DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

SIMD

Organización del Computador II

Segundo Cuatrimestre de 2020

Grupo 16

Integrante	LU	Correo electrónico
Belgorodsky, Martin Yoel	183/19	martin.belgorodsky@gmail.com
Dondo, Ignacio	97/19	dondoignacio@gmail.com
Miguez, Tomás	94/19	tomasmiguez99@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

Resumen

En el siguiente trabajo se presentan implementaciones de filtros para imágenes en formato .bmp, implementados en *ASM* y aprovechando las instrucciones SSE. Además, se estudian comparativamente dichas implementaciones con implementaciones en C.

Índice

1. Introducción

A lo largo de este trabajo práctico, implementamos distintos filtros para imágenes en código ensamblador, utilizando el set de instrucciones SSE. El objetivo del mismo incluyó afianzar los conceptos de SIMD estudiados durante las clases y experimentar sobre las implementaciones para apreciar su funcionamiento y eficiencia. Los filtros que decidimos implementar fueron imagen fantasma, color bordes y reforzar brillo.

2. Desarrollo

2.1. Consideraciones generales

Las imágenes se importan de archivos .bmp, con su ancho siendo no menor a 16 píxeles y múltiplo de 8. En primera instancia, estas imágenes se procesan, para luego pasarse a los filtros en forma de matriz de enteros positivos. Esta matriz se corresponde con la imagen, siendo la posición [0][0] de dicha matriz el píxel superior izquierdo de la imagen. Cada píxel se compone de 4 valores de 1 byte, B (blue), G (green), R (red) y A (alpha); que se almacenan en el orden dado. Cabe aclarar que el valor alpha se setea fijo en 255 para todos los píxeles de cualquier imagen que sea el resultado de un filtro.

2.2. Imagen fantasma

Este filtro superpone en la imagen original la misma en escala de grises y al doble de tamaño, desplazada según lo indiquen los offsets pasados por parámetro. En nuestra implementación, este filtro trabaja con cuatro píxeles simultáneamente. La principal causa por la que no se aplica el filtro a más píxeles por iteración es la limitada cantidad de registros *XMM* que se poseen, ya que usamos varios para guardar máscaras. A continuación se detalla la implementación de este filtro:

Antes de recorrer la imagen, diez máscaras son guardadas en registros *XMM* para evitar tener que acceder a memoria demasiadas veces en cada iteración. Por lo que, se cuenta con seis registros *XMM* en cada ciclo para realizar operaciones. Por otro lado, los registros de propósito general nos servirán como contadores o punteros a las imágenes. Además, en nuestra implementación tendremos dos ciclos anidados: uno que recorrerá las filas y otro que, para cada fila, recorrerá las columnas. Esto se debe a que por cada píxel se tiene que calcular qué píxeles de la imagen original se le superpondrán, y esto depende de los contadores actuales para columnas y filas y de los desplazamientos en x e y.

Por cada iteración, primero se obtienen mediante registros de propósito general, los índices de los píxeles que se superpondrán según la siguiente operatoria:

$$ii = \frac{i}{2} + offsety \qquad jj = \frac{j}{2} + offsetx$$

Notar que, al ser $\frac{i}{2}$ el cociente de dividir i por 2 entonces $\frac{i}{2} = \frac{i+1}{2}$, suponiendo que i es par. Lo mismo ocurre con j. Por lo que al procesar cuatro píxeles por cada iteración, se superpondrá uno para los primeros dos píxeles y otro para los últimos dos.

Luego, mediante registros *XMM* se realiza el siguiente cálculo:

$$b = (imgOriginal[ii][jj].red + 2 * imgOriginal[ii][jj].green + imgOriginal[ii][jj].blue)/4$$

Figura 1: Registro *XMM* después de obtener los píxeles de memoria

El cálculo es realizado luego de hacer copias del registro que carga los valores desde memoria, visualizado en la figura 1. Después de realizar estas copias, se realizan distintos *shuffles* con las máscaras que

XMM1	(RRR1+2*GGG1+BBB1)/4	(RRR1+2*GGG1+BBB1)/4	(RRR0+2*GGG0+BBB0)/4	(RRR0+2*GGG0+BBB0)/4

Figura 2: Registro *XMM* después de realizar el cálculo

fueron guardadas en registros *XMM* al inicio de la función. Luego, los valores de los registros fueron convertidos a valores de punto flotante de precisión simple mediante la instrucción *CVTDQ2PS* para realizar el cálculo de b. En el registro *XMM1* guardaremos estos valores, como se puede ver en la figura 2.

Más tarde, se cargan desde memoria los valores de los píxeles que serán superpuestos por los otros, es decir, se cargan desde memoria los píxeles actuales. Luego de aplicar *shuffles* con distintas máscaras, se convierten los valores de los registros a punto flotante de precisión simple, para poder multiplicarlos por $0{,}9$ y además, se dividen empaquetadamente los valores de *XMM1* por 2. Luego, se vuelven a convertir los valores de los registros *XMM* a enteros de 32 bits con la instrucción *CVTPS2DQ* y se empaquetan los registros mediante la instrucción *PACKSSDW* para tener valores de 16 bits y poder así realizar la siguiente suma saturada, con la instrucción *PADDSW*:

$$imgDestino[i][j].red = imgOriginal[i][j].red * 0{,}9 + \frac{b}{2}$$
$$imgDestino[i][j].green = imgOriginal[i][j].green * 0{,}9 + \frac{b}{2}$$
$$imgDestino[i][j].blue = imgOriginal[i][j].blue * 0{,}9 + \frac{b}{2}$$

