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

RGB2YUV:

```
1: j = 0
2: while j < cantPixeles do
3:     R = pixel[j].R
4:     G = pixel[j].G
5:     B = pixel[j].B
6:     pixel[j].Y = Satura(((66 * R + 129 * G + 25 * B + 128) >> 8) + 16)
7:     pixel[j].U = Satura(((−38 * R − 74 * G + 112 * B + 128) >> 8) + 128)
8:     pixel[j].V = Satura(((112 * R − 94 * G − 18 * B + 128) >> 8) + 128)
9:     j + +
10: end while
```

YUV2RGB:

```
1: j = 0
2: while j < cantPixeles do
3:     Y = pixel[j].Y
4:     U = pixel[j].U
5:     V = pixel[j].V
6:     pixel[j].R = Satura((298 * (Y − 16) + 409 * (V − 128) + 128) >> 8)
7:     pixel[j].G = Satura((298 * (Y − 16) − 100 * (U − 128) − 208 * (V − 128) + 128) >> 8)
8:     pixel[j].B = Satura((298 * (Y − 16) + 516 * (U − 128) + 128) >> 8)
9:     j + +
10: 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 mediante la utilización de las instrucciones del modelo de procesamiento SIMD. Los registros en este modelo son de 128 bits, con lo cual un píxel ocupa solo un cuarto. La decisión de trabajar solo de a un píxel a la vez 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 seria suficiente.

Para realizar las operaciones de la forma mencionada debimos proceder a desempaquetar el píxel de bytes a words y luego de words a double words, utilizando así los 128 bits del registro XMM ocupando 32 bits cada componente (los 24 bits de la parte alta de cada componente se extienden con ceros). Las instrucciones necesarias para realizar estos pasos de desempaquetado fueron `PUNPCKLBW` y `PUNPCKLWD`, las cuales desempaquetan la parte baja del registro de bytes a words y de words a double words respectivamente.

(ACA VA UN DIBUJO DEL MOVQ DE DOS BYTES A LA PARTE BAJA DEL XMM Y OTRO DIBUJO QUE MUESTRE EL DESEMPAQUETADO, O UN SOLO DIBUJO QUE LOS MUESTRE YA DESEMPAQUETADOS)

Una vez hecho esto, procedimos a copiar los datos a dos registros distintos del original para luego shiftearlos con la instrucción `PSRLDQ` dejando así a las componentes R, G y B en la misma parte de cada registro y luego aislamos en cada uno a las tres componentes mediante la utilización de mascaras. De este modo un registro contendrá solo una componente R en cierta posición del mismo, otro contendrá solo a G en la misma posición que el anterior y lo mismo con el tercer registro que solo contiene a B.

(ACA VA UNA IMAGEN QUE MUESTRE COMO SE SHIFTEARON LOS TRES REGISTROS Y LUEGO COMO SE AISLAN R, G Y B CON LA APLICACION DE MASCARAS)

El siguiente paso consistió en realizar la transformación lineal a cada componente realizando multiplicaciones entre los registros y mascaras con valores constantes y sumando registros entre si.

(PONER ACA DIBUJOS QUE MUESTREN COMO SE MULTIPLICO A LOS REGISTROS POR MASCARAS Y SE LOS SUMARON ENTRE SI PARA CONSEGUIR LA TL BUSCADA. NO HACE FALTA PONER COMO CAMBIA CADA COMPONENTE, SE PUEDE MOSTRAR COMO CAMBIA UNA SOLA Y LOS OTROS DOS CASOS SON ANALOGOS)

Por ultimo, luego de realizar todas las operaciones anteriores ya tenemos tres registros distintos donde cada uno tiene a las componentes R, G y B con la transformación lineal ya aplicada, con lo cual a partir de este momento las llamamos Y, U y V respectivamente. 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 pero esta vez hacia el lado opuesto al que se los shifteo antes, con la instrucción `PSLLDQ` para de esta forma acomodar a las componentes Y, U y V en las partes que les corresponde en cada registro. Después se juntó a las tres componentes en un solo registro con la instrucción `XORPS` y finalmente se copio la parte baja del registro resultante a su posición correspondiente en la imagen destino.

