



DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Practico II

A PC regalado, no se le mira procesador

Organizacion del Computador II
Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Rodrigo Kapobel	695/12	rok_35@live.com.ar
Nicolas Hernandez	122/13	nicojh22@hotmail.com
Luciano Saenz	904/13	saenzluciano@gmail.com



Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
<http://www.fcen.uba.ar>

Resumen

El presente informe tiene como objetivo implementar e investigar la eficiencia de diferentes tipos de filtros de imágenes mediante el uso del lenguaje de instrucciones assembler SIMD de intel. La problemática principal se basa en tratar de mostrar porque SIMD supone una mejora frente a implementaciones en otros lenguajes de mas alto nivel como puede ser C.

Índice

1. Introducción

Las tecnologías actuales de video en tiempo real y filtros de imagen son posibles desde hace unos años gracias a la eficiencia del código usado en sus implementaciones. Sería imposible una implementación de un reproductor streaming puramente en un lenguaje de programación de uso común como C, debido a la enorme cantidad de computos requerida.

Para solucionar este inconveniente se requería el procesamiento de información en paralelo y esto fue posible gracias a las instrucciones SIMD (single instruction multiple data) diseñadas y desarrolladas por intel, que son un subconjunto de las instrucciones bien conocidas assembler que utiliza esta arquitectura, las cuales permiten operar con varios datos a la vez, facilitando el cálculo de pixeles a la hora de procesar una imagen de video y realizar operaciones de punto flotante, muy importantes en la creación de gráficos por computadora para el cálculo de físicas, iluminación, etc.

También tiene aplicación en el procesamiento de filtros de imagen, obteniendo con los mismos, resultados notables frente a implementaciones con otros lenguajes usuales como el mencionado C.

1.1. Motivaciones

En el siguiente informe haremos uso del set de instrucciones de SIMD para implementar tres filtros, a saber: Sepia, Cropflip y Low Dynamic Range, los cuales serán explicados y desarrollados en el mismo. Propondremos además una variante de la implementación de cada filtro en lenguaje C.

Luego abordaremos sus características particulares y propondremos casos de estudio en base a los mismos para intentar responder a preguntas como: Qué implementación es mejor?. Primero se analizarán las metodologías para llevarlos a cabo y luego se darán a conocer los resultados obtenidos. Por último, en base a lo mostrado, obtendremos las conclusiones pertinentes.

2. Desarrollo

2.1. Imágenes

Las siguientes implementaciones operan con imágenes del tipo bmp y canales *alfa*, *red*, *green* y *blue* (*argb*) de 8 bits cada uno (de ahí la variante bmp 32 bits).

En memoria los datos de cada pixel estarán ordenados a la inversa, es decir, *bgra*, por lo cual al trabajarlos en registros de procesador, los mismos serán invertidos debido a que es el formato standard utilizado por la arquitectura intel para almacenar datos en memoria. El puntero de fuente obtenido en todas las implementaciones representa a la imagen a la cual se le quiere aplicar el filtro. La misma tiene una particularidad para la lectura y es que la primera fila leída es la última de la imagen.

Llamaremos *O* a la imagen de salida generada por cada filtro. Por ejemplo, el filtro identidad estaría caracterizado por la fórmula

$$\forall k \in (r, g, b, a) O_{i,j}^k = I_{i,j}^k$$

2.2. Implementaciones

Introduciremos cada filtro mencionandolos en el siguiente orden:

1. Cropflip
2. Sepia
3. Low dynamic range

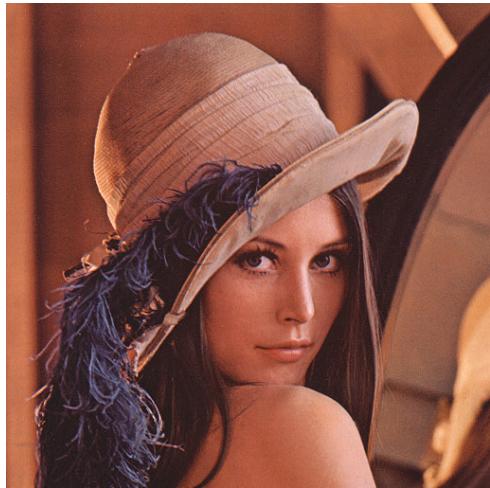
Para cada uno expondremos su idea principal, es decir, que efecto tiene sobre la imagen a la que se aplica, mostrando un caso de ejemplo. Luego expondremos las implementaciones, focalizando y detallando principalmente la implementación en lenguaje assembler SIMD.

2.3. Cropflip

Este filtro es una unión de dos filtros: crop y vertical-flip. Al aplicarlo sobre una imagen, recorta una parte de la misma y la volteá verticalmente. Para ello recibe cuatro argumentos delimitando un rectángulo de la imagen.

- *tamx*: Posee la cantidad de columnas, en pixeles, a recortar. Este número es múltiplo de 4.
- *tamy*: Contiene la cantidad de filas, en pixeles, a recortar.
- *offseex*: Columna, en pixeles, a partir de la cual se debe comenzar a recortar. Este número también es múltiplo de 4.
- *offsety*: Fila, en pixeles, a partir de la cual se debe comenzar a recortar.

El cuadro obtenido se devuelve espejandolo verticalmente. Para ello rearma las filas en orden inverso.



(a) Original



(b) Cropflip

Figura 1: Corte: 320x200 $offsetx : 100$ y $offsety : 0$

Si bien es el código más sencillo de implementar, incluso de manera eficiente, requiere algún tipo de explicación.

2.3.1. Assembler SIMD

En la implementación de SIMD se aprovecha el uso de paralelismo y la capacidad de almacenar hasta cuatro pixeles en un registro *xmm*.

Tomando entonces de a 4 pixeles, desde el inicio del rectangulo determinado por los parametros, se rearma la imagen, logrando finalmente, al recorrer todas las filas indicadas, el espejado vertical esperado.

```
.ciclo:  
    movdqu xmm1, [rdi]      ; p0|p1|p2|p3  
    movdqu [rsi], xmm1  
  
    lea rsi, [rsi + 16]  
    lea rdi, [rdi + 16]  
    loop .ciclo
```

Como puede observarse, mediante un ciclo podemos con la ayuda de un registro *xmm* transportar desde la fuente hasta el destino, 4 pixeles simultaneamente. Esto supondría en principio una ventaja sobre la siguiente implementación.

