DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Recuperatorio Trabajo Práctico II

Grupo: Spyro_/_Playstation_1

Organización del Computador II
Primer Cuatrimestre de 2015

Integrante	LU	Correo electrónico
Iván Arcuschin	678/13	iarcuschin@gmail.com
Martín Jedwabny	885/13	martiniedva@gmail.com
Dan Zajdband	144/10	dan.zajdband@gmail.com

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

Índice

1. Introducción **3**

2. Metodología para medición de tiempos **4**

3. Ejercicio 1 - Blur **6**
3.1. Desarrollo 6
3.1.1. Blur 1 6
3.1.2. Blur 2 8
3.2. Análisis y Resultados 10
3.2.1. Hipótesis planteadas 11
3.2.2. Experimentación 11

4. Ejercicio 2 - Merge **16**
4.1. Desarrollo 16
4.1.1. Merge 1 16
4.1.2. Merge 2 17
4.2. Análisis y Resultados 18
4.2.1. Hipótesis planteadas 19
4.2.2. Experimentación 20

5. Ejercicio 3 - HSL **22**
5.1. Desarrollo 22
5.1.1. HSL 1 22
5.1.2. HSL 2 24
5.2. Análisis y Resultados 28
5.2.1. Hipótesis planteadas 28
5.2.2. Experimentación 29

6. Conclusión **32**

A. Anexo - metodología para las mediciones **33**

1. Introducción

Los objetivos de este Trabajo Práctico fueron los siguientes:

- Mejorar nuestra capacidad de programación en lenguaje *Ensamblador*: desde trabajar con las instrucciones más usuales, hasta utilizar el set de instrucciones *SSE* (Streaming SIMD Extensions) que permite realizar operaciones sobre muchos datos al mismo tiempo, generalmente enfocadas en procesamiento multimedia.
- Comprender las ventajas y desventajas de programar en lenguaje *Ensamblador*: desde la diferencia de velocidad con la implementación en lenguaje *C*, hasta la complejidad asociada al programar en un lenguaje de bajo nivel.

De acuerdo a lo anterior, se dividió en tres partes centrales:

- En primer lugar, se realizaron 2 implementaciones distintas en lenguaje *Ensamblador* de 3 filtros de imágenes distintos (un total de 6 implementaciones). Estos filtros fueron: **Blur**, **Merge** y **HSL**.
- En segundo lugar, se realizaron distintos experimentos para cada algoritmo comparando sus distintas implementaciones: **C**, **ASM1** y **ASM2** (siendo las realizadas en lenguaje **C** dadas por la cátedra)
- Por último, dichos experimentos permitieron realizar un análisis de la performance en cada implementación y comparar sus ventajas y desventajas.

2. Metodología para medición de tiempos

A continuación vamos a detallar la metodología utilizada para recolectar los datos que luego se utilizarán en el análisis de las diferentes implementaciones.

La medición del tiempo de ejecución se realizó utilizando la instrucción de *x86 rdtsc* (utilizando como ayuda el archivo *rdtsc.h* provisto por la cátedra), que permite obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Obteniendo la diferencia entre los contadores antes y después de la llamada a la función, podemos obtener la cantidad de ciclos de esa ejecución. Esta cantidad de ciclos no es siempre igual entre invocaciones de la función, ya que este registro es global del procesador y se ve afectado por una serie de factores, a ser:

1. La ejecución puede ser interrumpida por el scheduler para realizar un cambio de contexto, esto implicará contar muchos mas ciclos (outliers) que los estrictamente necesarios para ejecutar nuestra función. En otras palabras, no tenemos garantizado, en principio, que nuestra función corra de inicio a fin sin que en el medio el procesador no se utilice para alguna otra función.

2. La frecuencia del reloj del procesador puede variar en el medio de la ejecución. Los procesadores modernos varían este valor para adecuarse a la demanda concreta en un tiempo determinado.

3. El procesador en el cual corre nuestra función puede ser utilizado por el sistema para atender alguna Interrupción (ya sea de Hardware o de Software). Esto se debe a que el sistema trata de *balancear* las interrupciones entre los distintos CPUs de una máquina para que todos funcionen de manera homogénea.

Para minimizar estas problemáticas ideamos una serie de estrategias que ayuden a tener una medición en tiempo real más fiable:

1. Se utilizó la herramienta *cpuset*[1] que permite controlar la ubicación de un proceso en la memoria y en los distintos procesadores. De esta forma, definimos un cpuset *sys* para los primeros $n-1$ cores que contuviera a todos los procesos del sistema y otro cpuset *rt* para el n-esimo core sin ningún proceso asignado.

   Al iniciar el script de medición de tiempos, le asignamos su *PID* a los procesos del cpuset *rt*, lo que lo aísla de cualquier interferencia que pueda provenir del scheduler del sistema, ya que este no puede iniciar procesos en el cpuset *rt*.

2. Se utilizó la herramienta *cpufreq-selector*[2] que permite controlar la frecuencia de un procesador en particular. De esta forma, fijamos la frecuencia del n-esimo core a la más baja posible..

3. Para mitigar las interrupciones al n-esimo core, se reemplazaron los valores en las *smp_affinity* de todas las interrupciones listadas en la carpeta */proc/irq*, de tal forma que el core aislado no esa elegible para procesar las interrupciones.

   Si bien esto es útil, sigue habiendo interrupciones mínimas en el n-esimo core, como por ejemplo sincronización con los otros cores e interrupciones de timer.

Para más información, se agrega en el apéndice A el script utilizado para inicializar el entorno de medición.

Otras consideraciones generales:

- Tomamos los tiempos de las versiones en C de los filtros con optimizaciones -O2.
- Todos los experimentos fueron hechos bajo las mismas condiciones.
- Tomamos más de 50 muestras de cada experimento.
- Sacamos los outliers y luego promediamos los resultados en cada caso.

[1] Referencia `http://man7.org/linux/man-pages/man7/cpuset.7.html`
[2] Referencia `http://csurs.csr.uky.edu/cgi-bin/man/man2html?cpufreq-selector+1`

Para analizar la diferencia de performance medida en tiempos de ejecución entre versiones de un mismo filtro, usamos el 'Cambio Relativo'. Esta cuenta se calcula de la forma:

$$\frac{\Delta T}{|T_1|} * 100 = \frac{T_2 - T_1}{|T_1|} * 100$$

o

$$\frac{\Delta T}{|T_2|} * 100 = \frac{T_1 - T_2}{|T_2|} * 100$$

Donde T_1 y T_2 son los tiempos de dos implementaciones. Usando este resultado podemos ver el cambio relativo de T_1 a T_2 (primera forma) o de T_2 a T_1 (segunda forma). Dicho de otra forma, en que porcentaje "mejora" o "empeora" T_1 con respecto a T_2 (si el cambio es negativo decimos que T_2 mejora a T_1, mientras que si es positivo decimos que T_2 empeora a T_1).

3. Ejercicio 1 - Blur

3.1. Desarrollo

El filtro de Blur consiste en representar en los píxel de la imagen original como el promedio de este con sus 8 vecinos inmediatos.

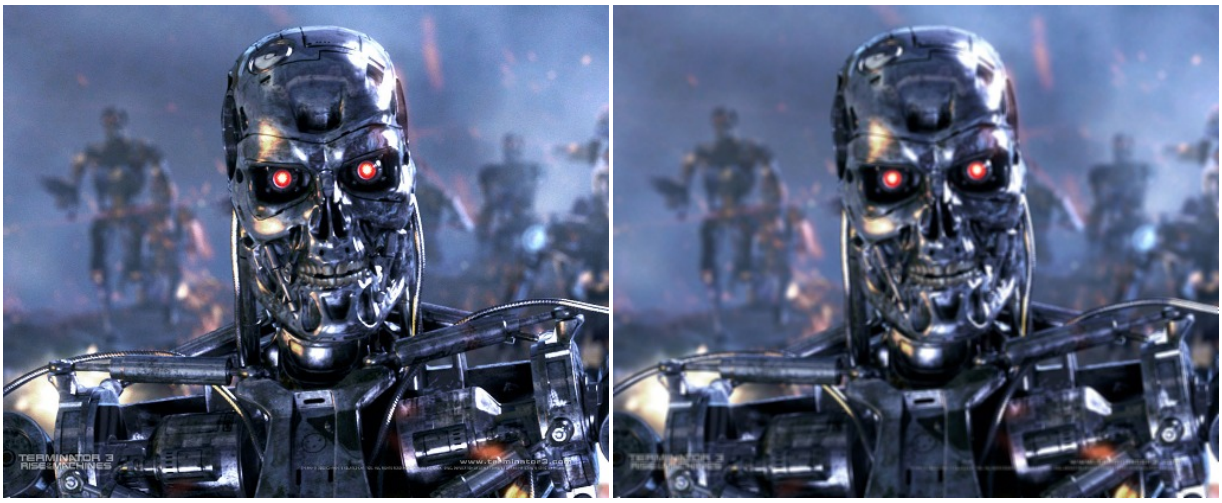


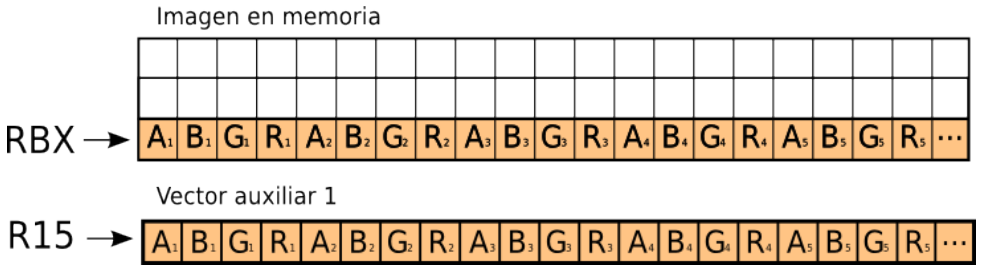
Imagen original | Resultado obtenido al aplicarle el filtro Blur

3.1.1. Blur 1

La primera implementación del filtro Blur se enfoca en entender como generar cada píxel de la imagen resultante y construir este proceso de la manera más sencilla posible. En esta implementación, cada píxel nuevo se genera en una iteración diferente del algoritmo.

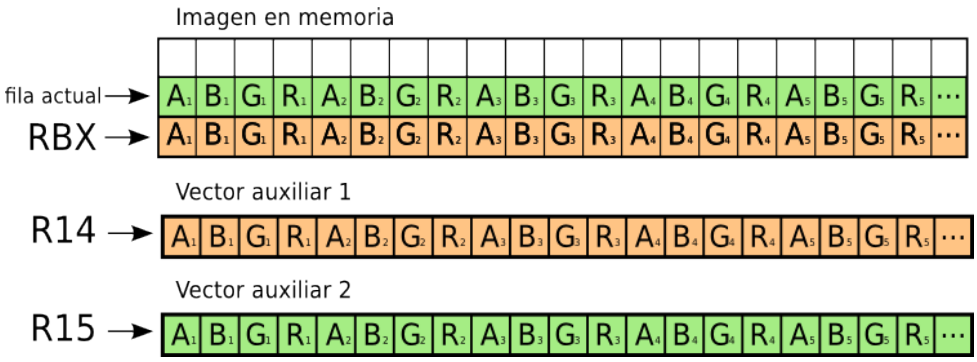
Procedimiento general Al comienzo del filtro se pide memoria para almacenar dos vectores del tamaño de una fila (ancho de la imagen), de modo que se puedan guardar los resultados parciales de filas que se tengan que reutilizar más adelante sin polucionar la imagen original. Los punteros a estos dos vectores se guardan en `R14` y `R15`. También se guarda el puntero al inicio de la imagen en el registro `RBX`.

Inicialmente, copiamos la primera fila de la imagen en el vector que apunta `R15`.



Luego, el algoritmo recorre las filas de la imagen y dentro de cada fila itera por cada columna de la misma.

Al momento de empezar a operar con una nueva fila, el algoritmo intercambia los punteros en `R14` y `R15`, y luego carga en `R15` la fila siguiente a la actual. De esta forma, tenemos en `R14` la fila actual original y en `R15` la fila siguiente a la actual (la fila en la cual deberíamos ir escribiendo los píxel resultantes), por lo que no perdemos información durante los cálculos de la fila.



Al finalizar todas las iteraciones, se libera la memoria pedida y cuyos punteros tenemos en `R14` y `R15`.

Iteración Por cada iteración se cargan los vecinos del píxel a procesar en 3 registros `XMM` utilizando la instrucción `MOVDQU` (sabemos hay un píxel de más cargado en cada registro).

Podemos imaginarnos la ubicación del píxel que estamos procesando respecto a sus vecinos como una matriz de 3x3 en la que la celda del centro es el píxel en cuestión.

La fila anterior a la actual se lee tomando como base el puntero en `R14` y se guarda en el registro `XMM0`, la fila actual se lee tomando como base el puntero en `R15` y se guarda en el registro `XMM1`, y la fila siguiente a la actual se lee tomando como base la fila actual más el ancho de la imagen y se guarda en el registro `XMM2`.

