DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Procesamiento de imagenes mediante instrucciones SIMD

Organización del computador 2

Primer Cuatrimestre de 2017

Grupo Estrellitas

Integrante	LU	Correo electrónico
Hofmann, Federico	745/14	federico2102@gmail.com
Lasso, Andrés	714/14	lassoandres2@gmail.com
Berríos Verboven, Nicolas	46/12	nbverboven@gmail.com

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Ciudad Universitaria - (Pabellón I/Planta Baja)

Intendente Güiraldes 2610 - C1428EGA

Ciudad Autónoma de Buenos Aires - Rep. Argentina

Tel/Fax: (++54 +11) 4576-3300

http://www.exactas.uba.ar

Índice

Parte I
Introducción

En el siguiente trabajo se buscó conseguir un primer acercamiento a las instrucciones SIMD, las cuales se utilizan en el procesamiento de datos de manera simultanea. Para ello implementamos distintos filtros para imágenes tanto en el lenguaje de programación C como en Assembler. Una vez implementados procedimos a comparar ambas versiones de cada filtro buscando así poder conocer mejor las ventajas y desventajas de cada una.
Ademas, realizamos una serie de experimentos con cada filtro con el objetivo de realizar un análisis lo mas completo posible para de esta manera lograr justificar por que una implementación funciona mejor que la otra.
Por ultimo, intentaremos exponer todos los datos de la manera mas detallada posible, describiendo paso a paso los algoritmos implementados, explicando de forma completa la experimentación realizada y mencionando siempre todos los detalles que consideremos relevantes para el desarrollo de este trabajo y la comprensión del mismo por parte del lector.

Parte II
Implementaciones

Las imágenes utilizadas en este trabajo están en formato BMP y cada píxel de las mismas esta compuesto por 4 componentes de 1 byte cada una, ocupando así 4 bytes cada píxel. Dichas componentes son A, B, G y R, representando la componente de transparencia, el color azul, el verde y el rojo respectivamente. Cada componente esta representada por un entero sin signo, con lo cual sus posibles valores están entre 0 y 255. El ancho de las imágenes es siempre superior a 16 píxeles y múltiplo de 4 píxeles. En ningún caso se alterará el valor de la componente A. Las imágenes son consideradas como matrices.

1. Convertir RGB a YUV y YUV a RGB

Este filtro realiza una transformación lineal de las componentes RGB, llevándolas así a un nuevo espacio donde las componentes obtenidas serán llamadas YUV.

De forma análoga, al convertir YUV a RGB se realiza el procedimiento inverso para volver al subespacio anterior.

En ambos casos la expresión se debe realizar algún tipo de saturación para que el resultado de la operación se encuentre entre 0 y 255. La manera en la que decidimos lograr esto fue quedándonos con el mínimo entre el resultado de la transformación lineal y 255, y luego buscando el máximo entre dicho resultado y 0.

Lo que se hace entonces es ir recorriendo toda la imagen componente por componente, aplicándoles su transformación lineal correspondiente y escribiendo dichas componentes en la imagen destino; imagen que obtenemos como resultado.

1.1. Pseudo código del algoritmo en C

Algoritmo 1 Pseudocódigo de la implementación en C de rgb2yuv

```
j = 0
while j < cantPixeles do
    R = pixel[j].R
    G = pixel[j].G
    B = pixel[j].B
    pixel[j].Y = Satura(((66 * R + 129 * G + 25 * B + 128) >> 8) + 16)
    pixel[j].U = Satura(((−38 * R − 74 * G + 112 * B + 128) >> 8) + 128)
    pixel[j].V = Satura(((112 * R − 94 * G − 18 * B + 128) >> 8) + 128)
    j + +
end while
```

Algoritmo 2 Pseudocódigo de la implementación en C de yuv2rgb

```
j = 0
while j < cantPixeles do
    Y = pixel[j].Y
    U = pixel[j].U
    V = pixel[j].V
    pixel[j].R = Satura((298 * (Y − 16) + 409 * (V − 128) + 128) >> 8)
    pixel[j].G = Satura((298 * (Y − 16) − 100 * (U − 128) − 208 * (V − 128) + 128) >> 8)
    pixel[j].B = Satura((298 * (Y − 16) + 516 * (U − 128) + 128) >> 8)
    j + +
end while
```

1.2. Descripción de implementación en assembler

Para la implementación del filtro convert RGB to YUV en este lenguaje decidimos procesar de a 1 píxel a la vez. Los registros en este modelo son de 128 bits, con lo cual un píxel ocupa solo un cuarto. La decisión partió del hecho de que, al realizar las operaciones de transformación lineal a cada componente de cada píxel, se precisa de al menos 32 bits por componente para evitar perdida de información, dado que cada componente puede tomar como máximo el valor 255 (máximo posible en 8 bits), y al sumarlo y multiplicarlo con otros valores, 32 bits serán suficientes para almacenar estos resultados, mientras que con 16 o con 8 no.

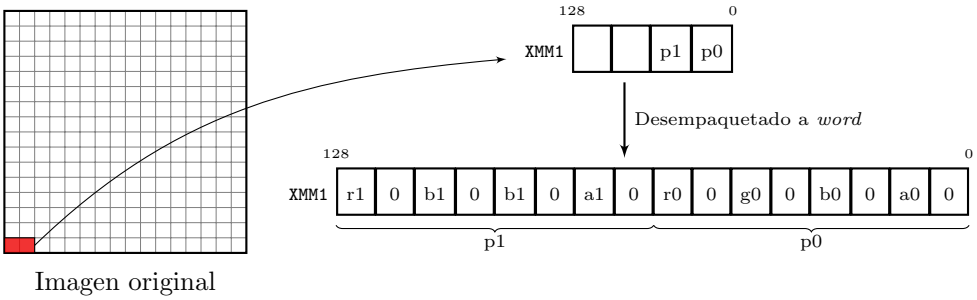


Figura 1: Movimiento y primer desempaquetado. Los de la imagen original (en rojo) se copian mediante la instrucción `MOVQ` a `XMM0`. Posteriormente, se desempaquetan las componentes de ambos de *byte* a *word* (`PUNPCKHBW` y `PUNPCKLBW`). Para esto último se utilizó un registro que contenía una máscara con 0 de forma de mantener el signo de R, G, B y A para futuras operaciones.

Para realizar las operaciones de la forma mencionada, se movieron dos píxeles de la imagen original a registros `XMM` y se desempaquetaron como se muestra en la figura 1. A continuación, se copió el resultado otras dos veces a diferentes registros y se aplicó una máscara para obtener en cada uno las componentes R, G y B por separado (Figura 2).

