Organización del Computador II
TP2

6 de mayo de 2015

Integrante	LU	Correo electrónico
Federico Beuter	827/13	federicobeuter@gmail.com
Juan Rinaudo	864/13	jangamesdev@gmail.com
Mauro Cherubini	835/13	cheru.mf@gmail.com

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción

El objectivo del trabajo practico es utilizar el set de instrucciones SIMD para el procesamiento de imagenes. Para esto se nos pidio implementar 2 versiones en ASM (x64) de 3 filtros diferentes (Blur, Merge, HSL), ademas se nos brindo una version en C para usar como guia.

Es importante destacar que las imagenes que vamos a procesar son multiplo de 4 pixeles y con un tamaño minimo de 16 pixeles, esto nos permite en algunos trabajar de a 4 pixeles sin preocuparnos de salir de la cantidad de memoria asignada a la imagen. Ademas el procesamiento de las imagenes se hace unicamente usando instrucciones SSE y durante el procesamiento de los mismos tratamos de mantenernos adentro de los margenes de error brindados por los test de la catedra.

Una vez implementado los 3 filtos y sus diferentes versiones se iniciara con la etapa de experimentacion, donde buscamos responder las preguntas brindadas por la catedra y de formar un mayor entendimiento de los algoritmos implementados para asi formar conclusiones y propuestas propias.

2. Filtro 1: Blur

2.1. Explicacion

El filtro *blur* consiste en para cada pixel (exceptuando los ubicados en el borde), tomar sus 8 vecinos y hacer un promedio para las componente `R`, `G` y `B` entre los 9 pixeles, este nuevo valor reemplaza los que teniamos en el pixel original. El promedio debe ser calculado respecto a los datos originales, es decir, que si al procesar un pixel y alguno de los vecinos ya fue procesado, debemos utilizar los datos del mismo antes de se procesado. Como referencia la catedra proveyo la implementacion del mismo en *C*.

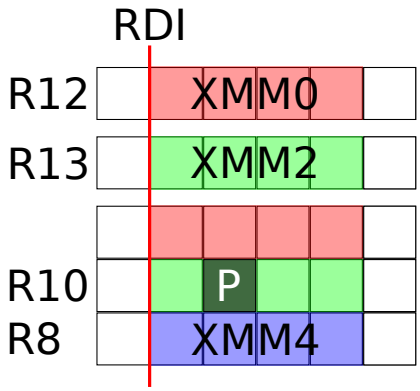
2.2. Implementacion 1

La primera implementacion nos pide que trabajemos de a 1 pixel por iteracion.

Los pixeles estan compuestos por 4 bytes: A R G B, esto nos permite cargar 4 pixeles en un registro `XMM`, como nosotros necesitamos 9 pixeles, ubicados de a 3 en 3 filas diferentes vamos a precisar 3 registros XMM para cargarlos (Para esto usamos los registros XMM0, XMM2 y XMM4). Ademas s utilizaron los siguientes 6 registro de proposito general:

`RDI` que lo usamos para iterar sobre el eje `X`
`R9` que lo usamos para iterar sobre `Y`
`R12` que es un puntero a la copia de la fila superior
`R13` que es un puntero a la copia de la fila actual
`R8` es un puntero a la fila inferior
`R10` es un puntero a la fila actual de la imagen (fila que estoy modificando)

Esto se puede visualizar en el siguiente grafico:



Donde `P` es el pixel a procesar.

Antes de comenzar el ciclo incicializo `R9` en 2 y posiciono el puntero a la imagen en memoria en la segunda fila de pixeles.

Al principio de cada ciclo de `Y` muevo `R8` a `R10`, aumento `R8` en una fila e inicializo el iterador en `X` (ubicado en RDI) en 0.

Muevo a los registros los 3 grupos de pixeles.

```
XMM0 = [R12 + RDI] = xx | p2 | p1 | p0
XMM1 = [R13 + RDI] = xx | p5 | p4 | p3
XMM2 =  [R8 + RDI] = xx | p8 | p7 | p6
```

Despues desempaquetamos los pixeles de *byte* a *word* para poder sumar los 9 pixeles sin saturacion, hacemos una copia de cada uno para poder desenpaquetar la parte inferior en un registro y la superior en otro (XMM1 = XMM0, XMM3 = XMM2, XMM5 = XMM4) y ademas llenamos un registro (XMM15) con ceros para expandir cada una de las componentes sin alterar el numero original. Una vez desempaquetados nos quedan los registros con los siguientes valores.

```
XMM0 = p1 | p0
XMM1 = xx | p2
```

```
XMM2 = p4 | p3
XMM3 = xx | p5
XMM4 = p7 | p6
XMM5 = xx | p8
```

Ahora sumamos los registros y luego hacemos la division.
Sumando XMM0, XMM2 y XMM3 en XMM15
XMM15 = p1 + p4 + p7 ; p0 + p3 + p6
Sumando XMM1, XMM3 y XMM4 en XMM14
XMM14 = x ; p2 + p5 + p8
Ahora hago una copia de XMM15 en XMM13 y la shifteo 8 bytes a la derecha
XMM13 = x ; p1 + p4 + p7
Por ultimo sumo XMM15, XMM14 y XMM13 en XMM15
XMM15 x ; p0 + p1 + p2 + p3 + p4 + p5 + p6 + p7 + p8

Para hacer la division optamos por multiplicar por $(2^{16}/9)+1$ y luego shiftear 16 bits a la derecha cada componente. Para eso tengo el valor por el cual voy a multiplicar en memoria precalculado, al principio del programa decido guardarlo en XMM12. Hago una copia de XMM15 en XMM14 y multiplico por XMM12 guardandome la parte superior de la mutiplicacion en XMM15 y la inferior en XMM14. Luego empaqueto los dos valores juntos como doubleword y los guardo en XMM14 y shifteo a 16 a la derecha. Por ultimo empaqueto los valores obtenidos como words y luego como bytes y los guardo en la posicion de memoria del pixel actual.

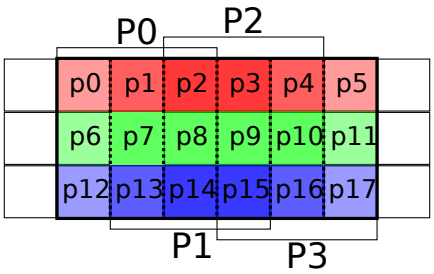
Aumento en 4 el iterador en x y comparo con el tamaño en bytes de una linea de pixeles, si es menor itero nuevamente en x.

Si es mayor o igual voy a hacer 2 copias de las lineas de pixeles siguientes (Que necesito para operar la siguiente fila).

2.3. Implementacion 2

La implementacion 2 nos pide trabajar de a 4 pixeles por iteracion.

