DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Procesamiento de Imagenes

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
López Luque Matias	192/14	matimatote@gmail.com
Rodríguez Santiago	094/14	santi_rodri_94@hotmail.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

Índice

1. Introduccion

El objetivo del presente informe es presentar la documentación de la implementación de tres filtros de imágenes, en C y en ASM, y realizar una comparación de desempeño con el propósito de ver diferencias de performance.

1.1. Filtro CropFlip

El filtro cropflip es una unión de dos filtros: crop y vertical-flip. Recorta una parte de la imagen original y la voltea verticalmente.

Este filtro toma como parámetro la cantidad de columnas y filas que queremos recortar y desde que columna y fila queremos comenzar a cortar la imagen original.

(a) Imagen Original

(b) Después de aplicar el filtro CropFlip

Figura 1: Filtro Cropflip

1.2. Filtro Sepia

El filtro Sepia es un filtro que cambia la información del color de cada pixel en la imagen, desplazando la intensidad del azul y, en menor medida, verde hacia el canal rojo. El filtro no toma parámetros.

(a) Imagen Original

(b) Después de aplicar el filtro Sepia

Figura 2: Filtro Sepia

1.3. Filtro Low Dynamic Range

El filtro Low Dynamic Range (LDR) transforma una imagen modificando cada pixel en base a sus vecinos. La imagen resultante intensifica los colores de los pixeles (acerca a 255 el valor de cada canal) que estan rodeados por pixeles intensos, y deja oscuros a aquellos rodeados por pixeles oscuros.

Este filtro toma un solo parametro, que es un Alpha entre -255 y 255 que describe la intensidad de la modificación de brillo. A valores negativos de alpha, el efecto sera inverso (pixeles en zonas claras de la imagen se oscurecen en vez de intensificarse).

(a) Imagen antes de aplicar el filtro

(b) Filtor con valor positivo de alpha

(c) Filtro con valor negativo de alpha

Figura 3: Filtro Low Dynamic Range

2. Desarrollo

2.1. Cropflip

Implementación en C

El filtro cropflip implementado en C consiste en dos ciclos anidados. El ciclo exterior itera sobre las filas y el interior sobre las columnas a cortar. Una vez dentro del ciclo interior copia todos los valores BGRA del píxel.

```
        while(i < tamy){
            j = 0;
            while(j < tamx){
                //Nos ubicamos donde nos corresponde en la matriz de pixeles
                bgra_t *p_d = (bgra_t*)&dst_matrix[tamy-i-1][j * 4];
                bgra_t *p_s = (bgra_t*)&src_matrix[offsety+i][(offsetx+j) * 4];
                //Copiamos los valores
                p_d->b = p_s->b;
                p_d->g = p_s->g;
                p_d->r = p_s->r;
                p_d->a = p_s->a;
                j = j+1;
            }
         i = i+1;
        };
```

Implementación en ASM-SIMD

El filtro cropflip implementado en ASM consiste en un cicloFilas que itera por las filas de la imagen y un cicloColumnas que itera por las columnas de a cuatro píxeles.

En el ciclo columna en vez de procesar 1 o 2 píxeles, aprovecha la tecnología SIMD y utiliza un registro XMM de 128 bits para guardar 4 píxeles y moverlos a la imagen destino.

```
                    XMM1 = [ p3 | p2 | p1 | p0 ]
```

2.2. Sepia

Implementación en C

La implementación en C del filtro realiza dos ciclos. El ciclo externo es sobre las filas, y el interno sobre las columnas. De esta manera se recorren los píxeles de izquierda a derecha y de abajo hacia arriba.
Para cada píxel, se toma el valor de sus tres canales (RGB) y se suman. Por último, se le es asignado a cada canal la suma multiplicada por un número decimal, que puede interpretarse como un porcentaje de la suma anterior que le corresponderá. Esto es, 50 % del valor total va al canal Rojo, 30 % al Verde y 20 % al Azul (0.5, 0.3 y 0.2). El valor nuevo de cada canal es revisado para evitar asignar valores mayores a 255.

```
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];
            *p_d = *p_s;
            suma = p_s->r + p_s->g + p_s->b;
            p_d->r = (suma * 0.5 > 255) ? 255 : (suma * 0.5);
            p_d->g = (suma * 0.3 > 255) ? 255 : (suma * 0.3);
            p_d->b = (suma * 0.2 > 255) ? 255 : (suma * 0.2);
            p_d->a = p_s->a;
        }
    }
```

Implementación En ASM-SIMD

La implementación de Sepia en ASM sigue la misma pauta que la implementación en C, pero aprovecha las instrucciones proporcionadas por la extensión SSE para realizar las mismas operaciones de manera más efectiva.

Como primer paso, se utilizan dos ciclos anidados para recorrer la imagen, utilizando registros de propósito general. Sin embargo, a diferencia de la versión de C, se traen 4 píxeles por vez, aprovechando los registros XMM de 128 bits. Estos 4 píxeles tienen precisión de DWORD (cada canal es un BYTE), y se traen al registro XMM0.

Se utilizan dos registros XMM (XMM0 y XMM1) para aumentar la precisión de los píxeles a QWORD (la precisión de los canales pasa a ser WORD) utilizando las instrucciones punpcklbw y punpckhbw. Esto es necesario para garantizar que a la hora de sumar no ocurra un overflow.

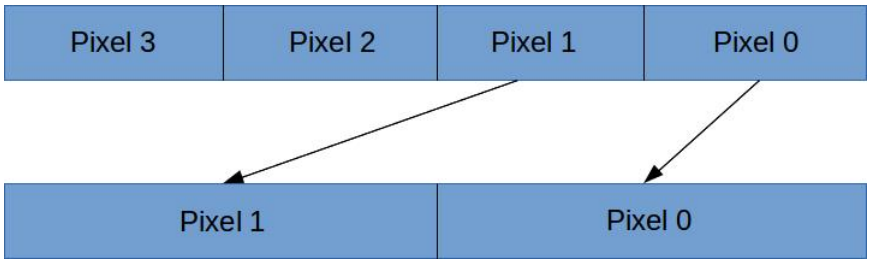


Figura 4: Aumento de precisión de la parte baja del registro XMM

Ahora trabajo con uno de estos registros, XMM1, que contiene 2 píxeles. El procedimiento para el XMM0 será análogo.

Copio el contenido de XMM1 en un registro XMM2 y shifteo 16 bits a la derecha (una WORD = un canal) el contenido de cada QWORD (píxel) de XMM2, lo cual me deja en las posiciones menos significativas de XMM1 y XMM2 los canales Azul y Verde respectivamente, como se puede apreciar en la Figura 5.

