



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Filtros de Imágen

Organización del Computador II
Primer Cuatrimestre - 2015

Integrante	LU	Correo electrónico
Christian Cuneo	755/13	chriscuneo93@gmail.com
Ignacio Lebrero	751/13	ignaciolebrero@gmail.com
Jorge Porto	376/11	cuanto.p.p@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
2. Desarrollo	2
2.1. Blur	2
2.1.1. Implementación en assembler 1	2
2.1.2. Implementación en assembler 2	3
2.2. Merge	5
2.2.1. Implementación en assembler 1	5
2.2.2. Implementación en assembler 2	6
2.3. HSL	8
2.3.1. Implementacion en assembler 1	9
2.3.2. Implementación en assembler 2	10
3. Experimentación	12
3.1. Blur	12
3.2. Merge	13
3.3. HSL	14
3.3.1. implementación	15
3.3.2. variacion 1	16
3.3.3. variacion 2	16
3.3.4. Experimentos Futuros	17
4. Conclusiones	17

1. Introducción

Los filtros de imagen son una herramienta poderosa a la hora de retocar una imagen, usados ampliamente en fotografía, publicidad, videojuegos, etc. Su uso brinda una gamma de opciones para modificar las imagenes de manera que sea mas flexible su edicion o analisis.

En este trabajo practico presentamos los metodos blur, merge y hsl ya existentes y los implementamos en lenguaje de ensamblador. Damos dos implementaciones de cada filtro siendo la segunda una optimizacion de la primera en merge y blur, y una variacion de implementacion C/Assembler a Assembler en hsl. Finalmente realizamos experimentos para comparar el tiempo de computo de los mismos.

El lenguaje C es uno de los más eficientes en cuestión de performance, pero es posible mejorarla implementando las funciones directamente en assembler. En este trabajo se pondra énfasis en las posible ventajas que puede tener un código en assembler con respecto a uno en C, y también las ventajas de usar las e instrucciones SIMD de la arquitectura x86-64.

2. Desarrollo

2.1. Blur

El filtro blur genera una imagen desenfocada artificialmente, que usa un metodo muy simple para lograrlo, a cada pixel se lo promedia con todos sus pixeles vecinos y se guarda ese resultado en el mismo pixel.

2.1.1. Implementación en assembler 1

El principio basico de la implementacion en assembler es que el barrido que se le hace a la imagen se realiza de arriba a abajo y de izquierda a derecha (viendo la imagen siempre como una matriz). Con este metodo se obtiene una ventaja muy interesante, uno reutiliza las filas ya cargadas a la hora de procesar el siguiente pixel, y solo tiene que cargar una fila mas por cada pixel a procesar, esto es una ventaja ya que uno busca una menor lectura de memoria, ya que es una tarea muy costosa. En el pseudocodigo siguiente se vera mas detalladamente el funcionamiento de este metodo:

```

1 for (x = 1 to n - 2):
2   xmm1 <-- img[x-1][0] , img[x][0] , img[x+1][0] , img[x+2][0]
3   xmm2 <-- img[x-1][1] , img[x][1] , img[x+1][1] , img[x+2][1]
4   xmm1 <-- borrarprimero(xmm1)
5   xmm2 <-- borrarprimero(xmm2)
6   xmm1 <-- sumapixels(xmm1)
7   xmm2 <-- sumapixels(xmm2)
8   for (y = 1 to n - 2):
9     xmm0 <-- xmm1
10    xmm1 <-- xmm2
11    xmm3 <-- img[x-1][y+1] , img[x][y+1] , img[x+1][y+1] , img[x+2][y+1]
12    xmm3 <-- borrarprimero(xmm3)
13    xmm3 <-- sumapixels(xmm3)
14    xmm0 <-- xmm0 + xmm1 + xmm2
15    xmm0 <-- promedio(xmm0)
16    img[x][y] <-- xmm0
17  end
18 end

```

Cada pixel contiene 4 valores de 1 byte (R, G, B y A), por lo tanto entran 4 pixels en un registro xmm, entonces cargo directamente 4 pixels de memoria. Hay que tener en cuenta que se cargan los pixels del [x-1] al [x+2], pero en el registro van a quedar al revez, siendo primero el [x+2].

La funcion "borrarprimero" va a utilizar la instruccion PINSRD para insertar 0 en los primeros 4 bytes, eliminando asi los datos de el primer pixel en el registro, que seria el pixel x+2, ya que es el pixel que no nos interesa, porque solo hay que promediar con los vecinos del pixel $\text{img}[x][y]$, es decir los pixeles $\text{img}[i][j]$ con $i \in (x-1, x, x+1)$ y $j \in (y-1, y, y+1)$.

La funcion "sumarpixels" va a sumar los pixeles dentro del registro por valor (R, G, B y A). Para lograr esto sin saturacion se necesita extender (con ceros) los valores de 1 byte a 2 bytes, para poder representar sin nungun problema la suma de todos los valores posibles. Para esto se utiliza la instruccion PUNPCKLBW y PUNPCKHBW (ambas con un registro nulo como source), para guardar en un registro los 2 pixels menos significativos y los 2 pixels mas significativos respectivamente. Se necesita guardar en dos registros ya que al ocupar cada dato el doble, van a entrar la mitad de datos en el registro. Luego de extender los valores se procede a sumar horizontalmente cada registro y luego se suman de forma empaquetada a word los dos registros, y nos queda en el qword menos significativo el pixel suma de los pixels que habia originalmente en el registro.

La función "promedio", dado un registro con la suma (de todos los píxeles a promediar) en el qword menos significativo, va a devolver un píxel con cada valor original dividido por 9, y va a devolver este píxel de forma que quede listo para ser grabado a memoria (quiere decir los R, G, B y A de 1 byte cada uno y ordenados). Para ello primero extiende (con ceros) de word a dword cada valor del píxel, utilizando la instrucción PUNPCKLWD con un registro nulo como source. Luego se convierte cada valor de entero de 32 bits a single-precision floating point, que es también de 32 bits, con la instrucción CVTDQ2PS. Entonces se procede a dividirlos por 9 en forma empaquetada utilizando la instrucción DIVPS, usando como source un registro previamente cargado con cuatro "9" en float. Luego se van a convertir de nuevo a enteros utilizando la instrucción CVTSD2DQ. En este punto ya sabemos que no importe que valores hayan tenido los 9 píxeles sumados, al promediarlos van a dar números entre 255 y 0 únicamente, entonces al convertirlos de nuevo a enteros sabemos que van a quedar valores que van a ocupar como mucho los 8 bits menos significativos de cada dword. Teniendo en cuenta esa información se puede proceder a hacer un shuffle (con la instrucción PSHUFB) para mover cada byte menos significativo de cada dword a la posición correcta del registro, quedando en el dword menos significativo del registro el píxel a guardar, con sus valores en el orden correcto. En esta instrucción se utiliza una máscara específica que se puede ver en la implementación.

Finalmente se graba a memoria el resultado utilizando la instrucción PEXTRD [mmx], xmm, 00b, siendo mmx el puntero al destino donde guardar los datos y xmm el resultado del promedio, se utiliza "00" ya que el resultado de calcular el promedio se encuentra en el dword menos significativo.

Una cuestión a tener en cuenta de esta implementación es que, al ser el array "img" la imagen a procesar y el destino donde guardar el resultado, se tuvo que asignar un espacio de igual tamaño al de "img" para guardar el resultado en ese espacio, ya que si se guarda en el mismo "img" se estarían promediando los píxeles con píxeles que son un promedio de por sí. Entonces se guarda el resultado en este espacio pedido usando malloc, y luego de procesar toda la imagen, se copia todo el contenido de este resultado a la imagen original usando un loop construido con jumps condicionales, y luego se hace un free de esta memoria.

También algo muy importante es que, como se ve en el pseudocódigo, uno siempre carga de a 4 píxeles, descartando el cuarto píxel en el proceso posterior. Esto va a comportarse de forma no deseada al llegar al $x = n - 2$, ya que al cargar el registro estarías agarrando datos que no pertenecen a la fila deseada, esto no es un problema al principio, ya que se estaría cargando el primer píxel de la siguiente fila, lo cual está permitido. Pero al llegar también al $y = n - 2$ uno estaría tratando de cargar algo que no nos pertenece, ya que estaríamos en la última fila de la imagen. Para esto la implementación se encarga de detectar cuando se llega a este caso en particular (etiqueta "lastpixel") y simplemente se cargan los últimos 4 píxeles, y se realiza un PSRLDQ para luego seguir normalmente con el algoritmo.

2.1.2. Implementación en assembler 2

La implementación número 2 hace el mismo procedimiento que la implementación 1, solo que procesa de a 4 píxeles por iteración, para esto se van a necesitar cargar 6 píxeles de la fila de píxeles a procesar, de la siguiente y la anterior, como se ve en la siguiente imagen:

0	1	2	3	4	5	5	4	3	2	3	2	1	0
6	7	8	9	10	11	11	10	9	8	9	8	7	6
12	13	14	15	16	17	17	16	15	14	15	14	13	12

Figura 1: Los píxeles que se deberán cargar al procesar los píxeles marcados con rojo. Se cargarán en 6 xmm como se muestra a la derecha

Luego estos registros se va a extender con ceros los valores de todos los píxeles (con las instrucciones PUNPCKLBW y PUNPCKHBW con source un registro xmm nulo), para ello se usa PUNPCKLBW en los 3 registros de la izquierda; luego PUNPCKLBW y PUNPCKHBW en los de la izquierda, sería lo mismo al revés, ya que hay dos píxeles que se repiten para cada fila:

xmm2	B ₅	G ₅	R ₅	A ₅	B ₄	G ₄	R ₄	A ₄
xmm5	B ₁₁	G ₁₁	R ₁₁	A ₁₁	B ₁₀	G ₁₀	R ₁₀	A ₁₀
xmm8	B ₁₇	G ₁₇	R ₁₇	A ₁₇	B ₁₆	G ₁₆	R ₁₆	A ₁₆

xmm1	B ₃	G ₃	R ₃	A ₃	B ₂	G ₂	R ₂	A ₂
xmm4	B ₉	G ₉	R ₉	A ₉	B ₈	G ₈	R ₈	A ₈
xmm7	B ₁₅	G ₁₅	R ₁₅	A ₁₅	B ₁₄	G ₁₄	R ₁₄	A ₁₄

xmm0	B ₁	G ₁	R ₁	A ₁	B ₀	G ₀	R ₀	A ₀
xmm3	B ₇	G ₇	R ₇	A ₇	B ₆	G ₆	R ₆	A ₆
xmm6	B ₁₃	G ₁₃	R ₁₃	A ₁₃	B ₁₂	G ₁₂	R ₁₂	A ₁₂

Figura 2: Registros luego de desempaquetar

Se extendieron los valores para poder sumarlos sin que se lleguen a saturar, luego se suman respetando las columnas usando PADDW de la siguiente manera:

xmm2	B ₅ + B ₁₁ + B ₁₇	G ₅ + G ₁₁ + G ₁₇	R ₅ + R ₁₁ + R ₁₇	A ₅ + A ₁₁ + A ₁₇	B ₄ + B ₁₀ + B ₁₆	G ₄ + G ₁₀ + G ₁₆	R ₄ + R ₁₀ + R ₁₆	A ₄ + A ₁₀ + A ₁₆
xmm1	B ₃ + B ₉ + B ₁₅	G ₃ + G ₉ + G ₁₅	R ₃ + R ₉ + R ₁₅	A ₃ + A ₉ + A ₁₅	B ₂ + B ₈ + B ₁₄	G ₂ + G ₈ + G ₁₄	R ₂ + R ₈ + R ₁₄	A ₂ + A ₈ + A ₁₄
xmm0	B ₁ + B ₇ + B ₁₃	G ₁ + G ₇ + G ₁₃	R ₁ + R ₇ + R ₁₃	A ₁ + A ₇ + A ₁₃	B ₀ + B ₆ + B ₁₂	G ₀ + G ₆ + G ₁₂	R ₀ + R ₆ + R ₁₂	A ₀ + A ₆ + A ₁₂

Figura 3: Registros luego de sumar, sumamos en xmm0, xmm1 y xmm2 ya que no se van a necesitar para el ciclo siguiente.

A partir de esas sumas se pueden obtener los valores resultantes para los 4 pixeles de la iteracion de la siguiente forma:

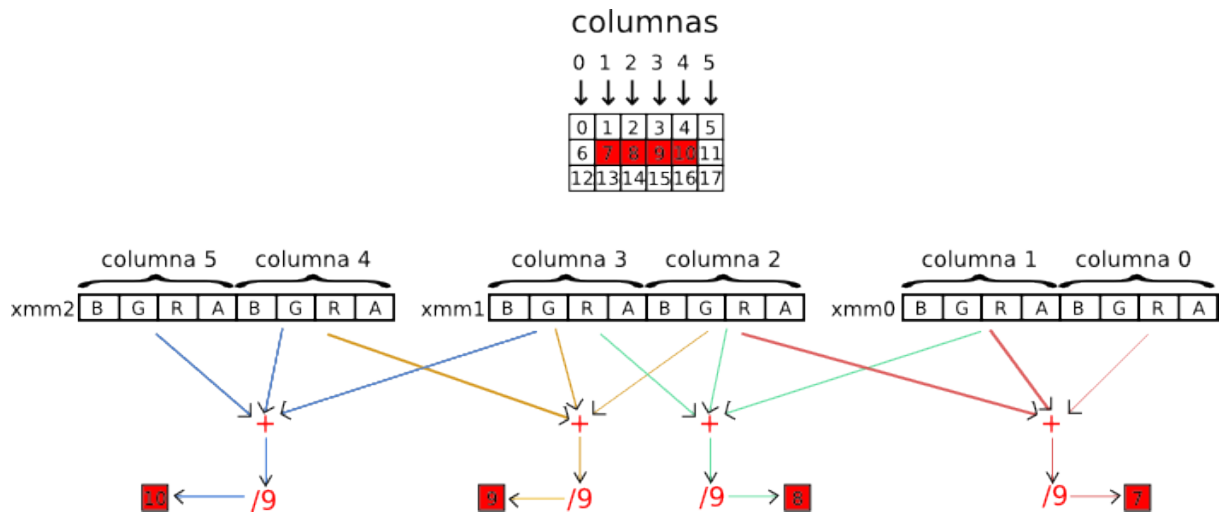


Figura 4: Pasos a seguir para cada pixel.

En el proceso para cada pixel, primero se extenderan los valores a dword con el mismo procedimiento explicado anteriormente (esto se hace para poder, mas adelante, pasarlo a single precision floating point in place). Luego se la operacion suma entre los registros se realiza con la instruccion PADDW, luego se convierten los valores a FP con CVTDQ2PS, se divide por 9 usando DIVPS, y se convierte de nuevo a enteros CVTQ2DQ. Por ultimo se hace el mismo shuffle que en la implementacion 1, para terminar grabando el resultado al bloque de memoria temporal.