2.3.2. C

Debido a que no disponemos de paralelismo, no nos es posible enviar desde la fuente al destino 4 pixeles simultaneamente. Por lo tanto cada pixel se opera unitariamente. Igualmente se puede llevar a cabo con la ayuda de dos simples ciclos anidados.

```

for (int i = 0; i < tamy; i++)
{
    for (int j = 0; j < tamx; j++)
    {

        bgra_t *p_d = (bgra_t*) &dst_matrix[(tamy-1)-i][j*4];
        bgra_t *p_s = (bgra_t*) &src_matrix[i+offsety][(j+offsetx)*4];

        p_d->b = p_s->b;
        p_d->g = p_s->g;
        p_d->r = p_s->r;
        p_d->a = p_s->a;

    }
}

```

2.4. Sepia

Esta operación consiste en cambiar la información de color de cada pixel de la siguiente manera:

$$O_{i,j}^R = 0,5.suma_{i,j}$$

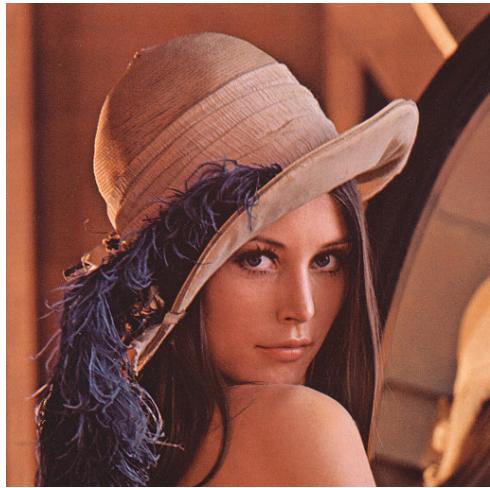
$$O_{i,j}^G = 0,3.suma_{i,j}$$

$$O_{i,j}^B = 0,2.suma_{i,j}$$

donde

$$suma_{i,j} = I_{i,j}^R + I_{i,j}^G + I_{i,j}^B$$

El efecto logrado es que realza más el canal verde.



(a) Original



(b) Sepia

Si bien es una operación sencilla, los tiempos de cómputo se ven comprometidos para imágenes muy grandes debido a la cantidad de operaciones en punto flotante: En una imagen de 512x512 pixeles tenemos $512 \times 512 \times 3 = 786432$ operaciones de punto flotante. Lo cual puede suponer un desafío si no se dispone de tecnologías como SIMD.

2.4.1. Assembler SIMD

Al igual que en el primer filtro, se aprovecha el paralelismo de SIMD y en particular la eficiencia del cálculo en punto flotante.

Para operar la imagen se procesa de a 4 pixeles. Luego cada pixel se lleva a un registro xmm extendiendo sus canales a 32 bits. Como no se asume que el alfa sea cero se utiliza una máscara para borrarlo previamente. Luego se realiza la suma de los canales para cada pixel y por último se convierte cada una a punto flotante 32 bits y se realizan los 3 productos para cada suma que corresponden a los nuevos canales r , g y b mediante el uso de operaciones de punto flotante.

Al final se restaura el canal alfa junto con el nuevo pixel calculado y se devuelve a la imagen destino.

```

movdqu xmm1, [rdi]; XMM1 = | p3 | p2 | p1 | p0 |
movdqu xmm2, xmm1 ; XMM1 = XMM2
movdqu xmm5, xmm1; respaldo XMM5 = XMM1

limpiar alfa
pand xmm1, xmm7; XMM1 = | 0 | r3 | g3 | b3 |...
pand xmm2, xmm7; idem

punpcklbw xmm1, xmm6; XMM1 = | p1 | p0 | con alfa limpio
punpckhbw xmm2, xmm6; XMM2 = | p3 | p2 | con alfa limpio

movdqu xmm3, xmm1
movdqu xmm4, xmm2
punpcklbw xmm1, xmm6; XMM1 = | 0 | r0 | g0 | b0 |
punpckhbw xmm3, xmm6; XMM3 = | 0 | r1 | g1 | b1 |
punpcklbw xmm2, xmm6; XMM2 = | 0 | r2 | g2 | b2 |
punpckhbw xmm4, xmm6; XMM4 = | 0 | r3 | g3 | b3 |

phaddd xmm1, xmm1; XMM1 = | r0 | g0 + b0 | r0 | g0 + b0 |
phaddd xmm1, xmm1; XMM1 = | suma0 | suma0 | suma0 | suma0 |
phaddd xmm2, xmm2; idem con pixeles 1, 2 y 3
phaddd xmm2, xmm2
phaddd xmm3, xmm3
phaddd xmm3, xmm3
phaddd xmm4, xmm4
phaddd xmm4, xmm4

; un unpack mas, multiplico de a un pixel

cvtdq2ps xmm1, xmm1; suma0 asFloat
cvtdq2ps xmm2, xmm2; suma2 asFloat
cvtdq2ps xmm3, xmm3; suma1 asFloat
cvtdq2ps xmm4, xmm4; suma3 asFloat

movups xmm0, [factores]

mulps xmm1, xmm0; XMM1 = |*|suma0*0.2|suma0*0.3|suma0*0.5|
mulps xmm2, xmm0; XMM2 = |*|suma2*0.2|suma2*0.3|suma2*0.5|
mulps xmm3, xmm0; XMM3 = |*|suma1*0.2|suma1*0.3|suma1*0.5|
mulps xmm4, xmm0; XMM4 = |*|suma3*0.2|suma3*0.3|suma3*0.5|

cvttps2dq xmm1, xmm1; |0|suma0*0.2 asInt|suma0*0.3 asInt|suma0*0.5 asInt|
cvttps2dq xmm2, xmm2
cvttps2dq xmm3, xmm3
cvttps2dq xmm4, xmm4

packusdw xmm1, xmm3;
packusdw xmm2, xmm4;
packuswb xmm1, xmm2; XMM1 = | p3' | p2' | p1' | p0' |

; restaurar canal alfa
pand xmm5, xmm8
paddb xmm1, xmm5