(ACA VAN DUBUJOS QUE MUESTRAN COMO SE EMPAQUETAN LOS REGISTROS NUEVAMENTE, DESPUES SE SHIFTEAN, SE HACE XORPS PARA JUNTARLOS EN UNO SOLO Y SE COPIAN A LA IMAGEN DESTINO)

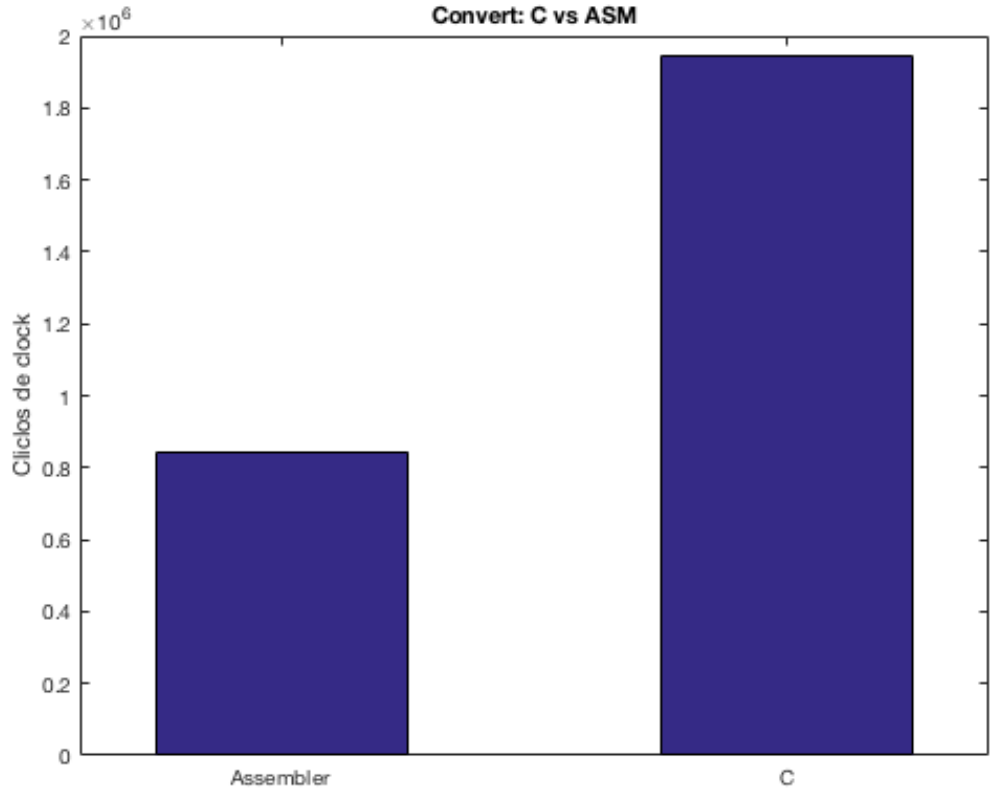
Luego de realizar todos estos pasos se vuelve al principio y se realizan los mismos pasaos con los dos píxeles siguientes y así hasta haber recorrido toda la imagen.

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.



