DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Modelo SIMD

Organización del Computador II
Primer Cuatrimestre de 2015

Grupo Crash Bash/Ps1

Integrante	LU	Correo electrónico
Ituarte, Joaquin	457/13	joaquinituarte@gmail.com
Maddonni, Axel	200 /14	axel.maddonni@gmail.com
Oller, Luca	667/13	ollerrr@live.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Introducción

En este trabajo se presentan y comparan tres implementaciones, dos en Assembler y una en C, para la realización de tres filtros de imágenes. Estos filtros son el filtro Blur, el filtro Merge y el filtro HSL.

Las imágenes que utilizamos son múltiplos de cuatro y mayores a 16x16.

El filtro Blur toma una imagen y la suaviza, el filtro Merge toma dos imágenes del mismo tamaño y las mezcla en una sola imagen, mientras que el filtro HSL convierte los píxeles RGB a HSL, una vez convertidos se los modifica y finalmente los píxeles regresan a RGB.

La primera parte del trabajo tratará de exponer y explicar el funcionamiento de cada una de las implementaciones en assembler de forma individual. Luego, desarrollaremos una comparación entre ellas analizando las ventajas y desventajas, exponiendo en qué casos es conveniente utilizar una u otra dependiendo del contexto de uso, cuáles son las causas y qué maneras hay de optimizarlas.

Al finalizar las explicaciones, se detallarán además las conclusiones obtenidas y las problemáticas que se nos presentaron en la realización del trabajo práctico.

2. Desarrollo

2.1. Blur

2.1.1. Implementacion 1

Explicación assembler

En esta implementación modificamos de a un pixel a la vez, comenzando a recorrer la primer fila de la matriz que representa a la imagen hasta que ésta se acabe y luego repetir el proceso con la fila siguiente hasta alcanzar el borde cuyos píxeles no son modificados. Para calcular los valores resultantes de cada componente del pixel, mantenemos en cada iteración un puntero al primero de los 9 píxeles necesarios para el cálculo correspondiente al filtro Blur (ver enunciado).

Para poder calcular correctamente los píxeles, debemos tener cuidado de no utilizar el valor modificado de los píxeles vecinos que utilizamos para realizar las cuentas.

Para lograr lo dicho anteriormente, al comienzo del código creamos un vector del tamaño equivalente a una fila de la imagen, que lo vamos a utilizar para guardar temporalmente n píxeles, siendo n la cantidad de píxeles que tiene una fila.

Este vector lo inicializamos con los píxeles de la primera fila de la imagen. Aunque ésto no es necesario, de esta manera nos evitamos realizar el caso base correspondiente a la primera fila de la imagen, ya que ésta no se modifica, fuera del ciclo correspondiente.

Nuestra función opera de la siguiente manera: Hay dos ciclos, uno dentro del otro, en donde el ciclo externo itera sobre las filas y el ciclo interno lo hace sobre las columnas.

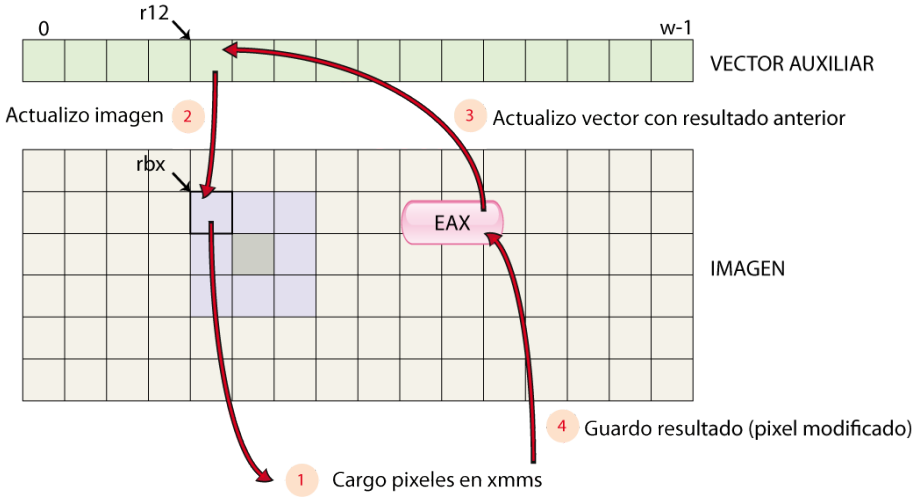

Figura 1: Desarrollo de Blur-ASM1.

1) En el ciclo interno guardamos en 3 registros xmm, en nuestro caso xmm0, xmm2 y xmm4, los primeros cuatro píxeles consecutivos de 3 filas contiguas. De estos 4 píxeles vamos a necesitar solamente 3, de forma que los píxeles que utilicemos sean los píxeles vecinos al pixel que vamos a modificar y a si mismo.

2) Una vez que los píxeles de la imagen están en los registros, modificamos el pixel de la imagen ubicado en la menor posición intercambiándolo con el pixel correspondiente del vector creado previamente, que contiene el pixel ya modificado correspondiente a esa posición de memoria.

3) Ahora que ya utilizamos la posición del vector para actualizar la imagen, guardamos el resultado que habíamos obtenido en la iteración anterior (que guardamos en EAX) en el vector. En caso que sea el primer pixel de la fila, antes de entrar al ciclo de la nueva fila se guarda en EAX el pixel correspondiente.

4) Realizamos las operaciones necesarias con los datos y guardamos el resultado (el pixel modificado) en EAX. Para sumar las componentes de los píxeles sin que ocurra overflow en nuestra representación, duplicamos el tamaño de cada una de ellas expandiendolas de byte a word. Dejamos dos píxeles en el registro en el que estaban (con el tamaño modificado) y los otros dos en otro registro xmm. Luego realizamos suma de words entre los registros, el resultado lo guardamos en xmm0. El siguiente paso es

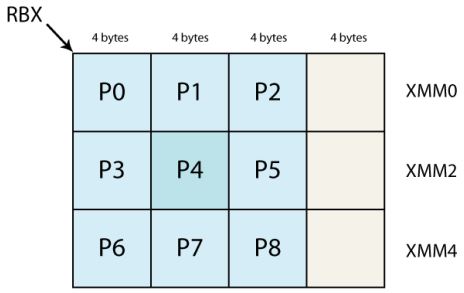

Figura 2: Desarrollo de Blur-ASM1.

shiftear a la derecha los registros xmm0, xmm2 y xmm4 y los sumamos para asi obtener en la parte baja del registro xmm0 la suma de los 9 pixeles involucrados. Duplicamos el tamaño de cada componente a 32 bits para convertir a float y dividir por nueve cada uno de ellos. Una vez concluído ésto, volvemos a convertir las componentes a enteros, y volvemos a empaquetarlas a su tamaño original, de forma que quede el pixel resultante en la parte más baja del registro xmm0. El resultado obtenido lo guardo en un registro para poder ubicarlo en el vector en la iteración siguiente. No lo podemos guardar en el vector ahora porque todavía necesitamos usar el valor correspondiente a esa posición en la próxima iteración.