Luego, se vuelven a empaqutar los registros *XMM* para que contengan los valores del último cálculo en enteros de 8 bits con la instrucción *PACKUSWB* y se vuelven a realizar *shuffles* con otras máscaras para acomodar los valores de los registros de la siguiente manera:

XMM0	DST.A3	0	0	0	DST.A2	0	0	0	DST.A1	0	0	0	DST.A0	0	0	0
XMM2	0	DST.R3	0	0	0	DST.R2	0	0	0	DST.R1	0	0	0	DST.R0	0	0
XMM3	0	0	DST.G3	0	0	0	DST.G2	0	0	0	DST.G1	0	0	0	DST.G0	0
XMM4	0	0	0	DST.B3	0	0	0	DTS.B2	0	0	0	DST.B1	0	0	0	DST.B0

Figura 3: Registros *XMM* después de aplicar las máscaras

Por último, mediante la instrucción *POR* se juntan todos los resultados en un único registro *XMM*:

XMM0	DST.A3	DST.R3	DST.G3	DST.B3	DST.A2	DST.R2	DST.G2	DTS.B2	DST.A1	DST.R1	DST.G1	DST.B1	DST.A0	DST.R0	DST.G0	DST.B0

Figura 4: Registro *XMM* luego de realizar toods los cálculos listo para cargar a memoria

Para finalizar, se guardan los valores a memoria con la instrucción *MOVDQU* y se actualizan registros de propósito general utilizados como contadores.

2.3. Color bordes

Este filtro funciona de la siguiente manera, tomamos los 8 píxeles vecinos al que estamos procesando, calculamos la diferencia de los de arriba de nuestro píxel con los de abajo, columna a columna; la diferencia de los de la izquierda con los de la derecha, fila a fila; tomamos el valor absoluto de esas diferencias y finalmente sumamos todos esos valores con saturación. Dicho resultado es el valor del píxel en la imagen destino. Como tenemos que tomar los 8 vecinos de cada píxel para procesarlo, esto no se puede hacer en los bordes de la imagen, entonces ponemos un borde blanco de 1 píxel en el resultado. En el caso puntual de nuestra implementación procesamos de a 2 píxeles por lo que si los píxeles a procesar son P1 y P2, cargamos los siguientes píxeles en cada iteración:

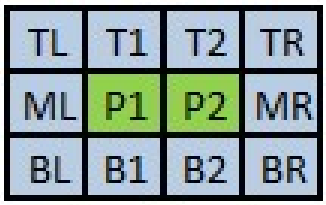


Figura 5: Valores que tenemos que cargar para cada iteración con respecto a la pos. de P1 y P2.

Cabe aclarar que cada casilla del anterior gráfico representa 4 bytes, uno por canal, por ejemplo, la casilla TL en realidad se subdivide en $|TLA|TLR|TLG|TLB|$.

Con esta notación, lo que queremos obtener finalmente es, en la posición de P1 en el resultado, el valor
$SAT(|T1 - B1| + |T2 - B2| + |TL - BL| + |TL - T2| + |ML - P2| + |BL - B2|)$
y en la posición de P2 en el resultado, el valor
$SAT(|T2 - B2| + |TR - BR| + |T1 - B1| + |T1 - TR| + |P1 - MR| + |B1 - BR|)$

Como debemos cargar 4 píxeles de 32 bits cada uno, podemos poner cada fila en un registro xmm (128 bits), asi que hacemos eso. Para esto traemos los 128 bits de memoria con MOVDQU.

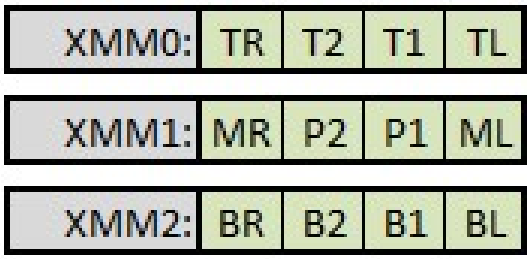


Figura 6: Registros xmm después de la primera carga.

Para alguna de las operaciones, vamos a necesitas mas de 8 bits por CANAL, debemos extender cada canal de cada píxel a 16 bits. Para esto usamos una combinación de MOVDQA, PSRLDQ y PMOVZXBW como se muestra en el código.

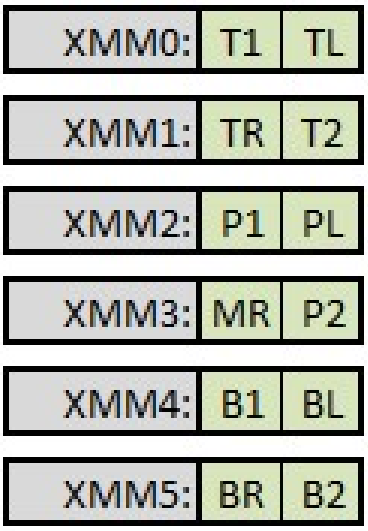


Figura 7: A partir de ahora cada espacio representa 64 bits.

Una vez obtenido esto, se puede pasar a calcular los valores buscados, que almacenamos en el registro a XMM15. Para ello realizamos las operaciones detalladas en los comentarios del código, hasta obtener el valor buscado de P2 en la parte alta de XMM15, y el valor buscado de P1 en la parte baja, aunque todavía no están saturados a 8 bits ya que los extendimos a 16 previamente (linea 72 a 101). Es por esto que utilizamos la operación PACKUSWB para volver nuestros 8 registros de 16 bits en 8 registros de 8 bits, que se almacenan en la parte baja de XMM15 (también en la alta, pero no nos interesa, linea 103). Finalmente, queremos que nuestros bits tengan el canal alpha seteado en 0xFF, para esto creamos una mascara de 128 bits, formada por 0xFF000000 4 veces, para hacer un or con XMM15, y así setear los bit correspondientes al valor adecuado (lineas 105-106). Ahora que obtuvimos el valor buscado, lo almacenamos en memoria, pero son solo 8 bytes que guardamos, los 2 píxeles resultado, por lo que usamos la instrucción MOVQ (linea 108).

