Resumen

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

Índice

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. Enunciado y solucion

3. Descripciones

3.1. 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. Cualquier tipo de shifteo, si pongo un inmediato para indicar la cantidad, son bits

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.

3.2. 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.Cualquier tipo de shifteo, si pongo un inmediato para indicar la cantidad, son bits

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 que esta delimitado por filas en vez de tamy y cols en vez de tamx (lo cual tiene logica porque estamos querien ver la imagen fuente completa, no solo una parte) y lo que ocurre en el for anidado por cada iteracion del ciclo 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
$xmm2 = |r2+g2+b2|r3+g3+b3|r0+g0+b0|r1+g1+b1|r2+g2+b2|r3+g3+b3|r0+g0+b0|r1+g1+b1|$

La notacion usada separa cada xmm en 8 posiciones de word, en donde cada una anota el contenido de la siguiente manera... a,b,r u g representa el elemento de un pixel (r es rojo, b azul, g verde, y a el de alineacion). A su vez, el numero que la acompana representa cual de los pixeles de los 4 que proceso es el que esta guardado en esa posicion y le corresponde ese elemento.

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 con 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 = $|r2'|r3'|r0'|r1'|r2'|r3'|r0'|r1'|g2'|g3'|g0'|g1'|b2'|b3'|b0'|b1'|$

(ahora veo las posiciones del registro de a bytes, por lo que hay 16 posiciones, empezando del 0)

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.

3.3. LDR

Aridad
En el filtro LDR se nos pasan los siguientes parametros:
- alpha: Valor entre -255 y 255 que indica la intensidad del filtro.
Ademas, asumimos como parametro de entrada implicitos:
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
Las filas, columnas pasadas son multiplos de 16. Cualquier tipo de shifteo, si pongo un inmediato para indicar la cantidad, son bits

Aclaraciones

En la implementacion de ldr_c.c, para una lectura mas clara del codigo, definimos unas cuantas etiquetas que aclarare al momento y se usaran en el codigo. Estas son
MIN (x,y), que dados dos paramentros comparables nos devuelve el minimo
MAX (x,y), la contraparte de minimo
P = 2 y sirve para bordes.
MAXSUM = 4876875, una constante de la formula del filtro.
Tambien, vamos a utilizar el termino, pixeles vecinos, para referirnos a los pixeles continuos al procesado (horizontal, vertical u diagonalmente) y continuos a estos. (en un contorno de 3x3 al pixel actual en el filtro)

Decripcion de LDR.c

En la funcion en si tenemos dos ciclos anidados, con los mismos bordes y avances que el sepia, por lo cual no andaremos en detalles. En cuanto a las acciones que realiza cada filtro en cada iteracion, estan descriptas a continuacion:

Nos posicionamos en la fila i, en la columna j (*4, por la cantidad de bytes que ocupan los pixeles) de tanto el puntero a la imagen de entrada como el de salida (que definimos p_s y p_d, respectivamente) como este filtro no jega con las posiciones pero la distribucion de la paleta, recorremos las dos imagenes

paralelamente, avanzado de a pixel x iteracion de ciclo, por toda una fila para el anidado, y por todas las filas en el ciclo completo.
Luego, debido a que los bordes de la imagen no deben ser modificados (contorno de 2 pixeles), usamos un if para comprobar si i o j representan las primeras columnas o filas, o las ultimas. En caso de que la guarda se cumple, simplemente designamos al puntero que apunte a la posicion de entrada, aprovechandonos del aliasing para pasar incambiado el valor a la imagen de salida. y continuamos el ciclo.
De no ser asi, estamos en uno de los pixeles a los que debemos aplicarle la formula del filtro, Definimos 3 long long r, g y b (Necesitamos 64bits porque al hacer ultima la division el numero mas grande posible es 255*255+255*4876875=4A20DEB6), y hacemos un subciclo de for (con un for anidado) en el cual sumamos, sumamos en el r los componentes de color del pixel; el contenido de sus vecinos en un recuadro de 3x3, como mostraba la formula del enunciado. (utilizamon un puntero que va apuntado al pixel iterado, se muve sumandole a p_s (fila actual en recorrido principal) el apunte a fila de vecino actual(l), y le sumamos k*cols para posicionarnos en la columna del vecino a procesar/ El p definido se usar para delimitar los marcos del subciclo, en cada for del subciclo yendo desde -p hasta p, avanzando de a uno).
Ahora que tenemos la suma en r, la multiplicamos por el alfa pasado, y los copiamos a b y g.(solo multiplicamos en 1 x que es mas rapido)
Realizamos la formula de componentes en r, g y b (en facto, multiplicamos el contenido por el valor del componente del pixel dactual homologo, lo dividimos x MAXSUM y le sumamos a eso el valor del comoponente del pixel actual)