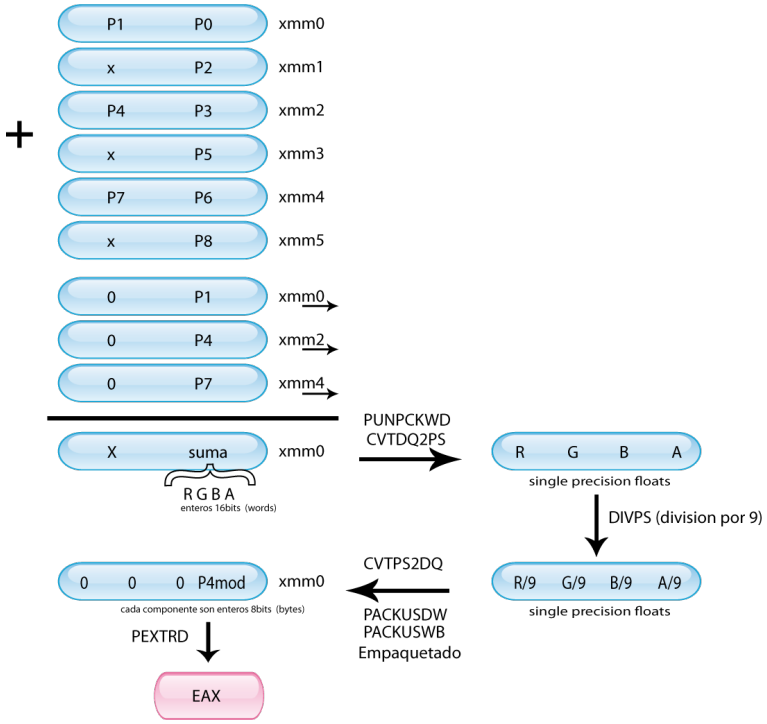

Figura 3: Desarrollo de Blur-ASM1.

Caso Borde: Antes de volver a iterar en el ciclo externo, se realiza el caso borde del final de la fila, que es análogo a una iteración del ciclo interno. Cuando se llega a la última columna de pixeles que se debe modificar, retrocedemos con el puntero al pixel anterior y tomamos los ultimos 3 pixeles de cada registro en vez de los 3 primeros. (Si no hariamos ésto, en la ultima fila intentariamos acceder a memoria que no es nuestra). Nota: Al finalizar el caso borde actualizamos el anteúltimo pixel de la fila y cargamos el resultado directo al vector, no hace falta guardarlo en EAX.

Una vez finalizado ésto, realizamos las operaciones necesarias para poder realizar la siguiente iteración del ciclo externo.

Al finalizar el ciclo de las filas, actualizamos la anteúltima fila con los valores modificados, guardados en el vector durante la última iteración del ciclo.

2.1.2. Implementacion 2

Explicación assembler

En esta implementación modificamos la imagen de a cuatro píxeles por vez, considerando a la imagen como una matriz de píxeles. El orden que seguimos para procesar los grupos de píxeles es: procesar los primeros cuatro píxeles de la imagen (sin contar los bordes) y hacer lo mismo con el siguiente grupo de píxeles hasta que llegamos al final de la fila, donde debemos procesar solo dos píxeles. Luego cambiamos de fila y repetimos el proceso hasta llegar a la última fila de la imagen.

Para lograr ésto debemos tener cuidado de no procesar píxeles con los valores modificados de sus vecinos. Al igual que en la anterior implementación, utilizamos un vector del tamaño de una fila de la matriz en donde guardaremos los valores modificados de la imagen hasta que hayamos utilizado a los píxeles que se deben reemplazar en todas las cuentas en las que se vean implicados. Este vector es inicializado con los píxeles correspondientes a la primera fila de la imagen para evitar problemas en la primer iteración del ciclo cuando actualizamos la imagen.

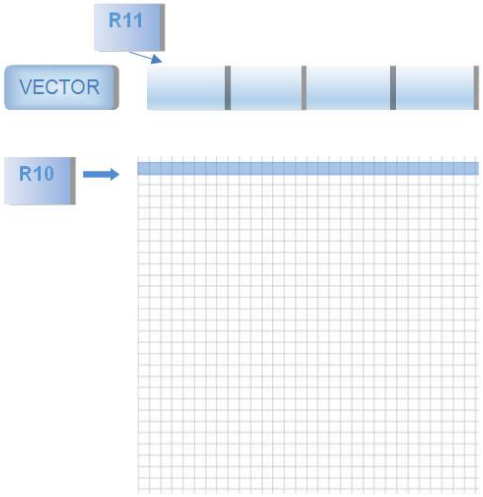

Figura 4: Desarrollo de Blur-ASM2.

Luego de haber inicializado el vector, comenzamos con los ciclos para el procesamiento de los píxeles utilizando dos ciclos: un ciclo externo que recorre las filas y un ciclo interno que recorre las columnas, contenido por el primero.

Al comenzar el ciclo externo, inicializamos los registros para manipular los datos que se van a utilizar en el ciclo interno. Estos datos son el ancho de la fila, que es nuestra referencia a cantidad de iteraciones a realizar en el ciclo interno, un puntero a la dirección de memoria del primero de todos los píxeles que vamos a utilizar, y un puntero al inicio del vector.

Luego comienza a realizarse el ciclo interno.

Al comenzar el ciclo interno, utilizamos nueve registros xmm para guardar los grupos de cuatro píxeles de la siguiente forma:

Figura 5: Desarrollo de Blur-ASM2.

Una vez cargados los píxeles en los registros, procedemos a actualizar los cuatro píxeles de la imagen original con los cuatro píxeles correspondientes al vector creado. Es decir, actualizamos el valor de los píxeles que ya terminamos de utilizar, y de manera análoga a la implementación 1, cargamos en el vector

el cuarto de los 4 pixeles que habíamos calculado en la iteración anterior y guardado en un registro, para no pisar los valores que necesitamos para actualizar la imagen.

Paso siguiente es duplicar el tamaño de las componentes de los píxeles en los registros xmm utilizados previamente. Como los registros ahora pasan a poder contener solamente dos píxeles, vamos a necesitar el doble de registros para poder guardar todos los píxeles necesarios.

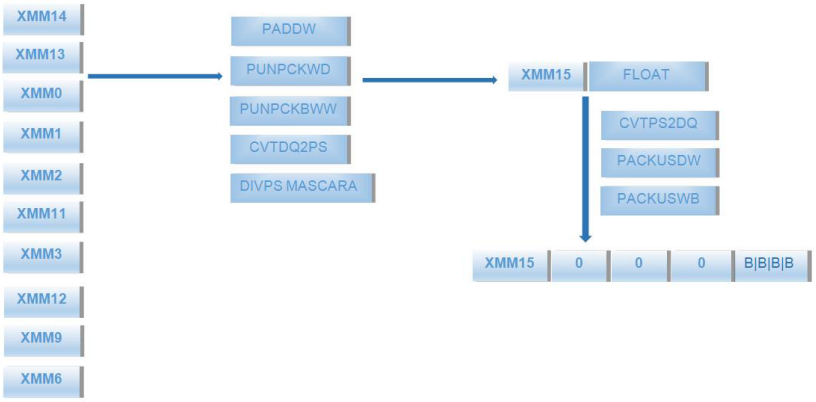

Figura 6: Desarrollo de Blur-ASM2.