2.4. Reforzar brillo

Este filtro se basa modificar la intensidad del brillo de la imagen, comparando un cálculo dentro de cada píxel contra un umbral dado. En base a esta comparación se determinará si se mantiene, aumenta o disminuye el brillo, estos coeficientes también son dados como parámetros. A continuación se enseña el cálculo y la comparación contra el umbral que se debe realizar para cada píxel de la imagen:
Consideremos el umbral dado como $U = [u_1, u_2]$. Los valores para aumentar o disminuir el brillo como v_+ y v_- respectivamente. Llamo C a cada color del píxel original, y C_r a cada color del píxel resultado.

$$x = (B + 2 \times G + R)/4$$

$$C_r = \begin{cases} C & \text{si } u_1 < x < u_2 \\ C - v_- & \text{si } x < u_1 \\ C + v_+ & \text{si } x > u_2 \end{cases} \qquad \forall\, C \in \{B, G, R\} \wedge \forall\, C_r \in \{B_r, G_r, R_r\}$$

Algoritmo: Para implementar el algoritmo se realiza un ciclo que aplica el filtro en 8 píxeles por vuelta. Antes de esto, se calcula la cantidad de píxeles que tiene la imagen dada, se guardan los punteros de *src* y de *dst*, se inicializa un contador de píxeles *index*, y por otro lado se guardan de forma extendida en registros *XMM* separados cada uno de los parámetros que determinan el umbral y el aumento o disminución de brillo. Para esto último, se guarda cada parámetros en un registro *XMM*, luego se satura convirtiendo a unsigned int de tamaño word (*PACKUSDW*) para los umbrales y a byte (*PACKUSWB*) para los sumadores/restadores de brillo (ya que los elementos con los que se compararán estarán guardados de esta forma), y luego se aplica una máscara que con la instrucción *PSHUFB* repite estos valores a lo largo de todo el registro.

En cada vuelta del ciclo primero se obtienen los primeros 8 píxeles, guardándolos en dos registros *XMM* (4 píxeles de 32 bits en cada uno). Luego comienza la etapa en la que se obtiene el cálculo *x* con los colores de cada píxel. Se hacen dos copias para cada registro, ya que debo preservar el registro original, y a cada copia les aplico máscaras diferentes con *PSHUFB* para reordenar los datos (Figuras 8 y 9) y luego sumarlos cómodamente.

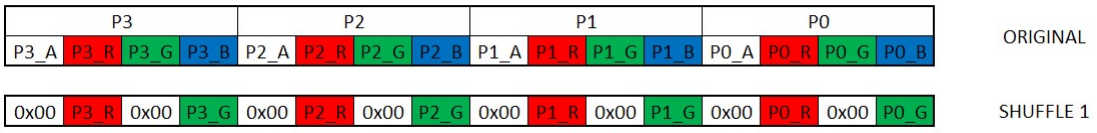


Figura 8: Máscara 1 - *mask_pshufb1*

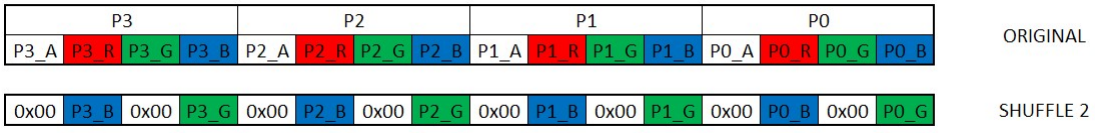


Figura 9: Máscara 2 - *mask_pshufb_2*

Posteriormente, se realiza una suma (*PADDW*) entre los shuffle de cada registro (Figura 10), obteniendo una suma parcial de cada píxel, la que luego completaremos con una suma en línea (*PHADDW*). En este último paso involucramos a ambos grupos de píxeles, ya que la instrucción nos lo permite, ahorramos pasos y aprovechamos el pipeline (Figura 11), nos quedan todos los valores *x* de cada uno de los 8 píxeles en un solo registro. Por otro lado, tendremos que luego dividir por 4 todos estos valores, para eso utilizamos un shift a derecha para word dentro del *XMM* (*PSRLW*).

Una vez obtenidos los valores, procedemos a comprarlos con los umbrales. Primero resguardamos el umbral inferior, ya que lo sobre-escribiremos en la comparación. Y realizamos *PCMPGTW* para cada umbral (en el caso del inferior, lo realizamos a la inversa), que es una comparación a word, y nos entrega *0xFFFF* o *0x0000* para cada píxel, si es mayor el valor o no, respectivamente.

