DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Procesamiento de imagenes mediante instrucciones SIMD

Organización del computador 2

Primer Cuatrimestre de 2018

Grupo Estrellitas

Integrante	LU	Correo electrónico
Hofmann, Federico	745/14	federico2102@gmail.com
Soberón, Nicolás	641/10	nico.soberon@gmail.com
Malbrán, Tomás	608/10	malbrantomas@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

Parte I
Introducción

En el siguiente trabajo se buscó conseguir un primer acercamiento a las instrucciones SIMD, las cuales se utilizan en el procesamiento de datos de manera simultanea. Para ello implementamos distintos filtros para imágenes tanto en el lenguaje de programación C como en Assembler. Una vez implementados procedimos a comparar ambas versiones de cada filtro buscando así poder conocer mejor las ventajas y desventajas de cada una.

Ademas, realizamos una serie de experimentos con cada filtro con el objetivo de realizar un análisis lo mas completo posible para de esta manera lograr justificar por que una implementación funciona mejor que la otra.

Por ultimo, intentaremos exponer todos los datos de la manera mas detallada posible, describiendo paso a paso los algoritmos implementados, explicando de forma completa la experimentación realizada y mencionando siempre todos los detalles que consideremos relevantes para el desarrollo de este trabajo y la comprensión del mismo por parte del lector.

Parte II
Implementaciones

Las imágenes utilizadas en este trabajo están en formato BMP y cada píxel de las mismas esta compuesto por 4 componentes de 1 byte cada una, ocupando así 4 bytes cada píxel. Dichas componentes son A, R, G y B, representando la componente de transparencia, el color rojo, el verde y el azul respectivamente. Cada componente esta representada por un entero sin signo, con lo cual sus posibles valores están entre 0 y 255. El ancho de las imágenes es siempre múltiplo de 4 píxeles. En ningún caso se alterará el valor de la componente A. Las imágenes son consideradas como matrices.

1. Blit

La idea de este filtro es, dada una imagen cualquiera, agregarle a ésta una imagen de Perón en su esquina superior derecha. Llamaremos a este proceso "*peronizar*" a dicha imagen.

Para que el filtro pueda ser aplicado, la imagen a procesar debe ser mas grande que la imagen de Perón, tanto su ancho como alto. Al mismo tiempo, no se copiará la imagen de Perón completa, sino que sólo se copiaran aquellos píxeles cuyos valores sean distintos del color magenta, es decir que sus componentes R, G y B sean distintas de 255, 0 y 255 al mismo tiempo respectivamente. En el caso en que un píxel sea magenta, ese lugar lo ocupará el píxel de la imagen original.

1.1. Pseudocódigo del algoritmo en C

Algoritmo 1 Pseudocódigo de la implementación en C de blit

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) del mismo tamaño con su ancho (cols) y alto (filas). Los tamaños de la fila de ambas imágenes (src_row_size y dst_row_size). La imagen a blitear (blit) y sus respectivos tamaños (bw, bh y b_row_size).

```
1: for i desde 0 hasta cantidad de filas do
2:     for j desde 0 hasta cantidad de columnas do
3:         if estoy en la esquina superior derecha then
4:             x = fila en imagen de Perón
5:             y = columna en la imagen de Perón
6:             if R == 255 y G == 0 y B == 255 then
7:                 resultado[i][j] = ImagenOriginal[i][j]
8:             else
9:                 resultado[i][j] = ImagenDePeron[x][y]
10:            end if
11:        else
12:            resultado[i][j] = ImagenOriginal[i][j]
13:        end if
14:    end for
15: end for
```

1.2. Descripción de implementación en assembler

Para la implementación del filtro blit en este lenguaje decidimos procesar de a 4 píxeles a la vez; la máxima cantidad posible ya que cada píxel ocupa 32 bits y los registros *xmm* son de 128 bits.

Lo primero que hace el algoritmo es separar registros para recorrer la matriz y para la imagen de Perón.

En el ciclo principal se comienza verificando dos condiciones: 1− si el tamaño de la imagen principal es menor a la de Perón, 2− si el registro que recorre la imagen principal esta parado sobre algún

píxel donde se debe ubicar la imagen de Perón.

1− Si se cumple esta condición, entonces durante el ciclo solo se mueven los píxeles de la imagen original hacia el destino.

2− Después de verificarse que el tamaño de la imagen es mayor o igual al de la imagen de Perón, si esta nueva condición no se cumple, el algoritmo copia los píxeles de la imagen principal directamente a la imagen destino. Pero si esta condición resulta verdadera, entonces se copia un píxel de la imagen de Perón en su lugar correspondiente en la imagen destino, sólo si este píxel no es de color magenta.

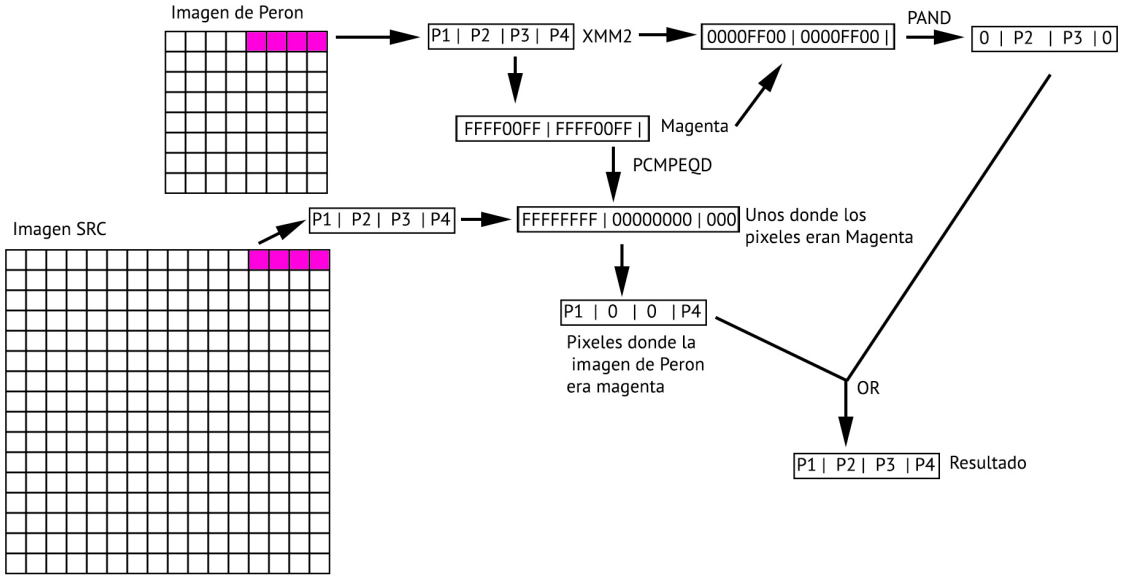

Figura 1: Movimiento y primer desempaquetado. Los de la imagen original (en magenta) se copian mediante la instrucción MOVQ a XMM0. Posteriormente, se desempaquetan las componentes de ambos de byte a word (PUNPCKHBW y PUNPCKLBW). Para esto último se utilizó un registro que contenía una máscara con 0 de forma de mantener el signo de R, G, B y A para futuras operaciones.

2. Monocromatizar

El filtro Monocromatizar consiste en convertir la imagen a escala de grises. Para poder realizar esto vamos a utilizar la fórmula propuesta en el enunciado.

2.1. Pseudocódigo del algoritmo en C

Algoritmo 2 Pseudocódigo de la implementación en C de monocromatizar

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) del mismo tamaño con su ancho (cols) y alto (filas). Los tamaños de la fila de ambas imágenes (src_row_size y dst_row_size).

```
1: for i desde 0 hasta cantidad de columnas do
2:     for j desde 0 hasta cantidad de filas do
3:         resultado[i][j]→A = ImagenOriginal[j][i]→A;
4:         componenteR = ImagenOriginal[j][i]→R
5:         componenteG = ImagenOriginal[j][i]→G
6:         componenteB = ImagenOriginal[j][i]→B
7:         maximoComponentes = max(componenteR, componenteG, componenteB)
8:         resultado[i][j]→R = maximoComponentes;
9:         resultado[i][j]→G = maximoComponentes;
10:        resultado[i][j]→B = maximoComponentes;
11:    end for
12: end for
```

2.2. Descripción de implementación en assembler

La idea de este filtro, es poder convertir cada pixel a escala de grises. Para poder realizar esta conversión utilizamos la siguiente modalidad:

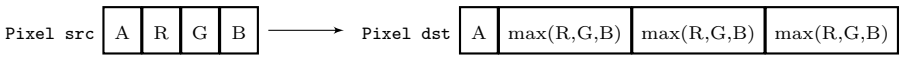

Figura 2: Modalidad para convertir cada pixel a escala de grises.

