



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	nicoh22@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	3
1.1. Motivaciones	3
2. Desarrollo	3
2.1. Imágenes	3
2.2. Implementaciones	4
2.3. Cropflip	4
2.3.1. Assembler SIMD	5
2.3.2. C	5
2.4. Sepia	6
2.4.1. Assembler SIMD	7
2.4.2. C	9
2.5. LDR: Low Dynamic Range	9
2.5.1. Assembler SIMD	10
2.5.2. C	14
2.6. Análisis experimental	15
2.6.1. Metodologías	15
2.6.2. Cosideraciones	17
2.7. Hipótesis	18
2.7.1. Comportamiento: Aumentando resoluciones	18
2.7.2. Performance caché: caliente vs. fria	18
2.7.3. Performance: implementaciones y filtros	19
2.7.4. Performance: versiones de <i>ldr</i>	19
3. Tests	22
3.1. Aumentando resoluciones	22
3.2. Performance: implementaciones y filtros	24
3.3. Performance: cropflip vs. cache	26
3.4. Performance: versiones de <i>ldr</i>	28

1. Introducción

Las tecnologías actuales de video en tiempo real y filtros de imagen son posibles, de manera eficiente y con gran calidad, desde hace unos años gracias a la eficiencia del código usado en sus implementaciones. Estos problemas intentaban resolverse con la ayuda de hardware adicional haciendo al producto final mucho más costoso, como es el caso de la FPU (floating-point unit) también conocido como coprocesador matemático, especializado en el cálculo de operaciones en coma flotante. En ausencia de una FPU, la CPU podía utilizar programas en microcódigo para emular una función en coma flotante a través de la unidad aritmético lógica (ALU), la cual reducía el coste del hardware a cambio de una sensible pérdida de velocidad.

Luego de unos años, el coste de los coprocesadores matemáticos fue lo suficientemente bajo como para incluir en el microprocesador de propósito general (CPU) las funciones del coprocesador matemático eliminando hardware adicional.

Sería casi imposible una implementación de un reproductor streaming como las actuales puramente en assembler o en un lenguaje de programación de uso común como C sin la ayuda de flags de optimización, debido a la enorme cantidad de computos requerida por la calidad de las imágenes generadas.

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 calculo 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 calculo de fisicas, iluminación, etc.

Tambien tiene aplicación en el procesamiento de filtros de imagen, obteniendo con los mismos, resultados notables frente a implementaciones puramente en assembler.

1.1. Motivaciones

En el siguiente informe haremos uso del set 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 voltea 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 recuadro obtenido se devuelve espejado 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 rectángulo 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, que corresponde en total a la fila de la imagen.

2.3.2. C

En principio, C compilado sin flags de optimización (es decir compilado con *O0*, modo default) el código final se resuelve puramente con instrucciones de assembler. Por lo tanto cada pixel se opera unitariamente.

```
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 512×512 píxeles 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 para el cálculo en punto flotante.

Para operar la imagen se procesa de a 4 píxeles. 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.

```
factores: DD 0.2, 0.3, 0.5, 0.0
alfamasc: DB 0, 0, 0, 0xFF, 0, 0, 0, 0xFF, 0, 0, 0, 0xFF, 0, 0, 0, 0xFF
alfainv: DB 0xFF, 0xFF, 0xFF, 0, 0xFF, 0xFF, 0xFF, 0, 0xFF, 0xFF, 0xFF, 0, 0xFF, 0xFF, 0xFF, 0
section .text
```

```
;Notacion:
;px = pixel input
;sumax = sumatoria de las componentes de px
;px' = pixel output deseado
;El contenido de los registros XMM se muestra del
; bit mas significativo al menos significativo
```

```
_sepia_asm:
;rdi *src
;rsi *dst
;edx int cols
;ecx int filas
;r8d int src_row_size
;r9d int dst_row_size
```

```
push rbp
mov rbp, rsp

mov eax, edx
mul ecx
mov ecx, eax
sar ecx, 2; ecx/4 me muevo cuatro pixeles por iteracion
```

```
movdqu xmm7, [alfainv] ; XMM7 = | 00 | FF | FF | FF | ...
movdqu xmm8, [alfamasc]; XMM8 = | FF | 00 | 00 | 00 | ...
movups xmm0, [factores]
pxor xmm6, xmm6
```

```
.ciclo:
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 |
```

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

```
phaddq xmm4, xmm4
```

Nombre Apellido, Nombre Apellido

```
; un unpack mas, multiplico de a un pixel
```

```
outdq xmm1, xmm1; suma0 as float
```


2.4.2. C

Como sucede en *cropflip*, sin flags de optimización el código compilado con *O0* no dispone de las ventajas de SIMD.

```
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 píxel y le añade un porcentaje α del de sus vecinos.

