Organización del Computador II
TP2

2 de mayo de 2015

Integrante	LU	Correo electrónico
Martin Baigorria	575/14	martinbaigorria@gmail.com
Julián Bayardo	850/13	julian@bayardo.com.ar
Gonzalo Ciruelos Rodríguez	063/14	gonzalo.ciruelos@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El presente trabajo practico tiene como objetivo aprender a utilizar instrucciones que operan con múltiples datos SIMD (Single instruction, multiple data), soportada por la familia de procesadores x86-64 de Intel. En general, este tipo de instrucciones se utilizan mucho en procesamiento digital de señales y procesamiento de gráficos. Debido a restricciones de la cátedra, solo utilizaremos SSE (Streaming SIMD Extensions) y no el set de instrucciones mas nuevo AVX (Advanced Vector Extensions).

1.1. Historia

El set de instrucciones MMX (MultiMedia eXtension) fue el primero en implementar SIMD en la arquitectura Intel en el año 1997, con los procesadores Pentium II. La idea era, como lo indica el nombre, permitir el procesamiento de operaciones sobre múltiples datos en simultaneo. Esto en general se conoce como paralelismo a nivel de datos, donde un único proceso es capaz de ejecutar estas instrucciones.

MMX define 8 nuevos registros de 64 bits, desde el MM0 al MM7. A nivel arquitectura, estos registros eran simplemente un alias de los registros de la FPU, por lo que cualquier operación en la FPU afecta a los registros MMi.

En el año 1999, con el objetivo de responder a las mejoras introducidas por la competencia, en general por el AltiVec en las PowerPc de Motorola y el sistema POWER de IBM, Intel introduce el set de instrucciones SSE con la serie de procesadores Pentium III. El set de instrucciones SSE tenia 70 nuevas instrucciones. Aquí Intel agrega 8 nuevos registros de 128 bits, XMM0 a XMM7, independientes de la FPU. Este set de instrucciones luego se actualizo con SSE2 (2001), SSE3 (2004), SSSE3 (2006) y SSE4(2006).

En el año 2011 finalmente se introduce al mercado las extensiones al set de instrucciones AVX (Advanced Vector Extensions), primero implementado por la linea de procesadores de Intel Sandy Bridge. AVX expande el tamaño de los registros de 128 a 256 bits, y se renombra a estos registros como YMM0-YMM7. Introduce las operaciones vectorizadas de tres operandos.

Intel introduce AVX2 en el año 2013 con la linea de procesadores Haswell, que expande muchas de las instrucciones de SSE y AVX a 256 bits.

Actualmente Intel esta por lanzar este año la linea de procesadores Xeon Phi Knights Landing, que busca expandir las operaciones y los registros de AVX2 a 512 bits.

Figura 1: Intel Pentium P5

Figura 2: Power PC

1.2. Filtros de Imágenes

Una aplicación típica de este tipo de instrucciones es el procesamiento de imágenes. Esto se debe a que una imagen esta representada por un mapa de píxeles, y en general al procesar imágenes se esta llevando a cabo el mismo tipo de procedimiento muchas veces. A continuación se analizaremos algunos filtros de imágenes, técnicas de implementación de los mismos, y haremos un análisis cuantitativo de las diferencias entre ellos, así como posibles mejoras y reflexiones respecto de las implementaciones puntuales.

A fines de simplificar el análisis, solo utilizaremos imágenes de tamaño HxW en formato bmp, tal que w sea múltiplo de 4. Notaremos a i como el índice de las filas, a j como el índice de las columnas y a k como el índice de los componentes de color de cada píxel. $m_{i,j,k}$ por lo tanto sera el componente k del píxel asociado a la fila i, columna j.

2. Blur

2.1. Introducción

Blur es un filtro que suaviza la imagen. Esto lo hace asignándole a cada píxel el promedio (media aritmética) con sus píxeles vecinos. Es decir:

$$m_{i,j} = (m_{i-1,j-1} + m_{i-1,j} + m_{i-1,j+1} + m_{i,j-1} + m_{i,j} + m_{i,j+1} + m_{i+1,j-1} + m_{i+1,j} + m_{i+1,j+1})/9$$

Las consecuencias de adoptar este método son fundamentalmente dos; en primer lugar, nótese que esto significa que no vamos a procesar los píxeles en los bordes, ya que no tienen la cantidad suficiente de píxeles vecinos.