Para eso voy a calcular primero los pixeles P0 y P1, y despues el P2 y P3 tratando de minimizar la cantidad de accesos a memoria.



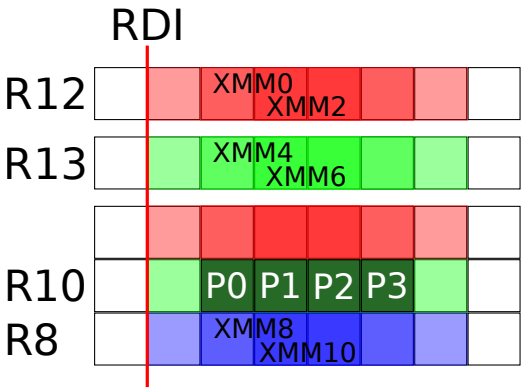
Para hacer los calculos vamos a tomar 6 set de 4 pixeles que nos permiten sumar en un solo registro 2 pixeles al mismo tiempo usando solo sumas, cada set ira adentro de un registro XMM. Ademas se necesitan 6 registros XMM mas para poder desenpaquetar los bytes a words y asi no tener saturacion.

Al igual que la implementacion uno necesitaremos los mismos 6 registros de proposito general.

Antes de comenzar el ciclo iniciamos el iterador en y (R9) y posicionamos el puntero a la imagen en la segunda fila de pixeles.

Al igual que en la primera implementacion nos guardamos una copia del puntero a la fila que vamos a incrementar y movemos R8 a la siguiente fila de pixeles. Ademas inicializamos el iterador de X (RDI) en 0.

Muevo a los registros los 6 pixeles.
XMM0 = p3 ; p2 ; p1 ; p0
XMM2 = p4 ; p3 ; p2 ; p1
XMM4 = p9 ; p8 ; p7 ; p6



XMM6 = p10 ; p9 ; p8 ; p7
XMM8 = p15 ; p14 ; p13 ; p12
XMM10 = p16 ; p15 ; p14 ; p13

Al igual que en la implementacion uno desenpaquetamos los registros haciendo una copia y desenpaquetando la parte superior en el registro original y la inferior en la copia. Para desenpaquetar uso un registro con ceros (XMM15)
XMM0 = p1 ; p0
XMM1 = p3 ; p2
XMM2 = p2 ; p1
XMM3 = p4 ; p3
XMM4 = p7 ; p6
XMM5 = p9 ; p8
XMM6 = p8 ; p7
XMM7 = p10 ; p9
XMM8 = p13 ; p12
XMM9 = p15 ; p14
XMM10 = p14 ; p13
XMM11 = p16 ; p15

Ahora sumo los registros XMM0, XMM1, XMM2, XMM4, XMM5, XMM6, XMM8, XMM9, XMM10 uno por uno con XMM15, al terminar XMM15 = p1 + p2 + p3 + p7 + p8 + p9 + p13 + p14 + p15 ; p0 + p1 + p2 + p6 + p7 + p8 + p12 + p13 + p14, esto como se ve en la imagen es igual a P1 — P0.

Ahora hago la division de cada pixel igual que en la primera implementacion, antes de dividir me guardo una copia de XMM15 en XMM0, divido por 9 y luego shifteo 8 bytes a la derecha y divido nuevamente.

Luego muevo los pixeles procesados P0 y P1 a memoria, posicionandolos con el offset correcto.

Ahora para procesar los pixeles P2 y P3 muevo 3 grupos mas de pixeles de la memoria a los registros XMM0, XMM4 y XMM8.
XMM0 = p5 ; p4 ; p3 ; p2
XMM4 = p11 ; p10 ; p9 ; p8
XMM8 = p17 ; p16 ; p15 ; p14

Ademas hago copias en XMM1, XMM5 y XMM9 y desenpaqueto de byte a word.
XMM0 = p3 ; p2
XMM1 = p5 ; p4
XMM4 = p9 ; p8
XMM5 = p11 ; p10
XMM8 = p15 ; p14
XMM9 = p17 ; p16

Nuevamente sumo los registros XMM0, XMM1, XMM3, XMM4, XMM5, XMM7, XMM8, XMM9, XMM11 uno por uno con XMM15, XMM15 = p3 + p4 + p5 + p9 + p10 + p11 + p15 + p16 + p17 ; p2 + p3 + p4 + p8 + p9 + p10 + p14 + p15 + p16, esto es P3 — P2.

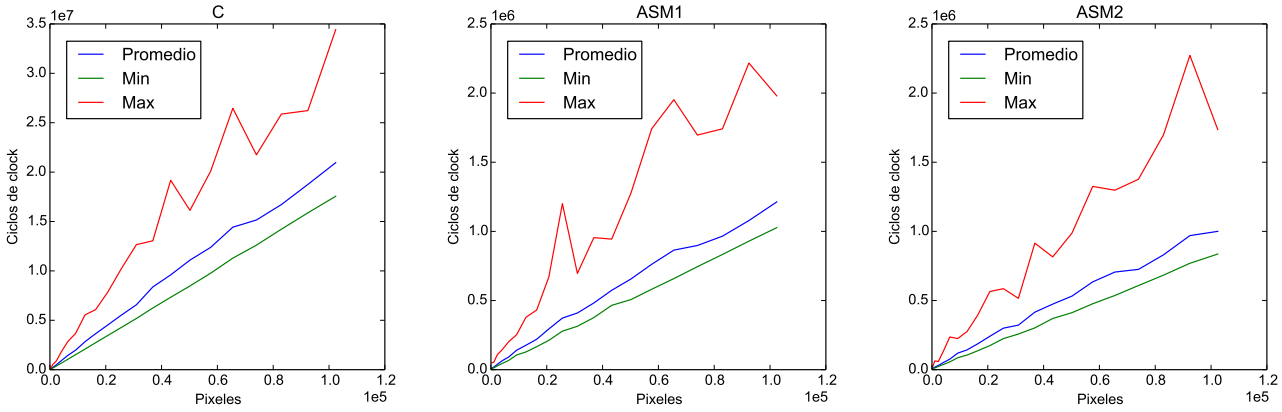
Hago la division al igual que arriba y guardo los pixeles procesados P2 y P3 en memoria.

Comparo el iterador con la cantidad de pixeles en una fila y repito hasta que llegue hasta los ultimos 16 bytes de la fila, en los ultimos 16 bytes me fijo de cargarlos en memoria en solo 3 registros XMM y procesarlos al igual que la primera parte del ciclo para asegurarme de no pasarme.

