

Organización del Computador II

TP2

7 de mayo de 2015

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

Reservado para la cátedra

Instancia	Docente	Nota
Primera entrega		
Segunda entrega		

Índice

1. Introducción	3
1.1. Historia	3
1.2. Filtros de Imágenes	3
1.3. Organización del informe	3
2. Blur	4
2.1. Introducción	4
2.2. ASM1	4
2.3. ASM2	5
2.4. Comentarios	8
2.5. Experimentación	8
3. Merge	10
3.1. C	10
3.2. ASM1	11
3.3. ASM2	11
3.3.1. Error	12
3.4. Experimentación	13
4. HSL	15
4.1. C	15
4.2. ASM1	15
4.3. ASM2	16
4.4. Experimentación	17
5. Conclusión	21

1. Introducción

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

1.1. Historia

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

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

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

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

1.2. Filtros de Imágenes

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

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

1.3. Organización del informe

Este informe está dividido en 3 secciones, en cada una se analizan los 3 filtros propuestos por la cátedra. En la primera sección, se analizan los algoritmos que utilizamos, explicandolos y analizando sus pros y sus contras.

Luego procede la parte de experimentación de cada filtro, en la que respondemos las preguntas de la cátedra, ademas de analizar los resultados de los tests de rendimiento. Además, en algunos casos que nos pareció interesante planteamos otros experimentos que hicimos y los analizamos.

2. Blur

2.1. Introducción

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

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

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

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

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

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

2.2. ASM1

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

	P1	P2	P3	P4
MEM	P5	P6	P7	P8
	P9	P10	P11	P12

Cuadro 1: Ilustración de la memoria en blur. En este caso en particular, se está procesando el píxel P6. Los píxeles rojos representan lo que debemos descartar al cargar de a 4 píxeles en los registros XMM. Los píxeles naranjas y el P6 son promediados para dar lugar a un nuevo píxel.

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

Luego, en el ciclo de las columnas, buscamos en la copia de la primera fila los 4 píxeles (16 bytes) correspondientes al iterador actual de las columnas. Por la razón explicada anteriormente, debemos levantar los canales por separado, de tal manera de no hacer lecturas inválidas.

Cada píxel ocupa 32 bits, 1 byte por cada componente ARGB. En la memoria, los archivos *.bmp* guardan los componentes de los píxeles en el orden A B G R. Como la arquitectura Intel es little-endian, al mover estos píxeles a un registro, no solo se invertirá el orden de los píxeles sino que también el de sus componentes.

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

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

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

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

XMM2	R_6	G_6	B_6	A_6	R_5+R_7	G_5+G_7	B_5+B_7	A_5+A_7
-------------	-------	-------	-------	-------	-----------	-----------	-----------	-----------

XMM3	R_{10}	G_{10}	B_{10}	A_{10}	R_9+R_{11}	G_9+G_{11}	B_9+B_{11}	A_9+A_{11}
-------------	----------	----------	----------	----------	--------------	--------------	--------------	--------------

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

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

Copiamos **XMM1** en **XMM2** y lo desplazamos 8 bytes a la derecha:

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

Sumando **XMM1** y **XMM2** en **XMM1**:

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

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

XMM1	9R	9G	9B	9A
-------------	----	----	----	----

Tomo el promedio de los componentes de cada píxel dividiendo por el registro

XMM7	9.0	9.0	9.0	9.0
-------------	-----	-----	-----	-----

XMM1	R_p	G_p	B_p	A_p
-------------	-------	-------	-------	-------

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

2.3. ASM2

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

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

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

El primero, cómo iterar la matriz, se resuelve observando que teniendo a **RAX** como un puntero a la esquina superior izquierda de la matriz es muy simple hacer referencia en memoria a las posiciones de la matriz: tenemos que $\mathbf{RAX} + 4*j$ es el $j + 1$ -ésimo píxel en la fila actual respecto del primer píxel, por lo que si tomamos $j \in \{0, \dots, 5\}$ tenemos a todos los píxeles de la primera fila. Luego, si queremos indexar otra fila, podemos tomar $\mathbf{RAX} + 4*w*i$, que nos determina al primer elemento en la i -ésima fila por debajo de la primera (en el ejemplo, si tomamos $i = 1$, esto apunta a la dirección de memoria de P_7), en este caso tenemos que $i \in \{0, 1, 2\}$.

Es decir, mantener este puntero nos permite iterar la matriz por bloques de 3×6 con la única contra de tener que actualizar el puntero a **RAX** cada vez que queremos cambiar de submatriz: $\mathbf{RAX} = \mathbf{RAX} + 16$ es un movimiento obligatorio cada vez que procesamos los cuatro píxeles que corresponden a un ciclo. Además, tenemos que mantener el número de fila en el que estamos.

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

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

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

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

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

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

input : Puntero a la matriz de píxeles representando la imagen, w su ancho, h su altura

output: La matriz de píxeles se actualizo, luego de haberse aplicado blur

crear vectores de para las primeras filas ($r10$ y $r11$)

$r10 \leftarrow$ fila 0 de la matriz (copiada)

inicializar el índice rax a la submatriz

for $i \leftarrow 1$ **to** $h - 2$ **do**

$rax \leftarrow$ nuevo índice de la submatriz, $rax + i * w * 4$

Swap ($r10$, $r11$)

$r10 \leftarrow$ fila i de la matriz (copiada)