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

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

donde

$$ldr_{i,j}^K = I_{i,j}^K + \alpha \frac{sumargb_{i,j}}{\max} \cdot I_{i,j}^K$$

$$sumargb_{i,j} = suma_{i,j}^r + suma_{i,j}^g + suma_{i,j}^b$$

$$\max = 5 * 5 * 255 * 3 * 255$$

255 y 0 corresponden a valores de saturación y finalmente $suma_{i,j}^K$ corresponde a:

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



(a) Original



(b) LDR

Figura 2: α : 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:

$$O_{i,j}^K = I_{i,j}^K \text{ si } i < 2 \vee j < 2 \vee i + 2 \leq t_{amy} \vee j + 2 \leq t_{max}$$

(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 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 valiendose 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 para el pixel 5
movdqu xmm1, [rdi + r12*pixelSize] ; Li3|Li2|Li1|Li0

movdqu xmm9, xmm1
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 xmm1, xmm7 ; 0|0|0|r3|0|g3|0|b3|0|0|0|r2|0|g2|0|b2
phaddw xmm1, xmm11 ; 0|0|0|0|r3|g3+b3|r2|g2+b2
punpcklwd xmm9, xmm11 ; r1|g1+b1|r0|g0+b0
punpcklwd xmm1, xmm11 ; r3|g3+b3|r2|g2+b2
paddw xmm1, xmm9 ; r3+r1|g3+b3+g1+b1|r2+r0|g2+b2+g0+b0
movdqu xmm9, xmm1
psrldq xmm9, 8 ; 0|0|r3+r1|g3+b3+g1+b1
paddw xmm1, xmm9 ; r3+r1|g3+b3+g1+b1|r2+r0+r3+r1|g2+b2+g0+b0+g3+b3+g1+b1
pslldq xmm1, 8 ; r2+r0+r3+r1|g2+b2+g0+b0+g3+b3+g1+b1|0|0
psrldq xmm1, 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|a4|r4|g4|b4

pslldq xmm9, 12 ; a4|r4|g4|b4|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
punpcklwd xmm9, xmm11 ; 0|r4|0|g4+b4|0|0|0|0
psrldq xmm9, 8 ; 0|0|r4|g4+b4
paddw xmm1, xmm9 ; 0|0|r2+r0+r3+r1+r4|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4
movdqu xmm9, xmm1
psrldq xmm9, 4 ; 0|0|0|r2+r0+r3+r1+r4
paddw xmm1, xmm9 ; 0|0|r2+r0+r3+r1+r4|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4
pslldq xmm1, 12 ; g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4|0|0|0
psrldq xmm1, 12 ; 0|0|0|g2+b2+g0+b0+g3+b3+g1+b1+g4+b4+r2+r0+r3+r1+r4
paddw xmm0, xmm1 ; suma hasta la i-esima 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. Asi 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, $canal \times sum \times max \times alfa$ 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 $3 \times (filas-2) \times (columnas-2)$ 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*.


```
punpcklbwAndCleanAlpha: DB 0x00, 0x88, 0x01, 0x89, 0x02, 0x8A, 0x83, 0x8B, 0x04, 0x8C, 0x05, 0x8D, 0x06, 0x8E, C  
punpckhbwAndCleanAlpha: DB 0x08, 0x81, 0x09, 0x82, 0x0A, 0x83, 0x8B, 0x84, 0x0C, 0x85, 0x0D, 0x86, 0x0E, 0x87, C  
saveOnePixelShifter: DB 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0xFF, 0xFF, 0xFF  
repeatDw: DB 0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03, 0x00, 0x01, 0x02, 0x03  
maxValue: DD 0x004A6A4B ; check this 4876875
```

```

section .text
;void _ldr_asm      (
;unsigned char *src, rdi
;unsigned char *dst, rsi
;int cols, edx
;int filas, ecx
;int src_row_size, r8d -> no se usa
;int dst_row_size, r9d -> no se usa
;int alpha) rsp-8

; r8 posicion actual
; r9 contador columnas

_ldr_asm:
push rbp
mov rbp, rsp
push rbx
push r12
push r13
push r14
push r15

xor rbx, rbx
xor r12, r12
xor r13, r13
xor r14, r14
xor r15, r15

mov ebx, [rbp+16] ; alpha
mov r13d, [maxValue] ; MAX

cmp ebx, -255
jnl .sinCambios
cmp ebx, 255
jg .sinCambios
cmp edx, 4
jle .sinCambios ; si tengo menos de cuatro filas terminar.
cmp ecx, 4
jle .sinCambios ; si tengo menos que cuatro columnas terminar.

xor r8, r8 ; posicion actual
xor r9, r9 ; j = 0
mov r8d, edx ; r8 = cols

mov r15d, edx ; r15d = cols
xor rdx, rdx
mov r14d, ecx ; r14d = filas.
sub r14d, 2 ; filas-2
xor rax, rax ; limpio para usar en multiplicacion.
mov eax, r15d
mul r14d ; edx:eax = r15d*r14d = cols*(filas-2).
mov ecx, edx
shl rdx, 32
mov ecx, eax ; ecx = r15d*r14d = cols*(filas-2). contador loop

```

Nombre Apellido, Nombre Apellido, Nombre Apellido	13
---	----

```
xor r11, r11
mov r11d, r15d
sub r11d, 2 ; cols-2 = colsToProcess
```