2.2. Merge

El merge nos permite a partir de dos imágenes y un valor entre cero y uno, obtener una combinación de estas ultimas según la proporción indicada por el valor.

2.2.1. Implementación en assembler 1

Se recibe por parámetro dos punteros a dos imágenes almacenadas en memoria como una matriz de pixeles. Como los pixeles de estas imágenes ocupan 4 byte, y la cantidad de pixeles de las mismas es multiplo de 4, utilizando los registros xmm podemos traer de a 4 pixeles. Luego incrementamos los punteros a la imagen en 16 bytes y volvemos a traer los pixeles de memoria. De esta manera tenemos

un ciclo principal cuya cantidad de iteraciones es la cantidad de pixeles dividido cuatro. Cada una de estas consiste en levantar de memoria cuatro pixeles y almacenarlos en un registro xmm, cuyo contenido puede verse en la figura 5.

B	G	R	A	B	G	R	A	B	G	R	A	B	G	R	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 5: Contenido del registro xmm al levantar de memoria cuatro pixeles

Luego utilizando la instrucción `pshufb` y una máscara apropiada ordenamos su contenido para que quede como se muestra en la figura 6

B	B	B	B	G	G	G	G	R	R	R	R	A	A	A	A
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Figura 6: Contenido del registro xmm luego de utilizar la instrucción `pshufb`

Utilizamos las instrucciones de desempaqueado de SIMD y un registro xmm lleno de ceros, para desempaquear la parte alta y baja, obtenemos registros xmm con el contenido como se muestra en la figura 7.

0	B	0	B	0	B	0	B	0	G	0	G	0	G	0	G
0	R	0	R	0	R	0	R	0	A	0	A	0	A	0	A

Figura 7: Contenido resultante de desempaquear parte alta y baja

Luego volvemos a desempaquear parte alta y baja de los dos registros obtenidos, y obtenemos cuatro registros xmm como se indica en la figura 8.

0	0	0	B	0	0	0	B	0	0	0	B	0	0	0	B
0	0	0	G	0	0	0	G	0	0	0	G	0	0	0	G
0	0	0	R	0	0	0	R	0	0	0	R	0	0	0	R
0	0	0	A	0	0	0	A	0	0	0	A	0	0	0	A

Figura 8: Contenido resultante de desempaquear parte alta y baja nuevamente

Utilizando la instrucción `cvtdq2ps` convertimos los cuatro valores de los registros de la figura 8, excepto el que contiene los bytes de transparencia(A), a tipo flotante. El objetivo es multiplicar cada color por value. Para hacer esto, previo al ciclo, utilizando la instrucción `shufps`, conseguimos en un registro xmm cuatro valores de tipo flotante con el valor que indica el índice de combinación de imágenes pasado por parámetro, y en otro cuatro valores con 1-value tal como se indica en la figura 9.

value	value	value	value
1 - value	1 - value	1 - value	1 - value

Figura 9: Contenido de los registros utilizados para multiplicar por value los colores

Utilizando la instrucción de `simd mulps`, multiplicamos los colores transformados a tipo flotante por value.

Se repite el procedimiento para la segunda imagen, excepto que no se desempaqueta el byte de transparencia(A), ya que solo interesa los bytes de transparencia de la primera imagen. En este caso multiplicamos por 1-value. Sumamos con la instrucción `addps` los valores obtenidos en la multiplicación, para los colores azul, verde y rojo. Luego convertimos a enteros de 32 bit, y con la instrucciones `packusdw`, y `packuswb`

empaquetamos de forma que queden en un xmm los bytes en orden azul, verde, rojo y transparencia. Este ultimo se consideran los bytes de la primera imagen almacenados en un registro xmm como se muestra la figura 8. Finalmente con la instrucción `pshufb` ordenamos los colores para que queden en el mismo orden en que ingresaron, y escribimos en memoria el resultado.

2.2.2. Implementación en assembler 2

Es analoga a la implementación anterior, pero en este caso se hacen la suma y multiplicación en numeros enteros. Se tiene en cuenta que la multiplicación de dos enteros, da como resultado un entero que puede ocupar el doble de tamaño. Para conseguir implementar el algoritmo con instrucciones enteras y obtener el mismo resultado(aunque se pierdan decimales), el metodo es multiplicar a todo por 256, hacer las multiplicaciones en enteros y finalmente dividir por 256.

Antes de empezar el ciclo, multiplicamos en punto flotante 256 y value, y al resultado lo convertimos a enteros. En este proceso se pierden decimales. Como value es un numero entre cero y uno, al multiplicar por 256 y pasarlo a enteros, tenemos un valor entre 0 y 256, con lo que ocupa menos de un byte de tamaño. Utilizando la instrucción `pshufb` y una mascara, almacenamos en un registro xmm 8 replicas de este valor. Luego hacemos la resta en enteros entre 256 y el valor obtenido en la multiplicación. Nuevamente tenemos un valor entero entre 0 y 256 que entra en un byte, almacenamos 8 replicas del mismo en un registro xmm tal como se ve en la figura 10. Estos registros seran utilizados para multiplicar los colores.

256-256*value	256-256*value	256-256*value	256-256*value	256-256*value	256-256*value	256-256*value	256-256*value
256*value	256*value	256*value	256*value	256*value	256*value	256*value	256*value

Figura 10: Contenido de los registros utilizados para multiplicar en enteros

El ciclo comienza igual que la implementación anterior, trayendo a un registro xmm 4 pixeles de la primer imagen desde memoria, ordenandolos con `pshufb` con una mascara para que queden como se indica en la figura 6. Utilizando `punpcklbw` y `punpckhbw` y un registro lleno de ceros obtenemos dos registros xmm, como pueden verse en a figura 7. Utilizando `punpcklwd` y `psrldq`, obtenemos finalmente tres registros xmm con el contenido tal como se indica en la figura 11.

0	0	0	A	0	0	0	A	0	0	0	A	0	0	0	A		
0	0	0	0	0	0	0	0	0	0	B	0	B	0	B	0	B	
0	R	0	R	0	R	0	R	0	R	0	G	0	G	0	G	0	G

Figura 11: Contenido de los registros luego de hacer operaciones de desempquetado

Utilizamos las instrucciones `pmullw` y `pmulhuw` para obtener las partes altas y bajas de la multiplicación de los colores(sin contar la transparencia) con el valor 256*value(convertido a entero) que se encuentra almacenado en un registro como se indica en la figura 10. Finalmente utilizamos la instrucción `punpcklwd` y `punpckhwd` para obtener el resultado final de la multiplicación. De esta manera realizamos multiplicaciones de enteros de 16 bit, con lo que obtenemos enteros de 32 bit. El registro que contiene los bytes de transparencia queda inalterado.

Se repite el procedimiento para la segunda imagen, pero sin guardar los bytes de transparencia. Dividimos los enteros por 256 utlizando un corrimiento a derecha de 8 bit con la instrucción `psrld xmm, 8`. Y luego sumamos en enteros con las instrucciones `paddb`. Se decide dividir primero y luego sumar para evitar una posible saturación. Con las instrucciones `packusdw` y `packuswb` empaquetamos el resultado, de forma que queden en un xmm los bytes en orden azul, verde, rojo y transparencia. Finalmente con la instrucción `pshufb` ordenamos los colores para que queden en el mismo orden en que ingresaron, y escribimos en memoria el resultado.