Realizo una suma del paquete WORD a WORD y obtengo la suma de los canales Azul y Verde en la WORD menos significativa de XMM1. Vuelvo a shiftear 16 bits y a sumar con XMM2, lo cual me termina dejando la suma de los 3 canales en la WORD menos significativa y, como se puede ver también en la Figura 5, la WORD mas significativa es el canal Alpha del píxel original (se le suma 0 cada paso).

Usando un shuffle, distribuyo la suma (la WORD menos significativa) a los 3 canales que corresponden a R, G y B.

Lo último que falta para este par de píxeles es realizar la multiplicación. Para esto, volvemos a aumentar la precisión de los píxeles a Double QWORD (Cada canal es un DWORD). De esta manera, podemos

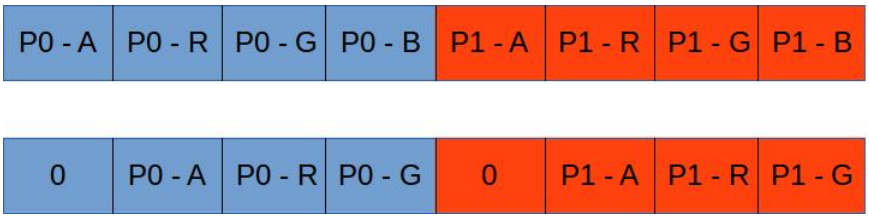


Figura 5: Ejemplo de como se utiliza el shift right logical QWORD para alinear los canales

convertir a cada canal en un Float, realizar la operación (multiplicar por los valores fijos 1.0, 0.5, 0.3 y 0.2 a los canales A, R, G y B respectivamente), y volver a convertirlos a enteros.

Esta conversión de entero a float y viceversa es costosa y conlleva a cierta perdida de precisión. Sin embargo, mediante experimentación pudimos ver que la diferencia es de menos de 1 para cada canal cuando se compara con la implementación de C.

Lo único que queda por hacer en este punto es procesar el otro par de píxeles, y volver a empaquetarlos a su precisión original, utilizando saturación para valores que se excedan de 255.

2.3. Low Dynamic Range

Implementación en C

La implementación de C, al igual que la del filtro Sepia, recorre la imagen a lo ancho y a lo alto y procesa cada píxel de manera individual.

Los píxeles que forman parte de las dos primeras o dos ultimas filas/columnas se consideran casos borde ya que no tengo los vecinos requeridos por la formula del filtro. Estos píxeles se copian del source al destino sin cambiar su valor.

Para todos los demas píxeles, se realiza un ciclo interno que recorre un cuadrado de 5x5 alrededor del píxel actual y suma los valores de sus canales RGB en Suma_i, j, donde i y j identifican al píxel actual. Esta suma es luego utilizada para modificar el valor de cada canal de acuerdo a la formula del filtro:

$$ldr^k_{i,j} = I^k_{i,j} + \alpha.\frac{Suma_{i,j}}{max}.I^k_{i,j} \tag{1}$$

Donde max es un valor fijo equivalente al maximo valor de $\alpha.Suma_{i,j}$ La división debe hacerse al final en el sumando derecho a fin de perder la menor precisión posible. Teniendo esto en cuenta, nuestra cuenta toma la siguiente forma.

$$ldr^k_{i,j} = I^k_{i,j} + \frac{\alpha.Suma_{i,j}.I^k_{i,j}}{max} \tag{2}$$

El codigo dentro de cada ciclo de la implementacion en C es el siguiente

```
//Codigo de cada ciclo:
bgra_t *p_d = (bgra_t*) &dst_matrix[i][j * 4];
bgra_t *p_s = (bgra_t*) &src_matrix[i][j * 4];

//Si es un caso borde se mantiene el color del source
if (i < 2 || i >= (filas - 2) || j < 2 || j >= (cols - 2)){
    p_d->r = p_s->r;
    p_d->g = p_s->g;
    p_d->b = p_s->b;
    p_d->a = p_s->a;
}
else {
    //Si no es un caso borde:
    suma = 0;
    for (int i1 = i - 2; i1 <= i + 2; i1++){
```

```
                for (int j1 = j - 2; j1 <= j + 2; j1++){
                    p_temp = (bgra_t*) &src_matrix[i1][j1 * 4];
                    suma = suma + p_temp->r + p_temp->g + p_temp->b;
                }
            }

            ldr = p_s->r + ((alpha * suma * p_s->r) / max);
            p_d->r = (ldr < 0) ? 0 : ((ldr > 255) ? 255 : ldr);

            ldr = p_s->g + ((alpha * suma * p_s->g) / max);
            p_d->g = (ldr < 0) ? 0 : ((ldr > 255) ? 255 : ldr);

            ldr = p_s->b + ((alpha * suma * p_s->b) / max);
            p_d->b = (ldr < 0) ? 0 : ((ldr > 255) ? 255 : ldr);

            p_d->a = p_s->a;
        }
```

Implementación En ASM-SIMD

Para decidir de que manera afrontar la implementación del filtro LDR en ASM, lo primero que tuvimos que hacer fue analizar la manera en la que podemos traernos los vecinos de un píxel. Como necesitamos 5 píxeles por fila, no nos alcanza un registro XMM para almacenar esa informacion en el procesador. Solo podemos guardar 4 píxeles por registro (4 píxeles * 4 canal/píxel * 8 bytes/canal = 128 bits).

La solución es acceder dos veces a memoria por cada fila de vecinos requeridos. Sin embargo, esto nos trae otro problema, que es que el segundo acceso a memoria nos trae 4 píxeles, de los cuales 3 no nos son de ninguna utilidad para procesar el píxel sobre el que estamos parado.

A continuación explicamos como se trabaja con este píxel descartando esta información adicional traída de memoria. Luego explicaremos como modificamos el procedimiento para, aprovechando todos los píxeles que se obtienen al acceder a la memoria en busca de vecinos, procesar 4 píxeles en vez de 1.

Teniendo en mente que en algún momento tendremos que dividir por max, evaluamos nuestras opciones, que eran, o bien transformar los valores a float, o bien realizar la división mediante multiplicación de enteros. La idea del segundo método es la siguiente.

Tenemos que realizar el siguiente calculo

$$\frac{X}{max} \tag{3}$$

Pero esto es lo mismo que hacer

$$\frac{X(\frac{2^{32}}{max})}{2^{32}} \tag{4}$$

Ahora bien, dividir por 2^{32} es sencillo. Basta con descartar los 32 bits menos significativos o hacer un shift right de 32 bits. Esto significa que

$$\frac{X}{max} = (X\frac{2^{32}}{max}) >> 32 \tag{5}$$

Lo único que necesitamos saber es cuanto es $\frac{2^{32}}{max}$

$$\frac{2^{32}}{max} \approx 880 \tag{6}$$

Entonces nos queda que

$$\frac{X}{max} = (880 * X) >> 32 \tag{7}$$

Como multiplicar por un entero y hacer shifts (en realidad descartar la parte baja de la multiplicación) son operaciones menos costosas que la conversión de entero a float, y 880 es un número pequeño (Entra en 2 bytes), decidimos usar este metodo.