```

2.4.2. C

Como sucede en *cropflip*, no disponemos de paralelismo, con lo que solo disponemos de la ayuda de ciclos anidados y accesos y operaciones unitarias.

```

for (int i = 0; i < filas; i++)
{
    for (int j = 0; j < cols; j++)
    {
        bgra_t *p_d = (bgra_t*) &dst_matrix[i][j * 4];
        bgra_t *p_s = (bgra_t*) &src_matrix[i][j * 4];
        aux = (int) p_s->r + (int) p_s->g + (int) p_s->b;
        p_d->r = (aux * 0.5 > 255)? 255 : aux * 0.5;
        p_d->g = (aux * 0.3 > 255)? 255 : aux * 0.3;
        p_d->b = (aux * 0.2 > 255)? 255 : aux * 0.2;
        p_d->a = p_s->a;
    }
}

```

2.5. LDR: Low Dynamic Range

El filtro *ldr* es el que más operaciones lleva a cabo de los tres. Toma una imagen y aplica un efecto que modifica la imagen según su iluminación. El filtro toma el valor de un pixel y le añade un porcentaje α de el de sus vecinos.

De esta manera, dado un porcentaje positivo, los pixeles rodeados por pixeles claros se vuelven aún más claros, mientras que los rodeados por pixeles oscuros se mantienen igual. La intensidad del efecto dependerá del porcentaje sumado. Para cada componente independiente del pixel (*r*, *g* y *b*) la fórmula matemática será:

$$O_{i,j}^K = \min(\max(ldr_{i,j}^K, 0), 255)$$

donde

$$\begin{aligned}
ldr_{i,j}^K &= I_{i,j}^K + \alpha \frac{\sum argb_{i,j}}{\max} \cdot I_{i,j}^K \\
\sum argb_{i,j} &= \sum a_{i,j}^r + \sum a_{i,j}^g + \sum a_{i,j}^b \\
\max &= 5 * 5 * 255 * 3 * 255
\end{aligned}$$

y finalmente $\sum a_{i,j}^K$ corresponde a:

$$\begin{aligned}
&I_{i+2,j-2}^K + I_{i+2,j-1}^K + I_{i+2,j}^K + I_{i+2,j+1}^K + I_{i+2,j+2}^K + \\
&I_{i+1,j-2}^K + I_{i+1,j-1}^K + I_{i+1,j}^K + I_{i+1,j+1}^K + I_{i+1,j+2}^K + \\
&I_{i,j-2}^K + I_{i,j-1}^K + I_{i,j}^K + I_{i,j+1}^K + I_{i,j+2}^K + \\
&I_{i-1,j-2}^K + I_{i,j-1}^K + I_{i-1,j}^K + I_{i-1,j+1}^K + I_{i-1,j+2}^K + \\
&I_{i-2,j-2}^K + I_{i-2,j-1}^K + I_{i-2,j}^K + I_{i-2,j+1}^K + I_{i-2,j+2}^K +
\end{aligned}$$



(a) Original



(b) LDR

Figura 2: $\alpha: 255$

Para reducir errores de redondeo, la division debe ser la última operación en realizarse. El resultado final será saturado de ahí \max y \min en la primer fórmula.

Además, dado que en los bordes no es posible calcular ldr por la ausencia de vecinos, se devolverá el valor original. Es decir:

$$O_{i,j}^K = I_{i,j}^K \text{ si } i < 2 \vee j < 2 \vee i + 2 \leq tamy \vee j + 2 \leq tmax$$

(con i indexado a partir de 0)

2.5.1. Assembler SIMD

En la implementación de ldr tenemos varios inconvenientes a solventar, pero el principal es el cálculo de la suma. Para lograr aprovechar el paralelismo que ofrece SIMD se realizan varias operaciones. Para cada fila del cuadrado que supone la sumatoria, se obtienen 4 pixeles en un registro xmm y luego se separan en dos registros diferentes, de manera tal que tengamos el los pixeles $p0$ y $p1$ en uno y $p2$ y $p3$ en otro con cada canal transformado a 16 bits. Para realizarlo se realiza un shuffle para cada caso.

Una vez obtenidos los dos registros anteriores, se suman horizontalmente los registros respectivos una vez y se transforma el resultado a 32 bits. Luego se aplican sumas verticales valiéndose de un registro de respaldo y varios shifts para eliminar las sumas que no signifiquen en la fórmula.

