DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Alejandro Mignanelli	609/11	minga_titere@hotmail.com
Facundo Barañao	480/11	facundo_732@hotmail.com
Ian Sabarros	661/11	iansden@hotmail.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

Resumen

En el presente trabajo se describe la problemática de procesar información de manera eficiente cuando los mismos requieren:

1. Transferir grandes volúmenes de datos.
2. Realizar las mismas instrucciones sobre un set de datos importante.

Índice

1. Objetivos generales

El objetivo de este Trabajo Práctico es mostrar las variaciones en la performance que pueden ocurrir al utilizar instrucciones SIMD cuando se manejan grandes volúmenes de datos que requieren un procesamiento similar, en comparación con implementaciones que no lo utilizan.

Para ello se realizarán distintos experimentos sobre tres filtros de foto, Blur, Merge y HSL, tanto en código assembler, que aproveche las instrucciones SSE brindadas para los procesadores de arquitectura Intel, como en código C compilado con gcc, al que se le aplicará el grado de optimización grados de optimización -O3.

2. Calidad de las Mediciones

Antes de medir tiempos y sacar conclusiones de los mismos, es importante tener un esquema básico de como se realizarán las mediciones en el presente Trabajo, y bajo que condiciones.

Los experimentos fueron realizados en una Intel core I7 960@ 3.20GH (8 cpus, 4096 MB RAM), en Windows 7 64 bits con la máquina virtual de la materia. Para que las pruebas sean más concisas y exactas, con la instruccion sudo nice se le ha asignado al scheduler un aviso de la importancia de la tarea, con el objetivo de asignarle más recursos.

Las mediciones se realizaron corriendo los filtros 1000 veces cada uno, cuyos resultados fueron podados, eliminando 200 outliers superiores y 200 outliers inferiores. Con los restantes 600 resultados, se realizó un promedio, y estos promedios son los que se pueden apreciar en los distintos gráficos. Los tiempos están medidos en ciclos de clock. Cabe destacar que las implementaciones de C se han ejecutado con optlvl O3.

3. Experimentación

Se han realizado 2 experimentos generales, y 3 particulares a un filtro, o sea 5 experimentos:

1. El primero esta orientado en medir los tiempos que tardan los distintos filtros para distintos tamaños de imágenes y para imágenes diferentes. Este experimento tiene como objetivo tener una idea del tiempo de ejecución del algoritmo en general, y comparandolo con otras implementaciones del mismo, tener una idea de cual tiene mejor tiempo de ejecución. En el filtro de merge en particular, nos dará una idea de la diferencia entre trabajar con ints o con floats.

2. El segundo esta orientado a observar que tanta influencia tienen tanto los accesos a memorias, como las operaciones aritméticas en el contexto de nuestros algoritmos. Con esto tendremos una idea de que factores limitan a nuestros algoritmos, y en caso de encontrar alguno, como podría resolverse. Para ello se colocarán instrucciones que provoquen accesos a memora y utilicen la ALU, y se observará como cambian los tiempos de ejecución.

3. El tercero es más particular del filtro hsl, que dada su particular implementación (tiene muchas llamadas a funcion dentro del ciclo), se transforma en un buen pivote para analizar cuestiones de overhead. Para ello, se removerán todos los llamados a funciones, cambiandolos por jmps, y se observarán los tiempo obtenidos.

4. El cuarto aplica sobre hsl, y complementa al primer experimeto. Mirando el código de hsl, creamos una imágen que provoca siempre que los if else del código entren por el último if, con la intención de que esto alentice el tiempo de ejecución del filtro, para mostrar que algunas imágenes tienen un mayor impacto en la performance de hsl que otras.

5. El quinto también aplica sobre hsl, pero en particular en una de las implementaciones de este(la primera en assembler).Se cambio el código original de manera tal que pueda aprovechar mejor las ventajas de las instrucciones SSE y se comparó con otras implementaciones y con la implementación original.

4. Blur

4.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Blur:

4.1.1. Implementación 1

Debido a que este filtro usa pixeles de otras posiciones de la imágen original para modificar cada pixel, consideramos pertinente contar con una copia de la imágen original, cuyo objetivo es tener un lugar seguro que no "ensucia" los datos de entrada de cada pixel que debo modificar. Cuando se pidió la memoria para esta copia, se le agregaron bytes de más, porque por como esta implementado, cuando se modificase el último píxel, se leería una posición de memoria que no pertenece a la imágen generando inconvenientes con VALGRIND. Esto se hace antes de un ciclo de ejecución del algoritmo, pero es importante remarcarlo. Una vez finalizado esto, dentro de un ciclo de ejecución, lo primero que hace nuestro algoritmo es cargar en tres xmm pixeles de manera tal de tener en un xmm los vecinos superiores, en otro xmm los vecinos inferiores, y en otro los vecinos del pixel a modificar, siempre tomando un pixel de más, como se muestra a continuación.

(cada campo tiene 32 bits)
$xmm0 = basura|vecino_{sup1}|vecino_{sup2}|vecino_{sup3}$

$xmm1 = basura|vecino_3\ |pixel\ |vecino_4$

$xmm2 = basura|vecino_{inf6}|vecino_{inf7}|vecino_{inf8}$

Estos xmms se cargan desde la imagen copia. Como tomamos un pixel que no tiene nada que ver con el filtro, consideramos apropiado transformalo en 0. Luego desempaquetamos los tres xmm, obteniendo lo siguiente:

(cada campo tiene 64 bits ; cada componente r,g,b,a tiene tamaño word)

$xmm0 = 0|b_{sup1}\ g_{sup1}\ r_{sup1}\ a_{sup1}$

$xmm1 = b_{sup2}\ g_{sup2}\ r_{sup2}\ a_{sup2}|b_{sup3}\ g_{sup3}\ r_{sup3}\ a_{sup3}$

$xmm2 = 0|b_4\ g_4\ r_4\ a_4$

$xmm3 = b_{pixel}\ g_{pixel}\ r_{pixel}\ a_{pixel}|b_5\ g_5\ r_5\ a_5$

$xmm4 = 0|b_{inf6}\ g_{inf6}\ r_{inf6}\ a_{inf6}$

$xmm5 = b_{inf7}\ g_{inf7}\ r_{inf7}\ a_{inf7}|b_{inf8}\ g_{inf8}\ r_{inf8}\ a_{inf8}$

Luego se realiza una suma vertical entre todos los xmm, obteniendo asi un xmm que tiene en una mitad la suma de una parte de los vecinos, y en la otra mitad la suma de la otra parte de los vecinos. Desempaquetamos ese xmm resultado en dos xmm y realizamos nuevamente una suma vertical, obteniendo asi un xmm con la suma de todos los vecinos del pixel a modificar. Convertimos ese numero a float, lo dividimos por nueve (que son la cantidad de vecinos del pixel), revertimos la conversión, obteniendo int, y luego empaquetamos el resultado hasta tener el tamaño original del pixel en un sector de un xmm. Luego, utilizando la función movmaskdqu, insertamos en la imagen original el pixel modificado, y seteamos los registros para comenzar con la siguiente iteración para modificar el próximo píxel.

4.1.2. Implementación 2

Esta implementación es similar a la anterior, con la excepción de que se procesarán 4 píxeles por iteración en lugar de uno. Al igual que en la implementación anterior, se copiará la imágen previamente, pidiendo bytes de más, y luego se entrará al ciclo. Dentro del ciclo, para procesar el primer pixel, se cargaran en tres xmms pixeles de manera tal de tener en un xmm los vecinos superiores, en otro xmm

los vecinos inferiores, y en otro los vecinos del pixel a modificar, siempre contando con un pixel extra en cada xmm, de igual manera que en blur 1. Luego, se procesará de igual manera que en blur 1 pero en vez de empaquetar e insertar en la imágen, se guardará el resultado en un xmm. Se hará lo mismo con el procesamiento del segundo pixel. Para el píxel tres se cargarán xmm inicialmente de la misma forma, pero se hará una copia de los xmm cargados, para ser usados en el pixel 4. Se procesa el pixel 3 de igual manera que el pixel dos y uno. Para procesar el pixel cuatro, se reacomodarán las copias de los xmm cargados en el procesamiento del pixel 3, de manera tal de ”simular.[el] hecho de que se hubieran cargado los pixels como en blur 1. Para eso, notemos que el pixel extra del procesamiento del pixel 3, es un vecino del pixel 4, y que el primer vecino de cada xmm del pixel 3, es el pixel extra del 4. Después de este reordenamiento, se procederá de igual manera que con los pixeles 1, 2 y 3. Luego de esto, se tienen en 4 xmm los 4 píxeles procesados, se empaquetan de manera tal de tener los 4 pixeles en un xmm y se insertan en la imágen original.