$rax \leftarrow$ nuevo índice de la submatriz, $rax + i * w * 4$

for $j \leftarrow 1$ **to** $w - 2$ **do**

if $j == w - 3$ **then**

$j \leftarrow j - 2$

end

 copiar datos de $r10$ y $r11$ a los registros *XMM*

 convertir las componentes de los registros a enteros de 16 bits

 reordenar los registros para que queden de a columnas

 sumar los registros, ahora hay 3 registros con las 6 columnas

for $k \leftarrow 0$ **to** 3 **do**

 sumar las columnas necesarias para el k -ésimo píxel

 convertir los canales a float

 dividirlos por 9

 convertir todo de vuelta a 1 byte con saturación

 guardar el píxel en memoria

end

$j \leftarrow j + 4$

$rax \leftarrow rax + 16$

end

end

liberar la memoria de los vectores

Algorithm 1: Algoritmo de blur procesando de a 4 píxeles

2.4. Comentarios

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

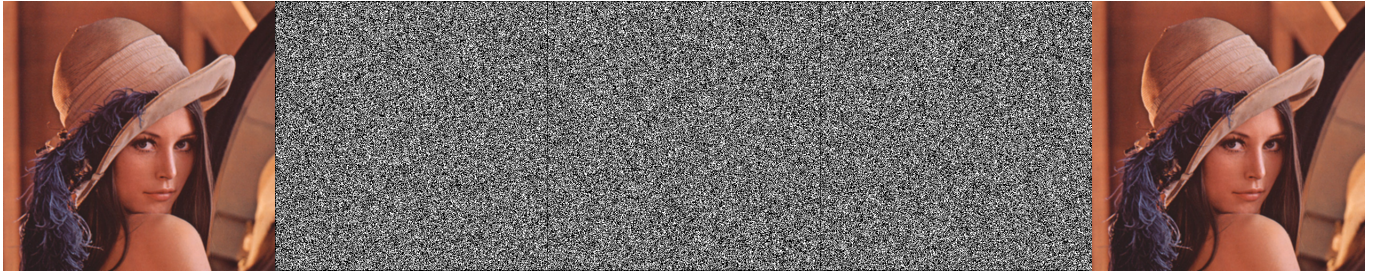


Figura 1: Blur C Figura 2: R channel Figura 3: G channel Figura 4: B channel Figura 5: Blur ASM

Aunque las im genes parec an exactamente iguales, hab an diferencias en algunos p xeles. Estas diferencias se deb an al redondeo hacia arriba llevado a cabo por la conversi n desde float a entero. Mientras que el c digo en C redondeaba hacia abajo por defecto, las conversiones en ASM redondeaban hacia arriba por defecto.

Para resolver esto, existe un flag de SSE llamado *MXCSR*. Para m s informaci n, ver la secci n 10.2.3.1 del Volumen 1 de la gu a de arquitectura Intel. Cada bit de este flag codifica alg n comportamiento de las operaciones SSE. El valor por defecto de este flag es *0x1F80*. Para que redondee hacia abajo, utilizando la codificaci n de los bits del flag, hab a que poner el bit 12 y 13 en 1. Por lo tanto, simplemente setteamos el flag con la instrucci n *ldmxcsr* en *0x7F80* y el algoritmo comenz  a andar sin problemas. Tuvimos esta misma problem tica en todos los otros filtros m s tarde.

2.5. Experimentaci n

Es f cil darse cuenta, ya sea mirando el c digo o comprendiendo qu  es lo que debe hacer el algoritmo de Blur, de que todas las implementaciones del mismo van tener una complejidad temporal de $O(w * h)$, indicando que la cantidad de operaciones que realizar  el algoritmo ser n de orden cuadr tico si tomamos $w = h = n$. Es decir, los gr ficos que generemos deber n tener la forma de una par bola a medida que n se vuelva m s grande. Lo interesante, y he aqu  donde radican las ventajas de hacer el c digo en Assembler, es que sabemos que las tres implementaciones diferir n en un m ltiplo constante de las otras (por la definici n de O). Intuitivamente, el c digo de ASM1 deber a ir aproximadamente 3 veces m s r pido por p xel que el de C, ya que procesa cada canal en paralelo con SIMD, y el c digo de ASM2 deber a ser unas 4 veces m s r pido que el de ASM1, ya que no s lo procesa los canales en paralelo, sino que adem s procesa los p xeles en grupos de a cuatro.

Cabe destacar, de cualquier forma, que nuestra intuici n probablemente falle: tanto las optimizaciones que pueda llegar a realizar el compilador de C al c digo, as  como tambi n cuestiones de acceso a memoria en las distintas implementaciones (inclusive las instrucciones utilizadas en cada caso, o el branch predictor), y particularidades respecto del estado actual del sistema en el que se corren los experimentos pueden hacer variar los n meros obtenidos ampliamente, como veremos m s adelante en el caso de HSL.

Haciendo un an lisis poco delicado, observemos que la versi n de ASM1 tiene 4 saltos condicionales, contra los 5 de ASM2, por lo que en principio, suponiendo que el branch predictor falle, la versi n de ASM2 podr a terminar siendo m s cara. Por otro lado, el hecho de procesar de a 4 p xeles nos dar  una ventaja dif cil de estimar con respecto al ASM1. En t rminos del acceso a memoria, ambas versiones hacen uso de la memoria relativamente poco con respecto a los otros algoritmos, entrando solamente con fines de obtener las filas necesarias para procesar los datos, por lo que estimamos que en principio no deber a ser el factor m s importante, a pesar que todos los accesos que realizamos son desalineados.

Una cosa que nos parece importante resaltar es la manera de la que levantamos los p xeles en la ASM1. Hay muchas

formas posibles, por ejemplo, levantar de a 4 siempre, excepto en la última iteración en la que se debe hacer algún malabar para no levantar la parte que causaría un invalid read. Esta forma uno pensaría que tiene un rendimiento muy superior porque se levantan menos cosas de memoria. Sin embargo, nuestra primera versión de la implementación levantaba así los píxeles, y tenía el problema del invalid read. Por esa razón, decidimos cambiarnos a la versión actual. La performance no se vio golpeada. Atribuimos esto a que una vez que leímos los datos, estos quedan en la cache, entonces la siguiente lectura va a ser mucho más rápida. Por otro lado, nos ahorramos algunos saltos condicionales que pueden causar algún que otro branch misprediction.