Ya que estamos procesando imágenes en RGBA, los valores almacenados están en formato de Enteros de 8 bits, por lo que los convertimos a Enteros de 16 bits (1 Word) utilizando las instrucciones `PUNPCKLBW` y `PUNPCKHBW`. Esto es necesario ya que si realizáramos las operaciones aritméticas en 8 bits correríamos el riesgo de tener *overflow*. Los nuevos valores se almacenan de la siguiente forma:

- La parte baja de `XMM0` en Enteros 8 bits va a `XMM0` como Enteros 16 bits.
- La parte alta de `XMM0` en Enteros 8 bits va a `XMM3` como Enteros 16 bits.
- La parte baja de `XMM1` en Enteros 8 bits va a `XMM1` como Enteros 16 bits.
- La parte alta de `XMM1` en Enteros 8 bits va a `XMM4` como Enteros 16 bits.
- La parte baja de `XMM2` en Enteros 8 bits va a `XMM2` como Enteros 16 bits.
- La parte alta de `XMM2` en Enteros 8 bits va a `XMM5` como Enteros 16 bits.

Ahora procedemos a sumar las componentes en los registros `XMM` mencionados utilizando la instrucción `PADDW`, que suma números empaquetados de a *Words*. El resultado de sumar las componentes de los píxel de la primera y segunda columna de la matriz de 3x3 quedan almacenados e la parte baja y alta respectivamente de `XMM0`. Y el resultado de sumar las componentes de los píxel de la tercera columna queda almacenado en la parte baja del registro `XMM3`.

Notemos que en la parta alta de `XMM3`, ahora tenemos la sumatoria de los píxel que seguían a nuestra matriz de vecinos de 3x3, por lo que limpiamos esa parte para poder realizar los siguiente pasos sin acarrear basura. Esto se realiza simplemente con la instrucción `PAND` y una máscara que tiene ceros en la parte alta y unos en la parte baja

Una vez realizado lo anterior, se procede a seguir sumando: utilizamos la instrucción `PADDW` y `PSRLDQ` para que nos quede en la parte baja de `XMM0` la sumatoria de todos los píxel de la matriz.

El siguiente paso es hacer la división de estos valores por 9, pero para eso necesitamos: primero convertirlos a Enteros de 32 bits y luego convertirlos a Floats Single Precision (de ahora en adelante Floats SP). Lo primero se realiza con la instrucción `PUNPCKLWD` y lo segundo con `CVTDQ2PS`. Una vez que hicimos esto, cargamos en el registro `XMM8` una mascara que tiene nueves en formato Floats SP, y realizamos la división de floats empaquetados entre `XMM0` y `XMM8` con la instrucción `DIVPS`, lo que nos deja el promedio de cada canal (R, G, B y A) en formato Float SP en `XMM0`.

Ahora, volvemos a convertir los datos a Entero de 8 bits con las instrucciones: `CVTPS2DQ`, `PACKUSDW` y `PACKUSWB` (en ese orden), lo que nos deja en los 32 bits mas bajos de `XMM0` los valores. Utilizamos los `PACK` sin signo ya que sabemos que nuestros valores no contenían números negativos.

Por último, se escriben los 32 bits más bajos de `XMM0` en el píxel del centro de la matriz con la instrucción `MOVD`, utilizando para esto el registro `RBX`. Nótese que para los cálculos de los píxel que siguen este valor escrito en la matriz no va a influir, ya que tenemos su valor original almacenado en el vector auxiliar al que apunta `R15`.

Observación: el último píxel de la imagen a procesar requiere un caso especial en el algoritmo, ya que si cargáramos los datos como para cualquier otro píxel estaríamos leyendo datos que se van de la posición de la imagen en memoria. Es por esto que lo que hacemos es restarle al contador de columnas un píxel, levantar cuatro píxel como hacíamos en una iteración normal, y luego shiftear los registros `XMM0`, `XMM1` y `XMM2` 4 bytes (1 píxel) con la instrucción `PSRLDQ`. Hecho esto, sigue la ejecución normal de la función.

3.1.2. Blur 2

La segunda implementación del filtro Blur óptimiza la implementación anterior al utilizar todo el potencial que brindan las instrucciones *SSE* para procesar 4 píxel en una sola iteración.

Procedimiento general La primera parte del algoritmo es similar a la del primero, al punto que se reutiliza el código de Blur 1. La diferencia principal consiste en que las iteraciones saltean de a 4 píxel en vez de a 1, siendo un punto no menor ya que se realizan 1/4 de las iteraciones que en el caso de Blur 1.

En cuanto a los vectores auxiliares, se utilizan los mismos registros para mantener sus punteros: `R14` y `R15`, y durante el algoritmo se manejan de la misma manera a la descripta en el Blur 1.

Iteración En este algoritmo, para poder procesar 4 píxel en la misma iteración se utilizan 6 registros `XMM` del siguiente modo:

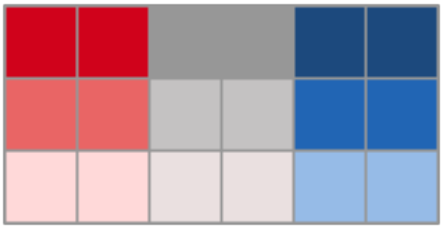


Figura 2: Registros `XMM` en Blur 2

En la Figura 2 se muestra los píxel ocupados por cada registro `XMM`. Los rojos corresponden a las columnas 1, 2, 3 y 4 mientras los azules corresponden a las columnas 3, 4, 5 y 6. Es por eso que los píxel de las columnas 3 y 4 se ven en gris, ya que se encuentran en ambos registros correspondientes a la misma fila.

La forma exacta en la que se guardan los píxel (todos en formato Enteros 8 bits) es:

- En `XMM0` los píxel 1 a 4 de la fila anterior a la actual.
- En `XMM1` los píxel 1 a 4 de la fila actual.
- En `XMM2` los píxel 1 a 4 de la fila siguiente a la actual.
- En `XMM3` los píxel 3 a 6 de la fila anterior a la actual.
- En `XMM4` los píxel 3 a 6 de la fila actual.
- En `XMM5` los píxel 3 a 6 de la fila siguiente a la actual.

Luego, convertimos los valores a Enteros de 16 bits (1 Word) utilizando las instrucciones `PUNPCKLBW` y `PUNPCKHBW`. La disposición de los valores queda (todos en formato Enteros 16 bits):

- En XMM0 los píxel 1 y 2 de la fila anterior a la actual.
- En XMM6 los píxel 3 y 4 de la fila anterior a la actual.
- En XMM9 los píxel 5 y 6 de la fila anterior a la actual.
- En XMM1 los píxel 1 y 2 de la fila actual.
- En XMM7 los píxel 3 y 4 de la fila actual.
- En XMM10 los píxel 5 y 6 de la fila actual.
- En XMM2 los píxel 1 y 2 de la fila siguiente a la actual.
- En XMM8 los píxel 3 y 4 de la fila siguiente a la actual.
- En XMM11 los píxel 5 y 6 de la fila siguiente a la actual.

A continuación, se realizan las sumas con la instrucción PADDW, quedándonos:

- En la parte baja de XMM0 la suma de la primera columna.
- En la parte alta de XMM0 la suma de la segunda columna.
- En la parte baja de XMM6 la suma de la tercera columna.
- En la parte alta de XMM6 la suma de la cuarta columna.
- En la parte baja de XMM9 la suma de la quinta columna.
- En la parte alta de XMM9 la suma de la sexta columna.

Luego, pasamos los valores a Enteros de 32 bits con las instrucciones PUNPCKLWD y PUNPCKHWD. Esto nos va a servir más adelante para realizar la división por 9 en formato Float SP. Los registros XMM quedan:

- En XMM0 la suma de la primera columna.
- En XMM1 la suma de la segunda columna.
- En XMM6 la suma de la tercera columna.
- En XMM7 la suma de la cuarta columna.
- En XMM9 la suma de la quinta columna.
- En XMM10 la suma de la sexta columna.

Ahora, tenemos que juntar los valores de las columnas correspondientes para formar los resultados de los píxel que queremos procesar. Para el primer píxel, necesitamos sumar las columnas 1, 2 y 3, para el segundo las columnas 2, 3 y 4, y así siguiendo. Utilizando la instrucción PADDD nos queda:

- En XMM0 la suma de las columnas primera, segunda y tercera.
- En XMM1 la suma de las columnas segunda, tercera y cuarta.
- En XMM6 la suma de las columnas tercera, cuarta y quinta.
- En XMM7 la suma de las columnas cuarta, quinta y sexta.

Una vez realizado esto, convertimos los valores a formato Float SP (usando la instrucción `CVTDQ2PS`) y nos queda dividir por 9 de la misma manera que se hizo en el Blur 1 (una máscara y la instrucción `DIVPS`), y convertir de vuelta a Enteros de 8 bits, utilizando las instrucciones: `CVTPS2DQ`, `PACKUSDW` y `PACKUSWB` (en ese orden).

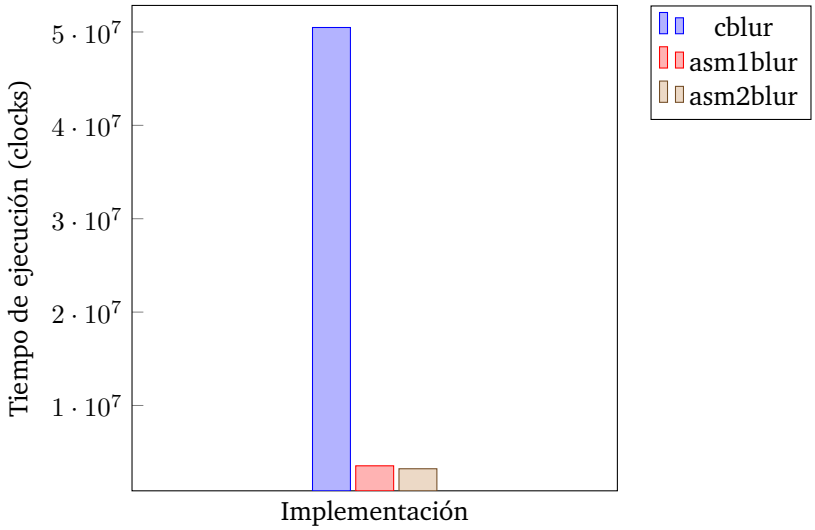
Por último, se acomoda los diferentes resultados para los píxel en el registro `XMM0` (utilizando instrucciones `PSLLDQ` y `POR`) y se baja a memoria utilizando la instrucción `MOVDQU`, ya que estamos escribiendo 4 píxeles a la vez.

Observación: similarmente a lo que pasaba en la función Blur 1, la última iteración del algoritmo requiere un caso especial, ya que si cargáramos los datos como para cualquier otro píxel estaríamos leyendo datos que se van de la posición de la imagen en memoria. Es por esto que lo que hacemos es levantar los primeros 4 píxel de forma normal en los registros `XMM0`, `XMM1` y `XMM2`, para luego copiarlos a los registros `XMM3`, `XMM4` y `XMM5` y shiftear los 8 bytes a derecha con la instrucción `PSRLDQ`. De esta forma, podemos continuar la parte principal de la iteración de forma normal, ya que nos quedaría:

- En `XMM0` los píxel 1 a 4 de la fila anterior a la actual.
- En `XMM1` los píxel 1 a 4 de la fila actual.
- En `XMM2` los píxel 1 a 4 de la fila siguiente a la actual.
- En la parte alta de `XMM3` ceros, y en la parte baja los píxel 3 a 4 de la fila anterior a la actual.
- En la parte alta de `XMM4` ceros, y en la parte baja los píxel 3 a 4 de la fila actual.
- En la parte alta de `XMM5` ceros, y en la parte baja los píxel 3 a 4 de la fila siguiente a la actual.

3.2. Análisis y Resultados

El siguiente gráfico muestra para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo.



Y la siguiente tabla resume los tiempos obtenidos para distintas imagenes (todas de 512x512).

Imagen	Tiempo C	Tiempo ASM1	Tiempo ASM2	Cambio relativo C vs ASM1		Cambio relativo ASM1 vs ASM2	
				C → ASM1	ASM1 → C	ASM1 → ASM2	ASM2 → ASM1
1	49464614	3551049	3172320	92.82 % mejor	1292.96 % peor	10.67 % mejor	11.94 % peor
2	50479122	3538585	3223099	92.99 % mejor	1326.53 % peor	8.92 % mejor	9.79 % peor
3	49667001	3522704	3189635	92.91 % mejor	1309.91 % peor	9.45 % mejor	10.44 % peor

Se observa que la implementación de mejor rendimiento es ASM2, seguida por ASM1 y luego por C. La implementación en C es en promedio apróximadamente 15 veces más lenta que ASM1, mientra que ésta última tarda en promedio un tiempo similar (pero mayor) que ASM2.

Observando los Cambios relativos obtenidos podemos ver que:

- La implementación C debería ser, en promedio, un 92 % mas rápida para tenér el mismo tiempo que ASM1.
- La implementación ASM1 debería ser, en promedio, un 1309 % mas lenta para tenér el mismo tiempo que C.
- La implementación en ASM1 debería ser, en promedio, un 9 % mas rápida para tenér el mismo tiempo que ASM2.
- La implementación en ASM2 debería ser, en promedio, un 11 % mas lenta para tenér el mismo tiempo que ASM1.