Volvemos al procesamiento de un píxel.

Lo primero que hacemos es realizar la suma de los vecinos. Para eso utilizamos un ciclo que, fila por fila y empezando desde la primera fila de los vecinos (esquina inferior izquierda en figura 6), trae los 5 píxeles necesarios, y realiza la suma parcial. Luego acumula el resultado en un registro acumulador. En la figura 6 podemos apreciar que el segundo registro XMM tendrá información no relevante para el caso que estamos procesando. Sin embargo no podemos realizar la lectura de ninguna otra forma, ya que los $P4$ de cada fila están separados en memoria por src_row_size bytes.

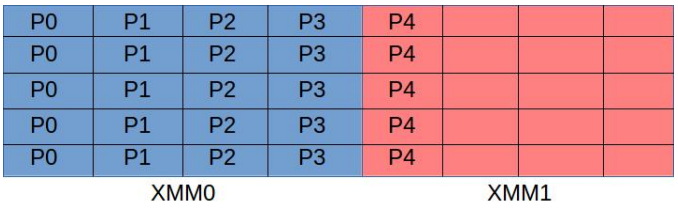


Figura 6: Esta imagen muestra la distribucion de los píxeles necesarios en memoria

En el ciclo que calcula la sumatoria ponemos un salto condicional para, cuando nos encontramos en la fila 2 (contando desde 0), traernos el píxel sobre el que queriamos procesar originalmente. Traemos este píxel al procesador durante el ciclo, y no antes, para ahorrar en accesos a memoria.

```
    cmp R13, 2
    jne .ciclo_vecinos_continuar
        pblendw XMM10, XMM0, 11110000b ; XMM10 = [ p1 | p0 | .  | .  ]
        pblendw XMM10, XMM1, 00001111b ; XMM10 = [ p1 | p0 | p3 | p2 ]
        pshufd XMM10, XMM10, 01001110b ; XMM10 = [ p3 | p2 | p1 | p0 ]
    .ciclo_vecinos_continuar:
```

Codigo para copiar la seccion central compuesta por los 2 píxeles mas significativos de XMM0 y los 2 menos significativos de XMM1.

Luego de finalizado el ciclo obtenemos en la WORD menos significativa de nuestro acumulador la suma de todos los valores RGB de todos los vecinos.

Realizamos la multiplicacion de esta suma por el valor alpha, ambos en precisión WORD, como lo muestra la figura 7.

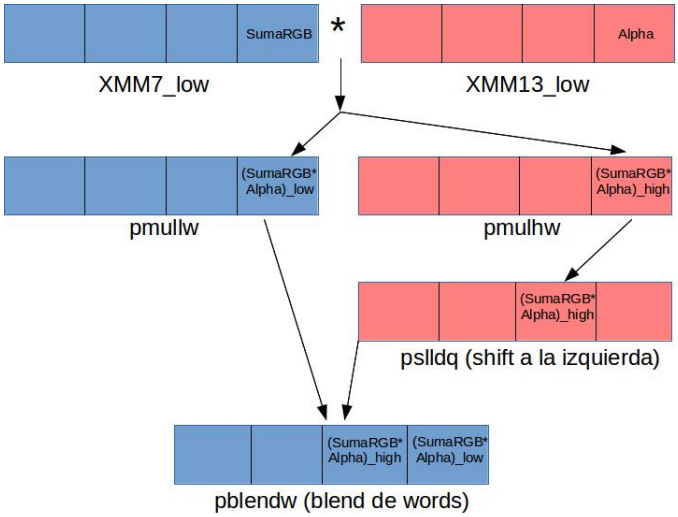


Figura 7: Esta imagen muestra la multiplicacion de SumaRGB por Alpha. Notese que solo representamos la parte baja de los registros XMM

El siguiente paso es traerme el píxel que voy a modificar a un registro XMM desde XMM10, y aumentar su precisión a DQWORD (cada canal a DWORD). Una vez tengo ese píxel, realizo la parte de la formula equivalente a:

$$Alpha \; . \; SumaRGB \; . \; I^k_{i,j} \tag{8}$$

Para eso, distribuyo mi resultado de $Alpha \; . \; SumaRGB$ a todas las DWORDS de un registro XMM, y hago una multiplicacion empaquetada DWORD a DWORD, quedandome solo con la parte baja.

Observacion: El resultado maximo de esta operacion es

$$5 * 5 * 255 * 3 * 255 * 255 = 1243603125 < 4294967296 = 2^{32} \quad (9)$$

Por lo tanto, me quedo solo con la parte baja del resultado de la multiplicacion, ya que la parte alta sera 0.

Lo ultimo que hacer es dividir por max. Como ya habiamos mencionado, vamos a hacer esto mediante multiplicacion de enteros. Entonces hago una multiplicacion empaquetada a cada canal por 880. El resultado que quiero esta en la parte alta del producto, por lo que realizo una serie de shifts y shuffles para reacomodar en un solo registro el resultado del sumando derecho de la ecuacion del filtro LDR.

Una vez obtenido este resultado, es solo cuestion de llevar ambos registros XMM (el del píxel original y el del sumando derecho de la ecuacion) a la misma precision, y realizar una suma empaquetada canal a canal. Por ultimo volvemos a empaquetar el píxel, usamos un blend para colocarlo en el lugar correspondiente en el registro XMM10 (el registro que tiene el píxel que estoy procesando), y cargar este registro a memoria en la imagen destino.

LDR - ASM: Pasando a procesar de a 4 píxeles

Un problema que trae la manera anteriormente descripta de aplicar el filtro LDR es que cuando accedemos a memoria para traer los vecinos del píxel a procesar estamos trayendo 20 píxeles que no necesitamos. Sin embargo, al traer 8 píxeles por fila, estamos trayendo no solo los vecinos de nuestro P0 (píxel a procesar), sino tambien los de P1, P2 y P3.

Viendo esto, decidimos que una mejor aproximacion seria utilizar toda la informacion que traemos de memoria por ciclo, y en el siguiente ciclo avanzar 4 píxeles en vez de 1.

Para lograr esta modificacion, agregamos una seria de ciclos con aplicaciones condicionales de mascaras que se encargan de, para cada porcion del codigo que se puede repetir para los 4 píxeles, adaptar el contenido de los registros para poder ser procesados.

Podemos ver un ejemplo de esto en el siguiente codigo, que es responsable de insertar el píxel una vez aplicado al filtro al registro XMM10 que contiene los píxeles originales.

```
;Muevo el pixel de nuevo a XMM10
cmp R14, 0
je .ciclo_ldrizacion_pixelxpixel_p0_return
cmp R14, 1
je .ciclo_ldrizacion_pixelxpixel_p1_return
cmp R14, 2
je .ciclo_ldrizacion_pixelxpixel_p2_return
cmp R14, 3
je .ciclo_ldrizacion_pixelxpixel_p3_return

.ciclo_ldrizacion_pixelxpixel_p0_return:
pblendw XMM10, XMM11, 00000011b ; XMM10 = [ p3 | p2 | p1 | p0_ldr ]
jmp .ciclo_ldrizacion_pixelxpixel_p_return
.ciclo_ldrizacion_pixelxpixel_p1_return:
pshufd XMM11, XMM11, 11110011b   ; XMM11 = [ 0  | 0  | p1_ldr | 0  ]
pblendw XMM10, XMM11, 00001100b ; XMM10 = [ p3 | p2 | p1_ldr | p0 ]
jmp .ciclo_ldrizacion_pixelxpixel_p_return
.ciclo_ldrizacion_pixelxpixel_p2_return:
pshufd XMM11, XMM11, 11001111b   ; XMM11 = [ 0  | p2_ldr | 0  | 0  ]
pblendw XMM10, XMM11, 00110000b ; XMM10 = [ p3 | p2_ldr | p1 | p0 ]
jmp .ciclo_ldrizacion_pixelxpixel_p_return
.ciclo_ldrizacion_pixelxpixel_p3_return:
pshufd XMM11, XMM11, 00111111b   ; XMM11 = [ p3_ldr | 0  | 0  | 0  ]
pblendw XMM10, XMM11, 11000000b ; XMM10 = [ p3_ldr | p2 | p1 | p0 ]
jmp .ciclo_ldrizacion_pixelxpixel_p_return
```

3. Resultados

3.1. Introduccion

Con el objetivo de medir la performance de los filtros en sus diferentes implementaciones, utilizamos como métrica la cantidad de ciclos de reloj insumidos en la ejecución de los mismos. Los ciclos fueron medidos por la función proporcionada por la cátedra, $MEDIR\_TIEMPO()$.

Todas las experimentaciones se corren sobre el mismo procesador, un Intel i7 de cuarta generación.

Decidimos ejecutar los experimentos deshabilitando las tecnologías SpeedStep (EIST) y TurboBoost de Intel, ya que suponemos que pueden llegar a generar inconsistencias en los valores obtenidos. EIST permite al procesador variar su frecuencia dínamicamente, y TurboBoost incrementa, de forma automática, la velocidad de procesamiento de los núcleos por encima de la frecuencia operativa nominal.

Utilizamos un script de python para ejecutar, primero en su version en C, y luego en ASM,cada filtro por separado, y con una serie de imágenes de tamanõs diferentes.

Para cada experimento se realizan 50000 iteraciones, con el objetivo de minimizar el impacto de los outliers, es decir, aquellas ejecuciones que ven un fuerte incremento de ciclos gastados debido a que fueron interrumpidas por el scheduler del sistema operativo.

Las figuras 10, 13 y 16 muestran gráficos preliminares que realizamos, pero descartamos al ver que no era la mejor manera de representar la diferencia de performance. Estos muestran la diferencia entre el tiempo total insumido a lo largo del experimento por cada filtro en cada implementación. Sin embargo, esto no tiene en cuenta que una diferencia de $10\,\%$ entre las implementaciones sobre una imagen de 16x16 puede encontrarse en los miles de ciclos, cuando la misma diferencia porcentual sobre el procesamiento de una imagen de 512x512 puede estar en el rango de los millones de ciclos, quitando relevancia a las diferencias en imágenes mas pequeñas.

Para solucionar este problema, hicimos los graficos de las figuras 9, 12 y 15, que toman la cantidad de ciclos insumidos de cada implementación como un porcentaje de los ciclos gastados por la versión de C sin optimizar para cada experimento, y luego promedian estos porcentajes.

3.2. Gráficos

3.2.1. Cropflip

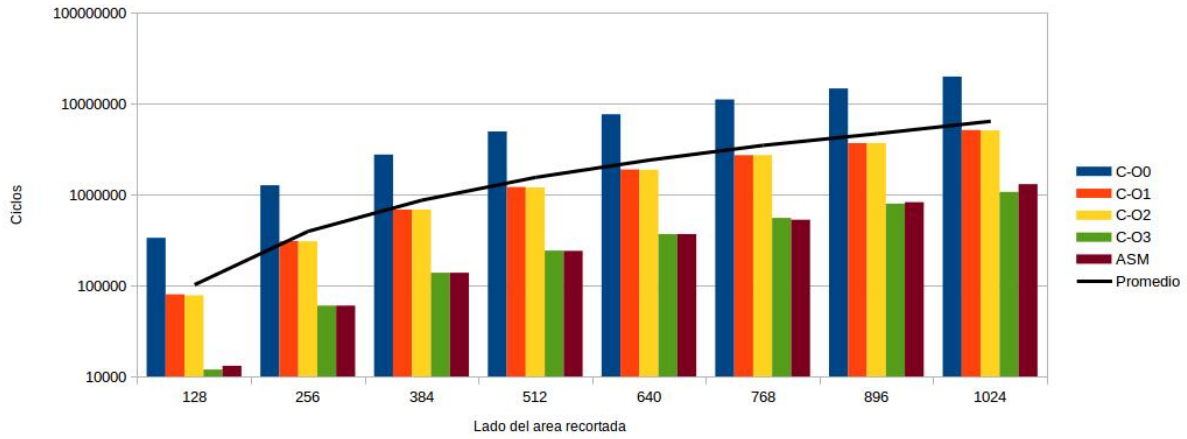


Figura 8: Todas las implementaciones de CropFlip al aumentar el tamaño de la zona a cortar sobre una imágen de 1024x1024

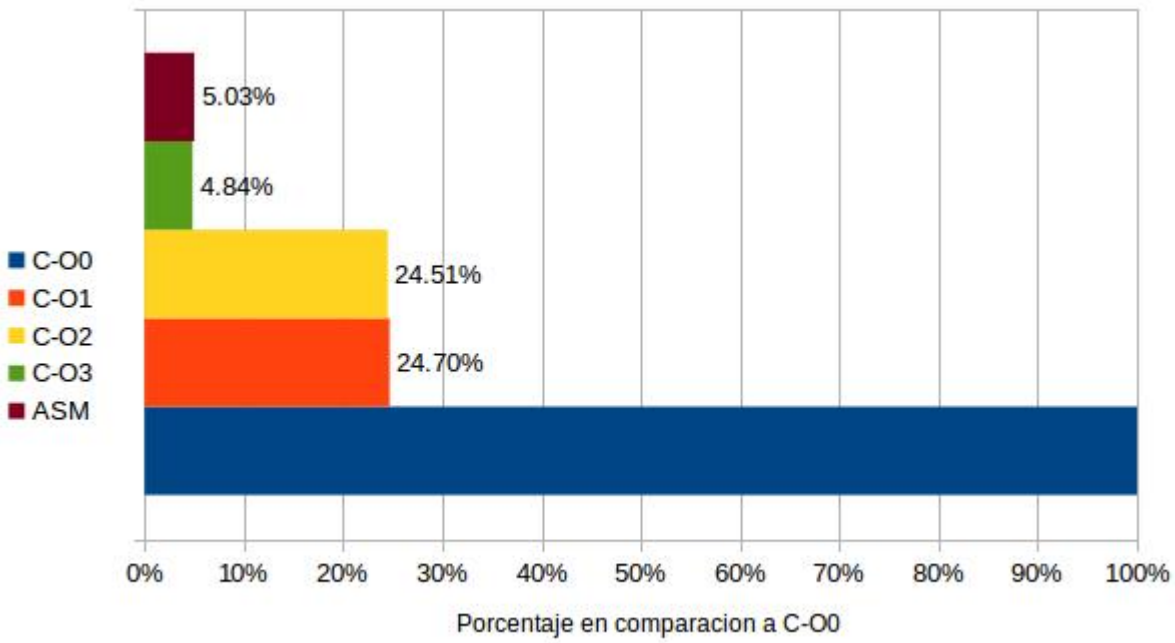
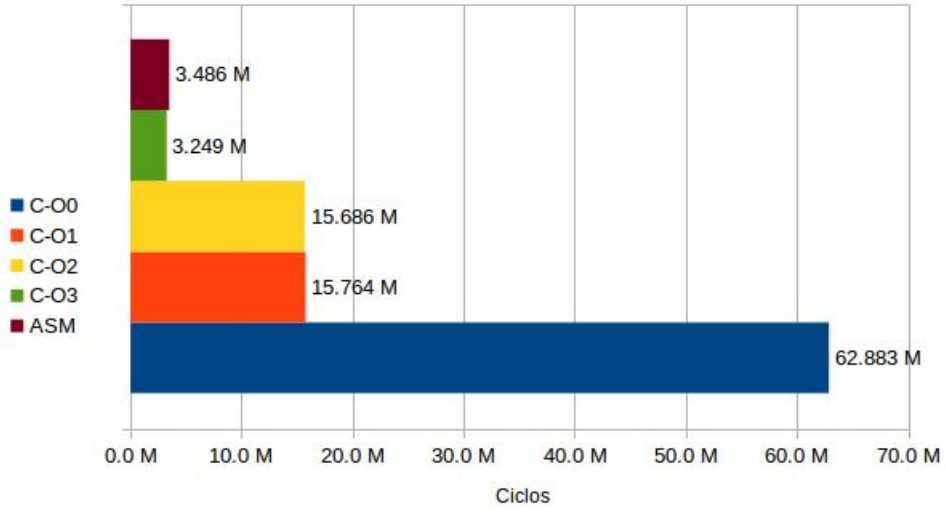
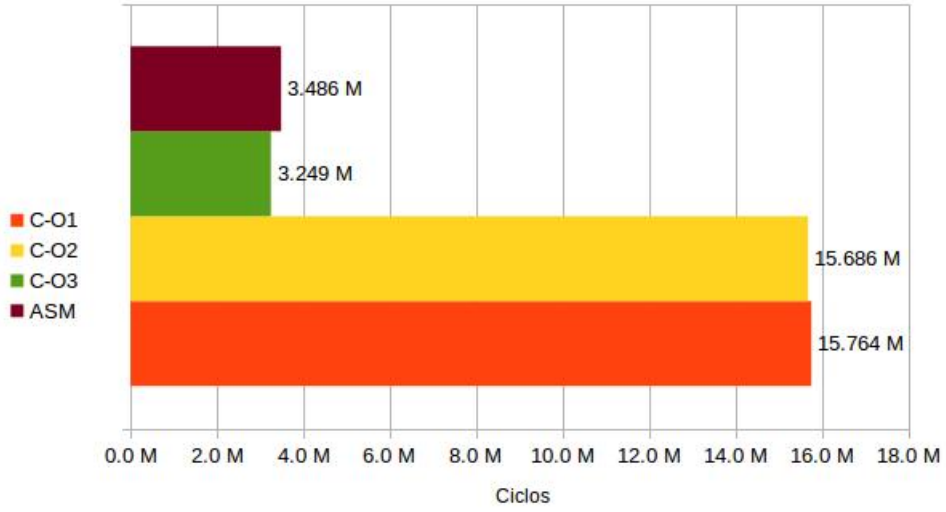


Figura 9: Promedio de porcentaje de ciclos gastados procesando en comparación a la implementación en C sin optimizar a lo largo del experimento de la figura 8.



(a) Todas las implementaciones.



(b) Detalle de versiones optimizadas y ASM.

Figura 10: Comparación de ciclos totales gastados procesando una imagen de 1024x1024, recortando secciones cuadradas de 128x128 a 1024x1024 en incrementos de 128 pixeles.

3.2.2. Sepia

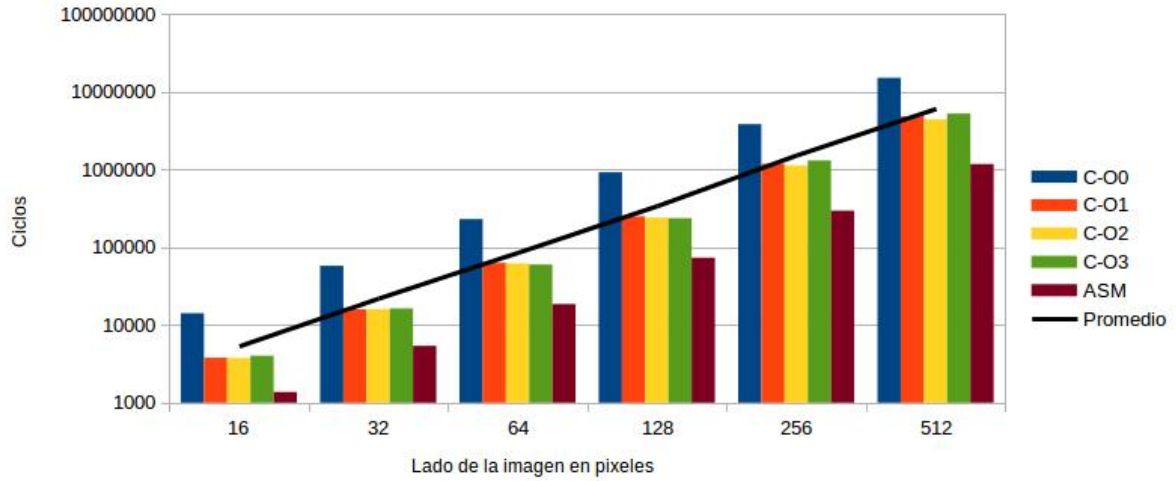


Figura 11: Todas las implementaciones de Sepia al aumentar el tamaño de la zona a cortar

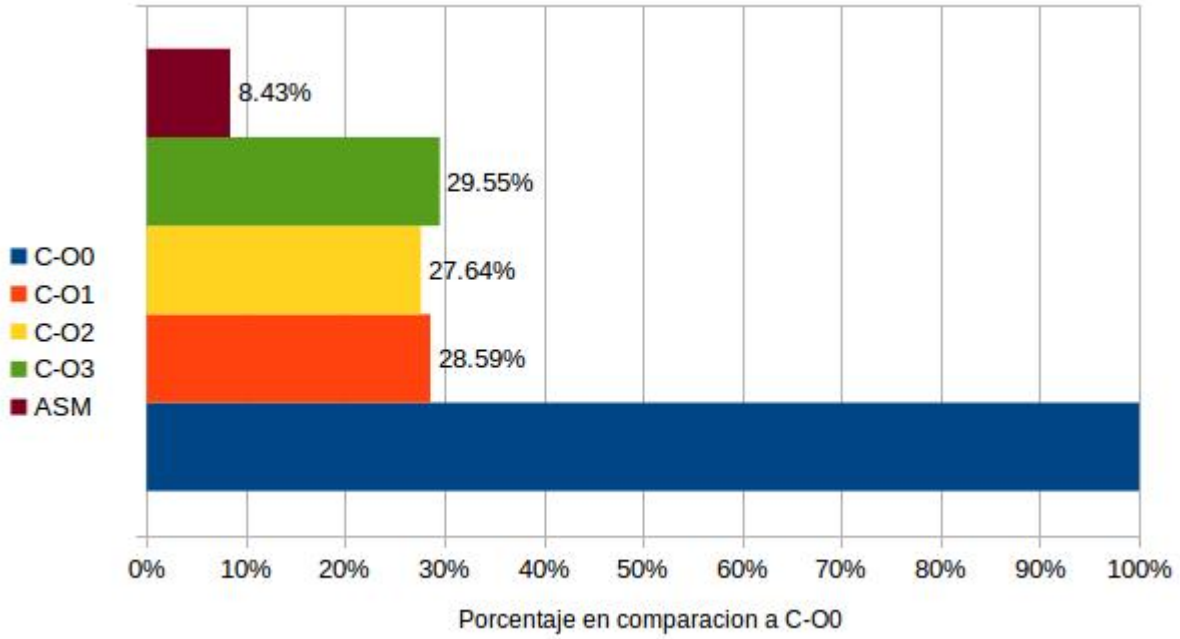


Figura 12: Promedio de porcentaje de ciclos gastados procesando en comparación a la implementación en C sin optimizar a lo largo del experimento de la figura 11.

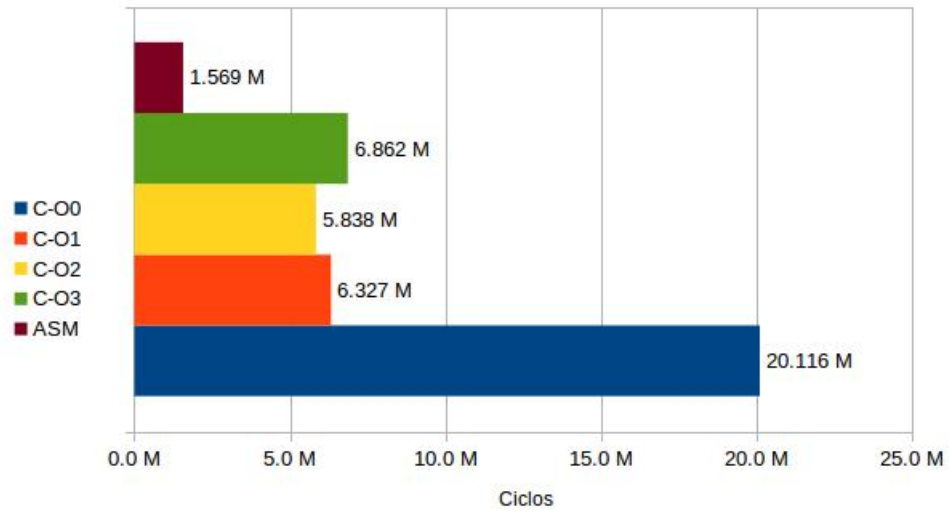


Figura 13: Comparación de ciclos totales gastados procesando imagenes cuadradas de 16x16 a 512x512, solo contando potencias de dos.

3.2.3. LDR

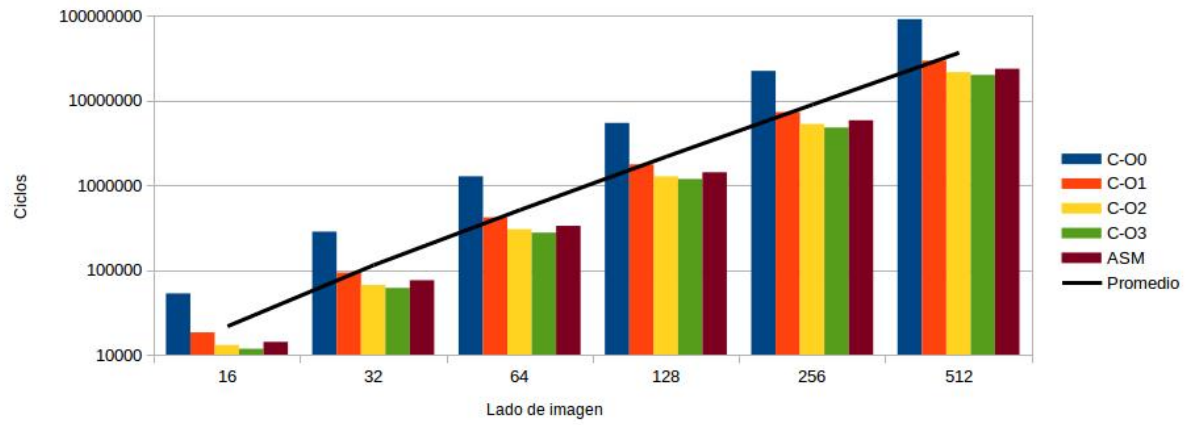


Figura 14: Todas las implementaciones de LDR al aumentar el tamaño de la zona a cortar

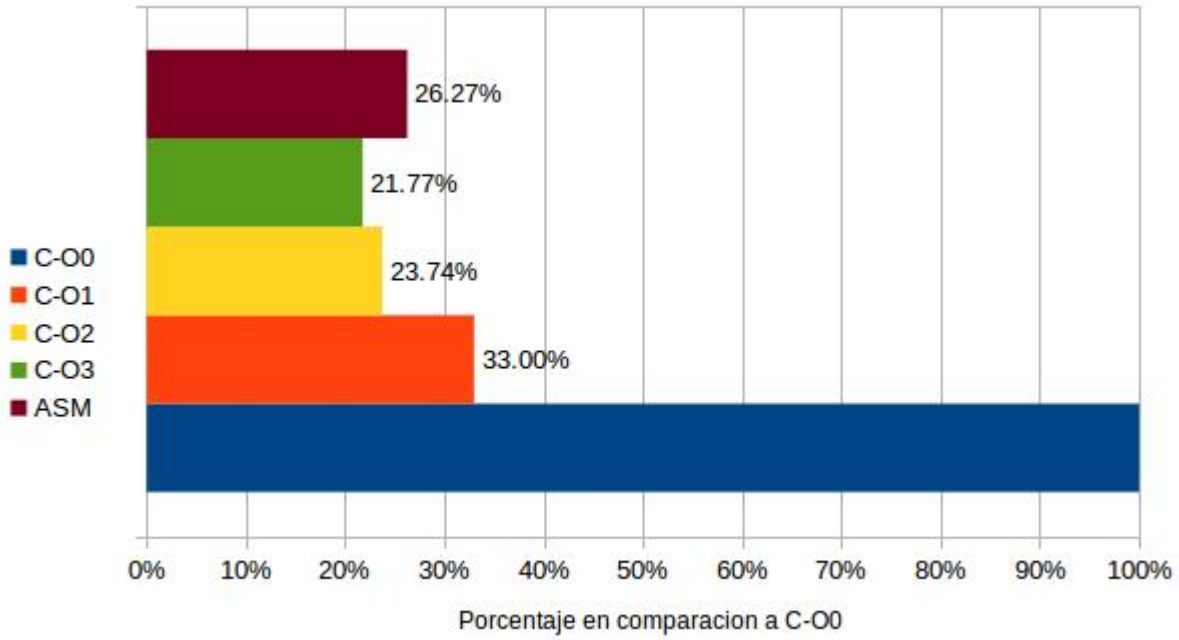


Figura 15: Promedio de porcentaje de ciclos gastados procesando en comparación a la implementación en C sin optimizar a lo largo del experimento de la figura 14.

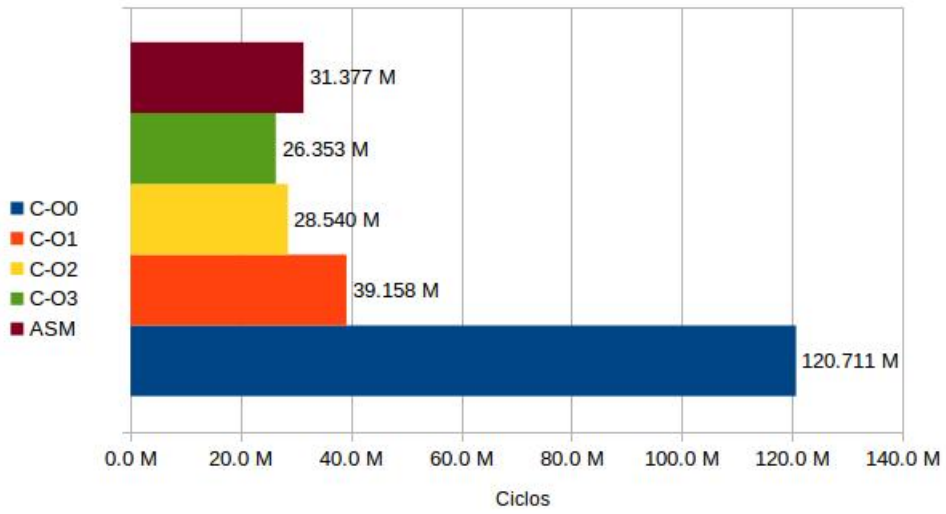


Figura 16: Comparación de ciclos totales gastados procesando imagenes cuadradas de 16x16 a 512x512, solo contando potencias de dos.

4. Discusión

Como podemos ver en las figuras 8, 11 y 14, es claro que, en general, la implementación de los filtros en ASM fue ampliamente superior a la implementación en C no optimizada. Sin embargo, los resultados varían en cuanto a la diferencia en performance, y la relación con diferentes niveles de optimización, filtro por filtro. Por este motivo vamos a analizar los resultados para cada filtro por separado.

4.1. Cropflip

La implementación del filtro cropflip en ASM fue la que tuvo la mayor diferencia de performance con respecto a su contra-parte en C sin optimizar. En promedio, a ASM le llevaba un $5{,}03\,\%$ del tiempo que a C realizar la misma operación. Este porcentaje varia entre un $4{,}73$ y un $5{,}62\,\%$ al cambiar el tamaño del area recortada.

La diferencia de performance entre la versión sin optimizar de C, y la versión de ASM se debe, en nuestra opinión, a que la implementación de C recorre los píxeles de manera individual, mientras que nuestra versión de ASM lo hace de a cuatro usando SSE. Sin embargo la mejora de velocidad de ASM es de más de cuatro veces. En nuestra opinión esto se debe a que la versión no optimizada de C puede estar realizando operaciones estándar que carecen de demasiado sentido en el filtro, generando un importante overhead.

Si miramos las versiones optimizadas de Cropflip, podemos apreciar que, aun utilizando el nivel O1 de optimización, vemos una mejora de 4 veces en la performance, un $24{,}70\,\%$ de ciclos gastados en comparación a C sin optimizar. Este valor sigue siendo aproximadamente 5 veces mas lento que la implementación en ASM, pero se acerca mucho mas a la relación que suponíamos se daría como resultado de procesar un píxel por ciclo, contra procesar cuatro.

La diferencia entre las optimizaciones O1 y O2 no es demasiada, como podemos apreciar en las figuras 8 y 9. En la figura 9 esta diferencia es de un $0{,}19\,\%$ de mejoría al ser comparados con la versión no optimizada de C, una diferencia demasiado pequenã como para ser considerada. Es posible que las versiones no difieran en mucho, y que esta diferencia se deba simplemente al error inherente a este tipo de mediciones.

La versión optimizada O3 tiene una performance similar (mejor incluso) que nuestra implementación en ASM. En los resultados individuales notamos que al recortar sectores de 16x16 y 1024x1024 (la totalidad de la imagen) fue donde O3 supero a ASM. En la primera, la diferencia fue de un $9{,}09\,\%$ y en la segunda de un $17{,}75\,\%$ a favor de O3. En los demás casos, la diferencia es tan pequeña que el promedio de las diferencias aproxima al $0{,}19\,\%$, por lo que suponemos que las implementaciones son similares. Sin embargo no sabemos con certeza porque en los casos mencionados anteriormente la disparidad de performance se vuelve tan severa.

4.2. Sepia

El análisis de las diferencias en performance para el filtro sepia es mas sencillo. La versión de ASM supera no solamente a la implementación en C sin optimizar, utilizando un $8{,}43\,\%$ de los ciclos en comparación, sino también a las versiones optimizadas con O1, O2 y O3. En este caso, ninguna de las optimizaciones se acerco al nivel de la implementación en ASM, que gastó, en promedio, un $29\,\%$ de ciclos procesando las imágenes en comparación a las versiones optimizadas. Nuevamente, estas diferencias se deben a que la implementación en ASM procesa cuatro píxeles por vez en vez de uno, haciendo en este caso operaciones matemáticas de a varios píxeles por vez (en algunos casos menos que cuatro simultáneamente).

Cuando analizamos las versiones optimizadas de C, vemos que no se cumple la misma tendencia que en el caso de Cropflip. La versión O1 mejora una proporción comparable en ambos filtros, y la O2 mantiene una performance similar. Sin embargo, en este caso aplicar la optimización O3 empeora la performance en $1\,\%$.

Los datos en los cuales esta basado el gráfico de la figura 11 nos muestran que, en todos los casos, las versiones optimizadas de C mejoran en cierta cantidad a la versión no optimizada, pero tienen una

performance algo peor al procesar las imagenes de 256x256 y 512x512 como podemos observar en el cuadro 1.

Lado Imagen	C-O0	C-O1	C-O2	C-O3	ASM
16	100.00 %	26.73 %	26.50 %	28.38 %	9.71 %
32	100.00 %	27.52 %	27.74 %	28.23 %	9.31 %
64	100.00 %	27.41 %	27.00 %	26.13 %	8.10 %
128	100.00 %	26.99 %	26.04 %	25.66 %	7.95 %
256	100.00 %	30.96 %	29.43 %	34.09 %	7.71 %
512	100.00 %	31.93 %	29.14 %	34.78 %	7.80 %

Cuadro 1: Performance en porcentaje de ciclos gastados en comparación a versión C-O0

Tomando como estándar la cantidad de ciclos gastados por la versión de C sin optimizar, podemos ver que tanto la versión O1 como la O2 sufren un retraso de entre $3\,\%$ y $4\,\%$ al procesar las imágenes de 256x256 y 512x512. La implementación de C con nivel de optimización O3, por otro lado, sufre un salto de alrededor de $9\,\%$ en los ciclos tardados. Esto provoca que, al tomar el promedio, la versión O3 tarde mas tiempo que las O1 y O2 en aplicar el filtro, a pesar de tener un rendimiento similar en imágenes mas pequeñas. No sabemos cual es el motivo para este cambio en performance, ya que la versión en ASM no lo sufre, y la versión no optimizada de C tiene un crecimiento lineal de ciclos gastados con respecto al tamaño de la imagen.

4.3. LDR

En el caso del filtro LDR, ASM vio la menor mejora de performance con respecto a la versión no optimizada de C. La implementación en ASM del filtro tardo el $26{,}27\,\%$ de los ciclos que su contra-parte en C. Esta mejora es debido a que, nuevamente, la versión ASM hace uso de instrucciones de SSE para procesar múltiples píxeles por vez. En este caso, el proceso se hace píxel por píxel, pero las mejoras se encuentran en la manera en la que se suman los vecinos y se realizan las cuentas pertinentes, además de que con un solo acceso a memoria se obtienen todos los píxeles relevantes para procesar cuatro píxeles por vez.

Una vez más, vemos un mejor desempenõ de la versión de C al ser optimizada. Como podemos ver en la figura 15, utilizar O1 como método de optimización provoca que se reduzca a la tercera parte la cantidad de ciclos insumidos. Esta reducción es menor a la de otros filtros al utilizar O1. Sin embargo, las versiones O2 y O3 mejoran aun mas la performance, superando ambas a nuestra implementación en ASM.

Utilizando los datos en los que se basa el gráfico de la figura 14, podemos ver que, en promedio, la optimización O2 es un $9{,}62\,\%$ más rápida con respecto a ASM, mientras que la O3 es $17{,}10\,\%$ más efectiva. Suponemos que esto se debe a que hay alguna manera de procesar los datos de forma mas eficiente de lo que logramos en nuestra implementación en ASM.

5. Conclusion

En general, obtuvimos un incremento en la performance bastante importante al comparar las implementaciones de ASM con las de C, de alrededor de 20 veces más rápido en Cropflip y Sepia, y 4 veces en LDR. A pesar de esto, la optimización de C hizo un muy buen trabajo, mejorando la velocidad de los filtros 4 veces como mínimo.

ASM es una buena opción para realizar este tipo de filtros, ya que vimos mejora de performance en todos los filtros. Sin embargo, a la hora de decidir por una implementación u otra, es importante tener en cuenta que las versiones de los filtros en C tuvieron un desarrollo más sencillo, y que las optimizaciones proporcionadas por el compilador GCC dieron resultados muy buenos, consiguiendo tiempos de ejecución cercanos a aquellos obtenidos con ASM.

5.1. Trabajos futuros

Hay ciertos resultados cuyo motivo no nos es del todo claro. Planteamos como objetivo para posibles futuras experimentaciones el ver que provoca que las implementaciones en ASM superen mucho más de cuatro veces a aquellas en C, como el usar SSE para procesar píxeles en paquetes de cuatro nos hizo pensar que sucedería.

Por otro lado, podríamos expandir nuestro entendimiento del porqué de las diferencias de mejora para las distintas optimizaciones de C haciendo una investigación sobre como se maneja esta funcionalidad del compilador GCC. Otra opción para solucionar este mismo problema sería el desensamblar los archivos generados por el compilador para las implementaciones en C, y compararlas al utilizar diferentes métodos de optimización.

Dejando de lado las diferencias de performance entre distintas implementaciones y distintos filtros, podríamos recurrir a otro tipo de experimentación, o bien utilizando otras métricas u otros métodos de medición, para inquirir en el motivo por el cual se producen picos de performance en imágenes que superan ciertos tamaños.

Otra opción seria ver como afectan las resoluciones de las imágenes a la performance cuando estas superan la capacidad del cache del procesador.