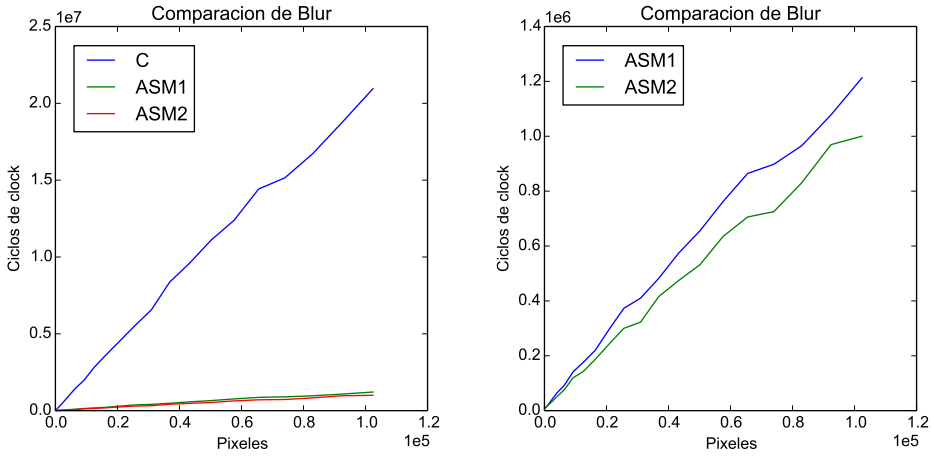
Al final del ciclo hago las copias, incremento el iterador de filas y checkeo si llegue al final.

2.4. Resultados

Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) con la imagen de lena brindada por la catedra (multiplos de 16x16, hasta 320x320), se corren 100 veces cada tamaño y despues se saca un promedio y se grafica el maximo, el minimo y el promedio.

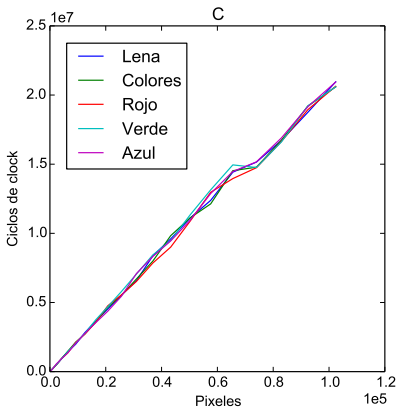


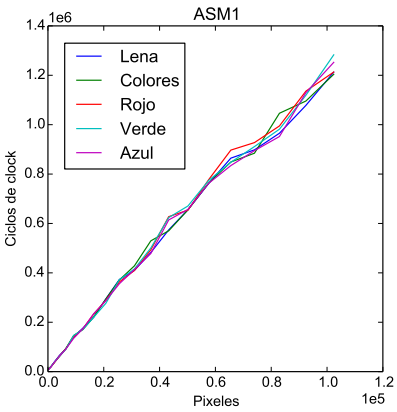
Ahora graficamos los 3 valores promedios de blur para ver cual de ellos es el mas rapido en general. Para ver mejor la diferencia entre ASM1 y ASM2 los graficamos aparte.

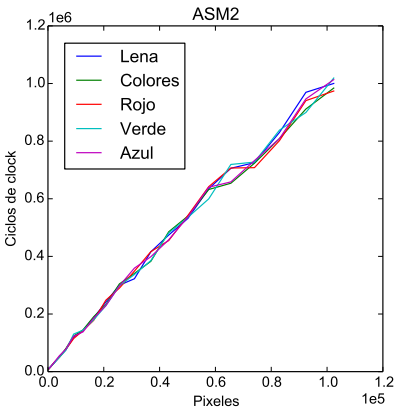


Este grafico nos muestra que el codigo de ASM2 pareceria ser el mas rapido. Antes de confirmar esto queremos ver que no haya ningun tipo de factor en la imagen que influya en el tiempo de ejecucion que no sea el tamaño.

Tanto en el codigo de C como en el ASM1 y ASM2 no poseen ninguna operacion que dependa de los pixeles a procesar, para probar que esto es realmente asi en la practica decidimos correr los test al igual que al principio pero esta vez usando la imagen de lena, colores y 3 imagenes mas completamente rojas, azules y verdes.







2.5. Conclusion

Luego de la experimentacion podemos concluir que ASM2 es el mas rapido en general.

La imagen a procesar no afecta en nada el tiempo de ejecucion y el tamaño lo afecta solo por que se deben procesar mas pixeles.

3. Filtro 2: Merge

3.1. Explicacion

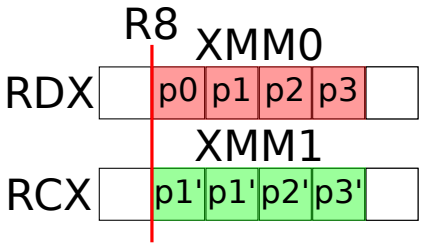
El filtro de merge consiste en tomar 2 imagenes del mismo tamaño y un float value (Con valor entre 0 y 1) y promediar los pixeles de cada imagen para crear una tercer imagen usando como parametro value.

3.2. Implementacion 1

La primera implementacion pide trabajar con floats procesando la mayor cantidad de pixeles posibles por iteracion.

Como sabemos que la cantidad de pixeles de las imagenes es un multiplo de 4 vamos a trabajar de a 4 pixeles que nos permite operar sin tener que procuparnos por salirnos de la memoria alocada a la imagen.

Para procesar los pixeles del merge vamos a usar 2 registros XMM para guardar los 4 pixeles como bytes que obtenemos de la memoria y 2 registros mas para guardar copias de los mismos. Ademas como vamos a utilizar una funcion auxiliar vamos a usar 4 registros XMM mas para guardar los resultados de la funcion. Y por ultimo tenemos los dos valores guardados en float en los registros XMM15 (value) y XMM14 (1 - value).



Ademas vamos a usar los siguientes registros de proposito general:
RDX puntero a la primera imagen
RCD puntero a la segunda imagen
R9 iterador en y R8 iterador en x (En bytes)

Es importante aclarar que antes de entrar en la iteracion creamos los registros XMM14 y XMM15 moviendo el value (XMM0) a XMM15 sufleandolo a todo el registro para que me quede v ; v ; v ; v y luego moviendo a XMM14 4 floats (1.0 ; 1.0 ; 1.0 ; 1.0) que tenemos almacenado en memoria y luego restandole a XMM14 XMM15.

Antes de comenzar a iterar en y inicializo el iterador de y en 0.

Al principio de cada ciclo de x inicializo el iterador en x en 0.

Luego muevo a los registros XMM0 y XMM1 los datos de la primera y segunda imagen. Ademas hago una copia de cada uno en XMM2 y XMM3 (XMM2 = XMM0, XMM3 = XMM1).

Ahora procedo a llamar a addPixels 4 veces y a guardar los valores en los registros XMM4, XMM5, XMM6, XMM7.

addPixels es una funcion auxiliar que hace lo siguiente:

Mueve los valores de las copias a XMM0 y XMM1 (XMM0 = XMM2, XMM1 = XMM3). Luego los desenpaqueta de byte a word y de word a doubleword usando el registro XMM10 el cual posee todos ceros. Los convierte a float, luego multiplica a XMM0 por XMM15 y a XMM1 por XMM14.