3.2.1. Hipótesis planteadas

En base a los rendimientos obtenidos planteamos las siguientes hipótesis:

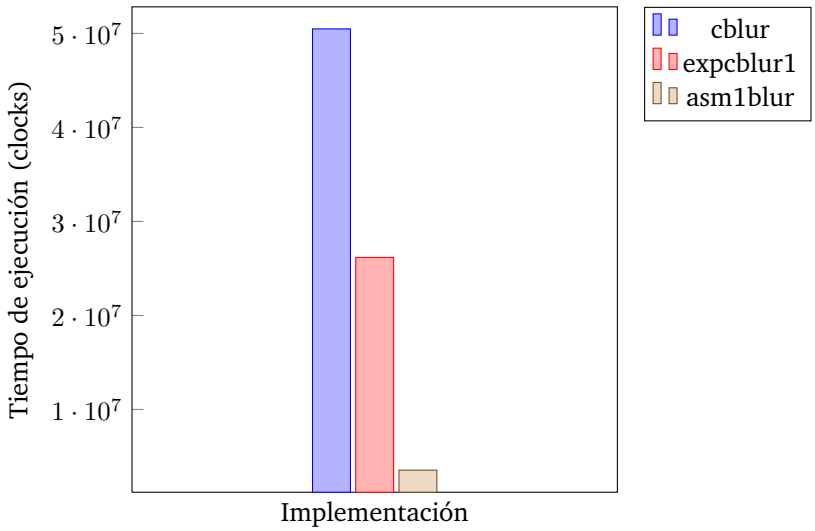
- La diferencia de velocidad entre C y ASM1 era un resultado predecible, y lo atribuimos a:
  1. La capacidad de realizar sumas en bloques en ASM1, que en C se deben realizar de a 1.
  2. La posibilidad de almacenar y manipular la información en registros `XMM` evita una gran cantidad de accesos a memoria que en C se hacen individualmente por dato.
- A su vez, la optimización de ASM2 sobre ASM1 a la hora de trabajar de a 4 píxel en vez de a 1 no parece ser tan importante como el pasaje de C a ASM1, y lo atribuimos a que:
  3. Si bien ASM2 aprovecha la vecindad de las sumas de los píxel para hacer entre 2 y 3 menos sumas por píxel que ASM1, los accesos a memoria requeridos (que son a su vez más esparsos que los de ASM1) terminan dañando el hit rate del cache, lo que provoca que se nivelen las implementaciones.
- 4. Como experimentación extra, se probó de modificar la implementación de ASM1 utilizando la instrucción `PSHUFB` para convertir los canales de Enteros de 8 bits a Enteros de 16 bits. El comportamiento esperado, por lo mencionado por la cátedra, es que los tiempos aumenten considerablemente.

3.2.2. Experimentación

Para validar las hipótesis planteadas, elaboramos diferentes experimentos:

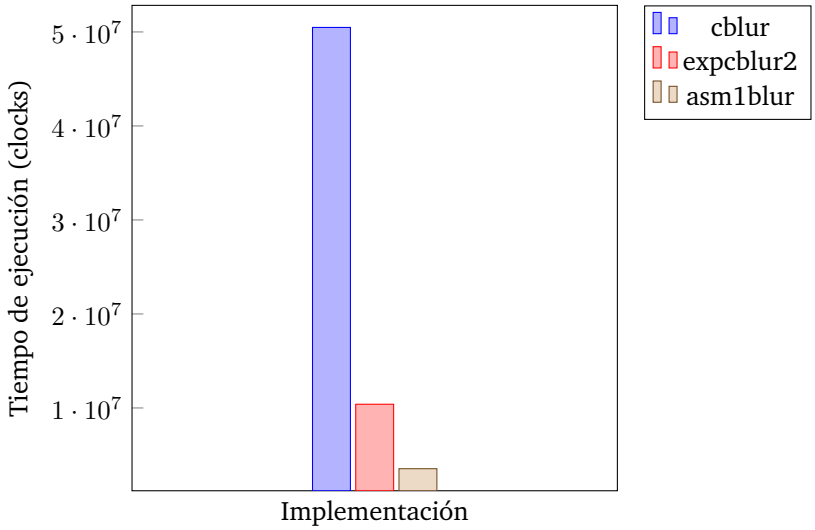
1. Realizar una implementación alternativa de C que realice 2/9 ($\sim 1/4$) de las sumas. Es decir, que en vez de sumar los 8 vecinos más el pixel en cuestion solo sume el pixel más algún vecino.
2. Realizar una implementación alternativa de C que realice 1/4 de los accesos a memoria. Es decir, que solo trabaje con un canal en vez de los 4 de ARGB. *Aclaración:* para no disminuir también las operaciones aritméticas, las operaciones sobre el canal elegido se hacen 4 veces, en vez de una sola.
3. Combinando los dos items anteriores, realizar una implementación alternativa de C que realice 2/9 ($\sim 1/4$) de las sumas y 1/4 de los accesos a memoria.
4. Realizar una implementación alternativa de ASM2 que haga la misma cantidad de accesos a memoria pero en las mismas direcciones que lo haría ASM1.
5. Realizar una implementación alternativa de ASM1 que utilice la instrucción `PSHUFB` en vez de `PUNPCKLBW` y `PUNPCKHBW`.

Experimento C vs ASM1 - 2/9 de las sumas Este experimento consta de una variación de *Blur C*, que toma solamente el valor del pixel a calcular y uno de sus vecinos al aplicar el filtro. La idea es entonces ver si la cantidad de operaciones aritméticas por cada pixel afectan notoriamente la diferencia de performance entre las implementaciones C y $ASM1$. El siguiente gráfico muestra los resultados obtenidos:

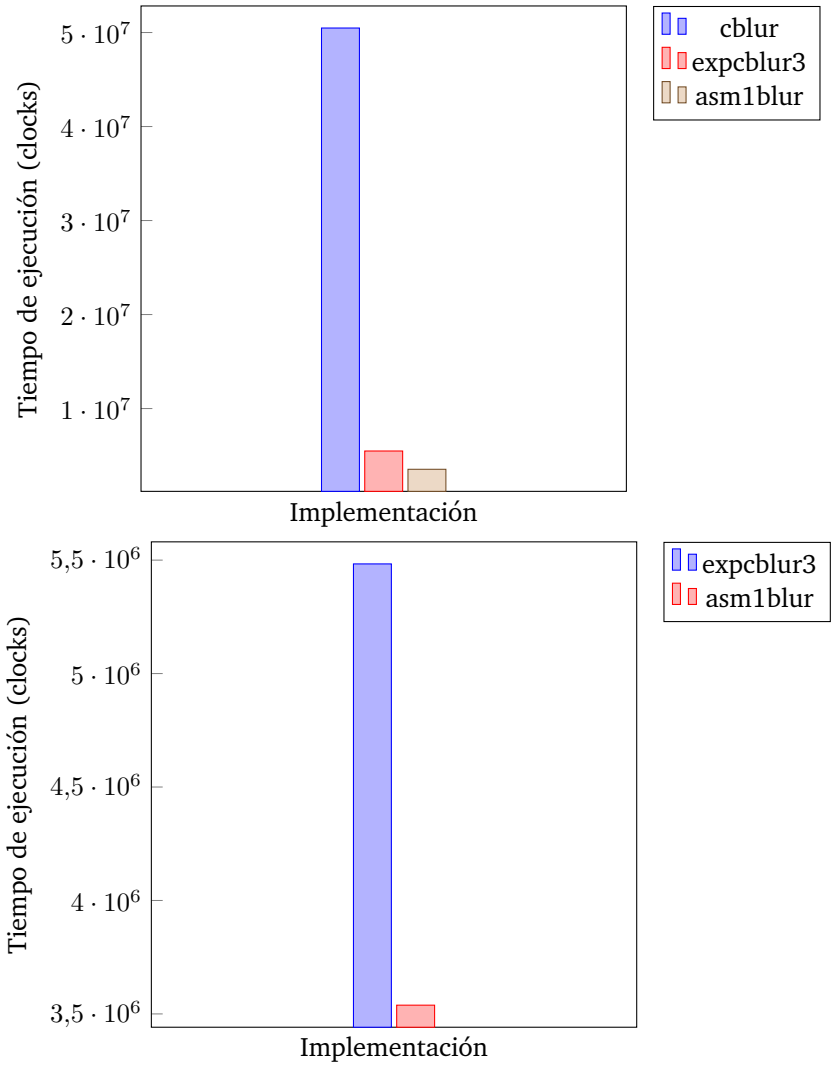


Como se puede ver, tomando solamente al pixel en cuestión y uno solo de sus vecinos, el tiempo de corrida de esta experimentación redujo considerablemente el tiempo de ejecución, pero todavía dista bastante de $ASM1$.

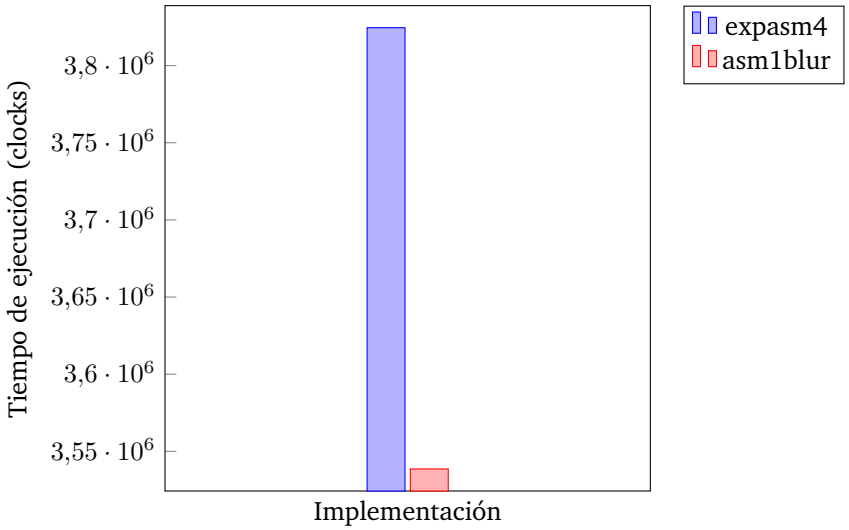
Experimento C vs ASM1 - 1/4 de accesos a memoria Trabajando con un solo canal en vez de los 4 (pero realizando la misma cantidad de operaciones aritméticas), pudimos ver que efectivamente la cantidad de accesos a memoria es un factor determinante a la hora de ejecutar el filtro de blur.



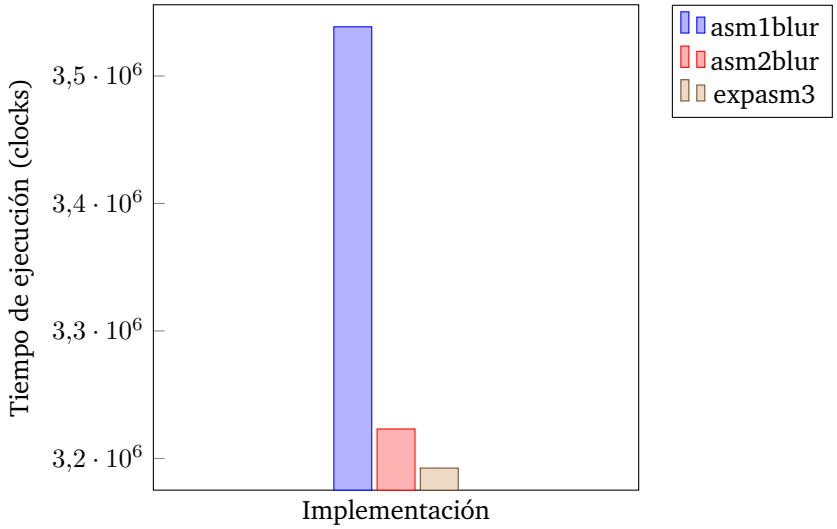
Experimento C vs ASM1 - 1/4 de accesos a memoria y 2/9 de sumas También a modo de experimentación, decidimos generar una tercera comparación entre C y $ASM1$. Esta experimentación no resiste más análisis que sostener que la aplicación de las 2 variaciones anteriores resulta en una combinación de los beneficios de aplicar operaciones aritméticas y menos accesos a memoria.



Experimento ASM1 - Shuffle de datos En la sección de análisis y resultados se menciona que utilizando instrucciones de shuffle, lentas por naturaleza, en vez de `PUNPCK(LH)BW` deberiamos obtener resultados peores que los obtenidos para ASM1. La experimentación muestra que si bien los resultados son efectivamente más pobres, la diferencia no es realmente significativa.



Experimento ASM1 vs ASM2 - Accesos a memoria y hitrate En un principio conjeturamos que la diferencia de performance no tan acentuada entre $ASM1$ y $ASM2$ se podía justificar a través de la lejanía de los accesos a memoria en $ASM2$. El siguiente gráfico muestra los resultados obtenidos:



Como se puede observar, este experimento mantiene niveles realmente similares de tiempo de corrida aunque se varíen las posiciones de memoria accedidas, intentando mejorar el hitrate. Queda entonces experimentar si es la cantidad de operaciones aritméticas la diferencia entre $ASM1$ y $ASM2$

Experimento ASM1 vs ASM2 - Operaciones aritméticas Queda ver si son las operaciones aritméticas las "culpables" de la discrepancia en el tiempo de ejecución de los algoritmos. Es por eso que implementamos una variación en $ASM2$, agregando 10 operaciones nuevas por cada iteración. Como se aprecia en el gráfico, si bien los tiempos de ejecución se asimilan, esta experimentación no logra explicar enteramente la mejora entre $ASM1$ y $ASM2$ que se termina de entender como un conjunto de mejoras respecto a las experimentaciones realizadas.