4.2. Diferencias de performance en Blur

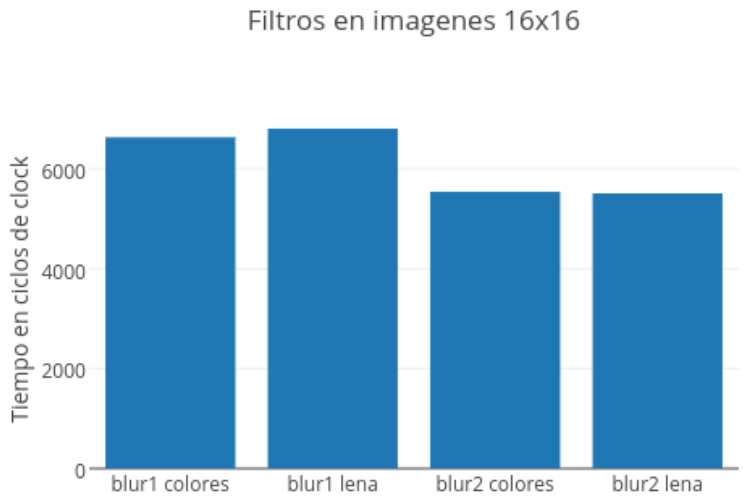

Figura 1: Figura 1

Promedios graficados con imagen de 16x16:

Blur1 colores: 6638

Blur1 lena: 6810

Blur2 colores: 5545

Blur2 lena: 5514

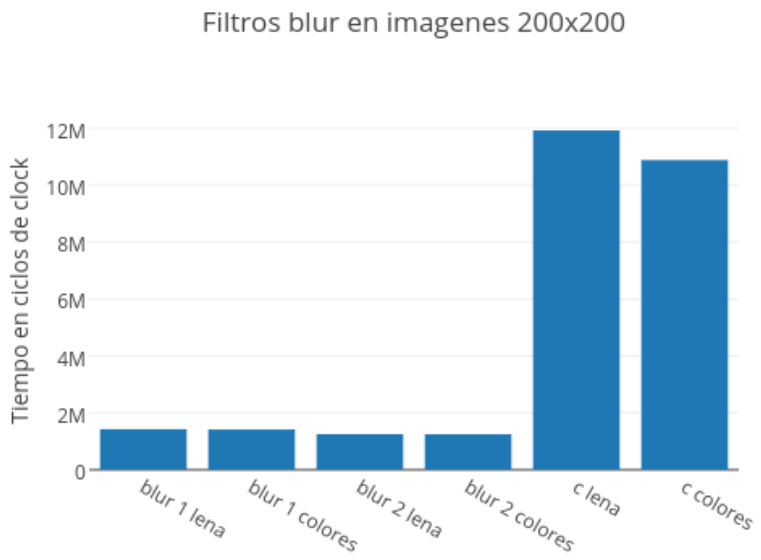

Figura 2: Figura 2

Promedios graficados con imagen de 200x200:

Blur 1 colores promedio: 1413634

Blur 1 lena promedio: 1426631

Blur 2 colores promedio: 1246833

Blur 2 lena promedio: 1250951

Blur c colores promedio: 10885030

Blur c lena promedio: 11926364

De estas primeras dos figuras obtenemos que no se puede apreciar una diferencia notable entre las distintas imagenes con las que contamos para los experimentos, lo que parecería indicar que la complejidad del filtro es en cierta medida independiente a la imagen que se quiere modificar. Si bien no fue incluido en este informe, también se corrio una comparación, con una imagen de tamaño 16x16, entre las 3 implementaciones, dando una diferencia aproximada de 50.000 ciclos de clock entre C y las distintas ASM (cuyos valores no superaban los 10.000 ciclos de clock).

Por otro lado, notamos una diferencia de performance relativamente pequeña, a favor de la implementación 2 del filtro blur, cuando se es comparada con la implementación 1 de dicho filtro. Esto no sucede cuando se compara las implementaciones en ASM con la implementación en C, donde la diferencia resulta abismal, a favor de las primeras. Resulta apropiado notar que, a medida que los tamaños de las imagenes crecen, la diferencia entre implementaciones de ASM se mantiene en cierto nivel, mientras que la diferencia con C crece proporcionalmente, siendo increiblemente ineficaz cuando se cuenta con imagenes grandes.

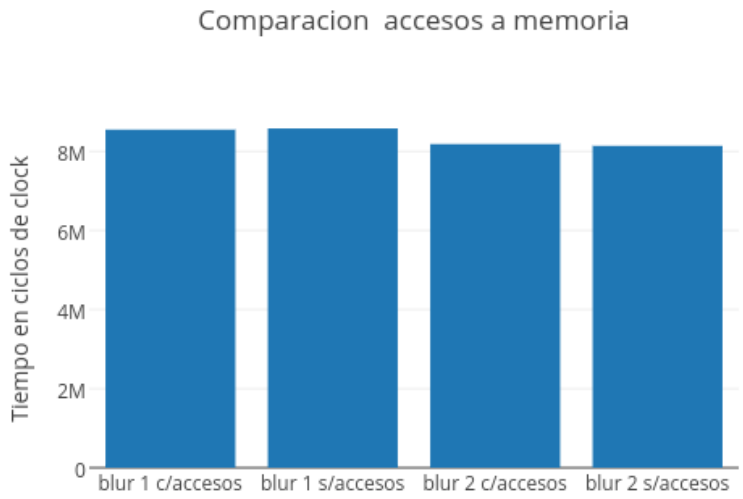

Figura 3: Figura 3

Promedios graficados con imagen 512x512:

Blur 1 en colores s/accesos: 8579777

Blur 1 en colores c/accesos: 8549115

Blur 2 en colores s/accesos: 8139803

Blur 2 en colores c/accesos: 8184807

Para esta comparación, entre el filtro normal y uno al que se le agregaron 6 accesos a memoria por ciclo de ejecución, se decidió ejecutar en el contexto de una imagen de 512x512 dado que esto nos daba como resultado una gran cantidad total de accesos a memoria. El resultado, teniendo en cuenta la cantidad de ciclos de clock totales que requirió cada filtro, no fue significativa. Esto nos lleva a pensar que tal vez, estos accesos no resultan tan relevantes para la performance. Esto se podría dar, ya que al no variar demasiado las direcciones a las accedemos, estas probablemente se encuentran en la cache para acelerar el proceso de acceso.

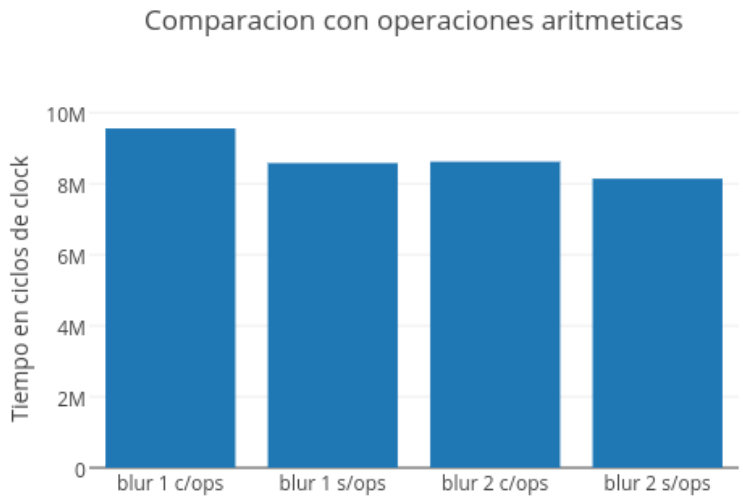

Figura 4: Figura 4

Promedios graficados con imagen 512x512:

Blur 1 en colores s/operaciones: 8579777

Blur 1 en colores c/operaciones: 9555116

Blur 2 en colores s/operaciones: 8139803

Blur 2 en colores c/operaciones: 8619389

Aquí, se introdujeron 6 operaciones aritméticas dentro del ciclo de cada implementación ASM. Así como en el experimento anterior, se corrió sobre una imagen de 512x512 por la misma razón, sólo que aplicada a las operaciones aritméticas. Se puede observar que según como se encuentre implementado el filtro, las operaciones pueden pesar más o menos en el tiempo de ejecución, dado que para la implementación 1 los cambios son más notables que para su contraparte en ASM. Teniendo en cuenta el gráfico anterior, no se pueden apreciar diferencias significativas entre aplicar operaciones aritméticas o accesos a memoria.