Luego de hacer esto queda en XMM0 a * v ; r * v ; g * v ; b * v y en XMM1 a * (1 - v) ; r * (1 - v) ; g * (1 - v) ; b * (1 - v).

Suma ambos y los pasa de float a doubleword, de doubleword a word y de word a byte.

Al final shiftea las copias (XMM2 y XMM3) 4 bytes a la derecha para que quede el siguiente pixel a procesar en los primeros 4 bytes de los mismos.

Una vez que tengo los valores lo unico que me falta hacer es pasarlo a memoria, incremento el iterador de x (R8 += 16) y hacer la comparacion, si no llegue al final de la fila de pixeles itero nuevamente.

Una vez que termine con la fila aumento el puntero de RDX en una fila, y hago lo mismo con el de RCX. Incremento en uno el iterador de y (R9++) y me fijo si llegue al final de la imagen, en caso de no ser asi itero nuevamente en y.

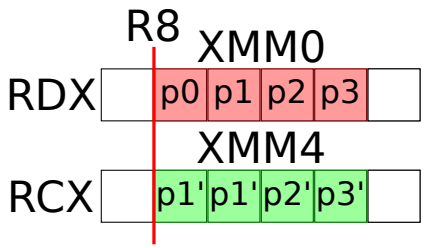
3.3. Implementacion 2

La segunda implementacion pide trabajar con enteros procesando la mayor cantidad de enteros posibles por iteracion.

Al igual que en la primera implementacion vamos a trabajar de a 4 pixeles al mismo tiempo por que nos permite iterar de manera segura.

Al igual que en la primera implementacion vamos a usar los registros XMM14 y XMM15 para guardar los valores por los cuales vamos a multiplicar los pixeles. Para calcularlos vamos a mover value (XMM0) a XMM15, luego lo shufleamos para que los 4 floats que entran en XMM15 sean v, movemos a XMM14 los floats en memoria (8192.0 ; 8192.0 ; 8192.0 ; 8192.0) y multiplicamos XMM15 por XMM14. Para finalizar le restamos a XMM14 XMM15 y convertimos ambos registros de float a doubleword. Al terminar esto nos queda:
XMM15 = v * 8192 ; v * 8192 ; v * 8192 ; v * 8192 XMM14 = 8192 - v * 8192 ; 8192 - v * 8192 ; 8192 - v * 8192 ; 8192 - v * 8192 = 8192 * (1 - v) ; 8192 * (1 - v) ; 8192 * (1 - v) ; 8192 * (1 - v)
Se elijio 8192 que es 2^{13} por que es el numero mas chico posible para usar y que el error que tenga al procesar la imagen este dentro de los parametros de los test.

Ademas vamos a usar los registros XMM0 a XMM7 para tomar los valores de memoria y desenpaquetar. Y XMM10 que es un registro con ceros tambien para desenpaquetar.



Los registros de proposito general que vamos a usar son los mismos que en la primera implementacion.

Antes de comenzar a iterar en incializamos el iterador de y (R9) en 0.

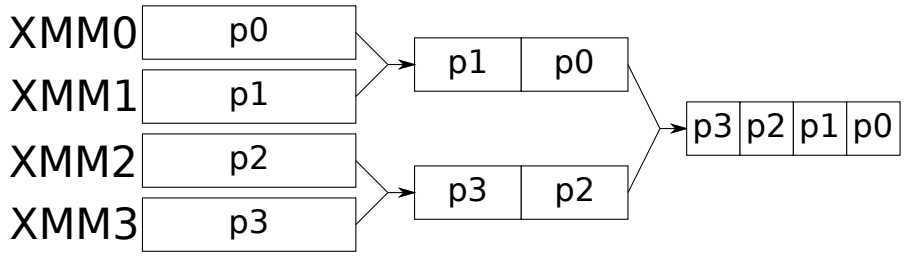
Al principio de cada iteracion de x inicializo el iterador de x (R8) en 0.

Luego muevo a XMM0 y XMM4 los grupos de pixeles de la primera y segunda imagen. Luego desenpaqueto de byte a word y de word a doubleword para que me queden los siguientes registros: XMM0 = p0 XMM1 = p1 XMM2 = p2 XMM3 = p3 XMM4 = p0' XMM5 = p1' XMM6 = p2' XMM7 = p3'

Despues multiplico XMM0, XMM1, XMM2 y XMM3 por XMM15 y lo shifteo 13 bits a la derecha, esto me da (p * 8192v) / 8192 = p * v

Hago lo mismo con XMM4, XMM5, XMM6 y XMM7 pero esta vez multiplicando por XMM14 y me quedan (p * 8192(1 - v)) / 8192 = p * (1 - v)

Luego me quedan en los registros cada pixel multiplicado por v o (1 - v) segun corresponda. Con un poco de paciencia y ingenio podemos enpaquetar las cosas para que queden los 4 pixeles como byte en un solo registro como se muestra en la siguiente imagen. Lo mismo funciona tanto para los dos grupos de pixeles.



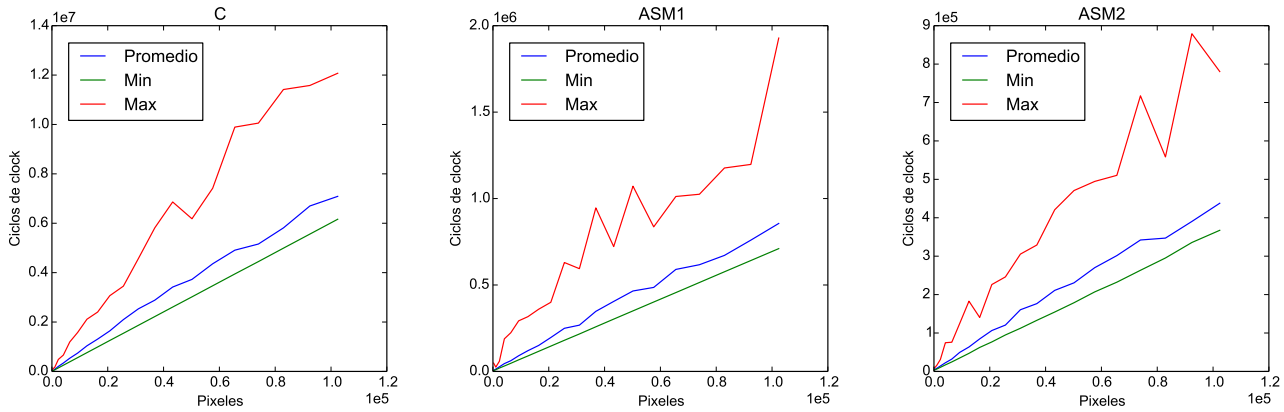
Al final nos queda en XMM0 p3 * v ; p2 * v ; p1 * v ; p0 * v y en XMM4 p3' * (1 - v) ; p2' * (1 - v) ; p1' * (1 - v) ; p0' * (1 - v)

Para terminar solo necesito sumarlos y moverlos a memoria. Esto se puede hacer con un solo mov.