Debido a que la suma que buscamos es de cinco pixeles, se obtiene el 5to pixel en otro registro y se lo opera de manera similar a los anteriores para luego sumarlo a los mismos acumulando la suma de cada fila en el registro particular $xmm0$

```

.cincoHorizontal:

; 16 12 8 4 0
; Li4|Li3|Li2|Li1|Li0
;VERSION STANDARD: Con 2 accesos - acceso extra paa el pixel 5
movdqu xmm13, [rdi + r12*pixelSize] ; Li3|Li2|Li1|Li0

movdqu xmm9, xmm13
pshufb xmm9, xmm6 ; 0|0|0|r1|0|g1|0|b1|0|0|0|r0|0|g0|0|b0
phaddw xmm9, xmm11 ; 0|0|0|0|0+r1|g1+b1|0+r0|g0+b0
pshufb xmm13, xmm7 ; 0|0|0|r3|0|g3|0|b3|0|0|r2|0|g2|0|b2
phaddw xmm13, xmm11 ; 0|0|0|0|r3|g3+b3|r2|g2+b2
punpcklwd xmm9, xmm11 ; r1|g1+b1|r0|g0+b0
punpcklwd xmm13, xmm11 ; r3|g3+b3|r2|g2+b2
paddd xmm13, xmm9 ; r3+r1|g3+b3+g1+b1|r2+r0|g2+b2+g0+b0
movdqu xmm9, xmm13
psrlqd xmm9, 8 ; 0|0|r3+r1|g3+b3+g1+b1
paddd xmm13, xmm9 ; r3+r1|g3+b3+g1+b1|r2+r0+r3+r1|g2+b2+g0+b0+g3+b3+g1+b1
pslldq xmm13, 8 ; r2+r0+r3+r1|g2+b2+g0+b0+g3+b3+g1+b1|0|0
psrlqd xmm13, 8 ; 0|0|r2+r0+r3+r1|g2+b2+g0+b0+g3+b3+g1+b1
movd xmm9, [rdi + r12*pixelSize + 16] ; 0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|a4|r4|g4|b4

pslldq xmm9, 12 ; a4|r4|g4|b4|0|0|0|0|0|0|0|0|0|0|0|0|0
pand xmm9, xmm8 ; 0|r4|g4|b4|0|0|0|0|0|0|0|0|0|0|0|0
pshufb xmm9, xmm7 ; 0|0|0|r4|0|g4|0|b4|0|0|0|0|0|0|0|0
phaddw xmm9, xmm11 ; 0|0|0|0|r4|g4+b4|0|0 -- maximo por w = 510 in []
punpcklwd xmm9, xmm11 ; 0|r4|0|g4+b4|0|0|0|0
psrlqd xmm9, 8 ; 0|0|r4|g4+b4
paddd xmm13, xmm9 ; 0|0|r2+r0+r3+r1+r4|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4
movdqu xmm9, xmm13
psrlqd xmm9, 4 ; 0|0|0|r2+r0+r3+r1+r4
paddd xmm13, xmm9 ; 0|0|r2+r0+r3+r1+r4|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4
pslldq xmm13, 12 ; g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4|0|0|0
psrlqd xmm13, 12 ; 0|0|0|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4
paddd xmm0, xmm13 ; suma de la i fila para el pixel ij.

add r12, r15
inc r10
cmp r10, 5
jl .cincoHorizontal

```

Una vez obtenida la suma, que al final tendrá un tamaño de 32 bits, se procede a realizar la fórmula de *ldr*. El primer paso es armar la un registro con el valor de *alfa* en sus primeras tres posiciones menos significativas que es donde se encontraran los valores de los canales. Así mismo, se crea un registro con el divisor *max* en las cuatro posiciones debido a que con el realizaremos la division.

Se crea un registro con la suma replicada en las tres posiciones menos significativas y se procede a calcular la fórmula, para lo cual tenemos la siguiente consideración:

Debido a que, $canal \times sumargbx \alpha$ cabe perfectamente dentro de un entero de 32 bits (el máximo es $75 \times 255 \times 255 \times 255$ o $75 \times 255 \times 255 \times 255 = +1,243,603,125 \in [-2,147,483,648 \text{ a } 2,147,483,647]$, no se convierte a punto flotante si no hasta la division, por lo tanto será la única operacion de punto flotante del algoritmo, aunque vale mencionar que si por cada pixel, exceptuando las primeras y últimas dos filas y columnas se aplica esta fórmula en total habrá $3x(\text{filas}-2)x(\text{columnas}-2)$ operaciones de punto flotante. Para una imagen de 512×512 supone 780300 divisiones de punto flotante, que se asemeja bastante a la cantidad de operaciones de punto flotante de *sepia*.

```

pxor xmm13, xmm13
movd xmm13, ebx
movdqu xmm12, xmm13 ; 0|0|0|alpha
pslldq xmm12, 4 ; 0|0|alpha|0
por xmm12, xmm13 ; 0|0|alpha|alpha
pslldq xmm12, 4 ; 0|alpha|alpha|0
por xmm12, xmm13 ; 0|alpha|alpha|alpha

pxor xmm14, xmm14
movd xmm14, r13d
movdqu xmm13, xmm14 ; 0|0|0|max
pslldq xmm13, 4 ; 0|0|max|0
por xmm13, xmm14 ; 0|0|max|max
pslldq xmm13, 4 ; 0|max|max|0
por xmm13, xmm14 ; 0|max|max|max
pslldq xmm13, 4 ; max|max|max|0
por xmm13, xmm14 ; max|max|max|max

movdqu xmm5, xmm0 0|0|0|0
pslldq xmm5, 4 0|0|sumargb|0
por xmm5, xmm0 0|0|sumargb|sumargb
pslldq xmm5, 4 0|sumargb|sumargb|0
por xmm5, xmm0 0|sumargb|sumargb|sumargb
pxor xmm14, xmm14
movd xmm14, [rdi + r8*pixelSize] ; 0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|alpha|r|g|b <- get pixel ij
movdqu xmm0, xmm14 ; 0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|alpha|r|g|b
pand xmm0, xmm15 ; 0|0|0|0|0|0|0|0|0|0|0|0|0|0|0|r|g|b
punpcklbw xmm0, xmm11 ; 0|0|0|0|0|r|g|b
punpcklwd xmm0, xmm11 ; 0|r|g|b
; 0|sumargb*r|sumargb*g|sumargb*b == 0|sumargb*r|sumargb*g|sumargb*b
pmulld xmm5, xmm0
; puede cambiar el signo segun alpha
; 0|alpha*sumargb*r|alpha*sumargb*g|alpha*sumargb*b
pmulld xmm5, xmm12
; 0|fp(alpha*sumargb*r)|fp(alpha*sumargb*g)|fp(alpha*sumargb*b)
cvtdq2ps xmm5, xmm5
cvtdq2ps xmm13, xmm13 ; fp(max)|fp(max)|fp(max)|fp(max)
divps xmm5, xmm13 ; 0|(alpha*sumargb*r)/max|(alpha*sumargb*g)/max|(alpha*sumargb*b)/max
cvtdq2ps xmm0, xmm0 ; 0|fp(r)|fp(g)|fp(b)
; 0|r+(alpha*sumargb*r)/max|g+(alpha*sumargb*g)/max|b+(alpha*sumargb*b)/max
addps xmm5, xmm0
cvttps2dq xmm5, xmm5 ; cast to dw signed
; 0|0|0|0|r+(alpha*sumargb*r)/g+max|(alpha*sumargb*g)/b+max|(alpha*sumargb*b)/max
packusdw xmm5, xmm11
packuswb xmm5, xmm11
; tengo los canales calculados saturados a byte.
; 0|0|0|0|r+(alpha*sumargb*r)/g+max|(alpha*sumargb*g)/b+max|(alpha*sumargb*b)/max
pand xmm14, xmm4 ; 0|0|0|0|r+(alpha*sumargb*r)/g+max|(alpha*sumargb*g)/b+max|(alpha*sumargb*b)/max
; 0|0|0|0|r+(alpha*sumargb*r)/max|g+(alpha*sumargb*g)/max|b+(alpha*sumargb*b)/max
por xmm14, xmm5