4.2.1. Conclusiones

Hasta el momento, las conclusiones parecerían ser que la imagen a procesar no es relevante para el filtro, que la diferencia entre performance de ASM y C es considerablemente grande, no siendo el caso de la diferencia entre ASM1 y ASM2, y también que las operaciones que agreguemos no alterarán mucho estos aspectos. Más adelante trataremos de generalizar estas consideraciones analizando los otros filtros en contextos distintos, así como también contextos similares.

5. Merge

5.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Merge (tener en cuenta que este filtro toma una imágen original, y una imágen que filtra):

5.1.1. Implementación 1

De las dos imágenes, tomamos 4 píxeles y los insertamos registros xmm (uno por cada imágen). A cada uno de estos, los desempaquetamos hasta obtener en 4 xmm, los cuatro pixeles de una imagen, y en otros 4 xmm los píxeles de la segunda imágen. Cada uno de los xmm anteriores, los convertimos a floats.

(cada campo tiene 32 bits)
Imágen original:

$xmm3 = b3_{orig}|g3_{orig}|r3_{orig}|a3_{orig}$ (cada campo es un float)

$xmm2 = b2_{orig}|g2_{orig}|r2_{orig}|a2_{orig}$ (cada campo es un float)

$xmm1 = b1_{orig}|g1_{orig}|r1_{orig}|a1_{orig}$ (cada campo es un float)

$xmm0 = b0_{orig}|g0_{orig}|r0_{orig}|a0_{orig}$ (cada campo es un float)

Imágen Filtro:

$xmm7 = b3_{filt}|g3_{filt}|r3_{filt}|a3_{filt}$ (cada campo es un float)

$xmm6 = b2_{filt}|g2_{filt}|r2_{filt}|a2_{filt}$ (cada campo es un float)

$xmm5 = b1_{filt}|g1_{filt}|r1_{filt}|a1_{filt}$ (cada campo es un float)

$xmm4 = b0_{filt}|g0_{filt}|r0_{filt}|a0_{filt}$ (cada campo es un float)

Luego, multiplicamos los xmm correspondientes a la imágen original por value y a los xmm de la imágen filtro por 1-value. Después de esto, sumamos los xmm que representan un pixel de una imágen con su correspondiente xmm de la otra imágen y los guardamos en xmm, obteniendo 4 xmm que representan los 4 píxeles filtrados. Los convertimos nuevamente a int, y los empaquetamos de manera tal de tener en un xmm los 4 píxeles. Luego los insertamos en la imágen original.

5.1.2. Implementación 2

Obtencion de K La resolucion del pasaje del float value a un int fue abordada intentando encontrar un K tal que el float value por 2^K sea un numero entero. Todo float es representado como un numero que contiene 1 bit de signo, 8 bits de exp, un int en exceso 127 y una mantisa de 23 bits.
$value = 0, MANTISA\ x\ 2^K$
Como value es menor a 1 y mayor a 0, su exponente es negativo. Una vez que tengamos el exponente del numero debemos ver cual es el K que lo convierte el exponente en 23, ya que este numero representaria a la mantisa en su totalidad como int.
Al extraer el exponente del float value vemos que: $exp_{encomplemento} = 127 - exp$ entonces: $150 - (exp_{encomplemento}) = 150 - 127 + exp$ por lo tanto obtenemos $exp + 23$. Luego notamos que $2^k\ x\ 2^{-k} == 2^0$ por lo tanto al hacer la cuenta anterior nos queda el numero K
Una vez que obtenemos el K si multiplicaramos nuestro float por 2^K terminariamos con un float que es solo la mantisa, entonces podriamos anular la parte superior del float, tanto el signo como el exponente y luego tratarlo como int.

Luego podemos construir el numero 2^K que se utilizara para dividir, facilmente, una forma serian shiftar un uno. Este numero podria ser muy grande por lo cual en casos que el exponente supere los 8 bits se generara un value que contenga todos ceros al dividir.

Luego de esta transformacion se procede a realizar la operatoria igual que merge 1.

5.2. Diferencias de performance en Merge

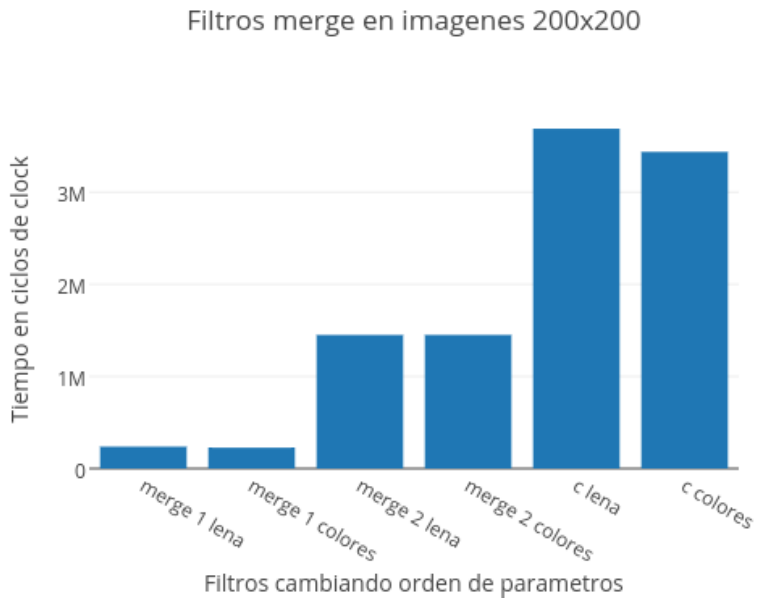

Figura 5: Figura 5

Promedios graficados con imagenes 200x200:

Merge 1 en colores: 225726

Merge 1 en lena: 237499

Merge 2 en colores: 1450444

Merge 2 en lena: 1449965

Merge c en colores: 3438000

Merge c en lena: 3689066

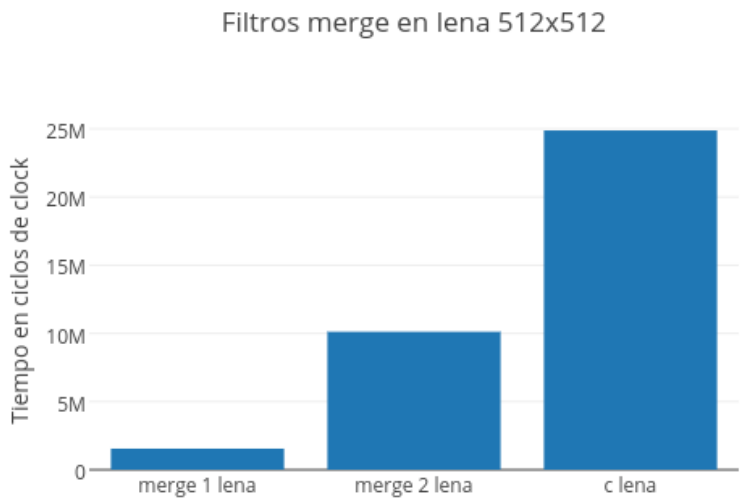

Figura 6: Figura 6

Promedios graficados con imagen 512x512:

Merge 1 lena: 1543888

Merge 2 lena: 10119142

Merce c lena: 24879522

Del análisis de estas dos figuras, podemos notar que, otra vez, la versión C del filtro es claramente inferior, inferioridad que no hace más que acentuarse a medida que las imagenes crecen y por otro lado, que la diferencia de imagenes no es relevante. En este caso, si se puede hacer una diferenciación entre ASM1 y ASM2, dado que ASM1, como muestran las figuras, posee un tiempo de ejecución que es claramente mejor al de ASM2. Creemos que esto se debe, en parte, a la posibilidad de que nuestro pasaje de float a int no haya sido realizado de manera óptima (contamos con un ciclo que realiza shifteos una cantidad k de veces, siendo k una variable que podría llegar hasta 127, siendo shift una instrucción que suponemos es costosa), lo que podría desembocar en que, en los lugares donde tal vez este cambio genera cambios favorables, se compensen con el tiempo que nos lleva convertir de float a int. Notamos también que la calidad de la imagen no mejora, lo que nos lleva a pensar que el pasaje de float a int no realizó ninguna mejora a nuestro algoritmo.

