Índice

1. Introducción

2. Rotar

2.1. Código C

2.2. Código Asm

2.3. Experimentación

2.3.1. Idea

2.3.2. Hipótesis

2.3.3. Resultados

3. Smalltiles

3.1. Código C

3.2. Código Asm

3.3. Experimentación

3.3.1. Idea

3.3.2. Hipótesis

3.3.3. Resultados

4. Pixelar

4.1. Código C

4.2. Código Asm

4.3. Experimentación

4.3.1. Idea

4.3.2. Hipótesis

4.3.3. Resultados

5. Combinar

Este filtro consiste en realizar una combinación de dos imágenes que dependa de un número real α entre 0 y 255.

Cabe destacar que este filtro permite reutilizar cálculos debido a dos factores. Uno de ellos es la forma que tiene cada píxel generado en la imagen resultante, que consiste en que si se tienen una imagen A y una imagen B de tamaño $m \times n$ entonces el píxel de la imagen generada ${I_{AB}}^{i,j}$ se calcula de la siguiente forma:

$${I_{AB}}^{i,j} = \frac{\alpha * ({I_A}^{i,j} - {I_B}^{i,j})}{255{,}0} + {I_B}^{i,j}$$

El otro factor es que nuestro filtro está optimizado para casos en los que la imagen B es el reflejo vertical de la imagen A. Es decir que se da que ${I_A}^{i,j} = {I_B}^{i,n-j+1}$ como se puede apreciar en la siguiente figura.

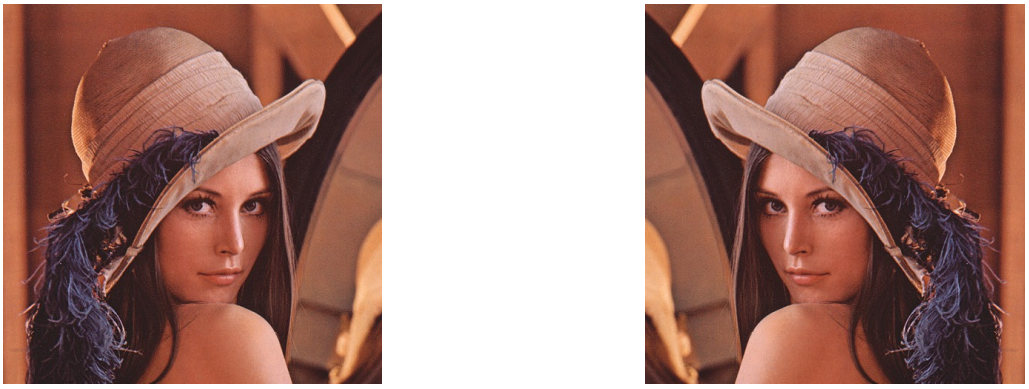

Figura 1: Se muestra la imagen A del lado izquierdo con la imagen B, el reflejo vertical de A, en la parte derecha.

Entonces se tiene que

$$\begin{aligned} {I_{AB}}^{i,j} &= \frac{\alpha * ({I_A}^{i,j} - {I_B}^{i,j})}{255{,}0} + {I_B}^{i,j} \text{ por la fórmula del filtro} \\ &= \frac{\alpha * ({I_A}^{i,j} - {I_A}^{i,n-j+1})}{255{,}0} + {I_A}^{i,n-j+1} \text{ dado que} {I_A}^{i,j} = {I_B}^{i,n-j+1} \end{aligned} \tag{1}$$

y análogamente

$$\begin{aligned} {I_{AB}}^{i,n-j+1} &= \frac{\alpha * ({I_A}^{i,n-j+1} - {I_B}^{i,n-j+1})}{255{,}0} + {I_B}^{i,n-j+1} \\ &= \frac{\alpha * ({I_A}^{i,n-j+1} - {I_A}^{i,j})}{255{,}0} + {I_A}^{i,j} \end{aligned} \tag{2}$$

Así, se obtiene

$$\begin{aligned} {I_{AB}}^{i,n-j+1} &= -1 * \frac{\alpha * [-1 * ({I_A}^{i,n-j+1} - {I_A}^{i,j})]}{255{,}0} - {I_A}^{i,n-j+1} + {I_A}^{i,n-j+1} + {I_A}^{i,j} \\ &= -1 * \left[\frac{\alpha * ({I_A}^{i,j} - {I_A}^{i,n-j+1})}{255{,}0} + {I_A}^{i,n-j+1}\right] + {I_A}^{i,n-j+1} + {I_A}^{i,j} \\ &= -1 * ({I_{AB}}^{i,j} - {I_A}^{i,n-j+1}) + {I_A}^{i,j} \text{usando la igualdad de (2)} \end{aligned} \tag{3}$$

Estos cálculos muestran que luego de hacer el procesamiento para generar un píxel de la parte izquierda de la imagen resultante se puede obtener el píxel que corresponde a la mitad derecha con pocos cálculos más. Más aún, si se denomina

$$P = \frac{\alpha * ({I_A}^{i,j} - {I_A}^{i,n-j+1})}{255{,}0}$$

se consigue

$${I_{AB}}^{i,j} = P + {I_A}^{i,n-j+1}$$

y

$$I_{AB}{}^{i,n-j+1} = -P + I_A{}^{i,j}$$

dando lugar a menos cálculos necesarios.