Lo primero que hacemos es inicializar registros apuntando a la imagen *src* (la que debemos modificar) y a la imagen *dst* (donde reconstruimos la nueva imagen).

También inicializamos un registro con el tamaño de la imagen, el cual se va decrementando en cada iteración del ciclo para poder comparar con 0, y ver cuando terminamos de recorrer toda la imagen.

Dentro del ciclo, traemos de a 4 pixeles por iteración, y trabajamos 2 pixeles a la vez. Antes de desempaquetar los datos, nos guardamos, utilizando una mascara, los valores correspondientes a cada pixel de la componente A.

Luego desempaquetamos los datos, y vamos a trabajar con **Pixel 1** y **Pixel 2**, los cuales vamos a almacenar en **XMM1**.

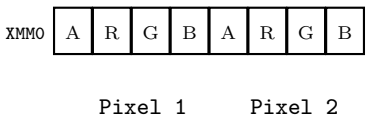

Figura 3: Pixeles 1 y 2 almacenados en XMM0

Copiamos **xmm0** a **xmm1** y utilizando **psllq** vamos a ir corriendo de a una componente hacia la izquierda, para poder ir comparando componente a componente y utilizando **pmaxsw** vamos a obtener el máximo.

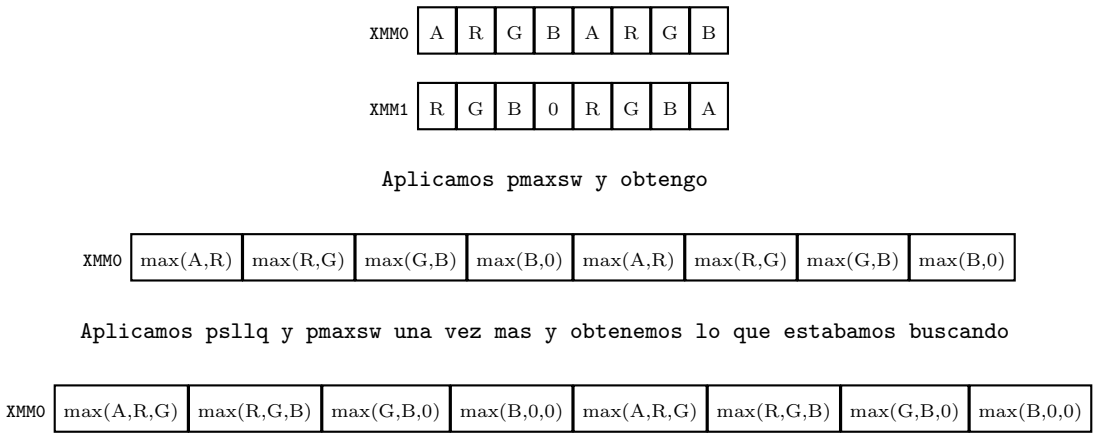

Figura 4: Procedimiento para obtener el máximo de cada componente

Repetimos el mismo procedimiento para los pixeles 3 y 4. Luego utilizando **pshufhw** y **pshuflw** vamos a poder guardar en **xmm7** y **xmm9** los valores correspondientes para los pixeles 1, 2, 3 y 4.

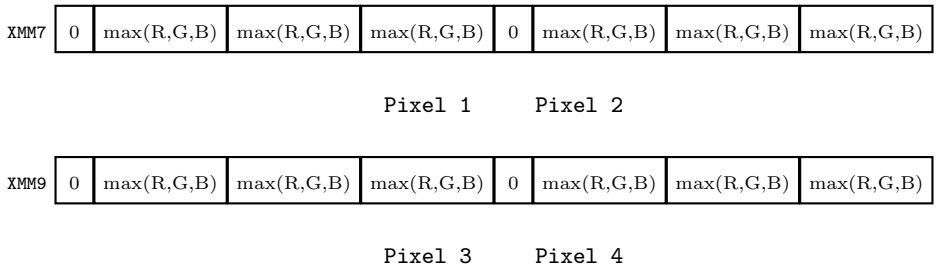

Figura 5: Estado previo a recuperar A en cada pixel

Luego empaquetamos los datos con la instrucción **packuswb**. Utilizamos los valores de A guardados inicialmente y los restauramos para cada pixel. Finalmente guardamos los datos en el registro de destino. De esta manera repetimos el proceso para cada pixel de la imagen, y logramos convertirla a escala de grises.

3. Efecto Ondas

La idea de este filtro es, por medio de un cálculo aplicado a cada píxel de la imagen, conseguir como resultado final un efecto visualmente asociado a ondas.

Para conseguir el efecto buscado, se tiene en cuenta la distancia euclidiana entre cada píxel y un segundo píxel fijo, marcado como centro de la onda. También se tiene un radio de onda fijo y una longitud de onda fija.

3.1. Pseudocódigo del algoritmo en C

Algoritmo 3 Pseudocódico de la implementación en C de Efecto Ondas

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) del mismo tamaño con su ancho (cols) y alto (filas). Los tamaños de la fila de ambas imágenes (src_row_size y dst_row_size). El centro de la onda (x0, y0).

1: **for** todos los píxeles de la imagen **do**
2: d = distancia(píxelActual, centroDeOnda)
3: r = (d - *radio*) / *longOnda*
4: k = r - parteEnteraDe(r)
5: a = $\frac{1}{(1+(r/3,4)\times(r/3,4))}$
6: t = k × 2 × PI - PI
7: t3 = t × t × t
8: t5 = t3 × t × t
9: t7 = t5 × t × t
10: sinTaylor = t - $\frac{t3}{6,0}$ + $\frac{t5}{120,0}$ - $\frac{t7}{5040,0}$
11: profundidad = a × sinTaylor
12: píxel = profundidad × 64 + píxelActual
13: píxel actual en imagen destino = min(max(píxel,0),255)
14: **end for**

3.2. Descripción de implementación en assembler

Lo primero que el algoritmo hace es mover las constantes a registros xmm.En el ciclo principal se mueve 1 píxel a un registro xmm y luego se los desempaqueta de manera que cada componente pasa a ocupar 32 bits. Posteriormente se convierte a esos cuatro números de enteros a floats para no perder precisión en los cálculos que les serán realizados. Se realiza una copia de este registro para pasarla por una máscara y obtener sólo la componente A, la cual no se modifica.

Después se llama a la función profundizar, con los parámetros x e y, los cuales representan la posición del píxel actual en la imagen original, y $x0$ e $y0$, los cuales marcan el centro de las ondas. Dentro de profundizar se copian los parámetros pasados a registros xmm distintos, convirtiéndolos a $float$ con la instrucción $cvtpi2ps$ y luego se lo copia 4 veces dentro de cada registro haciendo $pshufd$.

Con esos 4 registros listos se procede a calcular la distancia entre los puntos, luego las variables r, k y a utilizando las constantes preestablecidas y finalmente t, la cual se pasara por parámetro a través del registro $xmm0$ a la función $sinTaylor$.

Esta última función será la que calcule t^3, t^5 y t^7, luego las divide por tres constantes dadas y finalmente las junta en una cuenta y las devuelve a la función profundizar.

Por último, una vez obtenido el resultado de $sinTaylor$, se lo multiplica por a y se vuelve al ciclo principal del filtro. En todas las cuentas realizadas se utilizan operaciones de punto flotante, como $addps$, $subps$, $mulps$, $divps$ y otras.

Una ves de vuelta en el ciclo, se multiplica al resultado recién obtenido por 64 y se lo suma al píxel de la imagen original. Luego se satura a cada componente haciendo *minps* y *maxps* con registros que contienen los valores 255 y 0 respectivamente. Una vez hecho esto, se vuelve a convertir a enteros y se pasa al registro por una máscara que le quita la componente A y luego se lo suma al registro que preparamos previamente con la componente A original. Se mueve este resultado a la imagen destino y repetimos nuevamente todo el procedimiento con el siguiente píxel.

4. Edge

El filtro de Edge, es un filtro que a grandes rasgos busca los bordes de la imagen. Puede definirse como un borde a los píxeles donde la intensidad de la imagen cambia de forma abrupta. Si se considera una función de intensidad de la imagen, entonces lo que se busca son saltos en dicha función. La idea básica detrás de cualquier detector de bordes es el cálculo de un operador local de derivación. En este caso vamos a utilizar el operador de Laplace, cuya matriz es:

$$M = \begin{pmatrix} 0{,}5 & 1 & 0{,}5 \\ 1 & -6 & 1 \\ 0{,}5 & 1 & 0{,}5 \end{pmatrix}$$

4.1. Pseudocódigo del algoritmo en C

Algoritmo 4 Pseudocódico de la implementación en C de Edge

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) del mismo tamaño con su ancho (cols) y alto (filas). Los tamaños de la fila de ambas imágenes (src_row_size y dst_row_size).

```
1: i = 0
2: for j desde 0 hasta cantidad de columnas do
3:     destino[i][j] = ImagenOriginal[i][j]
4: end for
5: i = la ultima fila
6: for j desde 0 hasta cantidad de columnas do
7:     destino[i][j] = ImagenOriginal[i][j]
8: end for
9: for i desde 0 hasta cantidad de filas do
10:    j = 0
11:    destino[i][j] = ImagenOriginal[i][j]
12:    j = la ultima columna
13:    destino[i][j] = ImagenOriginal[i][j]
14: end for
15: for i desde 1 hasta cantidad de filas - 1 do
16:    for j desde 1 hasta cantidad de columnas - 1 do
17:        a = 0
18:        a+ = ImagenOriginal[i − 1][j − 1] ∗ 0,5
19:        a+ = ImagenOriginal[i − 1][j]
20:        a+ = ImagenOriginal[i − 1][j + 1] ∗ 0,5
21:        a+ = ImagenOriginal[i][j − 1]
22:        a+ = ImagenOriginal[i][j] ∗ −6
23:        a+ = ImagenOriginal[i][j + 1]
24:        a+ = ImagenOriginal[i + 1][j − 1] ∗ 0,5
25:        a+ = ImagenOriginal[i + 1][j]
26:        a+ = ImagenOriginal[i + 1][j + 1] ∗ 0,5
27:        destino[i][j] = minimo(maximo(a, 0), 255)
28:    end for
29: end for
```

4.2. Descripción de implementación en assembler

La idea de este filtro, es poder encontrar los bordes de la imagen. Primero generamos mascaras, para poder realizar las operaciones correspondientes a la matriz de laplace.

Inicializamos contadores:
R10 - contador imagen
R9 - contador alto

R8 - contador de tres iteraciones
RCX - contador ancho

Luego, realizamos un ciclo, para poder lograr que la primera fila de la imagen destino sea igual a la primera fila de la imagen fuente.

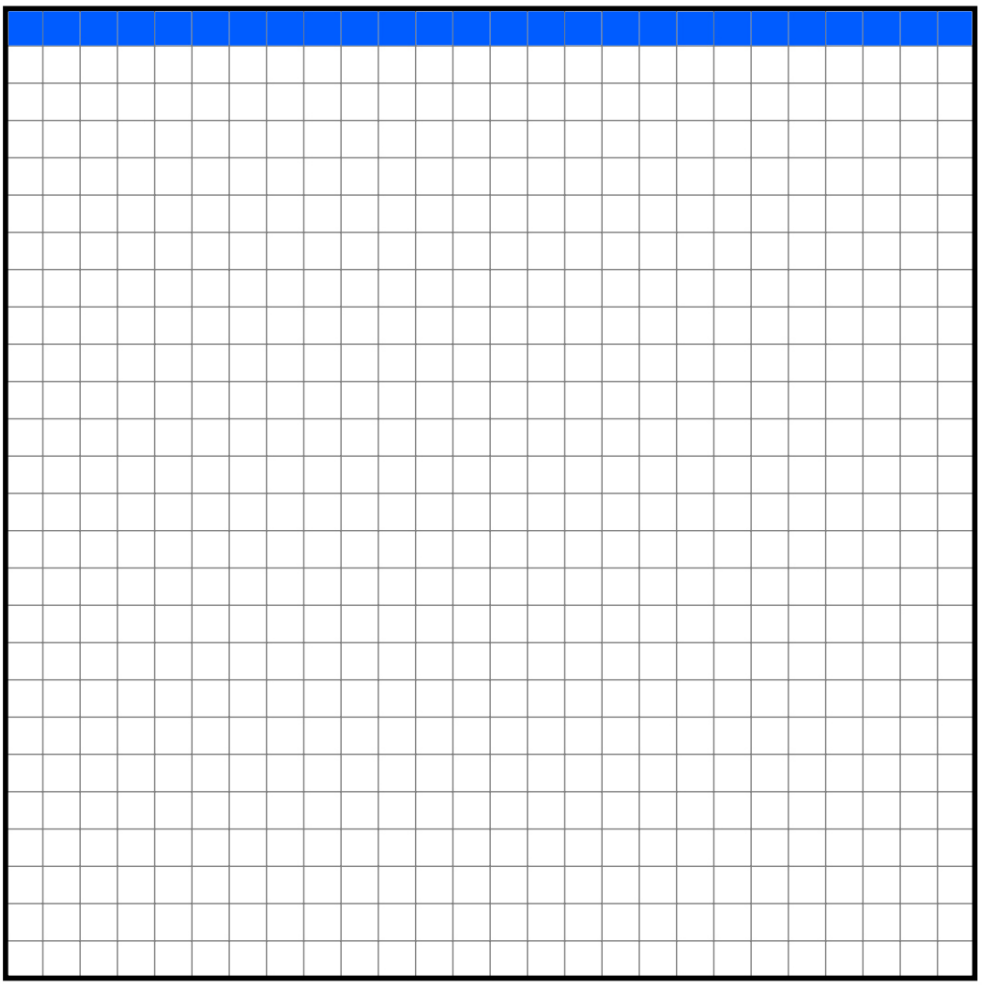

Figura 6: Ciclo inicial, el color azul representa los píxeles que son iguales a la imagen fuente

Una vez terminado el ciclo, comienza el ciclo principal. Durante el ciclo principal, la idea es ir realizando las operaciones correspondientes para poder obtener la imagen filtrada, vamos a traer de a 16 píxeles, y con ellos vamos a poder calcular 5 píxeles. Durante 2 iteraciones más nos vamos a mover de a 1 píxel para calcular 5 píxeles en la primer iteración y otros 5 en la siguiente. Por lo tanto antes de escribir vamos a tener calculados 15 píxeles que vamos a escribir en la imagen destino.

Ciclo Principal

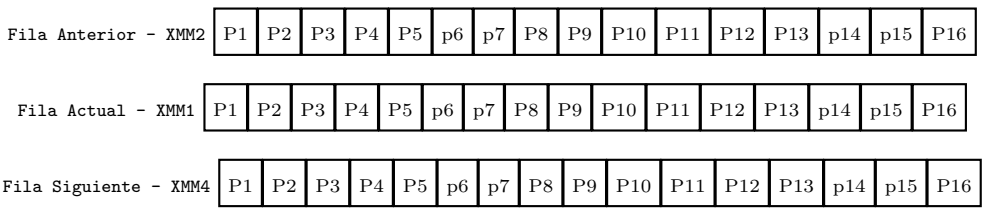

Figura 7: Fila actual almacenada en XMM0, Fila anterior almacenada en XMM2 y Fila siguiente almacenada en XMM4

A la fila anterior y fila siguiente debemos multiplicar por 0,5, 1, 0,5 repetido 5 veces. (El último byte lo dejamos en cero). Para ello primero a cada una de esas filas las vamos a desempaquetar de byte a word. Tanto en la parte baja como en la parte alta de cada fila, primero vamos a copiar en otro registro. Al primer registro le aplicamos una máscara para solo quedarnos con los valores que deseamos multiplicar por 0.5 (Los valores 1, 3, 4, 5, 7, 8...). Lo multiplicamos por 0.5, mediante un shift con la operación `PSRLW`. Al segundo registro le aplicamos una máscara para quedarnos con los valores que deseamos multiplicar por 1 (opuesta a la anterior). Luego sumamos los registros. Esto se hace tanto en la parte alta como en la baja del registro, la única diferencia es que la máscara cambia, dado que tenemos 8 words en cada parte y nuestra multiplicación se repite cada 3.