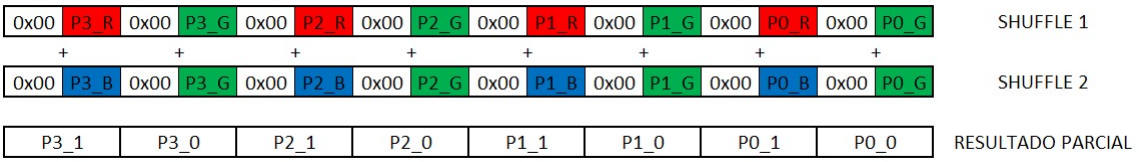


Figura 10: Resultado parcial de la suma

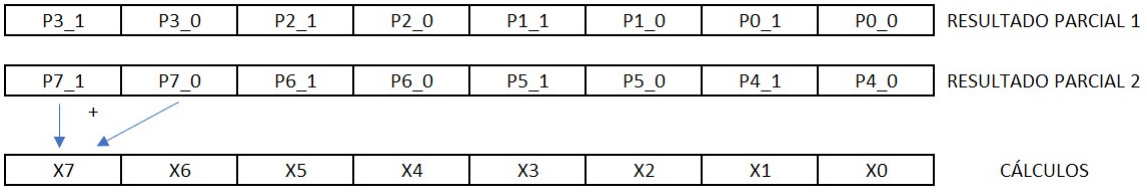


Figura 11: Resultado de la suma final

Luego de realizar la comparación, tendremos que transformarlos en una máscara para cada grupo de píxeles, y en base a eso sumar o restar. Esto nos sirve, ya que tenemos varios registros dentro, y será necesario aplicar distinta lógica para uno. Con una máscara se distribuyen los primeros 4 valores y con el otro, los segundos 4, extendidos a los 24 bits que deben ser modificados del píxel (recordemos que alpha no debe ser alterado). Finalmente sumaremos y restaremos los píxeles que originalmente preservamos contra estas máscaras generadas, dónde no corresponde sumar y/o restar habrá ceros y por lo tanto permanecen igual los valores.

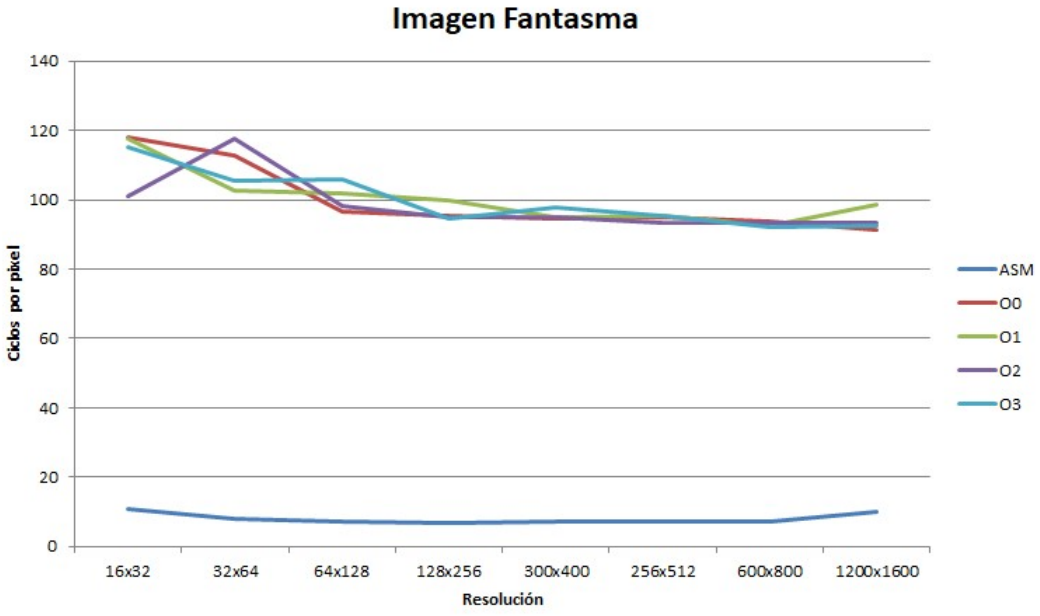
Estos valores se guardarán en *dst* y se continuará con el ciclo, avanzando los punteros de *src* y *dst*, y el *index* que se compara en cada ciclo con la cantidad de píxeles de la imagen (la cual obtuvimos al principio).