Con esto, ya tenemos en r g, b cada valor que queriamos para cada componente del pixel, salvo que puede estar desfasado, por haber superdao el max o minimo permitido. Para evitar eso, a cada componente del pixel de salida (al cual nos referimos con el puntero declarado anteriormoent p_d) y colocamos en el, el minimo entre 0 y el maximo de r, g o b y UCHAR_MAX(dado en limits.h), segun corresponda, saturandolo tanto superoior como inferiormente.
Seguimos el ciclo hasta que termine, cuando ya aplicamos el ciclo para toda la imagen, y salimos.

Decripcion de LDR.asm
En la implementacion de asm nos pasan los siguientes parametros en los sgtes registros:
rdi = src
rsi = dst
edx = cols
ecx = filas
r8d = src_row_size
r9d = dst_row_size
rbp+8 = alpha (la pila)
*extracto de ldr_asm.asm
Tambien, al principio del programa, tenemos definidos varias constantes que vamos a utilizar en el programa. En la descripcion diremos su nombre su nombre entre y entre parentesis () el valor definido en ellos. (ej: ejemplo(0).
La excepcion son las etiquetas max y max_f, definidas en las section. data del codigo en las cuales se definen 4 doublewords/floats de precision simple consecutivas respectivamente, (esto representa la constante max que requiere la formula del filtro) En particular
max: 4876875, 4876875, 4876875,48676875
max_f: 4876875.5, 4876875.5, 4876875.5, 4876875.5

En cuanto a la funcion en si, primero armamos la pila. Luego, armamos los contadores, para lo cual limpiamos la parte alta de los registros que vamos a usar (shifteando, para evitar ir a memoria) o los vaciamos con xors. Movemos a r10 el alpha para no acceder a memoria coda vez que lo necesitemos, y guardamos en r14d el numero de filas y en r15d el de columnas. Vamos a usarlos como representantes de los bordes, por lo cual le restamos a r14 2, para otener el indice de fila en la cual tenemos que dejar que procesar, y 4 a r15, para obtener la columna en la cual debemos dejar de procesar. (*desfasado a 2, o sea cuando rdi es cero procesamos los pixeles (i,2),(i,3),(i,4),(i,5), siendo i la fila actual*)

Utilizamos r11 como el indice de fila, y r12 como el de columna, (antes de entrar al ciclo xorceados a 0) y movemos a xmm14 max y xmm15 max_f, para cuando los necesitomos no ir a memoria

El recorrido del ciclo en si es de la continua manera:

Nos fijamos en cada iteracion, si la direccion a la que apuntamos, pertenece a alguno de los bordes. Para ello, ponemos una etiqueta borde_izq, donde se compara al contador de columnas con (r12d) con 2. Si es menor entonce es del borde y lo unico que hace es mover la direccion apuntada por el puntero a la imagen de entrada (rdi) + la columna actual (r12 * 4 x lo de los pixeles) a r13, y este a la direccion apuntada por la direccion de salida (rsi) + la columna actual (fijarse que rdi y rsi tienen un aumento paralelo en toda la ejecucion del programa).
Notar ademas que aunque estamos manejando pixeles estamos usando el registro r13, esto se debe a que solo dos pares de columnas(conjuntas) pertenecen al caso borde, por lo que si usaramos un xmm para hacer el cambio, agarrariamos mas pixeles de los cuales se constituye la excepcion. Lo mismo ocurre en el borde_der, donde tambien los usamos.
Sucesivamente, si r12d fuera mayor o igual a 2, u omitiendo estos pasos, nos metemos en la etiqueta borde_inf, donde nos fijamos si el contador de filas (r11) es menor a 2. Si lo es , realizamos la misma transaccion que en la etiqueta anterior entre imagenes de entrada y salida (pero usando xmm0 como intermediario, ya que en este borde se puede llevar de cuatro pixeles, y sin miedo al desfasaje, por ser multiplo de 16), si no lo es (o luego de hacerlo) entramos a la etiqueta borde_sup, donde comparamos que r11d sea menor a r14d (donde debiamos dejar de iterar filas) y si lo es pasamos a borde derecho.Si no lo es movemos sin cambiar analagomente a como antes, y le agregamos 4 a r12d(lo movemos a la siguiente columna).
Comparamos r12d con la cantidad de columnas (recuerden, las r12 = columna actual).Si es menor volvemos al loop, si no, vaciamos r12, incrementamos la fila (r11), los punteros de fila en la imagende destino y fuente (rdi, rsi), y nos fijamos si r11 llego a la cantidad de filas pasadas(en ecx), si no es asi terminamos, si no volvemos al loop.
Por otro lado, si llego al borde_der, nos fijamos que r12d sea menor que r15d (indice columna tenemos que dejar de procesar). Si no lo es, pasamos usando r13 los bits apuntados x r12 (*4 x pixel, + 8 porque estamos dos de desfasaje) le agregamos 4 a r12d (proxima columna) y volvemos a comparar si alcanzo el numemo de columnas, y basicamente lo mismo descripto en el parrafo anterior, culminando en repetir el ciclo o terminarlo.
En caso de ser menor, la direccion apuntada no forma parte de los bordes, (ya que para llegar hasta aqui tuvo que haber pasado todos las reestricciones de bordes) y es una direccion a la que debemos aplicarle el filro.
Una cosa importante a aclarar en el funcionamiento del ciclo es que solo se pueden llegar a incrementa rdi, y rsi si se llega al limite superior o derecho, y que no pueden llegarse al limite superior e inferior a la vez (a menos que pasen una imagen 4x4 u menor, en la cual este filtro no haria nada), y lo mismo con izquierda y derecha, por lo que si hay algo para procesar, luego de las primeras dos iteraciones iremos a procesary no habremos movido el rdi, rsi de la posicion inicial.
Esto es importante porque estamos aplicando el filtro desfazadamente, de manera que si rdi apunta a (i,j) nosotros procesamos (i+2, j+2),(i+2,j+3),(i+2,j+4),(i+2,j+5). Si hubieramos modificado rdi, o rsi, el desfasaje haria que dejaramos lugares sin pasar. (r12 en cambio, apunta al primer pixel a procesar x iteracion)

Luego de aclarado esto, pasemos a lo que se hace una vez estamos en condicion de hacerlo. Primero nos fijamos si r12d es 0 (osea, si empezamos fila nueva)

Si lo es, leemos con rbx la direccion del contenido de rdi(fila actual) y lo movemos a xmm0.
xmm0 = $|a03|r03|g03|b03|a02|r02|g02|b02|a01|r01|g01|b01|a00|r00|g00|b00|$
con rbx y direccionamiento, movemos a xmm5 los 4 pixeles continuos (derecha) con movdqu tambien; luego incrementamos rbx en r8 (tamano de fila de entrada, ahora rbx apunta a misma columna de la fila siguiente) movemos los 8 pixeles a xmm1 y xmm6, analogamente a como antes volvemos a incrementar

la fila apuntada por rbx (de igual manera) y volvemos a mover los 8 pixeles a xmm2 y xmm7, respectivamente.
En esta fila, iniciando en la parte alta de xmm8 y hasta la parte baja de xmm15, estan los 4 pixeles a procesar, por lo que para obtenerlos movemos a xmm8 y xmm15 los datos de los pixeles de xmm2 y xmm7 luego usamos psrldq (ya lo hemos iso en sepia) en xmm8 con 8, para limpiar la parte alta del registro..de manera analoga, usamos pslldq para limpiar la baja de xmm15. Sumamos rxmm15 y xmm12 , de a bytes, y guardamos el registro en xmm15. Luego, tenemos en r15 los pixeles efectivos a modificar.
Volvemos a aumentar de fila y mover los pixeless a xmm3 y xmm8, respectivamente, y una ultima vez mas, ahora guardandolos en xmm4 y xmm9.