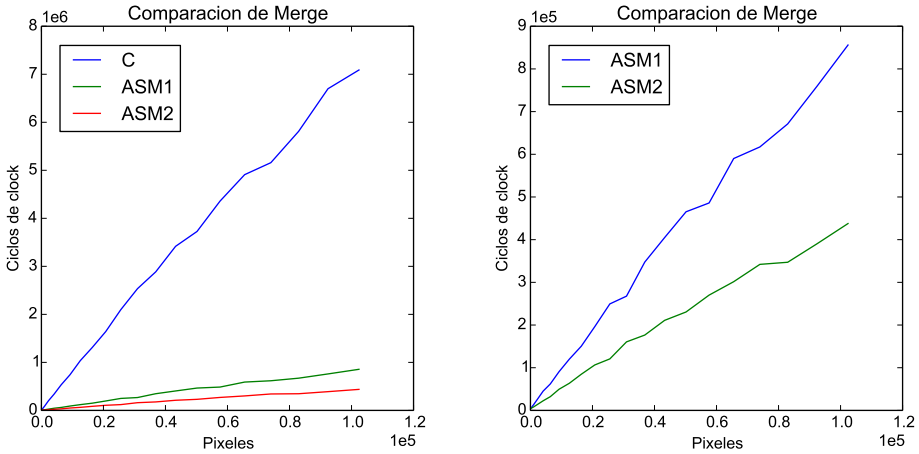
Luego al igual que en la primera implementacion checkeo si llegue al final sino itero en x nuevamenete y hago lo mismo con y.

3.4. Resultados

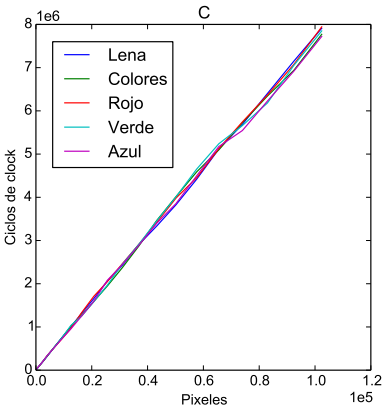
Para la experimentacion vamos a correr las 3 implementaciones (La version de C compilada con optimizaciones de nivel 3) con la imagen de lena brindada por la catedra (multiplos de 16x16, hasta 320x320), se corren 100 veces cada tamaño y despues se saca un promedio y se grafica el maximo, el minimo y el promedio.

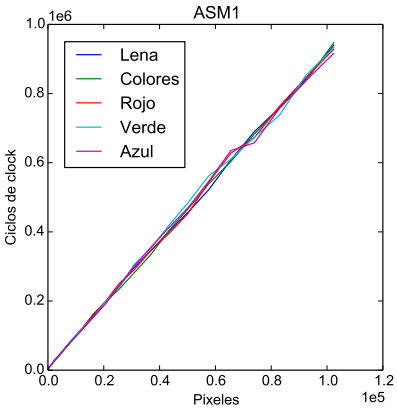


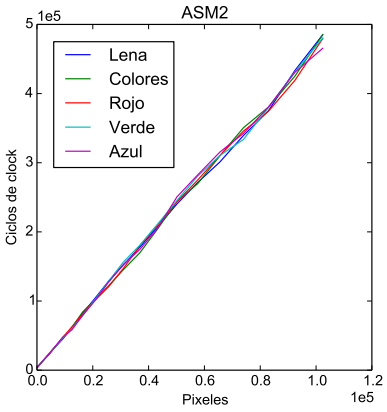
Tambien graficamos los 3 promedios en un grafico para ver cual de ellos es el mas rapido en general. Y para ver mejor la diferencia entre ASM1 y ASM2 los graficamos aparte.



Tambien queremos ver que la imagen no modifique el tiempo de ejecucion, tanto el codigo de C com el de ASM no poseen ningun salto condicional que depende de el valor de los pixeles.







3.5. Conclusion

4. Filtro 3: HSL

4.1. Explicacion

La idea de este filtro consiste en transformar una imagen del espacio RGB a HSL, sumarle un valor recibido por parametro a cada una de las componentes y luego convertir la imagen de HSL a RGB, esto debe hacerse sobre la totalidad de los pixeles. Para las conversiones de espacio y la suma, la catedra proveyo las ecuaciones apropiadas junto con una implementacion en C de las tres etapas.

4.2. Implementacion 1

Para la primer implementacion habia que implementar unicamente la etapa de *suma* en *Assembler*, utilizando las funciones de C provistas por la catedra para la conversion de RGB a HSL y viceversa. La ecuaciones de *suma* son las siguientes:

$$suma_h(h, HH) = \begin{cases} h + HH - 360 & \text{si } h + HH \geq 360 \\ h + HH + 360 & \text{si } h + HH < 0 \\ h + HH \end{cases}$$

$$suma_s(s, SS) = \begin{cases} 1 & \text{si } s + SS \geq 1 \\ 0 & \text{si } s + SS < 0 \\ s + SS \end{cases}$$

$$suma_l(l, LL) = \begin{cases} 1 & \text{si } l + LL \geq 1 \\ 0 & \text{si } l + LL < 0 \\ l + LL \end{cases}$$

Para las operaciones con punto flotante se empleo el tipo $float$. Para comenzar las operaciones de *suma*, partimos de los siguientes registros `XMM` con sus respectivos valores:

```
XMM0 ← LL | SS | HH | -
XMM1 ←  l |  s |  h | -
```

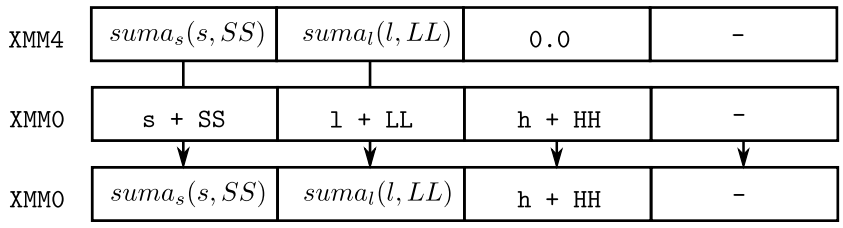
Tambien se cargaron en las siguientes mascaras:

```
XMM2 ← 1.0 | 1.0 |  0.0  | -
XMM3 ← 0.0 | 0.0 |  0.0  | -
XMM6 ← 0.0 | 0.0 | 360.0 | -
XMM7 ← 0.0 | 0.0 |  0.0  | -
XMM8 ← 0.0 | 0.0 | 360.0 | -
XMM9 ← 0.0 | 0.0 | 360.0 | -
```