2.5.2. C

Como ya hemos explicado, C compilado en modo default, carece de paralelismo en las operaciones y por lo tanto *ldr* se resolverá obteniendo cada pixel y operando unitariamente. Al final el algoritmo es similar pero con más accesos a memoria para lectura y escritura y operaciones de punto flotante unitarias.

```
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. Cuestiones como esta, serán tenidas en cuenta en la toma de mediciones.

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 vecinidad 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 último y algunas cuestiones 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 pondremos 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.

Cada vez que se corra un filtro se tomará la diferencia entre el comienzo y el final del proceso y se dividirá este valor por la cantidad de píxeles de la imagen (resolución de la imagen) para poder analizar todas las muestras en la misma unidad ($ticks/cantidaddepixeles$) y que además nos permitirá observar mejor cambios en el comportamiento de las muestras.

Luego al conjunto de mediciones tomadas para una misma imagen calcularemos una media α -podada 0.5 (promedio intercuartil) sobre el total de las corridas de un experimento. La poda se realizará a derecha, eliminando así los outliers más grandes que puedan existir (las muestras se ordenan de menor a mayor). Lo cual es más que suficiente para 100 iteraciones sobre una imagen (que es la cantidad de iteraciones que utilizaremos en todos los tests), que nos deja 50 muestras luego de aplicar la poda. Además tomaremos el desvío standard muestral, utilizando el mismo set de datos al que aplicamos la poda α , para poder medir que dispersion tienen los mismos con respecto a la media. De esta manera podremos saber que si la dispersion de los datos es muy grande, para dos curvas aparentemente distanciadas en promedio, tal vez no haya diferencias significativas.

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) Intel Core i5-2430M @ 2.40GHz con caché L2 de 3MB.

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 3MB el algoritmo siguiente debería garantizar que la caché se ocupará con información que no altere los resultados de los tests.

Además cambiando el valor de la variable *fria* por uno podremos testear la cache precalentada con la imagen de testeo para ver como influye en los tiempos de proceso de cada algoritmo el estar la imagen cacheada totalmente o parcialmente.

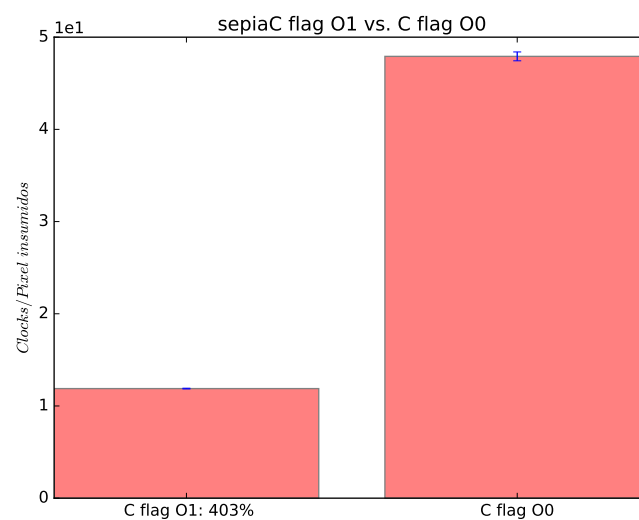
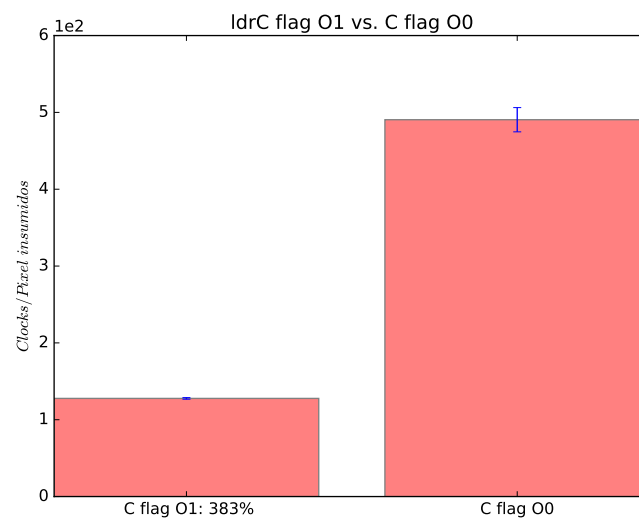
```
int fria = 1;
if (fria == 1) {
char *basura = (char*)malloc(sizeof(char)*786432);
srand(5);
int i = 0;
while (i < 786432) {
basura[i] = rand() % 27;
i++;
}
long int suma = 0;
i = 0;
while (i < 786432) {
suma += basura[i];
i++;
}
free(basura);
}else {
unsigned int *src_matrix = config->archivo_entrada;
unsigned int size = sizeof(config->archivo_entrada);
int argbSum = 0;
for (int i = 0; i < size; i++)
{
int argb = src_matrix[i];
argbSum += argb;
}
}
```


2.6.2. Cosideraciones

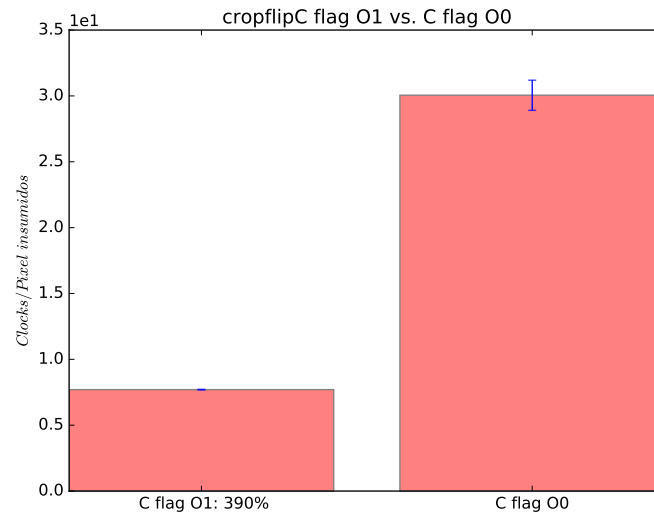
Para que las comparaciones entre código *asm* y *C* sean más justas, utilizaremos como version default de código *C* flag de optimización *O1*. El motivo de esta elección es que el código generado por *O0* no realiza grandes optimizaciones y solo utiliza *SIMD* para escalares por lo cual la performance es muy baja.

De todas maneras, uno de los fines de este informe es mostrar que sin flags de optimización, sería a día de hoy casi imposible realizar, por ejemplo, reproducción de video en tiempo real con gran carga de procesamiento en un computador de sobremesa. Es decir, que código generado con *SIMD* ya sea por un programador o por el compilador con ayuda de flags de optimización es muy importante para esta problematica.

Veamos entonces cuanta diferencia existe entre flags *O1* y *O0* para cada filtro.



Como podemos notar, se logran eficiencias elevadas utilizando solamente *O1* y en principio podriamos lograr más eficiencia utilizando *O2* y *O3*. Así que por fines prácticos eligiemos el último para ver que tan



lejos podemos llegar.

2.7. Hipótesis

2.7.1. Comportamiento: Aumentando resoluciones

Nos interesa saber como se comportan los algoritmos para cada filtro en sus dos variantes (*asm*, $C - O1$) cuando se varia el tamaño de 3. Es decir, analizaremos el comportamiento de los algoritmos frente a variaciones de tamaño

El resultado esperado para todos los filtros en ambas versiones es que el tiempo aumente linealmente para aumentos lineales en el tamaño de las imágenes. En particular como lo que haremos será tomar cada medición como $tics / (ancho \times alto)$ esta linealidad será en realidad una constante sobre las mediciones.

Es decir que no varia el tiempo para un pixel si no que varia el tiempo en funcion de la cantidad total de pixeles.

Además, sería deseable que la version implementada en *asm* se mantenga por encima en performance (por debajo en cuanto a tiempos) de $C - O1$.

En cropflip, en particular, tomaremos un tamaño proporcional a la imagen de corte para realizar las mediciones que será proporcional al tamaño de la imagne original:

1. $width - 16$
2. $height - 16$
3. $offsetY : 16$
4. $offsetX : 16$

De manera que el corte varie con el tamaño.

2.7.2. Performance caché: caliente vs. fria

El fin de este test es ver si existen diferencias entre correr los algoritmos con la cache en frio (es decir sin la imagen previamente leida) con respecto a caché caliente (imagen previamente leida).

A priori, el principio de localidad espacial con el cual trabaja la memoria caché podría hacer que la diferencia entre ambos tests no sea tan notable. Sabemos que una línea de caché en un procesador intel es de 64 bytes, por lo cual podremos almacenar hasta 16 píxeles en una línea. Veremos luego de los tests con más detalle porque podría influir esto.

Para *cropflip* tomaremos el corte de la misma manera que se realizará en el test de resoluciones. La versión de código utilizada en estos tests será en *asm*.

2.7.3. Performance: implementaciones y filtros

Como se comentó, el principal objetivo del informe es comprobar si *assembler SIMD* supone una ventaja frente a código compilado sin flags de optimización. Hemos podido comprobar que con flags *O1* se obtiene una performance elevada frente a *O0*.

En este test correremos todos los filtros comparando sus versiones de *assembler SIMD* contra *C – O1* y además incluiremos *C* con flags de optimización *O3* para ver que tan bien puede optimizar el compilador.

La idea es ver que ventaja supone programar a mano en *SIMD* frente a utilizar flags de optimización en el proceso de compilación.

2.7.4. Performance: versiones de *ldr*

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

A) Deseamos ver que sucede si igualamos la cantidad de accesos a memoria de la imagen en la implementación de *assembler SIMD* a la de *C – O1* (es decir, cambiando los accesos de a 128 bits por accesos de 32 bits) manteniendo las operaciones de los píxeles sin cambios y ver si aún así la versión de *assembler SIMD* sigue siendo más óptima.

El motivo de este test es comprobar que el principio de localidad espacial se cumple, es decir, que al acceder a una posición de memoria, una línea completa se cargará en memoria caché, por lo cual no importa si accedemos con instrucciones de 128 o 32 bits, siempre estaremos accediendo a la caché para las posiciones de memoria contiguas.

El código *assembler SIMD* sin accesos empaquetados cambia de la siguiente manera:

```
;mascaras
movd xmm7, [alfainv]
    pslldq xmm7, 4
movd xmm7, [alfainv + 4]
    pslldq xmm7, 4
movd xmm7, [alfainv + 8]
    pslldq xmm7, 4
movd xmm7, [alfainv + 12]
    pslldq xmm7, 4

movd xmm8, [alfamasc]
    pslldq xmm8, 4
movd xmm8, [alfamasc + 4]
    pslldq xmm8, 4
movd xmm8, [alfamasc + 8]
    pslldq xmm8, 4
movd xmm8, [alfamasc + 12]
    pslldq xmm8, 4

movd xmm0, [factores]
    pslldq xmm0, 4
movd xmm0, [factores + 4]
    pslldq xmm0, 4
movd xmm0, [factores + 8]
    pslldq xmm0, 4
movd xmm0, [factores + 12]
    pslldq xmm0, 4

;leer de memoria pixeles
;movdqu xmm1, [rdi]; XMM1 = | p3 | p2 | p1 | p0 |
movd xmm1, [rdi]
    pslldq xmm1, 4
movd xmm1, [rdi + 4]
    pslldq xmm1, 4
movd xmm1, [rdi + 8]
    pslldq xmm1, 4
movd xmm1, [rdi + 12]

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;escribir a memoria pixeles
;movdqu [rsi], xmm1
movd [rsi], xmm1
    psrldq xmm1, 4
movd [rsi + 4], xmm1
    psrldq xmm1, 4
movd [rsi + 8], xmm1
    psrldq xmm1, 4
movd [rsi + 12], xmm1
```

Veamos que el código C compilado con $O1$ tiene una cantidad de accesos para lectura y escritura a memoria similar a la de *assembler SIMD* modificado