```

2.5.2. C

Como ya hemos explicado, C carece de paralelismo en las operaciones, por lo tanto tendremos que lidiar de la manera tradicional, obteniendo cada pixel y operando unitariamente. Al final el algoritmo es

similar pero con más accesos a memoria para lectura y escritura.

```
int i = 0;
int indexSquare = c - ((cols*2)+2);
int sumargb = 0;

while (i < 5) {
    sumargb += src[indexSquare*4];
    sumargb += src[indexSquare*4+1];
    sumargb += src[indexSquare*4+2];
    sumargb += src[indexSquare*4+4];
    sumargb += src[indexSquare*4+5];
    sumargb += src[indexSquare*4+6];
    sumargb += src[indexSquare*4+8];
    sumargb += src[indexSquare*4+9];
    sumargb += src[indexSquare*4+10];
    sumargb += src[indexSquare*4+12];
    sumargb += src[indexSquare*4+13];
    sumargb += src[indexSquare*4+14];
    sumargb += src[indexSquare*4+16];
    sumargb += src[indexSquare*4+17];
    sumargb += src[indexSquare*4+18];
    indexSquare += cols; //siguiente fila.
    i++;
}

float sumargbf = sumargb;
float alphaf = alpha;
float maxf = 4876875;
float b = (float)src[c*4];
float g = (float)src[c*4+1];
float r = (float)src[c*4+2];
unsigned char a = src[c*4+3];

b = b + (alphaf*sumargbf*b)/maxf;

b = MIN(MAX(b,0), 255);

g = g + (alphaf*sumargbf*g)/maxf;

g = MIN(MAX(g,0), 255);

r = r + (alphaf*sumargbf*r)/maxf;

r = MIN(MAX(r,0), 255);

dst[c*4] = (unsigned char)b;
dst[c*4+1] = (unsigned char)g;
dst[c*4+2] = (unsigned char)r;
dst[c*4+3] = a;
```

2.6. Análisis experimental

En general, al aplicar filtros sobre imágenes, la performance puede verse afectada por varios motivos, como por ejemplo el scheduler del sistema, que puede generar caídas en el tiempo debido a que debe realizar operaciones de mayor prioridad antes de poder retornar al algoritmo. Otras cuestiones pueden

estar relacionadas con el acceso a memoria.

Como bien se sabe, la mayoría de los procesadores poseen una memoria integrada que es la más rápida luego de los registros, conocida como memoria caché, y que se subdivide en dos niveles: L1 (a su vez subdividida en datos e instrucciones) y L2 para datos. La idea de la misma es tener "a mano" datos que sean pedidos al sistema, trayendo consigo además los contiguos en un bloque de memoria (principio de vecindad espacial), de manera tal que al consultar por el mismo nuevamente o algún contiguo, se obtenga mucho más rápido desde esta memoria. Pero cuando los mismos no se encuentran, el procesador tiene que buscarlos en la memoria ram, siendo este proceso la causa en la caída de performance que más suele afectar a los algoritmos.

Este y algunos motivos más serán de estudio en este informe. Para ello introduciremos cada una de las hipótesis que se plantearán en base a lo que cada implementación en particular genere y propondremos un test a realizar, explicando la metodología para llevarlo a cabo y los resultados obtenidos con las conclusiones que correspondan.

Además, escogeremos una implementación en particular para llevar a cabo algunos tests especiales que surjan en base a resultados más o menos generales para tener una visión extra de lo que sucede en esos casos.

2.6.1. Metodologías

Evaluaremos con los distintos tests el rendimiento de cada implementación. El mejor o peor rendimiento de las implementaciones se basa en la toma de tiempos de ejecución. Como los tiempos de ejecución son relativamente pequeños, en general, se utilizará uno de los contadores de performance que posee el procesador.

La imagen elegida para los tests será la siguiente::



Figura 3: La última de Star Wars

El procesador utilizado para realizar todos los tests sera un Intel(R) Core(TM) i5-2500K CPU @ 3.30GHz con caché L2 de 2MB.

Para que la caché no influya en cada corrida se implementó un algoritmo sencillo que lo que hará es ocupar la caché L2 antes de cada corrida. Sabiendo que la caché es de 2MB el algoritmo siguiente debería garantizar que la caché se ocupará con información que no altere los resultados de los tests.

```
char *basura = (char*)malloc(sizeof(char)*2048);
srand(5);
int i = 0;
while (i < 2048) {
    basura[i] = rand() % 27;
    i++;
}
long int suma = 0;
i = 0;
while (i < 2048) {
    suma += basura[i];
    i++;
}
//llamada test
free(basura);
```



Figura 4: *ldr α = 255*



Figura 5: *sepia*



Figura 6: *cropflip 1240 1160 600 1000*

2.7. Hipótesis

2.7.1. Aumentando resoluciones

Nos interesa saber como varia el rendimiento de los algoritmos para cada filtro en sus dos variantes (*asm*, *C*) cuando se varia el tamaño de ??.

El resultado esperado es que para todos los filtros en ambas versiones el tiempo sea creciente a medida que el tamaño crece y si el aumento es cuadratico en el tamaño, también lo sea en el tiempo. Además, que la versión en implementada en *asm* se mantenga por encima de *C* siempre.