Para la fila actual, las operaciones son similares. La diferencia es que deseamos multiplicarla por $1, -6, 1$ repetido 5 veces. Nuevamente necesitamos desempaquetar de byte a word. Para multiplicar con -6, aplicamos una máscara que solo nos deja los valores que queremos multiplicar con -6, movemos a otro registro, los valores $0, -6, 0$ repetido las veces necesarias, y hacemos la multiplicación. Al resultado le agregamos los valores que teníamos que multiplicar con 1. Como en el caso anterior se usaron diferentes máscaras para la parte alta y baja.

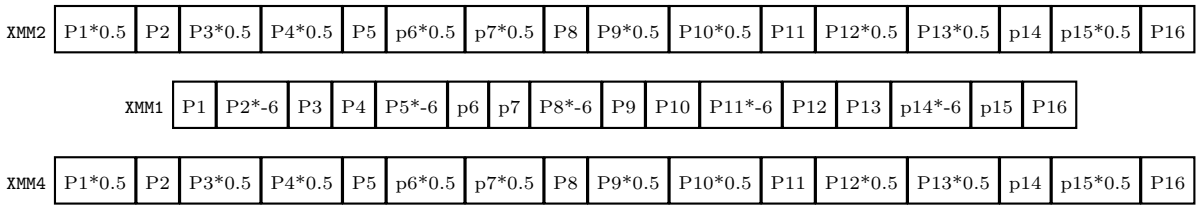

Figura 8: Filas luego de aplicar la matriz de laplace en cada píxel

Luego, sumamos la parte baja de cada registro y la parte alta. Primero sumamos los 3 registros, y luego tenemos que sumar horizontalmente de a 3 valores. Esto lo hacemos copiando a otro registro, y haciendo un shift para sumar. Notar que en la parte baja tenemos 8 valores, y al sumar de a 3, quedan 2, que les falta sumar 1 valor. Este valor es el primero de la parte alta, por lo que se lo sumamos por separado. De esta manera podemos obtener los 15 píxeles procesados que vamos a mover a memoria.

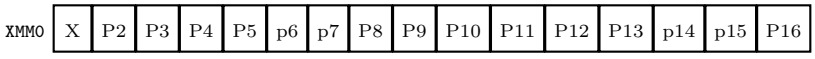

Figura 9: Registro que contiene los píxeles procesados

Para mover los píxeles a la imagen de destino consideramos 3 casos:

1. Si vamos a escribir los primero 16 píxeles de la fila, luego al píxel 1 que nos falta le movemos el valor de del primer píxel de la fila de la imagen fuente.
2. Si vamos a escribir los últimos píxeles de la fila debemos shiftear todo el registro 1 byte para que el píxel no calculado sea el último y a ese píxel le movemos el valor del la última columna de imagen fuente.
3. Si estamos en el medio, también vamos a shiftear todo el registro 1 byte para que el píxel no calculado sea el último y lo vamos a escribir así en la imagen destino. De esa forma el píxel no calculado no sobreescribe lo ya calculado.

Como generamos solo 15 píxeles en cada iteración, excepto al principio y al final que obtenemos los 16 píxeles, y nos movemos 14 píxeles, en la mayoría de los casos es muy probable que no estemos leyendo los últimos 16 valores al llegar al final de la fila. Por lo tanto antes de pasarnos y estar leyendo píxeles de la fila actual y la siguiente, movemos nuestro puntero que apunta al lugar donde queremos leer en la imagen fuente para que podamos generar los últimos 16 píxeles de la imagen, y no cualquier cosa. Esto implica que algunos valores posiblemente se calculen más de una vez y se sobreescriban por el mismo valor. Pero hace que el algoritmo soporte cualquier ancho superior a 18.

Cuando terminamos el ciclo, la imagen queda:

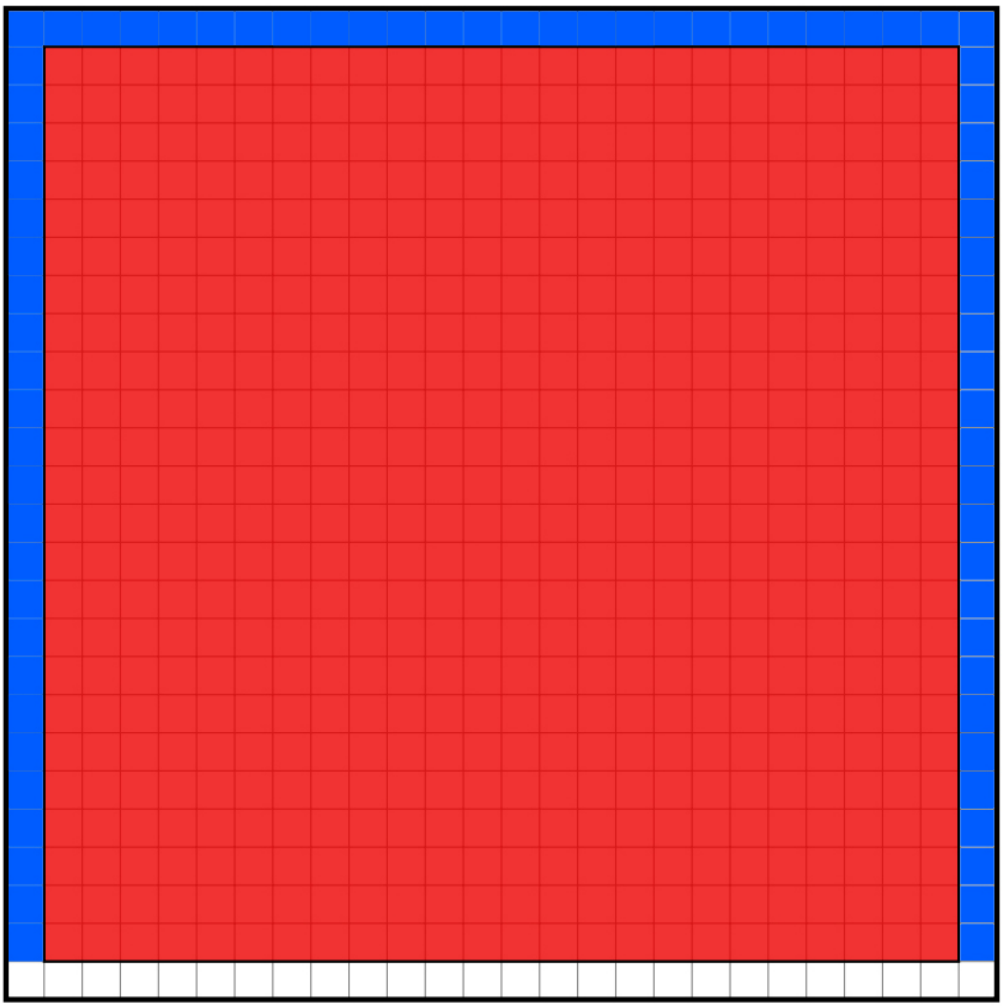

Figura 10: El color azul representa los píxeles que son iguales a la imagen fuente, el color rojo representa los píxeles procesados

Luego, vamos a dejar la última fila, igual a la imagen fuente. Realizamos un ciclo para poder llevar a cabo esto, y lograr obtener la imagen final.

