**Resumen**

En el presente trabajo se describe la problemática de ...

# Índice

**1. Objetivos generales** **2**

**2. Contexto** **2**

**3. Enunciado y solucion** **2**

**4. Descripciones** **2**
4.1. Cropflip . . . . . 2
4.2. Sepia . . . . . 4
4.3. LDR . . . . . 6

**5. Conclusiones y trabajo futuro** **6**

Figura 1: Descripcion de la figura
hola como va

# 1. Objetivos generales

El objetivo de este Trabajo Práctico es el de, mediante la implementacion y comparacion de distintos codigos de filtros de procesamiento de imagenes, el de ejercitar y demostrar nuestro entendimiento de las SIMD del assembler, asi como la capacidad de hacer un analisis comparativo entre las versiones de assembler y c, de sus vetajas y las limitaciones de cada codigo, asi como las reestricciones del filtro en si, de una manera clara y demostrativa.

# 2. Contexto

**Titulo del parrafo** Empezando 1.

```
struct Pepe {};
```

# 3. Enunciado y solucion

# 4. Descripciones

## 4.1. Cropflip

**Aridad** En el filtro cropflip se nos pasan los siguientes parametros:

- tamx: Cantidad de pıxeles de ancho del recuadro a copiar.
- tamy: Cantidad de pıxeles de alto del recuadro a copiar.
- offsetx: Cantidad de pıxeles de ancho a saltear de la imagen fuente.
- offsety: Cantidad de pıxeles de alto a saltear de la imagen fuente.

Se asumen que todos son multiplos de 16, y se toman estos dos parametros

-scr_row_size: tamano de una fila (en bytes) de la imagen de entrada
-dst_row_size: tamano de una fila (en bytes) de la imagen de salida

asumiendo a su vez que nos pasan los punteros a una imagen de entrada y de salida.

**Decripcion de cropflip.c**
En la implementacion de cropflip_c.c primero declaramos todas las variables que vamos a utilizar, y luego utilizamos un for anidado en otro para recorrer la imagen fuente y copiarla modificada a la imagen de destino.
Mas claramente, tenemos dos contadores j e i, representantes de las dimensiones del recorte que le hacemos a la imagen. j cuenta el numero de columnas e i el numero de filas. El primer for del ciclo incrementa las filas, mientras que el anidado incrementa las columnas, incrementando los contadores desde 0 hasta llegar a sus representaciones, realizando asi las acciones del filtro en toda la imagen.
En cuanto a las acciones del filtro, pixel x pixel, dentro del segundo for, utilizamos dos punteros a los pixels actuales de la imagen de entrada (source) y la de destino, llamados p_s y p_d.
El primero es direccionado a la posicion de i y j en cada ciclo, recorriendo la matriz pixel por pixel en el orden de las direcciones. (se multiplica por 4 j porque un pixel tiene 4 bytes) El segundo, por cada

entrada del ciclo, hace el sgte calculo. En el lado del las filas se va a la posicion que indica es la de inicio (marcada por un integer llamado offsety), se le suma el tamano de la imagen offset que queremos (guardada en tamx) y se le resta por cada ciclo el i de la fila actual(restandole 1 por el 0), quedando asi apuntando a la fila inversa a la que estamos en la de destino, como queremos hacer. Asimismo, en cada fila, nos colocamos en la columna necesaria sumandole a la posicion inicial dada en tamx(analogo a tamy) el valor actual de j ( $*4$ *, para avanzar de a pixel*). Alli, en cada ciclo el p_s esta apuntando la posicion donde esta el pixel que deberia estar en la p_d, por lo que lo unico que hay que hacer es igualar sus punteros y seguir el ciclo, y al terminar se teine cambiada toda la imagen.

**Decripcion de cropflip.asm**

Los parametros de cropflip_asm.asm estan pasados de la siguiente manera:
rdi = src
rsi = dst
edx = cols
ecx = filas
r8d = src_row_size
r9d = dst_row_size
[rsp+8] = tamx
[rsp+16] = tamy
[rsp+24] = offsetx
[rsp+32] = offsety
*extracto de asm
siendo src y dst los punteros a las imagenes de entrada y salida.