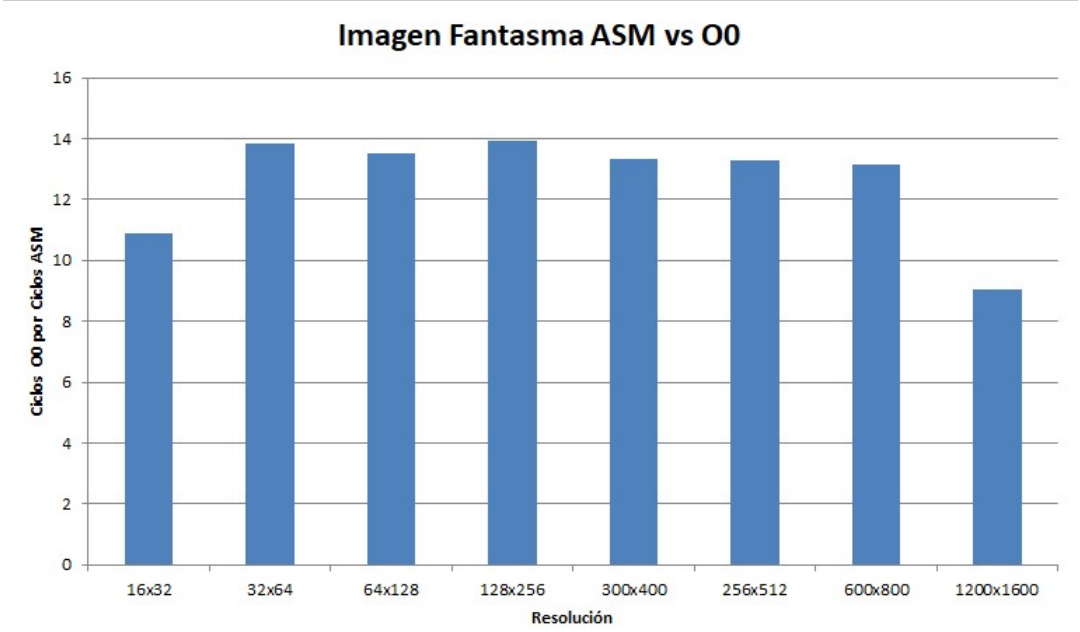
3. Comparación

3.1. Consideraciones generales

Vamos a comparar cada algoritmo contra C con distintos grados de optimización (O3, O2, O1 y O0), además, vamos a realizar dicha comparación para imágenes con distinta cantidad de píxeles (32x16, 64x32, 128x64, 256x128, 400x300, 512x256, 800x600 y 1600x1200). Cada medición es el promedio de 100 corridas. Para cada filtro, tenemos dos gráficos, uno en el que se comparan los ciclos de procesador por cantidad de píxel de cada implementación, y otro en donde se grafica cuantas veces mas rápido es en ejecutarse la implementación en *ASM* que en C.

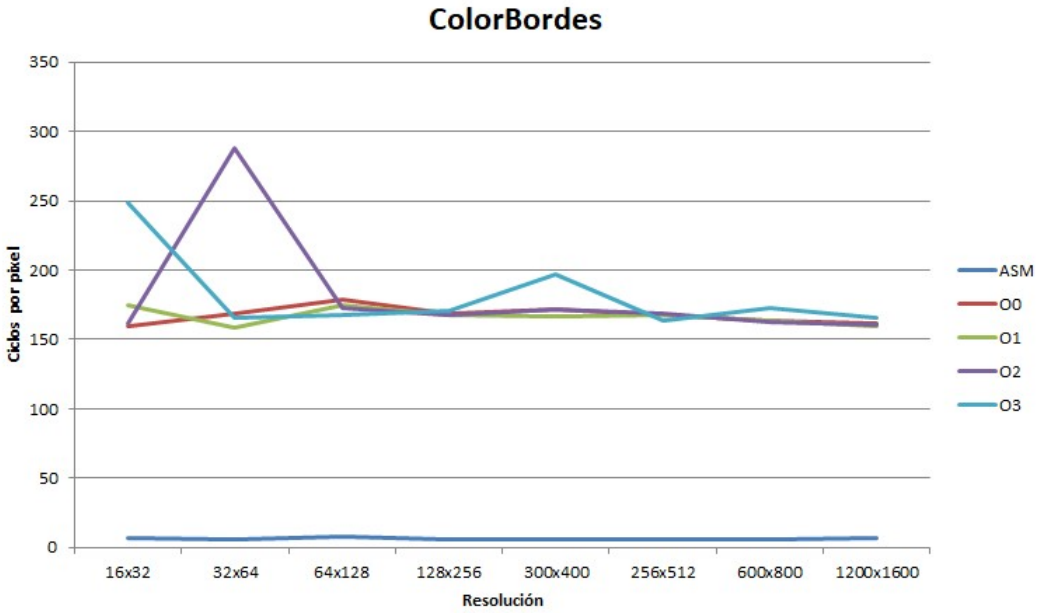
3.2. Imagen fantasma

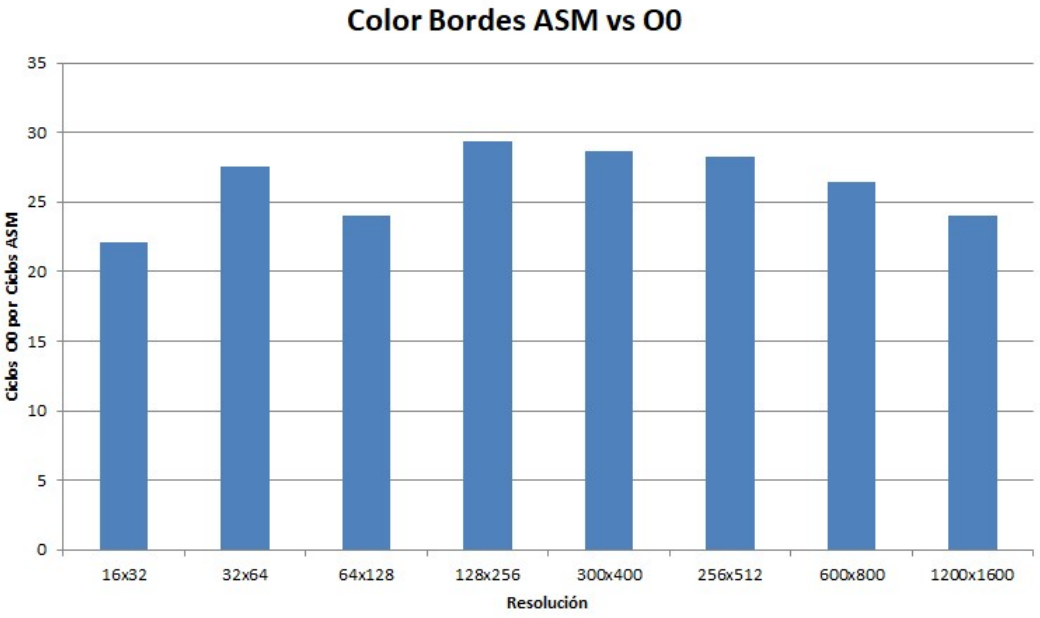




En promedio, la implementación de *ASM* de imagen fantasma es 12,6 veces mas rápida que la de C.