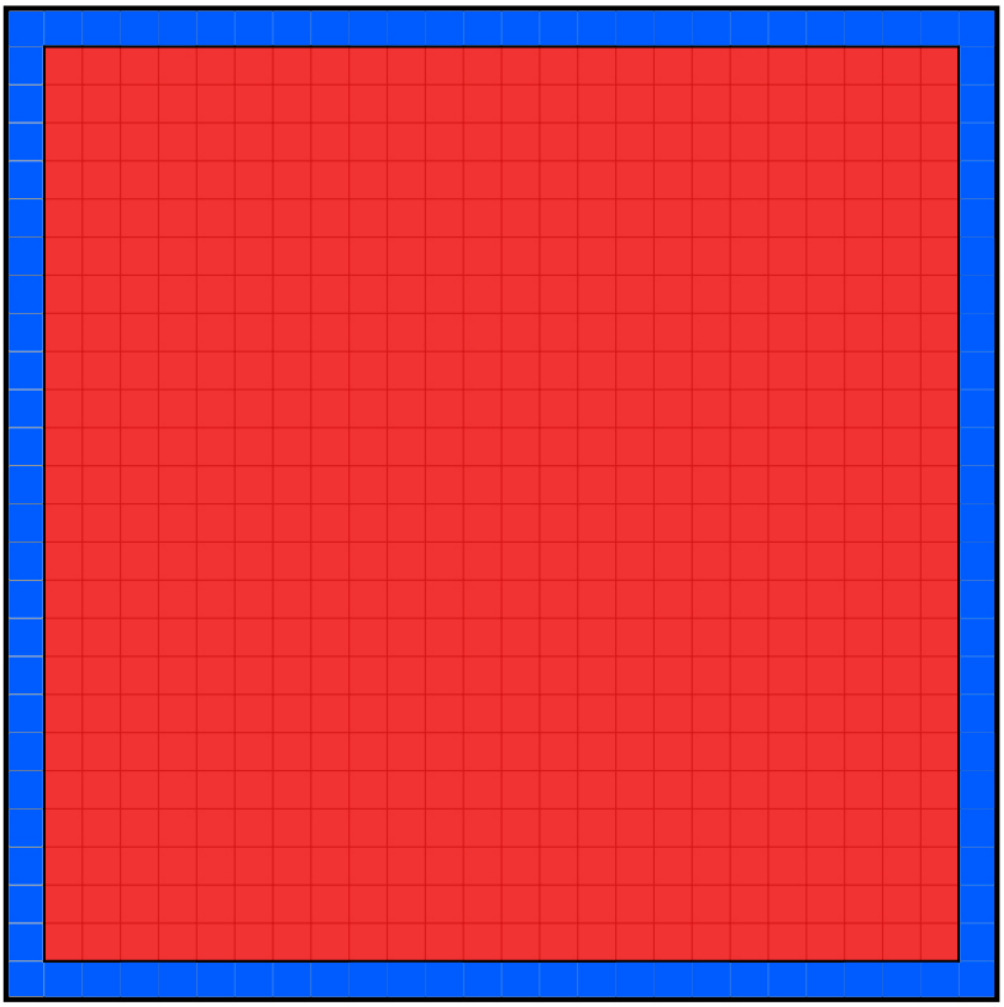

Figura 11: Imagen final, luego del algoritmo, el color azul representa los píxeles que son iguales a la imagen fuente, el color rojo representa los píxeles procesados

5. Temperatura

El filtro Temperatura consiste en convertir la imagen a en una que simula un mapa de calor. Para poder realizar esto vamos a utilizar la fórmula propuesta en el enunciado. Comenzamos por calcular la temperatura por cada píxel, la cual es equivalente a sumar los valores para R, G y B y luego dividir por 3. Luego dependiendo del valor de t se calculan los valore R, G y B de la imagen destino, usando formulas diferentes según el valor de t. Los valores de R, G y B para la imagen de destino se calcularon según la siguiente fórmula:

$$\mathsf{dst}_{(i,j)} < r, g, b >= \begin{cases} < 0, 0, 128 + t \cdot 4 > & \text{si } t < 32 \\ < 0, (t-32) \cdot 4, 255 > & \text{si } 32 \leq t < 96 \\ < (t-96) \cdot 4, 255, 255 - (t-96) \cdot 4 > & \text{si } 96 \leq t < 160 \\ < 255, 255 - (t-160) \cdot 4, 0 > & \text{si } 160 \leq t < 224 \\ < 255 - (t-224) \cdot 4, 0, 0 > & \text{si no} \end{cases}$$

5.1. Pseudocódigo del algoritmo en C

Algoritmo 5 Pseudocódigo de la implementación en C de temperature

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) del mismo tamaño con su ancho (cols) y alto (filas). Los tamaños de la fila de ambas imágenes (src_row_size y dst_row_size).

```
1: for i desde 0 hasta cantidad de filas do
2:     for j desde 0 hasta cantidad de columnas do
3:         Sean r_s, g_s y b_s los valores RGB de la imagen fuente
4:         Sean r_d, g_d y b_d los valores RGB de la imagen destino
5:         t = (r_s + g_s + b_s)/3
6:         if t < 32 then
7:             < r_d, g_d, b_d > = < 0, 0, 128 + t * 4 >
8:         else if t < 96 then
9:             < r_d, g_d, b_d > = < 0, (t − 32) * 4, 255 >
10:        else if t < 160 then
11:            < r_d, g_d, b_d > = < (t − 90) * 4, 255, 255 − (t − 90) * 4 >
12:        else if t < 224 then
13:            < r_d, g_d, b_d > = < 255, 255 − (t − 160) * 4, 0 >
14:        else
15:            < r_d, g_d, b_d > = < 255 − (t − 224) * 4, 0, 0 >
16:        end if
17:    end for
18: end for
```

5.2. Descripción de implementación en assembler

Para la implementación del filtro temperatura en este lenguaje decidimos procesar de a 4 píxeles a la vez; la máxima cantidad posible ya que cada píxel ocupa 32 bits y los registros xmm son de 128 bits. Aunque para calcular los colores de destino se proceso cada valor R, G y B para cada caso por separado. Por ejemplo si al calcular t para los 4 píxeles todos dan entre 32 y 96, luego se procede a calcular los verdes para los 4 píxeles simultáneamente, y luego los azules, y los rojo se dejan en 0. Si por el contrario el t calculado para el píxel 1 cae en un caso separado al del píxel 2, luego ambos casos se calculan por separado.

Leemos 4 píxeles en $xmm0$:

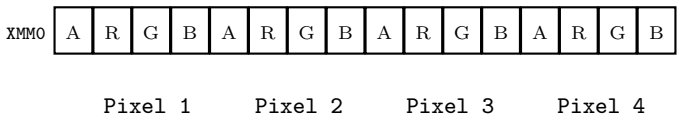

Figura 12: Pixeles 1 y 2 almacenados en XMM0

Comenzamos por calcular t. Para lo cual procedemos a crear 3 registros de forma que el primero tenga todos los rojos, el segundo todos los azules y el tercero todos los verdes en la parte baja de los registros, para los 4 píxeles leidos.

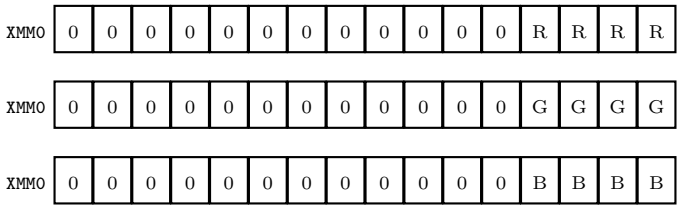

Figura 13: Los registros antes de realizar la suma

Luego como la suma va a superar 255, necesitamos desempaquetar de bytes a words usando la instrucción `PUNPCKLBW` dado que solo nos interesan las partes bajas de los registros. Una vez desempaquetados procedemos a sumar los 3 registros.

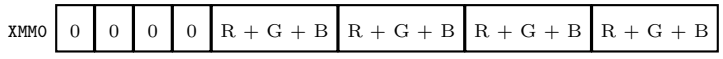

Figura 14: La suma de R, G y B

Para hacer la división necesitamos pasar a floats, por lo que primero desempaquetamos de words a double words con `PUNPCKLWD` y luego convertimos a floats con `CVTTDQ2PS`. Nuevamente solo nos interesa la parte baja, dado que solo necesitamos 4 valores de los 8 que tenemos. Dividimos por 3 y luego volvemos a convertir a entero por truncado. Ahora tenemos en un registro calculado el t para los 4 píxeles leidos.

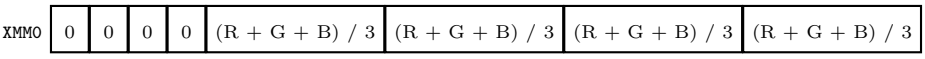

Figura 15: El calculo de T

Antes de comenzar a calcular los valores de R, G y B, nuestra idea fue crear un registro donde a cada pixel le asignamos $< r, g, b >=< t, t, t >$. O sea creamos un registro de la forma:

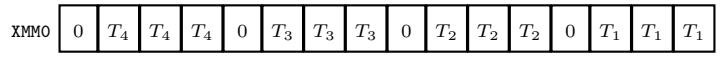

Figura 16: T asignado a R, G y B para cada píxel

Para esto empaquetamos de double word a word y luego de word a byte. En este punto el registro tiene 4 veces $t|0|0|0$, donde t cambia por cada píxel. Aplicamos un shuffle y obtenemos el resultado que queriamos.