En la funcion, luego de hacer la pila y vaciar los regisrtos (cosa que, salvo el r12 y r14, hacemos por mera precausion) movemos a la parte baja de los registros los datos que nos quedaron en la pila para evitarnos multiples accesos de memoria en el ciclo, y luego multiplicamos r12 (que tiene el tamx) *4 para pasar de cantidad de pixeles a bytes. (cada pixel tiene 4 bytes)
Luego, se vacia rax y se le suma r15d (donde quedo guardado offsety) y r13d (donde quedo guardado tamy ) - 1. Hasta ahora rax = offsety + tamy - 1. (parecido a la posicion de filas de la version c) luego, lo multiplicamos por r8d (que es el tama;o de fila), al hacerlo, ahora rax contiene el valor necesario para que un puntero, al sumarselo, avance offsety + tamy - 1 filas. Naturalmente, le sumamos esto a rdi, que tiene el puntero a la direccion inicial de la imagen de entrada, para moverla a la fila que queremos. luego, direccionamos rdi, a la columna pedida usando r14 (offsetx) x 4 (bytes) y un lea. Posicionados donde debe iniciar el ciclo (notar que estamos en la misma posicion en que un ciclo de la version c estaria en la primera ejecucion);
Entramos al ciclo. (vamos a usar vaciado rax y rbx como contadores/punteros) Este es muy sencillo. Usamos a xmm0 para mover 4 pixeles, desde contenido de rdi + rbx al contenido de rci + rbx. a rbx, puntero a columna actual, le sumamos lo necesario para ir al siguiente pixel a procesar y lo comparamos con la cantidad de columnas que nos piden, si es menor, seguimos en la misma fila, por lo que volvemos al ciclo. Si no es el caso, entonces debemos pasar a la fila siguiente. Vaciamos rbx para que el contador vuelva a la columna 0 (podemos hacer esto porque los tamanos son multiplos de 4); reducimos rdi x una fila (r8d) y le agregamos a rsi una (en r9d, son iguales). Notar como estas orden son paralelas al incremento de i en la implementacion c (en el direccionamiento).
Incrementamos rax para contar que pasamos una fila, y lo comparamos con la cantidad de filas a pasar, si es menor, volvemos al ciclo, si no lo es, toda la imagen esta pasada, por lo que desarmamos la pila y salimos.

En esta version, debemos mover el puntero de la imagen de entrada, pero hacemos este direccionamiento una vez y no muchas, ya que luego vamos restando. Tambien, como procesamos varios pixeles a la vez, tarda menos ciclos.

## 4.2. Sepia

**Aridad**
En el filtro sepia definimos los siguientes parametros (que vienen implicitos con la imagen de entrada, o nosotros definimos ):

src = puntero a direccion de pixel inicial de la imagen de entrada
dst = puntero a direccion de pixel inicial de la imagen de salida
cols = cantidad de columnas de la imagen fils = cantidad de filas de la imagen src_row_size = tamano de fila de la imagen de entrada dst_row_size = tamano de fila de la imagen de salida
Se asumen que todos son multiplos de 16.

**Decripcion de sepia.c**
En la implementacion de sepia_c.c tenemos de nuevo un doble ciclo anidado, usando a j e i como contadores de nuevo, representado lo mismo que en cropflip_c.c. Lo mismo ocurre con p_d y s_d. El recorrido de los for es el mismo, tambien, siendo lo unico que cambia lo que ocurre en el for anidado por cada iteracion del cilo hasta la salida de este, por lo tanto, focalizaremos en eso.
En este filro, a diferencia del cropflip, cada pixel del de entrada se corresponde con el de salida, por lo que tanto para p_d como para s_d se direccciona a la fila i de la columna j.(se multiplica por 4 j porque un pixel tiene 4 bytes)
Luego, por lo que se hace una vez posicionado, basicamente es la formula del sepia. Se utiliza un short suma, al cual se le suman los valores de rojo, vere y azul. y luego a cada parte del pixel actual de la imagen de salida se le asigna esa suma multiplicada por se consiguiente numero. (siendo, 0.2 para azul, 0.3 para verde, 0.5 para rojo, y manteniendo igual el a)
En cuanto al rojo, para calcular el valor que va destinado a ese lugar se shiftea la suma a la izq (dividiendo x 2), y, como es el unico valor donde pudo haber quedado overflow, se le pregunta si no es mayor al valor maximo de unsigned de byte, si lo es, se reemplaza al valor maximo. Se satura el valor.