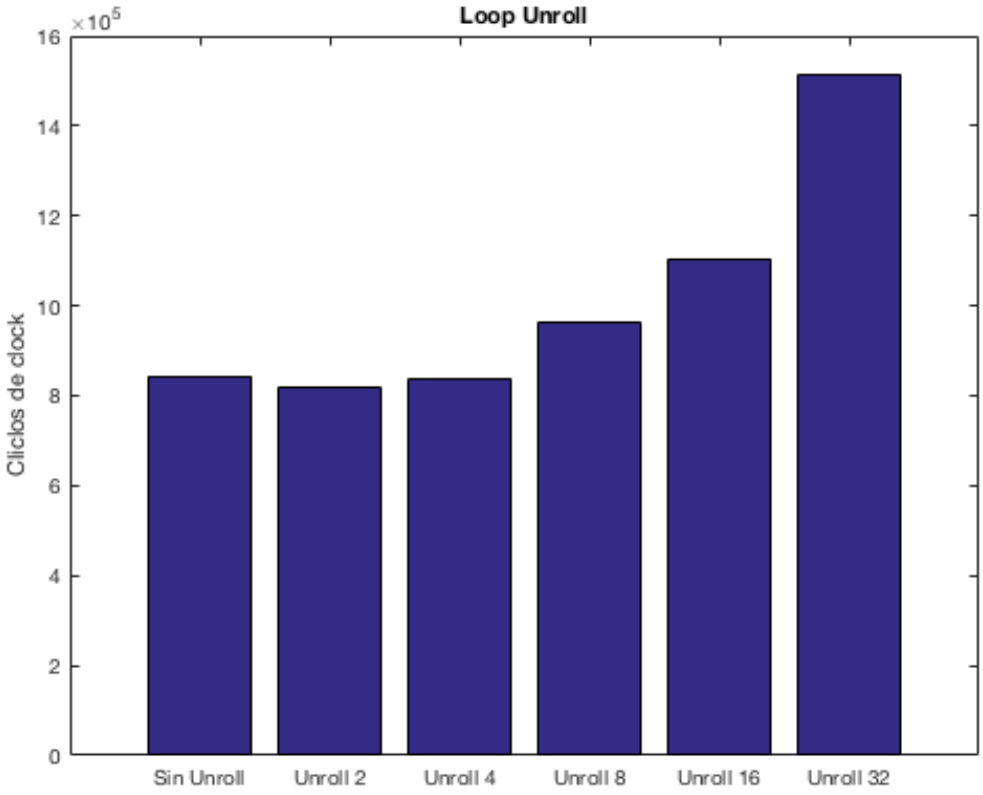
En el gráfico anterior podemos ver claramente como la versión del filtro implementada en Assembler tarda la mitad de ciclos de clock que la de C. Esto era de esperarse ya que justamente lo que buscamos mediante el uso de instrucciones SIMD es mejorar el tiempo de computo de estos filtros. Con este filtro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

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 se reorganizan de la siguiente manera, donde i recorre las filas, j las columnas y ambas comienzan en la posicion 0:

1. Buscamos los píxeles en las posiciones (i,j), (i, j+1), (i+1,j), (i+1.j+1).
2. Los copiamos en la imagen destino en las posiciones (i,j), (i, j+(Ancho/2)), (i+(Largo/2),j), (i+(Largo/2), j+(Ancho/2)).
3. i pasa a valer i+2, j pasa a valer j+2 y se repiten los pasos 1 y 2 hasta que se termine de recorrer toda la imagen.

Los píxeles no son modificados en ningún momento.

2.1. Pseudo código del algoritmo en C

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). Para la primera utilizamos dos registros, los cuales apuntan a la primera y segunda fila respectivamente. Luego, para la imagen resultante utilizamos 4 registros con las direcciones de las primeras posiciones de los 4 cuadrantes en los que se divide a la nueva imagen.

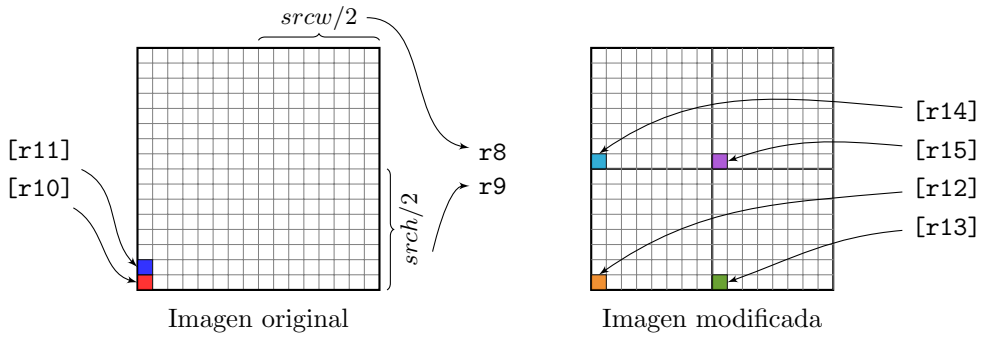


Figura 1: Esquema del contenido de los registros de propósito general utilizados en la implementación de *fourCombine* al comenzar la primera iteración del ciclo principal. Los registros `r10` a `r14` funcionan como punteros a las imagenes fuente y destino (representadas aquí como matrices de píxeles), 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.

También debimos inicializar un registro con el tamaño de la imagen, el cual utilizamos para comparar con 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.