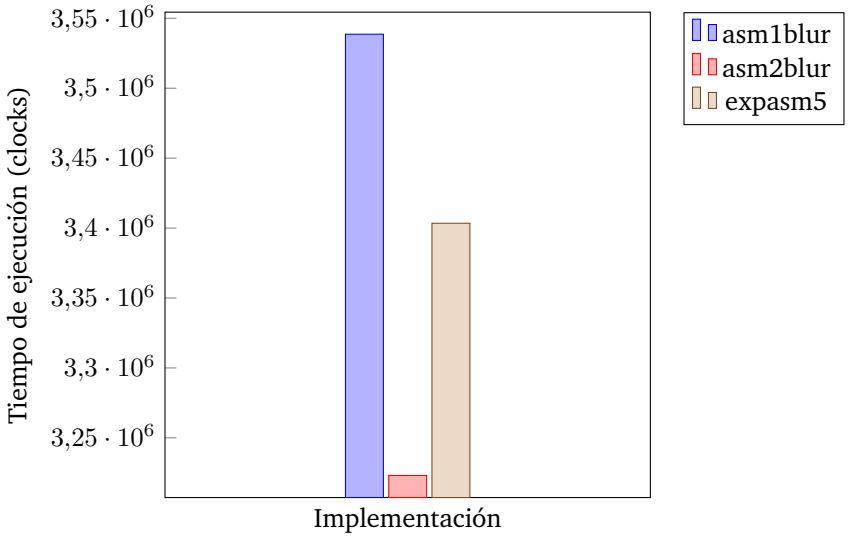
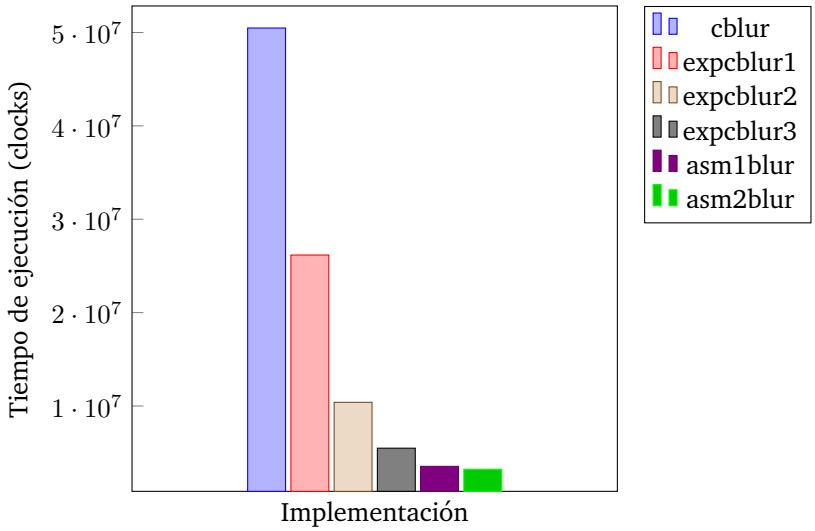


Gráfico de todas las experimentaciones


Tiempo de ejecución (clocks)
$5 \cdot 10^7$
$4 \cdot 10^7$
$3 \cdot 10^7$
$2 \cdot 10^7$
$1 \cdot 10^7$
Implementación
cblur
expcblur1
expcblur2
expcblur3
asm1blur
asm2blur

4. Ejercicio 2 - Merge

4.1. Desarrollo

El filtro Merge genera una nueva imagen a partir de 2 imágenes origen, generando por cada píxel de las imágenes originales uno nuevo que posee un porcentaje de la primera imagen y otro de la segunda, donde dicho porcentaje es un parámetro del filtro. Si por ejemplo el parámetro es 0.1, cada componente RGB de la imagen resultante posee un 10 % de la componente de la primera imagen y un 90 % de la segunda.

El efecto resultante refleja la imagen cuyo componente es más fuerte con la imagen r̈ecesivaẗdifuminada.

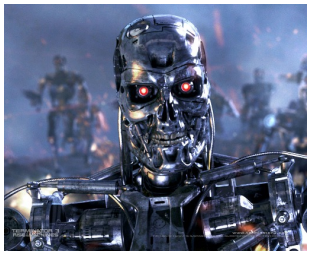
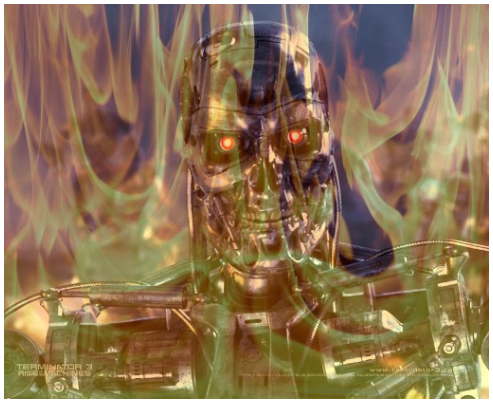


Imagen base 1

Imagen base 2



Resultado obtenido al aplicarle el filtro Merge (porcentaje de mezcla 0.42)

4.1.1. Merge 1

La primera implementación del filtro Merge se enfoca en entender como generar varios píxel de la imagen resultante y construir este proceso de la manera más sencilla posible (las cuentas necesarias para los pixeles se realizan en formato Float Single Precision). En ambas implementaciones, el algoritmo va a procesar de a 4 píxel a la vez.

Procedimiento general Lo primero que hace el algoritmo es empaquetar el *valor* Float SP pasado por parámetro (porcentaje de mezcla de las imágenes) en el registro `XMM0` utilizando las instrucciones `MOVDQU`, `PSLLDQ` y `ADDPS` (esta última se podría utilizar de la misma forma por la instrucción `POR`). De esta forma, nos queda en `XMM0` 4 veces el *valor* en formato Float SP.

Luego, preparamos en el registro `XMM15` 4 Floats SP empaquetados con *1-valor*. Para esto, cargamos en `XMM15` una mascara con 4 unos en formato Float SP, y le restamos el registro `XMM0` utilizando la instrucción `SUBPS`.

Además, calculamos en `RAX` la cantidad de bytes a procesar ($width * height * 4$) que nos va a servir para determinar cuando llegamos al final de la imagen.

Iteración El algoritmo del filtro Merge va a procesar de a 4 píxel a la vez en cada iteración. Para esto, cargamos 4 píxel de la imagen 1 en `XMM1` y 4 píxel de la imagen 2 en `XMM2` utilizando la instrucción `MOVDQU`.

Luego, convertimos los valores a Enteros de 16 bits (1 Word) utilizando las instrucciones `PUNPCKLBW` y `PUNPCKHBW`. La disposición de los valores queda (todos en formato Enteros 16 bits):

- En `XMM1` los píxel 1 y 2 de imagen 1.
- En `XMM3` los píxel 3 y 4 de la imagen 1.
- En `XMM2` los píxel 1 y 2 de imagen 2.
- En `XMM4` los píxel 3 y 4 de la imagen 2.

Ahora, debemos realizar el cálculo de los 4 píxeles nuevos. Vamos a mostrar como sería para el primer píxel, ya que los otros se construyen de forma análoga:

1. Copio `XMM1` a `XMM5` y `XMM2` a `XMM6` utilizando la instrucción `MOVDQU`.
2. Convierto la parte baja de `XMM5` y `XMM6` a Enteros de 32 bit en los mismos registros (para el segundo y cuarto píxel tendríamos que convertir la parte alta de su respectivo registro).
3. Convierto `XMM5` y `XMM6` a formato Floats SP en los mismos registros, utilizando la instrucción `CVTDQ2PS`.
4. Multiplico `XMM5` por `XMM0` y dejo el resultado en `XMM5`, utilizando la instrucción `MULPS`. Esto significa multiplicar cada canal del primer píxel de la imagen 1 por el $valor$ Float SP pasado por parámetro.
5. Multiplico `XMM6` por `XMM15` y dejo el resultado en `XMM6`, utilizando la instrucción `MULPS`. Esto significa multiplicar cada canal del primer píxel de la imagen 2 por $1 - valor$.
6. Luego, sumamos los canales del píxel en `XMM6` a los del píxel en `XMM5`, con la instrucción `ADDPS`.

Al final, tenemos en `XMM5`, `XMM7`, `XMM9` y `XMM11` los 4 píxel resultante en formato Float SP.

Lo siguiente que hacemos es convertirlos a formato Enteros 8 bits, utilizando las instrucciones: `CVTPS2DQ`, `PACKUSDW` y `PACKUSWB` (en ese orden).

Por último, se acomoda los diferentes resultados para los píxel en el registro `XMM5` (utilizando instrucciones `PSLLDQ` y `POR`) y se baja a memoria utilizando la instrucción `MOVDQU`, ya que estamos escribiendo 4 píxeles a la vez.

4.1.2. Merge 2

La segunda implementación del filtro Merge varía de la primera al reemplazar la aritmética que antes se realizaba con valores en formato Float SP por aritmética de enteros. Como el valor dado por parámetro para el porcentaje de mezcla es un Float, necesitamos idear un método para perder la menor precisión posible a la hora de hacer las cuentas:

Si tenemos un canal $R1$ y otro $R2$ entonces queremos calcular $R1 * v + R2 * (1 - v)$, pero podemos reescribir esa expresión como: $(R1 * v + R2 * (1 - v)) * (2^k/2^k) = (R1 * v * 2^k + R2 * (1 - v) * 2^k)/2^k$. Lo interesante es que calculando $v * 2^k$ y $(1 - v) * 2^k$ con un k suficientemente grande se obtiene un rango aceptable para la definición en enteros, reduciendo el error cometido. En nuestro caso consideramos que $k = 8$ es un valor ideal para generar buenos resultados, ya que el rango de valores que podemos expresar es de 0 a $2^8 - 1$ que es exactamente el rango que tiene cada canal en la representación RGB/RGBA.

Procedimiento general Análogamente a la implementación del Merge 1, debemos preparar dos registros especiales que contengan los valores necesarios empaquetados para después realizar las cuentas de a 4 píxel a la vez.

En primer lugar, calculamos 2^k cargando una máscara que tiene un uno en formato Entero 32 bits y lo shifteamos a izquierda k bits ($k = 8$). Luego, calculamos $v * 2^k$. Para esto, convertimos el 2^k a formato Float SP con la instrucción `CVTSI2SS` y lo multiplicamos por el $valor$ con la instrucción `MULSS`. A continuación, convertimos este valor obtenido a formato Entero de 32 bits con la instrucción `CVTSS2SI` y lo dejamos empaquetado en el registro `XMM4` con `MOVDQU`, `PSLLDQ` y `PADDD`. Idénticamente, dejamos empaquetado en `XMM3` el 2^k en formato Entero de 32 bits con las instrucciones `MOVDQU`, `PSLLDQ` y `PADDD`.

Ahora, preparamos en XMM15 $(1-v)*2^k$ empaquetado en formato Entero de 32 bits haciendo la resta empaquetada entre XMM3 y XMM4 con la instrucción PSUBD.

Antes de comenzar el ciclo, copiamos a XMM14 el empaquetado de $v*2^k$ que estaba en XMM4 y cargamos en XMM13 un 8 (el k elegido) en formato Entero 32 bits para luego usarlo para shiftear los resultados.

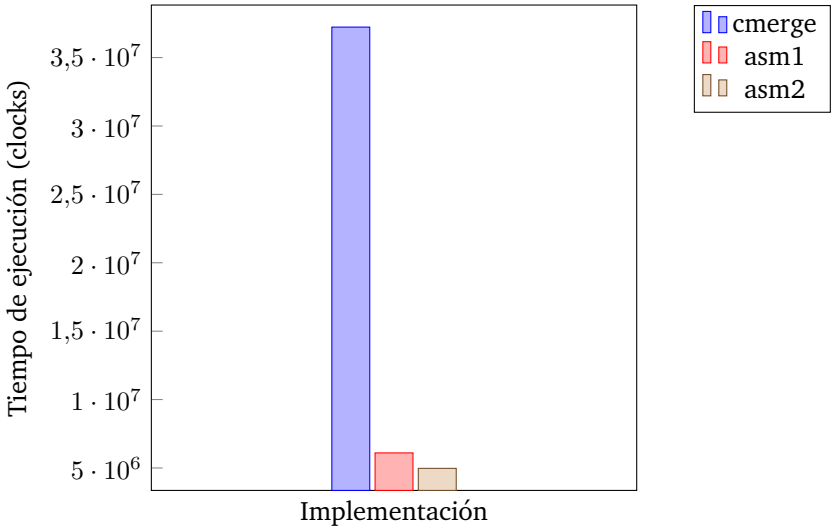
Iteración La iteración del Merge 2 es idéntica hasta la parte del calculo del píxel, en la cual hacemos calculamos 4 píxel de la siguiente forma (mostramos solo el primero ya que los otros se construyen de forma análoga):

1. Copio XMM1 a XMM5 y XMM2 a XMM6 utilizando la instrucción MOVDQU.
2. Convierto la parte baja de XMM5 y XMM6 a Enteros de 32 bit en los mismos registros (para el segundo y cuarto píxel tendríamos que convertir la parte alta de su respectivo registro).
3. Multiplico XMM5 por XMM14 y dejo el resultado en XMM5, utilizando la instrucción PMULLD. Esto significa multiplicar cada canal del primer píxel de la imagen 1 por el $v*2^k$ y guardar la parte baja de esa operación en el *DST*. No nos molesta solo guardar la parte baja ya que sabemos que original mente teníamos canales de 8 bits y el $v*2^k$ también es de 8 bits, así que a lo sumo nos puede llegar a quedar algo de 16 bits (a lo sumo se duplica la longitud).
4. Multiplico XMM6 por XMM15 y dejo el resultado en XMM6, utilizando la instrucción PMULLD. Esto significa multiplicar cada canal del primer píxel de la imagen 2 por el $(1-v)*2^k$ y guardar la parte baja de esa operación en el *DST*.
5. Sumamos los canales del píxel en XMM6 a los del píxel en XMM5, con la instrucción PADDD.
6. Shifteamos a derecha cada *DWORD* en XMM5 k bits (almacenado en XMM13) con la instrucción PSRLD. Esto es equivalente a dividir cada Entero de 32 bits en XMM5 por 2^k.