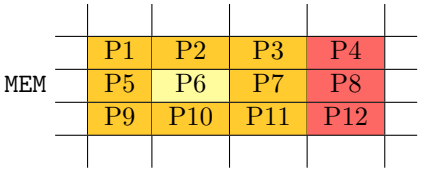
En segundo lugar, debemos tener en cuenta que el cálculo de blur en particular requiere utilizar datos de los elementos aledaños en la matriz, por lo que es imposible procesar la imagen con una complejidad espacial de $O(1)$, de cualquier forma debemos incurrir en un costo de espacio. Evaluamos dos formas de solucionar este problema:

- La primera, crear una nueva matriz en memoria con las mismas dimensiones que la matriz procesada, para poder calcular utilizando los datos originales y guardar en esta matriz. Este método tiene la desventaja de tener una complejidad espacial de $O(w * h)$, fuera de que se agrega un $O(w * h)$ de complejidad temporal para copiar los datos desde la nueva matriz creada hacia la matriz original. Además, este método tiene la desventaja de tener que cuidarse en el copiado para no sobre escribir el borde de la matriz original.
- La segunda, seguir la metodología utilizada en el código de C provisto por la cátedra: mantener dos punteros a memoria de tamaño w que guarden las dos primeras filas originales de la matriz que estamos procesando, y vayan corriéndose a medida que aumentamos la cantidad de filas. Este método toma $O(w)$ de complejidad espacial, con un $O(w)$ de complejidad temporal en el ciclo principal de las filas, para poder copiar los nuevos datos. La ventaja de este método con respecto al anterior reside en el menor uso de memoria, ya que en términos de complejidad temporal el trabajo se amortiza a lo largo de los ciclos.

Terminamos optando por el segundo método, en favor de la mejoría de complejidad espacial a cambio de un golpe en la complejidad conceptual del algoritmo y mayor dificultad en el código fuente. Procedemos a desarrollar los casos particulares de nuestras implementaciones.

2.2. ASM1

Siguiendo la idea del código en C, ASM1 recorre el mapa de píxeles de la misma manera. El código en C procesa cada componente del píxel por separado, mientras que la idea de esta versión es procesar todos los componentes de un píxel con SSE. La idea nuevamente es iterar toda la imagen, primero por filas y luego por columnas, reemplazando los píxeles en las posiciones correspondientes.



En primer lugar, copiamos la fila correspondiente al contador de filas. Esto da inicio al loop de las filas.

Luego, en el loop de las columnas, buscamos en la copia de la primera fila los 4 píxeles (16 bytes) correspondientes al iterador actual de las columnas. Cada píxel ocupa 32 bits, 1 byte por cada componente ARGB. En la memoria, los archivos *.bmp* guardan los componentes de los píxeles en el orden A B G R. Como la arquitectura Intel es little-endian, al mover estos píxeles a un registro, no solo se invertirá el orden de los píxeles sino que también el de sus componentes.

MEM	$A_1B_1G_1R_1$	$A_2B_2G_2R_2$	$A_4B_3G_3R_3$	$A_4B_4G_4R_4$

XMM1	$R_4G_4B_4A_4$	$R_4G_3B_3A_3$	$R_2G_2B_2A_2$	$R_1G_1B_1A_1$

Aquí nosotros hemos levantado 4 píxeles de memoria. Sin embargo notar que para el promedio de los vecinos de un píxel solo necesitamos los 3 primeros. Por lo tanto limpiamos el registro `XMM1` con un shift a la izquierda y luego uno a la derecha:

XMM1	0	$R_4G_3B_3A_3$	$R_2G_2B_2A_2$	$R_1G_1B_1A_1$

Luego, hacemos las operaciones de empaquetado y desempaquetado para expandir el tamaño de cada componente de píxel de 8 a 16 bits. Esto lo hacemos para que mas adelante cuando tengamos que sumar no tengamos overflow al sumar los componentes de dos píxeles.

XMM1	R_2	G_2	B_2	A_2	R_1	G_1	B_1	A_1
XMM2	0	0	0	0	R_3	G_3	B_3	A_3

Ahora sumamos `XMM1` y `XMM2` , poniendo el resultado en `XMM1` :

XMM1	R_2	G_2	B_2	A_2	R_1+R_3	G_1+G_3	B_1+B_3	A_1+A_3

Repetimos este procedimiento tres veces, uno para cada fila. Finalmente nos queda:

XMM1	R_2	G_2	B_2	A_2	R_1+R_3	G_1+G_3	B_1+B_3	A_1+A_3
XMM2	R_6	G_6	B_6	A_6	R_5+R_7	G_5+G_7	B_5+B_7	A_5+A_7
XMM3	R_{10}	G_{10}	B_{10}	A_{10}	R_9+R_{11}	G_9+G_{11}	B_9+B_{11}	A_9+A_{11}

Ahora sumamos los tres registros en `XMM1` . Por cuestiones de claridad, lo representamos de a píxeles únicos:

XMM1	3B	3G	3B	3A	6R	6G	6B	6A

Copiamos `XMM1` en `XMM2` y lo shifteamos 8 bytes a la derecha:

XMM2	0	0	0	0	3B	3G	3B	3A

Sumando `XMM1` y `XMM2` en `XMM1` :

XMM1	3B	3G	3B	3A	9R	9G	9B	9A

Ahora empaqueto la parte baja del registro para que cada componente de cada píxel pase de 1 a 2 bytes y poder ganar precisión al momento de dividir por 9.

XMM1	9R	9G	9B	9A

Tomo el promedio de los componentes de cada píxel dividiendo por el registro | 9.0 | 9.0 | 9.0 | 9.0 |.

XMM1	R_p	G_p	B_p	A_p

Finalmente escribo el registro en la posición de memoria correspondiente. Luego incremento el contador de las columnas o de las filas y vuelvo al ciclo correspondiente.

2.3. ASM2

En este caso, procedimos a procesar la imagen de a 4 píxeles (es decir, de a 16 bytes). Para lograrlo, nuestra idea fue dividir la imagen en matrices de 3x6:

$$\begin{pmatrix} P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & Q_1 & \cdots \\ P_7 & P_8 & P_9 & P_{10} & P_{11} & P_{12} & Q_2 & \cdots \\ P_{13} & P_{14} & P_{15} & P_{16} & P_{17} & P_{18} & Q_3 & \cdots \\ Q_4 & Q_5 & Q_6 & Q_7 & Q_8 & Q_9 & Q_{10} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

En donde buscamos procesar P_8, ..., P_{11} en una sola iteración, corriéndonos de submatriz a medida de avanzamos: cuando terminamos de procesar P_{11} tenemos que correrno a la submatriz que comienza en P_5 para continuar con el siguiente bache de píxeles. Esto nos presentó tres problemas fundamentales:

El primero, cómo iterar la matriz, se resuelve observando que teniendo a `RAX` como un puntero a la esquina superior izquierda de la matriz es muy simple hacer referencia en memoria a las posiciones de la matriz: tenemos que `RAX` $+4*j$ es el $j+1$-ésimo píxel en la fila actual respecto del primer píxel, por lo que si tomamos $j \in \{0, ..., 5\}$ tenemos a todos

los píxeles de la primer fila. Luego, si queremos indexar otra fila, podemos tomar `RAX` $+ 4 * w * i$, que nos determina al primer elemento en la i-ésima fila por debajo de la primera (en el ejemplo, si tomamos $i = 1$, esto apunta a la dirección de memoria de P_7), en este caso tenemos que $i \in \{0, 1, 2\}$.