La siguiente parte consiste en comparar el registro $xmm0$ con los diferentes valores de la fórmula para obtener una mascara y aplicársela al registro para tener los valores de t para los casos que queremos y 0 en el resto. La función de comparación usada fue `PCMTGB` que compara si el destino es

mayor a la fuente y deja en el destino $0xFF$ por cada byte que es mayor al valor en la fuente y el resto en $0x00$. Para poder usar esta función fue necesario invertir el orden a como lo implementamos en C. Comenzamos por comparar con $t > 223$. (Notar que al saber que t es entero, esto es equivalente a $t \geq 224$. Y luego podemos comparar con $t \geq 160$). Por el otro lado, la función de comparación no trabaja con números sin signo, por lo que antes de comparar le restamos a cada uno de los valores 128, usando `PXOR` con $0x80$. La comparación se hizo con el valor restado 128. Por ejemplo: en el primer caso seria la comparación $t > 224$ es equivalente a comparar $(t-128) > 224-128 = 95$. De esta forma nos ahorramos pasar de byte a word para comparar.

Comenzamos con el caso $t > 223$. Obtenemos una máscara realizando la comparación descripta anteriormente y la guardamos en un registro ($xmm2$). Luego procedemos a calcular los valores de R, G y B por separado. En el primer caso G y B son iguales a 0, por lo que solo nos queda calcular R. Para ellos usamos una máscara que nos deja solo los valores de los rojos y el resto en 0, que esta guardada en el registro $xmm13$. Tomamos un registro $xmm4 = xmm0$, le aplicamos la máscara $xmm2$ y luego la máscara $xmm13$. De esa forma en $xmm4$ tenemos t si $t > 223$ y esta en las posiciones de los rojos.

XMM4	0	0	0	0	0	T_3	0	0	0	0	0	0	0	T_1	0	0

Figura 17: Rojos con T suponiendo que T_3 y T_1 son mayores a 223

Necesitamos calcular para el rojo $255 - (t - 224) * 4$. Comenzamos con $xmm4$, a este le restamos otro que tiene 224 en los rojos donde $t > 223$. (Tenemos un registro donde todos los bytes son iguales a 224 al cual le aplicamos las máscaras $xmm2$ y $xmm13$). Teniendo $t - 224$, lo sumamos 2 veces para tener $(t - 224) * 4$. Notar que el valor no va a superar 255, ya que $t - 224$ es como mucho 31 y $31 * 4 = 124$ (Lo mismo ocurre en todos los casos). Finalmente generamos otro registro con 255 en los rojos donde $t > 223$ y a ese registro le restamos los calculado anteriormente.

XMM5	0	0	0	0	0	224	0	0	0	0	0	0	0	224	0	0
XMM3 - XMM5	0	0	0	0	0	$224 - T_3$	0	0	0	0	0	0	0	$224 - T_1$	0	0
XMM3 * 4	0	0	0	0	0	$(224 - T_3) * 4$	0	0	0	0	0	0	0	$(224 - T_1) * 4$	0	0
XMM5	0	0	0	0	0	255	0	0	0	0	0	0	0	255	0	0
XMM5 - XMM3	0	0	0	0	0	$255 - (224 - T_3) * 4$	0	0	0	0	0	0	0	$255 - (224 - T_1) * 4$	0	0

Figura 18: Calculo para los Rojos suponiendo que T_3 y T_1 son mayores a 223

El resultado lo guardando en otro registro $xmm1$ que inicialmente comienza en 0 para los valores de R, G y B, y $0xFF$ para el alpha, usando `POR`.

Cuando pasamos al segundo caso y necesitamos los t entre 160 y 223. Comenzamos por copiar $xmm0$ a $xmm3$. Luego necesitamos descartamos los valores de t mayores a 223. Por lo tanto invertimos la máscara del caso 1 haciendo un `PXOR` contra un registro con todos unos. En $xmm4$ tenemos una máscara que tiene 0 si $t > 223$ y 1 en el resto. Luego usando `PAND` entre $xmm3$ y $xmm4$ para obtener en $xmm3$ los valores de t si $t \leq 223$ y 0 en le resto. Finalmente podemos comparar contra un registro que tiene en todos los bytes 159 para generar un máscara que tiene 1 si $160 \geq t < 224$ y 0 en el resto.

Para calcular R, G y B hacemos los mismo que en el caso anterior. Usamos la máscara $xmm3$ y las máscaras que dejan R, B, o G en el valor que tienen y 0 en el resto para calcular R, G y B por separado. En el caso del rojo tienen que ser igual a 255, movemos a $xmm4$ un registro donde todos los bytes son 255, le aplicamos la mascara $xmm3$ y luego la máscara $xmm13$ y nos quedamos

con un registro con 255 en los bytes usados para el rojo si para ese píxel, $160 \geq t < 224$. Luego lo guardamos en $xmm1$. Para el Verde la idea es similar al caso anterior. Generamos un registro con t en los bytes para los verdes donde para ese píxel $160 \geq t < 224$ usando las máscaras $xmm3$ y $xmm14$. Le restamos 160, lo sumamos 2 veces para obtener $(t - 160) * 4$ y a 255 le restamos este valor. El 160 y el 255 se generan de registros que comienzan con esos valores en todos los bytes y se le aplican las máscaras para que solo queden en los bytes para los verdes donde para ese píxel $160 \geq t < 224$.

Repetimos el mismo proceso para $96 \geq t < 160$ y $32 \geq < 96$. Para el caso final $t < 32$ ya no necesitamos comparar al tener una mascara para $t > 31$, luego solo tenemos que aplicar la máscara invertida.

Escribimos el contenido del registro $xmm1$ en la imagen de destino y procedemos a hacer lo mismo para los siguiente 4 píxeles hasta llegar al final de la imagen.

Parte III
Experimentación

Para realizar la experimentación utilizamos distintos tamaños de imágenes, generados con el archivo de la cátedra para generar imágenes.

Para cada tipo de experimento, corrimos, el filtro a experimentar, 100 veces. En cada iteración nos guardamos el tiempo que tarda el algoritmo en correr. Este tiempo no contempla el tiempo que se demora en abrir la imagen, guardarla y liberar la imagen.

Luego tomamos el promedio de las mediciones para cada tamaño de imagen, y de esta manera pudimos graficar los tiempos para los distintos tamaños de imagen. Para realizar los gráficos utilizamos plot.ly

Algoritmo 6 tp2-Experimentacion.c - main que utilizamos para medir los experimentos

```
1: int cantIteraciones = 100;
2: filtro_t * filtro = detectar_filtro(&config);
3: filtro→leer_params(&config, argc, argv);
4: unsigned long start, end;
5: unsigned long res[cantIteraciones];
6: int j = 0;
7: for i desde 0 hasta cantIteraciones do
8:     imagenes_abrir(&config);
9:     MEDIR_TIEMPO_START(start);
10:    filtro→aplicador(&config);
11:    MEDIR_TIEMPO_STOP(end);
12:    unsigned long delta = end - start;
13:    res[i] = delta;
14:    imagenes_guardar(&config);
15:    imagenes_liberar(&config);
16: end for
17: unsigned long resToPrint = 0;
18: for j desde 0 hasta cantIteraciones do
19:    resToPrint += res[j];
20: end for
21: resToPrint = resToPrint / cantIteraciones;
```

6. Comparación C vs Assembler

Este experimento consiste en comparar los tiempos de computo de los algoritmos de Blit y Edge en sus versiones en Assembler contra las versiones de C, utilizando las distintas optimizaciones de GCC para compilar el código:

-O0: Este nivel desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel -O en CFLAGS o CXXFLAGS.
-O1: El nivel de optimización más básico. El compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación.
-O2: Un paso delante de -O1. Es el nivel recomendado de optimización, a no ser que el sistema tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación. Se puede utilizar SSE o AVX en este nivel pero no se utilizarán registros YMM a menos que también se habilite ftree-vectorize.
-O3: El nivel más alto de optimización posible. Activa optimizaciones que son caras en términos de tiempo de compilación y uso de memoria. El hecho de compilar con -O3 no garantiza una forma

de mejorar el rendimiento y, de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria. También se sabe que -O3 puede romper algunos paquetes. No se recomienda utilizar -O3. Sin embargo, también habilita -ftree-vectorize de modo que los bucles dentro del código se vectorizarán y se utilizarán los registros AVX YMM.