```

<sepia_c>:
test    ecx,ecx
jle     114 <sepia_c+0x114>
push    r15
push    r14
push    r13
push    r12
push    rbp
push    rbx
movsxd  r13,r8d
movsxd  r14,r9d
mov     rbp,rsi
mov     rbx,rdi
mov     r12d,0x0
movsd   xmm7,QWORD PTR [rip+0x0]      # 2c <sepia_c+0x2c>

movsd   xmm6,QWORD PTR [rip+0x0]      # 34 <sepia_c+0x34>

movsd   xmm5,QWORD PTR [rip+0x0]      # 3c <sepia_c+0x3c>

movapd  xmm4,xmm6
movsd   xmm3,QWORD PTR [rip+0x0]      # 48 <sepia_c+0x48>

movapd  xmm2,xmm6
mov     r9d,0xffffffff
jmp     f5 <sepia_c+0xf5>
mov     BYTE PTR [r11],al
movzx   eax,BYTE PTR [r10+0x3]
mov     BYTE PTR [r11+0x3],al
add     r8d,0x1
cmp     edx,r8d
je      e6 <sepia_c+0xe6>
lea     r10d,[r8*4+0x0]

movsxd  r10,r10d
lea     r11,[rdi+r10*1]
add     r10,rsi
movzx   eax,BYTE PTR [r10+0x2]
movzx   r15d,BYTE PTR [r10+0x1]
add     eax,r15d
movzx   r15d,BYTE PTR [r10]
add     eax,r15d
pxor    xmm0,xmm0
cvtsi2sd xmm0,eax
movapd  xmm1,xmm0
mulsd   xmm1,xmm7
mov     eax,r9d
ucomisd xmm1,xmm6
ja      af <sepia_c+0xaf>
cvtttsd2si eax,xmm1
mov     BYTE PTR [r11+0x2],al
movapd  xmm1,xmm0
mulsd   xmm1,xmm5
mov     eax,r9d
ucomisd xmm1,xmm4
ja      c8 <sepia_c+0xc8>
cvtttsd2si eax,xmm1
mov     BYTE PTR [r11+0x1],al
mulsd   xmm0,xmm3
mov     eax,r9d
mov     xmm0,xmm0
ja      57 <sepia_c+0x57>
cvtttsd2si eax,xmm0
jmp     57 <sepia_c+0x57>

```

B) Deseamos ver que sucede si en lugar de dividir en punto flotante empaquetados se reemplaza esa seccion de código por operaciones con enteros empaquetados: que version tiene mejor rendimiento? *assembler* con ops de punto flotante o *assembler* con ops en enteros?

Para llevarlo a cabo cambiaremos los valores del filtro que se aplican a cada canal: 0.2, 0.3, 0.5 por 2, 2, 2 en ambas versiones, dado que, para la version de enteros no disponemos de punto flotante y queremos tener una representación de la cual conozcamos los valores que manejamos.

Lo que si no haremos en la version de punto flotante será convertir los valores de los canales a enteros luego de aplicar la operación y no empaquetaremos en ninguna de las dos versiones los resultados que se obtengan. De esta manera eliminaremos la mayor cantidad de proceso extra para que las comparaciones sean lo más justas posibles. Luego todo será igual, pero a la ahora de aplicar el filtro la version de enteros será:

```
pmulld xmm1, xmm0; XMM1 = |*|suma0*2|suma0*2|suma0*2|
pmulld xmm2, xmm0; XMM2 = |*|suma2*2|suma2*2|suma2*2|
pmulld xmm3, xmm0; XMM3 = |*|suma1*2|suma1*2|suma1*2|
pmulld xmm4, xmm0; XMM4 = |*|suma3*2|suma3*2|suma3*2|
```

Y la version de punto flotante será:

```
mulps xmm1, xmm0; XMM1 = |*|suma0*2|suma0*2|suma0*2|
mulps xmm2, xmm0; XMM2 = |*|suma2*2|suma2*2|suma2*2|
mulps xmm3, xmm0; XMM3 = |*|suma1*2|suma1*2|suma1*2|
mulps xmm4, xmm0; XMM4 = |*|suma3*2|suma3*2|suma3*2|
```

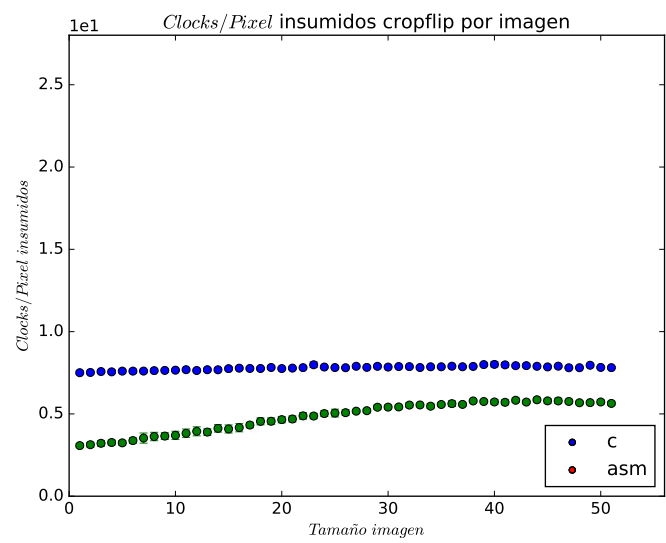
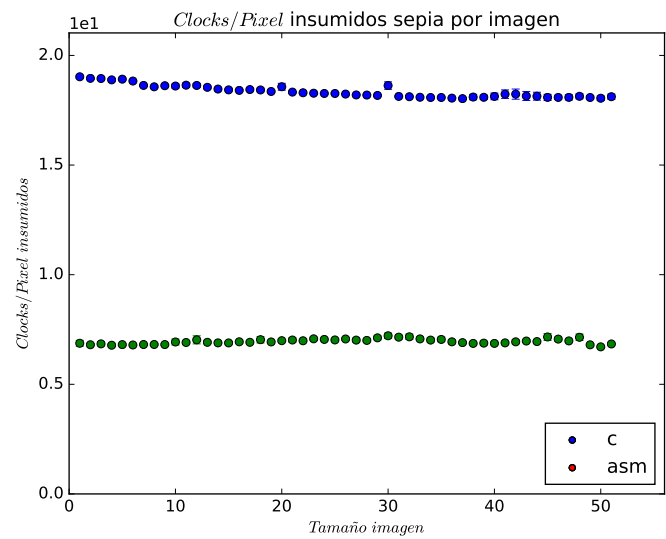
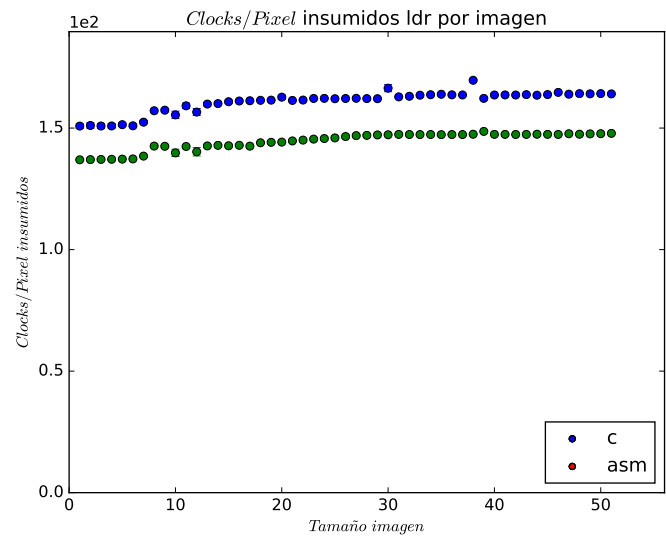
3. Tests

3.1. Aumentando resoluciones

Todos los test se realizan con la caché en frio. Se comapra además en un mismo gráfico las versiones de *assembler SIMD* y *C – O1*.

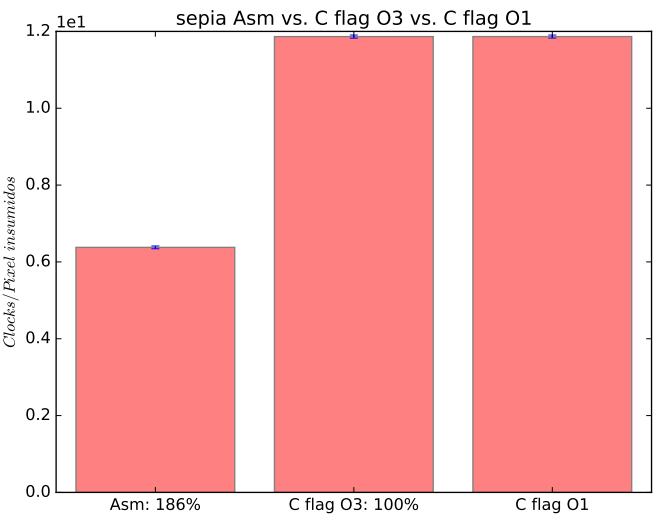
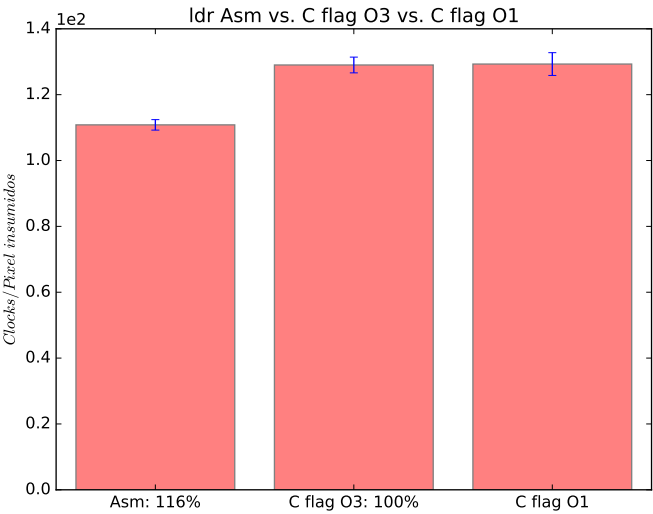
Para cada filtro e implementación: Se procesan 51 imágenes *lena32.bmp* desde la resolución 424x424 con el id: 1 (539KB) aumentando de 16x16 hasta la resolución 1224x1224 con el id: 51 (4.5MB). Cada imagen se corre 100 veces y a cada muestra, que serán los tics de reloj totales para procesar la imagen, se la divide por el tamaño de la imagen, y se toma el promedio muestral intercuartil y el desvio estandar sobre la poda realizada.

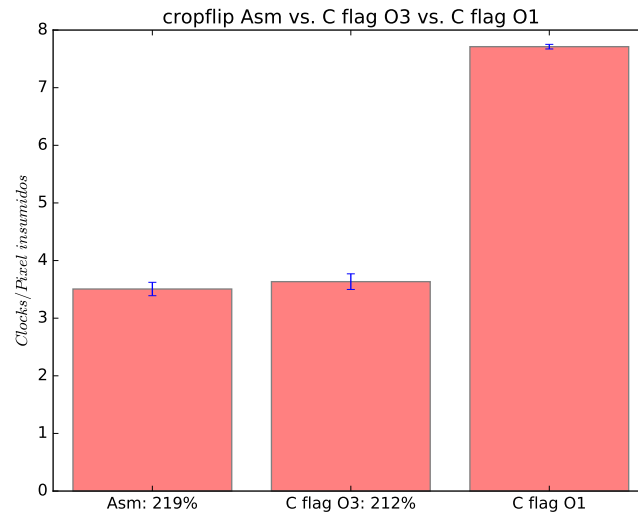
Puede comprobarse que las mediciones se mantienen comparativamente constantes tanto en la version de *assembler SIMD* como en *C – O1* Esto se corresponde con nuestra hipótesis inicial de que aumentos lineales en la cantidad de pixeles se corresponden con aumentos lineales en la cantidad de tiempo total, y que en particular por pixel el tiempo se mantiene aproximadamente constante como muestran los gráficos.



3.2. Performance: implementaciones y filtros

Para cada filtro e implementación (*assembler SIMD*, $C - O1$ y $C - O3$): Se procesa la imagen *lena32.bmp* 100 veces, se divide el resultado de cada muestra por el tamaño de la imagen (512x512) y se calcula el promedio intercuartil y su desvío estandar



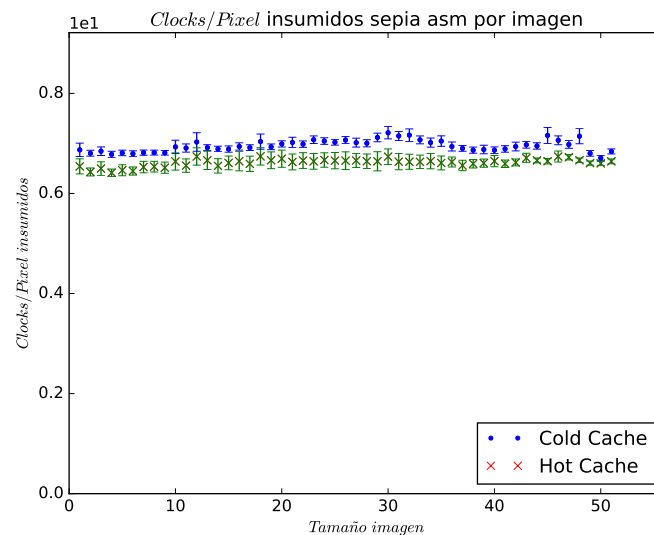
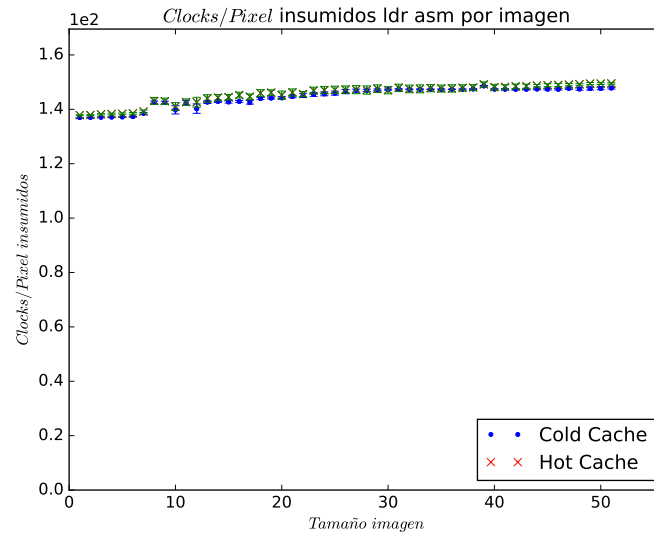