A continuación, realizamos las operaciones para calcular un pixel. Hacemos la sumatoria de las componentes de los píxeles, guardando el resultado en otro registro, transformamos ese resultado a punto flotante de precisión simple para poder divirlas a todas a la vez y obtener el promedio. Una vez calculado, lo transformamos nuevamente a entero.

Una vez calculado el pixel, lo transformo a su tamaño original, y copio el pixel al vector creado al comienzo del código.

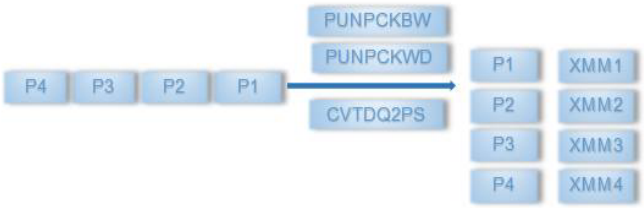

Figura 7: Desarrollo de Blur-ASM2.

Para calcular los otros tres píxeles, el procedimiento es similar al recién descrito, teniendo en cuenta la ubicación de los píxeles necesarios en los registros utilizados. El cuarto pixel, como se menciona antes, se guarda provisoriamente en un registro pues el valor que le correspondería en el vector sera utilizado al comienzo del próximo ciclo.

Al finalizar el ciclo interno, debemos procesar los últimos dos píxeles de la fila iterada debido a que el ciclo sólo procesa de a cuatro píxeles. Este procesamiento es análogo a una iteración del ciclo interno, pero manipulando con precaución los registros debido a que usamos menor cantidad de píxeles y de cantidad de registros xmm.

Finalizado el procesamiento de estos dos píxeles, cambiamos la fila que vamos a iterar en el ciclo interno de la siguiente iteración del ciclo externo.

El ciclo externo termina cuando no hay más filas de la imagen que procesar, dejando las últimas dos filas de la imagen sin modificar, y las modificaciones de la anteúltima fila de la misma imagen en el vector creado por nosotros.

Cuando termina de realizarse el ciclo externo, debemos copiar los datos del vector a la anteúltima fila de la imagen, y con esto finaliza la función.

2.2. Merge

2.2.1. Implementacion 1

Explicación Assembler

En la primera implementación del filtro Merge, realizamos dos ciclos anidados los cuales iteran sobre la fila y sobre las columnas de la matriz de la imagen respectivamente, similar a las implementaciones anteriores. En el ciclo que itera sobre las columnas, iteramos de a cuatro píxeles, recorriendo toda una fila dictada por el otro ciclo. Estos punteros se manipulan de manera análoga para ambas imágenes, dado que éstas tienen que ser de iguales dimensiones.

Al llegar al final de una fila, es decir, cuando el ciclo de las columnas termina, actualizamos los punteros de las imágenes sumándole al puntero el tamaño de la fila para cambiar a la siguiente, y realizamos las iteraciones hasta terminar de operar con todas las filas de la imagen.

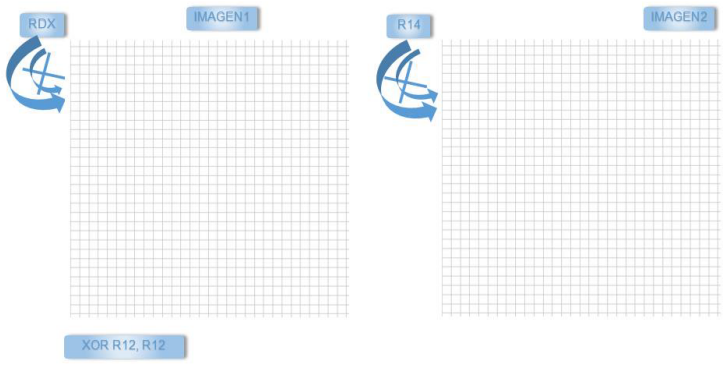

Figura 8: Desarrollo de Merge-ASM1.

Dentro del ciclo de las columnas guardamos en dos registros xmm los cuatro píxeles de cada imagen. Luego desempaquetamos cada registro en otros dos registros para incrementar el tamaño de las componentes de los píxeles y volvemos a repetir esta operación para obtener cada pixel ocupando un registro xmm, una componente por doubleword, y así convertirlos a float.

Luego realizamos el producto del RGB del píxel por nuestro value, dejando intacto A, y al píxel ubicado en la misma posición pero en la otra imagen hacemos el producto por 1-value. A estos valores los sumamos y convertimos a enteros, y los empaquetamos de dword a word y de word a byte para que el píxel recupere su valor original.

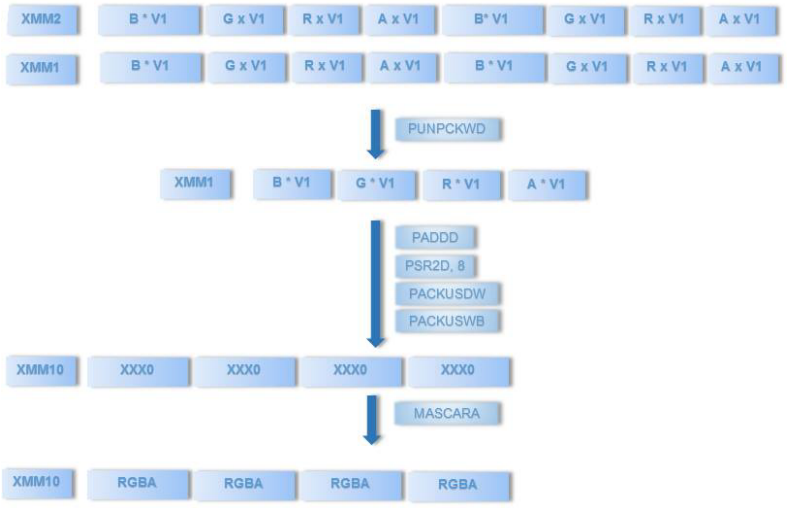

Figura 9: Desarrollo de Merge-ASM1.

Para finalizar, volvemos a guardar el pixel en memoria y termino de realizar la iteración, para volver a comenzar a repetir el proceso con los siguientes 4 píxeles.

2.2.2. Implementacion 2

Explicacion Assembler

En la segunda implementación del filtro Merge realizamos la implementación utilizando números enteros. Como value es un float entre 0 y 1, debimos encontrar una forma de manipular la función para que use valores en entero. Por ésto, debemos multiplicar a value por 256 y guardamos su valor y su complemento en los registros XMM0 y XMM15 con tamaño word en todas las posiciones posibles de los registros. Elegimos el valor 256, ya que luego al momento de dividir, al ser una potencia de 2 nos facilita la tarea. Además, tuvimos en cuenta de no pasarnos del valor máximo posible dado la cantidad de bits de un word, al momento de realizar las operaciones, para que no se produzca overflow.