A partir de acá realizamos exactamente lo mismo que realizábamos para el Merge 1.

4.2. Análisis y Resultados

El siguiente gráfico muestra para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo.



Y la siguiente tabla resume los tiempos obtenidos para distintas imagenes:

Imagen	Tiempo C	Tiempo ASM1	Tiempo ASM2	Cambio relativo C vs ASM1		Cambio relativo ASM1 vs ASM2	
				C → ASM1	ASM1 → C	ASM1 → ASM2	ASM2 → ASM1
1	37247796	6101585	4973417	83.62 % mejor	510.46 % peor	18.49 % mejor	22.68 % peor
1	37227535	6101773	4972185	83.61 % mejor	510.11 % peor	18.51 % mejor	22.72 % peor
1	37216738	6089785	4980718	83.64 % mejor	511.13 % peor	18.21 % mejor	22.27 % peor

La implementación más lenta es la de C, siendo esta aproximadamente 6 veces más lenta que ASM1. A su vez ASM1 es un 20 % más lenta que ASM2. Esto es consistente tanto para imágenes pequeñas como grandes.

Observando los Cambios relativos obtenidos podemos ver que:

- La implementación C debería ser, en promedio, un 83 % mas rápida para tenér el mismo tiempo que ASM1.
- La implementación ASM1 debería ser, en promedio, un 510 % mas lenta para tenér el mismo tiempo que C.
- La implementación en ASM1 debería ser, en promedio, un 18 % mas rápida para tenér el mismo tiempo que ASM2.
- La implementación en ASM2 debería ser, en promedio, un 22 % mas lenta para tenér el mismo tiempo que ASM1.

Ambas implementaciones (tanto C como ASM1) son muy simples y si miramos solamente las operaciones aritméticas de suma y producto, deberíamos ver una diferencia de 4x en favor de ASM teniendo en cuenta que con instrucciones de SIMD se pueden realizar sumas y productos de a 1 píxel a la vez. En cambio en C se puede realizar una suma o producto de una sola componente RGBA a la vez. Tanto C como ASM1 deben convertir los valores de uint8 a float, con lo cual la diferencia de tiempo entre el 4x esperado inicialmente y el 6x registrado lo atribuimos a la disminución en accesos a memoria. Para acceder a 4 píxel en ASM1 alcanza con almacenar en 1 registro XMM, en cambio en C se realiza un acceso por cada componente de cada píxel. Esto mejora sustancialmente los tiempos en ASM1.

En ASM2 notamos una mejora consistente de un 18 % respecto a ASM1. Esta mejora interesante está dada por el manejo de los datos en enteros en vez del uso de aritmética de punto flotante. Como se comenta en la sección de desarrollo, la idea principal es realizar los cálculos en enteros, llevando los números a órdenes que no alteren demasiado los resultados por el uso de enteros en vez de floats y luego dividir por el mismo factor para devolver los números al orden inicial (2^k).

Este 18 % de mejora demuestra el costo de convertir valores enteros a punto flotante y viceversa. En compensación, el código de ASM1 es más simple de entender y preciso, ya que no posee el error variable de usar 2^k como factor de multiplicidad.

Como implementación alternativa pensamos como modificar las cuentas para realizar una menor cantidad de operaciones de punto flotante, en eso se basa ASM3. Este filtro es 24 % más lento que ASM2, justificado simplemente por una mayor cantidad de operaciones aritméticas realizadas. ASM3 resultó más lento que ASM1 y no posee mejoras a la hora de leer el código, con lo cual sería descartado para una implementación futura del filtro Merge.

4.2.1. Hipótesis planteadas

En base a los rendimientos obtenidos planteamos las siguientes hipótesis:

- La implementación ASM1 es bastante más eficiente que la de C. Nuestro análisis previo indica que esto se debe a:
  1. Cantidad de accesos a memoria
  2. Cantidad de operaciones aritméticas realizadas
- También observamos una mejora en la eficiencia de ASM2 respecto a los tiempos de ASM1. Nuestra hipótesis acerca de la mejora se basa en:

1. El costo de la conversión de entero a punto flotante y viceversa
2. El uso de instrucciones de valores enteros

4.2.2. Experimentación

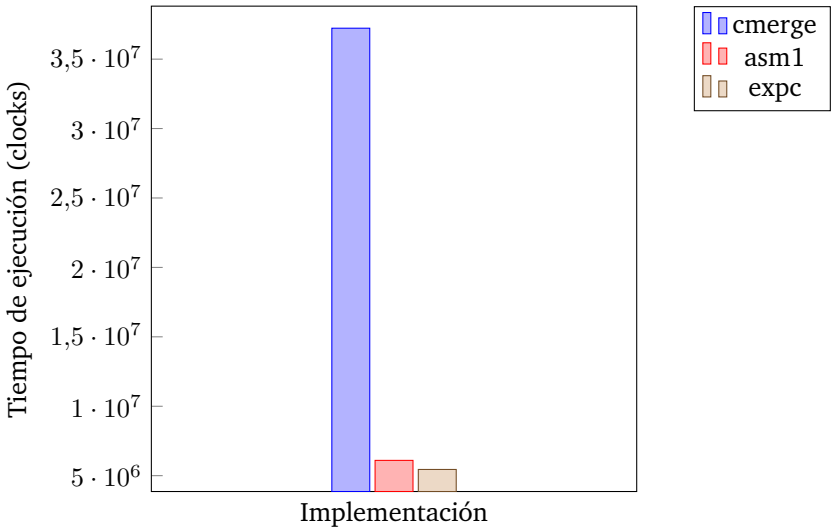
Para validar las hipótesis planteadas, elaboramos diferentes experimentos:

1. Realizar una implementación de C que solo hace merge de una porción de la imágen. La idea detrás de este experimento es observar si realmente es la cantidad de accesos a memoria y operaciones aritméticas la diferencia de tiempo entre C y ASM1.

2. Realizar una implementación de ASM1 que no convierte los valores a punto flotante. Esto nos va a dar una noción de la pérdida de performance relacionada con la conversión de y a punto flotante de los números enteros.

Estas implementaciones propuestas no buscan correctitud en los resultados sino validar las hipótesis planteadas, con lo cual los realizamos sabiendo que los filtros resultantes no serán correctos.

Experimento Merge C vs ASM



Este experimento busca explicar la diferencia de rendimiento entre el Merge en C y ASM1. El planteo del experimento consiste en aplicar el filtro en una porción de la imágen para comprobar si realmente la cantidad de iteraciones es el factor determinante para la mejora de performance en ASM1. Durante la experimentación inicial, el procedimiento realizado en ASM corría entre $1/7$ y $1/6$ del tiempo de corrida de C. Aplicando en este experimento el filtro para $1/7$ de los pixeles de la imágen vemos como los tiempos de ejecución se asemejan. Como la única variable modificada es la cantidad de iteraciones, efectivamente es este el factor determinante en el tiempo de corrida del algoritmo.

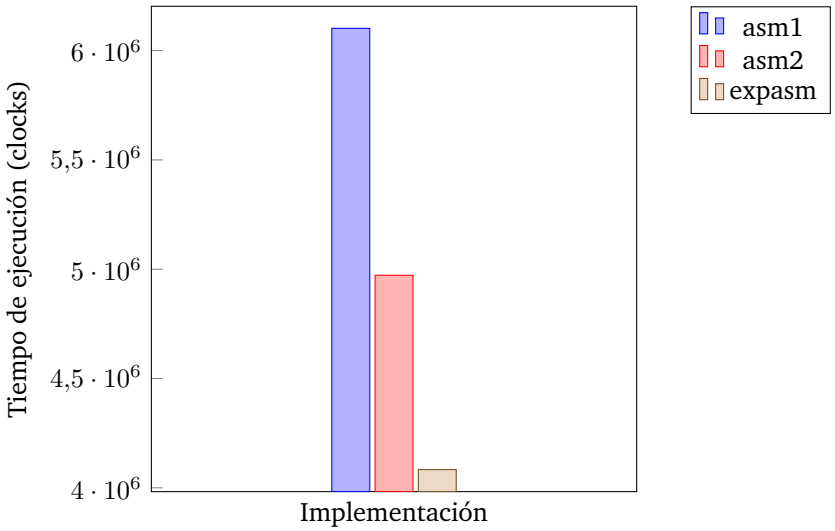
Esto muestra la importancia del uso de intrucciones SIMD para la aplicación del filtro. Solamente usando SIMD accedemos a un $X7$ de performance que es vital para muchas aplicaciones, sabiendo que perdemos expresividad por utilizar un lenguaje de más bajo nivel.

Experimento Merge conversión de tipos de dato La experimetanción para validar nuestra hipótesis de que la diferencia de performance entre ASM1 y ASM2 se debe a las instrucciones de conversion de datos constó de 3 etapas para lograr un experimento final satisfactorio.

El primer experimento toma la implementación de ASM1, omite las instrucciones de conversión y utiliza aritmética de enteros en vez de punto flotante. Si bien los resultados revierten la tendencia de la experimentación inicial, haciendo más rápida esta implementación a la de ASM2, carece de rigor ya que no se puede discernir si la mejora en performance viene de la omisión de las instrucciones de conversión o de la utilización de instrucciones de aritmética de enteros.

El segundo experimento propone entonces simplemente variar, a partir de ASM1, eliminando las conversiones de enteros a punto flotante, dejando así las instrucciones de aritmética en punto flotante. El problema que encontramos es que al no realizar las conversiones, los valores contenidos en las operaciones contienen "basura". Esto lleva a que las operaciones se encuentren con muchos casos edge de "NaN (Not a Number)", haciendo notoriamente más lenta la operatoria.

El tercer experimento, que elegimos tomar como el válido para probar nuestra hipótesis consiste en tomar la implementación de ASM1, eliminar las conversiones y, antes de empezar a operar en punto flotante, realizar $pxor$ por sobre los registros contenedores de los datos a procesar, llevandolos a un arreglo de $0's$ y así removiendo el problema de lentitud proveniente del contenido basura en los registros.



Como se puede ver en las gráficas, este experimento muestra que es la conversión de entero a punto flotante y viceversa el motivo principal de la diferencia de performance entre ASM1 y ASM2.

5. Ejercicio 3 - HSL

5.1. Desarrollo

La idea detrás de este filtro es llevar la representación de los píxel de RGB a HSL[3]. Las componentes HSL (Matiz, Saturación y Luminosidad) tienen más que ver con los aspectos a retocar de una imagen a ojos de un humano que los componentes de Rojo, Azul y Verde. Una vez transportados los píxel a esta representación se manipulan los valores de HSL sumándole los valores pasados por parámetro: HH, SS y LL. Los últimos dos se manipulan con saturación, mientras que el primero no (si $H + HH > 360$ entonces se el resta 360). Luego, se retorna a la representación RGB para depositar los píxel en la imagen resultante.

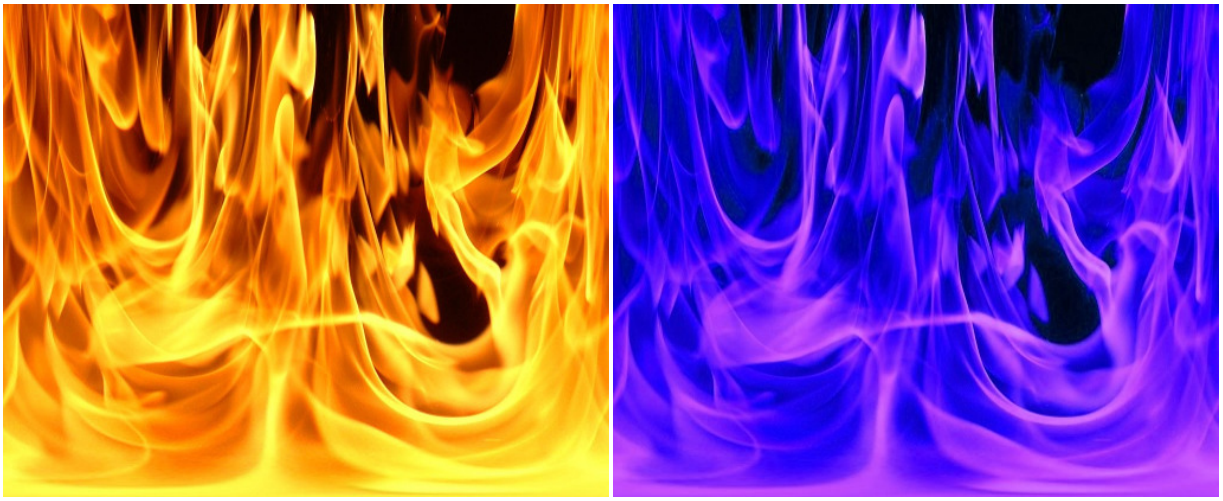


Imagen original

Resultado obtenido al aplicarle el filtro HSL con los parámetros H:220, S:0 y L:0

5.1.1. HSL 1

En la primera implementación del HSL se utiliza las funciones auxiliares en C *rgbTOhsl* y *hslTOrgb* dadas por la cátedra, y su mayor dificultad consiste en realizar los cálculos para sumar los parámetros HH, SS y LL a los píxeles convertidos a HSL (etapa *Suma*).