Como puede observarse, salvo en el caso del filtro *cropflip*, el flag de optimización *O3* no supone una ventaja sobre *O1* pero la implementación en *assembler SIMD* logra una buena performance en comparación. Al contrario, en *cropflip* puede observarse que *O3* está muy cerca de *assembler SIMD*. Esta diferencia la atribuimos al hecho de que el filtro *cropflip* no realiza operaciones aritmeticas sobre los pixeles de la imagen. Esto facilita la optimizacion del codigo por parte del compilador.

Con lo cual nuestra conclusion es que dependiendo del filtro o tarea que querramos realizar el compilador podrá optimizar más o menos y será cuestion del desarrollador evaluar si es necesario realizar el código en *assembler SIMD* sin ayuda del compilador.

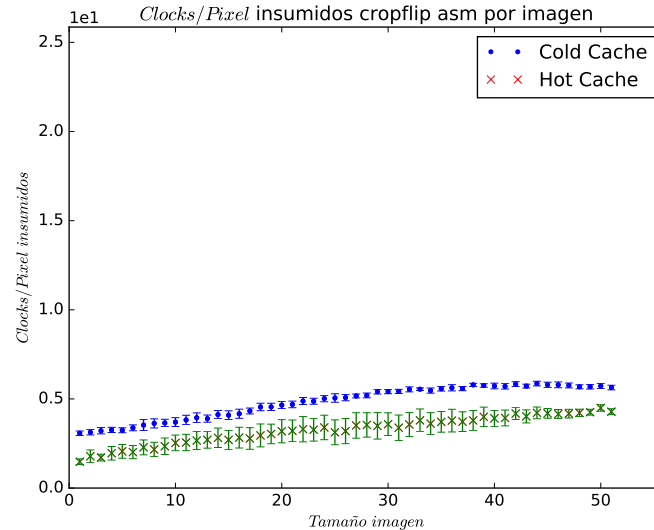
3.3. Performance: cropflip vs. cache

Para realizar el test, como se mencionó, se setea en el flag frio el valor uno para testear con la cache en frio y luego en cero para testear en caliente. Se procesan 51 imágenes *lena32.bmp* desde la resolución 424x424 con el id: 1 (539KB) aumentando de 16x16 hasta la resolución 1224x1224 con el id: 51 (4.5MB). Cada imagen se corre 100 veces y a cada muestra, que serán los tics de reloj totales para procesar la imagen, se la divide por el tamaño de la imagen, y se toma el promedio muestral intercuartil y el desvío estandar sobre la poda realizada.



Podemos observar que salvo en *ldr* todos los demás filtros mantienen el test de caché caliente por debajo del test en frio. Aunque por muy poca diferencia (menor al 0,2) En principio esto es algo que tiene sentido. Una linea de caché tiene 64 bytes, por lo cual como ya hemos mencionado solo entrarn 16 pixeles en una linea.

Para el test en caliente las imágenes de menos de 3MB estarán completamente en memoria caché y tendremos $16/16 \text{ hits} = 100 \%$ de *hits* por linea. Es decir que nunca habrá necesidad de ir a buscar a memoria RAM pixeles de la imagen.



Para el test en frio lo que sucederá no es muy diferente.