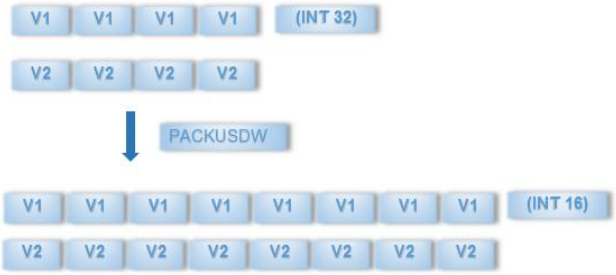

Figura 10: Desarrollo de Merge-ASM2.

Después de realizar esta manipulación de los datos, comenzamos a operar usando 2 ciclos anidados, copiando cuatro píxeles de cada imagen los registros xmm1 y xmm5 en cada iteración. En un tercer registro, en nuestro caso xmm10, guardamos la componente A de cada pixel ya que no debe ser modificada. Luego duplicamos el tamaño de las componentes de los píxeles de xmm1 y xmm5 y guardamos 2 píxeles por registro, usando además los registros xmm3 y xmm7 para poder contenerlos a todos.

A continuación procedemos a multiplicar los píxeles de los registros por value o su complemento, dependiendo a que imagen pertenecen, expandiendo el tamaño de las componentes de los resultados obteniendo los siguientes registros:

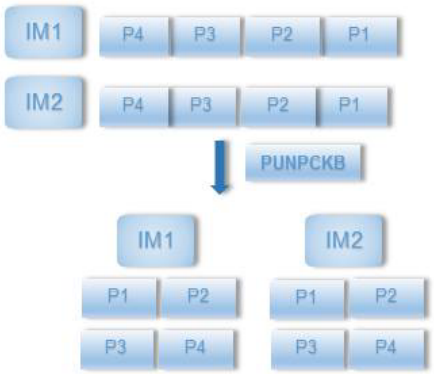

Figura 11: Desarrollo de Merge-ASM2.

Una vez obtenidos los píxeles de esta forma, procedemos a realizar la suma de los píxeles de una imagen con los correspondientes de la otra, y dividimos cada componente por 256 (el valor utilizado

al inicio para poder operar con enteros). Esta división la realizamos utilizando PSRLD XMMi, 8, que es equivalente a dividir cada componente por 256.

Finalizadas estas operaciones, continuamos por volver las componentes de los píxeles a su tamaño original, guardando los 4 píxeles en XMM1, y restaurando el valor de la componente A que habiamos guardado en XMM10.

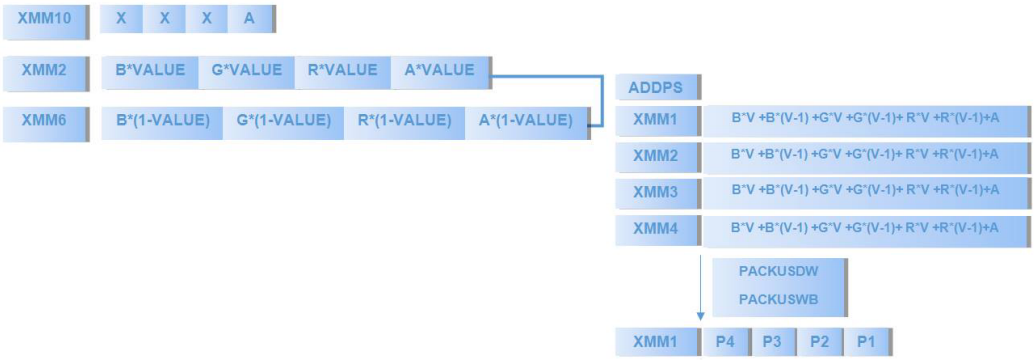

Figura 12: Desarrollo de Merge-ASM2.

Luego de terminar con las operaciones, guardo en memoria los píxeles modificados y repito las operaciones con los siguientes 4 hasta procesarlos a todos.

2.3. HSL

2.3.1. Implementacion 1

Explicación Assembler

La matriz se recorre de manera análoga a las implementaciones anteriores, utilizamos dos ciclos, uno para recorrer las filas y otro las columnas.

Esta implementación utiliza llamados a funciones de C para la transformación de formatos del pixel.

Para realizar ésto, guardamos en cada doubleWord del registro XMM0 los valores pasados como parámetro en la función, que corresponden al dato a sumar a cada componente del pixel. Además, creamos un vector en donde vamos a guardar el resultado de llamar a la función para transformar un pixel a formato HSL (utilizando una función C ya implementada).

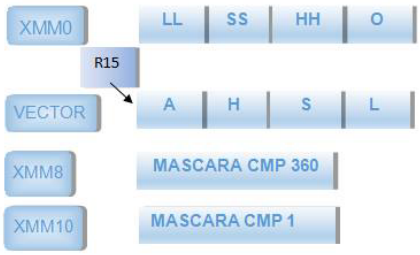

Figura 13: Desarrollo de HSL-ASM1.

Una vez realizado esto, procedemos a realizar las operaciones debidas a los píxeles iterando uno a uno todos los píxeles de la imagen.

En cada iteración comenzamos transformando las componentes del pixel rgb a componentes HSL con la función implementada en C correspondiente, usando el vector creado por nosotros como contenedor del resultado.

Una vez obtenido este resultado, lo cargamos en el registro XMM1 y le sumamos los datos correspondientes, que previamente habiamos guardado en XMM0. Luego controlamos que los valores obtenidos sean válidos y correctos según las fuciones dadas por la cátedra, implementando las funciones condicionales analizando si se cumplen las condiciones desde el final hacia el principio. Es decir, vemos si el último caso de la función condicional es válido. Si es válido, escribimos ese valor y procedemos a analizar el caso anterior de la función. Si también correcto cambiamos el valor escrito anteriormente por este, quedando como resultado final de la función el primer caso valido de la función.

Una vez finalizadas estas operaciones, guardamos el resultado en nuestro vector y procedemos a volver a convertir las componentes HSL a componentes rgb utilizando la función C correspondiente, la cual guarda el resultado final del pixel en la imagen.

Una vez terminadas todas las iteraciones, liberamos la memoria correspondiente a nuestro vector y terminamos la función.

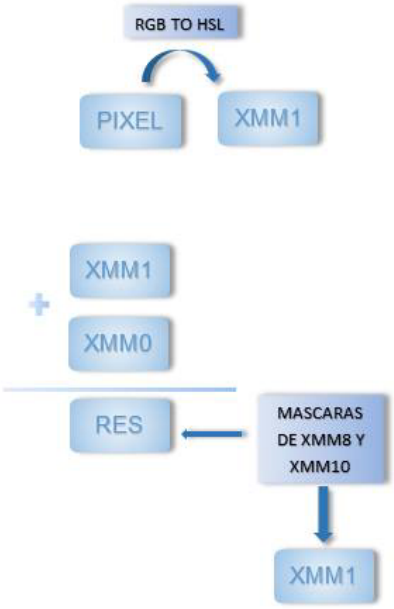

Figura 14: Desarrollo de HSL-ASM1.