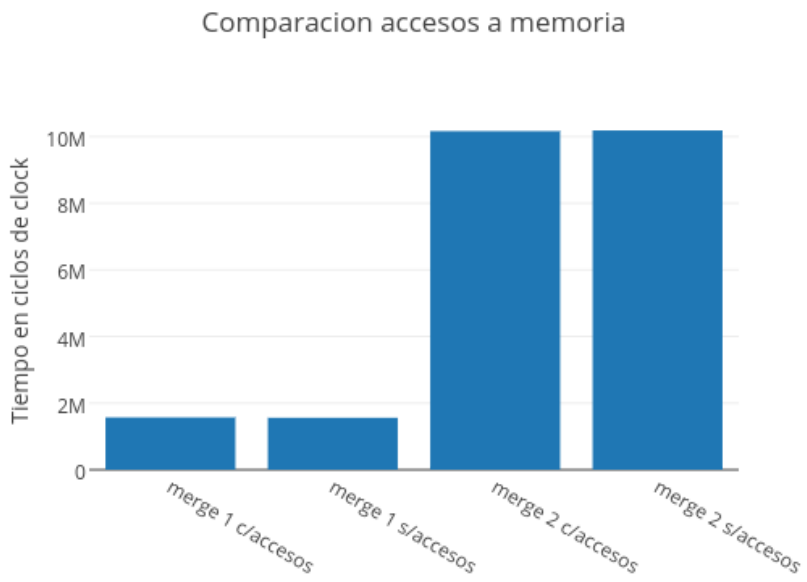

Figura 7: Figura 7

Promedios graficados con imagen 512x512:

Merge 1 s/accesos: 1559196

Merge 1 c/accesos: 1570018

Merge 2 s/accesos: 10184244

Merge 2 c/accesos: 10163918

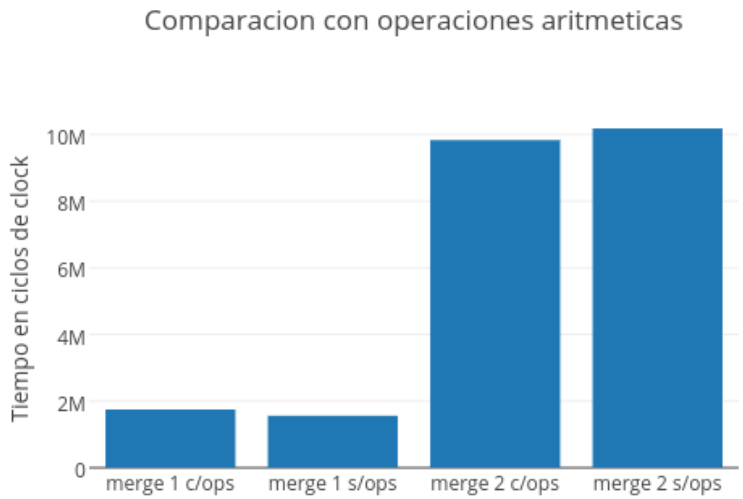

Figura 8: Figura 8

Promedios graficados con imagen 512x512:

Merge 1 s/operaciones: 1559196

Merge 1 c/operaciones: 1747563

Merge 2 s/operaciones: 10184244

Merge 2 c/operaciones: 9833556

Nuevamente, no hay ninguna diferencia que sea necesario recalcar, dado que los tiempos son prácticamente iguales. Reafirmando nuestra teoría de que en ciertos casos, las operaciones que agreguemos no alteran de manera notable la eficacia de nuestros algoritmos. Nos resulta curioso que en ciertos casos, el filtro que tiene presentes las operaciones resulte en un tiempo menor que el tiempo incurrido por el filtro sin operaciones. Sin embargo, dado que en general, el filtro que no presenta operaciones es mejor, nos inclinamos a pensar que esto se puede generalizar.

5.2.1. Conclusiones

Se sigue verificando nuestra teoría sobre la eficiencia de las implementaciones de ASM sobre las de C. A su vez, continuamos con la idea de que agregando operaciones de memoria y aritméticas, modifica desfavorablemente la eficiencia.

6. HSL

6.1. Idea de las implementaciones

A continuación se explicará un ciclo de ejecución de cada una de las implementaciones de Hsl:

6.1.1. Implementación 1

Esta implementación usa dos funciones en c, `rgbtohsl` y `hsltorgb` que hacen lo que sus respectivos nombres indican, transformar un pixel en formato rgb a hsl y de hsl a rgb respectivamente. La primera precisa que se le pase como parámetro un puntero a una dirección de memoria que pueda guardar 16 bytes, donde guardará el resultado, y la segunda recibe un puntero a una dirección de memoria que tenga un hsl para convertir en rgb e introducir en la imágen. Para esto, antes de entrar a un ciclo, se pide por malloc 16 bytes, cuyo puntero se guardara en un registro (a fines prácticos, r8). Otros aspectos a destacar antes de empezar a explicar la implementación de un ciclo, es que los valores HH, SS y LL se encuentran en tres xmm distintos de la siguiente manera:

(cada campo tiene 32 bits)

$xmm3 = 0|0|HH|0$

$xmm1 = 0|SS|0|0$

$xmm2 = LL|0|0|0$

Inicio mi ciclo de ejecución pasándole el píxel a la función `rgbtohsl`, obteniendo así en r8 un puntero al hsl del píxel. Luego paso el hsl a un xmm (digamosle xmm0), para poder trabajar con él, y a este le sumo los xmm que representan HH, SS y LL, obteniendo lo siguiente.

(cada campo representa 32bits)

$xmm0 = l + LL|s + SS|h + HH|a$

Este xmm, lo copio en otros dos xmm, uno al cual le aplico la función `roundps` junto con la opción de redondeo hacia abajo, y luego le aplico `cvtps2dq`, y al otro `cvttps2dq`. De esta manera, en el xmm original voy a guardar el resultado (este mantiene su tipo float), y los otros dos los uso para hacer comparaciones (de habernos dado cuenta antes de la instruccion cmpps, se podría haber evitado esto). Llamemos estas dos copias xmm4 (la de roundps+cvt) y xmm5(la de cvtt).Luego de estas preparaciones vamos al corazón del ciclo.

Cálculo de H: Muevo xmm5 a otros dos xmm, a estos los comparo con una máscara que tiene un 360 (en int) en la posición de h, y un 0 en el resto, siendo cada campo un doubleword. A uno de ellos lo comparo por igual, al otro por mayor, y a ambos les dejo en 0 todos los campos salvo el correspondiente a h. Muevo lo que hay en xmm4 a otros dos xmm y les aplico lo mismo que les aplique a las copias de xmm5, con la diferencia de que la máscara utilizada en este caso tiene un 0 (en int) en el lugar de h en vez de un 360 y el resto de los campos estan cubiertos por 1s. Luego de estas operaciones, se obtendría lo siguiente:

(cada campo tiene 32 bits, si la pregunta se responde por afirmativo, entonces todos los bits del campo

serán 1s, caso contrario, serán 0s)

Copias de xmm5:

$xmm8 = 0|0|h + hh = 360?|0$

$xmm9 = 0|0|h + hh > 360?|0$

Copias de xmm4:

$xmm10 = 0|0|h + hh = 0?|0$

$xmm11 = 0|0|h + hh > 0?|0$

Veamos que como xmm5 está convertido a int por truncamiento, efectivamente sirve para comparar si es mayor o igual a 360, puesto que si el número es menor, nunca redondea para arriba. Si bien es cierto que a un número como 360.4 lo transforma en 360, nos interesa poder decir si el número es mayor o igual, y no nos interesa por cual de los dos casos entre. Por otro lado, veamos que no sirve para comparar si es mayor o igual a 0, puesto que si el número fuera por ejemplo -0.4, el truncamiento lo transforma en 0, y obtengo una respuesta afirmativa, cuando debería ser negativa. Por eso para esto usamos las copias de xmm4, ya que al tomarlas por redondeo hacia abajo darán -1. Nuevamente, nos interesa el mayor o igual, y no en cual caso cayó específicamente, por lo cual el hecho de convertir números como 0.99 a 0 no tendrá relevancia.