Una vez que tenemos todo eso listo comienza el ciclo principal. En este lo que hicimos fue primero que nada hacer una comparación para ver si se llego al final de la imagen. En dicho caso el ciclo finaliza. En caso de no haber llegado al final se procede a copiar cuatro píxeles de la primera fila de la imagen, y cuatro de la segunda utilizando la instrucción `MOVDQU` y dos registros XMM. Luego utilizamos la instrucción `PSHUFD` y otros cuatro registros XMM para separar a los píxeles obtenidos en el paso anterior según el cuadrante que le corresponde a cada uno en la nueva imagen. Finalmente movemos las partes bajas de estos cuatro registros a las posiciones indicadas por los registros previamente inicializados con los cuatro cuadrantes.

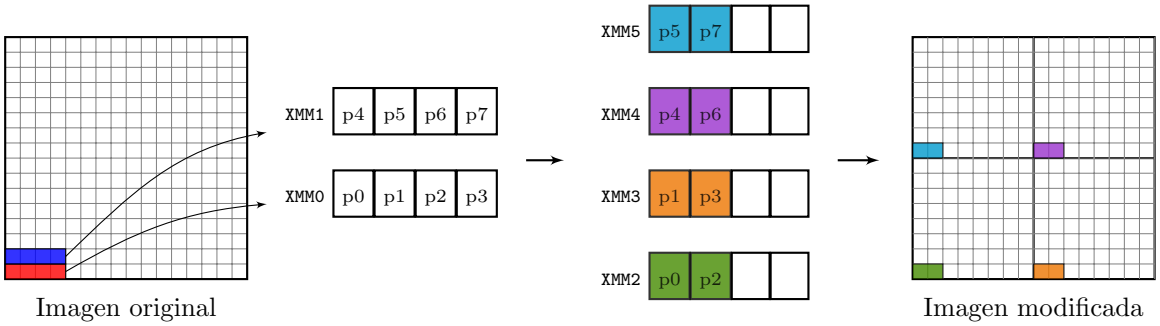
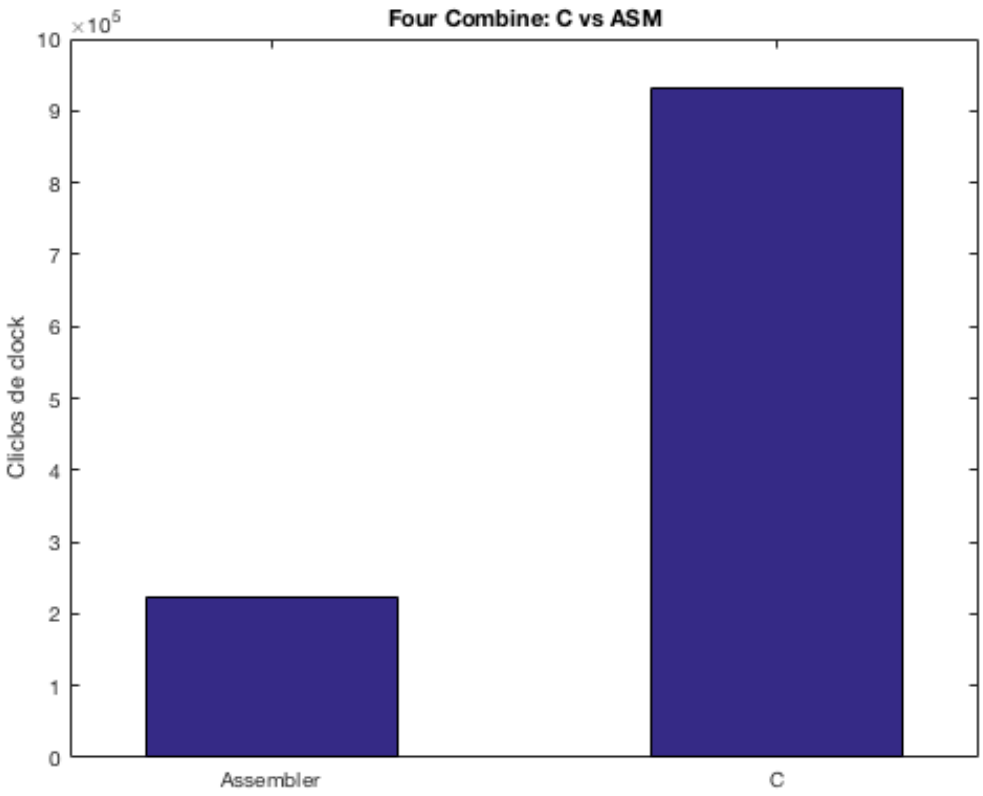


Figura 2: Una iteración del ciclo principal de *fourCombine*. Se copian 4 píxeles de la última fila de *src* y 4 de la penúltima (en rojo y azul, respectivamente) en los registros `XMM0` y `XMM1` con la instrucción `MOVDQU` luego de lo cual se realiza un shuffle (`PSHUFD`) para separar su contenido en 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. Los registros están representados con el bit menos significativo a la izquierda.