2.3.2. Implementacion 2

Explicación Assembler

En esta implementación realizamos el filtro HSL completamente en código assembler.

Para eso, realizamos la misma implementación de suma que la primera implementación del filtro explicada anteriormente.

Para realizar la conversión del formato rgb al formato HSL, transformamos el tamaño cada componente del pixel de byte a doubleword y las guardamos en el registro XMM0, los convertimos a valores en punto flotante y guardamos la componente de transparencia, dado que no se modifica, en el vector.

Calculamos el máximo y el mínimo entre las componentes rgb del pixel y con estos valores procedemos a calcular la matiz (componente H), luego la luminosidad (componente L) y finalmente la saturación (componente S) según las funciones dadas por la cátedra. Como las funciones para calcular las componentes son condicionales, hemos decidido realizar las operaciones desde el último caso hasta el primero, preguntando si las condiciones se cumplen, y reemplazando el valor anterior en caso de cumplirse. Esto ahorra ademas, comparar cada valor con el valor maximo y el minimo posible correspondiente a cada if. Luego, los resultados de estas operaciones son guardados en el vector.

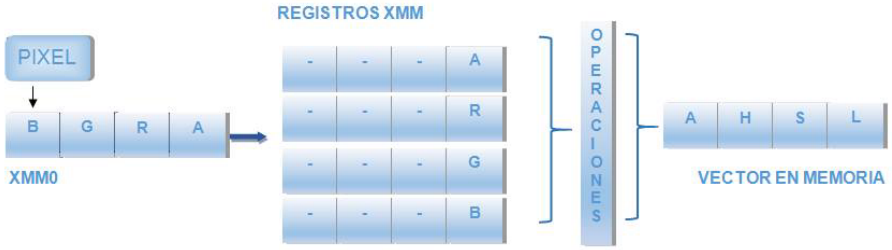

Figura 15: Desarrollo de HSL-ASM2.

Para realizar la conversión del formato HSL al formato rgb, necesitamos obtener algunos valores utilizando las fórmulas provistas por la cátedra, C, X y M, que los guardaremos en las partes bajas de XMM4, XMM5 y XMM6 respectivamente. Para calcular fabs utilizamos una mascara que pone en 0 el bit de signo, mientras que para calcular fmod utilizamos divisiones y conversiones a enteros y nuevamente a float para eliminar los decimales y poder hacer el calculo como muestra la figura. Luego procedemos a calcular los valores en formato rgb según los resultados de las funciones condicionales otorgadas también por la cátedra, utilizando la misma lógica sobre las condiciones como se hizo anteriormente en la función para transformar de rgb a HSL. En cada if, utilizamos instrucciones de SHUFFLE para ir modificando en caso de que sea necesario, los valores de R, G y B. El resultado de ésto es guardado en XMM4 y luego hacemos los cálculos de escala (multiplicar todas las componentes por 255) y los convertimos a enteros.

Figura 16: Desarrollo de HSL-ASM2.

Para finalizar, volvemos las componentes a su tamaño original en la parte baja de XMM4 y lo guardamos en su posición original en la imagen.

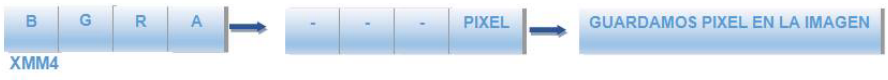

Figura 17: Desarrollo de HSL-ASM2.

3. Resultados

3.1. Consideraciones sobre los tests

- Los tests que implementamos para medir el rendimiento de los programas fueron implementados en C utlizando las instrucciones de assembler RDTSC para obtener el valor del Time Stamp Counter y CPUID para evitar que el código que queremos medir se realice con ejecución fuera de orden. Esta instrucción sincrónica, la usamos entonces, para serializar la ejecución del código.
- El código consiste en 1000 ejecuciones seguidas de las distintas implementaciones de los filtros, obteniendo los resultados con la cantidad de ciclos que tardo cada una de ellas en una planilla de excel .svc .
- Para evitar, en parte, el overhead, decidimos modificar las funciones run_[nombre_del_filtro] provistas por la cátedra, de modo que se comience a medir los ciclos justo antes de llamar a la funcion que aplica el filtro, sin contar las operaciones sobre los bmp que realiza antes y despues del llamado a esta.

```
int main(){
int ciclos = 1000;
FILE* archivo = fopen("Resultados_blur.csv", "w+");
char* src = "img/lena.bmp";
char* dst = "img/lenapruebaBLUR.bmp";

for (int i = 0; i<=ciclos-1; i++){
        for (int j = 0; j<=2; j++){
        run_blur(j, src ,dst, archivo);
    }
}
fclose (archivo);
return 0;
}
```

Figura 18: Código para testear rendimiento de Blur.

```
if(c==0){
            unsigned long start, end;
            unsigned long delta;
            RDTSC_START(start);

            C_blur(w,h,dataC);

            RDTSC_STOP(end);
            delta = end - start;

            fprintf(archivo, "%lu \t", delta);
             }
```

Figura 19: Extracto de código modificado de run_blur en run.c.

- Como primera observación, el código no arrojaba los mismos resultados si realizamos 1000 iteraciones seguidas de la misma implementación en lugar de ir alternandolas. Es decir, si hacíamos 1000 ejecuciones de C, luego las 1000 de ASM1, y finalmente las 1000 de ASM2, las últimas ejecuciones del código, principalmente las de C, observamos que el tiempo que tarda en ejecutar cada iteración se vuelve menor, en parte probablemente al aprovechamiento de la cache. Para solucionar ésto, decidimos alternar la ejecución de las implementaciones, de manera que nuestro código ejecute 1 de C, 1 de ASM2, 1 de ASM2, y luego comenzar de nuevo hasta completar las 1000 iteraciones.
- Una vez obtenidos los tiempos de cada iteración, usando las funciones que provee LibreCalc (Excel), calculamos la media de la muestra para cada implementación y su desvío estandar.

- Para filtrar los outliers, decidimos calcular el rango determinado por la fórmula a continuación, de manera que aquellas mediciones fuera de rango, sean sacadas de la muestra. De esa manera calculamos un nuevo promedio con la ”podada”de la muestra, y su nuevo desvío estandar.

  Rango aceptable para las mediciones:

$$(q_1 - 3 * IQR;\ q_3 + 3 * IQR),\ con\ IQR = q_3 - q_2$$

- Para realizar las mediciones utilizamos una notebook sony VAIO con procesador Intel Core i5, 5.7gb de memoria RAM y s.o. Ubuntu 14.04 de 64bits.

3.2. Experimento 1

Este primer experimento busca relacionar el rendimiento de las implementaciones de C, ASM1 y ASM2 de cada filtro, de manera general. Para ello, mostraremos en profundidad los resultados obtenidos por cada implementación para una imagen determinada, comparando con la versión no optimizada del compilador de C. (En este caso, lena.512x512.bmp, con value=0.5 en los casos del merge y hh=360, ss=0.2 y ll=0.1 en los casos del hsl.)