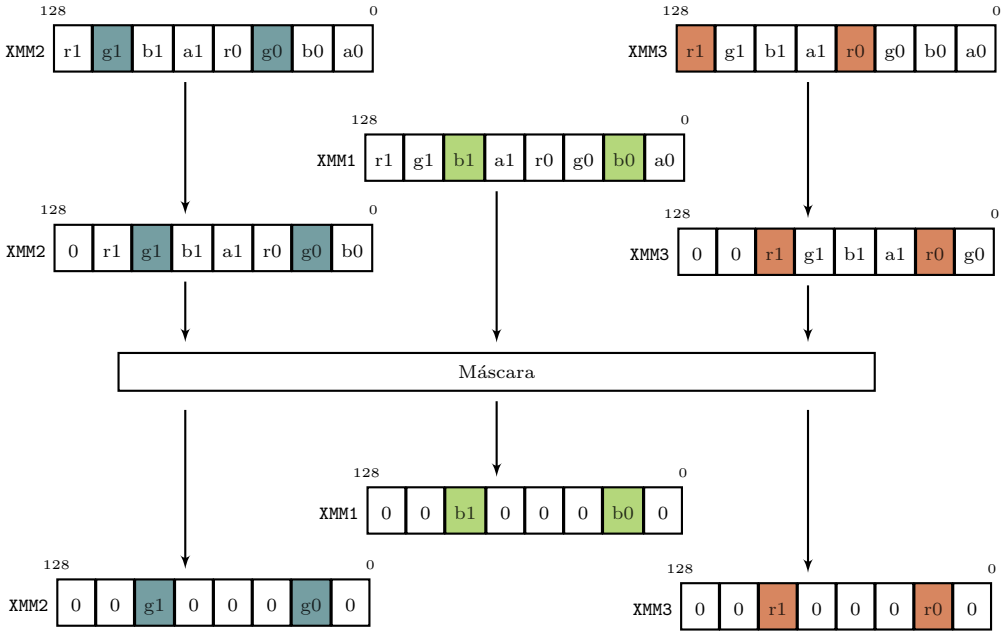


Figura 2: Obtención de componentes R, G y B. Se realiza un shift lógico de los registros `XMM2` y `XMM3` con las instrucciones `PSLLDQ` y `PSRLDQ` -respectivamente- para alinear las componentes R0, G0, R1 y G1 con sus correspondientes B del registro `XMM1` . A continuación, se realiza un `POR` de cada uno con una máscara que permite preservar únicamente las componentes deseadas en cada caso. Por último, se desempaquetan las partes altas y bajas de cada `XMM` de *word* a *doubleword* en registros separados (este proceso no se muestra aquí).

El siguiente paso consistió en generar las componentes Y, U y V mediante la aplicación de una transformación lineal a R, G y B realizando multiplicaciones y sumas entre los registros y máscaras con valores constantes (Figura 3).

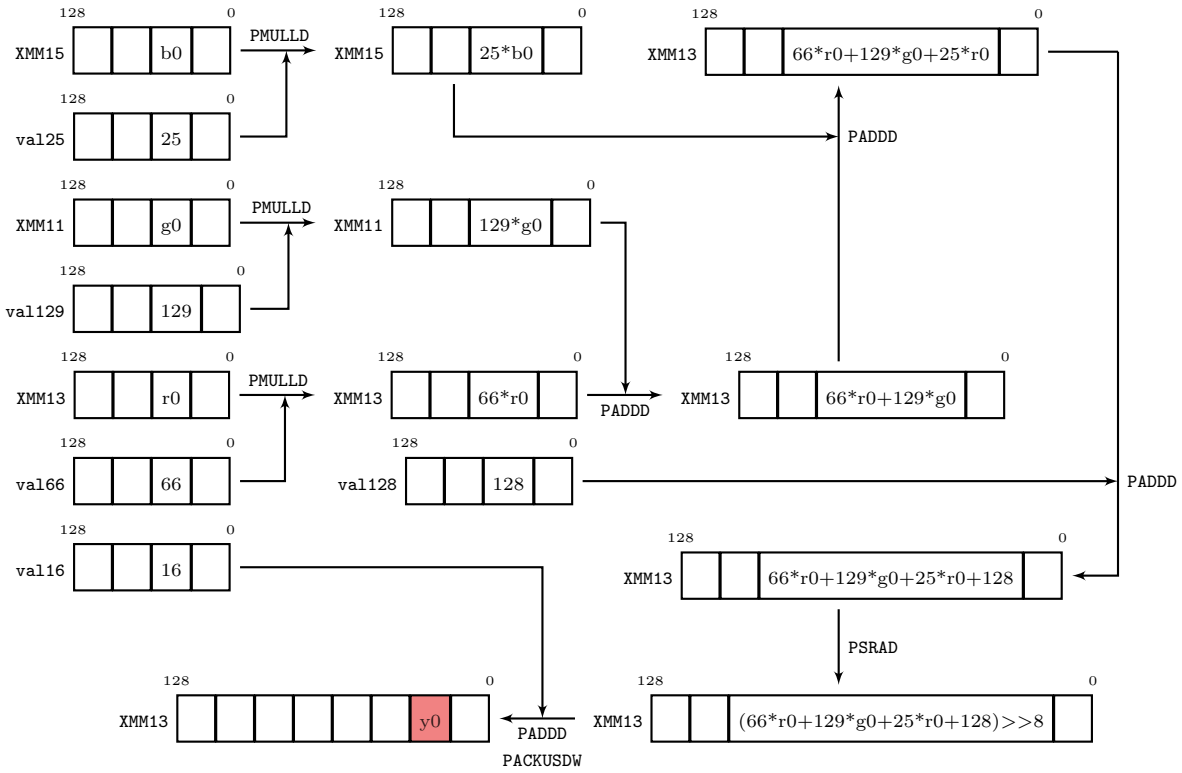


Figura 3: Esquema de las operaciones realizadas en el ciclo principal de *rgb2yuv* para obtener la componente Y a partir de R, G y B. Las instrucciones ubicadas en intersecciones son aquellas que trabajan con más de un registro, donde los caminos entrantes -a dicha intersección- corresponden a los operandos fuente y el saliente apunta hacia el operando destino. Los registros"que figuran como `val` son máscaras almacenadas en la memoria principal.

El ultimo paso consiste entonces en empaquetar a los tres registros de double word a word con la instrucción `PACKUSDW` y de word a byte con la instrucción `PACKUSWB`, las cuales realizan saturación sin signo, es decir que si los valores superaban 255 o eran menores a 0, entonces pasaban a ser 255 o 0. Después debimos volver a shiftear los registros para acomodar las componentes Y, U y V en las partes que les corresponde de cada registro. Después se juntó a las tres componentes en un solo registro y se copió la parte baja del registro resultante a su posición correspondiente en la imagen destino (Figura 4).

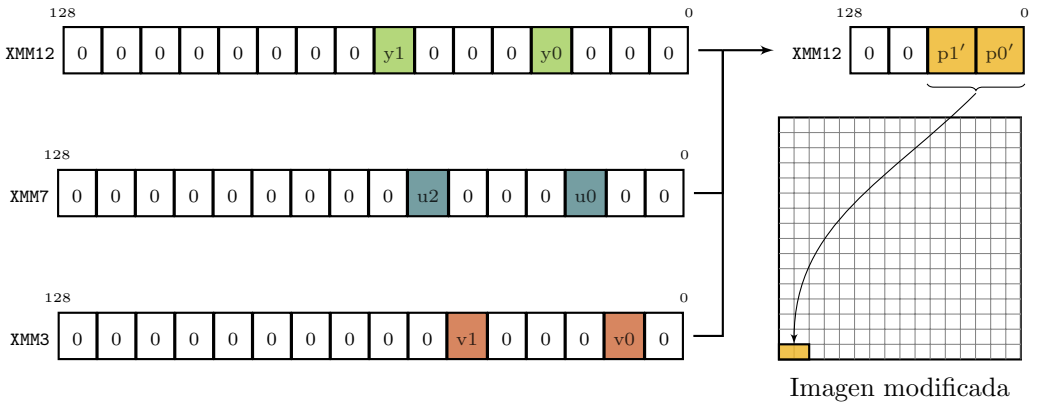


Figura 4: Fin de una iteración del ciclo principal de *rgb2yuv*. Se realiza un `POR` entre los registros que contienen las componentes Y, U y V de los dos píxeles modificados para obtener estos últimos en un mismo registro -`XMM12` en este caso- cuya parte baja se copia mediante un `MOVQ` a la posición indicada en la imagen destino.

Luego de realizar todos estos pasos se vuelve al principio y se realizan las mismas pasadas con los dos píxeles siguientes y así hasta haber recorrido toda la imagen.