En este momento, el contenido de los registros modificados:
xmm4 = $|a43|r43|g43|b43|a42|r42|g42|b42|a41|r41|g41|b41|a40|r40|g40|b40|$
xmm9 = $|a47|r47|g47|b47|a46|r46|g46|b46|a45|r45|g45|b45|a44|r44|g44|b44|$
xmm3 = $|a33|r33|g33|b33|a32|r32|g32|b32|a31|r31|g31|b31|a30|r30|g30|b30|$
xmm8 = $|a37|r37|g37|b37|a36|r36|g36|b36|a35|r35|g35|b35|a34|r34|g34|b34|$
xmm2 = $|a23|r23|g23|b23|a22|r22|g22|b22|a21|r21|g21|b21|a20|r20|g20|b20|$
xmm7 = $|a27|r27|g27|b27|a26|r26|g26|b26|a25|r25|g25|b25|a24|r24|g24|b24|$
xmm1 =$|a13|r13|g13|b13|a12|r12|g12|b12|a11|r11|g11|b11|a10|r10|g10|b10|$
xmm6 = $|a17|r17|g17|b17|a16|r16|g16|b16|a15|r15|g15|b15|a14|r14|g14|b14|$
xmm0 =$|a03|r03|g03|b03|a02|r02|g02|b02|a01|r01|g01|b01|a00|r00|g00|b00|$
xmm5 =$|a07|r07|g07|b07|a06|r06|g06|b06|a05|r05|g05|b05|a04|r04|g04|b04|$

y xmm15 =$|a25|r25|g25|b25|a24|r24|g24|b24|a23|r23|g23|b23|a22|r22|g22|b22|$
(los pixeles a procesar) (en el procesamiento no se modifican los registros de xmm5 a xmm9)

La notacion usada separa cada xmm en 16 posiciones, en donde cada una anota el contenido de la siguiente manera... a,b,r u g representa el elemento de un pixel (r es rojo, b azul, g verde, y a el de alineacion). A su vez, de los dos numeros que acompana cada letra, el que es contiguo representa la fila del vecino en la que me encuentro (siendo 0, dos filas + abjo, 1 una fila mas abajo, 2 en la fila actual, 3 una fila mas arriba, y 4 2 filas mas arriba) y el segundo, la columna vecina (de igual manera pero de 0 a 7, porque agarro ocho columnas por iteracion (ya que proceso 4 pixeles en cada una))
Agregamos 8 a rbx para estar listo en la proxima iteracion y vamos a modicar estos.

Si no estamos en la primera columna de la fila, podemos aprovecharnos de los datos cargados en estos registros, donde, de los registros que no modificamos para aplicar el filtro, seguro hay alguno con contenido que volveremos a necesitar en la proxima iteracion. El proceso de cargado es mu parecido, simplemente movemos la parte xmm5 a xmm0, xmm6 a xmm1, xmm7 a xmm2, xmm8 a xmm3, xmm9 a xmm4, y en los anteriores, ubicamos el contenido de lo apuntado x rbx
(rbx apunta a la fila rdi + la columnna r12(*4 x pixel) + 8 (por que ahora ya tenemos la parte derecha de los vecinos cargada) a xmm5, sumamos fila y ahora movemos lo apuntado por rbx a xmm6, denuevo a xmm7, volvemos a guardar el pixel a procesar en xmm15 (igual a como se hace al principio de la fila); se vuelve a sumar fila a rbx (sumando r8), se mueve contenido a xmm8, se vuelve a sumar a rbx y se pone el contenido en xmm9.
Asi, tenemos de nuevo cargado los pixeles a procesar, utilizando la mitad de llamados a memoria.