Filtro Blur

Implementación	C	ASM1	ASM2
Promedio sin podada	100052986,5	6664951	5460562
Promedio con podada	55340781,7668113	4256614,02164502	3676444,63465784
Desvío estándar sin podada	71610081,8818782	2856188,13697557	2005609,38988628
Desvío estándar con podada	9912205,09271232	628731,600331559	481408,065755489
Comparación con C (Porcentajes)	100,00%	7,69%	6,64%

Filtro Merge

Implementación	C	ASM1	ASM2
Promedio sin podada	35151468,5	2862264	2212913
Promedio con podada	22008679,1311301	1662354,5738576	1442829,12340426
Desvío estándar sin podada	22057078,1032933	2040435,81307328	1220788,7450202
Desvío estándar con podada	4756604,06921252	345749,611420407	288720,920554961
Comparación con C (Porcentajes)	100,00%	7,55%	6,56%

Filtro HSL

Implementación	C	ASM1	ASM2
Promedio sin podada	103668272,5	94494161	98231834
Promedio con podada	54893248,6604167	68525759,3784615	80719945,4959184
Desvío estándar sin podada	68388202,1593035	39123867,4780882	22101971,605724
Desvío estándar con podada	1617716,16958193	2127825,25103348	2154268,13457103
Comparación con C (Porcentajes)	100,00%	124,83%	147,05%

Nota: Los números representan cantidades de ciclos de reloj.

Figura 20: Tablas de Resultados para cada Implementación, 1000 iteraciones.

Como se puede notar en la tabla, para este caso, obtenemos los siguientes resultados con respecto al rendimiento de las implementaciones:

- En el caso de Blur, la implementación ASM1 corre aproximadamente 13 veces más rápido que la de C, mientras que la de ASM2 corre 15 veces más rápido. Ésto es lógico pues en la segunda implementación se procesa de a 4 píxeles a la vez, mientras que en la primera se procesa de a 1 por vez, sin embargo, no es una diferencia demasiado notable, teniendo en cuenta el desvío estandar de la muestra podada.
- En el caso de Merge, obtenemos aproximadamente las mismas relaciones con respecto a C. ASM1 funciona 13 veces más rápido que C, mientras que ASM2 15 veces más rápido. Ésto se debe a que

al trabajar con enteros, en ASM2, nos ahorramos las conversiones a floats de los componentes, además de un paso de desempaquetado extra que no es necesario ya que podemos trabajar directamente con los words enteros. Sin embargo, al observar las imágenes resultantes, aparece una pequeña, y a nuestro criterio, aceptable pérdida de precisión. Como mucho, los componentes de la implementación en enteros varían en 1 con respecto a las otras.

- Por último, en el caso de HSL, a diferencia de las anteriores, C le saca ventaja a las dos implementaciones de assembler. La de C tarda aproximadamente el 80 % de lo que tarda la de ASM1, y un 70 % de lo que tarda la de ASM2. Es decir, ASM2 es la de menor rendimiento. Existen diversos motivos que posibilitaron este resultado:
  - En principio, para las llamadas a rgbTOhsl y viceversa, fue necesario tener un vector auxiliar donde guardar los resultados de dichas funciones, haciendo que se necesite un nivel alto de accesos a memoria.
  - Además, la complejidad de las conversiones, en caso de ASM2, complejizaron el algoritmo, pues en C la secuencia de ifs se resuelve de manera más rápida que de la forma en que esta implementada usando instrucciones de SIMD. En assembler, y con esta implementación, es necesario corroborar todas las condiciones de los ifs anidados, y actualizar los valores en caso de que se cumplan, a diferencia de C, en donde una vez que se cumpla una de ellas no es necesario revisar las demás. Además, para complejizar aún más, se utilizan en varias ocasiones máscaras, que se traducen en más accesos a memoria, e instrucciones de SHUFFLE para facilitar la manipulacion del orden de los componentes durante la conversión.
  - Otro punto a tener en cuenta, es por ejemplo, la manera en que están implementadas las funciones fabs y fmod en C, pues en nuestra implementación fabs corresponde a un acceso a memoria y la aplicación de una máscara que modifica el bit de signo, mientras que para calcular fmod es necesario una serie de operaciones, incluida una conversión de float a entero. Ésto hace probable que éstas, y algunas otras funciones en C puedan estar implementadas más eficientemente.
  - Por último, y no menos importante, las llamadas reiterativas a C en ASM1 producen overhead.

El siguiente gráfico muestra de manera más clara las relaciones comentadas anteriormente entre las distintas implementaciones:

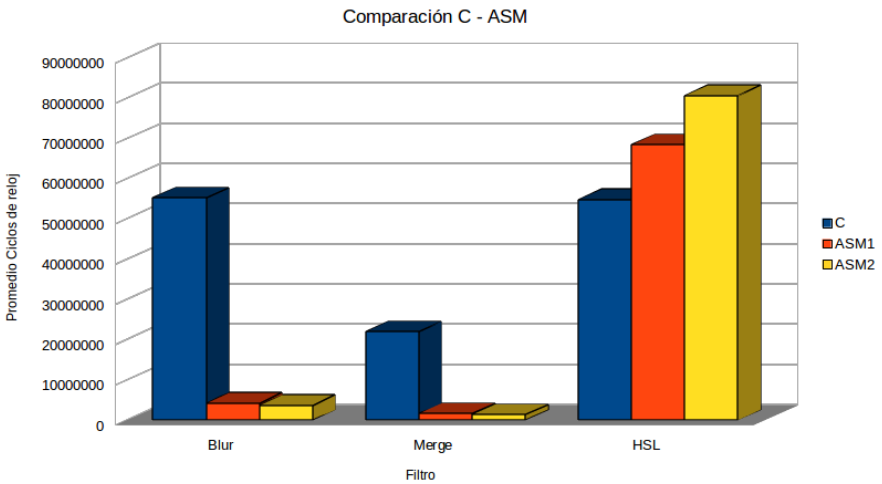

Figura 21: Gráfico de Barras, Comparación C-ASM, Experimento 1.

3.3. Experimento 2

En este experimento comparamos el rendimiento de las implementaciones con respecto a los diferentes tamaños de entrada. Para ello, modificamos la imagen lena.bmp y medimos los tiempos de ejecución para cada tamaño. Decidimos hacerlo solo para Blur y Merge para ganar claridad en el gráfico, ya que los valores de los tiempos promedio de HSL difieren bastante con los de Blur y Merge.