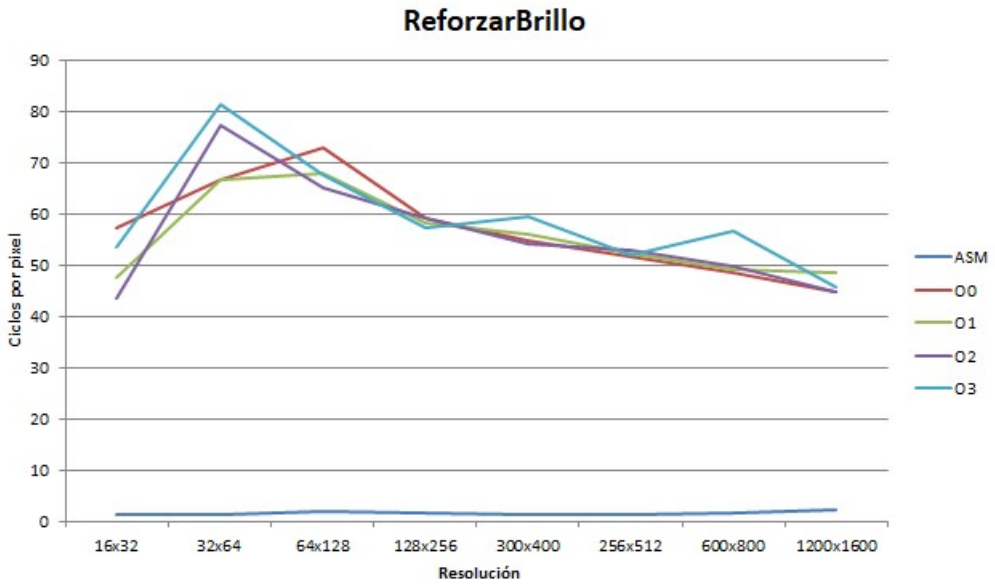
3.3. Color Bordes

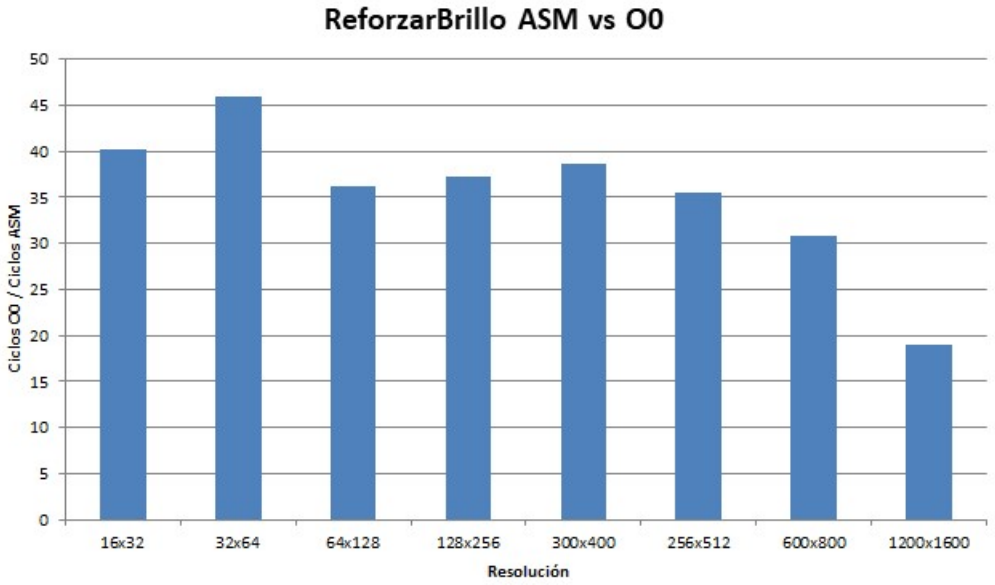




En promedio, la implementación de *ASM* de color bordes es 26,3 veces mas rápida que la de C.

3.4. ReforzarBrillo





En promedio, la implementación de *ASM* de reforzar brillo es 35,5 veces mas rápida que la de C.

4. Experimentos y resultados

4.1. Instrucciones ejecutadas por píxel en *Reforzar Brillo*

Para evaluar la cantidad de píxeles que se procesan por ciclo, para la implementación en ASM contra la de C, hipotetizamos en que la implementación realizada en ASM realiza menos instrucciones por píxel de lo que realiza la de C, ya que nuestra implementación fue basada en ASM, a diferencia de la implementación en C, y por más que tenga un optimizador para reducir saltos y reducir líneas, no suponemos que esto sea suficiente para superar a ASM. Luego concluiremos, en base a los experimentos, si esto se puede mejorar, y en ese caso, si mejora la performance del algoritmo.

Para comenzar con este análisis compilamos el código de C con *Compiler Explorer* y obtenemos el código en ASM, este proceso lo realizamos sin optimización forzada del compilador y luego con optimización *O3*, con estos resultados podremos hacer una comparación más sencilla contra ASM (este código se puede ver en el anexo, *ReforzarBrillo_sinOptimizacion.asm* y *ReforzarBrillo_O3.asm*). Notamos que el código de C traducido a ASM procesa de a un píxel, aún con optimización *O3*. Como se procesan distinta cantidad de píxeles por instrucción en las implementaciones, es necesaria una distinción en este apartado. En nuestra implementación de ASM todas las instrucciones se realizan con 4 píxeles dentro, y un caso con 8 píxeles. Para calcular la cantidad de instrucciones por píxel, contaremos las instrucciones dentro de cada vuelta de ciclo, y al final la dividiremos por la cantidad de píxeles que se procesa en cada vuelta.