Con estos valores en los registros, se procedio a realizar la suma de la componentes de `XMM0` y `XMM1` con la instruccion `ADDPS` guardando el resultado en `XMM0`, y luego se almcenaron dos copias del mismo en `XMM4` y `XMM5` mediante la instruccion `MOVDQU`

La copia de `XMM4` se utilizo junto con las mascaras `XMM2` y `XMM3`, para poder satisfacer $suma_s$ y $suma_l$. Primero se tomo el minimo entre `XMM4` y `XMM2`, almacenando el resultado en `XMM4`, luego se procedio a tomar el maximo entre `XMM4` y `XMM3`, nuevamente guardando el resultado en `XMM4`. Esto nos sirve para respetar la funcion de $suma_s$ y $suma_l$, ya que en caso de que nuestros valores de s o l se excedan de `1.0`, el mismo pasaria a ser `1.0`, el mismo razonamiento aplica en el caso de que alguna de esas dos componentes sea menor a `0.0`.

Una vez que tenemos los valores correspondientes en `XMM4`, procedemos a hacer un shuffle con el registro XMM0. El mismo lo hacemos con la instruccion SHUFPS, este responde al comportamiento mostrado en el siguiente grafico:



Con los valores correctos de $s + SS$ y $l + LL$, solo queda analizar $h + HH$. Para hacer esto empleamos las mascaras de `XMM6` y `XMM7`, procedimos a hacer una comparacion entre estas y el registro `XMM5` para obtener las nuevas mascaras que nos permitieron filtrar los condicionales de $suma_h$. La secuencia de operaciones fue la siguiente:

```
XMM6 ← XMM5 ≥ 360.0
XMM7 ← XMM5 ≥ 0
XMM6 ← XMM5 AND XMM8
XMM7 ← NOT(XMM7) AND XMM9
```

Para las comparaciones se utilizo `CMPPS` y para las ultimas dos se utilizo `ANDPS` y ANDNPS respectivamente. Una vez finalizadas estas operaciones, si $h + HH$ se excede o iguala a `360.0`, en el tercer float de `XMM6` tendriamos el valor `360.0` mientras que en el caso que $h + HH$ sea menor a `0.0`, tendriamos el valor `360.0` en el tercer float de `XMM7`, si no se cumplen esas condiciones tendriamos el valor `0.0` en el tercer float del registro que no haya cumplido con el condicional. Con estos valores, lo unica operacion faltante fue a `XMM0` restarle `XMM6`, a este luego sumarle `XMM7` y guardar el resultado en `XMM0`, esto nos permite satisfacer el condicional de $suma_h$, y como originalmente `XMM6` y `XMM7` tenia `0.0` en los primeros dos $float$, esto hace que la ultima suma y resta no afecte los valores de s y l, ya que tanto `XMM6` y `XMM7` siguen teniendo `0.0` en los primeros dos $float$. Teniendo en cuenta esto, tenemos finalmente en `XMM0` un registro que cumple con:

`XMM0` $\leftarrow suma_l(l, LL) \mid suma_s(s, SS) \mid suma_h(h, HH) \mid$ -

Con esto concluimos la etapa *suma* de la primer implementacion, solo resta llamar a la funcion $HSLtoRGB$ provista por la catedra y volcar el resultado de la conversion en memoria.

4.3. Implementacion 2

Para la segunda implementacion se nos pidio implementar las 3 etapas del filtro en *Assembler*. Para la etapa *suma* se utilizo la misma implementacion que la primera.

4.3.1. RGB a HSL

Al igual que el caso de *suma*, la catedra proveyo de las ecuaciones apropiadas. Las mismas estan presentadas a continuacion:

$$h(r,g,b) = \begin{cases} 0 & \text{si } cmax = cmin \\ 60 * (g-b)/d + 6 & \text{si } cmax = r \\ 60 * (b-r)/d + 2 & \text{si } cmax = g \\ 60 * (r-g)/d + 4 & \text{si } cmax = b \end{cases} \quad \text{(mód 360)}$$

$$l(r,g,b) = \frac{cmax + cmin}{510}$$

$$s(r,g,b) = \begin{cases} 0 & \text{si } cmax = cmin \\ (d/(1 - \texttt{fabs}(2 * l(r,g,b) - 1)))/255{,}0001 & \end{cases}$$

Donde $cmax = max(r,g,b)$, $cmin = min(r,g,b)$ y $d = cmax - cmin$.

4.3.1.1. Calculo de h

Primero procedimos a levantar las componentes del pixel de la memoria y copiarlo al registro XMM1, dicha operacion la hicimos con la instruccion MOVD. Despues desempaquetamos los datos de tipo *uint*8 a *int*, esto fue hecho mediante PUNPCKLBW y PUNPCKLWD en ese orden, aplicando las instruccion sobre XMM1 junto con algun registro XMM que contenga 0 en todos sus bits. Una vez con los valores en su tamaño correspondiente, procedimos a calcular cmax y cmin, esto lo logramos utilizando tres copias de XMM1, aplicando shifts a dos de ellas y tomando el minimo y maximo vertical almacenando el resultado en XMM5 y XMM6 respectivamente, los shifts fueron logrados mediante las instrucciones PSRLDQ y PSLLDQ segun el sentido en el cual deseemos shiftear. Hasta este momento tenemos en los registros:

XMM0 ←	B	G	R	A
XMM1 ←	B	G	R	0
XMM5 ←	cmin	-	-	-
XMM6 ←	cmax	-	-	-

El registro XMM0 contiene una copia con el canal *alpha* preservado, esta sera utilizada unicamente al final de la conversion. Con cmin y cmax en sus respectivos registros, procedimos a limpiarlos con una mascara y a hacer un shuffle con XMM6, copiando la componente cmax en las cuatro componentes de XMM2. Luego, hicimos una nueva copia de XMM1 en XMM3 y mediante un shift y un OR con XMM5, logramos juntar las componentes cmin, R, G y B en un solo registro. Entonces tenemos:

XMM5 ←	cmin	0	0	0
XMM6 ←	cmax	0	0	0
XMM2 ←	cmax	cmax	cmax	cmax
XMM3 ←	cmin	B	G	R

Los registros XMM2 y XMM3 fueron empleados para obtener la mascara para poder calcular la funcion $h(r, g, b)$. La misma fue obtenida utilizando la instruccion PCMPEQD entre XMM3 y XMM2, almacenando el resultado de la comparacion de igualdad en XMM3. Una vez obtenida esta mascara, la misma fue reordenada via un shuffle, para seguir el mismo orden que el condicional de la implementacion C, el nuevo orden fue guardado en el registo XMM4. En esta etapa tenemos:

XMM3 ←	cmin == cmax	B == cmax	G == cmax	R == cmax
XMM4 ←	R == cmax	G == cmax	B == cmax	cmin == cmax

Uno de los problemas encontrados durante la implementacion fueron los casos donde existian 2 o mas componentes que eran iguales a cmax, esto hacia que la mascara tuviese mas de una componente con 0xFFFFFFFF, lo cual hacia que los resultados al filtrar fuesen incorrectos. Para solucionar esto propusimos armar una mascara, la cual tuviese 0xFFFFFFFF en las componentes despues la primer componente distinta de cero de XMM4, lo unico necesario seria hacer XOR bit a bit entre la nueva mascara y XMM4. El grafico a continuacion ilustra la idea:

XMM4	0x00000000	0xFFFFFFFF	0x00000000	0xFFFFFFFF
	XOR	XOR	XOR	XOR
mascara	0x00000000	0x00000000	0xFFFFFFFF	0xFFFFFFFF
	=	=	=	=
XMM4	0x00000000	0xFFFFFFFF	0x00000000	0x00000000

Una vez que tenemos esta mascara, preparamos los registros XMM7, XMM8, XMM9 y XMM11 con los siguientes valores y tambien disponiamos de una seria de constantes en memoria. Los valores de los registros y constantes son los siguientes:

XMM7 ←	g	b	r	0
XMM8 ←	b	r	g	0
XMM9 ←	cmax	cmax	cmax	cmax
XMM11 ←	cmin	cmin	cmin	cmin
cte_suma ←	6.0	2.0	4.0	0.0
cte_60 ←	60.0	60.0	60.0	60.0

Se procedio a realizar las operaciones verticales apropiadas para poder satisfacer la funcion $h(r, g, b)$. La unica salvedad es que luego de efectuar las operaciones de resta entre XMM7 y XMM8, y la de XMM11 y XMM9, el resultado de ambas

fue convertido a *float* mediante la instruccion `CVTDQ2PS` y almcenado en los registros `XMM8` y `XMM10` respectivamente. Posteriormente se procedio con el resto de las operaciones con las instrucciones de punto flotante apropiadas, y se guardo el resultado en `XMM8`, a este luego se le aplico la mascara guardada en `XMM4`.

Otro problema que encontramos fue el caso `cmin == cmax`, aqui tenemos que `d` es igual a cero, con lo cual al proceder con las divisiones nos topariamos con un NaN, para solucionar esto decidimos armar una nueva mascara que contenga `cmin == cmax` en cada una de sus componentes. Para hacerlo alcanzo con utilizar el registro `XMM9` y `XMM2` (el mismo no fue modificado y mantiene los mismos valores que arriba) para hacer la comparacion de igualdad, el resultado de la misma fue guardado en `XMM9`. Con esto se procedio a hacer la operacion `NOT(XMM9) AND XMM8`, y para poder calcular el mód 360 de $h(r, g, b)$ se aplico la misma logica que en caso de *suma* ya que los calculos para h no van a excederse de 720. Finalmente tenemos en `XMM10`:

`XMM10` ← `-` | `-` | `-` | `h`

4.3.2. HSL a RGB

Tanto como en el caso de *suma* y $RGBtoHSL$, la catedra proveyo las ecuaciones necesarias para la conversion, las mismas son:

$$RGBAux(h,s,l) = \begin{cases} (c,x,0) & \text{si } 0 \leq h < 60 \\ (x,c,0) & \text{si } 60 \leq h < 120 \\ (0,c,x) & \text{si } 120 \leq h < 180 \\ (0,x,c) & \text{si } 180 \leq h < 240 \\ (x,0,c) & \text{si } 240 \leq h < 300 \\ (c,0,x) & \text{si } 300 \leq h < 360 \end{cases}$$

$$RGB(h,s,l) = (RGBAux(h,s,l)_r * 255, RGBAux(h,s,l)_g * 255, RGBAux(h,s,l)_b * 255)$$

Donde $c = (1 - fabs(2 * l - 1)) * s$, $x = c * (1 - fabs(fmod(h/60, 2) - 1))$ y $m = 1 - c/2$. Se tuvo que hacer un cambio, ya que en el enunciado las componentes `B` y `G` se encontraban invertidas, imposibilitando la conversion correcta.

Para poder hacer el calculo de `R`, `G`, `B` tuvimos que hacer el de `c`, `x` y `m`.

4.3.2.1. Calculo de `c`

Para el calculo de `c`, tenemos los siguientes registros, mascaras y constantes:

`XMM1`	←	`l`	`l`	`l`	`l`
`XMMn`	←	`s`	`s`	`s`	`s`
`cte_1`	←	`1.0`	`1.0`	`1.0`	`1.0`
`cte_2`	←	`2.0`	`2.0`	`2.0`	`2.0`
`masc_abs`	←	`0x7FFFFFFF`	`0x7FFFFFFF`	`0x7FFFFFFF`	`0x7FFFFFFF`

Con estos podemos hacer las operaciones correspondientes para satisfacer el calculo de `c`. El unico caso que requiere atencion particular es el de `fabs`, debido a que los numeros de punto flotante estan codificados con la norma IEEE 754, lo unico que hace falta para poder obtener el valor absoluto es colocar el bit mas significativo de cada *float* en cero, para hacer esto alcanza con aplicar la mascara `masc_abs` mediante la instruccion `ANDPS` al registro apropiado. Tras finalizar las operaciones tenemos en `XMM1`:

`XMM1` ← `c` | `c` | `c` | `c`

Posteriormente procedimos a limpiar el registro, dejando unicamente el *float* de las posicion menos significativa. Entonces obtenemos:

`XMM1` ← `0` | `0` | `0` | `c`

Esto lo hacemos para despues poder armar todas las posiblidades de $RGBAux(h, s, l)$.

4.3.2.2. Calculo de x

Al igual que con c, vamos a empezar por los registros:

```
XMM0     ←     l      |     s      |     h      |     a
XMM1     ←     0      |     0      |     0      |     c
XMM3     ←     h      |     h      |     h      |     h
cte_1    ←   1.0      |   1.0      |   1.0      |   1.0
cte_2    ←   2.0      |   2.0      |   2.0      |   2.0
cte_60   ←  60.0      |  60.0      |  60.0      |  60.0
masc_abs ← 0x7FFFFFFF | 0x7FFFFFFF | 0x7FFFFFFF | 0x7FFFFFFF
```

Con estos valores podemos hacer facilmente los calculos de x, para fabs aplicamos la misma logica que antes, el unico caso que requiere atencion particular es el de fmod. La funcion fmod realiza las siguientes operaciones:

$$fmod(a, b) = a - \left\lfloor \frac{a}{b} \right\rfloor * b$$

Para poder implementar esta funcion con instrucciones SSE, tenemos que encontrar una forma de tomar la parte entera de la division, para hacer esto decidimos utilizar la instruccion ROUNDPS, la misma si es uilizada con el inmediato 0x03 nos permite guardar una version redondeada con cero en la parte decimal del operando fuente. Una vez que disponimos de este resultado, solo queda aplicar la multiplicacion y resta correspondiente. Una vez finalizadas las operaciones requeridas, tenemos en XMM2:

```
XMM2 ← 0 | 0 | 0 | x
```

Los primeros tres $float$ quedan en cero puesto a que al hacer la multiplicacion con XMM1, el mismo tenia c unicamente en la ultima componente.