LLegado este punto, realizaremos un `pxor` entre las copias de xmm5 para saber si el h+HH es mayor o igual a 360 (supongamos que dejamos esto en xmm8), y otro `pxor` entre las copias de xmm4 para saber si el h+HH es mayor o igual a 0. A las copias de xmm4 las negaremos para saber ahora si h+HH es menor a 0(supongamos que dejamos esto en xmm10). Luego a estos dos registros le pasaremos la máscara utilizada para las copias de xmm5 (la que tiene un 360 en h), obteniendo lo siguiente:

(cada campo tiene 32 bits)

$xmm8 = 0|0|360\ si\ h + hh >= 360,\ 0\ sino\ |0$

$xmm10 = 0|0|360\ si\ h + hh < 360,\ 0\ sino\ |0$

Ahora veamos lo siguiente. Como un número no puede ser mayor o igual a 360 y menor que 0 a la vez, solo uno de ellos tendra un 360, y el otro tendrá 0. Entonces, convertimos estos xmm a floats, tomamos el xmm original, el que llamamos xmm0, le sumamos verticalmente xmm8, y le restamos xmm10, obteniendo asi el h+HH correspondiente según el filtro suma de hsl.

Cálculo de S y L: Son exactamente iguales entre si, teniendo en cuenta que su posición en el xmm es diferente, y muy parecidos al cálculo de h+HH hasta cierto punto, salvo por las siguientes diferencias:

- Se usa una máscara que tiene un 1 en l o s dependiendo el caso, en vez de la mascara que tiene un 360 en h.
- Se compara si los s+SS y l+LL son mayores iguales a 1, o mayores iguales a 0.

El punto en que empiezan a diferir es luego de llegar al siguiente checkpoint (como ya explicamos, los calculos sobre s y l son bastante similares):

(cada campo tiene 32 bits, si la pregunta se responde por afirmativo, entonces todos los bits del campo seran 1s, caso contrario, serán 0s)

Copias de xmm5:

$xmm8 = 0|s + SS = 1?|0|0$

$xmm9 = 0|s + SS > 1?|0|0$

Copias de xmm4:

$xmm10 = 0|s + SS = 0?|0|0$

$xmm11 = 0|s + SS > 0?|0|0$

Luego de esto, para calcular s, realizaremos un `pxor` entre las copias de xmm5 para saber si el s+SS es mayor o igual a 1 (supongamos que dejamos esto en xmm8), y otro `pxor` entre las copias de xmm4 para saber si el s+SS es mayor o igual a 0 (supongamos que esto esta en xmm10).

Luego, con ayuda de los xmm antes mencionados, chequeamos si s+SS es mayor o igual a 1, y de serlo le colocamos un 1.0 en el campo de s. De no ser mayor o igual a 1, pero ser mayor igual a 0, colocamos s+SS en el campo de s, y en otro caso, lo dejamos en 0, obteniendo la parte de s correspondiente con el filtro de hsl aplicado.

Al terminar las tres partes del filtro, y tenerlas en un xmm, copiamos el contenido de este en el puntero antes pedido, y llamamos a hsltorgb, quien coloca el pixel en la imágen.

6.1.2. Implementación 2

La segunda implementación es igual a la primera, salvo porque no contamos con las funciones hsltorgb y rgbtohsl, razón por la cual no se pidió memoria. En cambio, nosotros tuvimos que crear estas funciones en assembler. La explicación de un ciclo de ejecución se puede dividir en cuatro partes: La función principal, la funcion `rgbtohsl_asm`, la función `hsltorgb_asm` y la función `hago_suma`. Se explicarán primero las funciones que están contenidas en la función principal para mayor entendimiento

rgbtohsl_asm: A la función le entran 4 píxeles por un xmm, a los cuales primero los broadcastea de manera tal de dejar en cuatro campos de 32bits todos los r juntos, todos los g juntos, todos los b juntos y todos los a juntos. Luego, desempaquetamos el xmm hasta obtener cuatro xmms de manera tal que uno tenga todos los r, otro todos los g, etc. Luego de esto los xmma quedarían con esta forma:

(cada campo tiene 32 bits)

$xmm1 = r3|r2|r1|r0$

$xmm2 = g3|g2|g1|g0$

$xmm3 = b3|b2|b1|b0$

$xmm4 = a3|a2|a1|a0$

Cálculo los máximos entre r,g y b de cada píxel, y los dejo en un xmm. Hago lo mismo con los mínimos, y los pongo en otro xmm. Calculo la resta entre los máximos y los mínimos y los pongo en otro xmm (a

esta resta para fines prácticos la llamaremos d). Obtendríamos lo siguiente:

(cada campo tiene 32 bits)

$$xmm5 = max(r3, g3, b3)|max(r2, g2, b2)|max(r1, g1, b1)|max(r0, g0, b0)$$

$$xmm6 = min(r3, g3, b3)|min(r2, g2, b2)|min(r1, g1, b1)|min(r0, g0, b0)$$

$$xmm7 = max(r3, g3, b3) - min(r3, g3, b3)|max(r2, g2, b2) - min(r2, g2, b2)| \\ max(r1, g1, b1) - min(r1, g1, b1)|max(r0, g0, b0) - min(r0, g0, b0)$$

Luego de estos preparativos, la función se divide en tres partes:

1. Cálculo de L:

   Básicamente sumo verticalmente los xmm que tienen los máximos y los mínimos, los convierto en float y los divido por 510. Esto lo guardo en un xmm. Este xmm tiene los L de los 4 píxeles.

2. Cálculo de S:

   Tomo el xmm resultado del cálculo de L, y lo copio en otro xmm. A esta copia la multiplico por dos en cada campo, y luego le resto 1.0 a cada campo. Para calcular el fabs comparo por máximo a este xmm, y a otro xmm que sea el resultado de un xmm lleno de ceros restado verticalmente con el primer xmm. La idea detrás de esto es que, sea e un numero entero, entonces $fabs(e) = e$ o $-e$. Hasta ahora, me quedaría algo como:

   (cada campo tiene 32 bits)

   $$xmm7 = fabs(2 * L3 - 1)|fabs(2 * L2 - 1)|fabs(2 * L1 - 1)|fabs(2 * L0 - 1)$$

   Después a un xmm que tenga un 1.0 por campo con campos de 32 bits, le resto el resultado anterior y lo guardo en un xmm. Tomo el xmm que tenía los máximos - mínimos, los convierto a floats, y los divido por el xmm que me guarde anteriormente. A este resultado lo divido por un xmm que tenga en los cuatro campos el número 255.0001. Luego de esto, se obtendría lo siguiente:

   (cada campo tiene 32 bits)

   $$xmm9 -> d3/(1 - fabs(2 * L3 - 1))/255{,}0001 \mid d2/(1 - fabs(2 * L2 - 1))/255{,}0001 \mid \\ d1/(1 - fabs(2 * L1 - 1))/255{,}0001 \mid d0/(1 - fabs(2 * L0 - 1))/255{,}0001$$

   Luego compararé los máximos con los mínimos, y a aquellos que sean iguales, se les asignará un 0. En caso contrario se asignará el resultado anterior. Asi se encontraran los s para cada píxel.