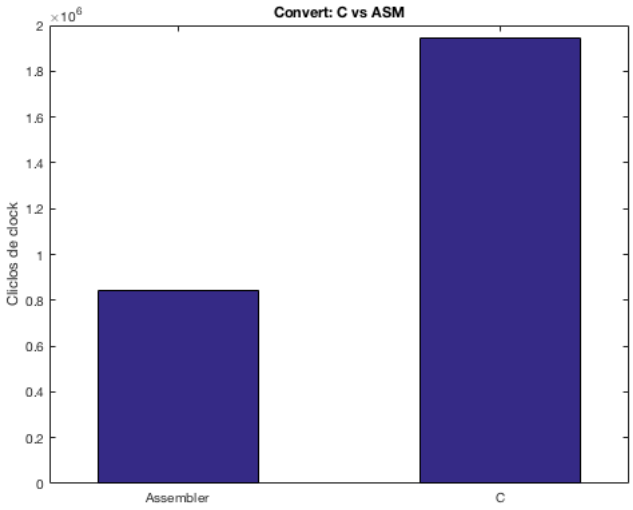
En el caso del filtro convert YUV to RGB el procedimiento realizado es el mismo, a diferencia de que esta vez el shift realizado en la transformación lineal es lógico, mientras que en el anterior filtro

era aritmético. Esto se debe a que en el anterior filtro se podían generar números negativos durante la transformación lineal y mediante ese shift evitábamos que un numero negativo pasara a ser positivo obteniendo así transformaciones erróneas.

1.3. Experimentación

1.3.1. Comparación C vs Assembler

El siguiente gráfico muestra la diferencia de ciclos de clock necesarios para completar el filtro $rgb2yuv$ entre la implementación del mismo en el lenguaje C y en Assembler. Los datos que se muestran los mínimos de 3000 mediciones distintas tomadas en de a 1000 en distintos momentos a cada implementación. A partir de ahora realizaremos este mismo procedimiento para todas las comparaciones medidas en ciclos de clock que realicemos en los distintos experimentos.



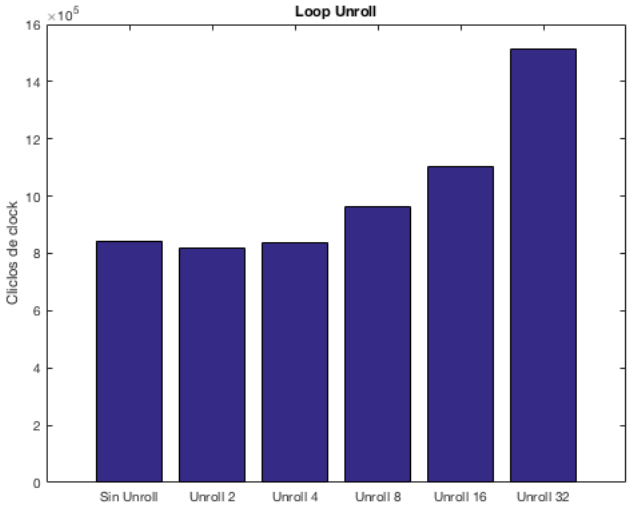
En el gráfico anterior podemos ver claramente como la versión del filtro implementada en Assembler tarda la mitad de ciclos de clock que la de C. Esto era de esperarse ya que justamente lo que buscamos mediante el uso de instrucciones SIMD es mejorar el tiempo de computo de estos filtros. Con este filtro en particular Assembler termina la ejecución en menos de la mitad de ciclos que C.

1.3.2. Loop Unroll

Este experimento consiste en realizar repetidas veces todo lo descripto en la sección anterior dentro del ciclo principal del programa antes de saltar nuevamente al comienzo del ciclo. Veamos un ejemplo para que quede mas claro: Supongamos que tenemos un ciclo que lo único que hace es sumarle 1 a una variable i y luego avanzar en uno la variable de iteración. Con la implementación de este experimento, dentro de ese ciclo ahora se le sumaria x veces 1 a dicha variable y luego se aumentaría en x a la variable de iteración. En nuestro caso decidimos hacerlo con $x = 2, 4, 8, 16$ y 32.

La hipótesis de este experimento es que el tiempo total de ejecución del programa debería verse disminuido a medida que aumente el x recién mencionado ya que los saltos condicionales cuestan tiempo significativo y desenrollando el ciclo estaríamos realizando menos saltos y consecuentemente perderíamos menos tiempo.

Los resultados obtenidos fueron los siguientes:



Como podemos observar, la cantidad de ciclos de clock necesarios para correr el filtro sin realizar Unroll es levemente mayor a la cantidad de ciclos necesarios cuando si se utiliza esta técnica desenrollando el ciclo 2 y 4 veces. Sin embargo, nos llamo la atención ver que cuando aumentamos la cantidad de veces que desenrollamos el ciclo comienzan a aumentar los ciclos del clock de manera significativa.

Para intentar evacuar las dudas decidimos confeccionar la siguiente tabla comparando cantidad de instrucciones y cantidad de accesos a memoria de todas las versiones de Unroll:

#Unroll	# Instrucciones	# Accesos lectura	#Accesos escritura	# Total accesos a memoria
Sin Unroll	8459431	2773091	317625	3090716
Unroll 2	8428431	2773091	317625	3090716
Unroll 4	8429431	2773091	317625	3090716
Unroll 8	8429431	2773091	317625	3090716
Unroll 16	8429431	2773091	317625	3090716
Unroll 32	8429431	2773091	317625	3090716

A pesar de unas pequeñas variaciones en la cantidad de instrucciones de las primeras dos versiones, el resto de los datos son exactamente iguales en todos los casos. Así podemos descartar que el problema encontrado se deba a una de estas dos cosas.

No conformes con esto, decidimos dar un paso mas y buscamos el *Miss Rate* de la memoria cache en cada una de estas implementaciones. Nuevamente los datos obtenidos no fueron concluyentes. En todos los casos se obtuvo 0.6 %.

Finalmente, luego de todas estas pruebas, no logramos descifrar a qué puede deberse el aumento de ciclos de clock a medida que se desenrolla mas el ciclo.

2. Four Combine

El filtro Four Combine consiste en reorganizar los píxeles de la imagen original obtenida como parámetro buscando conseguir como resultado 4 imágenes idénticas a la original pero mas pequeñas, de forma que sumando los tamaños de las 4 (las 4 del mismo tamaño) se obtenga el tamaño de la original.

Los píxeles no son modificados en ningún momento.

2.1. Pseudo código del algoritmo en C

Algoritmo 3 Pseudocódigo de la implementacción en C de fourCombine

```
1: i = 0
2: while i < cantFilas do
3:     j = 0
4:     while j < cantColumnas do
5:         pixel[i][j].dst = pixel[i][j].src
6:         pixel[i][j + cantColumnas/2].dst = pixel[i][j + 1].src
7:         pixel[i + cantFilas/2][j].dst = pixel[i + 1][j].src
8:         pixel[i + cantFilas/2][j + cantColumnas/2].dst = pixel[i + 1][j + 1].src
9:         j = j + 2
10:    end while
11:    i = i + 2
12: end while
```

2.2. Descripción de implementación en assembler

Para la implementación de este filtro lo primero que hicimos fue inicializar registros apuntando a la imagen *src* (la que debemos modificar) y a la imagen *dst* (donde reconstruimos la nueva imagen). También se inicializó un registro con el tamaño de la imagen, el cual se utilizó para comparar con

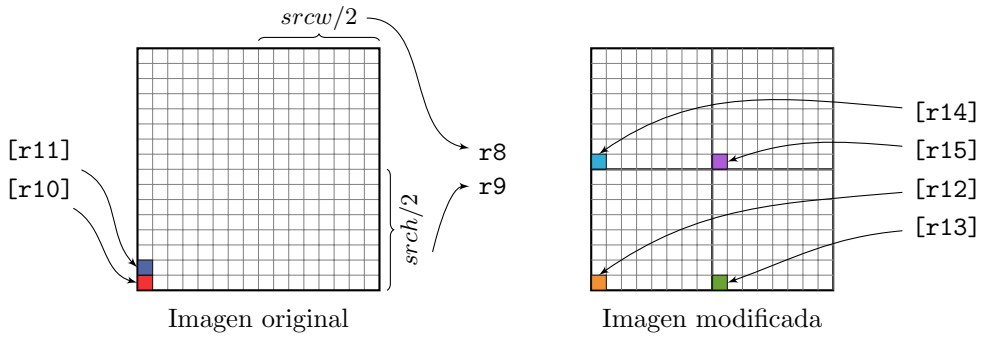


Figura 5: Esquema del contenido de los registros de propósito general utilizados en la implementación de *fourCombine* al comenzar la primera iteración del ciclo principal. Los registros `r10` a `r14` funcionan como punteros a las imagenes fuente y destino (representadas aquí como matrices de píxeles), mientras que en `r8` y `r9` se almacena la mitad de las dimensiones (en bytes) de la primera. No se muestran el registro `rbx`, cuyo valor se inicializa en 0 y actúa como contador de las columnas recorridas en una fila de *src*, ni `rax`, que contiene el tamaño en bytes de la imagen original.