4.3.2.3. Calculo de m

Para el calculo de m no hubo mayor particularidad, el mismo se llevo a cabo con los siguientes registros y mascaras:

```
XMM3  ← l | l | l | l
XMM4  ← c | c | c | c
cte_2 ← 2 | 2 | 2 | 2
```

El resultado final fue guardado en el registro XMM3, el mismo quedo asi:

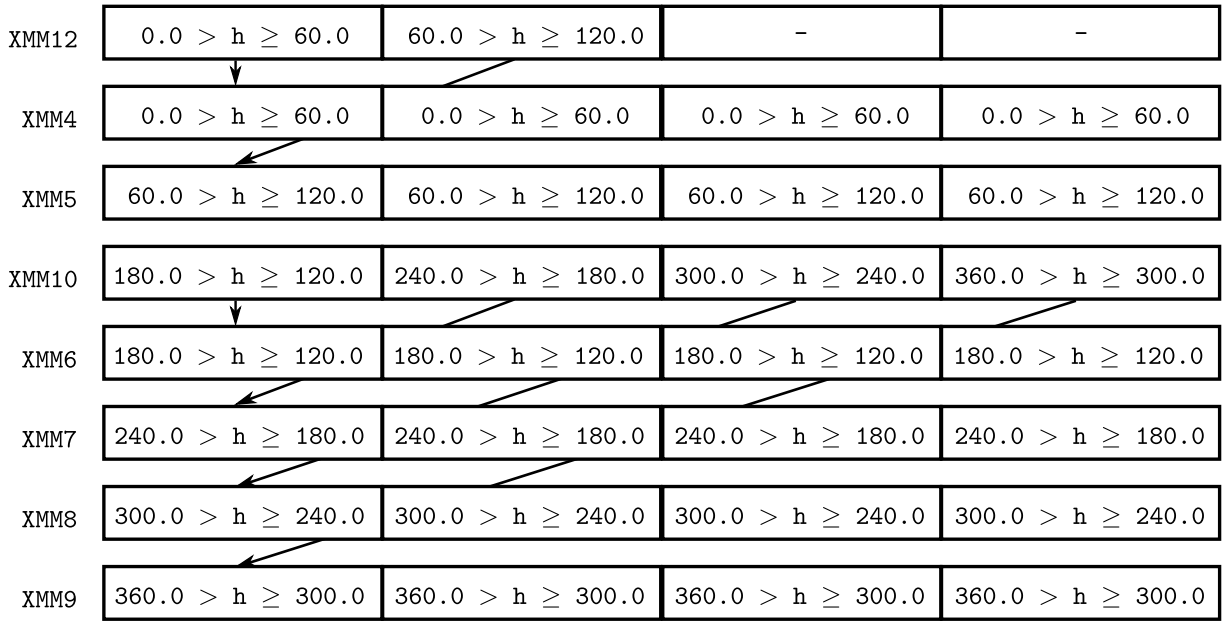
```
XMM3 ← m | m | m | m
```

4.3.2.4. Calculo de RGB

Para el calculo de $RGBAux$, se tuvieron que armar seis mascaras, las mismas filtraban los 6 casos de la funcion. Para armarlas se conto con los siguientes registros y constantes:

```
XMM9     ←   h   |   h   |   h   |   h
cte_cmp1 ← 180.0 | 240.0 | 300.0 | 360.0
cte_cmp2 ← 120.0 | 180.0 | 240.0 | 300.0
cte_cmp3 ←   0.0 |   0.0 |  60.0 | 120.0
cte_cmp4 ←   0.0 |   0.0 |   0.0 |  60.0
```

La idea consistio en comparar si XMM9 $<$ cte_cmp1 y XMM9 $\geq$ cte_cmp2, luego se procedio a unificar estas dos comparaciones con un AND, es decir, nos quedamos unicamente con los casos donde una componente de XMM9 sea menor a su respectiva componente de cte_cmp1 y mayor o igual a la de cte_cmp2, luego se almaceno el resultado en XMM10. Este mismo proceso fue repetido pero con cte_cmp3 y cte_cmp4, y el resultado fue almacenado en XMM12. Estos resultados luego fueron expandidos a 6 registros XMM mediante un shuffle, el proceso responde al siguiente grafico:



Con estas mascaras armadas, se procedio a armar las 6 posbiles combinaciones de `R`, `G` y `B`. Las mismas son:

```
XMM10 ← 0 | x | c | 0
XMM11 ← 0 | c | x | 0
XMM12 ← x | c | 0 | 0
XMM13 ← c | x | 0 | 0
XMM14 ← c | 0 | x | 0
XMM15 ← x | 0 | c | 0
```

Lo unico restante fue aplicar cada una de las mascaras formadas anteriormente a su correspondiente registro de la lista anterior mediante un `AND`, y luego sumar todo en un solo registro. Con eso ya teniamos el valor final de $RGBAux$ en el registro `XMM4`, ademas de este teniamos tambien el siguientes registro y constante:

```
cte_255 ← 255.0 | 255.0 | 255.0 | 255.0
XMM3    ←   m   |   m   |   m   |   m
```

Con estos registros, lo unico faltante fue realizar las operaciones aritmeticas indicadas en la funcion RGB, mencionada al comienzo del esta seccion, y unificar el resultado con el canal *alpha* del registro `XMM0`. Para hacer esto alcanzo con quedarnos unicamente con la componente menos significtiva y juntarla mediante un `OR` con el resultado final de RGB. Esto nos da el resultado final de la conversion, el siguiente paso fue convertir los numeros de tipo $float$ de 32 bits a $uint$ de 8 bits, para hacer esto basto con convertir de $float$ a int de 32 bits via la instruccion `CVTPS2DQ` y empaquetar el resultado de 32 bits a 8 bits mediante las instrucciones `PACKUSDW` y `PACKUSWB`, en ese orden. Una vez concluido el empaquetado, procedimos a volcar dicho resultado a la misma posicion de memoria a partir de la cual iniciamos el ciclo.