3. Cálculo de h:

   Tomo los xmm que tenían los r, g y b de los distintos píxeles, y los conviero a floats. Luego dejo en otros 3 xmm las restas de los r con los g, los g con los b, y los b con los r. Como ahora tengo que dividirlo por los d, pero este valor podría ser cero, teniendo en cuenta que si no es 0 es positivo, hacemos lo siguiente. Tomamos a los d, les resto uno, tomo su valor absoluto y le sumo 1. Si d era un valor positivo, debería obtener el mismo número. Caso contrario, o sea, si d era 0, debo obtener el número 2, que me va a servir solo para no dividir por cero. Haciendole esta modificación al d, los transformo a float, y ahora tomo a cada uno de los 3 xmms antes mencionados (las restas) y los

divido por d. Hasta acá el estado de los xmm debería ser el siguiente:

(cada campo tiene 32 bits)

$xmm12 = (r3 - g3)/d3 \mid (r2 - g2)/d2 \mid (r1 - g1)/d1 \mid (r0 - g0)/d0$

$xmm13 = (g3 - b3)/d3 \mid (g2 - b2)/d2 \mid (g1 - b1)/d1 \mid (g0 - b0)/d0$

$xmm14 = (b3 - r3)/d3 \mid (b2 - r2)/d2 \mid (b1 - r1)/d1 \mid (b0 - r0)/d0$

Luego, a lo que recién en el estado de los xmm llamé xmm14, debo sumarle 2, al que llamamos xmm12 hay que sumarle 4, y al que llame xmm13, hay que sumarle 6. Luego, a estos 3 hay que multiplicarlos por 60. El nuevo estado debería ser el siguiente:

(cada campo tiene 32 bits)

$xmm12 = 60 * (((r3 - g3)/d3) + 2) \mid 60 * (((r2 - g2)/d2) + 2) \mid 60 * (((r1 - g1)/d1) + 2) \mid 60 * (((r0 - g0)/d0) + 2)$

$xmm13 = 60 * (((g3 - b3)/d3) + 2) \mid 60 * (((g2 - b2)/d2) + 2) \mid 60 * (((g1 - b1)/d1) + 2) \mid 60 * (((g0 - b0)/d0) + 2)$

$xmm14 = 60 * (((b3 - r3)/d3) + 2) \mid 60 * (((b2 - r2)/d2) + 2) \mid 60 * (((b1 - r1)/d1) + 2) \mid 60 * (((b0 - r0)/d0) + 2)$

Ahora moveremos información a un xmm de la siguiente manera: si los máximos y los mínimos de dos campos son iguales, entonces en ese campo habrá un 0. Si el máximo de un campo es igual al r del píxel correspondiente, entonces el campo tendrá a su correspondiente en xmm13. De no ser el máximo del campo igual al r del píxel, pero si a g, entonces el campo tendrá a su correspondiente en xmm14. de no ser el máximo del campo igual al r ni al g del píxel, entonces será igual a b, y tendrá a su correspondiente en xmm12. Luego, por medio de comparaciones se observa si este xmm obtenido es mayor o igual a 360, y en caso de serlo se le resta 360.

Una vez terminado esto tenemos en 3s xmm distintos los h los s y los l de cada pixel, de la siguiente manera:

(cada campo tiene 32 bits)

$xmm1 = H3|H2|H1|H0$

$xmm2 = S3|S2|S1|S0$

$xmm3 = L3|L2|L1|L0$

$xmm4 = A3|A2|A1|A0$

Luego, tan sólo se acomoda la información usando `blendps`, `shuffles` y `shifts`, hasta obtener lo siguiente:

(cada campo tiene 32 bits)

$xmm1 = L3|S3|H3|A3$

$xmm2 = L2|S2|H2|A2$

$xmm3 = L1|S1|H1|A1$

$xmm4 = L0|S0|H0|A0$

y esto devuelve la función.

hago_suma: Es exactamente lo explicado en la implementación uno.

hsltorgb_asm: Esta función toma un hsl y lo transforma en un rgb. Su explicación se divide en varias partes:

1. Cálculo de c: Copiamos el hsl a otro xmm, duplicamos todos sus campos y les restamos 1.0. Luego, usamos el truco anteriormente explicado para aplicar fabs, y ubicaremos la resta entre 1.0 y este número en un xmm. Hasta el momento, este xmm debería ser algo así:

   (cada campo tiene 32 bits)

   $xmm4 = 1 - fabs(2 * l - 1)|irrelevante|irrelevante|irrelevante$

   Luego, copio nuevamente el xmm que tiene el hsl en otro xmm, y con un shifteo posiciono la s en el campo de la parte más alta del registro. Luego de hacer esto, multiplico el registro que shiftie con el que realice operaciones anteriormente, obteniendo asi:

   (cada campo tiene 32 bits)

   $xmm4 = s * (1 - abs(2 * l - 1)) = c|irrelevante|irrelevante|irrelevante$

   Y con un simple broadcasteo, consigo que en todos los campos del xmm se encuentre c.

2. Cálculo de x:

   Tomo el xmm que tiene hsl y lo copio a otro registro(el campo que nos interesará es el de h). A este lo divido por 60 y me preparo para realizar el fmod. Demonos cuenta que $fmod(x, y) = x - redondeohaciaabajo(x/y) * y$. Por esto, guardaremos en otro xmm el resultado anterior, luego en esa copia dividiremos todos los campos por dos, redondearemos hacia abajo, multiplicar los campos por 2, y luego al resultado original lo restaremos con este, obteniendo el fmod. Por el momento los xmm deberían quedar de la siguiente manera:

   (cada campo tiene 32 bits)

   $xmm1 = irrelevante|irrelevante|fmod(h/60{,}0, 2)|irrelevante$

   A este xmm le resto uno en cada campo y le calculo fabs como comentamos anteriormente. Luego tomo un xmm que tenga un 1.0 en cada campo, y le resto el resultado anterior. A esto lo multiplico por la c obtenida anteriormente, teniendo como resultado lo siguiente:

   (cada campo tiene 32 bits)

   $xmm1 = irrelevante|irrelevante|c * (1 - fabs(fmod(h/60{,}0, 2) - 1)) = x|irrelevante$

   Luego, broadcasteo de manera tal que tenga x en todos los campos.

3. Cálculo de m:

   Tomo el c antes obtenido, lo copio en un xmm, a esta copia la divido por 2, y luego copio otro xmm el hsl de entrada, a cuya copia le restare el resultado anterior. Como resultado de estas operaciones, obtendremos el siguiente xmm:

   (cada campo tiene 32 bits)

   $xmm3 = L - (c/2) = m|c/2|c/2|c/2$

   Luego usando shuffle barajo los campos para obtener una m en cada uno de ellos.

4. Cálculo de RGB:

   Usando blendps, un xmm con ceros, el xmm de las c y el de las x, obtengo el siguiente xmm:

   (cada campo tiene 32 bits)

   $xmm4 = 0|x|c|0$

   Usando este xmm, y máscaras obtenidas por medio de comparaciones me fabrico cada una de las siguientes soluciones, y las andeo con su comparaciòn correspondiente, luego sumo todos los casos, ya que solo uno es cierto y de esta manera obtengo lo que busco.

```
comparaciones:          soluciones:
0   <= h <60            b=x| g=0 | r=c | a=0
60  <= h <120           b=c| g=0 | r=x | a=0
120 <= h <180           b=c| g=x | r=0 | a=0
180 <= h <240           b=x| g=c | r=0 | a=0
240 <= h <300           b=0| g=c | r=x | a=0
300 <= h <360           b=0| g=x | r=c | a=0
```

   y asi en un xmm tengo los r,g,b,a en floats de manera correcta

5. Cálculo de escala:

   Al resultado anterior le sumo el xmm que tiene las m y los multiplico por un 255.0. Luego a esto lo convierto en int.

Luego de terminar estas etapas, tengo en un xmm lo siguiente

(cada campo tiene 32 bits)

$xmm7 = b|g|r|a$

Entonces lo desempaqueto dos veces de manera de obtener lo siguiente

(cada campo tiene 32 bits)

$xmm7 = irrelevante|irrelevante|irrelevante|bgra(cadaunode8bits)$

Luego, para finalizar, movemos esto a eax con movd.