Es decir, mantener este puntero nos permite iterar la matriz por bloques de 3x6 con la única contra de tener que actualizar el puntero a `RAX` cada vez que queremos cambiar de submatriz: `RAX` $=$ `RAX` $+ 16$ es un movimiento obligatorio cada vez que procesamos los cuatro píxeles que corresponden a un ciclo. Además, tenemos que mantener el número de fila en el que estamos.

El segundo problema es inherente a la forma de iterar que elegimos: dado que las imágenes que procesamos cumplen que $4|w$, siempre que estemos indexando de la forma indicada terminaremos procesando 2 píxeles de más al final de cada fila (es decir, los dos píxeles del borde). La forma que encontramos de resolver esto es a través de una simple comparación: en cada iteración verificaremos si estamos en el borde; en caso de estarlo, simplemente nos correremos dos píxeles hacia atrás y volveremos a procesar los dos píxeles anteriores. El beneficio de este método es que logra ser conceptualmente muy simple, y agrega una cantidad de instrucciones insignificante para el cálculo de complejidad. Además, es muy simple de implementar, en comparación con otras alternativas como comenzar a procesar de a un píxel a partir del borde.

El tercer y último problema es cómo efectuar los cálculos de blur. Intentamos hacerlo de la forma que nos pareció más intuitivo, nos armamos registros con la forma:

$$xmm_2 = \begin{pmatrix} P_2 + P_8 + P_{14} & P_1 + P_7 + P_{13} \end{pmatrix}$$

$$xmm_1 = \begin{pmatrix} P_4 + P_{10} + P_{16} & P_3 + P_9 + P_{15} \end{pmatrix}$$

$$xmm_3 = \begin{pmatrix} P_6 + P_{12} + P_{18} & P_5 + P_{11} + P_{17} \end{pmatrix}$$

Cabe destacar que cada una de las columnas de estos registros se corresponde con una columna de la submatriz, por lo que basta con sumar las columnas de forma adecuada y dividir el resultado por 9 (con las conversiones implícitas para preservar la precisión) a fines de obtener el valor RGBA que queremos para cada píxel. En concreto, nuestro algoritmo sigue el siguiente pseudocódigo:

input : Puntero a la matriz de píxeles representando la imagen, w su ancho, h su altura
output: La matriz de píxeles se actualizo, habiendosele aplicado blur

crear vectores de para las primeras filas ($r10$ y $r11$)
$r10 \leftarrow$ fila 0 de la matriz (copiada)
inicializar el indice rax a la submatriz

for $i \leftarrow 1$ **to** $h - 2$ **do**
  $rax \leftarrow$ nuevo índice de la submatriz, $rax + i * w * 4$
  `Swap` ($r10$, $r11$)
  $r10 \leftarrow$ fila i de la matriz (copiada)
  $rax \leftarrow$ nuevo indice de la submatriz, $rax + i * w * 4$
  for $j \leftarrow 1$ **to** $w - 2$ **do**
    if $j == w - 3$ **then**
      $j \leftarrow j - 2$
    end
    copiar datos de $r10$ y $r11$ a los registros XMM
    convertir las componentes de los registros a enteros de 16 bits
    reordenar los registros para que queden de a columnas
    sumar los registros, ahora hay 3 registros con las 6 columnas
    for $k \leftarrow 0$ **to** *3* **do**
      sumar las columnas necesarias para el k-esimo pixel
      convertir los canales a float
      dividirlos por 9
      convertir todo de vuelta a 1 byte con saturacion
      guardar el pixel en memoria
    end
    $j \leftarrow j + 4$
    $rax \leftarrow rax + 16$
  end
end

liberar la memoria de los vectores

Algorithm 1: Algoritmo de blur procesando de a 4 pixeles

2.4. Comentarios

Al comparar utilizando la imagen de diferencias la imagen generada por el blur de la cátedra y la de ASM1, notamos lo siguiente:

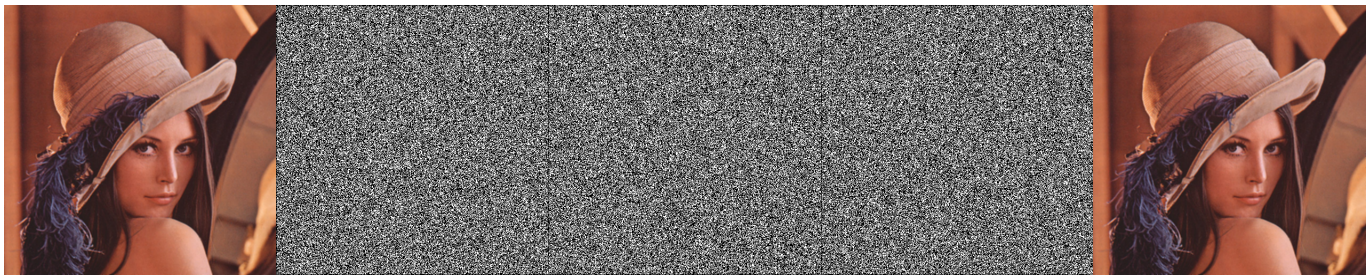


Figura 3: Blur C Figura 4: R channel Figura 5: G channel Figura 6: B channel Figura 7: Blur ASM

Aunque las imágenes parecían exactamente iguales, habían diferencias en algunos píxeles. Estas diferencias se debían al redondeo hacia arriba llevado a cabo por la conversión desde float a entero. Mientras que el código en C redondeaba hacia abajo por defecto, las conversiones en ASM redondeaban hacia arriba por defecto.

Para resolver esto, existe un flag de SSE llamado $MXCSR$. Para mas información, ver la sección 10.2.3.1 del Volumen 1 de la guía de arquitectura Intel. Cada bit de este flag codifica algún comportamiento de las operaciones SSE. El valor por defecto de este flag es $0x1F80$. Para que redondee hacia abajo, utilizando la codificación de los bits del flag, había que poner el bit 12 y 13 en 1. Por lo tanto, simplemente setteamos el flag con la instrucción *ldmxcsr* en $0x7F80$ y el algorítmo comenzó a andar sin problemas. Tuvimos esta misma problemática en todos los otros filtros más tarde.

2.5. Experimentación

3. Merge

3.1. C

El código de C es bastante simple. Lo que hace al inicio es castear ambos punteros a punteros a matrices de matrices, que tienen elementos de w bytes de tamaño, que a su vez tienen elementos de 4 bytes de tamaño.

Luego se realizan 3 loops, el primero sobre las filas, el segundo sobre las columnas, y el tercero sobre los canales (exceptuando el canal de transparencia, notar que ii se inicializa en 1).

Finalmente se realiza la cuenta que se debe hacer para hacer el merge, casteando todo adecuadamente (se castea el valor del canal, que es un entero de 1 byte a float, se lo multiplica por value o 1-value segun corresponda, se lo suma con el otro valor y se lo vuelve a castear a entero de 8 bytes).

3.2. ASM1

En la primera versión del codigo de ASM, iniciamos guardando los parámetros que nos pasan en algunos registros auxiliares para no perderlos mas adelante. Luego calculamos h*w dado que es la cantidad de iteraciones que vamos a realizar.

`RCX` nos servirá de iterador y `RBX` de puntero a la posición actual.

Luego precalculamos dos vectores que vamos a usar en todas las iteraciones

`XMM3`	$value$	$value$	$value$	1,0
`XMM4`	$1 - value$	$1 - value$	$1 - value$	0,0

Luego empieza el loop principal de la rutina. Aquí cargamos 1 pixel de cada imagen (solamente podemos cargar un pixel porque vamos a hacer operaciones con punto flotante de precisión simple, entonces cargar 2 pixeles en una misma iteracion tiene el mismo costo que cargar hacerlo en 2 iteraciones separadas.