otro que se va incrementando en cada iteración del ciclo principal para así saber cuando se termino de recorrer toda la imagen; y otro registro con el valor de la mitad de la longitud de las filas, que se compara en cada iteración con uno mas para saber cuando se llegó al final de una fila en algún cuadrante de la imagen resultante. El contenido de los registros se detalla en la Figura

Luego de esto se pasa el ciclo principal (Figura 6). Al final de cada iteración del mismo, se incrementa en 8 a los cuatro registros que apuntan a la imagen destino (2 píxeles cada uno), en 16 los dos que apuntan a la imagen fuente y se chequea si se llegó al final de la fila en los cuatro cuadrantes de la imagen destino. En caso afirmativo, se les suma la mitad de lo que miden las filas a los cuatro registros de la imagen destino y se les suma la longitud de las filas a los registros que apuntan a la imagen fuente y se vuelve al principio del ciclo. En el caso de no haber llegado al final de la fila, se continúa con la siguiente iteración. Nótese que haciendo las cosas de esta manera, cuando se llegue al final de la fila en alguno de los cuadrantes de la imagen destino, se llegara al final de la fila en los cuatro a la vez y también se llegara al final de la fila en la imagen fuente.

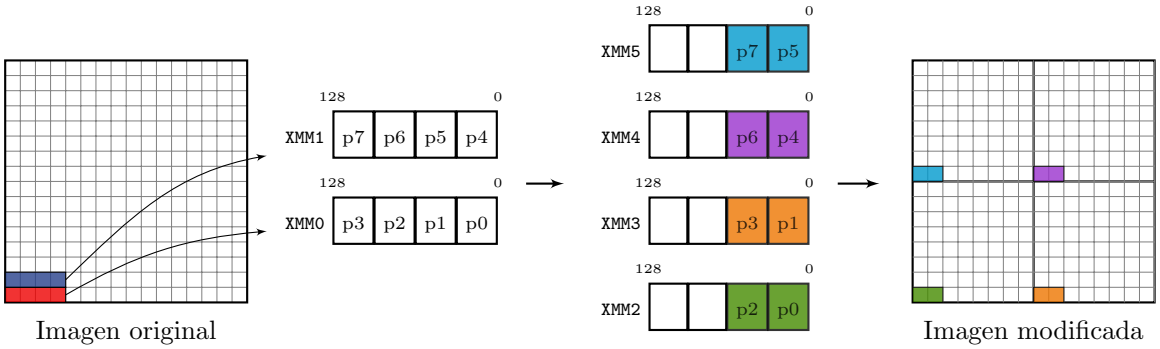
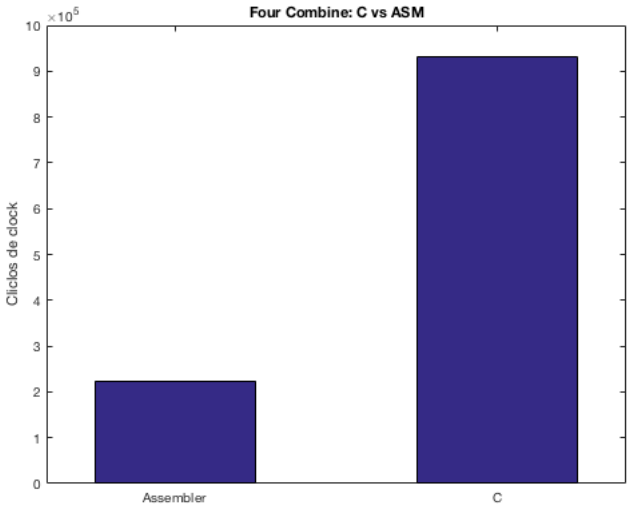


Figura 6: Una iteración del ciclo principal de *fourCombine*. Se copian 4 píxeles de la última fila de *src* y 4 de la penúltima (en rojo y azul, respectivamente) en los registros `XMM0` y `XMM1` con la instrucción `MOVDQU` luego de lo cual se realiza un shuffle (`PSHUFD`) para separar su contenido en los registros `XMM2` a `XMM5` según el cuadrante al que pertenezca cada uno en *dst*. Para finalizar, se copia la parte baja de estos cuatro registros (`MOVQ`) a las posiciones correspondientes en la imagen destino.

2.3. Experimentación

2.3.1. Comparación C vs Assembler

El siguiente gráfico muestra la diferencia de ciclos de clock necesarios para completar este filtro entre Assembler y C:

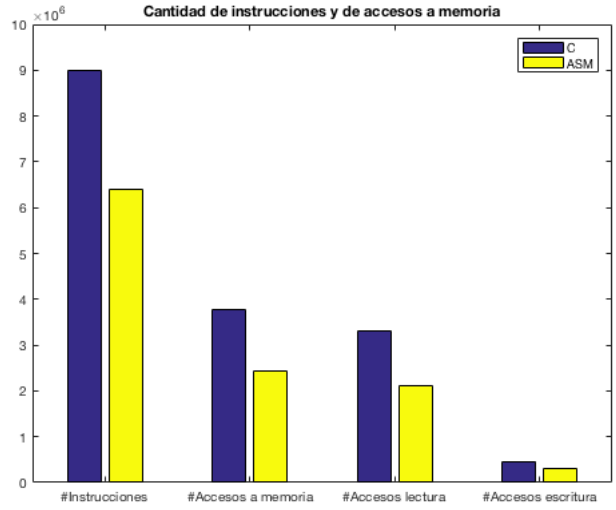


La versión en Assembler resulta considerablemente mas eficiente que la de C. En este caso, esto puede deberse a que la versión de C copia de a 1 byte a la vez, mientras que la de Assembler, por medio de registros de 128 bits, consigue copiar de a dos píxeles a la vez (8 bytes), rediciendo así de forma significativa la cantidad de instrucciones y de accesos a memoria.

En la siguiente tabla se expresa lo descripto en el párrafo anterior, comparando a ambas versiones del filtro en cuanto a cantidad de instrucciones y de accesos a memoria. Cada numero detallado en la ultima linea de la tabla muestra qué porcentaje de lo que utiliza C precisa Assembler para implementar lo mismo:

Lenguaje	# Instrucciones	# Accesos lectura	#Accesos escritura	# Total accesos a memoria
C	9000981	3303872	457827	3761699
Assembler	6390180	2103103	317625	2420728
Relacion asm/c	70.99%	63.66%	69.38%	64.35%

Para apreciar mejor estos datos veamos el siguiente gráfico:



Efectivamente vemos que el mejor funcionamiento de la implementación en el lenguaje Assembler se debe en gran parte a la disminución en cantidad de instrucciones y de accesos totales a memoria con respecto a la versión en C.

3. Máximo Cercano

Máximo Cercano busca el máximo de cada componente en una matriz de 7x7 píxeles donde el píxel P se encuentra justo al centro y luego realiza una serie de operaciones entre el nuevo píxel obtenido (donde las componentes R, G y B son las mayores en la matriz de 7x7 mencionada previamente), y sobre el píxel P. Dichas operaciones consiguen generar un nuevo píxel que contenga un porcentaje del píxel original (P) y otro porcentaje del máximo cercano. El porcentaje (α) sera representado con un numero de precisión simple (float) entre 0 y 1 y sera un parámetro de entrada de la función implementada. Debemos evitar perder precisión en los cálculos. La cuenta realizada es

$$dst[i,i] = src[i,i] * (1 - \alpha) + max * (\alpha).$$

Donde dst es la imagen destino y src es la imagen fuente. Esta operación se realiza con cada una de las componentes de cada píxel.