Por ultimo, aumentamos en 8 a los cuatro registros que apuntan a la imagen destino (2 píxeles cada uno), aumentamos en 16 a los dos que apuntan a la imagen fuente, y nos fijamos si llegamos 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 vuelve directamente al principio del ciclo y se repiten todos los pasos nuevamente. Notemos 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.

2.3. Experimentación

2.3.1. Comparación C vs Assembler



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

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.

(DIBUJO QUE MUESTRA EL RESULTADO DE APLICAR PSHUFD Y COMO QUEDA EL REGISTRO QUE CONTIENE 1-VAL)

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 hicimos fue ir guardando toda la matriz de 7x7 en registros XMM. Para ello guardamos en un registro un numero que sumándoselo al puntero a la imagen fuente, este pasa a apuntar 3 filas mas arriba y 3 píxeles hacia la izquierda del actual. Luego cargamos los 4 píxeles que comienzan en la nueva posición apuntada, en un registro XMM, sumamos 12 al puntero y cargamos otros 4 en otro registro. Acto seguido sumamos la longitud de una fila al valor descripto en el paso anterior, para ahora apuntar al comienzo de la fila que sigue en la matriz de 7x7 y cargamos otros dos registros y así hasta llegar al final de esta matriz. Notemos que de esta manera estaríamos cargando dos veces los píxeles que se encuentran en la columna del medio, pero no influye en absoluto en los resultados.

(DIBUJO DE COMO SE CARGA EL KERNEL DE 7X7 EN LOS 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.

(DIBUJO DE COMO FUNCIONA EL PMAXUB ENTRE TODOS LOS REGISTROS)

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.

(DIBUJO DE COMO SE COPIA Y SHIFTEA EL XMM DE MAXIMOS Y COMO A PARTIR DE ESO SE CALCULA EL MAXIMO DEFINITIVO)

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 *double word* y convertir el resultado de esto de *double quad word* 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.

(DIBUJO QUE MUESTRA COMO SE DESEMPAQUETA EL MAXIMO Y COMO SE HACE LA MULTIPLICACION CON VAL)

De forma análoga movemos a otro registro el píxel original, lo desempaquetamos y lo multiplicamos con el registro que contiene a `1-Val`.

Finalmente realizamos la suma de los registros que contienen a `Val` y a `1-Val` con la instrucción `ADDPS` indicando que es suma de *floats* nuevamente evitando perder precisión, convertimos el registro resultante a *double quad word* con la instrucción `CVTPS2DQ`, empaquetamos a *word* y luego a *byte* y movemos el píxel resultante a donde le corresponde en la imagen destino.