2.3. HSL

El espacio RGB que usamos hasta ahora está dado por un cubo donde cada componente corresponde a la intensidad de los colores primarios de la luz (rojo, verde y azul), para este filtro nos trasladamos al espacio HSL, este está dado por tres componentes: HUE(color), Saturation(Saturación) y Lightness(Luminosidad). Este espacio se representa graficamente como un bicono donde cada punto está determinado con: altura de la circunferencia a la que pertenece, radio de dicha circunferencia y ángulo dentro de la circunferencia, donde el color corresponde al ángulo, la saturación al radio y la luminosidad a la altura. El filtro consta de 3 etapas: (1) Transformación del espacio RGB a HSL del pixel, (2) suma de componentes y (3) transformación del espacio HSL a RGB cuyas formulas son las siguientes:

1. H:

0 si $c_{max} = c_{min}$
 $60 * ((g - b)/d + 6)$ si $c_{max} = r$
 $60 * ((b - r)/d + 2)$ si $c_{max} = g$
 $60 * ((r - g)/d + 4)$ si $c_{max} = b$
por ultimo si $h \geq 360$ entonces $h = h - 360$

L: $(c_{max} + c_{min}) / 510$

S:

0 si $c_{max} == c_{min}$
 $d / (1 - f_{abs}(2 * l - 1)) / 255,0001$ caso contrario
para este caso se debe dividir contra 255.0001f para que s supere 1, evitando la propagación de errores.

Donde $c_{max} = \max(b, g, r)$, $c_{min} = \min(b, g, r)$ y $d = c_{max} - c_{min}$

2. H:

$h + hh + 360$ si $h + hh \geq 360$
 $h + hh - 360$ si $h + hh < 0$
 $h + hh$ caso contrario

S:

0 si $s + ss < 0$
1 si $s + ss \geq 1$
 $s + ss$ caso contrario

L:

0 si $l + ll < 0$
1 si $l + ll \geq 1$
 $l + ll$ caso contrario

donde hh, ll y ss son valores pasados por parametro

3. RGB:

$r=c \ g=x \ b=0$ si $0 \leq h$ y $h < 60$
 $r=x \ g=c \ b=0$ si $60 \leq h$ y $h < 120$
 $r=0 \ g=c \ b=x$ si $120 \leq h$ y $h < 180$
 $r=0 \ g=x \ b=c$ si $180 \leq h$ y $h < 240$
 $r=x \ g=0 \ b=c$ si $240 \leq h$ y $h < 300$
 $r=c \ g=0 \ b=x$ si $300 \leq h$ y $h < 360$

Escala:

$b = (b + n) * 255$
 $g = (g + n) * 255$
 $r = (r + n) * 255$

donde:

$c = (1 - f_{abs}(2 * l - 1)) * s;$
 $x = c * (1 - f_{abs}(f_{mod}(h/60, 2) - 1))$
 $m = l - c/2$

2.3.1. Implementación en assembler 1

OBS: Para esta implementación usamos dos funciones provistas por la catedra (rgbTOhsl — hslTOrgb) y nos concentramos en (2), donde el algoritmo general va a recorrer la imagen como si fuera un vector, tomando pixel por pixel, transformandolo a HSL, luego procesandolo y finalmente transformandolo a RGB para devolverlo a memoria (para mas información sobre otras posibles implementaciones ver sección 4).

Recibimos como parametros una imagen representada en memoria de la misma manera que en los anteriores filtros, ancho, alto y tres valores hh, ss y ll, los cuales modificaran los 3 elementos de la representacion HSL del pixel. Primero acomodamos los datos de entrada de manera que nos queden dentro de un registro xmm, dado que cada dato ocupa 16bits los acomodamos de manera tal que los 3 elementos queden empaquetados.



Figura 12: Organización de valores de entrada en registro XMM

A continuación movemos a un registro xmm el valor que usaremos como limite superior de la suma para h por un lado y s y l por otro ya que las modificaciones seran diferentes para los dos casos, como limite inferior simplemente colocaremos 0 en un registro usando pxor.

Dado que la imagen está representada linealmente en memoria vamos a recorrerla como un vector dentro de un ciclo principal que se movera de a un pixel. Una vez dentro del ciclo procedemos a llamar a rgbTOhsl y movemos el pixel resultante a un registro xmm.

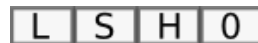


Figura 13: Contenido del registro XMM

Ahora sumamos los valores hsl con los que teniamos como parametro con addps(a) para luego revisar los limites.



Figura 14: Contenido del registro XMM

Para realizar las comparaciones correspondientes primero generamos los vectores que sumados o restados nos den los resultados que queremos en cada caso, para esto tomamos por un lado un registro con el valor 360(b) en la posición de h y procedemos al primer caso de h, para esto aplicamos `cmpleps` para crear una máscara(c) en donde si h es mayor a 360 nos quedarán 1s en los valores donde se debería ubicar h o 0s en caso contrario, luego aplicamos `pand` usando (b) y (c) para finalmente restarlo a (a), dando como resultado $h + hh$ ó $h + hh - 360$ dependiendo del caso, a continuación veremos como se aplica esta idea a s y a l.

0	0	360	0
L + LL	S + SS	H + HH	0

`cmpleps`

0	0	1	0
0	0	360	0

`pand`

0	0	360	0
L + LL	S + SS	H + HH	0

`subps`

L + LL	S + SS	H + HH - 360	0
--------	--------	--------------	---

Figura 15: ejemplo de estados usando `cmp` y `and` para el caso h mayor a 360

Para s y l, como su resultado debe ser 1 en caso de pasar el límite, simplemente colocamos $s+ss-1$ y $l+ll-1$ empaquetados en un registro aparte pero en las mismas posiciones que $s+ss$ y $l+ll$ respectivamente, luego aplicamos la misma idea que en la figura 10 pero comparando contra 1 en lugar de 360 y finalmente sumando $s+ss-1$ o $l+ll-1$ según corresponda para lograr que quede 1 ó $s+ss/l+ll$ en los valores finales. Los casos de los límites inferiores que se realizan inmediatamente después en el código son análogos pero comparando contra 0s, dando una sucesión de estados con la cual al llegar al final, (1) tendrá los valores $l+ll$ ó 1 ó 0, $s+ss$ ó 1 ó 0, $h+hh$ ó $h+hh+360$ ó $h+hh-360$ y 0 respectivamente en sus posiciones empaquetadas.

2.3.2. Implementación en assembler 2

Para esta implementación reutilizamos el código de la primera y nos concentramos en implementar (1) y (3), cuyas reglas son las siguientes:

reglas.....

Transformación del espacio RGB a HSL

Como parámetros tenemos el puntero al pixel de la matriz que queremos convertir y un vector de floats donde guardar la versión hsl del pixel. Primero levantamos el pixel a un registro de propósito general y lo movemos de a bytes a otros registros para separar los elementos RGBA del pixel, como cada uno ocupa 1B simplemente movemos entre registros de 1B y luego shifteamos al que contiene todo el pixel 1B a la derecha hasta que queden todos separados.

A continuación pasamos a calcular el máximo entre RGB, para esto acumulamos uno de los tres valores y lo comparamos contra los otros acarreado siempre al mayor, análogamente para calcular el mínimo.

Luego calculamos D, para esto simplemente colocamos el máximo en otro registro y le restamos el mínimo.