El objetivo, es ver que beneficios tiene utilizar código ASM sobre código C. Para este experimento se realizaron mediciones para distintos tamaños de imágenes. En cada imagen se corrió 100 veces por cada filtro, y luego se obtuvo un promedio, para poder representar los valores en un gráfico.

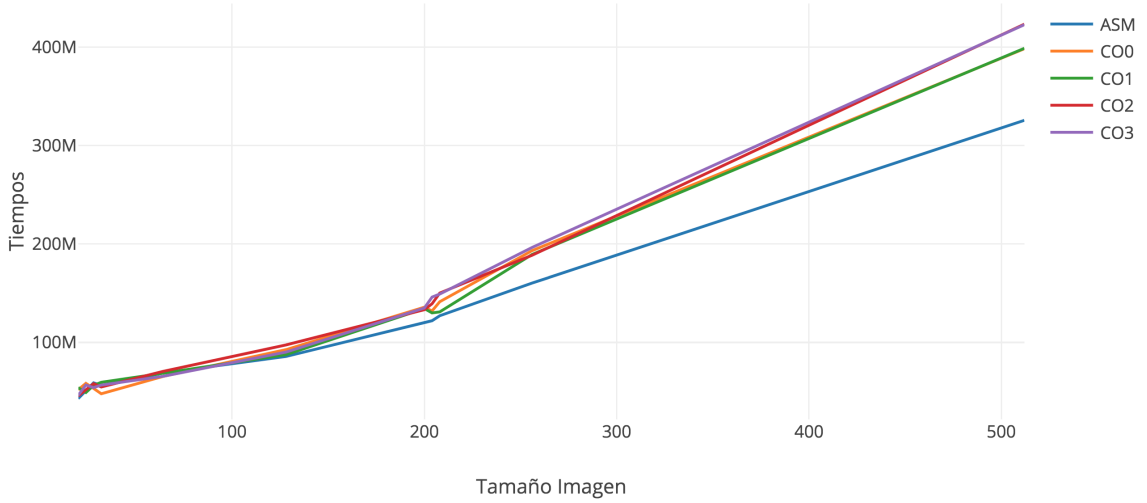

Figura 19: Gráfico que muestra los tiempos para el filtro Edge. Los tamaños de imagen utilizados fueron 20x20, 24x24, 28x28, 32x32, 64x64, 128x128, 200x200, 204x204, 208x208, 256x256 y 512x512

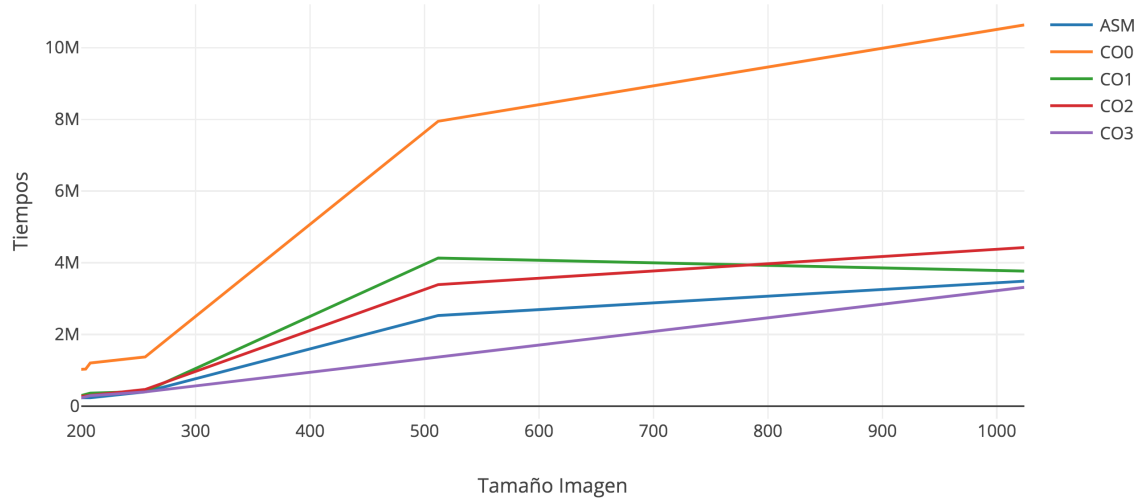

Figura 20: Gráfico que muestra los tiempos para el filtro Blit. Los tamaños de imagen utilizados fueron 200x200, 204x204, 208x208, 256x256, 512x512 y 1024x768

Analizando el gráfico obtenido, podemos afirmar que en el caso de Edge, las ventajas de utilizar ASM sobre C, son notorias. En el caso de Blit, la experimentación nos deja una puerta abierta para futuras investigaciones, ya que si utilizamos GCC con optimización O3, el código en C es mas rápido que ASM. En un futuro podemos realizar otro tipo de experimentación que nos permita entender el comportamiento.

7. Float vs Int

Este experimento consiste en comparar los tiempos de computo del algoritmo de monocromatizar en su versión en Assembler normal, contra el mismo algoritmo modificado, que convierte los datos a punto flotante, y realiza todas las operaciones de máximo en punto flotante. El objetivo de esto es ver cuánto cuesta, en términos de tiempo, convertir enteros a punto flotante para operar, en vez de operar directamente con enteros.

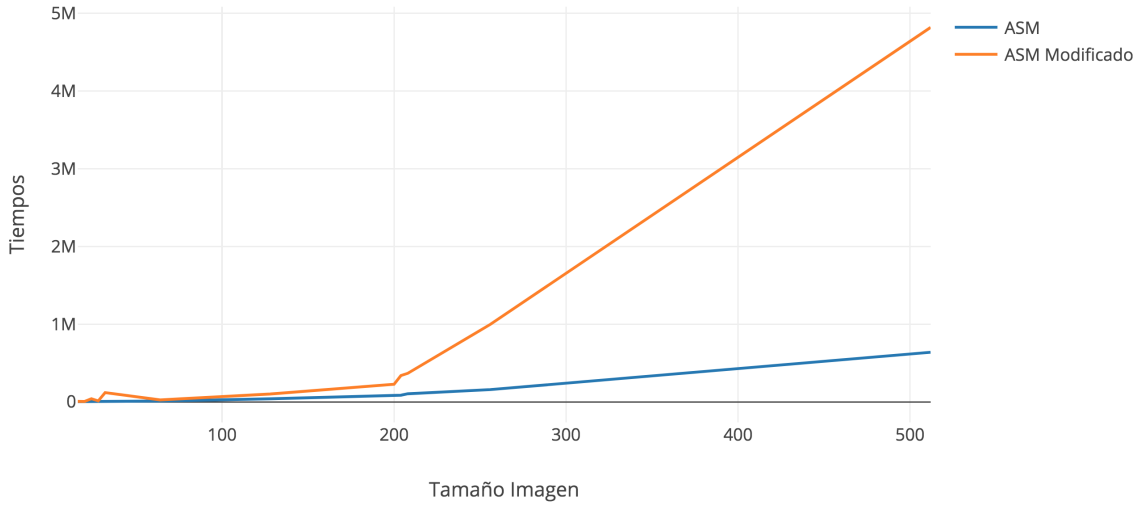

Figura 21: Gráfico que muestra los tiempos para los distintos tamaños de imagen, para las dos implementaciones

Analizando el gráfico obtenido, podemos afirmar que trabajar con punto flotante es considerablemente mas lento que trabajar con enteros. Esto nos da la pauta, que siempre que se pueda trabajar con enteros, vamos a tener un algoritmo mas rápido.

8. Escritura en memoria

Este experimento consiste en comparar los tiempos de computo del algoritmo del efecto ondas en su versión en Assembler normal, contra el mismo algoritmo pero sin escribir en memoria, es decir, sin mover los píxeles modificados a la imagen destino. La imagen generada con esta modificación es simplemente una imagen en negro, ya que no se escribe nada en el espacio reservado para dicha imagen destino.

El objetivo de esto es ver cuánto cuesta, en términos de tiempo, escribir en memoria. Nuestra hipótesis es que tras esta modificación, dicho tiempo se reducirá de forma notable.