Además, la performance de nuestros algoritmos de Blur no se verá afectada por ninguna otra característica de la imagen que el tamaño: ninguno de los algoritmos actúa según condiciones sobre el contenido de las imágenes. Las tres implementaciones simplemente procesan datos, sin comparar con ninguna otra cosa que no sea el número de columna o de fila. En concreto, esperamos que nuestras implementaciones en Assembler sean ambas más rápidas que la de C, y que entre ellas ASM2 sea más rápida que ASM1 (fundamentalmente por el procesamiento en paralelo de a muchos píxeles).

Observemos que, efectivamente, se cumple lo que esperábamos: los ciclos de reloj crecen de forma cuadrática en relación al tamaño de la imagen. Además, notese que tienen relativamente poco desvío una implementación con respecto a la otra, suponemos que esto quiere decir que tienen comportamientos regulares con respecto al pipeline y sus accesos a memoria (de otra forma, tendríamos más picos en el gráfico). Nos sorprendió mucho que la implementación de ASM1 se asimile tanto a la de ASM2, ya que esperábamos una mayor amplitud entre las curvas; por esto, corrimos un nuevo experimento tomando tamaños de imagen mucho más grandes:

Esta vez, encontramos una brecha mucho más marcada entre las distintas implementaciones, lo que nos asegura que efectivamente el crecimiento de las constantes escondidas en la complejidad temporal está haciendo una diferencia importante a medida que aumenta la escala. Para finalizar, tomamos un ejemplo más de cerca para observar más precisamente las disparidades:

Como habíamos notado antes, el hecho de estar utilizando imágenes pequeñas no ayuda a que se note la diferencia entre ASM1 y ASM2. Pensamos que uno de los posibles motivos para esto es que los algoritmos que escribimos no están optimizados en tanto a sus accesos a memoria: ambos hacen accesos desalineados. Una posible mejora para el algoritmo de ASM2 sería utilizar los registros AVX para poder obtener más píxeles de memoria, disminuyendo la cantidad de accesos, y permitiéndonos un trabajo mucho más rápido, procesando de a 8 píxeles en simultáneo. Otra posible mejora sería pedir que las direcciones de memoria con las que trabajamos estén alineadas a 16 bits. Esto nos permitiría realizar siempre accesos alineados, lo que debería mejorar la performance de nuestros algoritmos.

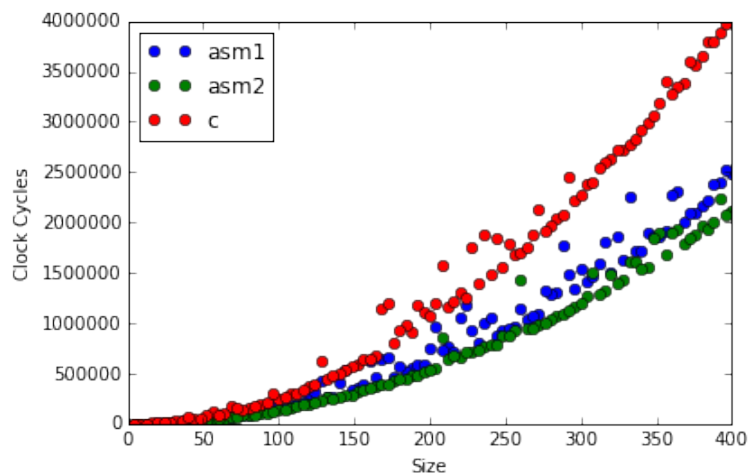


Figura 6: Performance de C compilado con -o3 y -ffastmath, ASM1 y ASM2. Utilizamos un conjunto de imágenes cuadradas con tamaños múltiplos de 4 y píxeles tomados de una distribución uniforme, y medimos los resultados de 100 corridas, tomando como estadístico el mínimo de la cantidad de ciclos de reloj en todas las corridas.

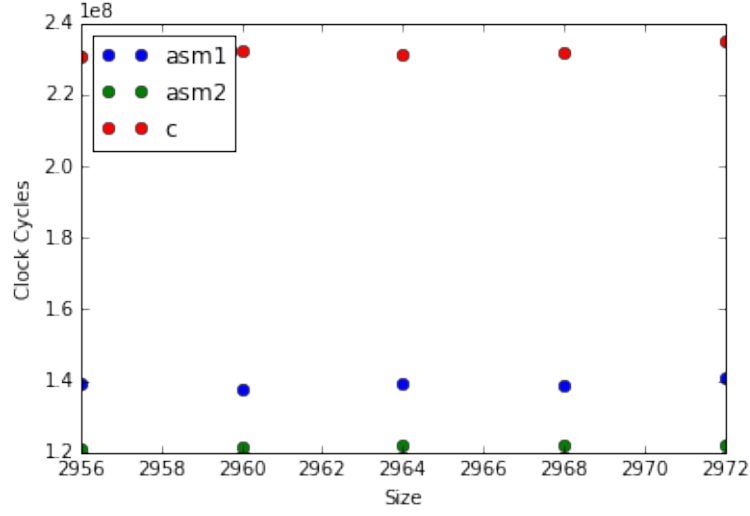


Figura 7: Performance de C compilado con -o3 y -fastmath, ASM1 y ASM2. Utilizamos un conjunto de imágenes cuadradas con tamaños múltiplos de 4 y píxeles tomados de una distribución uniforme, y medimos los resultados de 100 corridas, tomando como estadístico el mínimo de la cantidad de ciclos de reloj en todas las corridas.