Una vez que tengo en los registros cargados, hacemos un puncpckbw (ya usado) entre xmm10 (contiene basura) y xmm0. y luego usando psrlw xmm10 (era el destino) y shiftenado 8 cada word, limpiando la basura y obteniendo los 2 pixeles menos significativos, con cada uno de sus elementos ahora en tamano word.
Solo para quedar mas claro

Antes de punpcklbw (en tamano byte)
xmm0 = $|a03|r03|g03|b03|a02|r02|g02|b02|a01|r01|g01|b01|a00|r00|g00|b00|$

xmm10 = $|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|*|$
* significa basura
luego de punpcklbw (en tamano byte)
xmm0 = igual
xmm10 = $|a01|*|r01|*|g01|*|b01|*|a00|*|r00|*|g00|*|b00|*|$
Luego del psrlw (en tamano byte)
xmm0 = igual
xmm10 = $|0|a01|0|r01|0|g01|0|b01|0|a00|0|r00|0|g00|0|b00|$
Hacemos lo mismo con la parte alta de xmm0, (con punpckhbe)ahora guardandp en xmm11.
Despues, hacemos el mismo desempacamniento de xmm1, ahora la parte baja en xmm0 (que ya no necesitamos mantener) y en xmm12 la alta.
Sumamos de a word (los elementos de los pixeles estan separados de a word) xmm10 y xmm0; y tambien xmm11 y xmm12 (lo hacemos con paddw) Hacemos lo mismo con el contenido de xmm2, xmm3 y xmm4 (usando xmm0 y xmm12 como registros temporales.)

Tenemos en xmm10 : $|p41+p31+p21+p11+p01|p40+p30+p20+p10+p00|$
Tenemos en xmm11 : $|p43+p33+p23+p13+p03|p42+p32+p22+p12+p02|$
(donde p, dado que i sea el numemo significa ri +bi +gi)

usamos psllq en ambos (con 8) para vaciar, de cada p, se valor a (rotamos los valores tambien) luego sumamos horizontalmente xmm10 y xmm11, y luego xmm10 con sigo mismo (como en sepia) para que nos quede (tanto en la parte alta como en la baja del registro) las sumas que queremos. lo movemos a xmm0.
Hacemos lo mismo (nada mas que empezando con xmm10, y ahora usando xmm1 y xmm2 como registros temporales) con las columnas de la derecha, de manera que al final queda:

xmm0: $|p43+p33+p23+p13+p03|p42+p32+p22+p12+p02|p41+p31+p21+p11+p01|p40+p30+p20+p10+p00|p43+p33+p23+p13+p03|p42+p32+p22+p12+p02|p41+p31+p21+p11+p01|p40+p30+p20+p10+p00|$
(los ai de cada p en 0, y debido a las sumas horizontales, cada $pij = |rij + gij + bij|$)

xmm10 :$|p47+p37+p27+p17+p07|p46+p36+p26+p16+p06|p45+p35+p25+p15+p05|p44+p34+p24+p14+p04|p47+p37+p27+p17+p07|p46+p36+p26+p16+p06|p45+p35+p25+p15+p05|p44+p34+p24+p14+p04|$

utilizamos punpckhwd (que hace lo mismo que punpcklbw pero con words en vez de bytes y consiguiendo double words en vez de words) en xmm0 con sigo mismo para poder, limpiando la parte alta de cada doubleword (con psrld , y constante 8) tener en xmm0 todo lo que tenia anteriormente, en doublewords (las partes repetidas las vacie en el shifteo) hago lo mismo con xmm10.
Guardo xmm0 en xmm1. Luego, shifteo a la derecha 4 con psrldq para que quede ahora la doubleword menos significativa del registro sea la que antes esta en la 1ra posicion del registro (viendo las posiciones como de 0 a 3, de un doubleword cada una)y se la sumo a xmm0, el objetivo de esto es obtener cada 0 de xmm0 la suma de todos los sus vecinos contenidos en este registro, para lo cual es necesario repetir este procedimiento 2 veces mas.