2.7.2. Performance: implementaciones y filtros

La meta principal del informe es comprobar que *asm* es mejor que la implementación en *C*. Para esto correremos todos los filtros comparando sus versiones de *assembler*, *C* y además incluiremos *C* con flags de optimización, en particular *O3* para ver que tan bien puede optimizar el compilador.

El resultado esperado es que *assembler* debería ser superior, es decir insumir una menor cantidad de ciclos de reloj que las otras dos versiones. Aunque no esperaríamos un mal rendimiento de *C* con optimización.

2.7.3. Performance: *cropflip* vs. *caché*

El filtro *cropflip* parece poseer una particularidad. Utiliza accesos a memoria aleatorios que en principio deberían afectar el rendimiento en cuanto a si los datos pedidos están cacheados o no. Si el corte realizado es muy ancho que alto, por principio de vecindad espacial, el algoritmo debería tener un buen rendimiento, no así, cuando el corte es más alto que ancho.

Llevaremos a cabo entonces, un test para comprobar si este hecho tiene relevancia, esperando que, verdaderamente sea un factor influyente. Lo que haremos, como se comentó anteriormente, será limpiar la caché en cada corrida, para que así cada test no se vea afectado por el anterior. Luego esperaríamos ver que cuanto más alto que ancho sea el corte peor rendimiento tendrá el algoritmo debido a que como los datos no se encuentran cacheados, tendrá más accesos a memoria ram.

2.7.4. Performance: versiones de *ldr*

Elegimos el filtro *ldr* para analizar dos teorías relacionadas a SIMD.

A) Si igualamos la cantidad de accesos de la versión de *assembler* a la de *C* aún así la versión de *assembler* será más óptima.

B) Si en vez de dividir en punto flotante se reemplaza esa sección de código por operaciones con enteros: qué versión tiene mejor rendimiento? *assembler* con ops de punto flotante o *assembler* con ops en enteros?

```

;TEST 2: OPERACIONES CON ENTEROS
pmulld xmm5, xmm0 ; 0|sumargb*r|sumargb*g|sumargb*b == 0|sumargb*r|sumargb*g|sumargb*b
; puede cambiar el signo segun alpha
pmulld xmm5, xmm12 ; 0|alpha*sumargb*r|alpha*sumargb*g|alpha*sumargb*b
mov r10, rdx ; salvo resto
xor rdx, rdx
xor rax, rax
movd eax, xmm5 ; alpha*sumargb*b
div r13d ; alpha*sumargb*b/max
pxor xmm10, xmm10
movd xmm10, eax ; 0|0|0|alpha*sumargb*b
pslldq xmm10, 4 ; 0|0|alpha*sumargb*b|0
xor rdx, rdx
xor rax, rax
psrlldq xmm5, 4
movd eax, xmm5 ; alpha*sumargb*g
div r13d ; alpha*sumargb*g/max
movd xmm10, eax ; 0|0|alpha*sumargb*b/max|alpha*sumargb*g/max
pslldq xmm10, 4 ; 0|alpha*sumargb*b/max|alpha*sumargb*g/max|0
xor rdx, rdx
xor rax, rax
psrlldq xmm5, 4
movd eax, xmm5 ; alpha*sumargb*r
div r13d ; alpha*sumargb*r/max
movd xmm10, eax ; 0|0|alpha*sumargb*b/max|alpha*sumargb*g/max|alpha*sumargb*r/max
mov rdx, r10 ; recuperar resto

```

3. Tests

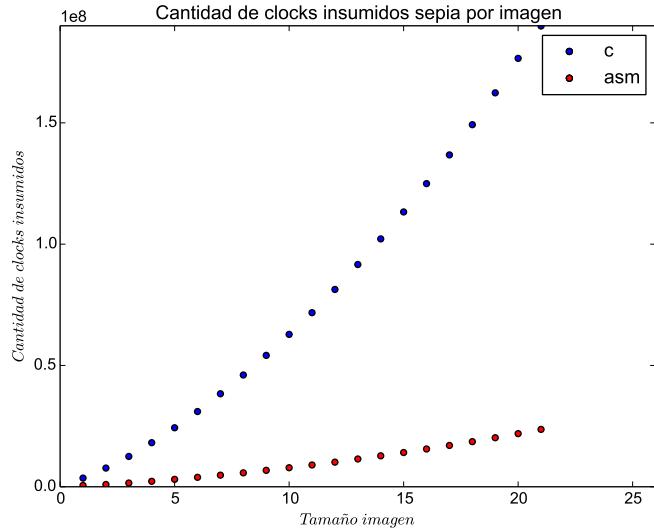
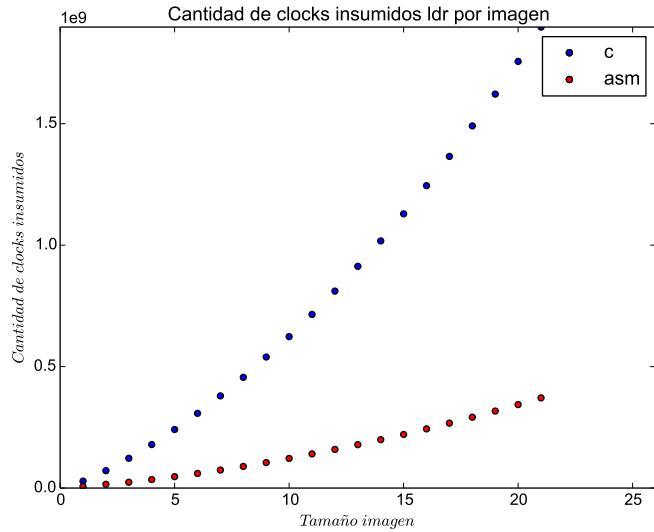
3.1. Aumentando resoluciones

Parámetros del test:

Cantidad de iteraciones por filtro 10. Aumentos 100x100.

$\alpha = 150$

Arrancamos en 1740x60 y vamos aumentando hasta 3840x2160.