Al no estar la imagen previamente en caché, el primer acceso será un *miss* pero como el controlador caché utiliza el principio de localidad espacial traera a caché un bloque contiguo a una linea de la caché. Con lo cual los 15 accesos siguientes serán *hits* por tendremos $15/16 \text{ hits} = 93,75 \%$ de *hits*. Cuando la imagen supere el tamaño de la caché lo que puede estar sucediendo es que parte de la imagen esta en memoria caché, con lo cual se sigue obteniendo un rendimiento por debajo del test en frio.

El caso del filtro *ldr* puede deberse a que las operaciones realizadas no dependen tanto de si la imagen está en caché o no. En particular, por como fué implementada la suma, el controlador caché podria estar desalojando y alojando lineas continuamente con lo cual no se beneficia tanto de las ventajas de esta memoria.

3.4. Performance: versiones de ldr

Para cada implementación (*assembler SIMD* y *C – O1*): Se procesa la imagen *lena32.bmp* 100 veces, se divide el resultado de cada muestra por el tamaño de la imagen (512x512) y se calcula el promedio intercuartil y su desvío estandar.

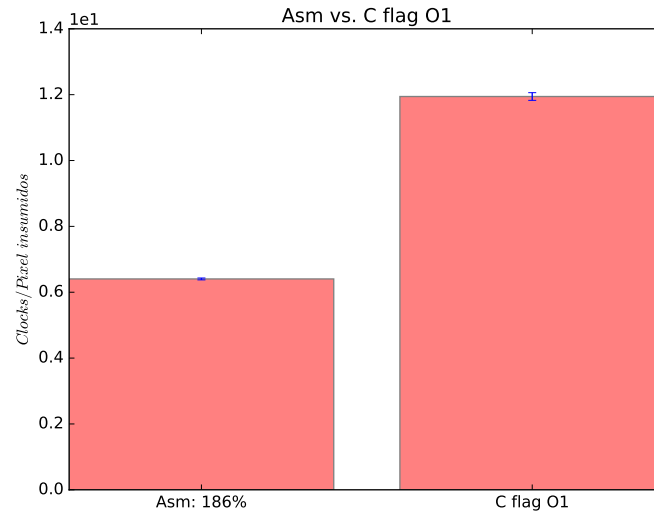


Figura 4: Test A

A) *assembler SIMD* supera claramente a la version implementada en *C – O1*. Como se prevee, usar instrucciones *SIMD* empaquetadas para acceder a memoria tanto en escritura como lectura no supone una ventaja frente a accesos escalares.

Como se comentó, para el caso de las lecturas, el controlador de caché utiliza el principio de localidad espacial. Por esta razón siempre habrá la misma cantidad de *misses* y *hits* sin importar si usamos instrucciones empaquetadas o de escalares para acceder a los datos.

La diferencia radica principalmente en que los datos se operan con instrucciones *SIMD* a diferencia de la versión compilada en *O1* que utiliza instrucciones *assembler* para 64 bits.

B) Como puede observarse, la performance lograda al operar con enteros empaquetados es exagerada. Por lo cual siempre que pueda optarse por no realizar cálculos de floats empaquetados estaremos tomando una buena decisión.

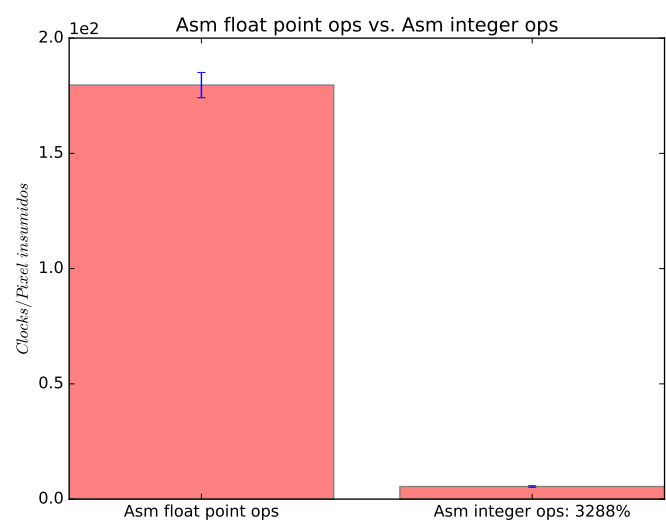


Figura 5: Test B