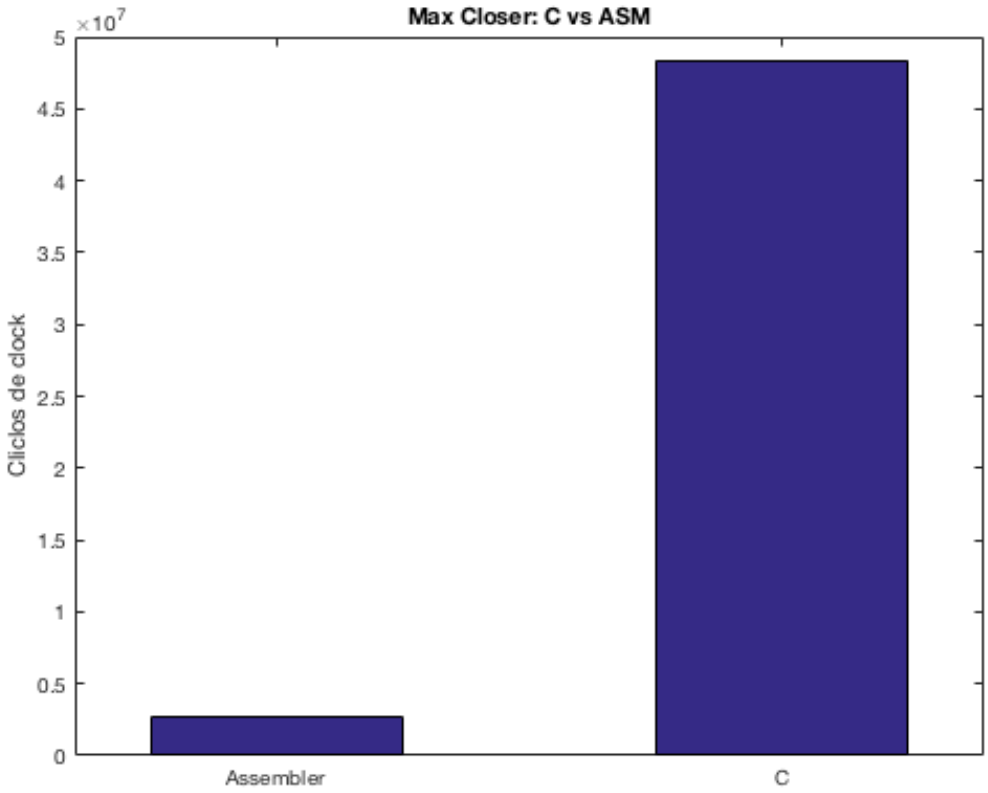
(DIBUJO QUE MUESTRA LA SUMA, EL EMPAQUETADO Y COMO SE MUEVE A LA IMAGEN DESTINO)

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*.



(FALTA DECIR ALGO SOBRE ESTE RESULTADO)

3.3.2. Recorrer imagen por columnas

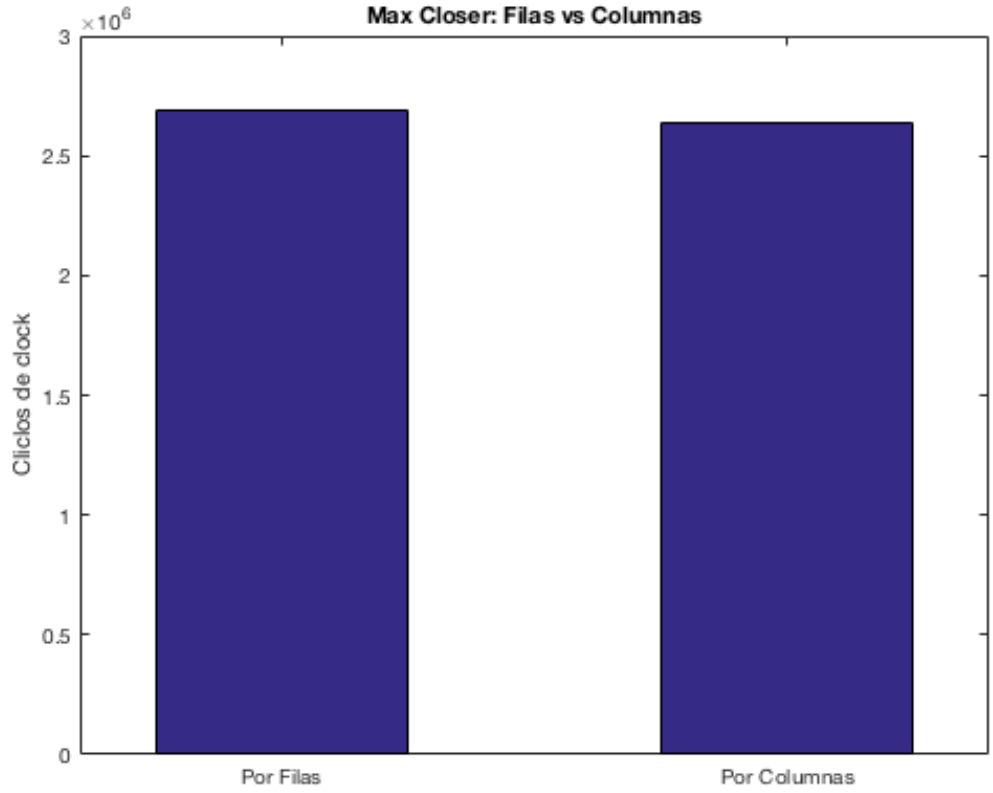
Como vimos en la sección anterior, la matriz generada por la imagen fue recorrida por filas. Cada vez que se terminaba de procesar un píxel, se le sumaba 4 al puntero que apuntaba a la imagen original y de esta manera se avanzaba al siguiente píxel en la fila. Con este experimento cambiaremos la forma 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.