**Decripcion de sepia.asm**
En la implementacion de sepia_asm.asm nos pasan los sgtes parametros en los sgtes registros
rdi = src
rsi = dst
edx = cols
ecx = filas
r8d = src_row_size
r9d = dst_row_size
*extracto de sepia_asm.asm
Primero armamos la pila y como en el cropflip_asm.asm multiplicamos *4 (con shift) las columnas para que sea el numero de bytes x columna, en vez de pixels. Luego, vaciamos rax y rbx para usarlos de contador/punteros, y vaciamos la parte alta de los registros r8 y r9, que luego usaremos.
En el ciclo movemos xmm0 a la direccion del puntero a la imagen de entrada (rdi) + el contador de columnas, y agarramos, 4 pixeles. Luego, copiamos el registro en xmm1 y xmm9 con movdqa (alineado por asumpciones)
Luego, aplicamos punpckhbw con xmm8 (que, antes de entrar al ciclo, pusimos en 0) que pone en el registro de destino los bytes con mas significativos de ambos registros intercalados, siendo el mas alto el mas alto del registro de entrada (source). En este caso, xmm8 es la entrada, por lo cual el efecto es el de extender los dos primeros datos de bytes a words, contenidos en xmm0. Usamos, una instruccion analoga que se refiere a la parte baja para hacer lo mismo con xmm1, ahora teniendo los 4 pixeles en dos registros, con cada elemento siendo de tamano word.
Copiamos el contenido del registro en xmm0 en xmm2 y el de xmm1 en xmm3 y los shifteamos a la derecha(de a quadruple, osea, de a mitades de registro) 8 lugares,para poner los a en 0 (cave resaltar que cambiamos el orden de rojo, verde y azul, tambien).
Luego sumamos ambos registros horizontalmente con la instruccion phaddw. Esta, como su nombre lo

indica, ve alregistro destino como un lugar para 8 words, y rellena las cuatro posiciones mas significativas de este agrupando los datos que habia en ella, de dos en dos, siendo la posicion (word) mas significativa nueva la suma de las dos words mas altas de el destino antes de la suma, y sucesivamente. Para las cuatro posiciones menos significativas del registro destina, realizaba el mismo agrupamiento pero sumando las words del operando fuente en lugar del de destino. Nosotros realizamos estas suma entre xmm2 y xmm3, y luego del registro destino entre estos (xmm2) consigo mismo para conseguir la suma de elementos del pixel. seria un poco asi:

Registros antes de la suma horizontal (agrupados x words)
$xmm2 : |r0|g0|b0|0|r1|g1|b1|0|$ $xmm3 : |r2|g2|b2|0|r3|g3|b3|0|$
luego de la primera suma horizontal (solo nos importa el destino)
$xmm2 = |r2 + g2|b2|r3 + g3|b3|r0 + g0|b0|r1 + g1|b1|$
luego de la segunda suma horizontal
$xmm2 = |r2+g2+b2|r3+g3+b3|r0+g0+b0|r1+g1+b1|r2+g2+b2|r3+g3+b3|r0+g0+b0|r1+g1+b1|$