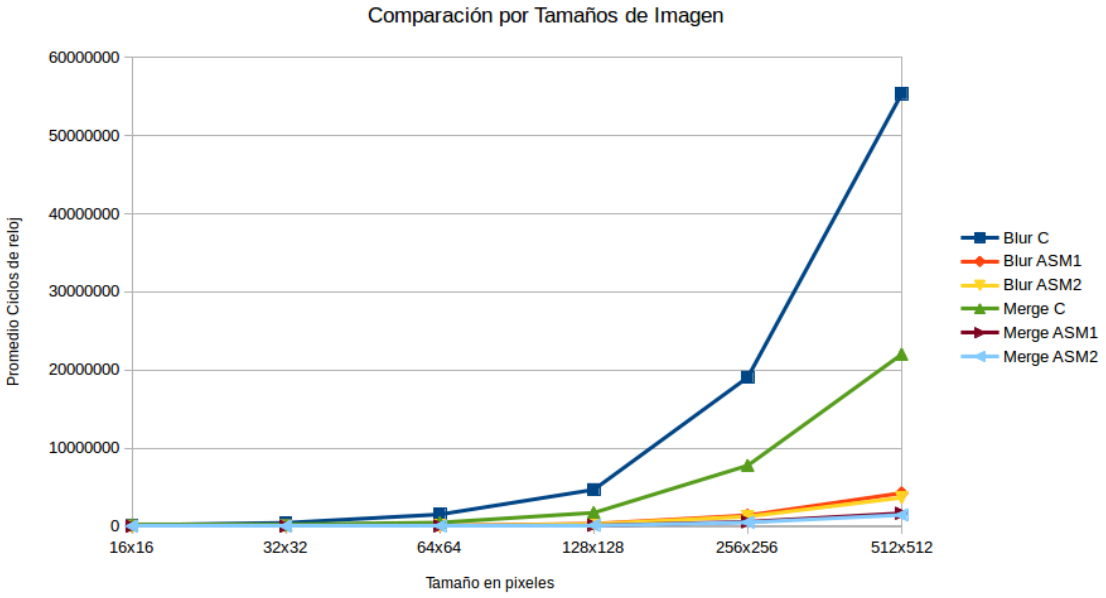

Figura 22: Gráfico de Líneas, Experimento 2.

Como resultado importante, vemos que al aumentar el tamaño de la entrada (en cada punto se cuadriplica el tamaño de la imagen), la version de C crece mucho más rápido que la de ASM, es decir, la cantidad de ciclos que le toma a C procesar imágenes cada vez más grandes aumenta más velozmente que lo que le lleva a las funciones en asm. De manera que cuanto mayor es el tamaño de la imagen, más notoria es la ventaja de los programas realizados en asm.

3.4. Experimento 3

En este experimento, comparamos cada implementación de los filtros contra las distintas optimizaciones posibles del compilador gcc para la version implementada en C.

Observamos que la eficacia de los programas implementados en C depende directamente del grado de optimización con el que lo compilamos. En los casos de Blur y Merge, aún comparando con la optimizacion 3, las versiones en assembler andan más rápido que las de C, mientras que HSL de C supera a las de ASM con más amplia claridad a medida que aumentamos el nivel de optimizacion.

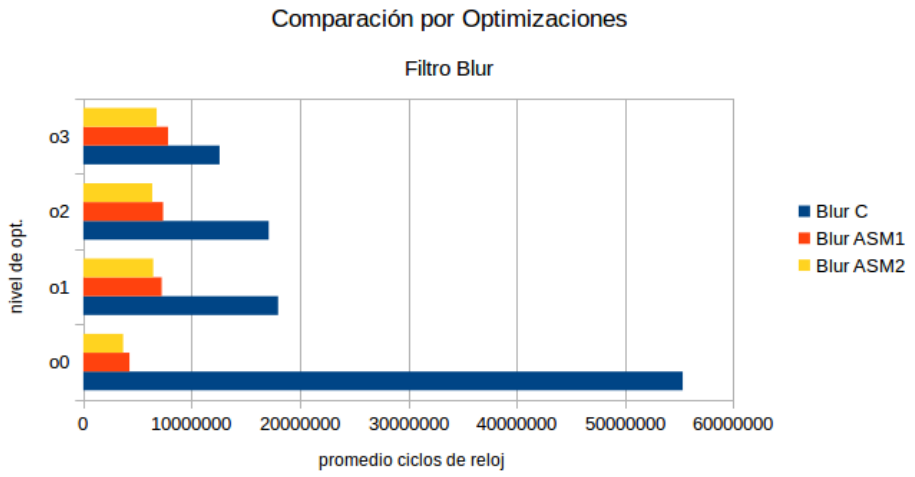

Figura 23: Filtro Blur, Experimento 3.

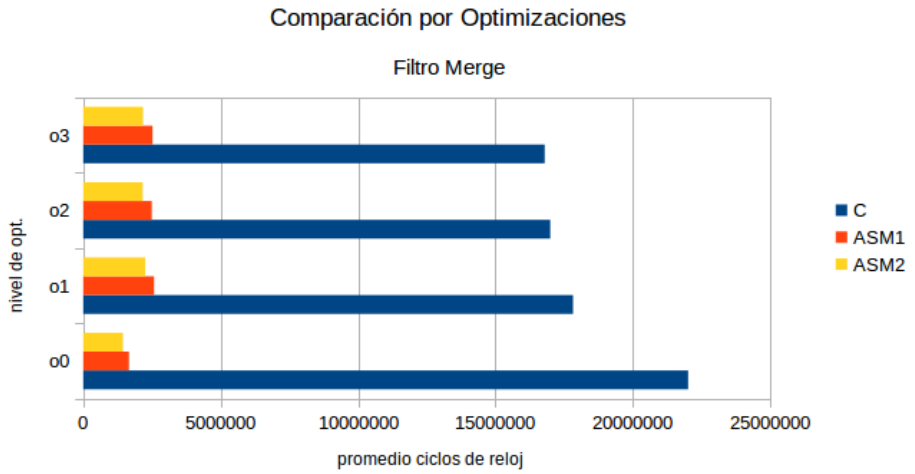

Figura 24: Filtro Merge, Experimento 3.

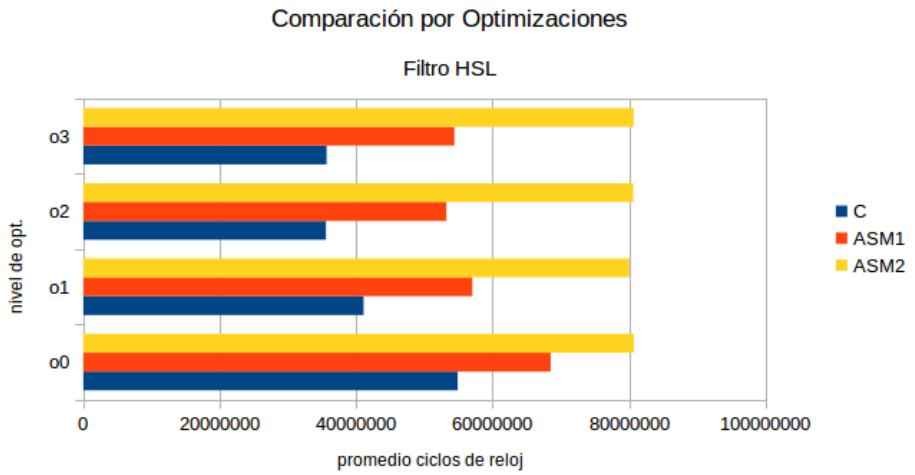

Figura 25: Filtro HSL, Experimento 3.

3.5. Experimento 4

En este experimento ejecutamos y comparamos las implementaciones con imágenes de distintos colores de fondo, siendo estos Rojo, Verde, Azul, Blanco y Negro.