Procedimiento general Antes de comenzar el ciclo de iteraciones se prepara un vector *hsl_params_dato* de 128 bits (almacenado en la pila) conteniendo en 4 números en formato Float SP la información de HSL utilizando las instrucciones `MOVDQU`, `PSLLDQ` y `POR`. El resultado obtenido queda (empezando por la parte más baja):

- 0.0
- H
- S
- L

Luego, calculamos en `R15` la cantidad de bytes a procesar ($width * height * 4$) y le sumamos la posición de memoria donde empieza la imagen, así nos va a servir para determinar cuando llegamos al final de la ella.

[3]Referencia `http://es.wikipedia.org/wiki/Modelo_de_color_HSL`

Iteración Por cada píxel se llama, al comienzo de la iteración, a la función de conversión de RGB a HSL (`CALL` *rgbTOhsl*). Los parámetros que se le pasan son: en `RDI` un puntero a la dirección del píxel actual y en `RSI` un puntero a un vector *hsl_suma_dato* de 128 bits (almacenado en la pila). Esto produce un vector con 4 números de punto flotantes de 32 bits.

Luego se suman en formato Float SP los valores del píxel convertido a HSL con los valores pasados por parámetro. Para esto, se copia la información en *hsl_params_dato* y *hsl_suma_dato* a los registros `XMM1` y `XMM0` respectivamente, para luego realizar la suma con la instrucción `ADDPS`.

A continuación, se copia la suma obtenida en `XMM0` al vector *hsl_suma_dato* de 128 bits, que va a contener el resultado final en la pila. Además, cargamos en `XMM3` una mascara que tiene un 360 en formato Float SP en la primera DWORD (contando desde la parte baja).

Ahora pasamos a hacer los chequeos para que los resultados de la suma sean correctos. Necesitamos que los resultados de las componentes L (luminosity) y S (saturation) saturen entre 0 y 1, mientras que la componente H (hue) puede pasarse, pero siempre entre 0 y 360 (menor estricto que 360). Para esto, tenemos dos partes de código por separado: una que verifica los limites superiores y otra que verifica los limites inferiores.

Para verificar los **limites superiores**:

- Copiamos en `XMM1` una mascara en formato Float SP que tiene los valores máximos permitidos para cada canal: 360, 1 y 1.
- Con la instrucción `CMPLTPS` hacemos la comparación por menor estricto entre los valores en `XMM0` y los valores en `XMM1`.
- Ahora, copiamos en `EDI` el resultado de $H < 360$, en `ESI` el resultado de $S < 1$ y en `EDX` el resultado de $L < 1$ con la instrucción `MOVD`.
- Comparamos `EDI` con 0 (FALSE) (con la instrucción `CMP`). Si no es igual, pasamos a evaluar `ESI` (el salto se realiza con la instrucción `JNE`). Si es igual, utilizamos la instrucción `SUBSS` para dejar en `XMM0` los valores de antes pero con el Hue restado en 360 (la resta se hizo contra el registro `XMM3`) y se pisa el valor del hue en el vector *hsl_suma_dato* con el contenido en `XMM0` con la instrucción `MOVSS`.
- Comparamos `ESI` con 0 (FALSE) (con la instrucción `CMP`). Si no es igual, pasamos a evaluar `EDX` (el salto se realiza con la instrucción `JNE`). Si es igual, simplemente movemos el máximo para S desde la máscara de valores máximos a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción `MOV`.
- Realizamos lo mismo para `EDX` que para `ESI`, excepto que en el caso de que no sea igual a FALSE, pasamos a chequear los limites inferiores.

Para verificar los **limites inferiores** se realiza de forma análoga a la anterior:

- Ponemos en `XMM1` una mascara en formato Float SP que tiene los valores mínimos permitidos para cada canal: 0, 0 y 0. Esto lo hacemos con la instrucción `PXOR XMM1, XMM1` ya que rellena todos los bits con ceros, lo cual representa a 4 floats SP con valor 0.
- Con la instrucción `CMPNLTPS` hacemos la comparación por menor estricto entre los valores en `XMM0` y los valores en `XMM1`.
- Ahora, copiamos en `EDI` el resultado de $H \geq 0$, en `ESI` el resultado de $S \geq 0$ y en `EDX` el resultado de $L \geq 0$ con la instrucción `MOVD`.
- Comparamos `EDI` con 0 (FALSE) (con la instrucción `CMP`). Si no es igual, pasamos a evaluar `ESI` (el salto se realiza con la instrucción `JNE`). Si es igual, utilizamos la instrucción `ADDSS` para dejar en `XMM0` los valores de antes pero con el Hue sumado en 360 (la suma se hizo contra el registro `XMM3`) y se mueve el nuevo Hue a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción `MOVSS`.

- Comparamos ESI con 0 (FALSE) (con la instrucción CMP). Si no es igual, pasamos a evaluar EDX (el salto se realiza con la instrucción JNE). Si es igual, simplemente movemos el mínimo para S desde la máscara de valores mínimos a su posición correspondiente en el vector *hsl_suma_dato* con la instrucción MOV.
- Realizamos lo mismo para EDX que para ESI, excepto que en el caso de que no sea igual a FALSE, pasamos a la parte final del ciclo.

Para terminar, llamamos a la función de conversión de HSL a RGB (CALL *hslTOrgb*). Los parámetros que se le pasan son: en RDI un puntero al vector vector *hsl_suma_dato* y en RSI copiamos el valor de RBX, es decir, la posición de memoria de donde sacamos el pixel, así sobrescribiendo ese valor por el calculado en nuestro filtro.

5.1.2. HSL 2

En la segunda implementación del HSL se agregan los algoritmos *rgbTOhsl* y *hslTOrgb* en lenguaje ensamblador.

Procedimiento general e Iteración El procedimiento general antes de la iteración es igual a HSL1. Sin embargo, para aprovechar ciertas facilidades de Assembler cambiamos el ciclo principal y el chequeo de cotas con el fin de ahorrarnos accesos a memoria: $rgbTOhsl$ ahora recibe en RDI un puntero a un píxel pero el valor respuesta lo devuelve en el registro XMM0 (esto lo podemos hacer ya que esta todo en Assembler y podemos garantizar la sanidad de los registros). A su vez, en la función $hslTOrgb$ el píxel en formato HSL viene en el registro XMM5 y la posición de memoria donde guardar la respuesta en el registro RDI (de nuevo, esto lo podemos hacer ya que esta todo el código en Assembler). Además vamos a mantener los parámetros de entrada 'hh', 'ss' y 'll' en el registro XMM15 para despues ahorrar accesos a memoria (esto lo podemos hacer ya que esta todo en Assembler y me aseguro de que nada lo afecte en ninguna función).

El nuevo ciclo principal realiza lo siguiente por cada pixel:

- Llama a $rgbTOhsl$ poniendo RDI un puntero al píxel actual (mantenido en RBX) y recibe el valor respuesta (pixel en formato HSL) en el registro XMM0.
- Suma a XMM0 los parametros de entrada del filtro (contenidos en el registro XMM15, ya que me aseguro en las demás funciones de no modificarlo) a cada una de sus componentes con la instrucción ADDPS.
- Copia el contenido de XMM0 a XMM5 (en XMM5 vamos a mantener la respuesta y XMM0 lo vamos a usar para mantener el valor original, que vamos a necesitar después).
- Pone todos los bits del registro XMM1 en 0 (asi representando cuatro Floats SP con valor 0.0).
- Hace MAXPS de XMM5 con XMM1, asi asegurándonos que ninguna de las componentes resultantes sean menores que 0.
- Asigna al registro XMM1 los valores $|1{,}0|1{,}0|360{,}0|1{,}0|$ mediante la instrucción `MOVUPS XMM1, [hsl_max_dato]` (donde la posición hsl_max_dato contiene el valor $|1{,}0|360{,}0|1{,}0|1{,}0|$).
- Luego, hacemos MINPS de XMM5 con XMM1, asi asegurándonos que todas las componentes resultantes sean a lo sumo el máximo en cada caso.
- Para chequear la cota superior del valor 'HUE' del pixel: con la instrucción CMPLTSS nos fijamos si el valor de hue resultante es menor a 360.0. Si esto es falso, le restamos 360.0 a esta componente en el registro XMM6 con la instrucción SUBSS e insertamos el nuevo valor en XMM5 con la instrucción `INSERTPS XMM5, XMM6, 0x10` (0x10 es el parámetro que indica como insertar, en este caso quiere decir que insertamos el valor del cuarto Float SP de XMM6 en el tercer Float SP de XMM5 que es donde tiene que estar el Hue del pixel).

- Para chequear la cota inferior del valor 'HUE' del pixel: con la instrucción `CMPNLTSS` nos fijamos si el valor de hue resultante es mayor o igual a 0.0. Si esto es falso, le sumamos 360.0 a esta componente en el registro `XMM6` con la instrucción `ADDSS` e insertamos el nuevo valor en `XMM5` con la instrucción `INSERTPS XMM5, XMM6, 0x10`.
- Finalmente, teniendo el valor del pixel HSL que queriamos en el registro `XMM5` ponemos en `RDI` el valor de `RBX` (posición de memoria del pixel actual en la imagen) y llamamos a $hslTOrgb$, que recibe como parámetros la posición donde guardar el pixel en `RDI` y el pixel HSL a convertir en `XMM5`.
- Después le sumamos a `RBX` el tamaño de un pixel para ir al próximo y saltamos nuevamente al principio del ciclo, asi iterando hasta que termina la imagen.

RGB a HSL La función $rgbTOhsl$ recibe en `RDI` un puntero a un píxel en forma de Enteros sin signo de 8 bits (píxel de 32 bits). Lo primero que hacemos es cargar los valores apuntados por `RDI` en `XMM0` con la instrucción `MOVD`, para luego convertir la representación de Enteros de 8 bits a Enteros de 32 bits con las instrucciones `PUNPCKLBW` y `PUNPCKLWD` en el mismo registro `XMM0`.

Una vez que tenemos esto, calculamos el máximo y mínimo de los componentes RGB utilizando las instrucciones `PMAXUD`, `PMINUD` y `PSLLDQ` lo que nos deja en los 32 bits más altos de `XMM1` $cmax = max(R, G, B)$ y en los 32 bits más altos de `XMM2` $cmin = min(R, G, B)$. Utilizando la instrucción `PSUBD` dejamos en los 32 bits más altos de `XMM3` $d = max(R, G, B) - min(R, G, B)$. Lo movemos a los 32 bits más bajos de `XMM3` con la instrucción `PSRLDQ` y lo copiamos en `ECX` usando `MOVD`.

Antes de empezar con los cálculos de cada componente de HSL, dejamos en los 32 bits más bajos de `XMM4` $cmax + cmin$ y en los 32 bits más bajos de `XMM1` $cmax$ utilizando las instrucciones `PADDD` y `PSRLDQ`. Además, limpiamos el registro `XMM14` con la instrucción `PXOR` para ir dejando los resultados de cada canal.

Cálculo de L

- Convertimos el valor en `XMM4` ($cmax + cmin$) a formato Float SP con la instrucción `CVTDQ2PS`.
- Cargamos en el registro `XMM13` una mascara que en los 32 bits más bajos un 510 en formato Float SP y lo demás todos unos en formato Float SP.
- Dividimos `XMM4` por `XMM13` con la instrucción `DIVPS`.
- Copiamos el valor obtenido a los 32 bits más altos de `XMM14` con las instrucciones `MOVDQU` y `PSLLDQ`.

Cálculo de S

- Comparamos `ECX` (d) con 0 (instrucción `CMP`). Si es igual, no hacemos nada y pasamos a calcular H (salto con instrucción `JE`). Caso contrario, seguimos.
- Multiplicamos el valor obtenido para L por 2 y le restamos 1. Esto lo hacemos cargando en `XMM5` una mascara que tiene en los 32 bits más bajos un 1 en formato Float SP y en el resto ceros, luego se lo suma contra si mismo con `ADDPS` para obtener el 2. Con la instrucción `MULPS` multiplicamos el 2 por L y lo dejamos en `XMM5`. Luego, volvemos a cargar la máscara con el 1 en los 32 bits más bajos y se la restamos a `XMM5` usando `SUBPS`. Este resultado parcial queda en `XMM4`.
- Lo siguiente es aplicarle la función $fabs$ a el valor en `XMM4`. Para esto, cargamos una máscara que tiene todos unos excepto en el primer bit de cada DWORD ($0x7fffffff$), y se la aplicamos con la instrucción `PAND` a `XMM4`, con lo que efectivamente estamos seteando el bit de signo de cada Float en `XMM4` a cero, convirtiéndolos a positivos en caso de que no lo fueran.
- Le restamos a 1 el resultado de $fabs$ de la misma forma que antes le restábamos 1 a $2 * L$. El resultado parcial queda en `XMM5`.