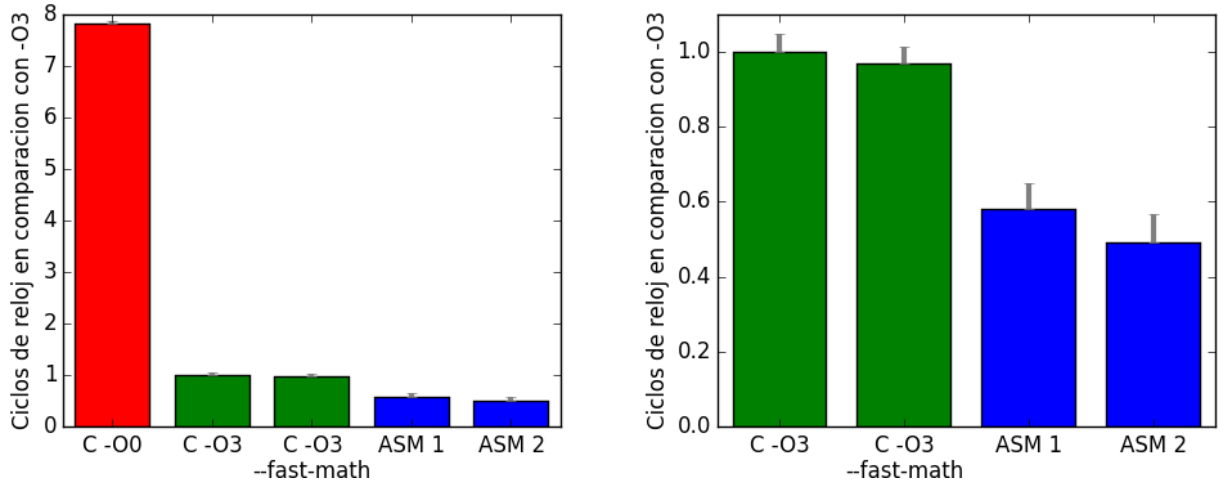


Figura 8: Performance de C compilado con -o0, C compilado con -o3 y -fastmath, ASM1 y ASM2. Utilizamos una imagen roja de 400×400 , con píxeles tomados de una distribución uniforme, y medimos los resultados de 100 corridas, tomando como estadístico el mínimo de la cantidad de ciclos de reloj en todas las corridas.

3. Merge

3.1. C

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

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

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

3.2. ASM1

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

En el ciclo lo que hacemos es tener dos vectores de floats

XMM3	<i>value</i>	<i>value</i>	<i>value</i>	1,0
XMM4	$1 - \textit{value}$	$1 - \textit{value}$	$1 - \textit{value}$	0,0

Y levantamos de a un píxel, cuyos canales convertimos a float y multiplicamos por los registros exhibidos anteriormente. Luego convertimos a enteros de 8 bits nuevamente, empaquetando.

Notar que no se produce overflow. Tomemos por ejemplo el canal R, y supongamos que el valor del canal R para ambas imágenes es 255. Entonces $255\textit{value} + 255(1 - \textit{value})$ da 255, que entra en 8 bits (si por errores de redondeo diera más, no importa, ya que el empaquetado lo hacemos saturado, entonces se queda en 255).

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

Aunque en líneas generales esta rutina es igual a la de C, esperamos que tenga mejor rendimiento por dos razones. Primero, en esta rutina tenemos que hacer menos saltos, dado que solo tenemos un ciclo, mientras que la versión de C tiene 2 ciclos. Segundo, porque si el compilador no optimiza a la perfección las operaciones, es muy probable que realice las operaciones de los canales en serie, en vez de como lo hacemos nosotros, en paralelo. Es decir, en el código de C los canales se van a procesar uno detrás del otro, mientras que en nuestro código se van a procesar todos juntos.

3.3. ASM2

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

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

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

De esta manera obtenemos en XMM3 los siguientes valores

XMM3	256 <i>v</i>	256 <i>v</i>	256 <i>v</i>	256	256 <i>v</i>	256 <i>v</i>	256 <i>v</i>	256
------	--------------	--------------	--------------	-----	--------------	--------------	--------------	-----

Donde *v* es *value*.

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

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

```
psubw xmm4, xmm3
```

obtenemos en XMM4 exactamente lo que queremos, es decir que si sumamos $\textit{int16}$ a $\textit{int16}$ en XMM3 y XMM4 da 256. Luego comenzamos el ciclo principal, que es muy similar al anterior.

Viendo esta implementación creemos que su desempeño va a ser aún mejor que el de la primera implementación en Assembler, dado que hacemos operaciones con números enteros que son mucho más rápidas que las operaciones con

punto flotante. Sin embargo, hacer operaciones con números enteros implica un trade-off de velocidad por precisión, aunque el código es mucho mas rápido también es (muy poco) más inexacto. Analizaremos este error a continuación.

3.3.1. Error

En la segunda versión de merge se comete, obviamente, más error que en la primera, dado que estamos trabajando con enteros, debemos perder precisión cuando convertimos $256v$ a entero.

Por esta razón, aproximadamente el 30 % de las veces, el resultado difiere en más de 2 con el output de C de la cátedra. Esto se debe a que los métodos de cómputo son fundamentalmente distintos, uno con mucha mas precisión que la otra.

Pese a esto, el 100 % de los tests pasan con un error de 3, es decir, pese a que existe error, es despreciable. El error (por ejemplo, para el canal R) se puede expresar de la siguiente manera

$$\begin{aligned} & \left| \lfloor R_1 v \rfloor + \lfloor R_2(1 - v) \rfloor - \left\lfloor \frac{R_1 \lfloor 256v \rfloor}{256} \right\rfloor - \left\lfloor \frac{R_2(256 - \lfloor 256v \rfloor)}{256} \right\rfloor \right| \\ & \leq \left| \lfloor R_1 v \rfloor - \left\lfloor \frac{R_1 \lfloor 256v \rfloor}{256} \right\rfloor \right| + \left| \lfloor R_2(1 - v) \rfloor - \left\lfloor \frac{R_2(256 - \lfloor 256v \rfloor)}{256} \right\rfloor \right| \end{aligned}$$

Se puede verificar fácilmente con Wolfram-Alpha o algún software similar que es menor que 3.

3.4. Experimentación

Antes de comenzar a testear, supusimos que la performance de nuestros programas en Assembler iban a ser muy superiores a las de C. Creemos esto dado que vamos a aprovecharnos mejor de las utilidades de SSE que lo que puede hacer un compilador.

Esperamos también que la performance mejore aún más en la segunda implementación, dado que todas las operaciones las realizamos con enteros en vez de con punto flotante.

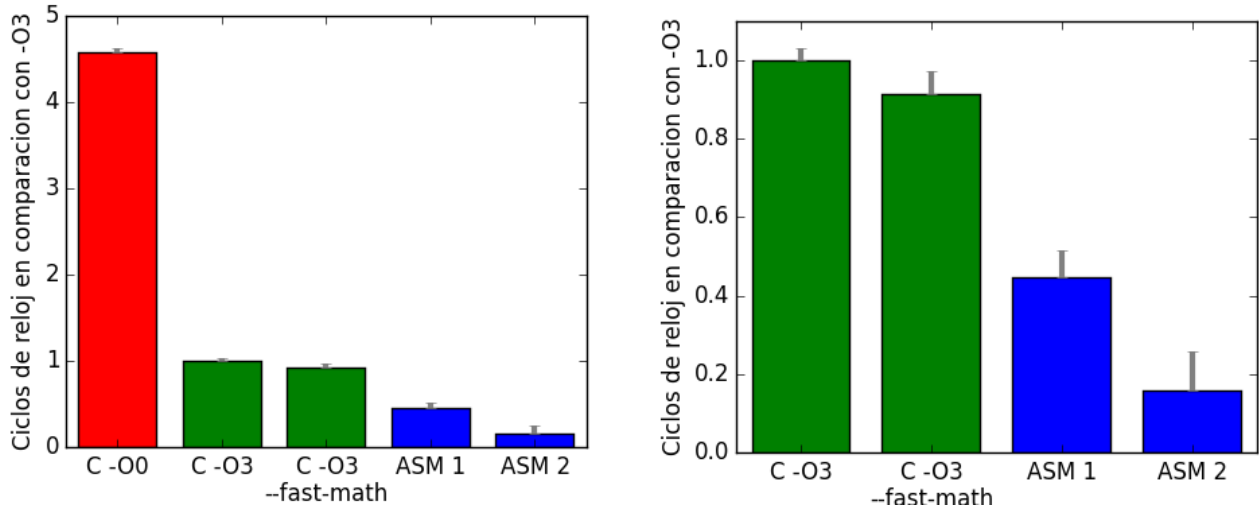


Figura 9: Comparación de la cantidad de ciclos de reloj utilizada por diferentes variantes de merge. Para tener una medida absoluta: la cantidad de ciclos de reloj promedio de C -O3 es 4,293,199. Tamaño de la muestra: 200 imágenes de 16×10^4 píxeles. Se indica el mínimo con la barra y el promedio con una línea gris.

Viendo los resultados, confirmamos nuestras hipótesis. De hecho, la performance de nuestros programas son mejores que las esperadas.

Como puede verse en los gráficos de escalabilidad, la performance de todos los algoritmos depende linealmente de la cantidad de píxeles de la imagen, sin embargo lo que cambia entre las diferentes implementaciones es la pendiente de esta función lineal.

La performance no depende de la imagen, dado que la operación que se realiza siempre es la misma, sin depender de los píxeles en cuestión. Sin embargo, hay una cuestión con el error. Si el α que nos pasan por parámetro tiene muchos decimales detrás de la coma, es muy probable, y de hecho pasa, que nuestra segunda implementación en Assembler pierda precisión frente a la implementación de C. Esto fue explicado anteriormente, sin embargo creímos pertinente nombrarlo.

La única mejora que se nos ocurre para el código propuesto de Assembler es, si supiéramos que el inicio de la matriz está alineado a 16 bytes, entonces podríamos levantar de a 4 píxeles y analizarlos todos juntos (haciendo una especie de ciclo unrolling, es decir, teniendo 4 ciclos normales dentro de 1). Esto es posible que mejore la performance, sin embargo es necesario como precondition que la matriz esté alineada a 16 bytes, ya que si no lo está y usamos una instrucción para leer alineado, los resultados serán desastrosos.

La comparación entre el programa de C y el de Assembler no es lo suficientemente justa. Esto se debe a que la versión de C por cada píxel, tiene 3 accesos a memoria, mientras que nuestra implementación tiene un acceso a memoria por píxel. Esto genera obviamente un speedup enorme, sobre todo en computadoras con memoria de velocidad limitada. Aunque es posible que la gran mayoría de los accesos sean a cache, la performance ganada es significativa.

El acceso a memoria en nuestros códigos de Assembler es lo mínimo indispensable. La única mejora que se podría hacer con respecto al acceso a memoria es lo que fue nombrado anteriormente es la idea de hacer un unroll del ciclo y leer de a 4 píxeles, pero creemos que la performance no mejorará significativamente.