El margen de 3 píxeles de la imagen es pintado de blanco, es decir que las componentes de dichos píxeles tendrán el valor 255. De este modo no hay que preocuparse por los bordes al momento de armar la matriz de 7x7 alrededor de cada píxel a modificar.

3.1. Pseudo código del algoritmo en C

Algoritmo 4 Pseudocódico de la implementación en C de maxCloser

```
1: i = 0
2: while i < cantFilas do
3:     j = 0
4:     while j < cantColumnas do
5:         if distanciaAlBorde < 3 then
6:             pixel[i][j].dst = 255
7:         else
8:             k = −3
9:             while k < 3 do
10:                l = −3
11:                while l < 3 do
12:                    maxR = max(maxR, pixel[i + k][j + l].R)
13:                    maxG = max(maxG, pixel[i + k][j + l].G)
14:                    maxB = max(maxB, pixel[i + k][j + l].B)
15:                    l + +
16:                end while
17:                k + +
18:            end while
19:            pixel[i][j].dst = pixel[i][j].src ∗ (1 − α) + maxRGB ∗ α
20:        end if
21:        j + +
22:    end while
23:    i + +
24: end while
```

3.2. Descripción de implementación en assembler

A diferencia de los anteriores filtros, maxCloser recibe un parámetro mas como entrada, y es de tipo float. A dicho valor lo llamaremos `Val`. En assembler, este parámetro ingresa en los 4 bytes de la parte baja del registro `XMM0`.En nuestra implementación, lo primero que hicimos con este registro fue utilizar la instrucción `PSHUFD` para que el dato se repita cuatro veces en el registro. Tambien inicializamos otro registro XMM con el valor 1 cada 4 bytes como el anterior y le restamos `XMM0` para obtener `1-Val` cada 4 bytes.

Después de hacer esto y de preparar otros registros con valores que necesitaremos comienza el ciclo principal del programa. Lo primero que se hace dentro de este es hacer una comparación entre dos registros para ver si se llego al final de la imagen que estamos recorriendo. De ser así el programa finaliza. Caso contrario se llama a la función `inRange`, la cual se encarga de verificar si el píxel con el que se esta por trabajar se encuentra dentro del margen de 3 píxeles de proximidad a alguno de los bordes de la imagen. En el caso de hallarse dentro de dicho margen, se realiza un salto condicional a la etiqueta `pintarBlanco`, donde se copia el valor 255 a la imagen destino en la posición del píxel en cuestión, se suma 4 al puntero a la imagen fuente y se vuelve al ciclo principal. Cuando el píxel apuntado por el puntero a la imagen fuente no se encuentra en ese margen, se procede a buscar el máximo en la matriz de 7x7 que rodea al píxel.

En esta parte del programa, lo que se hizo fue almacenar la matriz de 7x7 en registros XMM (Figura 7). Si bien es cierto que de esta manera se cargan dos veces los píxeles que se encuentran en la columna del medio, esto no influye en el resultado final, como se verá más adelante.

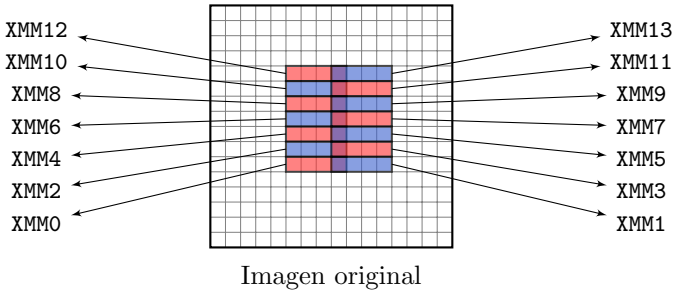


Figura 7: Almacenamiento de un kernel de 7x7 en registros `XMM`.

Teniendo ya toda la matriz cargada, utilizamos la instrucción `PMAXUB` de a dos registros, pasando por todos y obteniendo así los 16 bytes máximos en un registro XMM.Como lo que queremos no son 4 píxeles sino uno solo, generamos 3 copias del registro que contiene los máximos y utilizamos la instrucción `PSRLDQ` para shiftear a las tres nuevas copias 4, 8 y 12 bytes y volvemos a buscar los máximos entre estos 4 registros obteniendo así un registro XMM que contendrá al máximo de cada componente (R, G y B) en los 4 bytes de su parte baja.

Ya tenemos el píxel máximo, falta combinarlo con el píxel original. Para ello lo primero que hicimos fue desempaquetar de *byte* a *word* y luego de *word* a *doubleword* y convertir el resultado de esto de *double quadword* a *single precision float* con la instrucción `CVTDQ2PS`. De este modo evitamos perder precisión al multiplicar a este registro con el que contiene a `Val`. Esta ultima operación la hacemos con la instrucción `MULPS` la cual se utiliza para multiplicar *floats* empaquetados. El resto puede verse en la Figura 8.

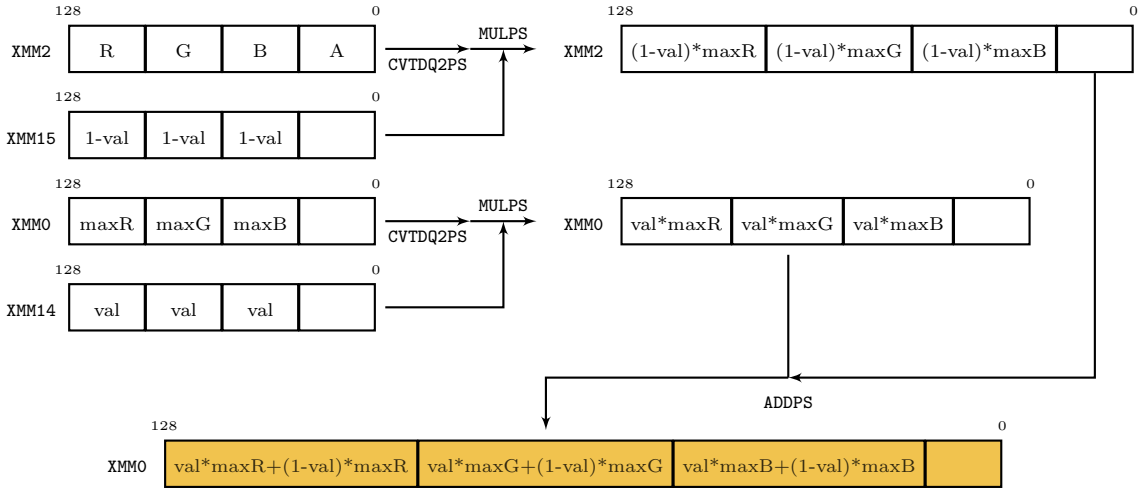


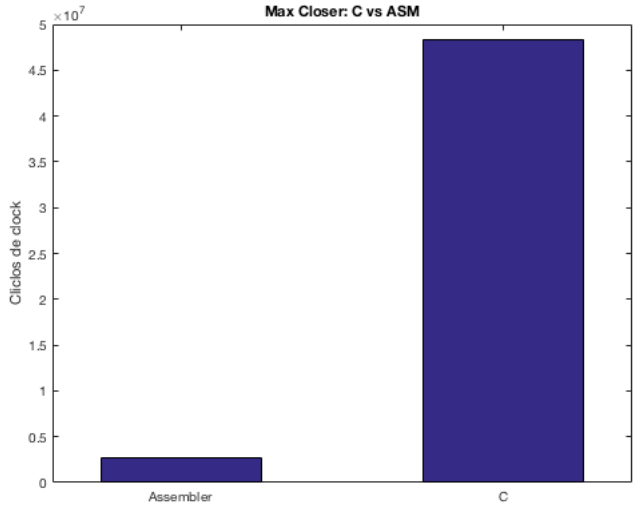
Figura 8: Combinación del máximo del kernel con el píxel original. Ambos valores son convertidos a números empaquetados de punto flotante de precisión simple para poder luego ser multiplicados con los registros que contienen `val` y `1-val`. Finalmente se suman ambos productos y el resultado final se empaqueta a *word*, luego a *byte* y se copia en la imagen destino.