Después hacemos unpacking para llevarlos a la forma que queremos y los convertimos a float, y los multiplicamos con los vectores que anteriormente habiamos generado, convenientemente. Entonces los registros tendran la pinta

`XMM1`	$R_1 value$	$G_1 value$	$B_1 value$	$A_1$1,0
`XMM2`	$R_2(1 - value)$	$G_2(1 - value)$	$B_2(1 - value)$	$A_2$0,0

Luego los sumamos en xmm1, y queda la suma

`XMM1`	$R_2(1 - value) + R_1 value$	$G_2(1 - value) + G_1 value$	$B_2(1 - value) + B_1 value$	$A_2$0,0 + $A_1$1,0

Luego, lo pasamos a enteros de 32 bits, y lo packeamos a enteros de 8 bits nuevamente. Notar que no se produce overflow. Tomemos por ejemplo el canal R, y supongamos que el valor del canal R para ambas imágenes es 255. Entonces $255 value + 255(1 - value)$ da 255, que entra en 8 bits (si por errores de redondeo diera más, no importa, ya que el packing lo hacemos saturado, entonces se queda en 255).

Finalmente escribimos en el lugar de la memoria correspondiente el valor obtenido y listo, repetimos este proceso hasta terminar.

3.3. ASM2

Al igual que en la rutina anterior, luego de armar el stackframe guardamos los parametros en algunos registros auxiliares para no perderlos. El resto del proceso es el mismo que antes, hasta que llegamos al precalculo de los vectores de *value*. Como aquí hay que hacer operaciones con enteros, en vez de punto flotante como antes, vamos a calcular las cosas de manera diferente.

En vez de guardarnos *value*, vamos a guardarnos $int16(256 * value)$. ¿Por qué? Porque de esta manera vamos a poder cargar de a dos pixeles y hacer las operaciones mucho mas rápido que antes. Además así guardamos un valor que antes estaba entre 0 y 1 en un valor que esta entre 0 y 256, es decir, un valor que podemos representar con una buena aproximación entera.

De esta menera nos armamos el vector igual que antes, solo que multiplicado por 256. Luego lo pasamos a entero y luego lo packeamos consigo mismo en la parte alta.

```
cvtps2dq xmm3, xmm3
packuswb xmm3, xmm3
```

De esta manera obtenemos en `XMM3` los siguientes valores

`XMM3`

$256v$	$256v$	$256v$	256	$256v$	$256v$	$256v$	256

Donde v es *value*.

Ahora, nos gustaría tener en el otro registros los numeros tal que, sumados con los de `XMM3` , dan 256. Para eso, nos aprovechamos de la representación complemento a 2, dado que lo que queremos en realidad son los inversos aditivos (en 8 bits) de estos números en el registro `XMM4` . Entonces usamos que el inverso aditivo de un número es el negado bit a bit más 1.

Cargamos en `XMM4` 8 enteros de 16 bits con valor 257, ya que es 256+1. Luego, al hacer la diferencia

```
psubw xmm4, xmm3
```

obtenemos en `XMM4` exactamente lo que queremos, es decir que si sumamos int16 a int16 en `XMM3` y `XMM4` da 256.

Luego comenzamos el loop principal. Cargamos lo que nos interesa (2 pixeles) y los unpackeamos a words (16 bits), con ceros en la parte alta. Luego hacemos los productos de las partes bajas

```
pmullw xmm1, xmm3
pmullw xmm2, xmm4
```

De esta manera, los registros quedan de la siguiente forma

`XMM1`	$R_{11}\lceil 256v\rceil$	$B_{11}\lceil 256v\rceil$	$B_{11}\lceil 256v\rceil$	$256A_{11}$	$R_{12}\lceil 256v\rceil$	$G_{12}\lceil 256v\rceil$	$B_{12}\lceil 256v\rceil$	$256A_{12}$
`XMM2`	$R_{21}(256-\lceil 256v\rceil$	$B_{21}(256-\lceil 256v\rceil)$	$B_{21}(256-\lceil 256v\rceil)$	0	$R_{22}(256-\lceil 256v\rceil)$	$G_{22}(256-\lceil 256v\rceil)$	$B_{22}(256-\lceil 256v\rceil)$	0

En realidad, en cada uno de esos registros tenemos la parte baja del producto, pero es importante notar que la parte alta es siempre 0. ¿Por qué? Simplemente porque el canal siempre es menor o igual que 255 y $\lceil 256v\rceil$ es menor o igual que 256, entonces el producto es siempre menor que 65280, que en especial es menor que 65535, el máximo numero representable por enteros sin signo en 16 bits.