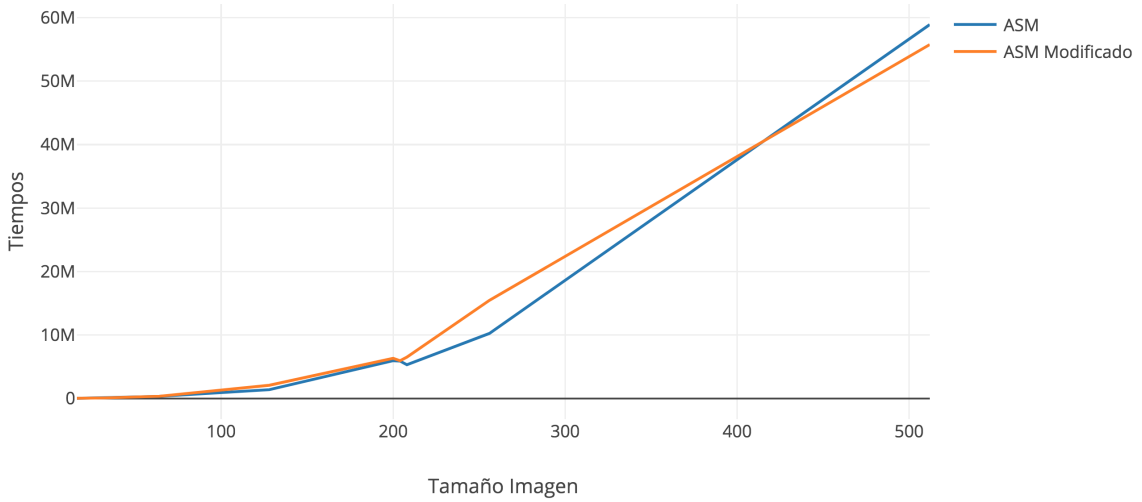

Figura 22: Gráfico que muestra los tiempos para los distintos tamaños de imagen, para las dos implementaciones

Analizando el gráfico, podemos ver que recién a partir de imágenes con tamaños mayores a 420, el algoritmo modificado es mejor al original. A medida que los tamaños son mayores, mas visible se hace esa diferencia. De este modo, nuestra hipótesis parece confirmarse para imágenes a partir de cierto tamaño en adelante. De todas maneras notamos que escribir en memoria no le toma tanto tiempo al cpu. Aunque si dicha diferencia puede llegar a millones de ciclos.

9. División aproximada

Figura 23: La imagen de la izquierda corresponde al filtro con división por 3 en punto flotante. La imagen de la derecha corresponde al filtro con división por 3 aproximada en entero. Se puede apreciar que las diferencias son casi imperceptibles

Dado que a simple vista las diferencias son casi imperceptibles, realizamos un Diff utilizando la herramienta de la cátedra, para poder identificar las diferencias entre las dos imágenes. Usamos una tolerancia 0, pero experimentando con la tolerancia, notamos que usando una tolerancia igual a 17, ya no hay diferencias.

Figura 24: La imagen representa las diferencias entre las imágenes de la *figura 24*, de izquierda a derecha, podemos apreciar las diferencias para las componentes A, R, G y B

Para el filtro de temperatura, reemplazamos la división hecha en punto flotante por una hecha con shifteos, pero aproximada. En la versión original primero fue necesario desempaquetar de word a double word, luego convertir de entero a punto flotante, hacer la divisón en punto flotante, volver a convertir a entero y finalmente empaquetar de double word a word. Realizando la division con shifts, primero nos ahorramos pasar de word, a double word, de convertir a punto flotante y hacer el proceso inverso. Para hacer la división con shifteos se uso el siguiente algoritmo en c:

$a >> 2 + a >> 4 + a >> 6 + a >> 8$

En asembler implica copiar el registro que contiene los a, 4 veces. Para nuestro caso a es la suma de r, g y b, en un registro $xmm0$, que contiene dicha suma para los primeros 4 píxeles en words. Para cada uno hacer un shift con la operación `PSRAW`, y los valores 2, 4, 6 y 8 respectivamente. Y finalmente sumar los 4 registros.

La división hecha con shifts es equivalente a hacer:

$a \times \frac{1}{4} + a \times \frac{1}{16} + a \times \frac{1}{64} + a \times \frac{1}{256}$

Lo cual es equivalente a:

$a \times (\frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \frac{1}{256}) = a \times \frac{85}{256} = a \times 0{,}33203125$

Como dijimos anteriormente, la división no es exacta, pero los resultados varían en hasta 17, comparados con la división exacta, y a los ojos humanos, el resultado es casi el mismo.

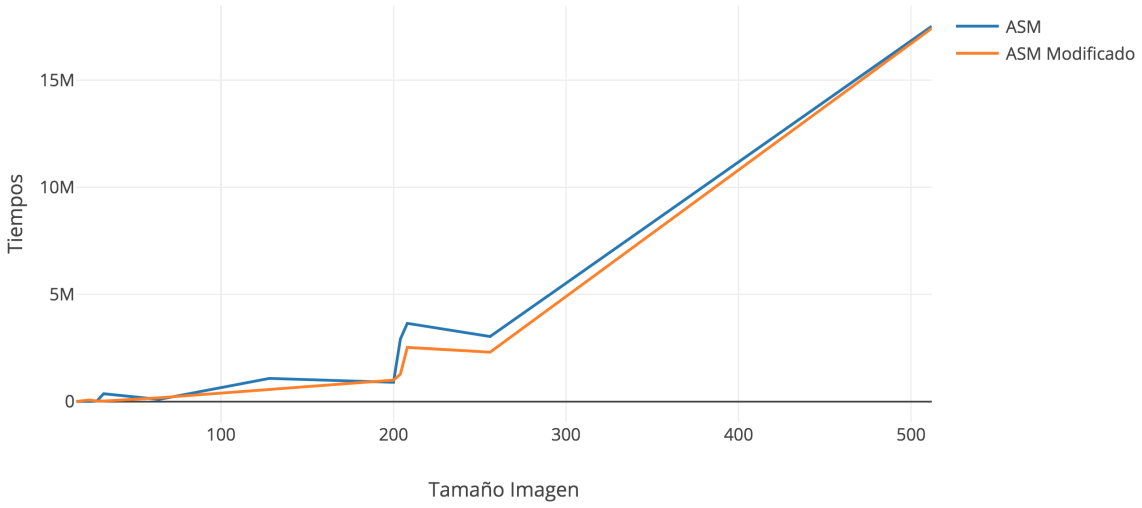

Figura 25: Gráfico que muestra los tiempos para los distintos tamaños de imagen, para las dos implementaciones

Analizando el gráfico, podemos ver que realizar divisiones utilizando la operación Shift, es un poco

mas eficiente, que realizar las divisiones, utilizando operaciones de división. Pero no tanto como esperábamos. Esto posiblemente sea porque estamos reemplazando 5 instrucciones que hacen la división en punto flotante por 12 que la hacen shifts. Aunque estas 12 sean posiblemente instrucciones bastantes rápidas en comparación a las de punto flotante, la operación de punto flotante es 1 sola, y hay solo 2 conversiones adicionales, y las otras dos son un empaquetamiento y un desempaquetamiento que dudamos que tomen mucho tiempo.

Parte IV
Conclusiones

A lo largo de este trabajo pudimos ver como el uso de instrucciones SIMD mejoro la implementación de los filtros en todos los casos en comparación con las implementaciones en el lenguaje C. El trabajar con datos de manera simultánea redujo de manera considerable la cantidad de accesos a memoria, debido a la posibilidad de leer y escribir en la misma de a bloques con la utilización de registros de 16 bytes. Al mismo tiempo, se redujo en gran medida la cantidad de instrucciones ya que con los registros recién mencionados se consiguió procesar varios datos al mismo tiempo.
Si bien los experimentos realizados nos fue de gran ayuda para aclarar dudas y confirmar hipótesis, en algunos casos los resultados no fueron del todo concluyentes y quedaron abiertos a experimentaciones futuras.
En el caso de Blit, por ejemplo, vimos que la versión en C era mas eficiente en términos de tiempo que la de Assembler. Pensamos que esto podría llegar a cambiar para imágenes suficientemente grandes. Otra posibilidad es que la implementación de dicho algoritmo no este programada de la mejor manera.
Por otro lado, en los filtros monocromatizar y temperature obtuvimos resultados muy similares.
En definitiva, si bien en general la velocidad del procesamiento aumenta con las implementaciones en Assembler, lo que se pierde en este caso es en cantidad de líneas de código y simplicidad.