- A continuación, dividimos d por el valor en XMM5. Primero, cargamos una máscara que tiene todos unos excepto en los 32 bits más bajos y le hacemos un POR con XMM5. Segundo, convertimos el valor d que estaba en XMM3 en Entero 32 bits a XMM4 en formato Float SP con la instrucción CVTDQ2PS. Y luego dividimos XMM4 por XMM5 con la instrucción DIVPS.
- Por último, cargamos en XMM5 una máscara que tiene todos unos excepto en los 32 bits más bajos, que tiene el valor $255{,}001f$ (todos en formato Float SP), y dividimos XMM4 por XMM5 con la instrucción DIVPS, lo que nos deja efectivamente en XMM4 el valor $d/(1 - fabs(2 * L - 1))/255{,}0001$.
- Copiamos el valor obtenido en los 32 bits siguientes a L en XMM14 con las instrucciones PSLLDQ y ADDPS.

Cálculo de H Para este calculo de H nos dimos cuenta que las cuentas necesarias se pueden llevar a cabo en paralelo y simplemente elegir el resultado final una vez realizadas las cuentas, comparando el valor de $cmax$ con los distintos canales RGB.

- Comparamos ECX (d) con 0 (instrucción CMP). Si es igual, no hacemos nada y pasamos a calcular H (salto con instrucción JE). Caso contrario, seguimos.
- Con las instrucciones MOVDQU, PADDD, PSRLDQ y PSLLDQ preparamos en XMM5 los valores: [G,B,R,-]. Esto se lo restamos a los datos iniciales ([R,G,B,A]) en XMM0 con la instrucción PSUBD y nos queda en XMM4: [R-G,G-B,B-R,-].
- Luego, con las instrucciones CVTDQ2PS, MOVDQU, PADDD y PSLLDQ preparamos en XMM3 un empaquetado con formato Float SP con los valores: [d,d,d,d] y esto se lo dividimos a lo que habíamos obtenido en XMM4 con la instrucción DIVPS.
- Cargamos en XMM6 una máscara con los valores [4,6,2,0] en formato Floats SP y se lo sumamos a XMM4 con la instrucción ADDPS.
- Cargamos en XMM6 una máscara con los valores [60,60,60,60] en formato Floats SP y se lo multiplicamos a XMM4 con la instrucción MULPS.
- Ahora, preparo en EDX, EAX, R8D y R9D los valores de $cmax$ y los canales B, G y R respectivamente, para hacer las comparaciones.
- Comparo EDX con R9D. Si es igual, movemos el valor $60 * ((G - B)/d + 6)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Comparo EDX con R8D. Si es igual, movemos el valor $60 * ((B - R)/d + 2)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Acá no hace falta comparar, ya que $cmax$ no era igual a R o G, entonces es igual a B. Movemos el valor $60 * ((R - G)/d + 4)$ en XMM4 a EDX con las instrucciones PSRLDQ y MOVD y saltamos al final. Sino, pasamos al siguiente caso.
- Ahora, tenemos que chequear que el valor en EDX sea menor igual a 360. Esto lo podemos hacer comparando EDX con 360 y con la instrucción JL. En caso de que sea mayor, cargamos una máscara con un 360 en los 32 bits más bajos y hacemos la resta en los registros XMM con la instrucción SUBPS
- Al final, copiamos el valor obtenido en los 32 bits siguientes a S en XMM14 con las instrucciones PSLLDQ y ADDPS.

Por último, copiamos el Alpha original (en formato Float SP) en los 32 bits más bajos de XMM14 con las instrucciones PSLLDQ, PSRLDQ, CVTDQ2PS y POR. El resultado en el registro XMM14 se copia al registro XMM0 con la instrucción MOVDQU, ya que es en XMM0 donde la función llamadora espera el resultado (esto lo podemos hacer ya que esta todo en Assembler y podemos garantizar la sanidad de los registros).

HSL a RGB El pasaje de HSL a RGB es muy similar al de RGB a HSL respecto al tipo de operaciones aritméticas y de conversión utilizadas, con la salvedad de que se realizan en otro orden.

Como primer paso, limpiamos el registro `XMM14` con la instrucción `PXOR` para ir almacenando los resultados parciales. Luego, cargamos el píxel en formato HSL que viene en el registro `XMM5` (esto lo podemos hacer ya que esta todo en Assembler y podemos garantizar la sanidad de los registros) y movemos cada canal a la parte más baja de un registro distinto: L a `XMM1`, S a `XMM2` y H a `XMM3` (se usan las instrucciones `PSLLDQ`, `PSRLDQ` y `MOVDQU`).

Cálculo de c El calculo de c es idéntico al cálculo de S en la función $rgbTOhsl$, excepto que una vez obtenido $1 - fabs(2 * L - 1)$ lo multiplicamos por S utilizando la instrucción `MULPS`, dejando el resultado en los 32 bits más bajos de `XMM14`.

Cálculo de x

- Primero, cargamos en `XMM6` una máscara en formato Float SP con los valores: [60,60,60,60] y la utilizamos para dividir `XMM4` (H) con la instrucción `DIVPS`, dejando el resultado en `XMM4`.
- Ahora tenemos que calcular la función $fmod(H/60, 2)$. Para esto, primero cargamos en los 32 bits más bajos de `XMM5` un 2 en formato Float SP y se lo dividimos a `XMM4`, dejando el resultado en este último. Luego, copiamos `XMM4` a `XMM6` con la instrucción `MOVDQU` y convertimos `XMM6` a Entero 32 bits con la instrucción `CVTTPS2DQ`, que realiza una conversión truncada, quedándonos en `XMM6` la parte entera de $(H/60)/2$. Volvemos a convertir `XMM6` a Float SP, y se lo restamos a `XMM4` con las instrucciones `CVTPS2DQ` y `SUBPS`. Una vez tenemos esto, solo falta multiplicar lo obtenido por 2 con la instrucción `MULPS`. El resultado parcial queda en `XMM4`.
- A continuación se realizan las restas de unos y el calculo de $fabs$ de forma idéntica a como se hizo en el cálculo de S en la función $rgbTOhsl$.
- Por último, multiplicamos el resultado parcial por c con la instrucción `MULPS` y dejamos el resultado en los 32 bits más bajos de `XMM13`.

Cálculo de m El cálculo de m es bastante sencillo comparado al recién hecho. Utilizamos la instrucción `DIVPS` para dividir por 2 el valor de c y luego se lo restamos a L con la instrucción `SUBPS`. El resultado lo dejamos en los 32 bits más bajos de `XMM12`.

Elección de los canales RGB Ahora que tenemos los valores de c, x y m, solo nos queda asignárselos a cada canal R, G y B, dependiendo del valor de H. Para esto, vamos a tener una serie de condicionales. Como H es un valor en formato Float SP, tenemos que hacer la comparación usando los registros `XMM`. Es por eso que tenemos una serie de máscaras en formato Float SP que tienen los valores: [0,0,0,60], [0,0,0,120], ..., [0,0,0,300]. Dependiendo de en que caso hayamos caído, vamos a ir guardando en `XMM9`, `XMM10` y `XMM11` los valores de R, G y B respectivamente.

Empezando por la primera de las máscaras, se carga en el registro `XMM6` con la instrucción `MOVDQU` y se compara con el valor de H en `XMM3` con la instrucción `CMPLEPS`. Nos queda entonces en los 32 bits más bajos de `XMM6` el valor de $60 \leq H$. Luego, movemos este valor al registro `EAX` y lo comparamos con cero (FALSE).

Si `EAX` no es igual a cero (instrucción de salto `JNE`), quiere decir que $60 <= H! = FALSE \iff 60 <= H == TRUE$ y pasamos a evaluar la siguiente máscara (120). Caso contrario, copiamos `XMM9` a `XMM14` ($R = c$), `XMM10` a `XMM13` ($G = x$) y seteamos en cero `EAX` ($B = 0$) (con la instrucción `PXOR`) y saltamos a la última parte del algoritmo.

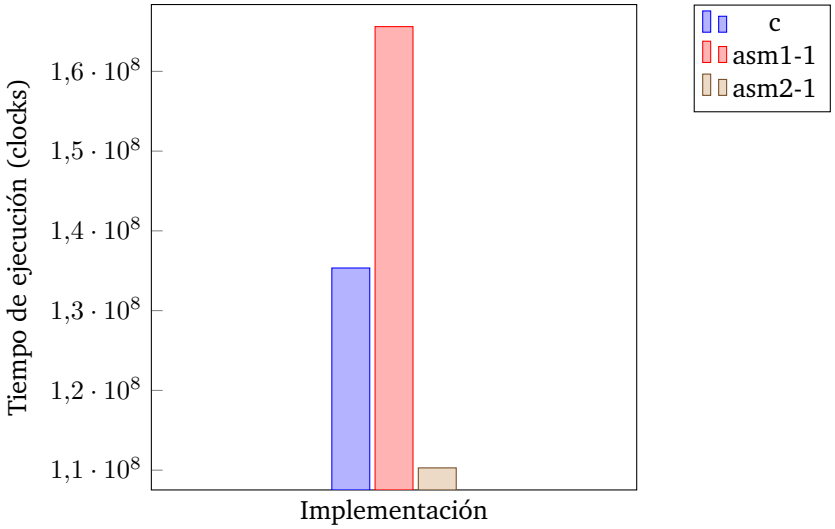
El resto de los casos condicionales se desarrolla de manera idéntica a la descrita para el primer caso. En el caso que $300 <= H$ no nos hace falta evaluar $360 <= H$ y asignamos directamente los valores correspondientes.

A continuación, le sumamos a `XMM9`, `XMM10` y `XMM11` el valor de m en `XMM12` con la instrucción `ADDPS`. Luego, acomodamos los valores de R, G y B en el registro `XMM9` de forma: [R,G,B,0] y lo multiplicamos por una máscara cargada en `XMM6` con [255,255,255,255] en formato Float SP.

Convertimos `XMM9` y el Alpha original a Enteros 32 bits con la instrucción `CVTPS2DQ` y los juntamos en el registro `XMM0` con las instrucciones `PSLLDQ`, `PSRLDQ` y `PADDD`. Por último, convertimos los valores en `XMM9` a enteros de 8 bits con las instrucciones `PACKUSDW` y `PACKUSWB`, y escribimos el resultado en la dirección apuntada por `RDI` usando `MOVD`.

5.2. Análisis y Resultados

El siguiente gráfico muestra para una misma imagen, los rendimientos obtenidos corriendo las diferentes versiones del algoritmo.



Y la siguiente tabla resume los tiempos obtenidos para distintas imagenes (todas de 512x512).

Imagen	Tiempo C	Tiempo ASM1	Tiempo ASM2	Cambio relativo C vs ASM1		Cambio relativo ASM1 vs ASM2	
				C → ASM1	ASM1 → C	ASM1 → ASM2	ASM2 → ASM1
1	131962942	164047712	111603874	24.31 % peor	19.56 % mejor	31.97 % mejor	46.99 % peor
2	135343929	165607349	110284333	22.36 % peor	18.27 % mejor	33.41 % mejor	50.16 % peor
3	35966679	44981411	30539176	25.06 % peor	20.04 % mejor	32.11 % mejor	47.29 % peor

Se observa que la implementación de mejor rendimiento es ASM2, seguida por C y luego por ASM1. Observando los Cambios relativos obtenidos podemos ver que:

- La implementación C debería ser, en promedio, un 23 % mas lenta para tenér el mismo tiempo que ASM1.
- La implementación ASM1 debería ser, en promedio, un 19 % mas rápida para tenér el mismo tiempo que C.
- La implementación en ASM1 debería ser, en promedio, un 32 % mas rápida para tenér el mismo tiempo que ASM2.
- La implementación en ASM2 debería ser, en promedio, un 48 % mas lenta para tenér el mismo tiempo que ASM1.

5.2.1. Hipótesis planteadas

A partir de las ventajas que supone el modelo de computo SIMD, nos propusimos realizar varias implementaciones con el fin de analizar las diferencias de performance entre:

- ASM1 vs C:
  1. Realizar las operaciones en la version ASM1 en bloques (comparaciones, sumas y restas) y compararla con los resultados de la version en C.

2. Almacenar y manipular la información en registros `XMM` asi evitando una gran cantidad de accesos a memoria que en C se hacen individualmente por dato.

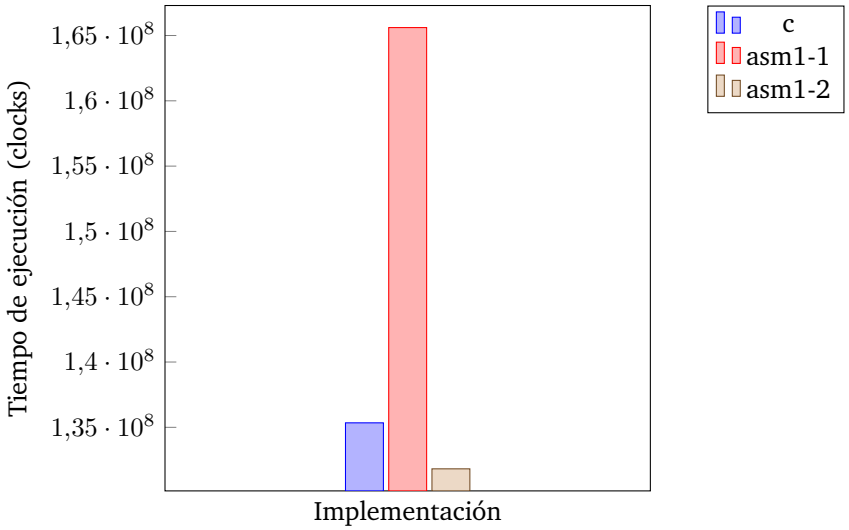
- ASM2 vs ASM1 vs C:

1. Realizar las conversiones de formato en la version ASM2 con instrucciones Packed en bloques (comparaciones, sumas y restas). Compararla con los resultados de la version en C.
2. Hacer el pasaje de parámetros en ASM2 (los pixeles) en registros `XMM` asi evitando accesos a memoria que en C se hacen siempre.