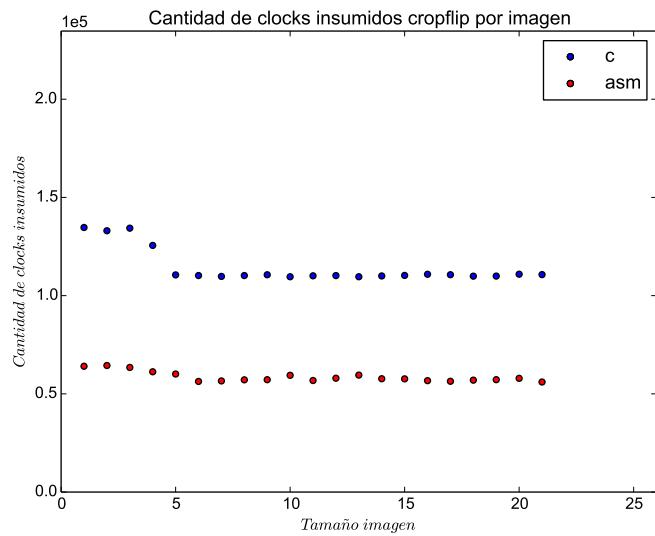
LDR Y SEPIA

Según nuestra hipótesis, con un aumento de forma cuadrática en el tamaño de las imágenes se obtiene una curva casi cuadrática, la curva resultante es muy parecida a una cuadrática pero con un menor crecimiento, en un principio supusimos que esto tenía que ver con que la imagen era más ancha que alta, así que invertimos el tamaño de la imagen original de 3840x2160 a 2160x3840 y generamos los mismos aumentos pero invertidos. El resultado obtenido es idéntico al original, por lo cual no tiene sentido mostrar un gráfico. Suponemos entonces que el factor de anchura o altura de la imagen en estos dos filtros no influye si aumentan los dos en iguales proporciones.

Luego se plantea que con dejar fijo un tamaño y aumentar el otro el resultado sería un aumento lineal de tiempos. Elegimos dejar fija la altura y aumentar el ancho desde 40x2160 a 3840x2160 y el resultado es que sí, efectivamente el aumento es lineal.

CROPFLIP

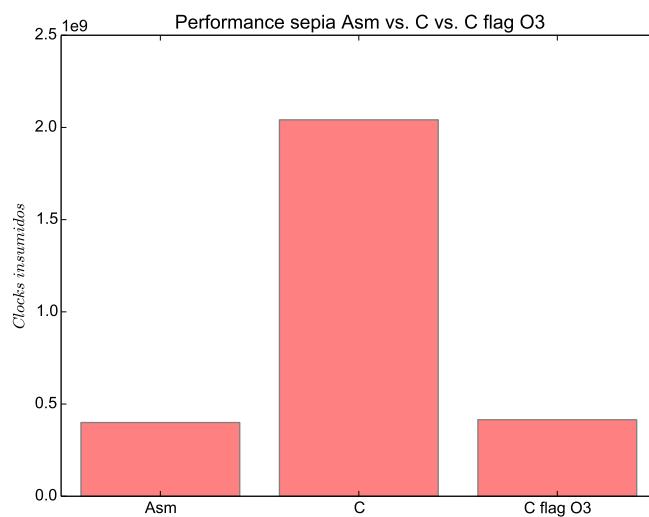
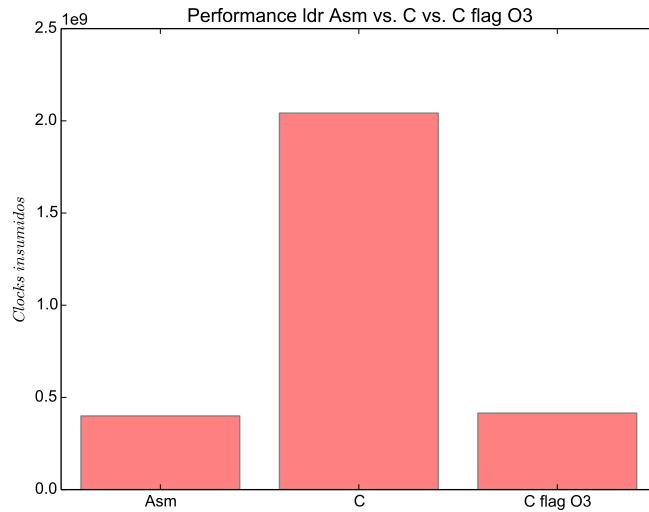
CORTE: 60 140 0 0

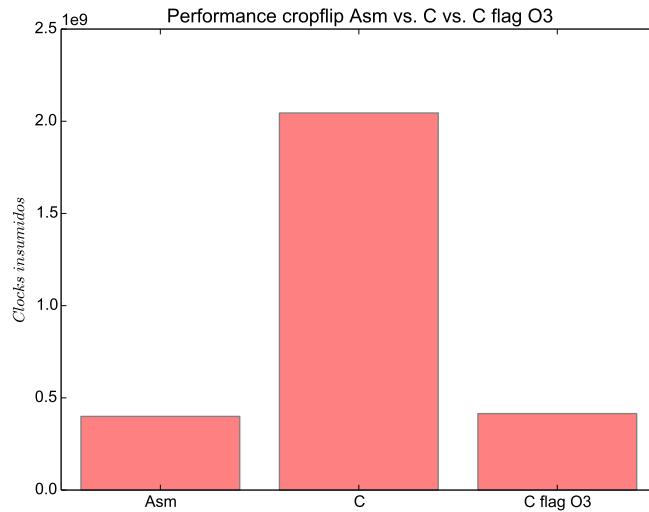


Por último observar que en cropflip se mantiene constante, es decir no cumple con lo esperado, (aunque si respeta que assembler es mejor que C).

3.2. Performance: implementaciones y filtros

Parámetros del test: Cantidad de iteraciones por filtro-version 20.