La Función principal: Esta función tan solo llama a las otras para que hagan todo. Básicamente lo que hace es:toma 4 píxeles, llama a rgbtohsl_asm que deja en 4 xmm los hsl de los 4 píxeles;luego a cada píxel le aplica hago_suma y le aplica hsltorgb, y desde eax lo pone en la parte baja de un xmm al cual luego shiftea 4 bytes hacia la izquierda, llenando un xmm con 4 pìxeles filtrados, y poniendolos luego en la imágen nuevamente.

6.2. Diferencias de performance en HSL

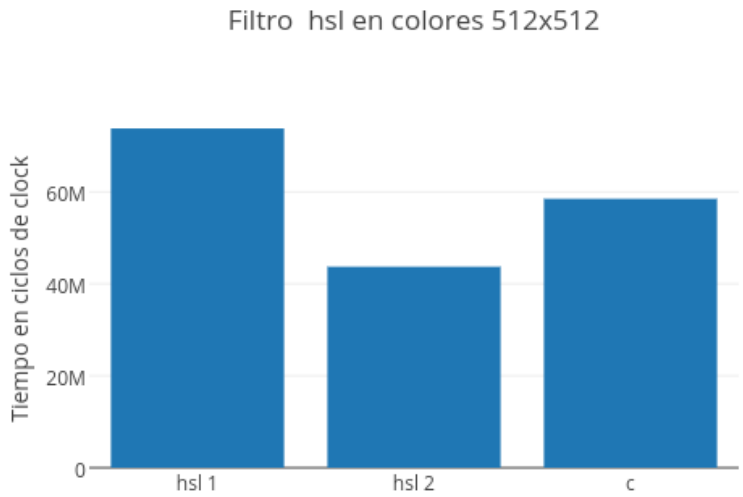

Figura 9: Figura 7

Promedios graficados con imagen 512x512:

Hsl 1 en colores: 73798573

Hsl 2 en colores: 43722805

Hsl C en colores: 58492751

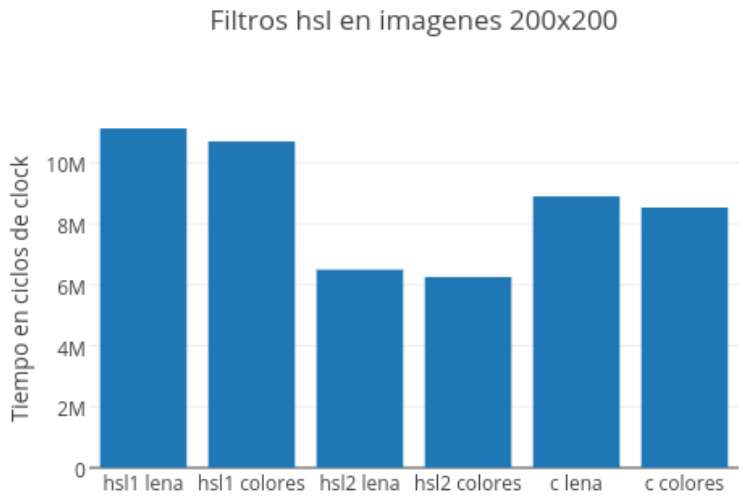

Figura 10: Figura 8

Promedios graficados con imagenes 200x200:

Hsl 1 en colores: 10698487

Hsl 1 en lena: 11123647

Hsl 2 en colores: 6250614

Hsl 2 en lena: 6491505

Hsl C en colores: 8530391

Hsl C en lena: 8897357

Luego de ver estas dos figuras podemos decir que el filtro HSL es un caso particular. Sería el primer caso en el que una función de ASM resulta más ineficiente que la implementación en C. Creemos que esto sucede ya que la implementación 1 del filtro hace uso de funciones en C `rgbTohsl` y `hslTorgb`. En en aspecto del pasaje de rgb a hsl y viceversa, ambos filtros son iguales ya que usan C. La diferencia entre ambas implementaciones esta cuando se busca aplicar la función SUMA. Para esta función, ambas implementaciones son similares en el hecho de que procesan de a 1 pixel a la vez, la diferencia esta en que este procesamiento resulta mas engorroso al realizarse en ASM, dándole la ventaja a la implementación de C (que realiza los pasos de manera mas eficaz). Por otro lado, se nota que la implementación 2 del filtro resulta ser la más eficaz, ya que si bien la función SUMA resulta igual que en la implementación 1, podemos realizar las conversiones de formato de pixel usando ASM (lo que nos permite por un lado convertir de rgb a hsl procesando 4 pixels a la vez), optimizando los tiempos en ese aspecto.

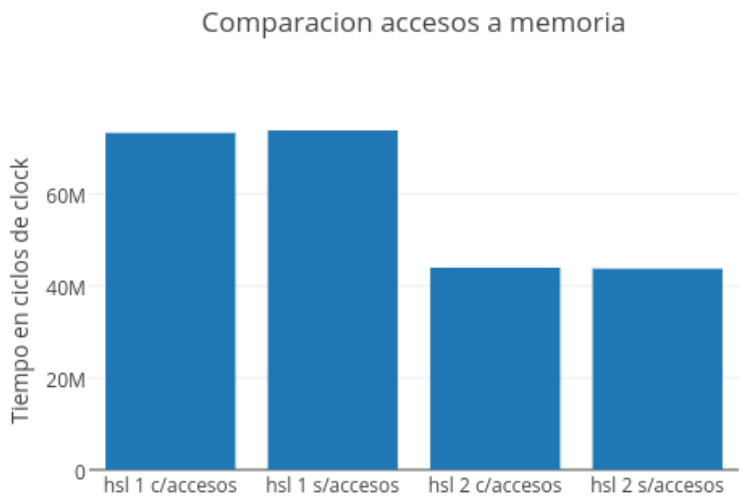

Figura 11: Figura 9

Promedios graficados con imagen 512x512:

Hsl 1 en colores s/accesos: 73798573

Hsl 1 en colores c/accesos: 73251358

Hsl 2 en colores s/accesos: 43722805

Hsl 2 en colores c/accesos: 43931534

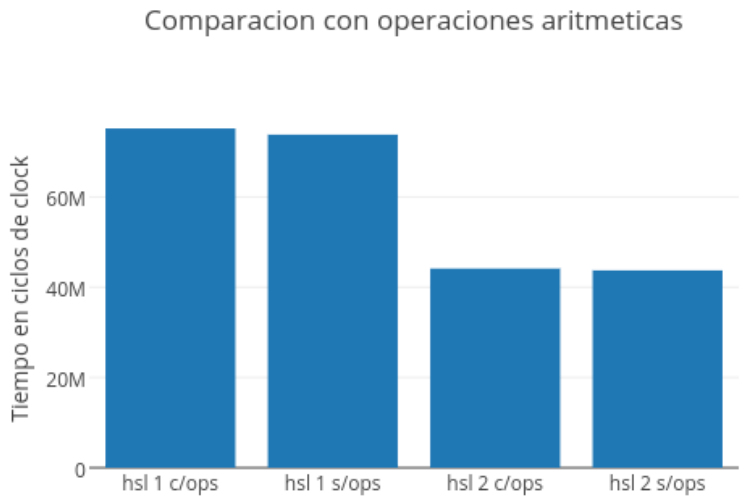

Figura 12: Figura 10

Promedios graficados con imagen 512x512:

Hsl 1 en colores s/operaciones: 73798573

Hsl 1 en colores c/operaciones: 75178963

Hsl 2 en colores s/operaciones: 43722805

Hsl 2 en colores c/operaciones: 44129529

Podemos observar que tanto los accesos a memoria, como las operaciones aritméticas no produjeron un aumento significativo en la velocidad de ejecución de nuestros programas, al igual que en anteriores filtros. La conclusión sobre este tipo de experimentos se verá en el apartado Conclusiones y trabajo futuro.