Para terminar, se le suma 4 al registro que indica cuantas posiciones de la imagen se recorrieron y se vuelve al comienzo para continuar con el siguiente píxel a procesar.

3.3. Experimentación

3.3.1. Comparación C vs Assembler

En la siguiente imagen se muestran los resultados de la comparación entre la versión C y Assembler del filtro *maxCloser*.



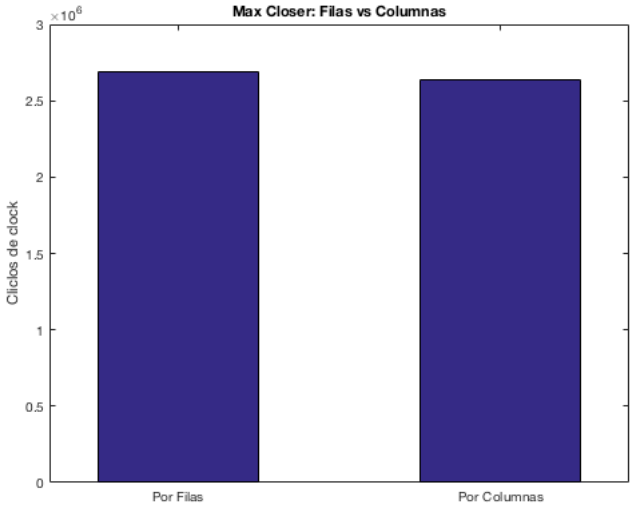
Nuevamente se ve como la implementación del mismo filtro en Assembler resulta ser mas rápida en términos de ciclos de clock que la de C. En este caso la versión de C realiza 10 veces mas ciclos de clock que la de Assembler.

3.3.2. Recorrer imagen por columnas

Como vimos en la sección anterior, la matriz generada por la imagen fue recorrida por filas. Cada vez que se terminaba de procesar un píxel, se le sumaba 4 al puntero que apuntaba a la imagen original y de esta manera se avanzaba al siguiente píxel en la fila. Con este experimento cambiaremos la forma de recorrer esta matriz y lo haremos por columnas. Es decir que cada vez que terminemos de procesar un píxel le sumaremos una fila al puntero que apunta a la imagen original para que este apunte al píxel inmediatamente siguiente dentro de la misma columna del anterior.

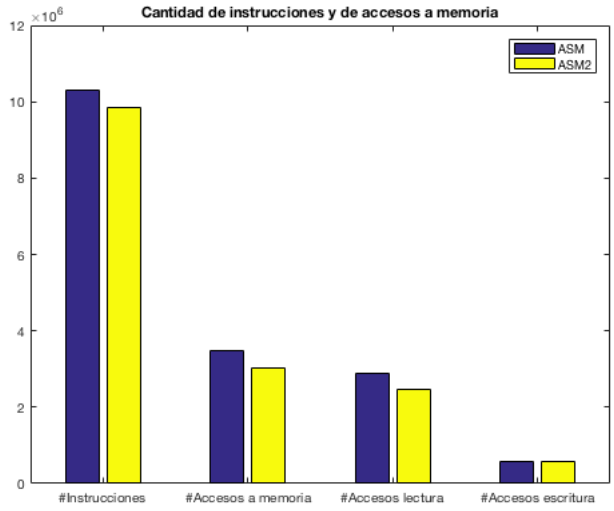
Lo que esperamos ver con esta nueva implementación es una notoria disminución en el tiempo total de ejecución del filtro, debida a la reducción en la cantidad de accesos a memoria. Con la anterior implementación cada vez que se procesaba un nuevo píxel se producían 14 accesos a memoria (2 accesos por cada fila de la matriz de 7x7 que se traía a registros XMM). Recorriendo por columnas lo único que debemos hacer cada vez que procesamos un nuevo píxel es cambiar los dos registros que contenían a la primera fila de la matriz de 7x7, copiando a estos la nueva fila correspondiente a la nueva matriz que rodea al nuevo píxel. De este modo solo realizaremos 14 accesos a memoria cada vez que se arme la matriz de 7x7 por primera vez para cada columna y el resto de las veces solo 2 accesos a memoria.

El gráfico a continuación muestra la diferencia de ciclos de clock entre la primera implementación del filtro (recorriendo a la matriz por filas) y la nueva (recorriendo a la matriz por columnas):



De acuerdo al gráfico anterior, si bien la segunda implementación parece tardar menos ciclos de clock, la diferencia entre ambas versiones no es tan notoria como esperábamos.

Para despejar las dudas decidimos realizar otro experimento en el cual medimos la cantidad de instrucciones y de accesos a memoria realizados por ambas versiones de la implementación del filtro en cuestión. Los resultados se pueden observar en el siguiente gráfico:



Si bien las diferencias entre las distintas versiones de la implementación en Assembler no son tan notorias como las de *C vs Assembler*, de todos modos se puede ver que la segunda versión es superior a la primera en todos los casos.

4. Zoom Lineal

Este filtro genera una nueva imagen cuyas dimensiones son el doble de la imagen original (dsth = 2*srch, dstw = 2*srcw) mediante una interpolación lineal entre pixeles, como se indica en la siguiente figura.

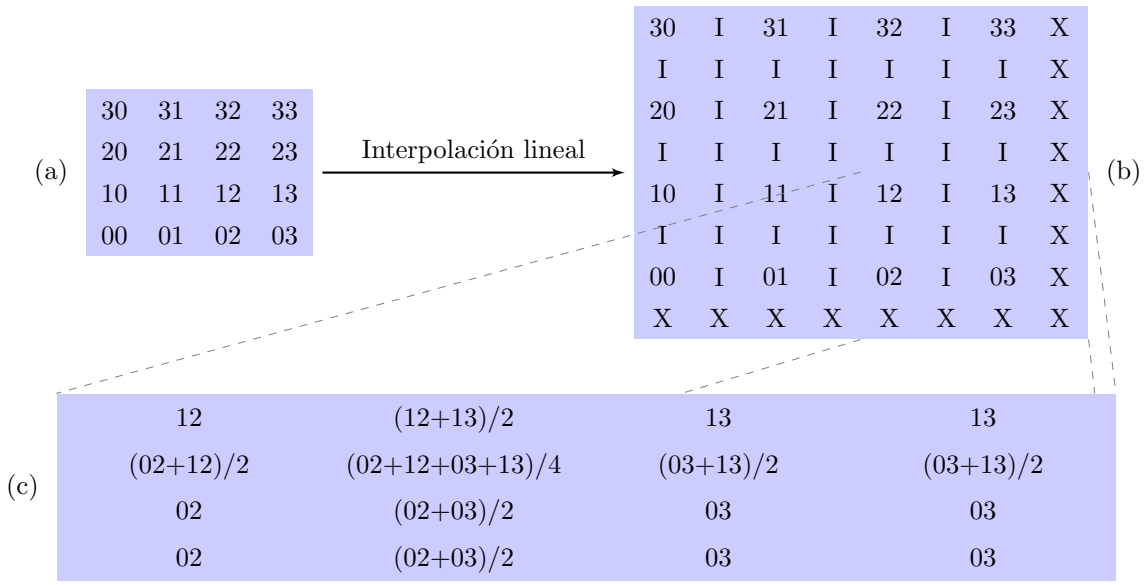


Figura 9: Esquema del funcionamiento general del filtro *linearZoom*. Se muestran la imagen original (a), la imagen generada a partir de la interpolación lineal (b) y un detalle de esta última (c) donde se aprecia el contenido de los píxeles. Las operaciones que se presentan como entre píxeles se realizan componente a componente.