ahora en xmm2 tenemos la suma de elementos para 4 pixeles (repetida dos veces) lo copiamos en xmm4 y lo shifteamos x word a la izquierda (/2 cada palabra), astutamente asi obtenemos los nuevos componentes rojo de cuatro pixeles
Ahora usamos la instruccion punpckhwd entre smm2 y xmm8 para hace algo parecido a lo de antes, pasar los datos en xmm2 de word a doubleword(perdiendo los 4 menos significativos, pero como se repiten no nos importa) luego usamos la intruccion cvtdq2ps para convertir los datos a floats (porque vamos a mutiplicar con floats), lo copiamos a xmm3 y multiplicamos, usando mulps, xmm2 con xmm5 y xmm3 xon xmm6 (donde se guardan coefb y coefg, respectivamente) que multiplica floats de precision simple. Ahora tenemos en xmm2 y xmm3 los valores de azul y verde nuevos de los cuarto pixeles procesados. Usamos cvttps2dq para volver a doublewords enteras (que son mas rapidas)
Ahora solo nos queda agrupar los pixeles de manera correspondiente., usamos packusdw entre xmm2 y xmm3 para, ahora que no tenemos que hacer mas operaciones, convertirlas en words (el objetivo aqui es volver a tener los 4 pixeles en un registro) de manera saturada.(la parte alta de cada doubleword deberia estar en 0, asi que no hay problema) packusdw hace eso, en las 4 posiciones mas significativas los de source achicados, en los otras 4 las del destino.

Luego, hacemos packuswb, que hace lo mismo que la instruccion anterior perod e words a bytes, entre xmm2 y xmm4. Al final el registro resultante es este:
xmm2 = $|r2'|r3'|r0'|r1'|r2'|r3'|r0'|r1'|g2'|g3'|g0'|g1'|b2'|b3'|b0'|b1'|$

En este registro, aunque logramos juntar todos los componetes que queriamos (excepto las a) tenemos el problema de que las cosas estan rotadas (por lo de antes con el a, ) y no estan los pixels en orden.
Para arreglarlo, usamos pshufb entre xmm2 y xmm7. (en xmm7 esta guardada una mascara que definimos en section,data). Pshufb intercambia los bytes del registro de destino de acuerdo a las posiciones instruccionadas en el registro fuente para cada uno. (si pongo ff se pone 0)
En este caso:

Lo que tengo en xmm2 (arriba la posicion de registro, abajo el contenido)
$|f|e|d|c|b|a|9|8|7|6|5|4|3|2|1|0|$
$|r2'|r3'|r0'|r1'|r2'|r3'|r0'|r1'|g2'|g3'|g0'|g1'|b2'|b3'|b0'|b1'|$

Lo que quiero en xmm2
$|f|e|d|c|b|a|9|8|7|6|5|4|3|2|1|0|$
$|0|r0'|g0'|b0'|0|r1'|g1'|b1'|0|r2'|g2'|b2'|0|r3'|g3'|b3'|$

La mascara en xmm7 (copiada antes del inicia del ciclo, definida en section.data), en hexagesimal.

$|f|e|d|c|b|a|9|8|7|6|5|4|3|2|1|0|$

0x02, 0x06, 0x0a, 0xff, 0x03, 0x07, 0x0b, 0xff, 0x00, 0x04, 0x08, 0xff, 0x01, 0x05, 0x09, 0xff

De esta manera, gracias a esta instruccion, casi termino de reubicar los pixeles, ahora estan en orden. SOlo falta colocar las a. Para ello, agarro xmm9, ( xmm9 : $|a0|r0|g0|b0|a1|r1|g1|b1|a2|r2|g2|b2|a3|r3|g3|b3|$) y le aplico psrld, que shiftea de a doublewords hacia la derecha, y la cantidad de lugares a shiftear (debo llenar con 0 tres elementos, cada uno tiene 8 bytes, entonces multiplico por 3 por eso).
Utilizo psrld inmediatamente despues (con los mismos parametros) para que las a del registro xmm9 quede en los lugares libres de el registro xmm2; entonces sumo de a byte estos dos (usando paddb) y muevo el resultado a la direccion de la imagen de destino correspondiente.

Para terminar, sigo el ciclo analogamente a como lo hice en cropflip, con la excepcion de que ahora tanto rdi como rsi avanzan a la fila siguiente (en vez de a la anterior), y al terminar de recorrer la imagen desarmo la pila y salgo.

### 4.3. LDR

## 5. Conclusiones y trabajo futuro