Después calculamos H, para esto separamos los casos posibles en donde simplemente seteamos los registros que luego serán operados, esto se logra comparando el máximo contra los posibles valores, saltando, seteando los registros según los valores que les correspondan, convirtiéndolos en valores de punto flotante con `cvtdq2ps` y finalmente utilizando `subps`, `divps`, `addps` y `mulps` para llegar a la fórmula que les

corresponda.

$$60 * ((d-b)/d + 4)$$

Figura 16: (contenido del registro xmm al final del caso 3, para mas informacion sobre los casos ver el codigo fuente)

por ultimo revisamos que no haya caido fuera de rango, para esto aplicamos la misma técnica que en la primera implementacion usando una comparacion y luego aplicando pand y subps para llegar a h ó h-360.

Ahora calculamos L, para esto tomamos el maximo y le sumamos el minimo, lo pasamos a un registros xmm, lo convertimos y aplicamos una division contra una posicion en memoria que contiene 510.

Por ultimo queda calcular S, en caso de que sea igual al minimo simplemente lo seteamos en 0, caso contrario pasamos a efectuar las cuentas. Movemos d a un registro xmm y lo convertimos a punto flotante, luego usamos addps, mulps y subps para llegar a $2^*l - 1$ donde aplicamos pand contra una mascara que contiene todos 1 excepto en el bit mas significativo, de esta manera seteamos en 0 el bit de signo de la representación del float que contiene $2^*l - 1$, convirtiendolo en $\text{abs}(2^*l - 1)$. Finalmente dividimos a d por $\text{abs}(2^*l - 1)$ y luego por 255.0001 desde una direccion en memoria que lo contiene.



Figura 17: Representación del punto flotante en memoria

Terminado lo anterior procedemos a empaquetar los datos con punpckldq de manera que dentro del registro quede LHSa para que cuando lo volquemos a memoria sea de la forma ASHL.

Transformación del espacio HSL a RGB

Como parametros tenemos un vector de floats donde se encuentra la version hsl del pixel y el puntero al pixel de la matriz donde queremos colocar su version RGB. Primero levantamos el pixel a un registro xmm y separamos sus componentes entre otros registros xmm usando punpckldq, punpckhdq, psrldq y psllldq.

Luego calculamos C, para esto levantamos de memoria un 1 a un registro xmm y pasamos a realizar las cuentas usando addss, mulss y subss hasta llegar a 2^*l-1 donde aplicamos pand contra una mascara de la misma que en la transformacion anterior para lograr $\text{abs}(2^*l - 1)$, luego le restamos a 1 el resultado y lo multiplicamos por S.

A continuación calculamos X, para esto pasamos el valor 60 a un registro xmm, lo convertimos a float y dividimos H por 60, luego para calcular $\text{fmod}(h/60, 2)$, simplemente restamos 60 hasta que el numero sea menor a 2 ya que como el primer operando es $h/60$, ese numero estara entre 0 y 60, como siguiente operacion resto y aplico la misma idea de mascara para lograr $\text{abs}(\text{fmod}(h/60, 2) - 1)$ y finalmente resto 1 y multiplico por C.

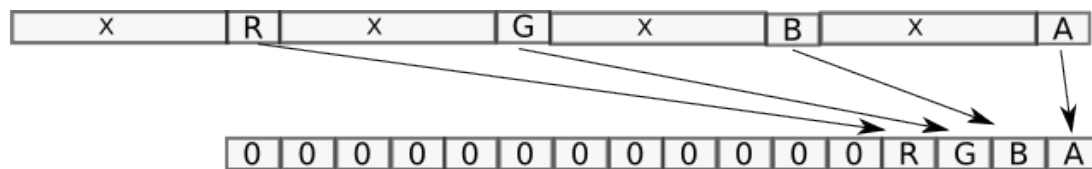
En este punto nos queda calcular M y luego ordenar lo que ya calculamos, empecemos con lo primero. Simplemente copio el valor C a otro registro, lo dividimos por dos y se lo restamos a l.

Ahora vamos a calcular RGB, en esta parte nos dedicamos a ver a que valor(r, g o b) corresponden X y C respectivamente, para esto preguntamos linealmente, aumentando de a 60 en que caso cae H, sumandole 60 y comparando contra H con comiss para luego efectuar un jnb de ser necesario. Dentro de cada caso se tiene un registro xmm que contiene a c y un registro de proposito general que contiene a X y, dependiendo del caso, se shiftea c hasta el lugar que corresponda a la componente a la que debe ir y luego se aplica pinsrd para insertar X en la posicion que le corresponda.

Finalmente nos queda calcular la escala, para esto primero colocamos M en un registro xmm, lo shiftea-

mos y sumamos con `pslldq` y `addps` contra `m` hasta que queden tres valores iguales empaquetados en la posiciones donde deben ir `r`, `g` y `b`, a continuacion se suma este registro con `addps` al que contenia a `C` y `X` y se lo multiplica por 255 contra un registro que previamente habia levantado ese valor de memoria, para lograr $(X + m) * 255$ en cada una de las componentes. Luego se inserta `A` en la ultima posicion y se convierte a entero truncando el valor, se tomo esta decisión ya que la conversión normal redondeaba al entero más cercano.

Para ordenar los datos(ya que deben tener 1B cada uno y a esta altura tienen 4B) aplicamos `pshufb` sobre el registro que contiene `RGBA` para pasar dichos valores a 1B(tomando solo el primer byte de estos) en sus posiciones finales y luego moverlo a memoria(figura 14).

Figura 18: Uso de `pshufb`

3. Experimentación

Se llevaron a cabo experimentos para comparar el tiempo de computo de las distintas versiones de los filtros, en assembler y en C, con distintas optimizaciones. Se estudio el tiempo de computo para distintos parámetros de los filtros, y utilizando distintos conjuntos de imágenes: Normal(incluyen todos los colores), Black(imágenes negras), White(imágenes blancas).

3.1. Blur

En las siguientes figuras encontramos los resultados obtenidos para el filtro blur:

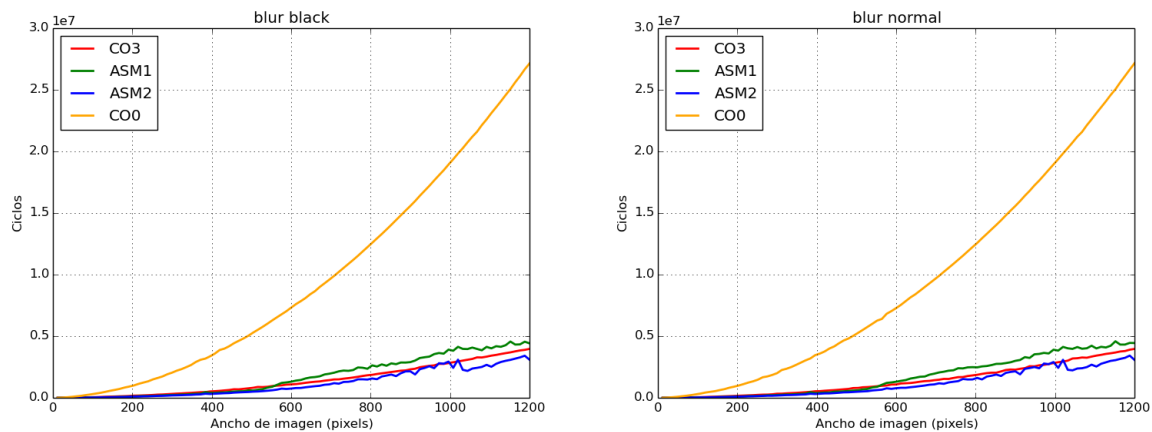


Figura 19: Rendimiento del filtro blur para imágenes negras

Figura 20: Rendimiento del filtro blur para imágenes normales.