4. HSL

4.1. C

El código de C, al igual que el resto, es bastante sencillo. Lo que hace es loopear sobre todos los píxeles, hacer una conversión de RGB a HSL, hacer las sumas correspondientes, y luego volver a convertir a RGB.

Lo malo de la implementacion es que el código sin optimizar de C hace más operaciones de las necesarias, ya que no usa todo el poder de las operaciones en SSE, que nosotros intentamos utilizar al máximo.

4.2. ASM1

En la versión primera versión del código de assembler la operatoriaes bastante distinta a la de C. Al principio calculamos en xmm4 el vector de números que debemos sumarle a cada pixel hsl, con los parámetros que nos pasaron. De esta manera,

XMM4	l	s	h	0

Donde h,s,l son los que nos pasaron como parámetro y el 0 es lo que le tenemos que sumar a la transparencia (nada). Este registro tenemos que guardarlo en la pila, dado que cuando llamamos a rgbTOhsl, nos puede pisar los registros xmm pues la calling convention de C no especifica nada sobre que no se puedan pisar (de hecho en algunos casos lo pisa, fue un bug que tardamos en encontrar).

Tambien tenemos que malloc'ear un float para llamar a las funciones rgbTOhsl y hslTOrgb. Podríamos usar la pila, pero nos resultó mas fácil usar este método.

Luego comenzamos a loopear.

Lo primero que hacemos es llamar a la función rgbTOhsl, de manera que en nuestro puntero a float que malloc'eamos nos que nos va a quedar el valor hsl del pixel en el puntero.

Lo que hacemos después es recuperar los parámetros, que estaban en la pila (dado que posiblemente el llamado a rgbTOhsl haya pisado el registro).

Luego cargamos los registros que vamos a suar para comparaciones y sumas y restas y finalmente guardamos el valor del pixel en hsl en xmm3

XMM3	LL, SS, HH, AA		L	u

ego sumamos este registro con el registro que contiene los parámetros, como indica el filtro, de manera que queda

XMM3	l+LL	s+SS	h+HH	AA

Va a ser útil para mas adelante tener un ejemplo, así que supongamos que XMM3 vale

XMM3	0.5	-0.322	380	255

Ahora comienza la operatoria de saturación, entonces vamos a armar los siguientes registros

XMM5	1-(l+LL)	1-(s+SS)	-360	0

XMM6	-(l+LL)	-(s+SS)	360	0

Siguiendo el ejemplo anterior, los registros quedarían

XMM5	0.5	1.322	-360	0

XMM6	-0.5	0.322	360	0

Entonces procedemos a formar estos registros, usando la menor cantidad de instrucciones posibles, como siempre.

Luego nos armamos 2 registros más, que vamos a usar para las comparaciones

XMM12	1	1	360	256

XMM13	0	0	0	0

Luego comparamos estos registros con nuestro registro XMM3 (nótese que como SSE carece de comparaciones de mayor o igual, hay que dar vuelta los registros y hacer una comparacion de menor o igual).

Por lo tanto, con el ejemplo anterior, XMM12 y XMM13 quedan así:

XMM12	0h	0h	ffffffh	0h

XMM13	0h	ffffffh	0h	0h

Luego les hacemos un and entre los registros XMM12 y XMM5 y entre XMM13 y XMM6 , dword a dword, de manera seleccionar lo que vamos a querer sumar. En el ejemplo esto queda

XMM5	0	0	-360	0

XMM6	0	0.322	0	0

Recordemos el valor de XMM3

XMM3	0.5	-0.322	380	255

Ahora les sumamos estos registros a XMM3 , para terminar de llevar a cabo nuestro plan

XMM3	0.5	0	320	255

Y listo, todo terminó como queríamos.

Ahora solo falta volver a convertir este numero a RGB y escribirlo a la memoria, de lo que se va a ocupar la funcion hslTOrgb.

5. Conclusiónes