5.1. Código C

5.2. Código ASM

Dado que cada píxel tiene 4 bytes y los registros XMM tienen 16 bytes, se levantan 4 píxeles contiguos desde la posición i, j y otros 4 píxeles desde la $i, n - j + 1$ en otro registro.

Se mostrará cómo se genera el píxel $I_{i,j}$ de la imagen resultante.

Se levantan 4 píxeles de la mitad izquierda en el registro XMM1:

XMM1:

A_{j+3}	R_{j+3}	G_{j+3}	B_{j+3}	A_{j+2}	R_{j+2}	G_{j+2}	B_{j+2}	A_{j+1}	R_{j+1}	G_{j+1}	B_{j+1}	A_j	R_j	G_j	B_j
16															0

y 4 en espejo de la mitad derecha en XMM3:

XMM3:

A_{n-j+4}	R_{n-j+4}	G_{n-j+4}	B_{n-j+4}	A_{n-j+3}	R_{n-j+3}	G_{n-j+3}	B_{n-j+3}	A_{n-j+2}	R_{n-j+2}	G_{n-j+2}	B_{n-j+2}	A_{n-j+1}	R_{n-j+1}	G_{n-j+1}	B_{n-j+1}
16															0

Teniendo

XMM9:

0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16															0

se ejecuta la instrucción **punpcklbw xmm1, xmm9** que da como resultado

XMM1:

0	A_{j+1}	0	R_{j+1}	0	G_{j+1}	0	B_{j+1}	0	A_j	0	R_j	0	G_j	0	B_j
16															0

y luego **punpcklwd xmm1, xmm9** para seguir desempaquetando. De esta forma quedan 4 bytes para cada componente del píxel y se podrán convertir a floats para mayor precisión en los cálculos que implican el α.

XMM1:

0	0	0	A_j	0	0	0	R_j	0	0	0	G_j	0	0	0	B_j
16															0

Se copió el contenido de XMM1 a XMM10 y con el registro que contenía al píxel $n - j + 4$, obtenido de forma análoga a como se obtuvo el píxel j

XMM8:

0	0	0	A_{n-j+4}	0	0	0	R_{n-j+4}	0	0	0	G_{n-j+4}	0	0	0	B_{n-j+4}
16															0

se realizaron las restas entre componentes (**psubd xmm10, xmm8**) antes de convertir a float el registro XMM10 (**cvtdq2ps xmm10, xmm10**) dado que realizar la resta con float es una operación mucho más costosa que con enteros y además no había chances de perder precisión con la resta.

XMM10:

$A_j - A_{n-j+4}$	$R_j - R_{n-j+4}$	$G_j - G_{n-j+4}$	$B_j - B_{n-j+4}$

16 0

Se multiplicó por α al registro XMM10 (**mulps xmm10, xmm0**) y luego se dividió por 255,0 (**divps xmm10, xmm14**) obteniéndose

XMM10:

$\frac{\alpha * (A_j - A_{n-j+4})}{255,0}$	$\frac{\alpha * (R_j - R_{n-j+4})}{255,0}$	$\frac{\alpha * (G_j - G_{n-j+4})}{255,0}$	$\frac{\alpha * (B_j - B_{n-j+4})}{255,0}$

16 0

Para esto previamente, fuera del ciclo, se había movido el α que había llegado en los últimos 4 bytes de XMM0 a las otras 3 double words del registro con la instrucción **pshufd xmm0, xmm0, 00000000b** y se había puesto en XMM14 un 255.0 en cada double word declarando **mascara255: dd 255.0, 255.0, 255.0, 255.0** en **section .rodata** y luego haciendo **movdqu xmm14, [mascara255]**. De esta forma, al realizar estas operaciones fuera del ciclo, se minimizan los accesos a memoria y se evita repetir operaciones innecesarias.

Convirtiendo XMM8 a float con **cvtdq2ps xmm8, xmm8** y sumándolo a XMM10 con **addps xmm10, xmm8** se obtuvo

XMM10:

$\frac{\alpha * (A_j - A_{n-j+4})}{255,0} + A_{n-j+4}$	$\frac{\alpha * (R_j - R_{n-j+4})}{255,0} + R_{n-j+4}$	$\frac{\alpha * (G_j - G_{n-j+4})}{255,0} + G_{n-j+4}$	$\frac{\alpha * (B_j - B_{n-j+4})}{255,0} + B_{n-j+4}$

16 0

Antes de finalizar el procesamiento, se realizó **movups xmm15, xmm10** y **subps xmm15, xmm8** para conservar en XMM15 lo que en el desarrollo de cuentas apareció como P al explicar cómo se podían reutilizar cálculos.