(DIBUJO QUE MUESTRE COMO ANTES SE GENERABAN LOS KERNELS DE 7X7 UNO AL LADO DEL OTRO, CON 14 ACCESOS A MEMORIA, Y AHORA SE GENERAN UNO ABAJO DEL OTRO, CON 2 ACCESOS A MEMORIA, Y CUANDO SE LLEGA HASTA ABAJO SE SUBE PASANDO A LA SIGUIENTE COLUMNA Y SOLO SE REALIZAN 14 ACCESOS CUANDO SE

ESTA AL PRINCIPIO DE CADA COLUMNA)

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):



(FALTA DECIR ALGO SOBRE ESTOS RESULTADOS)

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.

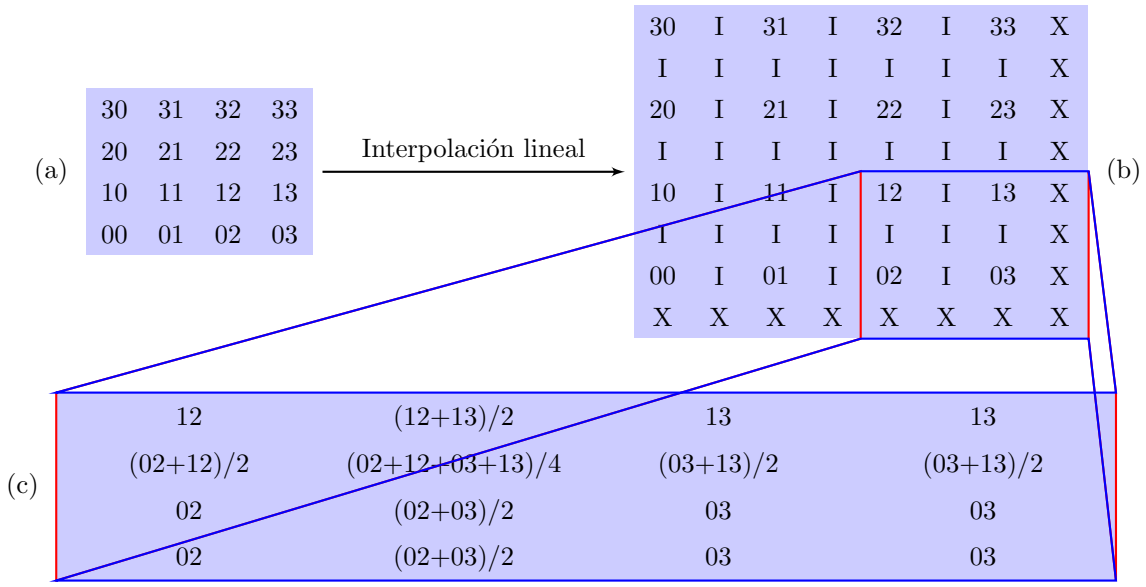


Figura 3: Esquema del funcionamiento general del filtro *linearZoom*. Se muestran la imagen original (a), la imagen generada a partir de la interpolación lineal (b) y un detalle de esta última donde se aprecia el contenido de los píxeles. Las operaciones que se presentan como entre píxeles se realizan componente a componente.

4.1. Pseudocódigo del algoritmo en C

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. [1] Mediante la adición del ancho de *src* multiplicado por 4 a `RBX`, se fueron recogiendo datos de dos filas simultáneamente.

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 ...

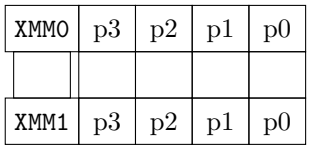


Figura 4: Otra figura más.