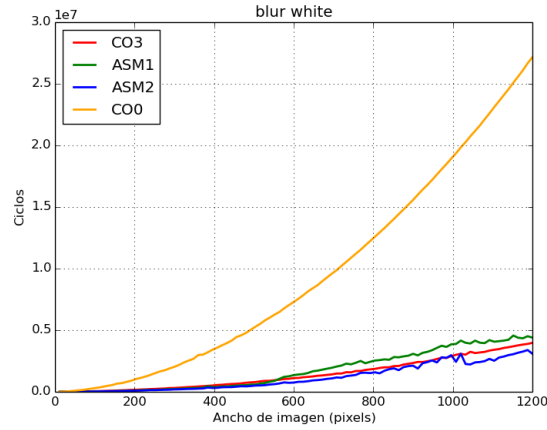


Figura 21: Rendimiento del filtro blur para imágenes blancas.

Las dos implementaciones en assembler, utilizan una memoria temporal del tamaño de la imagen para ir almacenando los resultados parciales. Como trabajamos con imágenes pequeñas, esto no resulta costoso, pero podría ser evitado con un algoritmo mas complejo.

La segunda implementación en assembler en comparación con la primera resulta tener un rendimiento un poco mejor. Esto se debe a que esta ultima realiza tres lecturas de memoria y una escritura para escribir en la memoria temporal un pixel, en cambio la segunda utiliza cuatro lecturas y cuatro escrituras de memoria para escribir en la memoria temporal cuatro pixeles. Esta ultima podría ser mejorada para que con una escritura escribir los cuatro pixeles en memoria, en lugar de hacerlo con accesos separados.

La implementación en C sin optimización resulta ser mucho mas costosa que las otras, lo cual se puede deber a que por cada color realiza una suma y un acceso a memoria, lo que implica una cantidad de accesos a memoria muy grande. La implementación en C optimizada resulta tener un rendimiento mejor que una version de assembler, pero peor que el otro.

Al no tener saltos condicionales, obtenemos el mismo rendimiento para los distintos tipos de imágenes.

3.2. Merge

En las siguientes figuras encontramos los resultados obtenidos para el filtro merge:

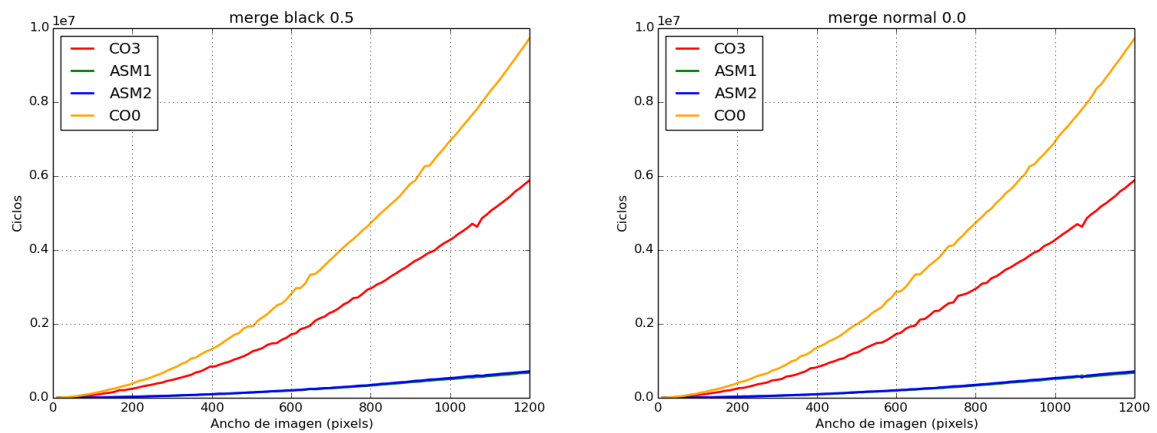


Figura 22: Rendimiento para un value de 0.5, imágenes negras. Figura 23: Rendimiento para un value de 0.0, imágenes normales.

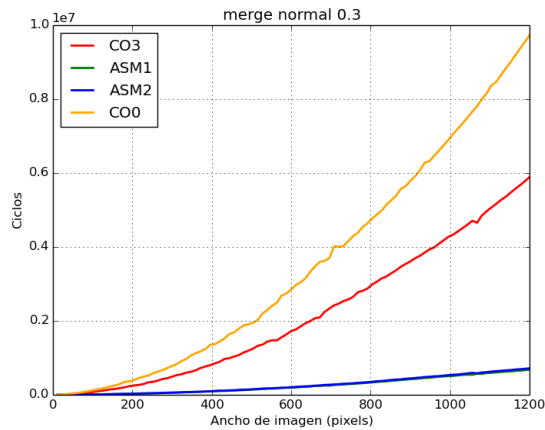


Figura 24: Rendimiento para un value de 0.3, imágenes normales.

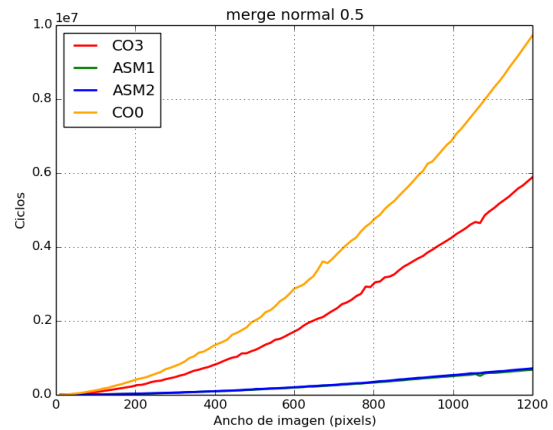


Figura 25: Rendimiento para un value de 0.5, imágenes normales.

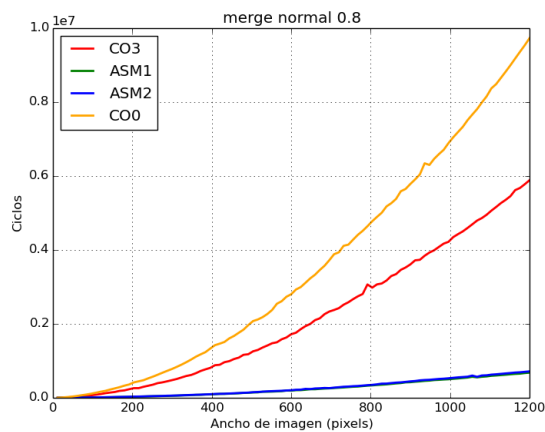


Figura 26: Rendimiento para un value de 0.8, imágenes normales.

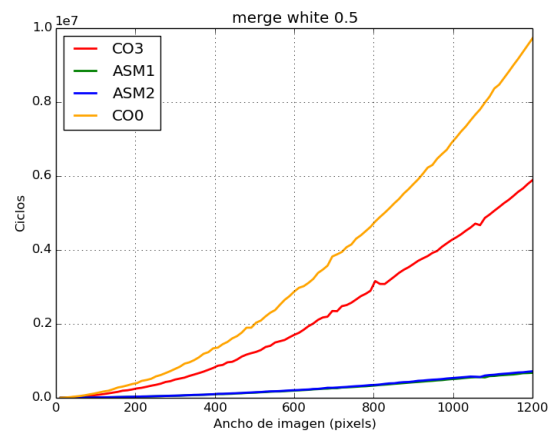


Figura 27: Rendimiento para un value de 0.5, imágenes blancas.