4.1. Pseudocódigo del algoritmo en C

Algoritmo 6 Pseudocódico de la implementación en C de linearZoom

Input: Dos matrices de pixeles que representan a la imagen original (src) y a la modificada (dst) con sus respectivos anchos y altos (srcw, dstw, srch, dsth).

```
dst_row ← 0
for i = 0 → srcw − 1 step 1 do
    dst_col ← 0
    for j = 0 → srch − 1 step 1 do
        dst[dst_row][dst_col] ← src[i][j]
        dst_col ← dst_col + 2
    end for
    dst_row ← dst_row + 2
end for
for i = 1 → dsth step 2 do
    for j = 1 → dstw − 1, step 2 do
        dst[i][j] ← (dst[i][j − 1] + dst[i][j + 1])/2
    end for
end for
for j = 0 → dstw − 1 step 2 do
    for i = 2 → dsth step 2 do
        dst[i][j] ← (dst[i − 1][j] + dst[i + 1][j])/2
    end for
end for
for i = 2 → dsth step 2 do
    for j = 1 → dstw − 1 step 2 do
        dst[i][j] ← (dst[i − 1][j + 1] + dst[i − 1][j − 1] + dst[i + 1][j + 1] + dst[i + 1][j − 1])/4
    end for
end for
for col = 0 → dstw step 1 do
    dst[0][col] ← dst[1][col]
```

end for
for $row = 0 \rightarrow dsth$ **step** 1 **do**
$dst[row][dstw - 1] \leftarrow dst[row][dstw - 2]$
end for

4.2. Descripción de la implementación en assembler

Para simplificar, llamaremos *src* y *dst* a las imágenes original y modificada respectivamente.

La implementación en ASM del zoom lineal puede separarse, a grandes rasgos en 3 ciclos: uno principal, donde se generó la mayor parte de *dst*, uno para el borde derecho y otro para el inferior.

4.2.1. Ciclo principal

Para esta parte se recorrió *src* con un puntero almacenado en `RBX`, empezando desde la tercera fila desde abajo. [1] Mediante la adición del ancho de *src* multiplicado por 4 a `rbx`, se fueron recogiendo datos de dos filas simultáneamente. Luego, al comienzo del ciclo se tienen dos punteros: `rbx` y otro que llamaremos `rbx'` que apunta a la misma columna en la fila inmediatamente superior.

Una vez almacenados 8 píxeles en dos registros `XMM` con la instrucción `MOVDQU`, se copia cada uno para tener 2 copias de cada píxel, se los ordena y desempaqueta como se indica en la Figura 10

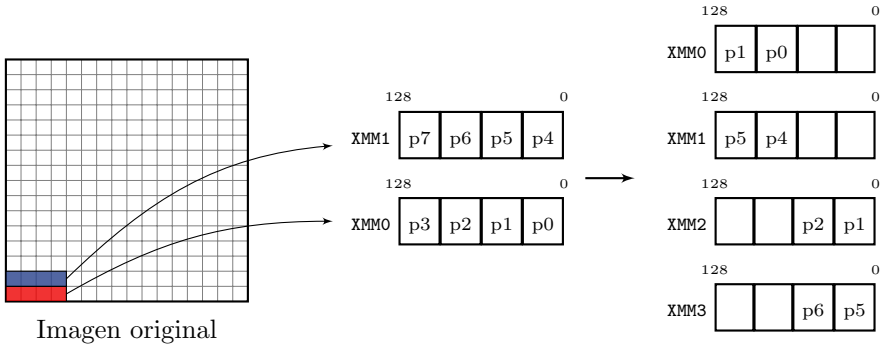


Figura 10: Comienzo de la primera iteración de *linearZoom*. Los píxeles marcados en rojo y azul se copian con `MOVDQU` a `XMM0` y `XMM1`. Los contenidos de cada uno se copian a los registros `XMM2` y `XMM3`, se shiftean con `PSRLDQ` y `PSLLDQ` y luego se desempaquetan a *word* las partes altas de los dos primeros y las partes bajas de los dos últimos. Los píxeles que figuran como p7 y p3 no se utilizan durante en ciclo; solamente están presentes debido a que se intentó ahorrar accesos a memoria levantando tanta información como cupiera en un registro de 128 bits.

A continuación, se procede a generar dos vectores que serán luego filas de *dst* (ver Figura ...) respetando el formato descripto en la Figura 9; es decir, a partir de los píxeles que se llevan inicialmente a `XMM0` y `XMM1` como se ve en la Figura 10 se desea obtener resultados de la forma presentada en la Figura

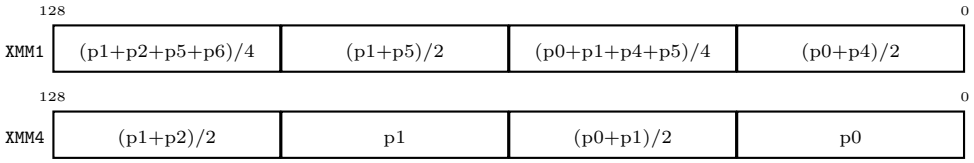


Figura 11: Resultados de las operaciones del ciclo principal de *linearZoom*, que se copian a la imagen destino

Por una cuestión de implementación, en esta parte se incluye el borde inferior de `dst`. Este detalle es corregido posteriormente en el tercer ciclo del programa.

Finalmente, se realiza un `MOVDQU` para copiar el resultado final en la imagen destino. Para esto se cuenta con un puntero almacenado en `r13`, que inicialmente contiene la dirección de memoria donde comienza *dst*. Al igual que con `rbx`, es posible acceder a la fila superior por medio de la operación $r13 + 4 * dstw$ para así poder modificar ambas simultáneamente.

[1] Estas imágenes se almacenan en la memoria principal de forma invertida. Es decir que la posición apuntada inicialmente por *src* corresponde al primer píxel de la última fila.

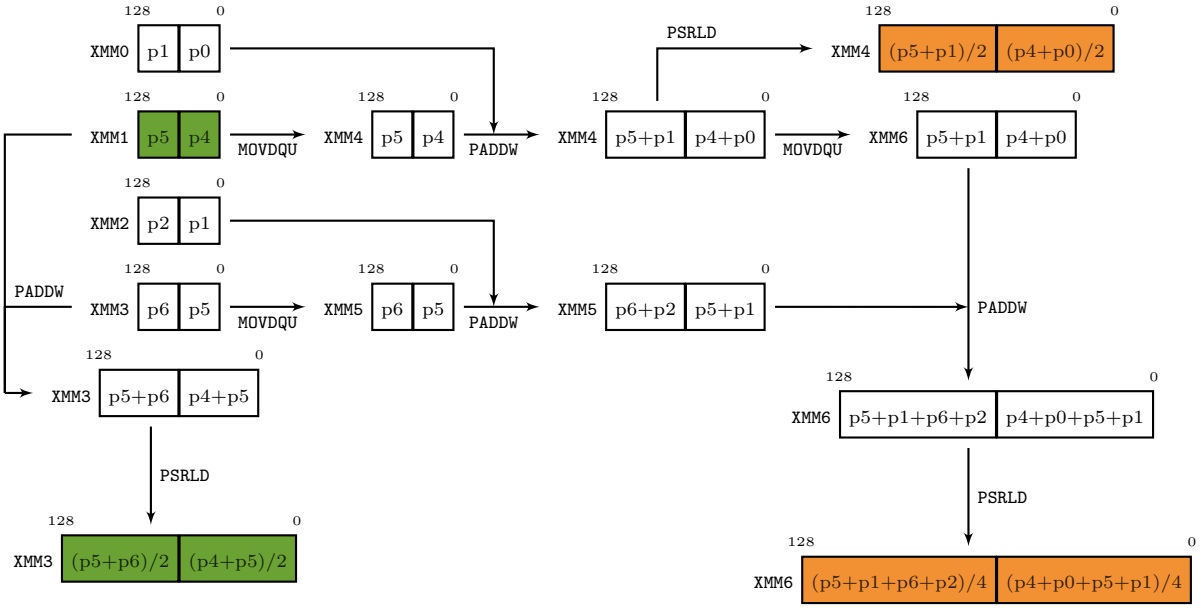


Figura 12: Esquema de las operaciones realizadas en el ciclo principal de *linearZoom* para obtener las filas de la imagen destino. Las instrucciones ubicadas en intersecciones son aquellas que trabajan con más de un registro, donde los caminos entrantes -a dicha intersección- corresponden a los operandos fuente y el saliente apunta hacia el operando destino. Los registros coloreados son los que, al finalizar esta etapa, se empaquetan (`PACKUSWB`), combinan (`POR`) y acomodan con `PSHUFD` para quedar como se indica en la Figura 11. El `POR` se realiza entre los registros de igual color.