La diferencia entre operar con punto flotante y con enteros es significativa. Aunque ambas implementaciones en Assembler son mucho mas rápidas que la implementación de C, la que opera con enteros es nuevamente (bastante) más rápida que la que opera con punto flotante. Esto se debe a que en general las operaciones con enteros son mucho mas rápidas que las de punto flotante.

4. HSL

4.1. C

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

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

4.2. ASM1

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

<code>XMM4</code>	l	s	h	0
-------------------	---	---	---	---

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

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

Luego comenzamos a loopear.

Luego de convertir el pixel en cuestión de `rgb` a `hsl`, vamos a tener su valor en `XMM3`.

<code>XMM3</code>	LL	SS	HH	AA
-------------------	----	----	----	----

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

<code>XMM3</code>	<code>l+LL</code>	<code>s+SS</code>	<code>h+HH</code>	AA
-------------------	-------------------	-------------------	-------------------	----

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

<code>XMM3</code>	0.5	-0.322	380	255
-------------------	-----	--------	-----	-----

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

<code>XMM5</code>	<code>1-(l+LL)</code>	<code>1-(s+SS)</code>	-360	0
-------------------	-----------------------	-----------------------	------	---

<code>XMM6</code>	<code>-(l+LL)</code>	<code>-(s+SS)</code>	360	0
-------------------	----------------------	----------------------	-----	---

Siguiendo el ejemplo anterior, los registros quedarían

<code>XMM5</code>	0.5	1.322	-360	0
-------------------	-----	-------	------	---

<code>XMM6</code>	-0.5	0.322	360	0
-------------------	------	-------	-----	---

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

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

<code>XMM12</code>	1	1	360	256
--------------------	---	---	-----	-----

<code>XMM13</code>	0	0	0	0
--------------------	---	---	---	---

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

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

<code>XMM12</code>	0h	0h	ffffffh	0h
--------------------	----	----	---------	----

<code>XMM13</code>	0h	ffffffh	0h	0h
--------------------	----	---------	----	----

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

<code>XMM5</code>	0	0	-360	0
-------------------	---	---	------	---

<code>XMM6</code>	0	0.322	0	0
-------------------	---	-------	---	---

Recordemos el valor de `XMM3`

<code>XMM3</code>	0.5	-0.322	380	255
-------------------	-----	--------	-----	-----

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

<code>XMM3</code>	0.5	0	320	255
-------------------	-----	---	-----	-----

Y listo, todo terminó como queríamos.

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

4.3. ASM2

En la segunda implementación utilizamos fuertemente la primera.

Lo que hicimos en ambas rutinas de conversión es bastante directo, siguiendo los algoritmos proveídos por la cátedra. El mayor problema que tuvimos en ambas rutinas fue que tenemos que cargar constantemente constantes que vamos a usar durante los procesos. Esto hace que la ejecución de las rutinas de conversión sea mucho mas lenta de lo que podría ser si tuviéramos más registros para guardar las constantes que usamos todo el tiempo.

Este es definitivamente el factor más limitante de nuestra implementación y esperamos que impacte fuertemente en el rendimiento.

4.4. Experimentación

Al igual que en los anteriores filtros, esperábamos que el rendimiento de nuestra implementación en assembler sea más rápida que la de C. Sin embargo los resultados nos dijeron lo contrario.

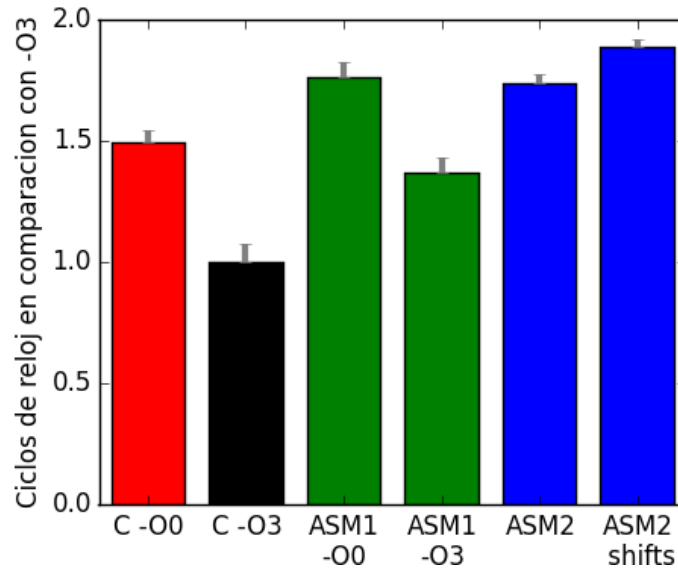


Figura 10: Comparación de la cantidad de ciclos de reloj utilizada por diferentes variantes de hsl. Para tener una medida absoluta: la cantidad de ciclos de reloj promedio de C -O3 es 31,620,627. Tamaño de la muestra: 200 imágenes de 16×10^4 píxeles. Se indica el mínimo con la barra y el promedio con una línea gris.

Los resultados fueron devastadores, dado que no los esperábamos. Al igual que el resto de los filtros, el comportamiento de los algoritmos es lineal sobre la cantidad de píxeles, pero la pendiente de nuestras implementaciones de assembler es mayor que las de C.

Por esta razón nos propusimos a hacer un amplio análisis de la situación, para poder descubrir la razón del bajo rendimiento.

La principal limitación de nuestra implementación de assembler son, claramente, los accesos a memoria. En cada rutina (de ambas implementaciones) debemos cargar muchas constantes que usaremos a lo largo de las distintas cuentas que debemos hacer. Esto es muy caro para nosotros, dado que los accesos a memoria son de lo mas limitante en lo que concierne a la performance. A esto atribuimos principalmente nuestro pobre desempeño frente a la implementación de C. Veremos más adelante otras circunstancias que apoyan esta teoría.