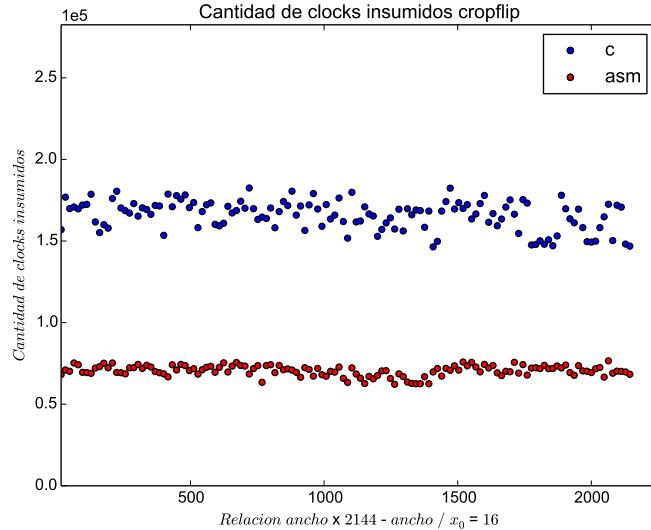




Concluimos que los algoritmos escritos en *assembler* son más rápidos que los escritos en *C*. Aunque podemos observar que en *C* compilado con flag de optimización *O3* llega a tener una performance muy similar.

3.3. Performance: cropflip vs. cache

Parámetros del test: Cantidad de iteraciones por filtro-version 20. empieza con un corte de 16x2144 y va restando al alto 16 y sumando al ancho la misma cantidad hasta llegar a 2144x16. tiene offsetX = 0 y offsetY = 500.



Observamos que el tiempo insumido en las diferentes corridas se mantiene comparativamente constante. Por lo cual, concluimos que la caché no parece ser un factor tan relevante en este caso como suponiamos.

3.4. Performance: versiones de ldr

Parámetros del test: Cantidad de iteraciones por filtro-version 20. ALFA 150

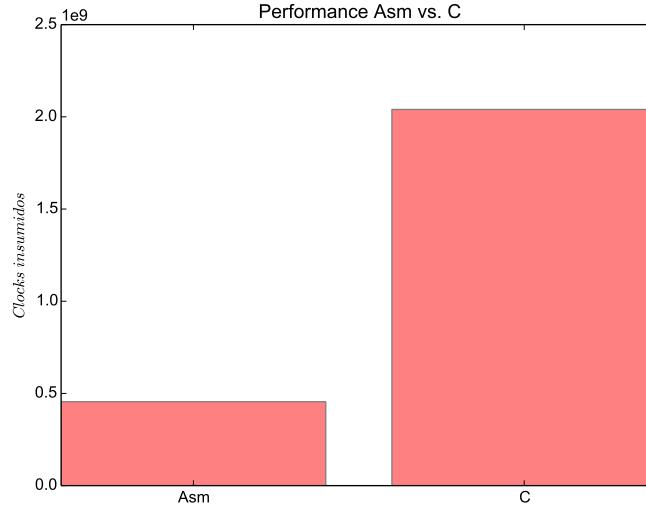


Figura 7: Test A

A) assembler supera claramente a la versión implementada en C. Claramente los accesos no suponen un problema, dado que por principio de vecindad espacial, en la cache al leer un pixel, se traera consigo varios contiguos. La diferencia radica en el procesamiento paralelo que SIMD nos permite.

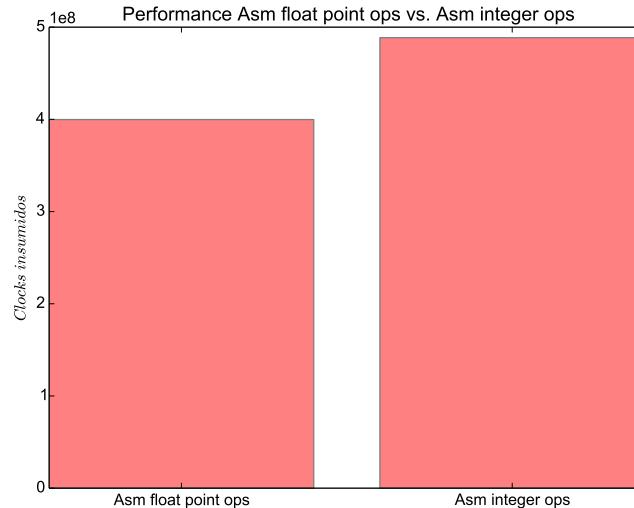


Figura 8: Test B

B) La versión con operaciones en punto flotante insume menor cantidad de clocks del reloj que la versión con operaciones en enteros: Para este caso particular, al menos, el hecho de tener que dividir cada escalar por separado no supone una ventaja, todo lo contrario. Y al final es más conveniente operar en punto flotante, dado que ya veniamos procesando en paralelo. Esto es particular a la implementación y a los cambios requeridos. Muy probablemente si se hubiese llevado a cabo en *sepia* hubiese arrojado otros resultados, debido a que podríamos operar con enteros sin perder paralelismo.

4. Conclusiones

Como pudimos comprobar, la importancia de SIMD para resolver este tipo de problemáticas es muy significativa. Pudo observarse que los tiempos para un algoritmo en un lenguaje convencional como *C* son muy elevados llegando a requerir aproximadamente 4 veces más de tiempo para realizar la misma tarea que en lenguaje *assembler* (es decir que con assembler se obtiene una eficiencia del 400 %)

Pudimos observar detalles particulares a cada implementación. En especial, obtuvimos que para el filtro *ldr* no era conveniente operar con enteros, pero esto parece estar más asociado al tipo de implementación y los cambios requeridos para lograr divisiones con enteros. Por ejemplo, si se hubiese realizado este mismo test sobre *sepia* muy probablemente hubiese arrojado otros resultados debido a que operaríamos con enteros sin perder paralelismo.

Otro resultado que pudimos observar, también con el filtro *ldr*, es que los accesos a memoria no suponen una perdida de rendimiento, dado que traer de a cuatro o traer de a un pixel es igual, debido a que la caché utiliza principio de vecindad espacial y al leer un pixel la cache trae varios pixeles contiguos. La diferencia radica principalmente en poder realizar operaciones de forma paralela, siendo nuevamente este el punto fuerte de SIMD.