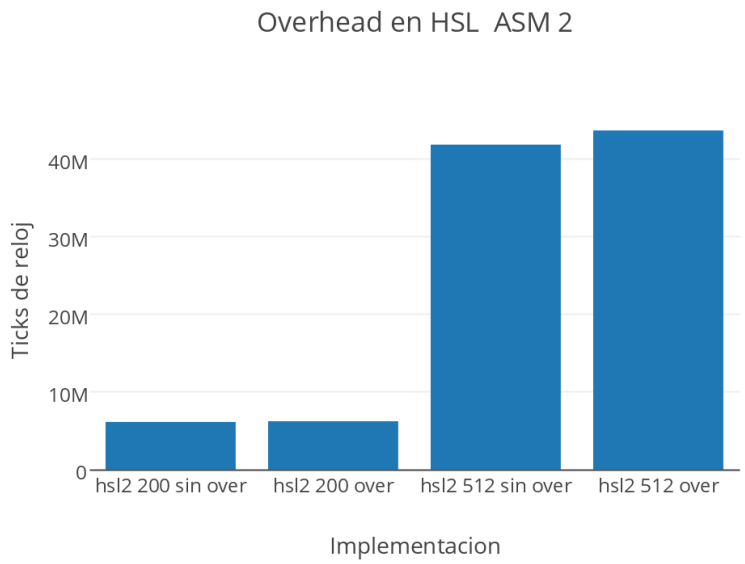

Figura 13: Figura 11

Promedios graficados con imagenes de 200x200 y 512x512:

Hsl2 en colores 200x200 sin overhead: 6168787

Hsl2 en colores 200x200 con overhead: 6250614

Hsl2 en colores 512x512 sin overhead: 41904317

Hsl2 en colores 512x512 con overhead: 43722805

Este experimento fue diseñado especialmente para el filtro HSL2, dado que por como fue concebido (descompuesto en distintos modulos para hacer más fácil su lectura), contaba con una gran cantidad total de llamadas a funciones. Lo que se hizo fue, copiar el código de los distintos módulos, dentro de la función principal, eliminando la necesidad de realizar instrucciones `CALL` y `RET`. Estas instrucciones fueron reemplazadas por saltos `JMP`, que a diferencia de los saltos condicionales, cuya cantidad permaneció intacta, resultan predictivos en su funcionamiento lo que tiene como consecuencia cierta estabilidad en la complejidad temporal de nuestras implementaciones.

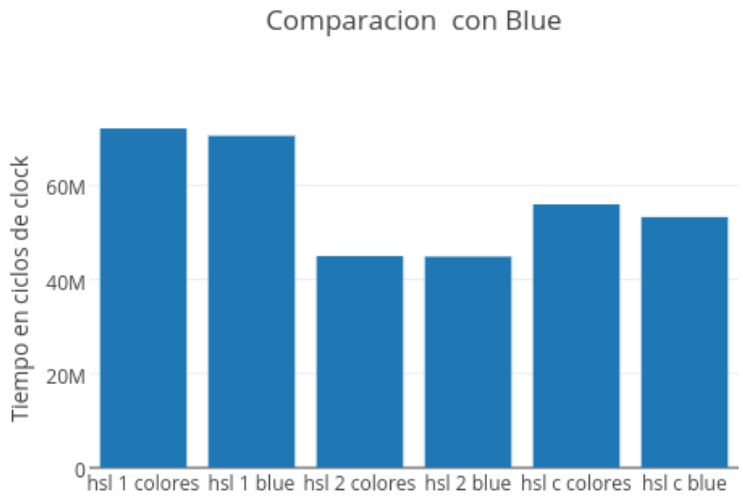

Figura 14: Figura 12

Promedios calculados con imagen 512x512:

Hsl 1 en blue: 70529116

Hsl 1 en colores: 72094082

Hsl 2 en blue: 44855243

Hsl 2 en colores: 44972370

Hsl C en blue: 53266887

Hsl C en colores: 55964800

En este experimento se puede observar una mejoría al ejecutar los filtros en la imagen Blue, siendo esta más marcada en las implementaciones ASM1 y C y muy leve en ASM2. Este hecho resulta curioso dado que el experimento se planteó con la idea de buscar un caso que maximizara la cantidad de operaciones que deben hacer los ifs de las funciones de C, buscando empeorar los tiempos de ejecución. Lo que ocurrió en cambio fue una mejoría que, creemos, se debe al hecho de que, al ser una imagen completamente azul (color que elegimos debido a sus atributos rgb y hsl), siempre entraba por el último sector de los ifs que se encontraban en dichas funciones. Gracias a esta particularidad, suponemos que el predictor de saltos cumplió su tarea, y al observar que siempre se entraba por el mismo lugar del if, logró acelerar bastante el proceso. Es correcto notar que, en la implementación ASM2, no disponemos de saltos condicionales que se puedan ver afectados por particularidades de la imagen fuente, por lo tanto el tiempo de ejecución se mantiene relativamente estable. Por último, y haciendo uso de experimentos anteriores, se pone en evidencia que, cambios bruscos en imagenes (pasar de colores a una imagen de 1 solo color, por ejemplo) puede variar significativamente la eficiencia de algunos filtros.

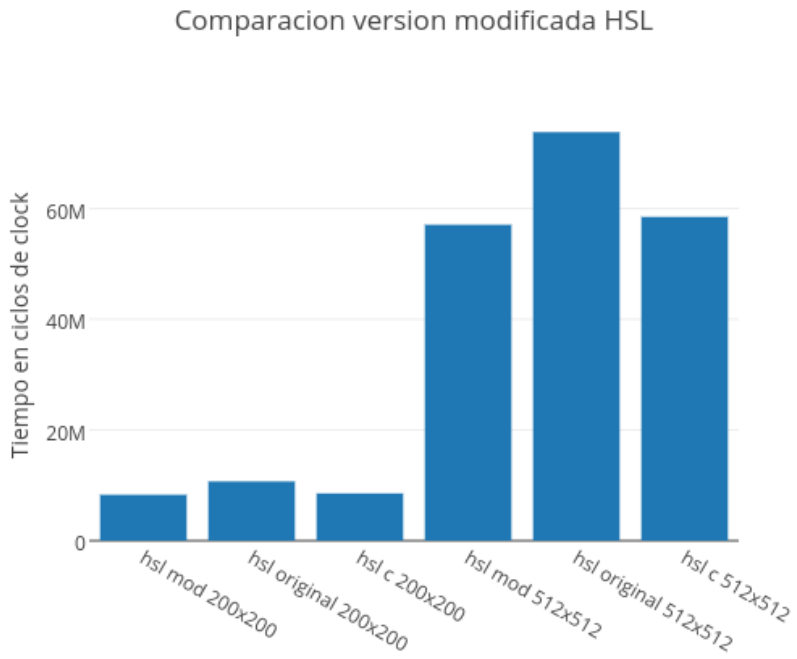

Figura 15: Figura 13

Promedios en imagenes 200x200 y 512x512:

Hsl 1 modificado en colores 200x200: 8294890

Hsl 1 original en colores 200x200: 10698487

Hsl C en colores 200x200: 8530391

Hsl 1 modificado en colores 512x512: 57071615

Hsl 1 original en colores 512x512: 73798573

Hsl C en colores 512x512: 58492751

Se puede observar que la versión modificada de la implementación ASM1 del filtro HSl mejoró considerablemente la performance de su predecesora verificando el resultado esperado de este experimento. Sin embargo, lo que se logró fue una pequeña ventaja en comparación con la implementación de C. Por ende, concluimos que el filtro ASM2 resulta victorioso.

6.2.1. Conclusiones

Si bien se dio el caso de que una implementación en ASM resultara mas ineficaz que una en C, esto se pudo salvar haciendo unas modificaciones al código de la función en ASM, reafirmando nuestra hipótesis. Nos queda más claro que, al momento de manejarnos con saltos condicionales, la herramienta de predicción de saltos juega un papel muy importante.

7. Conclusiones y trabajo futuro

Como primera conclusion de este trabajo práctico podemos decir la gran ventaja que reperesenta la utilizacion de operaciones vectoriales a la hora de realizar operaciones paralelizables sobre un gran volumen de datos. Sin embargo, si bien es interesante aplicarlo a imágenes, tiene una mayor utilidad en el procesamiento multimedia, en donde el tiempo que se ahorra una tarea impacta de manera muy significativa en la calidad del programa en si.

Como trabajo futuro, una idea interesante a desarollar, podria ser construir o plantear un compilador de C que tome la mejor parte de los compiladores actuales, esto es, las diversas optimizaciones que realiza, y las integre con SIMD. De esta manera se podria, en teoria, obtener mejores resultados que los aqui mostrados.