Habiendo dicho esto, también es cierto que una vez que pedimos un dato de memoria, este debería guardarse en la cache, siendo su acceso mucho más rápido. Pero la cache sigue siendo aún mas lenta que los registros del procesador, por lo que acceder a memoria (por mas que sea cache) sigue siendo un factor limitante.

Al ver los resultados, nos propusimos cambiar la implementación de assembler. Nuestra primera versión utilizaba muchos shifts, y dado que en una clase vimos que el desempeño de los shifts era peor que el desempeño de los shuffles, decidimos probarlo. El resultado, como se ve en la primera imagen, no fue del todo el esperado. Sí, hubo una mejora de performance, pero no fue significativa, ni nos permitió acercarnos a la implementación en C, nuestro principal objetivo.

Como se ve en esta imagen claramente, este cambio solo nos permitió un magro 10 % de ganancia sobre la implementación de C.

Algo importante que notar es que esto no significa que usar shuffles es 10 % más rápido que usar shifts, si no que es bastante más rápido. Esto se debe a que nosotros hacemos el análisis del programa completo, si se analizan por separado las partes en las que esta decisión incumbe, se llega a speedups de hasta el 40 %, dependiendo de la computadora.

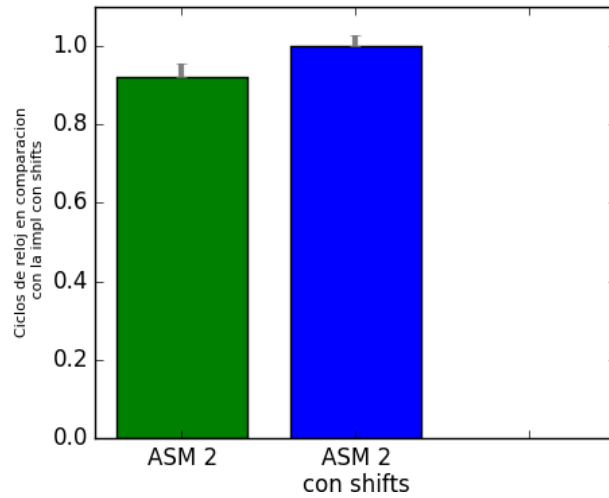


Figura 11: Comparación de la cantidad de ciclos de reloj utilizada por dos variantes de hsl2. Tamaño de la muestra: 200 imágenes de 16×10^4 píxeles. Se indica el mínimo con la barra y el promedio con una línea gris.

Por otro lado, nuestros algoritmos no dependen fuertemente en saltos condicionales, usan los necesarios, por lo tanto no creemos que este sea un factor limitante del rendimiento.

La siguiente limitación de nuestra implementación es la operatoria de las conversiones de RGB a HSL y viceversa. Esto se debe a que las operaciones que se deben hacer son largas y costosas, por mas que esten lo mejor optimizadas posibles.

A esto se le suma la dificultad de operar de a muchos píxeles juntos, dado que solo se puede en pequeñas partes. Por ejemplo, en hslTOrgb se podría calcular de forma paralela c , x , m para 4 píxeles así como la operatoria de los shuffles. Sin embargo, la complejidad del código crece enormemente, sumado a que la operación de rgbTOhsl y la de Suma no son así de simples de paralelizar. Por estas razones, optamos por no probar esta variante del código, pero es una opción a tener en cuenta.

Comentando partes de códigos, pudimos obtener una partición tentativa de cuanto tarda cada operación de la segunda implementación de hsl, algo que nos parece vital a la hora de analizar la performance. Los resultados fueron los siguientes:

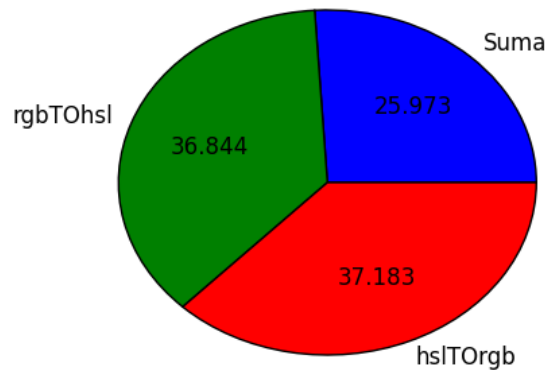


Figura 12: Análisis comparativo del porcentaje de tiempo dedicado a cada instancia. Tamaño de la muestra: 100 imágenes de 16×10^4 píxeles

Este gráfico debe interpretarse de la siguiente manera: el tiempo que se dedica en cada loop a hacer cada una de esas operaciones, es aproximadamente el indicado en el porcentaje.

A priori parecería que el proceso de Suma es más lento de lo que debería ser, lo cual es posible, dado que se cargan muchas veces las mismas cosas de memoria. Esto es inevitable en la primera implementación de hsl (dado que las llamadas a las funciones de C nos rompen todos los registros y no podemos guardar nada), mientras que podría ser evitable en la segunda implementación.

Sin embargo, viendo el código de nuestra implementación no parece que se pudiera mejorar demasiado (sin cambiar los algoritmos drásticamente), dado que los principales cambios que podrían realizarse son pasar de shifts a shuffles o extracts/inserts. Pero estas optimizaciones, como vimos anteriormente, no dan un speedup muy grande.

Lo último que queda inspeccionar para encontrar una razón por la cual el código de C le gana al nuestro, es mirando el output de assembler de GCC.