Podemos ver que la implementación en C tiene un rendimiento menor, aun utilizando un compilador que optimiza. Esto se debe a que la imágenes se encuentran almacenadas en memoria en forma de matrices de pixeles, en la implementación en assembler se traen desde memoria cuatro pixeles a un registro xmm, a diferencia la implementación en C, en la cual se hace un acceso a memoria por color. Con lo que la cantidad de accesos a memoria es mucho menor en la implementación en assembler, lo que implica un tiempo de computo mucho menor. Además en la implementación en assembler, utilizando las instrucciones simd se hacen multiplicaciones y sumas de a cuatro colores, a diferencia de la C, en la estas operaciones se hacen color por color.

Tanto las implementaciones en C como en assembler de este filtro presentan el mismo rendimiento para los distintos tipos de imágenes, y solo dependen de la cantidad de pixeles a procesar. Esto se debe a que ambas implementaciones no presentan saltos condicionales.

Para las implementaciones en assembler, no se aprecia una diferencia significativa en el tiempo de computo al operar con enteros o con floats. La imagen resultante para enteros tiene un error un poco mayor, pero esta diferencia no es apreciable a simple vista.

Cuando la cantidad de pixeles a procesar es grande, la diferencia de tiempo de computo entre la implementación en C y su version en assembler.

3.3. HSL

Para experimentar con este filtro primero analizamos un poco el codigo. Este se divide en 3 etapas: (1)RGB a HSL, (2)suma y (3)HSL a RGB.

3.3.1. implementación

Analizando (2) podemos observar que realiza siempre la misma cantidad de operaciones dado que no contiene ningún salto condicional, por lo que podemos asumir que es constante (ver Figuras 22 y 23).

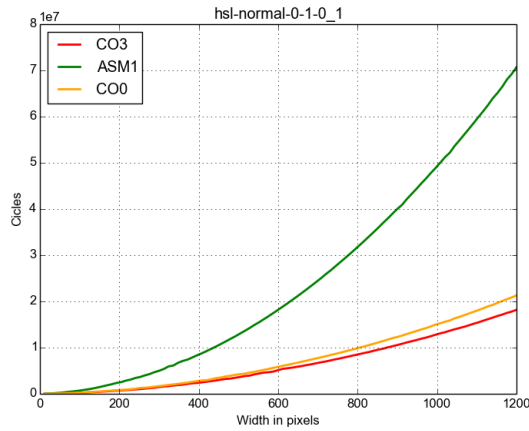


Figura 28: Rendimiento para un H de 0, imágenes normales.

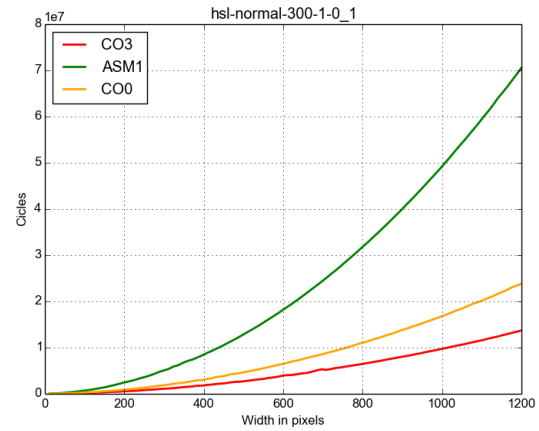


Figura 29: Rendimiento para un 0 de 300, imágenes normales.

La implementación de (2) en HSL1 si bien utiliza operaciones empaquetadas como addps o cmpleps para operar h o l y s al mismo tiempo, tiene un rendimiento mucho menor que la implementación en C con O0 y O3 (ver figure 22 y 23) ya que es la implementación standard del filtro, en la sección 4 proponemos una variación que podría aumentar este rendimiento considerablemente.

Ahora analisemos HSL2, mas en particular (1). Con las imagenes normales, se esperó que los tiempos de corrida fueran mejores en terminos de rendimiento que las implementaciones de C con optimización pero en lugar de eso se puede observar que solo corre más rapido en los casos donde hh es 0 o 360 (ver figuras 24, 25 y 26), suponemos que se debe a que este cacheando el acceso a memoria debido a que el valor no cambia en la suma ya que: $h + hh \geq 360 \Rightarrow h = h + hh - 360$, reemplazando hh por 360 nos queda: $h + 360 - 360 = h$ entonces no varia y dado que no estamos modificando s y l, el valor que querramos asignar al momento de aplicar "movdqu [dir], xmm", donde xmm contiene los valores l, h y s finales, será el mismo que al principio, por lo que no habria necesidad de efectuarlo y el valor que fue cacheado al momento de levantar memoria por primera vez continuara cacheado, evitando acceder nuevamente a memoria.

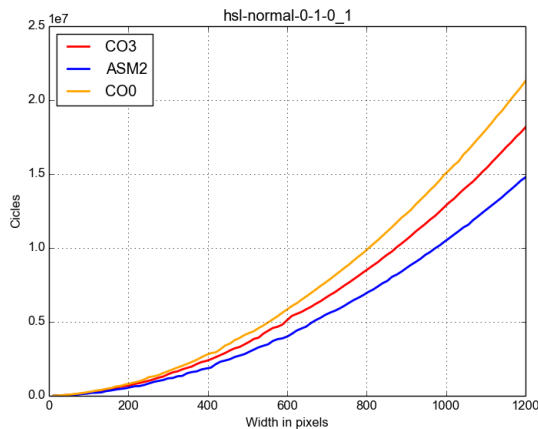


Figura 30: Rendimiento para un H de 0, imágenes normales.

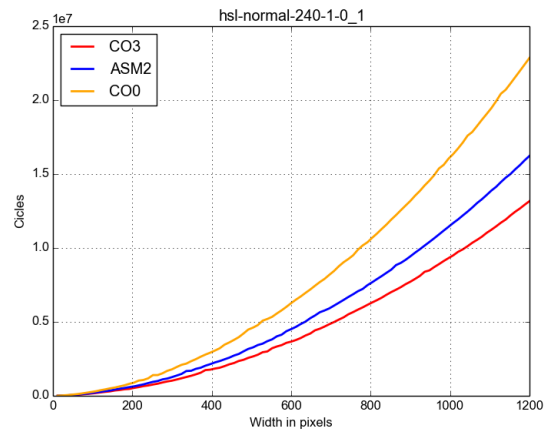


Figura 31: Rendimiento para un H de 240, imágenes normales.