Tomamos inicialmente el caso de la implementación C optimizada en *O3*, y consideramos el peor caso para todos los ciclos, y nos da que se realizan 38 instrucciones en cada ciclo (y por lo tanto por cada píxel), pero hay que considerar que además se realizan 3 *call* a funciones de C, las tomamos como 1 instrucción por practicidad.

Por otro lado, nuestra implementación nativa de ASM realiza 41 instrucciones por ciclo, pero en cada vuelta se procesan 8 píxeles, por lo que serían menos, quedarían 5,125 instrucciones por píxel.

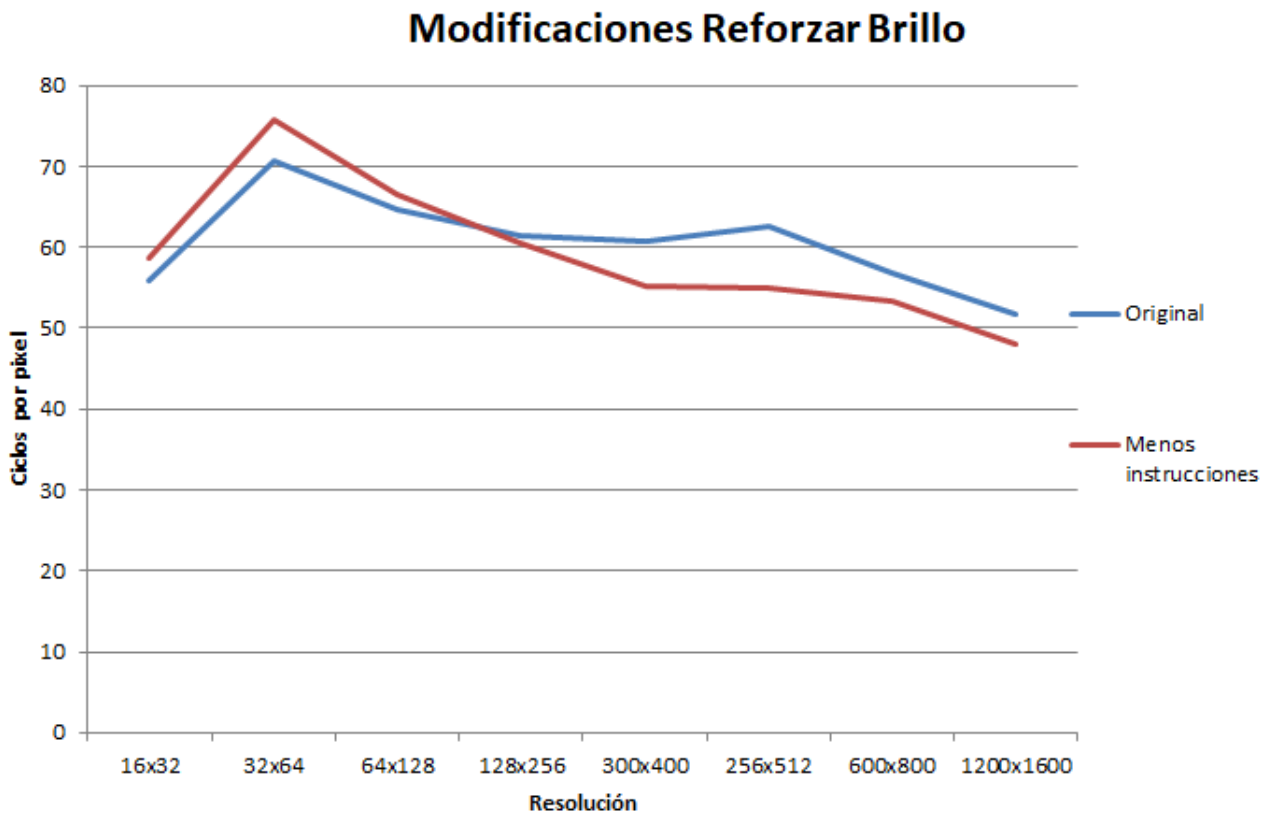


Figura 12: Comparación de las implementaciones en *C* para *Reforzar Brillo*

Vemos una gran diferencia entre ambos. Vamos a modificar el código de C para ver si es posible obtener una versión compilada en ASM con menos instrucciones por ciclo. Para esto modificamos el algoritmo, quitando una de las 2 comparaciones que se hacen para el peor caso, de esta forma obtenemos 29 líneas por ciclo para el peor caso. Una leve mejora con el anterior caso. Realizamos los tests de eficiencia que se habían realizado previamente en 7 y los comparamos con el algoritmo original de C, para ver si nos da una mejora final en la performance.

Como podemos observar en el gráfico de la Figura 4.1, hay una leve mejora en la performance para imágenes más grandes, logrando reducir el tiempo de ciclos del procesador por píxel.

Concluímos en que no es posible disminuir la cantidad de instrucciones por píxel en la implementación en C, ya que va de a un píxel, y la implementación de ASM está en ventaja con ese aspecto. Incluso se podría haber reducido la cantidad de líneas en ASM, si se consideran de antemano las posiciones en las que cae cada registro luego de aplicar instrucciones de sumas, y evitamos algunas aplicaciones de máscaras de reordenamiento.

4.2. Accesos a memoria en *Imagen Fantasma*

En el segundo experimento intentaremos descubrir a qué se debe la diferencia de performance entre las implementaciones en *C* y en *ASM* para el filtro *Imagen Fantasma*. Lo que queremos ver es si las limitaciones de performance de la implementación en *C* son causadas por los accesos a memoria.