Lo primero que se nota en el código de assembler outputeado por GCC es el uso de saltos condicionales. Como el código de este algoritmo depende fuertemente en condicionales (ifs), el hecho de poder utilizar saltos inteligentemente como lo puede hacer un compilador (en vez de máscaras como usamos nosotros) puede dar ventajas.

Otra posible explicación es que el código generado por GCC carga menos cosas a memoria en cada loop (aunque no tantas). Esto se debe principalmente a que tiene un uso mas ajustado y eficiente de los registros, por lo tanto tiene algunos de sobra para guardar datos a los que va acceder seguido.

En conclusión, podemos ver que obtuvimos algunas respuestas en cuanto a las preguntas sobre el rendimiento en comparación de nuestro código de assembler vs. el código de C. Como resultado, podemos ver que nuestro déficit de rendimiento radica principalmente en la gran cantidad de accesos a memoria que realizamos y, en un segundo lugar, a que las cuentas y operatorias que realizamos no son óptimas. La mejor propuesta que se nos ocurre sería levantar de a 4 pixeles (aunque la operatoria se complejizaría mucho) o utilizar saltos condicionales para ahorrar trabajo (cosa que no hicimos porque todo debía hacerse con SSE).

Para concluir con la experimentación de hsl, proponemos un experimento muy interesante. A diferencia de los otros filtros, la implementación de C es susceptible a cambios drásticos en la performance dependiendo de la imagen que se usa. Esto se debe a que el filtro tiene muchos condicionales.

Entonces nos propusimos a diseñar 2 imágenes, una tal que siempre entre al primer condicional de todos los if's

(Suma, hslTOrgb, rgbTOhsl), y otra tal que siempre entre al último. Uno tiende a creer que, aunque las comparaciones son operaciones, su efecto sobre la performance es casi nulo. Sin embargo nos sorprendimos.

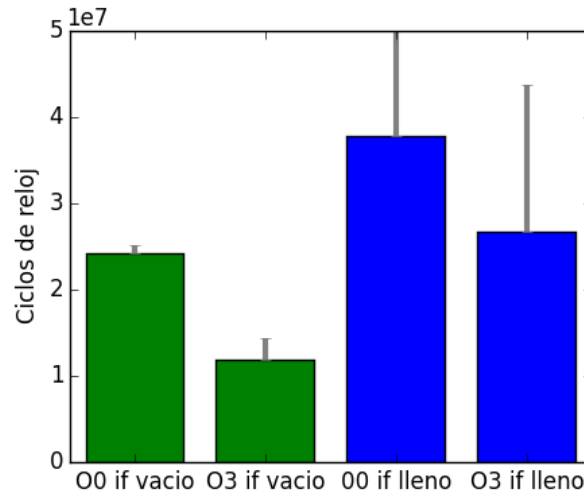


Figura 13: Análisis comparativo de la cantidad de ciclos de clock que tardan los programas en procesar una imagen de 16×10^4 píxeles. Se indica con una barra de error la media podada. Tamaño de la muestra: 100 imágenes.

if vacio quiere decir que los parámetros son todos 0, y la imagen es toda negra. Se puede verificar fácilmente que una tal imagen siempre entra en todos los primeros condicionales.

if lleno quiere decir que los parámetros son 99, 0, 0

5. Conclusión

La primer conclusión de este trabajo es que programar en Assembler, aunque difícil, es remunerador: si se programa usando el paradigma SIMD de buena manera, la performance que se obtiene es comparativamente alta. Por esta razón creemos que es importante conocer el paradigma y la tecnología SSE, dado que son la herramienta que nos permite hacer el tuneo más fino de nuestras aplicaciones. A pesar de esto, y como vimos en el caso de HSL, utilizar Assembler puede ser un arma de doble filo: el control cuasi-absoluto del sistema que obtenemos posiblemente termine resultando en algoritmos eficientes que escribiendo en un lenguaje de más alto nivel lo suficientemente optimizado y estudiado. En un punto, esto es esperable: la dificultad en poder estimar qué es lo que hacen todas las capas de abstracción del procesador a medida que escribimos nuestro código deriva en pérdidas de posibles caminos para optimizar nuestro algoritmo.

La segunda conclusión es que hay que medir con mucho cuidado en qué situaciones realmente conviene utilizar Assembler: el costo en horas de programar las rutinas en Assembler es mucho más alto que programarlas en C, y la diferencia en velocidad puede terminar resultándonos poco favorable. En sí, lo más conveniente es tratar de evitar Assembler a no ser que realmente veamos una oportunidad de optimización demasiado notoria.

La tercer conclusión es que hay que tener mucho cuidado con qué instrucciones utilizamos en nuestro código: instrucciones (o conjuntos de ellas) cuya semántica es equivalente pueden tener diferencias operacionales que causen una pérdida de eficiencia difícil de cuantificar en nuestros algoritmos. Más aun, como vimos en clase, estas diferencias operacionales pueden derivar en grandes diferencias de performance si variamos los modelos de procesador que utilizamos.

La cuarta y última conclusión, es que la complejidad temporal como la medimos es engañosa. Si bien podemos desde un punto de vista teórico calcular el orden de complejidad de un algoritmo, este no termina reflejando en la realidad cuál va a ser su performance: los procesadores modernos tienen demasiadas capas de abstracción que el modelo matemático no logra reflejar. Mecanismos como la caché y el pipelining efectivamente destruyen cualquier tipo de predictibilidad con respecto al tiempo de ejecución del algoritmo. De la misma forma, los compiladores podrían terminar transformando un algoritmo con cierta complejidad en uno con mejores características. En concreto, la medida de complejidad temporal sólo nos proporciona una medida de cómo va a crecer”, y no tanto de cuánto va a crecer”.