4.2.2. Ciclo para generar el borde derecho

En este caso se recorre el borde derecho de *src* tomando los **dos** últimos píxeles de cada fila (y también los dos de la fila superior) con la instrucción `MOVQ`. Si bien podrían haberse levantado los datos de la memoria con `MOVDQU`, el resultado hubiera sido el mismo pues se habría utilizado la misma cantidad de píxeles.

Las operaciones realizadas en este ciclo fueron las mismas que la sección del ciclo principal representada el la figura 12 con la única diferencia de que, al finalizar, debió modificarse la constante del `PSHUFD` para ajustar los vectores obtenidos a la forma de las dos filas superiores de la Figura 9 (c).

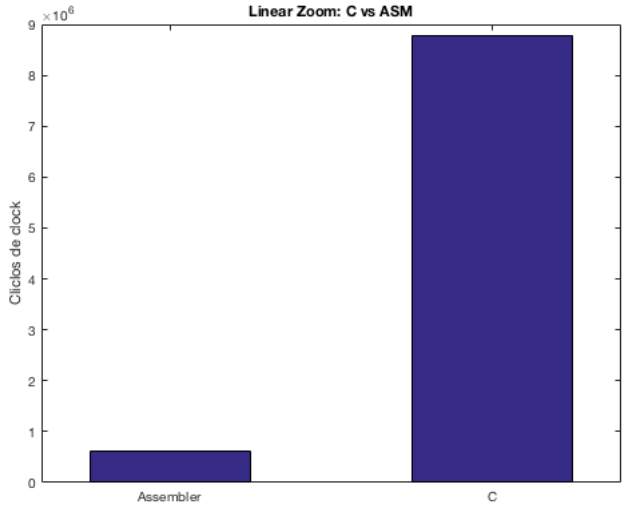
4.2.3. Ciclo para generar el borde inferior

Lo que se hace aquí es recorrer la última fila (primera en la memoria) de *src* tomando píxeles de a 4 con `MOVDQU` y generando resultados de la forma que poseen las dos filas inferiores de la Figura 9 (c) mediante un procedimiento análogo al que puede verse en la figura 12, pero en donde solamente se realizan las operaciones para obtener los resultados que en dicho esquema se muestran en verde. Se dispone de un puntero a la última fila de *dst* y a la penúltima de modo que el vector resultante de las operaciones del ciclo se copie por igual en ambas, ya que la fila $n-1$ de *dst* (donde n es la altura de la imagen destino) es igual a la $n-2$, como se detalla en la Figura 9 (c).

4.3. Experimentación

4.3.1. Comparación C vs Assembler

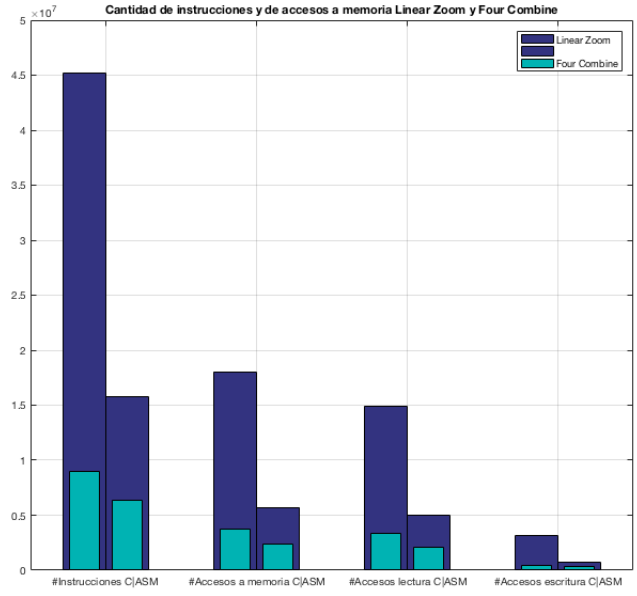
En el siguiente gráfico se muestra la comparación de ciclos de clock necesarios para completar la ejecución del filtro *Linear Zoom* en Assembler y C:



Una vez mas se ve como la versión en Assembler vuelve a ser mucho mas rápida que la de C. Ademas, habiendo ya analizado este factor en los filtros anteriores, parecería que esta diferencia entre las dos implementaciones crece proporcionalmente a la complejidad de los filtros. Para poder ver esto ultimo de forma mas clara observemos las comparaciones de los datos obtenidos con las implementaciones de este filtro y las de *Four Combine*, el cual resulta ser el filtro menos complejo entre los 4 implementados a lo largo de este trabajo:

Filtro	Lenguaje	# Instrucciones	# Accesos lectura	#Accesos escritura	# Total accesos a memoria
Four Combine	C	9000981	3303872	457827	3761699
	Assembler	6390180	2103103	317625	2420728
	Relacion asm/c	70.99%	63.66%	69.38%	64.35%
Linear Zoom	C	45215271	14949237	3123027	18042264
	Assembler	15777672	5013926	697749	5711675
	Relacion asm/c	34.89%	33.54%	22.34%	31.66%

El siguiente gráfico refleja los datos de la tabla:



Los datos recién mostrados muestran claramente como, a medida que aumenta la complejidad del

filtro implementado (complejidad medida en cantidad de instrucciones y accesos a memoria), aumenta en mayor medida la diferencia entre la implementación en C y en Assembler. Es decir que para filtros mas complejos parecen ser realmente notables las mejoras proporcionadas por el uso de instrucciones SIMD.

Parte III
Conclusiones

A lo largo de este trabajo pudimos ver como el uso de instrucciones SIMD mejoro la implementación de los filtros en todos los casos, en comparación con las implementaciones en el lenguaje C. El trabajar con datos de manera simultanea redujo de manera considerable la cantidad de accesos a memoria, debido a la posibilidad de leer y escribir en la misma de a bloques con la utilización de registros de 16 bytes. Al mismo tiempo, se redujo en gran medida la cantidad de instrucciones ya que con los registros recién mencionados se consiguió procesar varios datos al mismo tiempo.

Si bien los experimentos realizados nos fue de gran ayuda para aclarar dudas y confirmar hipótesis, en algunos casos los resultados no fueron del todo concluyentes y quedaron abiertos a experimentaciones futuras. En el caso de *loop unroll* por ejemplo, intentamos abordar el problema por diversos lugares dentro de los conocimientos con los que contamos hasta el momento y no logramos conseguir resultados esclarecedores. En este sentido, quedara pendiente indagar sobre factores externos a los estudiados en este trabajo, para así poder realizar mas experimentos con la intención de comprender a fondo el problema mencionado.

Finalmente, nos resulto evidente que cuando mas complejo es el problema a implementar, mas notoria se vuelven las mejoras producidas por la utilización de instrucciones SIMD. Es decir que es en los casos en que mas recursos se necesitan para el procesamiento de datos, cuando realmente toma mas sentido el uso de estas instrucciones.

Vale aclarar que siempre que se consigue mejoras es a costa de algo, y en este caso, las mejoras obtenidas por la implementación de instrucciones SIMD resultan en un incremento no menor en cantidad de lineas de código en comparación con la implementación en C, así como también aumenta en gran medida la dificultad de comprender dicho código.