Ahora, en xmm0:$|p43+p33+p23+p13+p03|p43+p33+p23+p13+p03+p42+p32+p22+p12+p02|p43+p33+p23+p13+p03+p42+p32+p22+p12+p02+p41+p31+p21+p11+p01|p43+p33+p23+p13+p03+p42+p32+p22+p12+p02+p41+p31+p21+p11+p01+p40+p30+p20+p10+p00|$

Ahora, sumamos a xmm0 con xmm10 (todas la sumas en esta parte se hacen con paddd, para sumar de a doubleword). Al hacerlo, en la posicion 0 de xmm0 ya tenemos todos los componentes de pixeles vecinos sumados del pixel menos significativo, resultado al que nos referiremos como sumargb(22).
Para obtener la suma correspondiente a los otros 3 pixeles que procesamos, shifteamos a la izquierda xmm10 con pslldq y 4, (limpiando posicion mas baja ocupada) y se lo sumamos a xmm0. Hacemos esto 2 veces mas.
Ahora, en xmm0:$|sumargb25|sumargb24|sumargb23|sumargb22|$

Por otro lado, utilizo pinsrd ;que copia una doubleword del operando fuente, (r10d tiene solo una) y lo copia en el operando destino; de acuerdo al tercer operando inmediato, con xmm3, r10d(donde estaba el alpha) y 0. Osea que muevo el alpha a la posicion 0 de xmm3.
Luego, utilizo pshufd en xmm3, consigo mismo y 0 para replicar el alpha entodas las doublewords del registro
(explicacion: la operacion copia doublewords del operando , y los inserta en las posiciones segun el tercer operando inmediato pasado. este inmediato es de ocho bits, donde lo que hay en los 2 bits menos significativos indican cual

posicion del nregistro fuente es pasada a la posicion 0 de la del destino, los siguientes 2 con la posicion 1 y asi con los 8. Como queremos la posicion del fuente replicada en todas las posiciones del destino, alcanza con un 0 de 8 bits).
Ahora, para seguir la formula, multiplico coda doubleword de xmm3 (las sumas) por alpha, usando pmulld (con xmm10 como destino), y lo copio a xmm1 y xmm2. xmm0, xmm1 y xmm2 tienen el mismo valor.
Ahora movemos xmm15 (los pixeles principales) a xmm3. shifteamos de a doubleword a la izquierda 24, para quedarnos solo con los azules de cada pixel, (y luego 3 shifteamos a la derecha 24, para ponerlos en la parte baja de cada doubleword). luego, copiamos xmm15 en xmm4 y shifteamos pero con 16 a izq, 24 a derecha, para quedarme con el verde. y lo mismo con el rojo y xmm10 (shifteando 8 a izq y 24 derecha).
Terminamos teniendo los uzules de los pixeles en xmm3, los verdes en xmm4 y los rojos en xmm10, asi que multiplicamos xmm0, xmm1 y xmmm2 (las sumas x el alpha) x ellos, y terminamos teniendo colorji*alpha*sumargbi (donde i va de 22 a 25, y color puede ser r, g o b), en cada uno. luego los convertimos en floats de simple precision para dividirlos por max_f (en xmm13), que tiene cuatro veces el maximo de la formula, usando divps.
Ahora tenemos colorji*alpha*sumargbi/max, y nos falta sumarle a cada uno el color que le correspondia, lo hacemos sumandoles xmm3, xmm4 y xmm10 respectivamente. Conseguimos que en los registros destino esten la formula del filtro de cada color para los cuatro pixeles procesados.
Ahora solo nos queda reponer los a de cada pixel (y reordenar los colores, que cuando los vaciamos rotamos) Desempaquetamos los colores a words, packusdw contra sigo mismos. (nos quedan, los registros repetidos dos veces). Luego, utilizamos pshufhw, que es una version de pshufd para words, y en la que voy, a explayarme.
Los pshfhw (y los pshflw) son hechos, siempre en este codigo con un registro y si mismo, para reordenar sus datos.