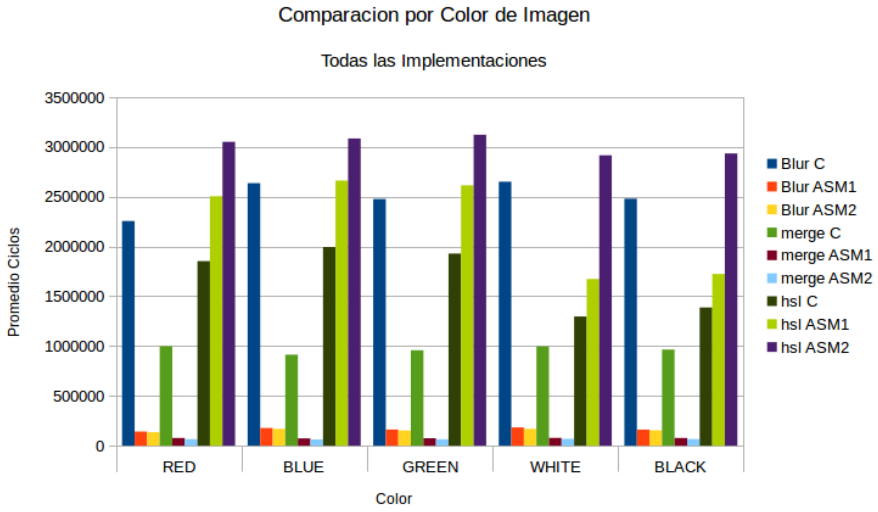

Figura 26: Comparación Filtros, Experimento 4.

Como se ve en la figura, en la mayoría de las implementaciones no se nota una diferencia importante, todas se comportan de manera similar, excepto HSL, que como se ve en detalle en la próxima figura, sus implementaciones de C y de ASM1 tardan menos con imágenes blancas o negras.

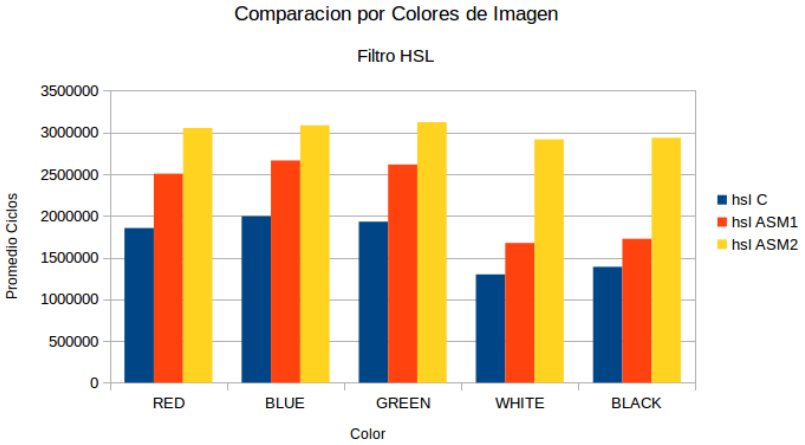

Figura 27: Comparación Filtros HSL, Experimento 4.

Ésto se debe a que para realizar las conversiones de rgb a hsl y viceversa, la complejidad del algoritmo para píxeles en blanco o negro se simplifica, ya que en ambos casos entrara en el primero de los ifs de las conversiones, sin necesidad de entrar a los else, mientras que en el ASM2 si debera ejecutar todas las ramas, reemplazando en caso de que sea necesario por el nuevo valor. Como conclusión podemos sacar, que ASM2 corre con desventaja con respecto a ASM1 y C para todos los píxeles cuyas componentes coincidan en su valor, ya que en esos casos también se simplifica el cálculo para la conversión de un modelo a otro en la funcion de C. (Si los componentes son iguales, cmax==cmin , haciendo que el valor de h sea 0, y si al sumar queda menor de 60, cuando se convierta a RGB tambien alcanzara con el primer if, sin necesidad de ejecutar los otros). Hay que tener en cuenta que por cada if, nuestra implementacion utiliza instrucciones de SHUFFLE, cuyos costos ifluyen directamente en el rendimiento de ASM2.

3.6. Experimento 5

Como último experimento, decidimos analizar si podríamos optimizar los algoritmos reduciendo la cantidad de saltos condicionales. Para ello propusimos dos maneras de comprobar si esto valdría la pena. Las mediciones de este experimento se realizaron, por comodidad, sobre las implementaciones del filtro Merge. Por un lado, medimos el promedio de 1000 iteraciones aplicando la función a dos imágenes de 32x32 y 1024x1 pixel respectivamente, de manera que ambas tienen la misma cantidad de píxeles, pero la segunda sólo entrará una vez al ciclo de filas. Por otro lado, modificamos el algoritmo de manera que se recorra la imagen tan sólo en un ciclo, calculando la cantidad de iteraciones como el alto en píxeles de la imagen por el ancho de la imagen multiplicado por 4 (es decir, por el ancho en bytes de una fila de la imagen).

Filtro Merge

Tamaño de Imagen	*512x512*	*512x512*
	Versión con dos ciclos de saltos	**Versión con 1 ciclo de saltos**
C	22008679,1311301	22665118,345912
ASM1	1662354,5738576	1684454,94375
ASM2	1442829,12340426	1479067,13263158

Tamaño de Imagen	*32x32*	*1024x1*
	Imagen Cuadrada	**Imagen Alargada**
C	157979,93559322	171676,212253829
ASM1	10817,4025374856	12291,9731182796
ASM2	9326,9359267735	10630,8998923574

Nota: Los números representan cantidades promedio de Ciclos reloj para 1000 iteraciones.

Figura 28: Resultados para Filtros Merge con menos saltos, Experimento 5.

Como se muestra en la tabla, en ambos casos no obtuvimos mejoras significativas, sino que se comportaban muy similar entre las dos. Con esto llegamos a la conclusión de que en imágenes relativamente pequeñas, utilizar 1 o 2 ciclos para recorrer la matriz no modifica la performance del algoritmo. Queda pendiente probar que sucede en mapas de bits realmente grandes, pero consideramos que estos filtros no trabajan sobre imagenes de tal tamaño.

4. Conclusiones

Vimos que el conjunto de instrucciones de SSE puede utilizarse para sacar provecho de la paralelización de cómputos y el acceso a memoria de a bloques de datos contiguos. Observamos que en los casos que se logró procesar más de un dato a la vez el rendimiento del algoritmo fue mayor.

Además, observamos que el código de assembler generado por el compilador GCC, sin optimización, produce una gran cantidad de accesos a memoria al utilizar variables locales en la pila lo cual produce una penalización en el rendimiento del proceso.

Las circunstancias por las cuales una medición de rendimiento puede diferir de otra pueden ser muy variadas. Es importante tener en cuenta una serie de medidas para reducir al mínimo posible el error cometido, algunas de ellas utilizadas y explicadas en la sección de experimentos. Probar por nuestra cuenta y ver resultados concisos nos hizo entender mejor como maneja el procesador este tipo de operaciones, y qué tener en cuenta al momento de diseñar un algoritmo, en este caso para resolver funciones que involucran datos multimedia.