Por un lado, la implementación en *ASM* de *Imagen Fantasma* realiza en un primer momento diez accesos a memoria para guardar en registros *XMM* las máscaras que serán utilizadas más tarde. Luego, para cada iteración realiza tres accesos a memoria: uno para obtener los píxeles que se superopondrán a los actuales, otro para obtener los píxeles actuales y por último otro para guardar en memoria los píxeles resultado. Como itera $width \times height$ veces entonces en total realiza $3 \times width \times height + 10$ accesos a memoria.

Por otro lado, las implementación en *C* de *Imagen Fantasma* realiza nueve accesos a memoria por cada iteración y, como itera $height \times width$ veces entonces en total realiza $9 \times height \times width$ accesos a memoria.

Luego de realizar unos cambios en el código en *C* para realizar menos accesos a memoria, pasando de nueve accesos a solo tres esperamos que la cantidad de ciclos que tarde la nueva versión de la implementación en *C* se reduzca un poco ya que la cantidad de accesos a memoria se redujo por tres y que los accesos a memoria no son tan eficientes.

Para este experimento corrimos el código para cada imagen 200

Como podemos ver en la figura 13, los cambios realizados en la implementación en *C* no son tan notorios. Para imágenes de baja resolución se obtuvieron mejores resultados con la implementación original. Creemos que esto se debe a que al ser una imagen de pocos píxeles entonces el costo de guardarnos en cada iteración una copia de cada píxel es más caro que realizar más accesos a memoria. A su vez, se ve que el costo por píxel va decreciendo a medida que aumenta la resolución de la imagen para la implementación modificada pero que no dista mucho del que se tiene con la implementación original. Al haber obtenido resultados similares con ambas implementaciones podemos concluir según el experimento realizado que los accesos a memoria no causan las diferencias de performance entre *C* y *ASM*.

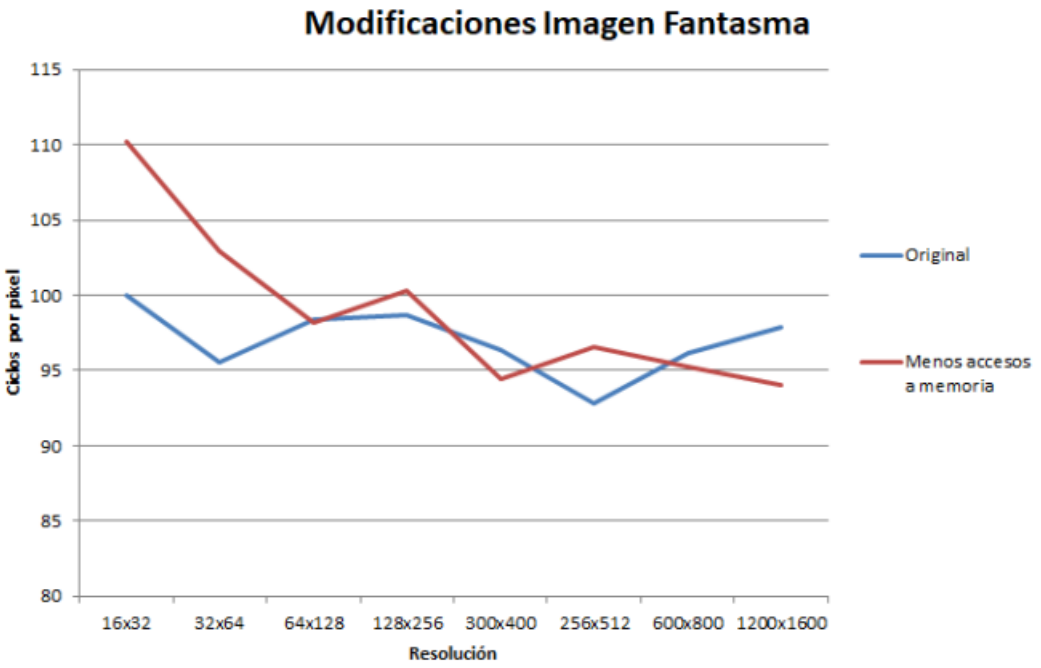


Figura 13: Comparación de las distintas implementaciones en *C* para *Imagen Fantasma*

5. Conclusión

A la hora de implementar un filtro neighbour 8, con canales de 8 bits, los registros *XMM* y las instrucciones SIMD se adaptan a la perfección a los requerimientos. La naturaleza matricial de las imágenes vuelve sencilla la carga de píxeles pertenecientes a N8, y los 128 bits disponibles por registro son justo suficiente para cargar una fila del vecindario de 2 píxeles adyacentes. Además, en caso de no necesitar píxeles adyacentes, y solo trabajar con valores del píxel actual, se pueden procesar hasta 8 píxeles a la vez por registro. Comparativamente con C, es claro que el tiempo de ejecución es mucho menor (no menos de 10 veces menor). Sin embargo el tiempo de desarrollo y debuggeo que requieren las aplicaciones *ASM* es significativamente mayor. Esto puede deberse en gran medida a nuestra experiencia en el área, pero es innegable, y razonable, que la cantidad de lineas de código es significativamente mayor. Todo esto nos lleva a concluir que implementar funciones en *ASM* debe estar reservado para cosas que tienen un requerimiento elevado de eficiencia (por ejemplo, si quiero aplicarle filtros a videos en tiempo real), ya que el costo en horas hombre para producir dicho código es mucho mas elevado que en otros lenguajes.