A continuación, se

[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.

Algoritmo 1 Your mother's algorithm

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 → dsth step 1 do
    dst[row][dstw − 1] ← dst[row][dstw − 2]
end for
```

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.

La razón de tomar información de dos filas simultáneamente fue que, al igual que en el ciclo anterior, era necesario calcular el promedio de las componentes de 2 píxeles vecinos (tanto en filas como en columnas) y el de los 4 que comparten un vértice con uno de `dst`. (??????????????????)

Las operaciones realizadas en este ciclo fueron las mismas que la sección del ciclo principal representada el la figura ... con la única diferencia de que, al finalizar, los vectores obtenidos tenían la forma

p_{m-1}	p_{m-1}	$(p_{m-1} + p_{m-2})/2$	p_{m-2}

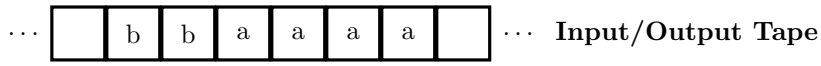
$(p_2 + p_1)/2$	p_1	$(p_1 + p_0)/2$	p_0

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

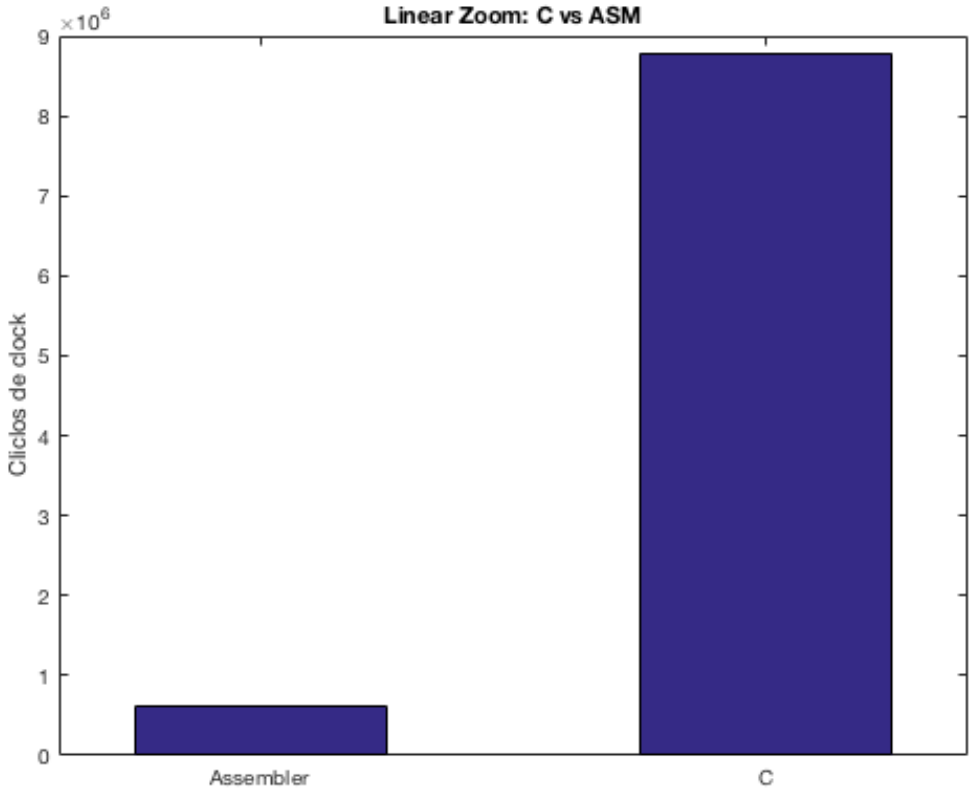
$(p_2 + p_1)/2$	p_1	$(p_1 + p_0)/2$	p_0

mediante un procedimiento análogo al realizado el el ciclo principal. La diferencia reside en que, a diferencia de lo que puede verse en la figura ..., aquí solamente se calcula el promedio de las componentes de los píxeles horizontalmente. 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. Esto último obedece al hecho de 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 ...



4.3. Experimentación

4.3.1. Comparación C vs Assembler



Parte III
Conclusiones