El caso donde el maximo y el minimo de r, g y b son iguales realiza menos operaciones dado que por las formulas que conlleva calcular H y S, en este caso simplemente colocara un 0 en H y continuara a calcular L y por ultimo solo colocara un 0 en S siendo mucho mas corto el proceso.(ver figuras 27 y 26)

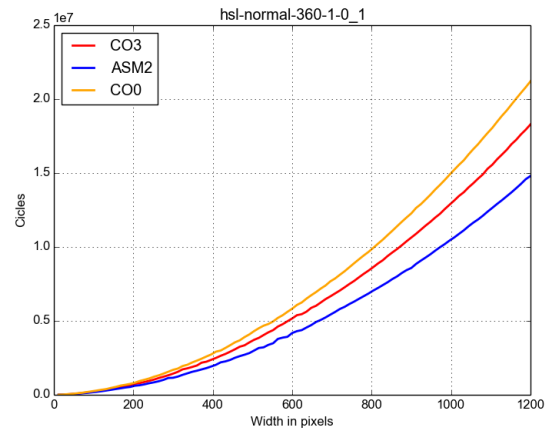
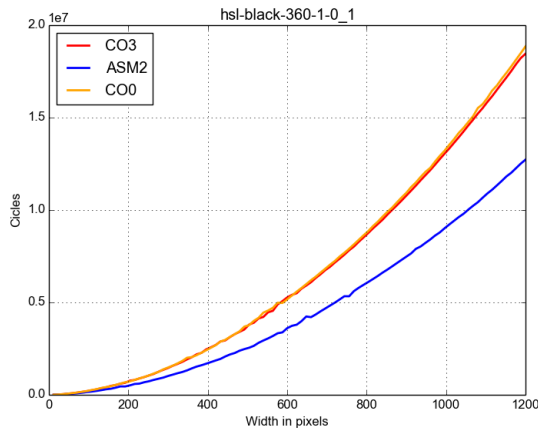


Figura 32: Rendimiento para un H de 360, imágenes negras

Figura 33: Rendimiento para un 0 de 360, imágenes normales.

3.3.2. variacion 1

Probamos cambiar dentro de la función `calcRGB` la manera de ordenar los datos, en lugar de aplicar `pslldq` y luego `pins` usamos `pshufd`, dando como resultado un rendimiento casi parecido donde la variacion fue un poco mas lenta (figuras 28 y 29)

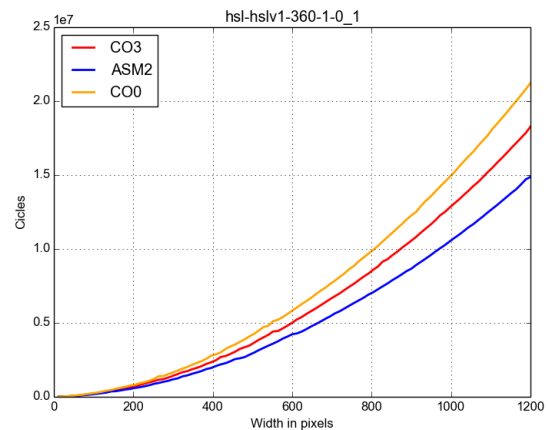
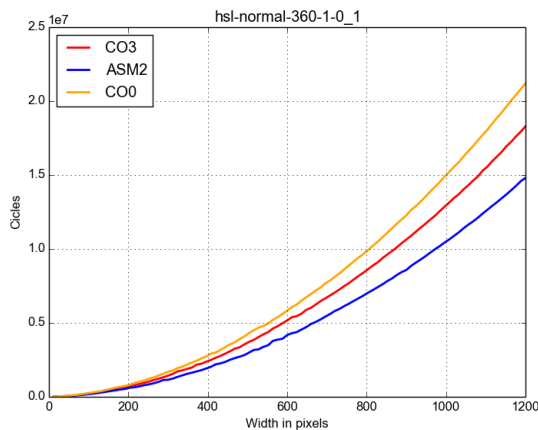


Figura 34: Rendimiento para un H de 360, imágenes normales.

Figura 35: Rendimiento para un H de 360, imágenes normales.

3.3.3. variacion 2

Como segunda variacion decidimos cambiar la manera en que esta implementada la funcion `fmod` dentro de `calcX`.^{en} (3) la cual hasta el momento resta 60 hasta que el valor sea menor a 0. La propuesta fue usar la siguiente formula: $mod(n, m) = n - (parteentera(n/m) * m)$ usando `cvtts2si` para truncar la parte decimal del dato y luego `cvtsi2ss` para devolverlo a float. Esta variacion bajo el rendimiento dado que la conversion de float a entero y viceversa es muy costosa. (ver figuras 30 y 31)

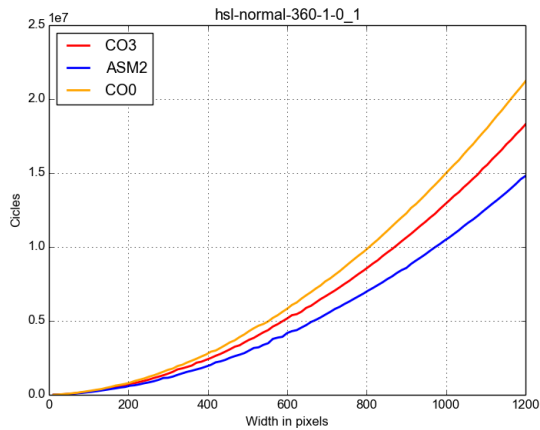


Figura 36: Rendimiento para un H de 360, imágenes normales.

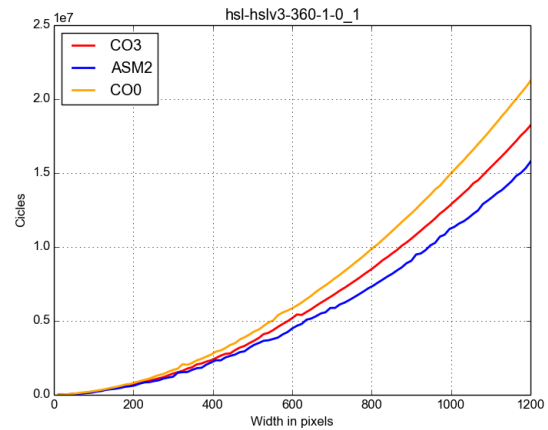


Figura 37: Rendimiento para un H de 360, imágenes normales, variacion 2.

3.3.4. Experimentos Futuros

1. Si observamos (3) podemos llegar a tener mas operaciones mientras h sea mayor ya que la funcion `calcRGB` pregunta linealmente si h pertenece a un intervalo determinado, incrementandolo de a 60 y volviendo a preguntar, entonces si $0 \leq h < 60$ deberia entrar mas rapido que si $300 \leq h < 360$, para esto se podría usar una imagen que tengo uno de los colores (ej: solo rojo) y comparar los tiempos contra una que tenga solo blanco.
2. Implementar parte entera en la variacion2, eliminando la parte decimal de la mantisa de la representación en punto flotante del numero a mano podria aumentar bastante el rendimiento.

4. Conclusiones y trabajo futuro

Pudimos corroborar el uso del assembler para conseguir algoritmos mas eficientes temporalmente, aunque al ser un lenguaje de mas bajo nivel sea mas difiil de escribir. En este sentido se utilizaron las instrucciones SIMD de assembler, permitiendonos mejorar la eficiencia, al efectuar varias operaciones juntas. Sin embargo muchas veces es necesario preparar los registros para efectuar estas instrucciones y esto puede no ser sencilllo, o tener un costo temporal elevado.

Además la ganancia en rendimiento de efectuar operaciones SIMD no es tan importante como reducir la cantidad de accesos a memoria. Se pudo apreciar que al reducir la cantidad de estos ultimos, se logra una mejora en el rendimiento significativa.

Como trabajo pendiente queda la implementación de una variante de la primera implementación de HSL en donde para sumar las componentes h , s y l en lugar de agarrar un solo pixel, ordenar los datos de manera que podamos tener 4 h 's, l 's y s 's a la vez y procesar 4 pixeles al mismo tiempo.