando al respecto.

3.2.5. Overhead de instrucciones convert

El objetivo de este experimento es analizar el impacto de la utilización de instrucciones convert, y comparar esa implementación contra otra que no las utilice. Recordemos que el algoritmo para el filtro *Zig-Zag* requiere realizar una división. Para nuestra implementación estándar en `ASM`, realizamos el ya explicado algoritmo para dividir mediante una multiplicación y un shift. Compararemos esa implementación contra una que realiza la división de forma más tradicional, realizando conversiones de entero a punto flotante de precisión simple, y viceversa.

El experimento se realizará para el filtro *Zig-Zag*, para todo el set de imágenes de prueba, para las dos implementaciones mencionadas. Incluimos también, la diferencia porcentual en ciclos de reloj, de la implementación que utiliza instrucciones convert, contra la implementación que no las utiliza, para la imagen de prueba de 1024x1024 píxeles.

Hipótesis

Esperamos ver que nuestra implementación estándar sea más rápida que la que usa converts, por el hecho de que evita performance penalties.

Resultados del experimento

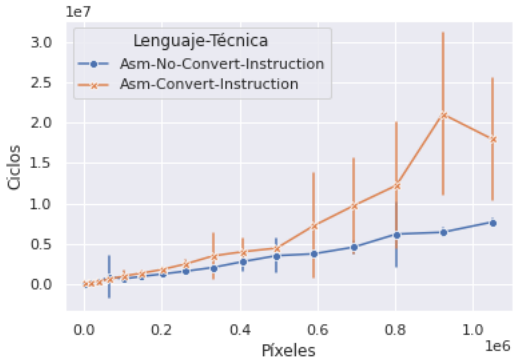


Figura 18: Análisis de overhead por utilización de instrucciones convert.

	Utilización de instrucciones convert
Diferencia Porcentual contra Asm sin converts	110 %

Cuadro 5: Diferencia porcentual en ciclos de reloj de la implementación en ASM utilizando instrucciones convert contra la implementación en ASM que no las utiliza, para la imagen de prueba de 1024x1024 píxeles.

Si bien los resultados en la implementación que utiliza instrucciones de conversión tienen asociado un desvío estándar bastante grande, la tendencia que se observa es bastante clara, aún considerando el error. La diferencia porcentual nos dice que la implementación sin instrucciones convert es un poco más de dos veces más rápida que la que si las utiliza. El resultado observado es compatible con las hipótesis planteadas.

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, instrucciones de conversión, 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. Además, sería interesante seguir experimentando acerca de como mejorar los tiempos de ejecución disminuyendo las *branch penalties*, mediante loop-unrolling, y diseños para eliminar usos de instrucciones de jump.

5. Bibliografía

[1] Torbjörn Granlund. Peter L. Montgomery. *Division by Invariant Integers using Multiplication*.