5.2.2. Experimentación

Como se puede apreciar en la tabla de resultados, el rendimiento de C es mejor que el de HSL1. Esta baja en la performance está dada por el esfuerzo extra, medido en cantidad de operaciones necesarias, para operar sobre componentes particulares de una imagen. Ademas existe un overhead debido a que la función tiene que llamar a las rutinas externas de conversión entre RGB y HSL escritas en C.

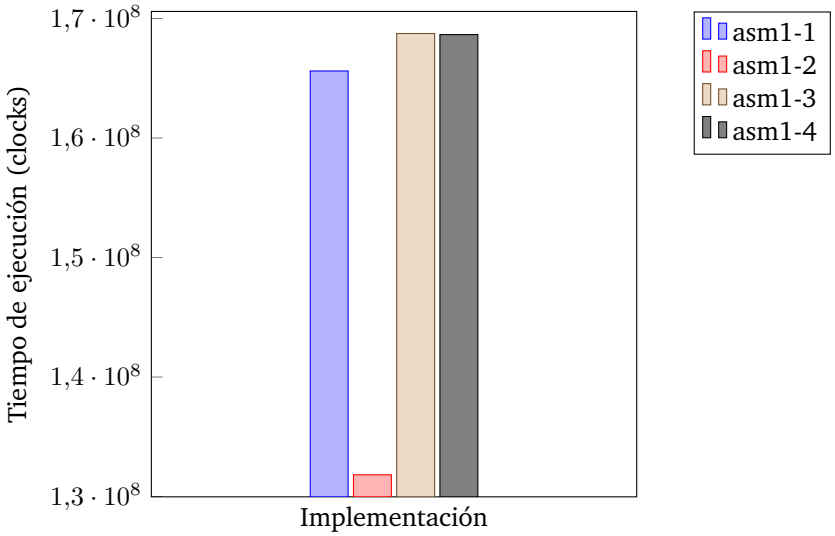
Revisando el código generado en Assembler desde el código en C, pudimos ver que la gran mayoría de las instrucciones realizadas son del tipo Single Scalar, tanto para los accesos a memoria como para sumar, restar y las demás operaciones. Por esto, hicimos un experimento 'asm_1_2' similar a la versión de C a partir de nuestra propia implementación en Assembler que, a diferencia de la primer versión, realiza todos los accesos a memoria, calcular máximos y mínimos, comparaciones, sumas y restas con operaciones Single Scalar para punto flotante. Los resultados entonces fueron muy parecidos a la versión en C:



El hecho de que 'asm_1_2' sea mas rapida que 'asm_1_1' lo atribuimos a que, por un lado, 'asm_1_1' realiza algunas operaciones de comparacion y shift como `CMPNLTPS` y `PSRLDQ` para poder operar en paralelo y en 'asm_1_2' no necesitamos hacer shift y usamos operaciones Single Scalar porque operamos por separado cada componente y las podemos buscar tambien por separado. Ademas, notamos una diferencia notable de velocidad cuando buscamos las componentes a la vez con la operacion `MOVDQU` (y tambien con `MOVDQA`, `MOVUPS` y `MOVAPS`) en 'asm_1_1', siendo esto mucho mas lento que buscar las componentes por separado con `MOVSS` en 'asm1_2' (notar que basta con traer tres componentes en este caso, porque el alpha no cambia).

Adicionalmente, realizamos dos experimentaciones mas. El experimento 'asm_1_3' consistió en traer los datos de memoria a un mismo registro XMM y desde allí operar las componentes por separado (pero a diferencia de 'asm_1_1' usamos las operaciones `CMP` scalar, `MAXSS` y `MINSS` y shifts para modificar las componentes con operaciones mayormente Single Scalar). En 'asm_1_4' fuimos un paso mas y tratamos de ademas realizar las comparaciones y la toma de máximos y mínimos con operaciones Packed Single, así paralelizando las operaciones y minimizando los accesos a memoria, ahora de mayor tamaño (128 bits). Sin embargo, las dos experimentaciones tuvieron resultados adversos, así siendo mas lentos que

'asm_1_2'. Esto se debe a que el escenario no es ideal para realizar cuentas en paralelo, ya que es una cuenta relativamente corta donde los 3 componentes que hay que modificar requieren distintas operaciones (como max, min, cmp, insert en registros XMM) y tratar de realizarlas con instrucciones Packed en paralelo conlleva el uso de instrucciones mas caras que podrían ser reemplazadas por otras instrucciones mas baratas (como las usadas en 'asm_1_2').

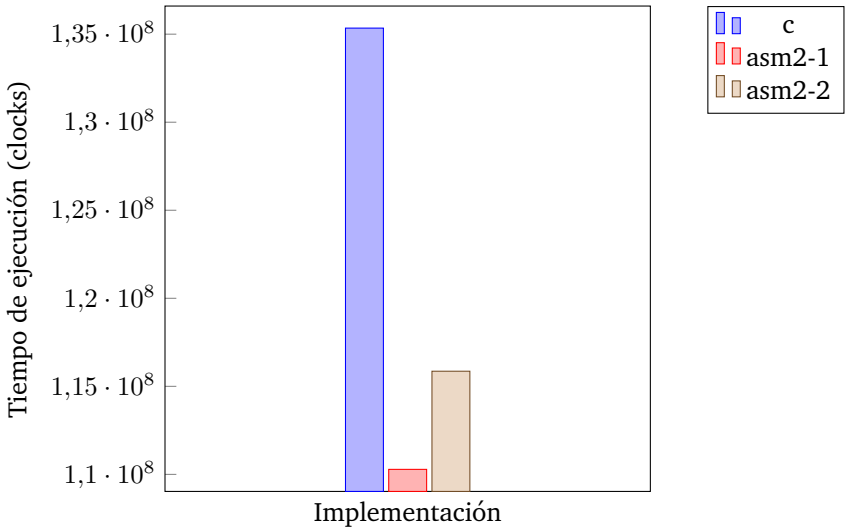


Por lo tanto concluimos que las instrucciones de SIMD son ideales para trabajar sobre operaciones iguales sobre datos contiguos pero no para hacer operaciones distintas que tiene más sentido realizar directamente sobre escalares.

La mejoría de ASM2 respecto a C la vemos relacionada con la paralelización de las instrucciones y con la falta de necesidad de pasar varios parámetros por memoria. Revisando el desensamblado del código en lenguaje C, vemos que realiza varias instrucciones del tipo Single Scalar, tanto para sumar, restar, multiplicar y dividir como para acceder a memoria.

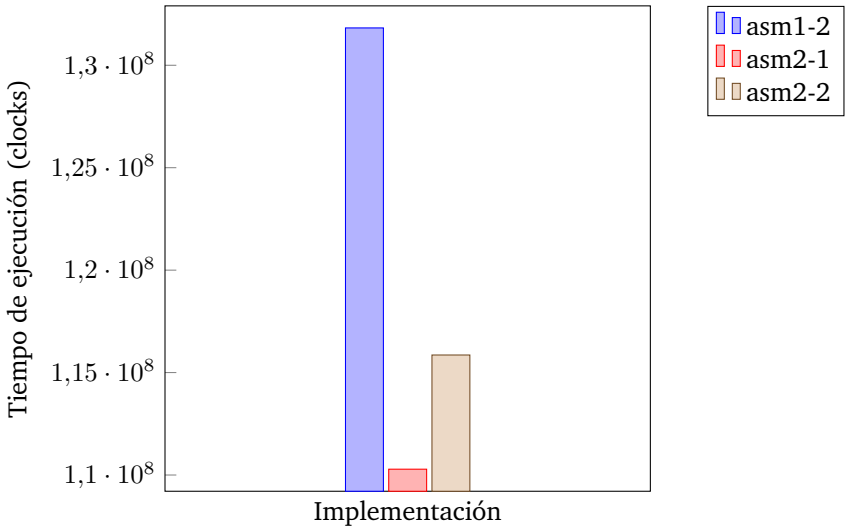
Ademas, la versión en C guarda en memoria los resultados de pasar el pixel de rgb a hsl, mientras que la versión en Assembler los pasa por registros XMM. Esto es una ventaja para nosotros y la podemos aprovechar ya que ya sabemos cuando los registros XMM no se corrompen al hacer CALL en las funciones. También pudimos tomar ventaja de esto en el pasaje de hsl a rgb, mandando el valor respuesta por el registro XMM5, así también ahorrándonos accesos caros a memoria.

En base a estas hipótesis, realizamos dos versiones diferentes de ASM2. En 'asm_2_1' pasamos los parámetros mencionados por registros XMM y en 'asm_2_2' los pasamos los parámetros por memoria, de la misma manera en que se pasan en las versiones C y ASM1.



Los resultados se condicen con nuestras hipótesis ya que 'asm_2_1' es consistentemente mas rápida que 'asm_2_2'. A la vez, tanto 'asm_2_1' como 'asm_2_2' es mas rápida que la versión en C, así también condiciendo nuestra hipótesis de que mediante Assembler pudimos aprovechar el procesamiento en paralelo.

Por ultimo comparamos las versiones 'asm_1_2' con 'asm_2_1' y 'asm_2_2':



Como vemos, la version 'asm_2_2' (que por cierto no utiliza las instrucciones Single Scalar que aprovecha 'asm_1_2') ya es mucho más rápida que 'asm_1_2', lo que quiere decir que la paralelización en las operaciones de conversión entre HSL y RGB (para un lado y para el otro) tiene mayor mejora de performance que pasar las instrucciones a Single Scalar en el ciclo principal donde se modifican las componentes. Esto tiene sentido ya que:

- En este caso, la paralelización de las instrucciones aprovecha las caracteristicas del modelo SIMD, que como ya vimos tambien en los otros filtros conlleva muchas mejoras de performance cuando hay que realizar muchas cuentas parecidas (a diferencia del caso 'asm_1_1' vs 'asm_1_2').
- Las rutinas de conversión hacen muchas mas cuentas que el ciclo principal que modifica las componentes.

Por ultimo observamos que la versión mas rapida de todas fue 'asm_2_1' y esto era predecible porque ya habiamos visto que tenia mejor performance que 'asm_2_2', que a su vez es más rápida que la versión en C y 'asm1_2' (la más rápida de todas las versiones 'asm_1_x').

6. Conclusión

Partiendo desde las implementaciones en lenguaje C de los algoritmos, realizamos distintas versiones en lenguaje Assembler x86. Si bien se pierde la expresividad que facilita la implementación, se ven mejoras en prácticamente todos los casos, debido a las ventajas de operar en bajo nivel. Particularmente, pudimos ver mejoras significativas de rendimiento mediante el uso de instrucciones SIMD. Esto se debe a dos factores clave: el acceso a memoria de a varios bloques, que minimiza la cantidad de accesos (operación cara) y la paralelización de los procesos. La ganancia de performance mas significativa entre versiones de Assembler la tuvimos cuando pudimos acceder a mayor cantidad de pixeles por ciclo de procesamiento. Ademas, la paralelización de operaciones como sumar, dividir, etc.. que en vez de realizarlas para cada dato por separado, pudimos realizar para varios datos a la vez, nos llevo a una mejora de performance significativa aun cuando no pudimos acceder a varias pixeles en memoria a la misma vez, por ejemplo, para operar sobre las diferentes componentes de un pixel.

A. Anexo - metodología para las mediciones

Script utilizado para inicializar el entorno de mediciones:

```
#!/bin/bash
# From SO: http://stackoverflow.com/questions/9072060/one-core-
    exclusively-for-my-process

mkdir /cpuset
mount -t cpuset none /cpuset/
cd /cpuset

mkdir sys # create sub-cpuset for system processes
/bin/echo 0-2 > sys/cpus # assign cpus (cores) 0-2 to this set
/bin/echo 1 > sys/cpu_exclusive
/bin/echo 0 > sys/mems

mkdir rt # create sub-cpuset for my process
/bin/echo 3 > rt/cpus # assign cpu (core) 3 to this cpuset
/bin/echo 1 > rt/cpu_exclusive
/bin/echo 0 > rt/mems
/bin/echo 0 > rt/sched_load_balance
/bin/echo 1 > rt/mem_hardwall

# move all processes from the default cpuset to the sys-cpuset
for T in `cat tasks`; do echo "Moving " $T; /bin/echo $T > sys/tasks;
    done

# disable interruptions for the isolated cpu
find /proc/irq/ -name "smp_affinity" -exec sh -c "echo 7 > {}" \;

# set fixed frequency for the isolated cpu
cpufreq-selector -c 3 -f 800000

# start task in specific isolated cpu
$1 & /bin/echo $! > /cpuset/rt/tasks    # Move the just created task PID
     to the tasks of the exclusive CPU
```