Por último en el procesamiento de la parte izquierda se convirtió a entero XMM10 con **cvtps2dq xmm10, xmm10** y se empaquetó con los resultados de los demás píxeles usando instrucciones como **packusdw xmm11, xmm10** y **packusdw xmm13, xmm12** para pasar de double word a word,

XMM11:

F_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}

16 0

luego **packuswb xmm13, xmm11** para pasar de word a byte,

XMM11:

F_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}	F_{Aj+2}	F_{Rj+2}	F_{Gj+2}	F_{Bj+2}	F_{Aj+3}	F_{Rj+3}	F_{Gj+3}	F_{Bj+3}

16 0

y finalmente **pshufd xmm11, xmm13, 0x1b** quedando todo listo para mover el contenido del registro a donde correspondiera.

XMM11:

F_{Aj+3}	F_{Rj+3}	F_{Gj+3}	F_{Bj+3}	F_{Aj+2}	F_{Rj+2}	F_{Gj+2}	F_{Bj+2}	F_{Aj+1}	F_{Rj+1}	F_{Gj+1}	F_{Bj+1}	F_{Aj}	F_{Rj}	F_{Gj}	F_{Bj}

16 0

Para finalizar la parte derecha se multiplicaron todos los valores de las double words de XMM15 por -1 (**mulps xmm15, [menos1]** estando menos1 definido en **rodata** como **menos1: dd -1.0, -1.0, -1.0, -1.0**) y se prosiguió con las conversiones y sumas como anteriormente.

5.3. Experimentación

5.3.1. Idea

5.3.2. Hipótesis

5.3.3. Resultados

6. Colorizar

6.1. Código C

El codigo C se trata de una conjuncion de Fors, el exterior que recorre desde la segunda fila hasta la ante ultima, y el interior que recorre desde la segunda columna hasta la ultima, dejando asi afuera a todos los bordes, tal como el enunciado decia. Luego en cada iteracion del ciclo interior, que es donde se hace las opearciones que modifican la imagen, lo que hacemos es crear un arreglo de unsinged chars, res", que es donde guardamos los maximos de cada canal en comparacion a todos los pixeles lindantes del pixel en el cual estemos parado.

- Res [0] ← MaximoLindantesAzul
- Res [1] ← MaximoLindantesVerde
- Res [2] ← MaximoLindantesRojo

Luego con estos tres valores calculamos el alpha correspondiente de cada canal por el cual voy a multiplicar a cada uno. Y por ultimo reescribimos el pixel final, en la imagen source con cada canal multiplicado por dicho alpha.

6.2. Código Asm

El Código en ASM se trata tambien de una conjuncion de ciclos. El recorre las filas desde la segunda hasta la anteUltima, y el interior recorre las columnas desde la segunda hasta la anteultima, pero saltando de a dos pixeles, que es la cantidad que procesamos simultaneamente con instruccciones SSE. El ciclo interior consta de tres partes, la primera es tan pronto se levanta de memoria los pixeles a procesar y todos sus lindantes, calculamos en dos registros los maximos de cada canal, de ambos pixeles a procesar con respecto a todos sus lindantes, y los guardamos ambos en un registro xmm. Luego la segunda parte es calcular el maximo de los maximos de ambos al mismo tiempo. Luego atravez de una par de operaciones, muy especificas para explicar, logro tener un registro xmm de dw, con cada dw representando un canal de cada pixel, en el orden establecido (argb), donde tengo un 1-alpha en la posicion del canal que no teine el maximo de los maximos, y un 1+alpha en la posicion que tiene al maximo de los maximos, y luego concluyo multiplicando a cada pixel por su registro con los alphas indicados, lo reduzco a tamaño de 32 bits por pixel, los uno y los escribo en el destino.

6.3. Experimentación

6.3.1. Idea

En la experimentación de este filtro al igual que en el resto vamos a comparar el rendimiento respecto a los ciclos de clock, que tiene la funcion colorizar en C desde -o0 a -o3 contra asm. Sin embargo luego vamos a contrastar la funcion de asm, contra sigo mismo pero primero agragando jumps de forma que no influya el flujo del programa solo para molestar al jump predictor. Luego vamos a correrlo tal cual esta, y de apoco vamos a ir desenrollando el Código. Como dijimos tiene un ciclo externo y uno interno, por lo que vamos a desenrrollar primero el interno 2 veces, luego 4,16,32 y ver que pasa, y finalmente vamos a saltar a desenrrollarlo completamente, cosa de no dejar casi ninguno controlador de flujo.

6.3.2. Hipótesis

Nuestra hipótesis es que el rendimiento va a ir mejorando a medida que vayamos cambiando los programas respectivamente a como los fui mencionando, osea el mas lento va a ser el Código de asm, molestando al jmp predictor, y el mas rapido el desenrollando el Código 32 veces. El ultimo test, por mas que desenrollemos todo el programa creemos justamente que va a ser el mas lento, por el tamaño del Código, creemos que al hacer un código de tamaño mayor al de la cache, en un momento el pc va a estar haciendo muchisimos mas miss en la cache, que los que ahorra sacando los controladores de flujo.

6.3.3. Resultados

Graficos lindos de lucia :D
ciclos promedio colorizar original 10000 iteraciones: