DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Lo bueno de las lentejas

Organización del Computador II

Primer Cuatrimestre de 2016

Integrante	LU	Correo electrónico
Figari Francisco	719/14	figafran@gmail.com
Mariotti Ignacio	651/14	mariotti.ignacio@gmail.com
Sabogal Patricio	693/14	pato.sabogal@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. Introducción

El objetivo de este trabajo es evidenciar las ventajas de ASM sobre C en cuanto a la optimización del tiempo de computo. Para ello realizaremos la implementación y experimentación de distintos tipos de filtros de imágenes[1]:

- **Cropflip** consiste en recortar de la imagen original y luego realizar un *flip*.
- **Sepia** modifica los canales de colores con el objetivo de dar importancia al canal *red*. De esta manera obtenemos una imagen rojiza.
- **Low dynamic range** busca aumentar la iluminación de aquellos píxeles que ya son más luminosos. Para ello cada píxel se procesa considerando sus vecinos.

Cada filtro tendrá por lo menos una implementación en C y otra en ASM. Una vez realizadas las implementaciones, realizaremos distintos experimentos para acercarnos a nuestro objetivo. Estos buscarán comparar los tiempos de cómputo e intentarán los resultados obtenidos.

(a) Imagen Original

(b) Cropflip

(c) Sepia

(d) LDR

[1] En la respectiva sección de desarrollo de cada filtro se detalla su funcionamiento.

2. Desarrollo

2.1. Cropflip

2.1.1. Descripición

El filtro cropflip busca achicar la imagen (*crop*) y espejarla respecto del eje x (*flip*). Abarcaremos sólo el caso donde el ancho y alto del *crop* son múltiplos de 16. Tanto para las implementaciones en ASM como para la implementación en C se utilizó el mismo algoritmo: recorrer las filas de la matriz de origen y copiarlas en el destino. Lo anterior se realiza sobre los píxeles que nos definan los valores *offsetx*, *offsety*, *tamx* y *tamy*.

Se hicieron cuatro implementaciones distintas de cropflip: una en C y tres en ASM. Dos de las implementaciones en ASM nacieron a lo largo de las experimentaciones, por lo que sus motivos se explican en la sección de experimentación. A continuación se explican las cuatro implementaciones (todas análogas debido la reducida complejidad del problema):

2.1.2. Cropflip en C

Se recorren individualmente los píxeles de la matriz de destino a ser copiados Se realiza una cuenta con los índices y los parámetros de entrada que nos indique el próximo píxel a copiar en la matriz de origen. El recorrido se hace utilizando un doble ciclo, recorriendo las columnas en el ciclo exterior (de abajo para arriba), y las filas en el ciclo interior (de izquierda a derecha).

2.1.3. Cropflips en ASM

En las tres implementaciones se utiliza la instrucción `movdqu` de SIMD para copiar de a cuatro píxeles. Se diferencia según la forma en la cual recorren la matriz que es nuestra imagen.

Filas decrecientes :

Se recorren las filas de la matriz de origen de abajo hacia arriba. Se copian las filas en la matriz de destino de arriba hacia abajo.

Filas crecientes:

Se recorren las filas de la matriz de origen de arriba hacía abajo. Se copian las filas en la matriz de destino de abajo para arriba. Respecto de la implementación anterior, `filas_crecientes` tiene la ventaja de recorrer la imagen de destino como un largo vector de píxeles.

Cachemiss :

En esta implementación decidimos ir copiando de a columnas (en lugar de a filas). Se leen y escriben columnas de ancho cuatro, lo cual no presenta problema pues las dimensiones de las imágenes son múltiplo de 4. Con está implementación se buscó que las escrituras en la matriz de destino no fueran continuas como sí pasa en el resto de las implementaciones.

2.2. Sepia

2.2.1. Descripción

Este filtro modifica los canales de colores, reemplazando cada color por las suma de los tres canales multiplicada por una constante. (0.5 para el rojo, 0.3 para el verde y 0.2 para el azul). El canal queda intacto. Los constantes que se eligen son aquellas que caracterizan al filtro sepia, pues permiten aumentar la relevancia del color rojo en la imagen.

2.2.2. Sepia en C

La implementación en C del filtro sepia es sencilla. Sin optimizaciones del compilador, se procesa de un píxel a la vez, y un canal a la vez. Recorremos la matriz por filas y columnas para acceder a cada píxel. Calculamos la suma de los tres canales y en tres variables de tipo double (una para cada canal del píxel) guardamos la suma por la constante del canal. Luego se saturan los canales y se asignan al píxel destino.

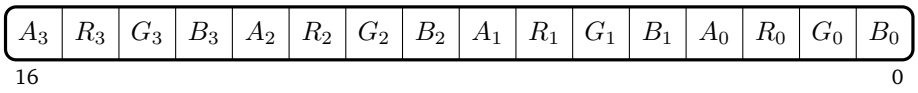
Una implementación alternativa es utilizar floats en vez de doubles al multiplicar por la constante. Se detallará más en los experimentos.

2.2.3. Sepia en ASM

Esta implementación resulta más compleja pero permite un mejor procesamiento. La idea es levantar cuatro píxeles por iteración de ciclo y procesarlos en paralelo, aprovechando lo más posible las instrucciones de SIMD. Al procesar de a cuatro píxeles se puede intuir una mejora por sobre la implementación en C. Por especificación del trabajo, las imágenes a filtrar van a tener una cantidad de píxeles múltiplo de 4. Si no lo fueran, la solución es calcular el resto modulo 4 y realizar un ciclo extra luego del principal para procesar los píxeles restantes.

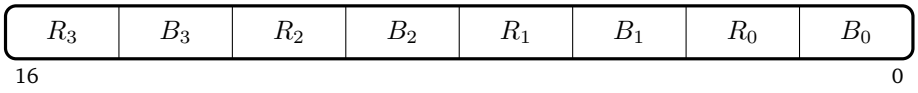
El ciclo principal itera $cantidadPixeles/4$ veces. Consta de dos partes principales: calcular la suma y multiplicar por la constante.

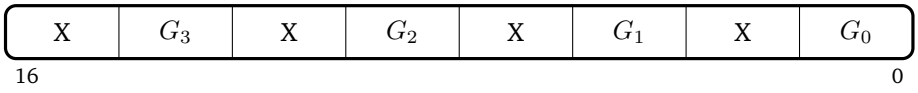
Cuando los datos se levantan de memoria, el registro donde se cargan queda partido de la siguiente manera:



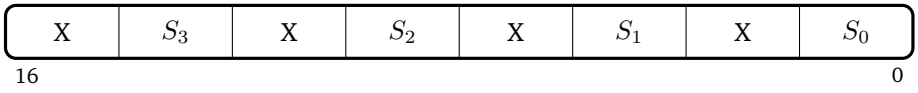
La suma de los tres canales de colores en peor caso es $255 * 3 = 765$, se necesitan al menos dos bytes para representar este número, por lo que no se puede resolver con sumas horizontales.

Se realiza una serie de desplazamientos de manera que los canales queden extendidos a 2 bytes, con B y R en el mismo registro.





Se hace una suma paralela, un desplazamiento del primer registro y se suma nuevamente.

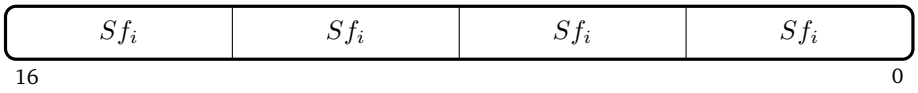


$S_i = R_i + B_i + G_i$

Para la multiplicación por las constantes se realizaron dos implementaciones alternativas:

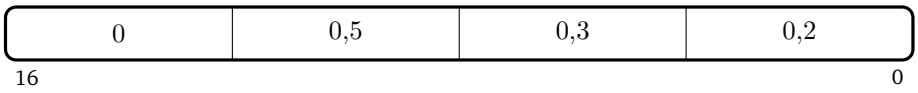
Multiplicación Float

Se extienden las sumas a 4 bytes y se convierten en floats. Luego se separan replican las sumas en cuatro registros distintos con `shufps` de modo que cada registro represente un píxel distinto.



Sf_i : Suma de los canales del pixel i convertida a floats

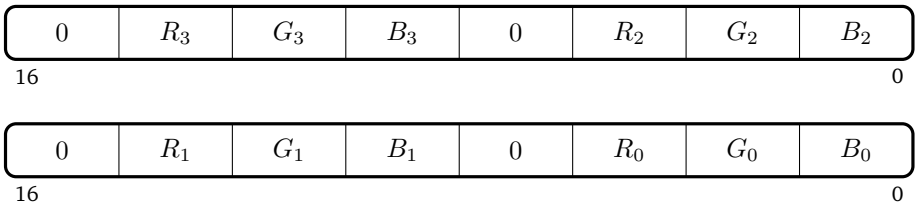
Se realiza una multiplicación paralela de cada registro con otro que tiene las constantes de sepia.



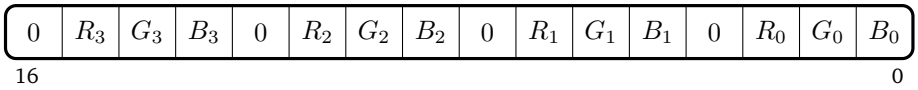
Notar que este último registro se carga previo al comienzo del ciclo para evitar accesos a memoria. También se multiplica por la constante 0 para facilitar el restablecimiento del canal A.

Se realiza la conversión a enteros.

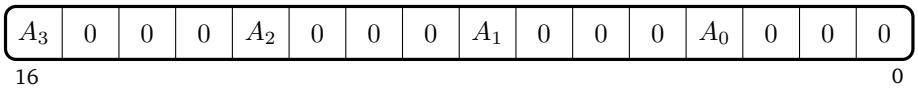
Para empaquetar se agrupan los pixels 3 y 2 en un registro y 1 y 0 en otro. Es importante el orden del empaquetado para no cambiar el orden de los pixels.



Se empaqueta nuevamente de modo que se restablezca la estructura original en memoria.



Utilizando una máscara se prepara el registro original para restablecer el canal A.



Se suman los registros y se guardan los cuatro píxeles en el destino y se continua iterando.

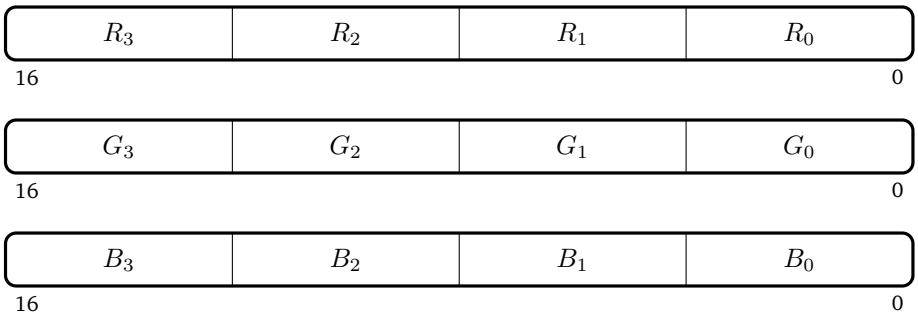
Hay otra manera de realizar la multiplicación por las constantes. En vez de separar las sumas, multiplicarlas directamente por 0.5 para obtener las componentes rojas de todos los píxeles, 0.3 para las verdes y 0.2 para las azules. De manera que queden en tres registros cada canal de los píxeles resultantes. Este método implica dos conversiones menos y una multiplicación menos pero el empaquetamiento no es sencillo, se puede ver con más detalle en la implementación del sepia con división entera.

División entera

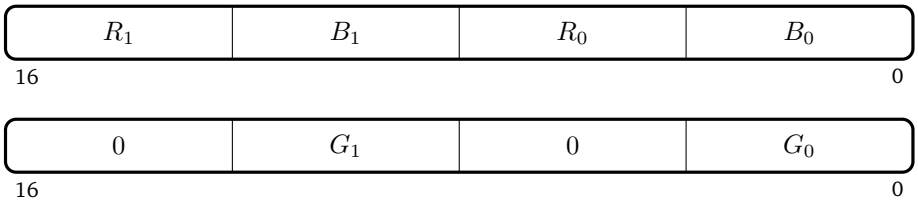
Esta implementación se obtuvo del texto *Optimizing Assembly*[1]. El principio básico de este algoritmo es que $\frac{a}{b} = a * \frac{2^n}{b} * \frac{1}{2^n}$. Entonces $\frac{a}{b} \approx a * \frac{2^n}{b} >> n$. El valor n óptimo es un valor que se puede calcular en tiempo de compilación. Este calculo no resulta un inconveniente para nuestro algoritmo pues multiplicamos solamente por tres valores diferentes que son $0{,}2$, $0{,}3$ y $0{,}5$ lo cual es igual a multiplicar por $\frac{1}{5}$, $\frac{3}{10}$ y $\frac{1}{2}$. En nuestro caso, sólo vamos a tener que aplica este algoritmo para multiplicar por $\frac{1}{5}$. $0{,}3$ es equivalente a $\frac{3}{10} = 3 * \frac{1}{2} * \frac{1}{5}$. Para calcular los tres valores necesito entonces multiplicar por 3, dividir por 2 (shifteando) o aplicar el algoritmo para multiplicar por $1/5$.

El valor óptimo de n para resolver $\frac{a}{b}$ es $n = c - 1 + w$, donde c es la cantidad de bits significativos del divisor b y w es el tamaño en bits del tipo int que estemos utilizando. Como queremos multiplicar por $1/5$, y el binario de 5 es $101b$, $c = 3$. Además nuestros enteros son de 16 bits, entonces $w = 16$. Obtenemos $n = 3 - 1 + 16 = 18$. Para multiplicar por $\frac{1}{5}$ tenemos que multiplicar entonces por $\frac{2^{18}}{5}$ y luego desplazar 16 (o equivalentemente $18 - 16 = 2$) hacía la derecha.

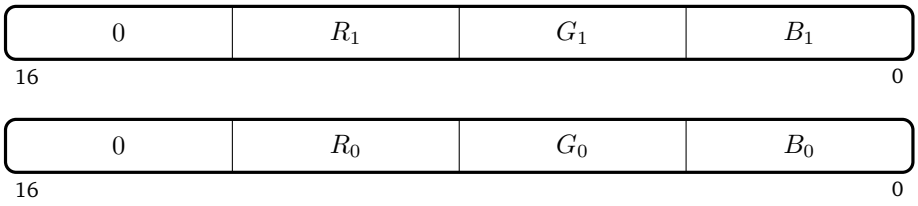
Se realiza una copia por cada canal de color de las sumas. De manera que los canales queden separados en registros distintos y se procesan por separado.



Solo queda empaquetar y guardar en memoria. Primero se trabaja con la parte baja de los registros, se desempaquetan los canales rojo junto con el azul y verde junto con un registro vacío.



Luego mediante un desempaquetado de parte baja y alta se separan los pixels 1 y 0.



Se realizan los mismos pasos para los pixels 3 y 2. Una vez separado cada pixel en un registro distinto se empaqueta de manera similar a la implementación alternativa, se restablece el canal A y se guarda en memoria.

2.3. Low Dynamic Range

2.3.1. Descripción

Este filtro busca modificar cada canal incrementando o decrementando su intensidad en base a los píxeles vecinos. Para cada píxel:

- Se calcula la suma de todos los canales salvo el *alpha* de sus vecinos tomando un cuadrado de 5x5.
- Se la multiplica por un $\alpha \in \{-255, .., 255\}$.
- Se la divide por el máximo valor posible para normalizar $(5 * 5 * 255 * 3 * 255)$.
- Finalmente, a cada canal se le suma el resultado anterior multiplicado por el mismo canal.

No se procesan los píxeles con coordenadas $(x, y) : x \leq 2 \vee y \leq 2 \vee x > cols - 2 \vee y > filas - 2$ dado que no tienen suficientes vecinos.

2.3.2. Low Dynamic Range en C

Si bien LDR requiere bastantes cuentas para procesar un píxel, el código en C sigue siendo bastante sencillo. Se recorre la matriz y se procesan los píxeles uno a uno. Solo son procesados los píxeles con suficientes vecinos. Dado que la imágen destino recibida por parámetro se inicializa con una copia del origen, se recorre desde la segunda fila y segunda columna hasta la cantidad de filas y columnas - 2 respectivamente. Se calcula la suma de los vecinos para un píxel y se multiplica contra α y contra los tres canales

```
int r = suma*alpha*p_s->r
int g = suma*alpha*p_s->g
int b = suma*alpha*p_s->b
```

Se castea a double para dividir por max (máximo posible valor), para forzar al compilador a realizar las operaciones con doubles. Luego se suma al canal original.

```
r = p_s->r + ((double) r)/max;
g = p_s->g + ((double) g)/max;
b = p_s->b + ((double) b)/max;
```

Si el numero es negativo pasa a ser 0 y si es mayor a 255 se satura. Como en sepia, se pueden intercambiar los tipos doubles por floats.

2.3.3. Low Dynamic Range en ASM

La función LDR en *assembler* esta dividida en dos partes principales: El cálculo de la suma y el procesamiento en base a la suma de cada canal. El ciclo principal itera sobre el rectángulo interior de la imagen. Empezando desde la fila 2 hasta la fila $filas - 2$, y columna 2 hasta columna $cols - 2$. Cada iteración procesa cuatro píxeles.

Suma de vecinos

Se realizaron dos implementaciones distintas para la suma de vecinos de LDR, una “naive” y otra “optimizada”.

Suma Optimizada: El calculo de vecinos se resuelve utilizando la suma de columnas de la siguiente manera.

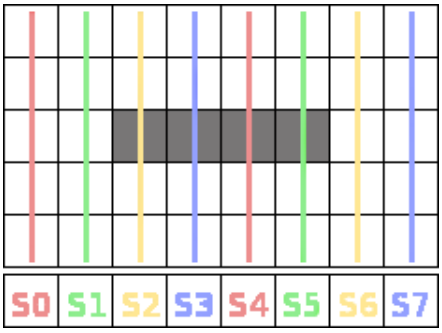


Figura 1: Suma de columnas. Cada cuadrado representa un píxel, los grises son los que se procesan en un determinado ciclo de iteración.

El ciclo que acumula la sumatoria comienza levantando en dos registros un píxel de cada columna, de la 0 a la 3 para uno, y de la 4 a la 7 para el otro. Se calcula la suma de los componentes de cada píxel de la misma manera que se realiza en el filtro sepia. La suma total de las columnas como máximo puede ser $255 * 3 * 25 = 19125 =$ 0x4AB5, que se puede representar con 4 bytes, por lo que se puede acumular cada suma de columna en 4 bytes. Este ciclo requiere un promedio de 2.5 accesos a memoria por píxel procesado.

Una vez calculada la suma de todas las columnas en dos registros como indica la imagen, lo que queda es, para cada píxel a procesar, calcular su propia suma. La idea es alinear correctamente las columnas para que por cada píxel se sume lo correspondiente.

```
    | 00 | S3 | S2 | S1 | -->
       +    +    +    +
    | S3 | S2 | S1 | S0 |
       +    +    +    +
<-- | S7 | S6 | S5 | S4 |
```

SX indica la suma de la columna X, las flechas indican desplazamientos. Se acumula el total para cada pixel en el registro intermedio.

Suma Naive : La suma *Naive* consta de, por cada píxel a procesar, realizar diez accesos a memoria para levantar sus vecinos y calcular la suma. Para hacer esto creamos la función *calcularSuma*, que dada una columna calcula a partir de la misma la sumatoria de los canales en un ancho de 4 píxeles.

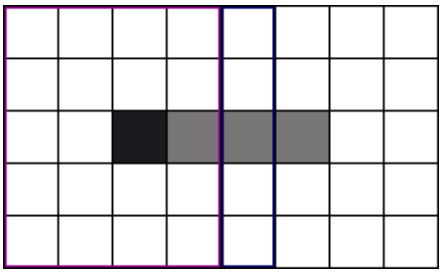
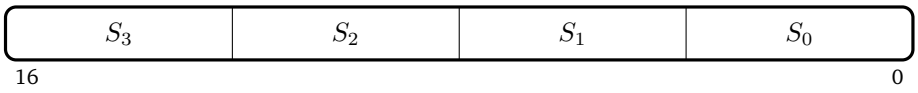


Figura 2: Suma de vecinos. El píxel pintado de negro es el que se va a procesar, los píxeles con borde violeta y azul son los vecinos.

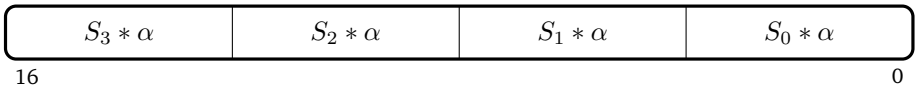
El violeta indica lo que se aprovecha con la primer llamada a *calcularSuma*, el azul la segunda llamada. Hay un desperdicio de tres columnas al realizar la segunda llamada. El resultado de la primer llamada se procesa mediante sumas horizontales. La suma de los vecinos se guarda en un registro resultado y se desplaza para trabajar el siguiente píxel. Una vez que se realizó el calculo de los cuatro píxeles (8 llamadas a *calcularSuma* en total), se procesan los canales de color.

Procesamiento de color

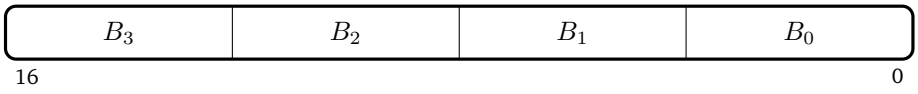
Se tiene un registro con la suma de los vecinos de los cuatro píxeles a procesar en el ciclo.



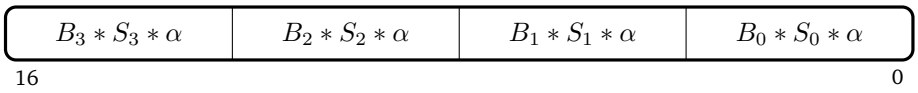
Se realiza el producto entero contra el parámetro α del filtro. No hay pérdida de precisión. Es importante destacar que si bien este producto se puede representar en 4 bytes dado que α se encuentra acotado por -255 y 255, las instrucciones de *assembly* requieren realizar el producto en dos partes.



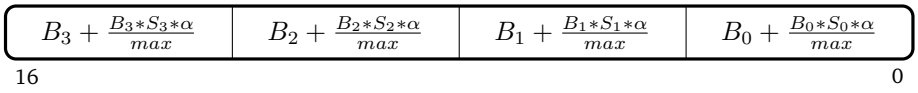
Este registro se copia en otros dos, cada uno representa un canal de color del resultado final. El procesamiento de los distintos canales es similar. Se prepara un registro con el componente a calcular representado en 4 bytes.



Se realiza el producto entera con las sumas calculadas anteriormente, teniendo las mismas consideraciones en cuanto a la representación del mismo que con la multiplicación con α.



Para realizar la división por la constante *max* se preparó en memoria un valor de tipo float con el contenido $\frac{1}{max}$. El resultado anterior es entonces convertido en float y multiplicado en paralelo con esta constante. Luego se suma al canal de colores original para terminar el proceso.



Los canales R y G se calculan de la misma manera. Dado que se tiene los tres componentes separados el empaquetamiento se realiza de la misma manera que en la división por enteros del filtro sepia[2].

[2]Pág. 6

3. Experimentación

En la siguiente sección se realizarán comparaciones de las diferentes implementaciones de cada filtro, así como características particulares de los distintos algoritmos. Todas las comparaciones con filtros implementados en C utilizan variables tipo float y fueron compiladas con el flag -O3 a menos que se indique lo contrario en el experimento. Los archivos para replicar los experimentos se encuentran en la carpeta codigo/experimentos. La caracterización de las implementaciones se dividió principalmente en dos categorías: precisión y tiempos de ejecución (que se medirán en cantidad de ciclos).

3.1. Detalle sobre los experimentos de tiempo

Las imágenes utilizadas para comparar los tiempos de ejecución fueron generados por el script `codigo/tests/generar_tamanos_imagenes_exponenciales.py`. Éste se obtuvo a partir de una modificación del script `1_generar_imagenes.py` aportado por la cátedra. Se obtuvieron 12 imágenes de distintos tamaños. Para comparar las respectivas cantidades de ciclos de reloj se utilizó el *script* `run_time_script` (que se copió y modificó ligeramente según el experimento a realizar). El *script* toma tres parámetros:

Filtro puede ser 'cropflip', 'sepia' o 'ldr' y corresponde al filtro al cual se le desea medir el tiempo.

Implementación puede ser 'c' o 'asm' e indica si se utiliza la implementación C o ASM.

Cantidad indica la cantidad de veces que se aplicará el filtro sobre cada imagen. Al correr varias veces el filtro seremos capaces de lidiar con posibles outliers.

Como el filtro se aplica **cantidad** de veces para cada imagen obtenemos **cantidad** valores con los cuales definiremos la cantidad de ciclos que toma aplicar un filtro a dicha imagen. Para esto eliminamos los *outliers*. Dado que tenemos **cantidad** valores, asumimos que obtenemos una distribución normal[3]. Para eliminar los *outliers* usaremos la regla 68-95-97: si definimos μ como la media de estos valores, y σ como la varianza, esta nos indica que la probabilidad de que una variable este en el rango $[\mu-2\cdot\sigma; \mu+2\cdot\sigma]$ es aproximadamente 95 %. Si nos quedamos con los valores en este intervalo, nos quedamos entonces con aproximadamente el 95 % de los valores y al mismo tiempo eliminamos los *outliers*.

3.2. Floats y Doubles en C

Cuando comenzamos a implementar los distintos filtros en ASM, una cuestión que se presentó fue la elección de tipo *double* o *float* para las multiplicaciones. Para decidirlo realizamos una serie de experimentos utilizando las implementaciones en C.

La diferencia principal entre los dos tipos radica en la cantidad de bits que se utilizan para las representaciones, *double* 64bits y *float* 32bits. Al tener mayor definición, es de esperar que el tipo *double* tenga menor perdida de precisión al realizar operaciones y, al mismo tiempo, que las operaciones requieran más ciclos de reloj.

3.2.1. Experimento 1: Perdida de precisión

La idea de este experimento es evaluar la diferencia de precisión que se genera en los algoritmos de C al trabajar con floats y con double tanto en LDR como sepia.

Datos de entrada y procedimiento: Utilizamos para la comparación el archivo "colores32.bmp" en la carpeta img, dado que contiene un rango muy grande de colores representables por nuestro formato de imagen. Compilamos dos versiones de los algoritmos de C de los filtros LDR y sepia, una implementada con el tipo *float* y una con el tipo *double*. Procesamos la imagen con ambos filtros y ambas implementaciones. Para verificar diferencias utilizamos el programa `bmpdiff` con parámetro **epsilon** 0, que chequea píxel a píxel diferencias en los canales de colores con un rango de diferencias de **epsilon**.

[3] Sabemos por el teorema del límite central que la distribución de la suma de variables aleatorias tiende a distribución normal. Si consideramos la cantidad de ciclos como una variable aleatoria obtenemos entonces una distribución normal, razón por la cual utilizamos la regla 68-95-97.

Hipótesis: Suponiendo que la implementación de *double* tiene una precisión más cercana a la "correcta", creemos que las imágenes procesadas con *float* van a presentar una discrepancia de valores comparandolas con las procesadas con *double*.

Resultados: Bmpdiff no mostró diferencias en ninguna imagen.

Discusión y justificación de los resultados Creemos que la causa de no tener perdida de precisión, es que la diferencia que se genera entre utilizar floats y doubles se anula al convertir a enteros y saturar. No se están haciendo multiplicaciones lo suficientemente significativas para cambiar el valor en la conversión a enteros.

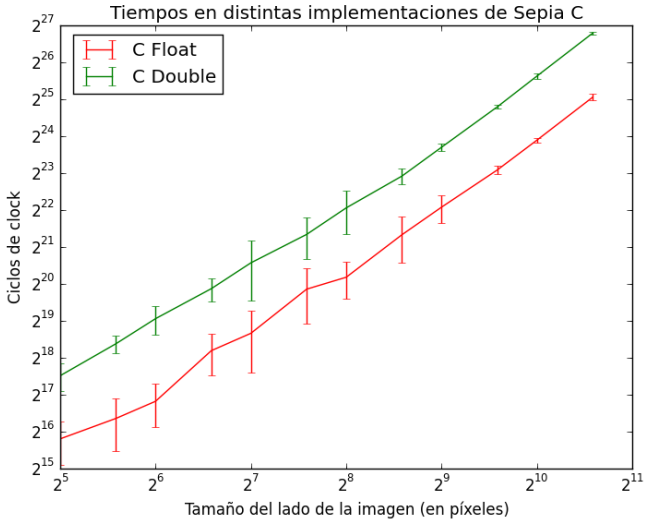
3.2.2. Experimento 2: Rendimiento

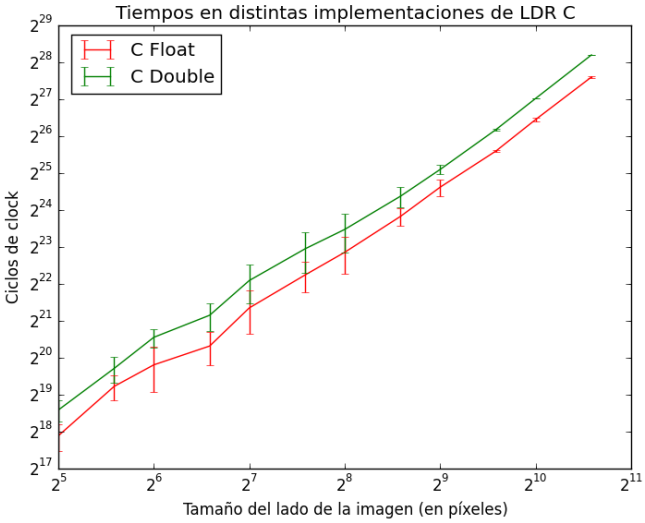
Vamos a comprobar las diferencias en tiempo de cómputo entre las implementaciones de float y double en C.

Datos de entrada y procedimiento: Usamos los mismos datos de entrada que para el experimento 1, solo que esta vez registramos los tiempos en cada corrida.

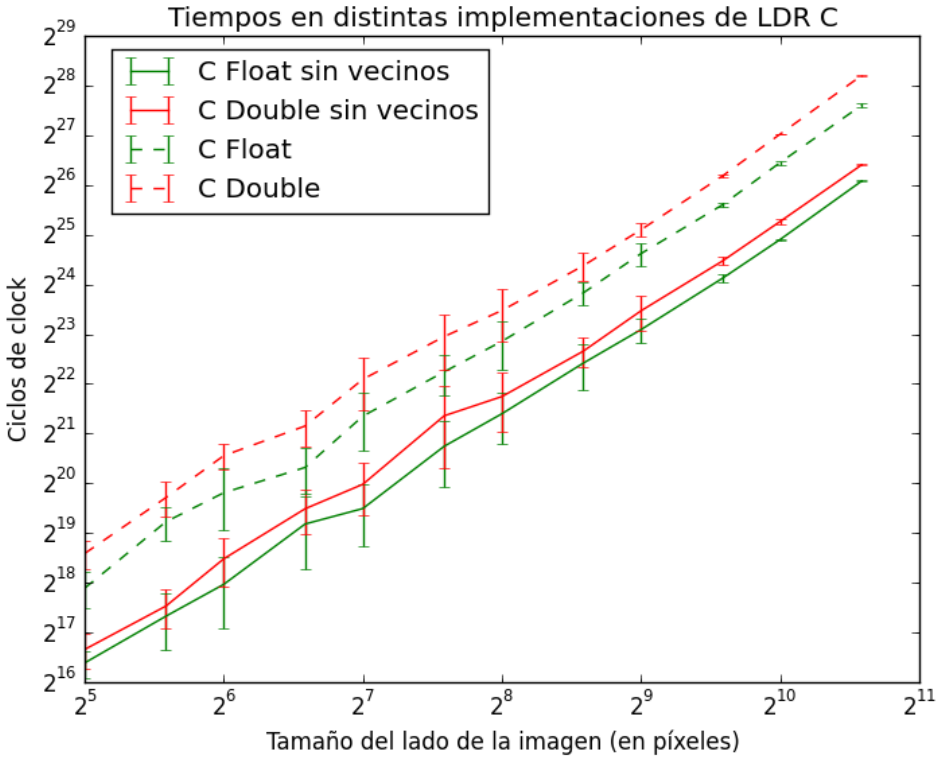
Hipótesis: El filtro implementado con floats debería ser notoriamente más rápido en ambos filtros.

Resultados:





Discusión y justificación de los resultados: A primera vista, parecería que los resultados confirman nuestra hipótesis, en el caso de sepia, el algoritmo con *float* corre cuatro veces más rápido que el de *double*. Mientras que en LDR, la diferencia no es tan marcada. Inicialmente pensamos que la diferencia entre las implementaciones en LDR se debía al cálculo de los vecinos (acceso a memoria y suma de vecinos). Al no tener operaciones de multiplicación que requieran punto flotante, realizamos nuevamente los experimentos evitando este cálculo.



La diferencia es aún menor aunque no deja de ser significativa. Para explicar la discrepancia evaluamos el código ASM generado al compilar ambos filtros.

En el caso de LDR (sin vecinos) el código generado para ambas implementaciones es exactamente igual, la diferencia son las instrucciones que trabajan con escalares *double* o *float*. Con este resultado podemos concluir que las operaciones con *double* son efectivamente más lentas que las operaciones con *float*.

Para sepia los códigos son similares, pero no idénticos. El sepia de que trabaja con *double* genera un código más largo. Por lo que es razonable suponer que la diferencia que muestra el gráfico no solo se debe a las operaciones flotantes, sino que el algoritmo *double* utiliza más instrucciones en ASM.

Conclusión: Habiendo verificado que no hay pérdida de precisión y las diferencias en velocidades favorecen trabajar con *float*, los experimentos indican que no se obtiene beneficio implementar estos filtros utilizando *double*.

También se debe tener en cuenta es que la vectorización en ASM permite el cálculo en paralelo de 4 valores utilizando *float* y 2 utilizando *double*, lo que puede incrementar aún más la diferencia en cuanto a tiempo de cómputo.

3.3. LDR

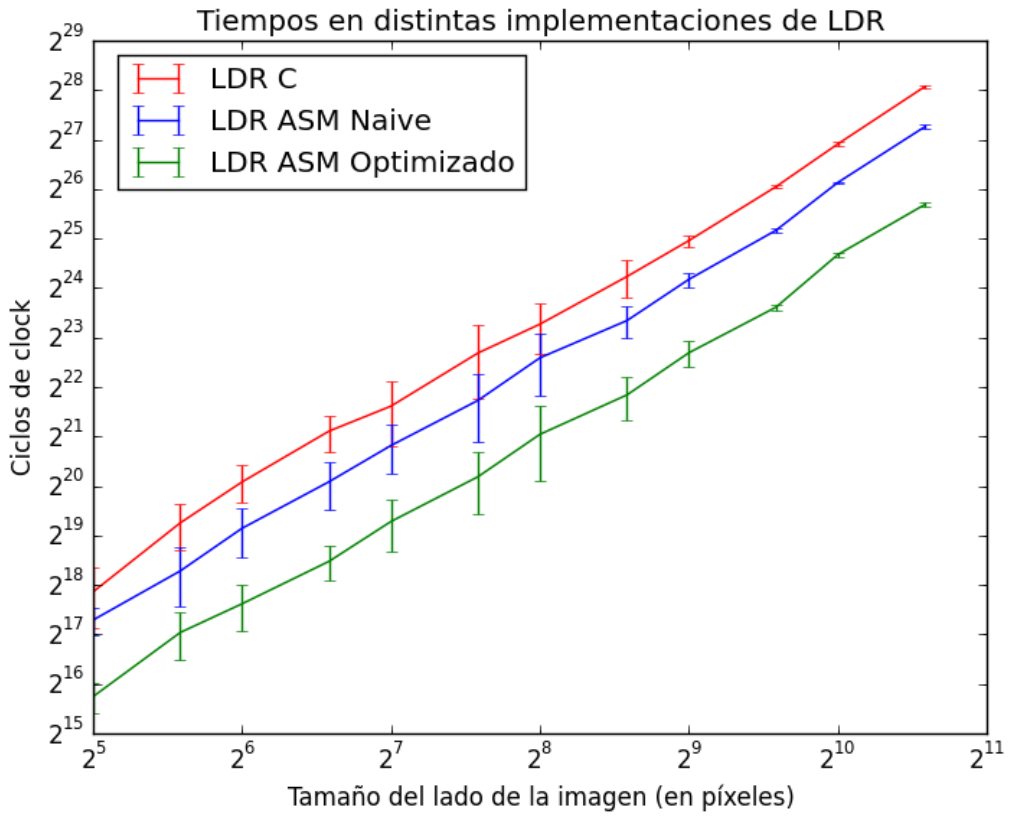
3.3.1. Experimento 1: Evaluación de tiempos

Comparamos los tiempos de ejecución de tres implementaciones distintas del algoritmo LDR: LDR C, LDR Naive, LDR Optimizado; explicados en la sección desarrollo.

Hipótesis: La implementación Optimizada tendrá menor tiempo que Naive, dado que la segunda requiere por cada píxel a procesar 10 accesos a memoria para realizar el calculo de los vecinos[4], mientras que la primera requiere un promedio de $2{,}5$. También es de esperar que ambas tengan un mejor rendimiento respecto a la implementación en C.

[4]Detallado en la correspondiente sección en Desarrollo. Pág. 8

Resultados:



Justificación y discusión: Se cumplieron las hipótesis propuestas. En cuanto al rendimiento del algoritmo implementado en C, realizamos una evaluación del código ASM generado por el compilador gcc y notamos que no pudo vectorizar el algoritmo de manera eficiente, por lo que es razonable la diferencia respecto a las implementaciones optimizadas manualmente.

3.3.2. Experimento 2: Evaluación de Sumas y Accesos

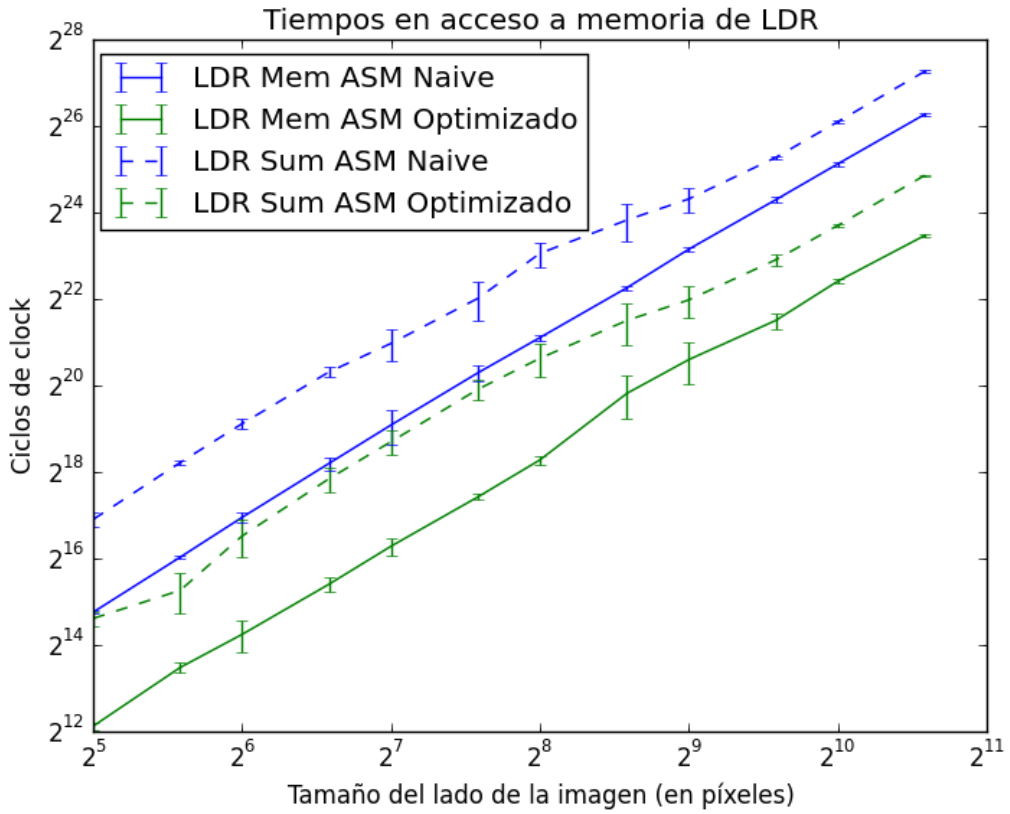
Vamos a realizar un análisis un poco más extensivo para evaluar si efectivamente la diferencia entre las implementaciones Naive y Optimizado radica en la cantidad de accesos a memoria.

Hipótesis: La diferencia principal entre ambos algoritmos se debe a la cantidad de accesos a memoria. Es importante notar que la memoria caché puede jugar un papel importante en esta evaluación, dado que la lectura de memoria se realiza de manera ordenada.

Datos de entrada y procedimiento: Modificamos los algoritmos Naive y Optimizado. Removimos el procesamiento de los colores que es igual en ambos algoritmos y nos concentramos en el cálculo de los vecinos.

El cálculo de los vecinos consta de dos partes, diferentes para ambas implementaciones, el acceso a memoria para traer los píxeles vecinos y la suma de los colores de los mismos. Evaluamos ambas por separado removiendo código de las implementaciones.

Resultados:



Discusión: Si bien los accesos a memoria marcan una diferencia entre ambos algoritmos, el mayor peso de los mismos radica en el cálculo de la suma. Suponemos que la memoria caché está reduciendo mucho el tiempo de acceso a los píxeles cercanos. Un posible experimento futuro sería replicar este experimento deshabilitando la misma o con una de menor tamaño.

3.4. Cropflip

En esta sección experimentaremos respecto de la efectividad de las instrucciones `SIMD`. Al mismo tiempo analizaremos características de las distintas implementaciones, por lo que también estudiaremos la influencia de la memoria cache.

Aclaración: En los experimentos siguientes se comparan los *miss rates* a la cache de distintas implementaciones así como sus tiempos de ejecución (en ciclos de clock). Para obtener los datos de cache se utilizó *valgrind*, corriendo:

```
valgrind --tool=cachegrind {resto_de_los_comandos}
```

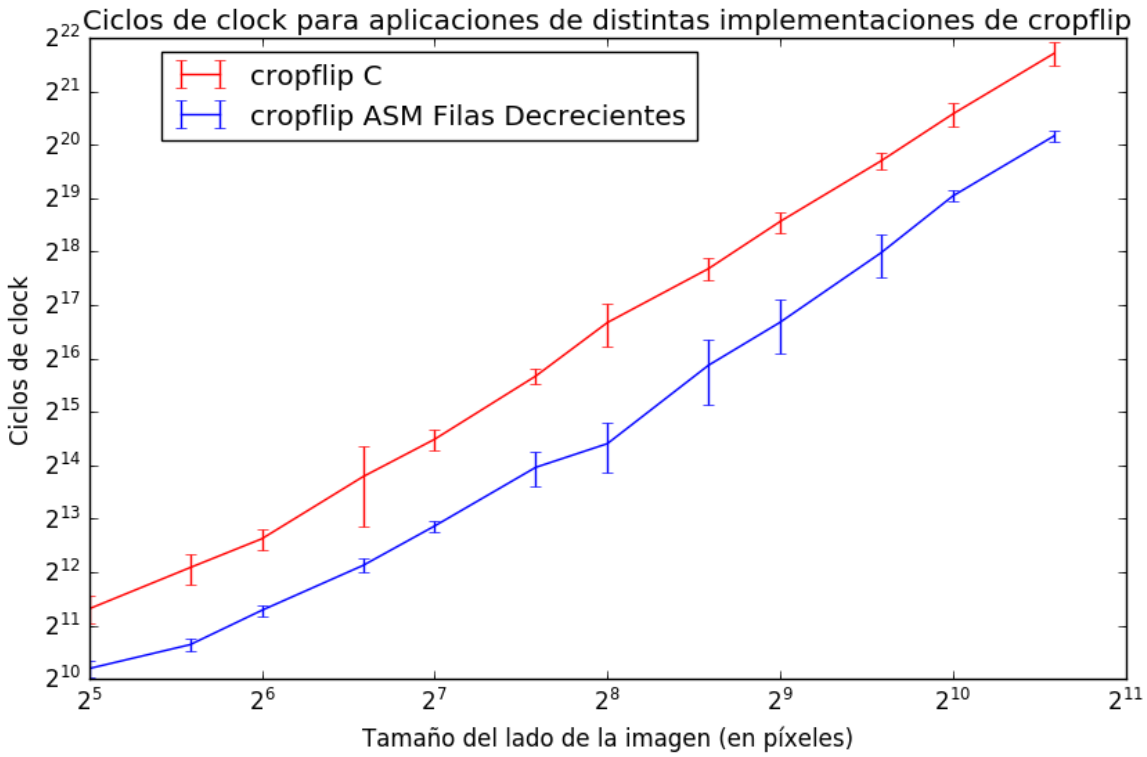
con la imagen `lena.1536x1536.bmp` obtenida a partir del *script* de la cátedra. Se puede replicar el experimento corriendo el mismo comando, obteniendo resultados que respetan las mismas proporciones.

Para los tiempos de ejecución se utilizó el *script* explicado previamente.

3.4.1. Experimento 1: Autovectorización

Inicialmente se realizaron dos implementaciones: la de C y la de `ASM_filas_decrecientes`. Por defecto, el compilador gcc que utilizamos encarga de autovectorizar, *ie* paralelizar las operaciones que sea posible. En consecuencia se autovectoriza el copiado de los píxeles en la implementación en C, utilizando instrucciones `SIMD`. El objetivo del trabajo es justamente evidenciar las ventajas de la utilización de estas instrucciones. Por lo tanto, agregamos al compilador la opción `-fno-tree-vectorize`. Esta se encarga de evitar la autovectorización, por lo que cada píxel será copiado individualmente.

Hipótesis: Dado que la implementación en C toma los píxeles individualmente, mientras que la implementación `ASM_filas_decrecientes` copia de a cuatro píxeles, nuestra primer hipótesis es que `ASM_filas_decrecientes` tendrá un menor tiempo de ejecución.



Resultados: Se cumple la hipótesis. La implementación en C es entre dos y cuatro veces más lenta que la implementación en ASM. Vemos entonces como efectivamente la utilización de instrucciones `SIMD` permite acelerar las aplicaciones del filtro *cropflip*.

3.4.2. Experimento 2: Influencia al recorrer por filas

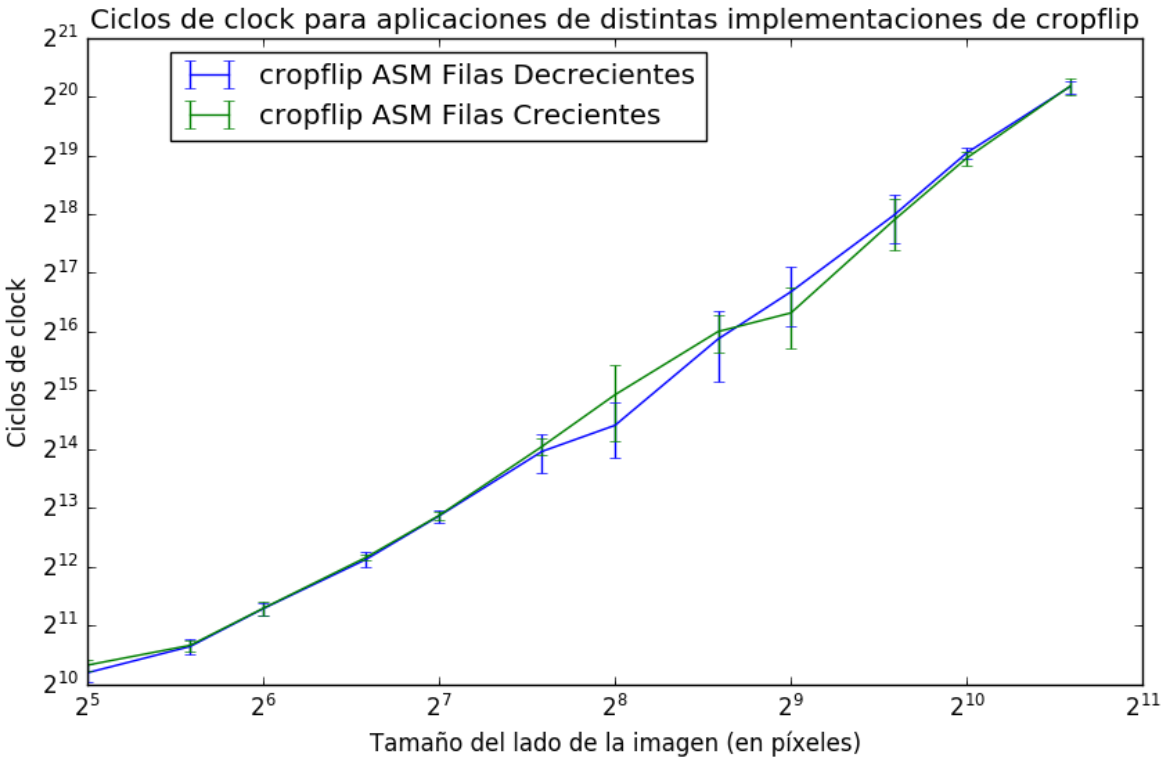
En la implementación `ASM_filas_decrecientes` se escriben las filas de la imagen de destino de arriba hacia abajo. Se realiza entonces un salto cada vez que se llega al final de una fila. Las escrituras no son entonces en posiciones contiguas de principio a fin. Por otro lado, en la implementación `ASM_filas_crecientes` se escriben las filas de la imagen de destino de abajo hacia arriba. Al mismo tiempo la matriz de la imagen de destino tiene el tamaño del *crop*. Las escrituras en el destino son entonces en posiciones contiguas del principio al fin de la aplicación del filtro (tal como si escribiéramos de manera contigua un vector).

Por su parte las lecturas a la imagen de origen son análogas en ambas implementaciones: dado que realizamos un recorte, las lecturas realizan saltos siempre que se llegue al final de la fila.

Hipótesis: La implementación `ASM_filas_crecientes` tendrá un tiempo de ejecución menor que la implementación de `ASM_filas_decrecientes`. Esperamos también que `ASM_filas_decrecientes` tenga más misses.

Resultados:

Ambas implementaciones tienen resultados similares y ninguna resalta respecto de la otra.



Implementación	Taza de *misses*
`ASM_filas_decrecientes`	0.7 %
`ASM_filas_crecientes`	0.7 %

Figura 3: Tazas de *misses* de las distintas implementaciones

Observando la figura 3 vemos que ambas tazas de misses fueron las mismas. Las hipótesis planteadas no se comprobaron.

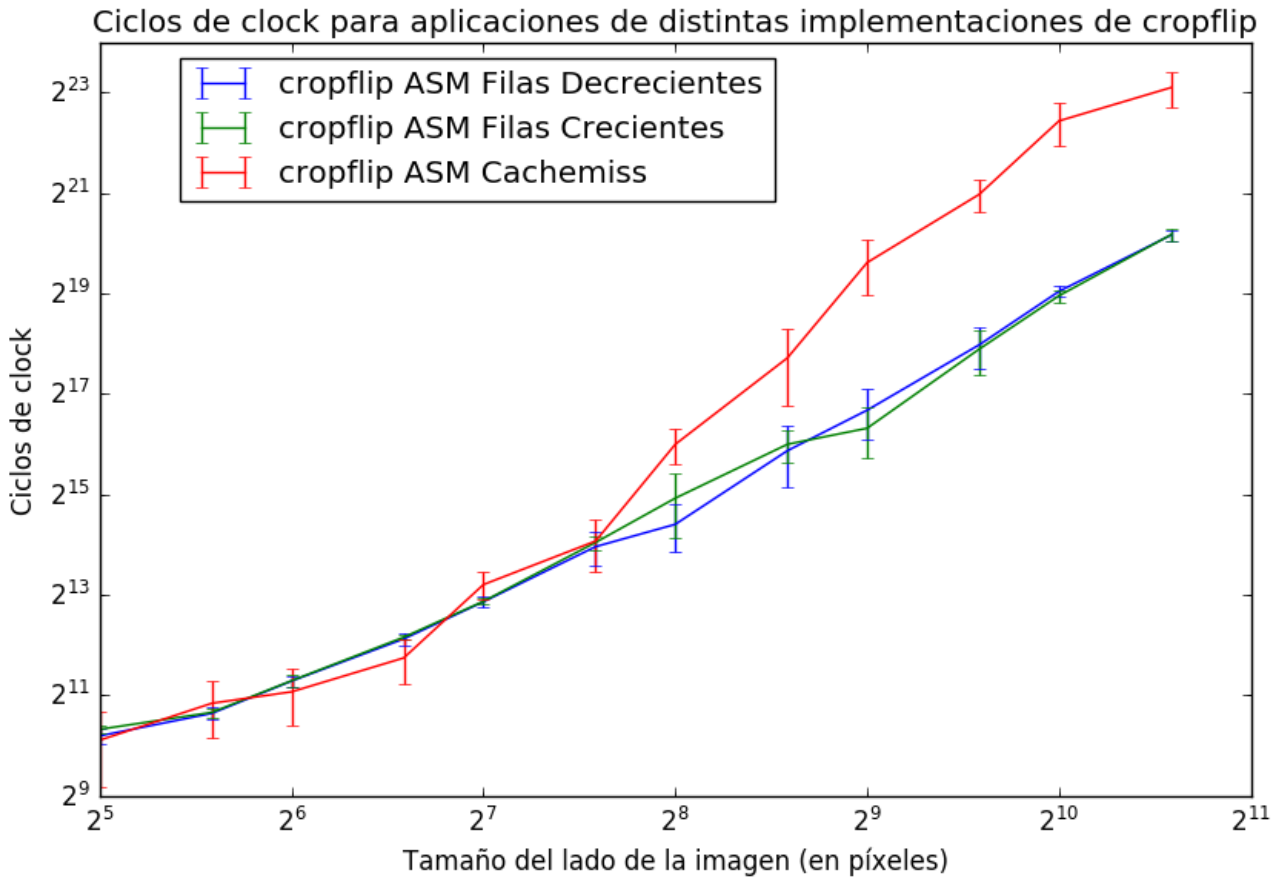
Justificación: El error en el razonamiento es intuir que al escribir de manera contigua la memoria, se necesiten menos lineas de cache. En efecto, si escribimos las filas de la matriz de destino de arriba hacia abajo, la memoria que se pida para la fila superior también puede ser útil para la fila inferior. Lo mismo ocurre si se recorren las filas de abajo hacia arriba. Por lo tanto ambos casos son similares y no hay diferencias en los tiempos de ejecución. Para expandir el análisis decidimos entonces implementar otra alternativa, `cropflip_ASM_cachemiss`, que recorre la matriz por columnas.

3.4.3. Experimento 3: Influencia al recorrer por columnas

La implementación `cropflip_ASM_cachemiss` lee y escribe las filas por columnas en lugar de hacerlo por filas. Para aprovechar las instrucciones `SIMD` consideraremos columnas de cuatro píxeles de ancho.

Hipótesis: La implementación `cropflip_ASM_cachemiss` tendrá mayores tiempos de ejecución que las otras implementaciones escritas en ASM.

Resultados:



Implementación	Taza de *misses*
`ASM_filas_decrecientes`	0.7 %
`ASM_filas_crecientes`	0.7 %
`ASM_cachemiss`	0.9 %

Cuadro 1: Tazas de *misses* de las distintas implementaciones

En el gráfico notamos que si bien para imágenes chicas los tiempos son parecidos, a partir de imágenes de ancho 256 aparece una diferencia de tiempo. Esta diferencia se ensancha a medida que agranda el tamaño de la imagen.

Para imágenes pequeñas los tiempos son los mismos, pero a partir de cierto punto los resultados coinciden con la hipótesis planteada, lo cual se observa en la tabla 1 y en el gráfico de comparación de tiempos de ejecución.

Justificación: Al estar recorriendo la matriz por columnas, no se accede a la memoria de manera contigua. Por lo tanto, si la columna es lo suficientemente grande (tal que al terminar de recorrer la columna ya fueron reemplazadas en el cache los datos correspondientes a los datos de las primeras filas), no estarán presentes en la memoria cache la filas de la imagen destino que necesitamos. Sin embargo al recorrer la imagen por filas no se produce este problema. Por lo tanto si se supera este tamaño, recorrer por columnas será menos eficiente. Suponemos que este tamaño está directamente relacionado con el tamaño de la memoria cache.

Conclusión: En la figura 4 se grafican los tiempos de ejecución resultantes de aplicar la implementación C **con autovectorización**. Notemos como los tiempos son similares a los de las implementaciones por filas de ASM. Ya que el cropflip es un problema sencillo, las implementaciones tienen poca complejidad. Por

lo tanto, la optimización del compilador de C permite obtener implementaciones tan eficientes como en ASM.

Por otro lado en esta sección podemos también evidenciar la importancia de la memoria cache: en efecto, la forma en la cual recorríamos la matriz que representa la imagen nos mostró como pueden cambiar los tiempos de ejecución. Tendría sentido realizar un análisis con distintos tamaños de caché, y ver luego la influencia de esta al aplicar el filtro *cropflip* a imágenes de distintos tamaños.

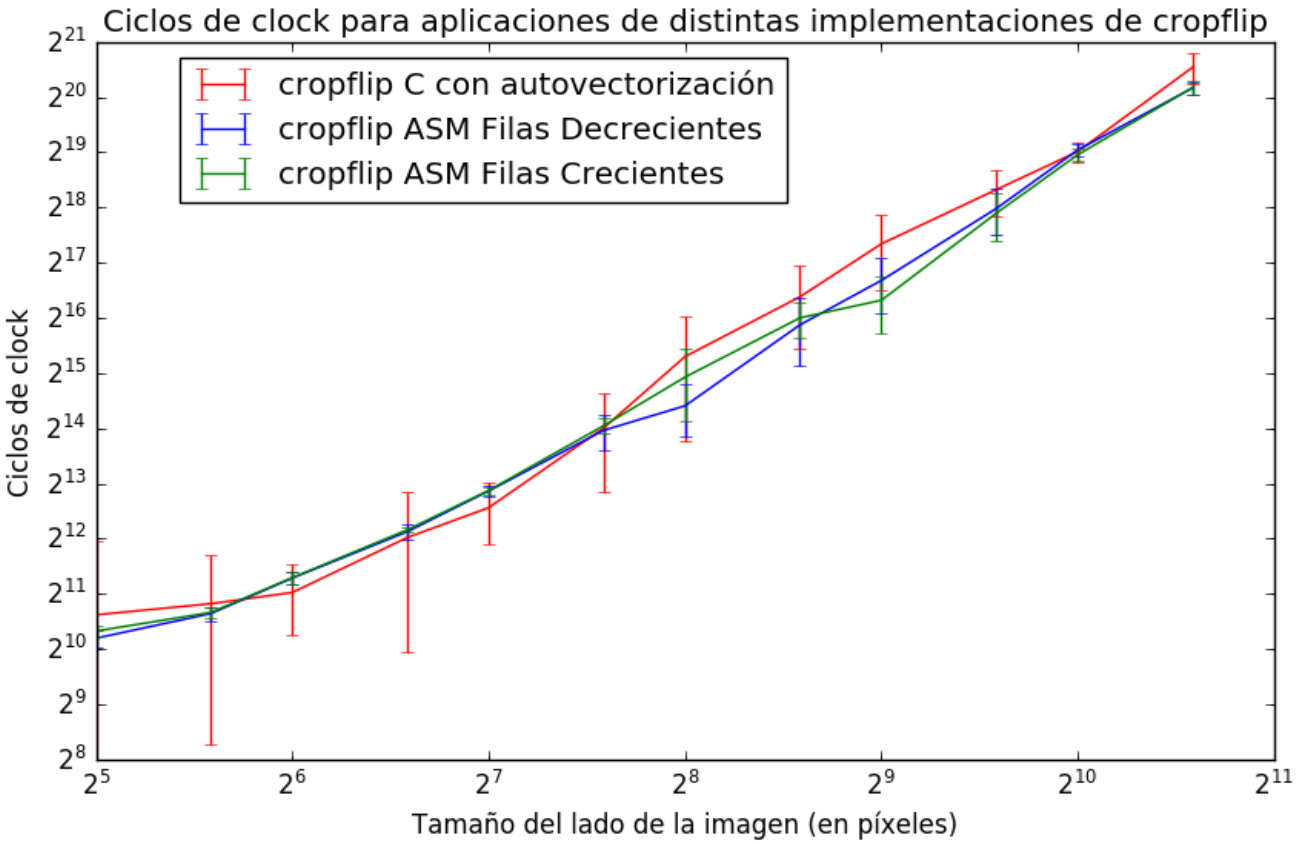


Figura 4

3.5. Sepia

El objetivo de esta sección será nuevamente evidenciar las ventajas de utilizar instrucciones SIMD. También se evaluará una implementación de división alternativa buscando obtener mejores resultados.

3.5.1. Experimento 1: C vs ASM

En este primer experimento compararemos los tiempos entre una implementación en C del sepia (que considera los píxeles individualmente) contra los tiempos de una implementación en ASM (que utiliza instrucciones SIMD).

Hipótesis: La implementación en ASM tendrá mejores tiempos al aplicar el filtro.

Resultados:

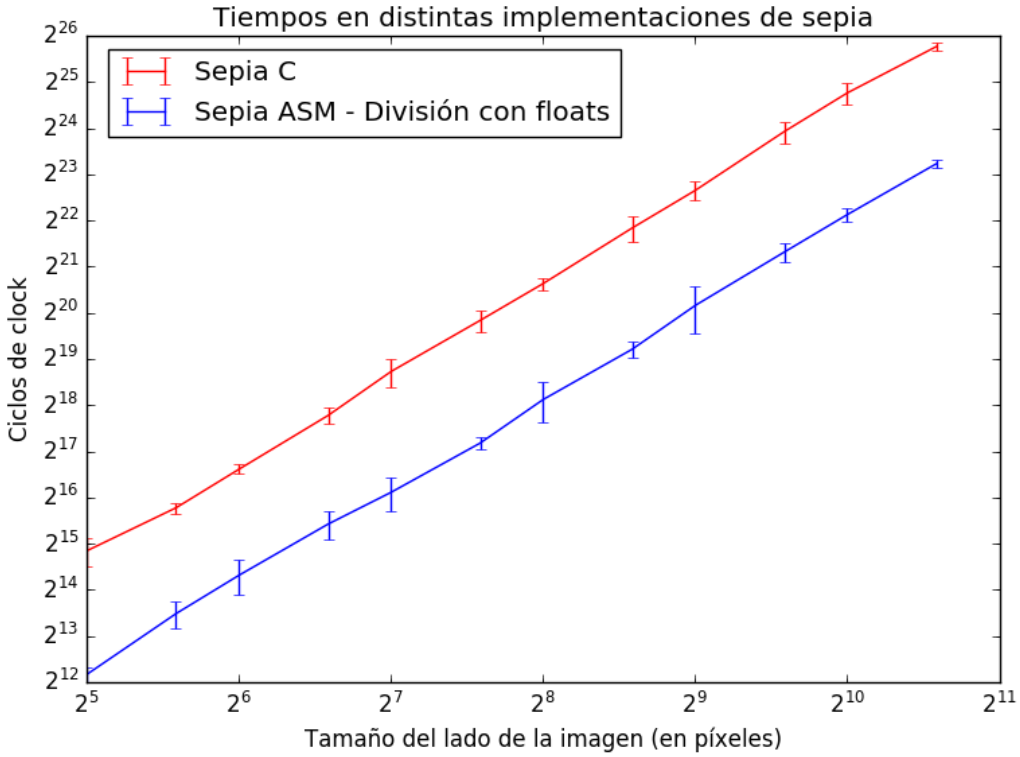


Figura 5: Tiempos en C contra tiempos en ASM

En la figura 5 vemos los tiempos obtenidos con cada implementación. La implementación en C resultó aproximadamente cuatro veces más lenta que la implementación en ASM. Este resultado es razonable si consideramos que estamos paralelizando las operaciones al punto de tratar cuatro píxeles en simultáneo.

Discusión: Es evidente entonces que la utilización de instrucciones SIMD permite mejorar los tiempos de cómputo de nuestros filtros. Sin embargo la utilización de estas instrucciones es sólo alguna de las herramientas que podemos utilizar para optimizar los tiempos. Buscamos entonces una implementación alternativa que ataque la división de otra manera.

3.5.2. Experimento 2: División por enteros

La implementación de ASM alternativa se basa en la división entera explicada en el desarrollo. Esta corresponde a uno de los tantos algoritmos que Agner Fog explica en su informe[2].

Hipótesis: Dado que el objetivo de Agner Fog es optimizar subrutinas, esperamos que la implementación alternativa otorgue mejores resultados que la implementación original.

Resultados:

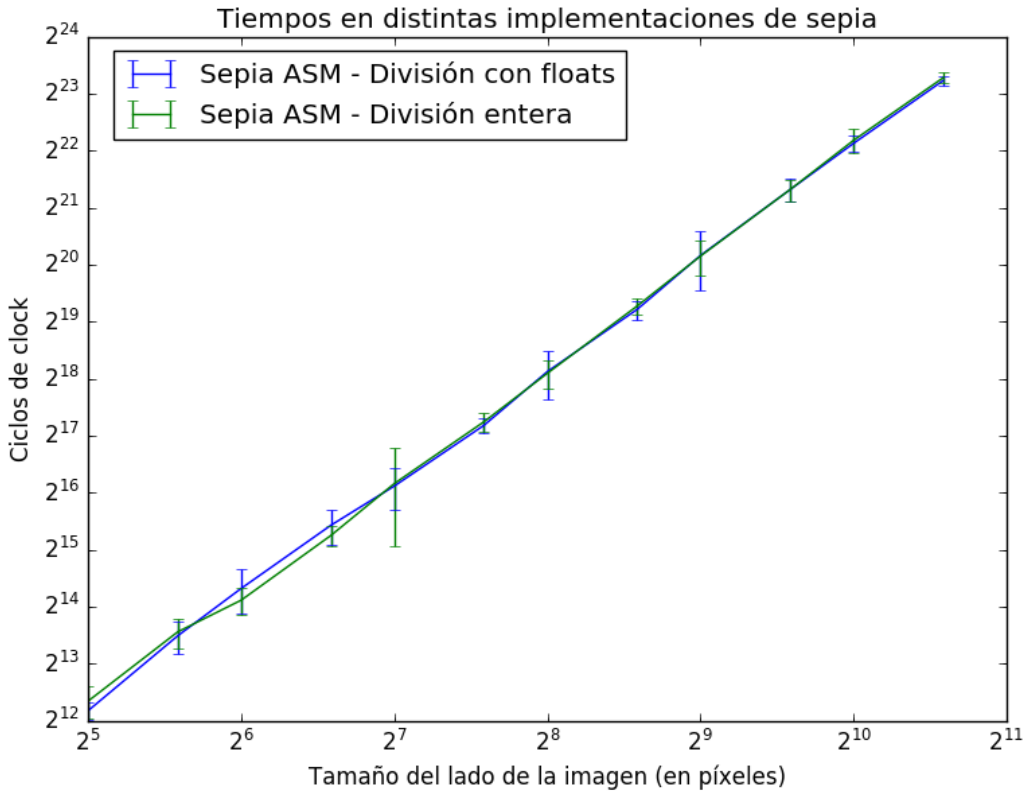


Figura 6: Tiempos en ASM para diferentes algoritmos

En la figura 6 vemos los tiempos obtenidos para nuestra implementaciones. Nuestra hipótesis fue errónea: ambas implementaciones obtuvieron tiempos cuyas diferencias son despreciables.

Discusión: La implementación alternativa no otorga mejores resultados. Es posible que el peso de las divisiones se vea tapado por el resto de las operaciones SIMD y que le hayamos dado demasiado importancia a las divisiones que se hacen a lo largo del filtro. En futuros experimentos podría analizarse que tan relevante son estas operaciones.

Al mismo tiempo, si se desea seguir realizando una optimización se podrían llevar a cabo otros experimentos que analicen distintos aspectos del código. El trabajo de Agner Fog podría darnos distintos puntos de partida para estos análisis

4. Conclusión

Los experimentos realizados permiten rescatar ciertos puntos importantes. En particular se hizo visible la importancia de la memoria cache, diferentes implementaciones pueden realizar un mejor aprovechamiento de la misma. El caso del filtro cropflip es un ejemplo de esto, se puede ver como al recorrer la matriz de maneras distintas se obtienen distintos tiempos de ejecución. Mientras que en el filtro LDR permite que el acceso a memoria para traer los datos de vecinos sea más rápido que el cálculo de la suma de los mismos.

Por otro lado, notamos que las implementaciones en floats no resultaron menos precisas que las implementaciones en double, pero si más rapidas. Dados estos resultados concluimos que en el contexto de los fitros del trabajo, no es necesario utilizar doubles, pues no presentan ventajas. Si bien utilizar multiplicación por enteros no garantizó una mejora de rendimiento, suponemos que está estrictamente relacionado con la microarquitectura del procesador en cuestión. Queda para futuros experimentos su análisis en distintas arquitecturas.

Bibliografía

[1] Agner Fog. *Optimizing subroutines in assembly language: An optimization guide for x86 platforms*. URL: `http://www.agner.org/optimize/optimizing_assembly.pdf` (visitado 04-05-2016).

[2] *Instruction tables*. URL: `www.agner.org/optimize/instruction_tables.pdf` (visitado 03-06-2016).