Ahora, el estado de xmm0, xmm1 es :
xmm0: $|b25'|b24'|b23'|b22'|b25'|b24'|b23'|b22'|$
xmm1: $|g25'|g24'|g23'|g22'|g25'|g24'|g23'|g22'|$
colocamos g25' y los demas con apostrofe para denotar que es distinto al color con el que empezamos.

queremos tener
xmm0 : $|b25'|b25'|b24'|b24'|b23'|b23'|b22'|b22'|$
xmm1 : $|g25'|g25'|g24'|g24'|g23'|g23'|g22'|g22'|$
usamos pshufhw y el inmediato 11111010b (y xmm0, luego haremos lo mismo con xmm1). Esto funciona asi pshfhw nos mueve las palabras de la parte alta de un registro a la parte alta de otro, segun las posiciones indicadas por el inmediato. En este caso, ahora viendo las posiciones de un registro como de 0 a 7 words, siendo 0 la mas baja, manda la posicion 6 de xmm0 a la posicion 4 y 5, y la posicion 7 a la posicion 6 y 7.
xmm0:$|b25'|b25'|b24'|b24'|b25'|b24'|b23'|b22'|$
luego usamos pshflw, que hace lo mismo en la parte baja, con inmediato 01010000b (y xmm0, y luego se hara con xmm1). Este manda la posicion 0 a la 0 y uno, y la uno a la 2 y 3, por lo que queda
xmm0 : $|b25'|b25'|b24'|b24'|b23'|b23'|b22'|b22'|$
como querieamos. xmm1 es analogo. tambien lo es xmm12, (donde esta el rojo)

Ahora que tenemos las cosas repetidas, shifteamos cada word, de xmm0 16 a la derecha, y a los de xmm1 a la izquierda, para que donde uno este en 0 el otro este el color buscado, y los summamos de a word, obteniendo en xmm0 los verdes y azules en orden de los pixeles.
Vuelvo a desempaquetar xmm0, ahora desempaquetado a bytes, (obtengo dos veces lo mismo repetido, no hay ninguna disturbancia porque, al aplicar la formula, lo que queda en cada color ES de tamano de byte(saturado)), luego, vuelvo a usar los mismos shuffle de antes (de a word), y me quedan como antes salvo que ahora, en vez de haber un color por word, esta el verde en la parte alta de cada word, y el azul en la baja.
Ahora, shifteamos a la derecha 16 bits cada dword para dejar la parte alta vacia, mientras conservamos los colores en la parte baja de estas. Shifteamos a la derecha 32 bits xmm2 para obtener el rojo en la posicion que lo queremos. Volvemos a desempaquetarlo a bytes, y lo shuffeamos como antes, y lo volvemos a shiftear 32 bits luego de eso, pero a la izquierda, para que quede en la pocision correcta para sumar con xmm0.
xmm0 = $|0|r25'|g25'|b25'|0|r24'|g24'|b24'|0|r23'|g23'|b23'|0|r22'|g22'|b22'|$
Sumamos los dos de a byte. Y ahora solo nos queda poner los a, lo cual hacemos agarrando xmm15,(pixeles a procesar sin tocar) y los shifteamos a la izquierda cada doubleword 24 bits, (para vaciar todo salvo las a de cada pixel) y luego 24 bits a la derecha para que queden en la parte mas significativa de cada word.luego suma de a byte con xmm0 como destino y deja en este los cuatro pixeles con el filtro aplicado.
Movemos el puntero de pixeles a rsi (puntero a imagen de salida), le sumamos 2 filas a rbx y movemos xmm0 al contenido de rbx + columna actual (r12 ,*4 x los pixeles) + 4 x el desfasaje.
Incrementamos el puntero a columna actual (r12d), y preguntamos si acabo la fila, si no lo hizo iteramos de nuevo, si lo hizo, lo reiniciamos, (xor) incrementamos el contador de filas, y los punteros de las imagenes fuente y destino

para que avancen una fila (sumandoles tamano de fila), y si r11 es menor a rcx (cant de filas) entonces iteramos de nuevo, si no, terminamos la imagen, por lo cual desarmamos la pila y salimos.

.

4. Conclusiones y trabajo futuro