DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

Organización del Computador II

Segundo Cuatrimestre de 2016

Grupo: **Daniel Olano - Pabellon de Aragon**

Apellido y Nombre	LU	E-mail
Luaciano Saenz	904/13	saenzluciano@gmail.com
Federico Sassone	602/13	fede.sassone@hotmail.com
Guido Teren	749/14	guidoteren@gmail.com

Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

Índice

1. Introducción **3**

2. Implementaciones **3**
2.1. Smalltiles . . . 3
2.1.1. Pseudocódigo del algoritmo en C: . . . 3
2.1.2. Descripción de implementación en ensablador: . . . 3
2.1.3. Descripción del ciclo: . . . 4
2.1.4. Comparación con la implementación C . . . 5
2.1.5. Comparacion con distintos tamaños de imagen . . . 5
2.1.6. JUMP vs LOOP . . . 6
2.2. Pixelar . . . 8
2.2.1. Pseudocódigo del algoritmo en C: . . . 8
2.2.2. Descripción de implementación en ensablador: . . . 8
2.2.3. Descripción del ciclo: . . . 8
2.2.4. Comparación con la implementación C . . . 11
2.2.5. Comparación de precisión . . . 12
2.2.6. Comparación diferentes tamaños . . . 12
2.3. Combinar . . . 13
2.3.1. Descripción de implementación en ensablador: . . . 13
2.3.2. Antes del ciclo . . . 13
2.3.3. Descripción del ciclo . . . 13
2.3.4. Experimentacion . . . 15
2.4. Colorizar . . . 21
2.4.1. Descripción del ciclo . . . 21
2.4.2. Lectura de píxeles vecinos . . . 21
2.4.3. Búsqueda de máximos por canal . . . 21
2.4.4. Hallar $\Phi_{R,G,B}$ . . . 22
2.4.5. Escritura de pixel destino . . . 22
2.4.6. Comparación con la implementación C . . . 23
2.4.7. Rendimiento . . . 24
2.5. Rotar . . . 26
2.5.1. Descripción del ciclo . . . 26
2.5.2. Comparación con la implementación C . . . 26
2.5.3. Rendimiento . . . 26

3. Conclusión **28**

1. Introducción

En este trabajo practico se llevo a cabo la implementación de una serie de filtros gráficos, en lenguaje C y lenguaje ensamblador, con el objetivo de familiarizarnos con el modelo de procesamiento SIMD y conocer sus ventajas y desventajas.

Nuestro objetivo en el informe es tener un primer acercamiento al método de investigación y presentación de resultados de forma exhaustiva.

Lo haremos primero haciendo una descripción de los filtros con los que trabajaremos, luego buscando presentar los resultados de las implementaciones realizadas de forma fácil de leer y finalmente buscando una explicación a estos resultados, obteniendo una conclusión final.

2. Implementaciones

Consideramos a una imagen como una matriz de píxeles de tamaño M, m filas y n columnas. Cada píxel está determinado por cuatro componentes: los colores azul ,verde y rojo ,y la transparencia . En nuestro caso particular cada una de estas componentes tendrá 8 bits (1 byte) de profundidad, es decir que estarán representadas por números enteros en el rango $[0, 256)$.

2.1. Smalltiles

El filtro Smalltiles consiste en repetir la imagen original 4 veces, de forma mas chica en la imagen destino. Es decir, que si originalmente se tiene una imagen de tama no w x h, en el destino se tendran 4 imagenes de tamañno $w/2 \times h/2$, una en cada cuadrante. Para copiar los pxeles, a cada (i; j) de la primera imagen destino, le corresponde el valor de la posicion (2i; 2j) en la imagen fuente. Esto significa que para cada subimagen i de la imagen destino, esta tendrá un cuarto de los pixeles de la imagen fuente original.

2.1.1. Pseudocódigo del algoritmo en C:

for $i \leftarrow 1$ to $alto/2$ **do**
  for $j \leftarrow 1$ to $ancho/2$ **do**
    $dst[i][j*4] \leftarrow src[i*2][j*2*4]$;
    $dst[i][(ancho*4)/2 + j*4] \leftarrow src[i*2][j*2*4]$;
    $dst[alto/2 + i][j*4] \leftarrow src[i*2][j*2*4]$;
    $dst[alto/2 + i][(ancho*4)/2 + j*4] \leftarrow src[i*2][j*2*4]$;
  end for
end for

2.1.2. Descripción de implementación en ensablador:

Esta implementación consiste en la realización de dos ciclos anidados para avanzar las columnas por cada fila. Utilizando registros del modelo SIMD pudimos recibir cuatro pixeles consecutivos en un registro, lo que nos permitió realizar menos ciclos. Al recibir los pixeles se posicionan en un registro de manera que se descartan los que debemos omitir y dejandolos preparados para copiarlos en los cuatro bloques de la imagen destino.

2.1.3. Descripción del ciclo:

- El primer paso realizado dentro del ciclo fue obtener de la imagen los próximos 4 píxeles de la fila actual en el registro `XMM1`.

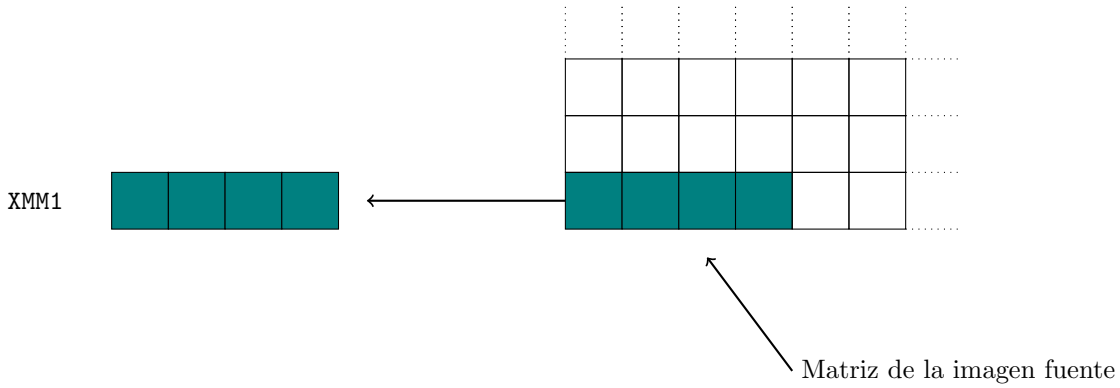


Figura 1: Copia de pixeles desde la imagen.

- El segundo paso consiste en usar la instrucción `PSHUFD` que nos permite posicionar los pixeles de la forma que queramos. Usamos como fuente y destino el mismo registro `XMM1`. Aquí hay que tener en cuenta que a cada (i; j) de la primera imagen destino, le corresponde el valor de la posición (2i; 2j) en la imagen fuente.

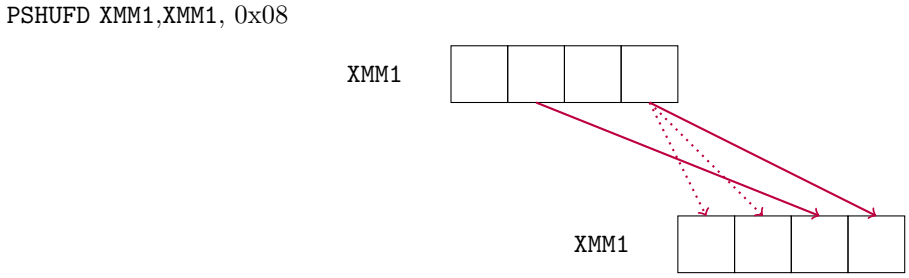


Figura 2: shuffle sobre el registro 1.

- El tercer paso es copiar la parte baja del registro `XMM1` en los cuatro cuadrantes de la imagen destino.

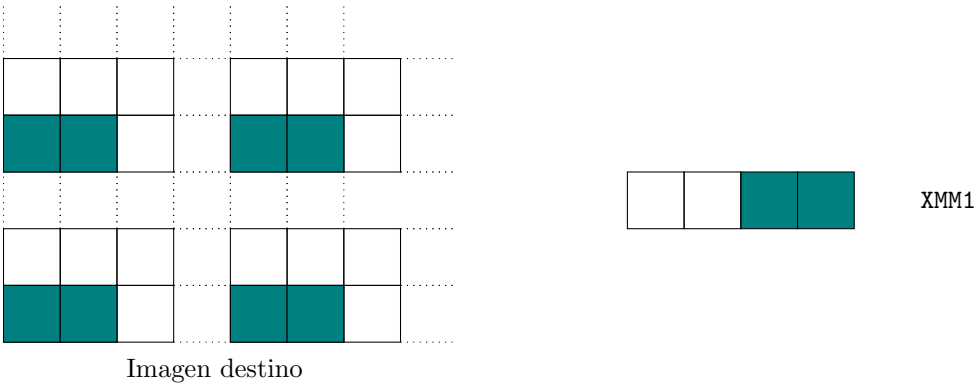


Figura 3: shuffle sobre el registro 1.

- El cuarto y ultimo paso consiste en posicionar los punteros de las imagenes para poder repetir el ciclo con los siguientes pixeles. Dado que los pixeles se copian en 4 cuadrantes de la imagen destino en un mismo ciclo, el puntero a la matriz destino solo se mueve por 1/4 de la imagen. En cuanto

al puntero de la matriz fuente, recorre toda la columnas tomando de a 4 pixeles y solo la mitad de las filas ya que las impares deben omitirse.

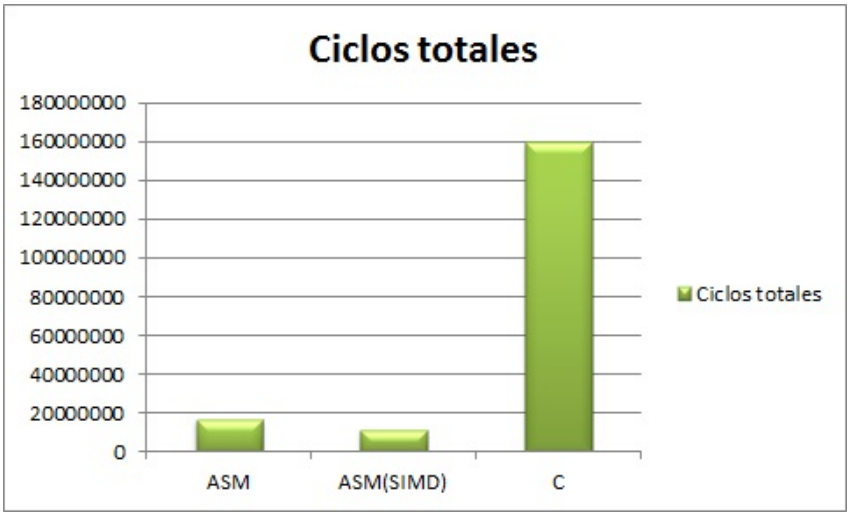
2.1.4. Comparación con la implementación C

En esta sección mostraremos en un gráfico el resultado obtenidos en la comparación entre la implementación de C y la de ASM.

Partimos de la hipótesis de que nuestra implementación en ASM sería más eficiente que la de C, y como se ve claramente en el gráfico, la implementación de ASM es mucho más eficiente que la de C, ya que procesa el filtro deseado en una menor cantidad de ciclos. Esto se debe en parte a la utilización del modelo SIMD, que nos permite, en este caso, operar con el doble de pixeles que con la implementación en C.

Como se podrá apreciar, la cantidad de ciclos efectuados por el algoritmo en C son mucho mayores que la de ASM, por lo que observamos que el buen rendimiento de ASM no se debe solo a la utilización de SIMD. Para poder apreciar la mejora de la implementación con SIMD procedimos a implementar el filtro nuevamente en ASM pero sin utilizar dicho modelo e incluimos su eficiencia en el mismo gráfico.

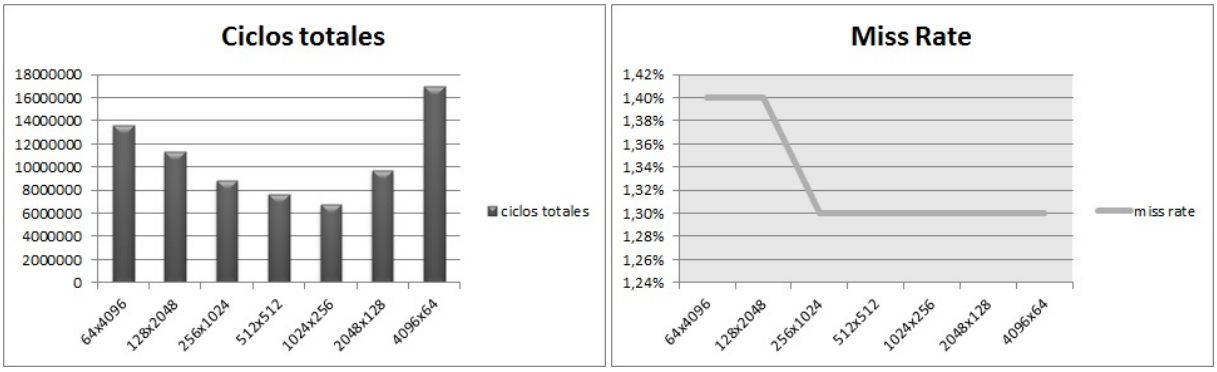
Como vemos, si bien SIMD mejoro en gran parte nuestro algoritmo de ASM la mayor parte de las mejoras del algoritmo en C no son a causa de la utilización de dicho modelo



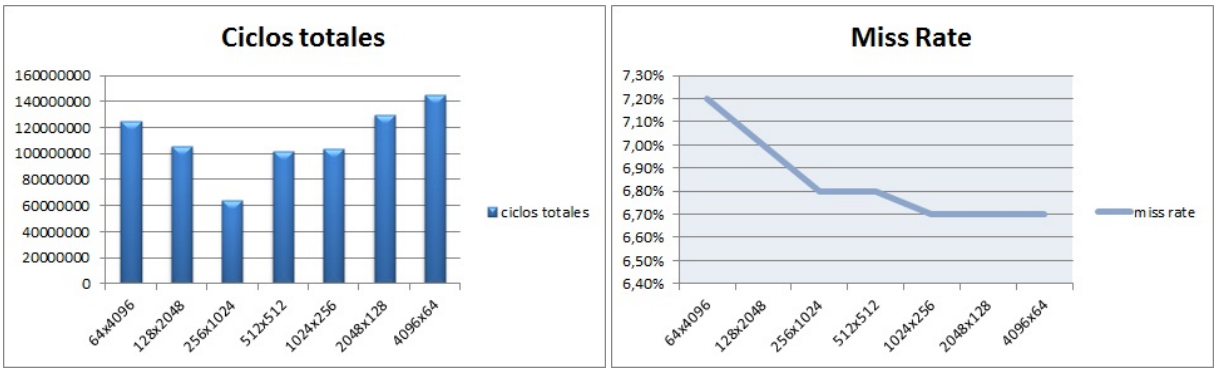
2.1.5. Comparacion con distintos tamaños de imagen

El siguiente experimento se basó en la hipótesis de que con un menor ancho de imagen produciría un mayor **miss rate** que un ancho mayor, lo que ocasionaría, al no encontrar el dato necesario en la cache, tener que acceder a memoria, generando una mayor cantidad de ciclos totales. Para probar nuestra hipótesis tomamos distintas imágenes de igual cantidad de pixeles pero distribuidas de diferente forma, haciendo variar la cantidad de filas. Los resultados obtenidos fueron detallados en los gráficos al final de la subsección para distintas cantidades de iteraciones.

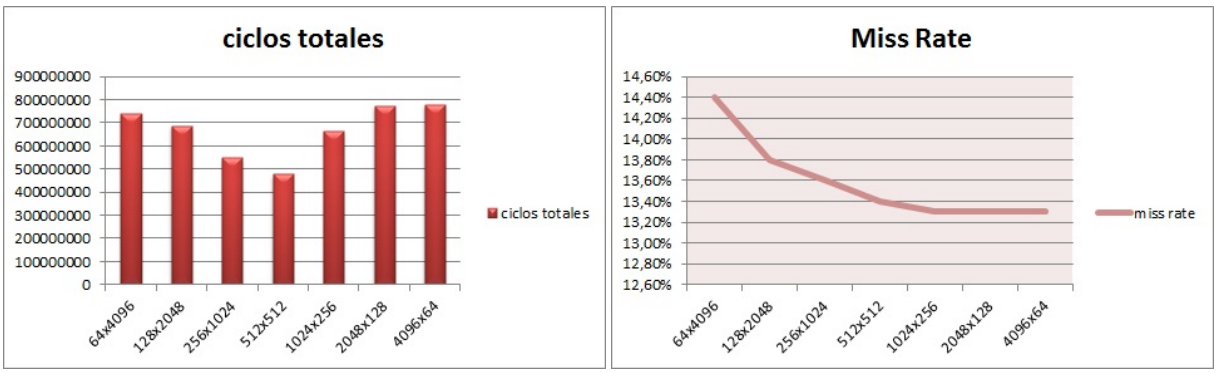
Como se puede observar en dichos gráficos, si bien el **miss rate** disminuye a un mayor ancho de imagen la cantidad de ciclos no se comporta de igual manera en todas las imágenes. En un principio, tanto los ciclos como el miss rate, disminuyen frente al aumento del ancho de la imagen. Sin embargo, cerca de la imagen promedio (512x512) el **miss rate** comienza a estabilizarse y la cantidad de ciclos a aumentar.



Comparación tamaños: 10 iteraciones



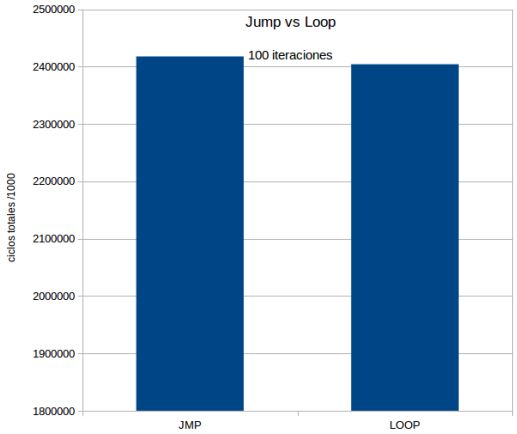
Comparación tamaños: 100 iteraciones

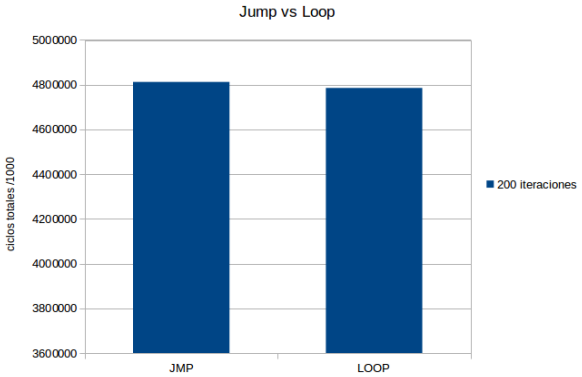


Comparación tamaños: 1000 iteraciones

2.1.6. JUMP vs LOOP

En esta seccion vamos a plantear un experimento el cual nos va a ayudar a ver como afectan a la performance los saltos condicionales. Para ello vamos a evitarlos utilizando la funcion loop. Nuestra hipotesis es que la implementacion con Loop va a tener un mejor rendimiento frente a la implementacion con jump. Los resultados son los siguientes:





Los graficos confirman nuestra hipotesis: al evitar realizar comparaciones, la implementacion con loop tiene un leve mejor rendimiento que la implementacion con jump.

2.2. Pixelar

El filtro Pixelar consiste en partir la imagen original en bloques de 2×2 píxeles. Por cada componente por separado, y para cada bloque de la imagen original, se genera un bloque del mismo tamaño en la imagen destino y a cada uno de sus píxeles se le asigna el promedio de los valores de los píxeles del bloque de la imagen original.

2.2.1. Pseudocódigo del algoritmo en C:

for $i \leftarrow 1$ to *alto* **do**
  for $j \leftarrow 1$ to *ancho* **do**
    $*ps_0 \leftarrow src[i][j * 4];$
    $*ps_1 \leftarrow src[i][(j + 1) * 4];$
    $*ps_2 \leftarrow src[i + 1][j * 4];$
    $*ps_3 \leftarrow src[i + 1][(j + 1) * 4];$
    $bi \leftarrow ((ps_0 \rightarrow b + ps_1 \rightarrow b + ps_2 \rightarrow b + ps_3 \rightarrow b)/4);$
    $gi \leftarrow ((ps_0 \rightarrow g + ps_1 \rightarrow g + ps_2 \rightarrow g + ps_3 \rightarrow g)/4);$
    $ri \leftarrow ((ps_0 \rightarrow r + ps_1 \rightarrow r + ps_2 \rightarrow r + ps_3 \rightarrow r)/4);$
    $(dst[i][j * 4] \rightarrow b) \leftarrow bi;$
    $(dst[i][j * 4] \rightarrow g) \leftarrow gi;$
    $(dst[i][j * 4] \rightarrow r) \leftarrow ri;$
    $(dst[i][(j + 1) * 4] \rightarrow b) \leftarrow bi;$
    $(dst[i][(j + 1) * 4] \rightarrow g) \leftarrow gi;$
    $(dst[i][(j + 1) * 4] \rightarrow r) \leftarrow ri;$
    $(dst[i + 1][j * 4] \rightarrow b) \leftarrow bi;$
    $(dst[i + 1][j * 4] \rightarrow g) \leftarrow gi;$
    $(dst[i + 1][j * 4] \rightarrow r) \leftarrow ri;$
    $(dst[i + 1][(j + 1) * 4] \rightarrow b) \leftarrow bi;$
    $(dst[i + 1][(j + 1) * 4] \rightarrow g) \leftarrow gi;$
    $(dst[i + 1][(j + 1) * 4] \rightarrow r) \leftarrow ri;$
  end for
end for

2.2.2. Descripción de implementación en ensablador:

Esta implementación consiste en la realización de un ciclo donde se suman cuatro pixeles y se divide ese resultado por 4. Gracias al modelo SIMD pudimos recibir cuatro pixeles consecutivos en un registro, lo que nos permitio realizar menos ciclos, y dado que la operación de suma se realizaba entre pixeles de un bloque de 2×2 debimos trabajar con 8 pixeles para aprovechar al máximo esta ventaja, esto fue posible dado que tanto el ancho como el alto de las imágenes era múltiplo de cuatro, y solo trabajábamos con 4 pixeles por fila.

2.2.3. Descripción del ciclo:

- El primer paso realizado dentro del ciclo fue obtener de la imagen los próximos 4 píxeles de la fila actual en el registro `XMM1` y los proximos 4 píxeles de la fila siguiente en el registro `XMM3`, para luego trabajar con 2 bloques de 2×2 a la misma vez.

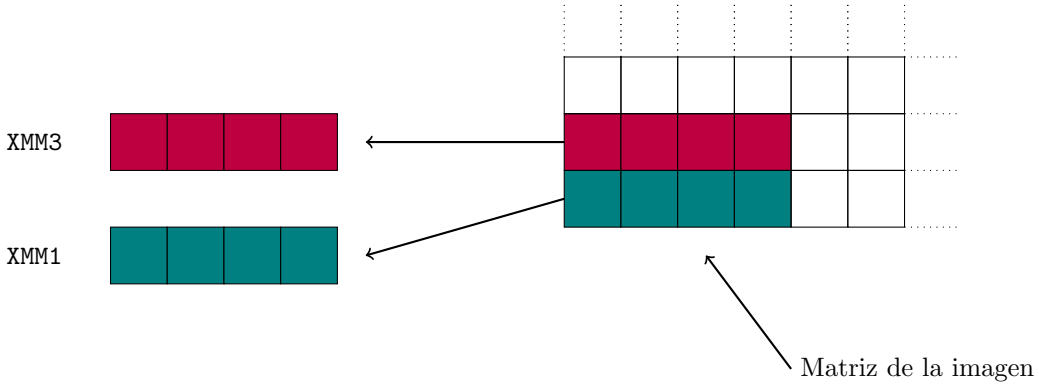


Figura 4: Copia de pixeles desde la imagen.

- Aunque logramos obtener más pixeles por el modelo SIMD, dado que se debe realizar una suma entre pixeles de cada una de sus componentes, las cuales poseen valores entre 0 y 255, si no se operan de manera correcta se puede perder información ya que dicha suma supera el valor 255 y no lo podamos representar con un solo byte. Por esta razón procedimos a desempaquetar cada componente a Word para almacenar de manera completa el resultado, lo que nos dividió el problema en dos, ya que entonces necesitábamos dos registros para poder almacenar los cuatro pixeles. El desempaquetado se realiza con la instrucción `PUNPCKHBW` y `PUNPCKLBW`. Obteniendo de esta manera en los registros `XMM1` y `XMM2` el valor de los píxeles en words de la fila actual y en `XMM3` y `XMM4` los de la siguiente fila, quedandonos asi, dos pixeles por registro. Para la realización de este proceso utilizamos el registro `XMM7`, el cual fue seteado a cero previamente. La parte alta del registro `XMM1` es desempaquetada en el mismo registro, mientras que la parte baja se desempaqueta guardando el resultado en el registro `XMM2` como se muestra a continuación. Lo mismo pasa con `XMM3`, `XMM4`.

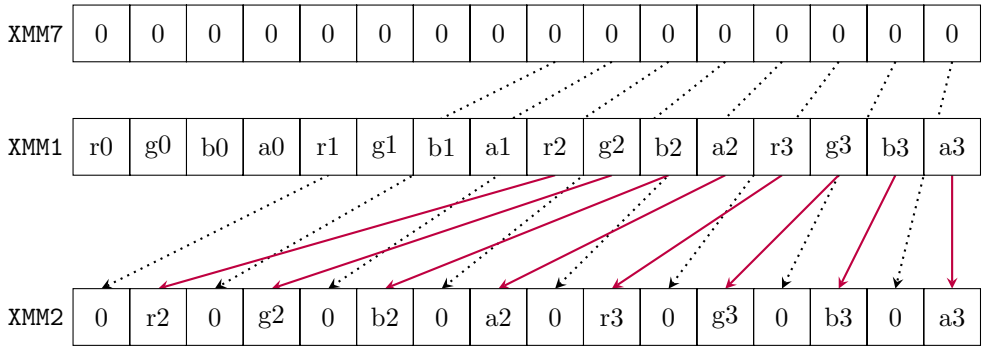


Figura 5: Desempaquetado de los bytes a words, parte baja(`PUNPCKLBW`).

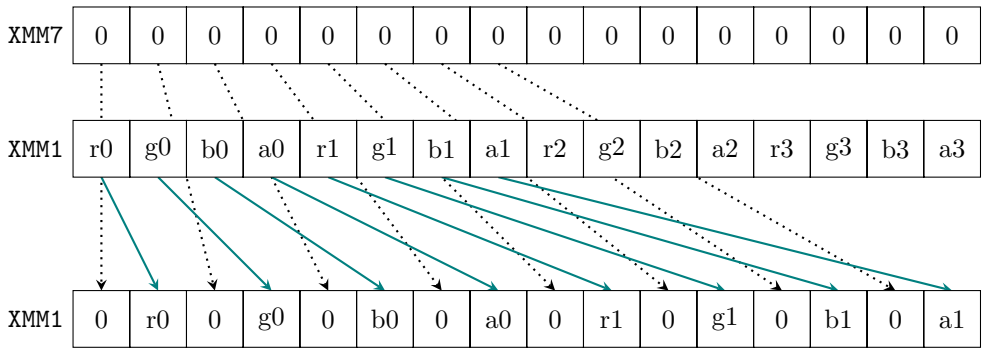


Figura 6: Desempaquetado de los bytes a words, pate alta(`PUNPCKHBW`).

- El tercer paso del ciclo es sumar el valor de cada bloque de 2×2. Para esto utilizamos la instruccion `PADDW` para sumar los valores de los píxeles de la fila actual con los valores de los píxeles correspondientes en la siguiente fila, obteniendo de esta manera en los registros `XMM1` y `XMM2` las sumas parciales de los bloques, es decir la suma de los pixeles de forma vertical.

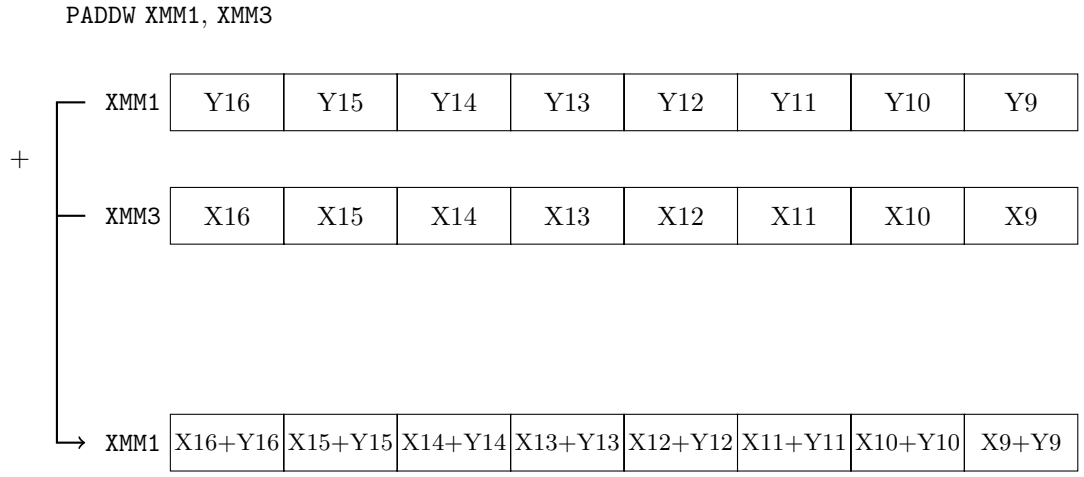


Figura 7: Suma parcial de los bloques que se están procesando.

Luego se copia el resultado en un nuevo registro(`XMM3`) y se mueve uno de ellos 64 bits hacia la derecha para poder realizar la suma de las dos sumas parciales alojadas en cada registro. Este proceso se realiza tanto para `XMM1` como para `XMM2`. Solo nos va a importar lo obtenido en la parte baja del registro.

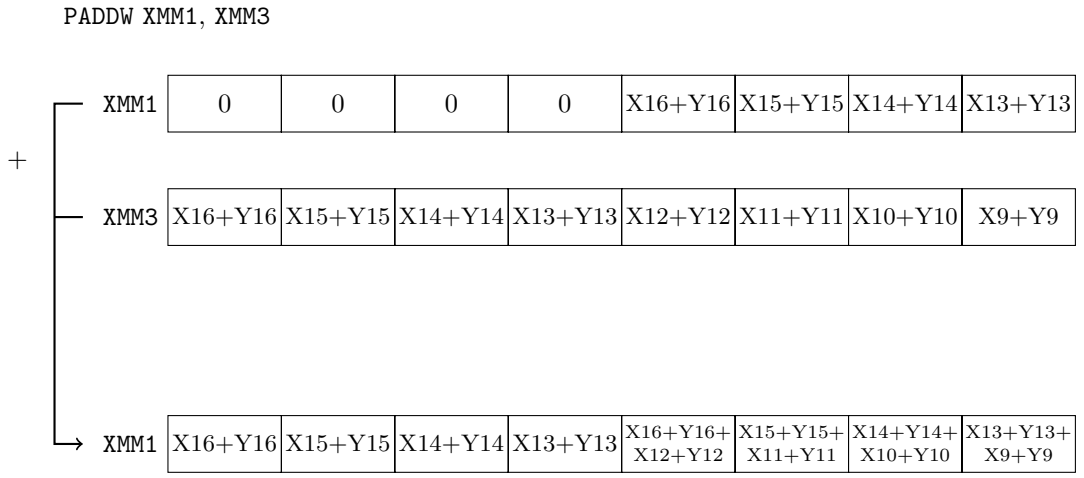


Figura 8: Suma total.

- En el cuarto paso se procede a hacer la división para obtener finalmente el promedio, para ello utilizaremos la instrucción `DIVPS` . Esta instrucción opera con punto flotante de precisión simple, por lo que debemos desempaquetar una vez más y convertir a punto flotante. Como de nuestro registro solo nos importaba la parte baja nos quedaremos solo con la parte baja del desempaquetado y utilizaremos la instrucción `CVTDQ2PS` que convierte nuestro registro con doubleword a punto flotante. Finalmente aplicaremos `DIVPS` con la ayuda de una máscara que posee 4 en formato punto flotante para cada componente, permitiéndonos obtener el promedio.
- El último paso consiste en convertir el resultado en entero con la instrucción `CVTPS2DQ`, volver a empaquetar los pixeles a bytes, utilizando `PACKUSDW` y luego `PACKUSWB`, y guardar el resultado en todo el bloque de 2×2 de la memoria correspondiente.

XMM1	0	0	0	r1	0	0	0	g1	0	0	0	b1	0	0	0	a1

XMM1	0	r1	0	g1	0	b1	0	a1	0	r1	0	g1	0	b1	0	a1

Figura 9: `PACKUSDW`

XMM2	0	r2	0	g2	0	b2	0	a2	0	r2	0	g2	0	b2	0	a2

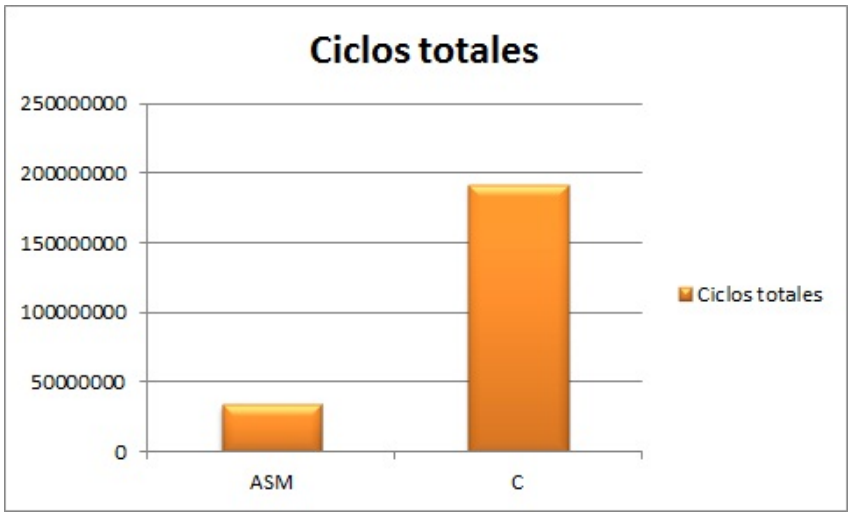
XMM1	0	r1	0	g1	0	b1	0	a1	0	r1	0	g1	0	b1	0	a1

XMM1	r1	g1	b1	a1	r1	g1	b1	a1	r2	g2	b2	a2	r2	g2	b2	a2

Figura 10: `PACKUSWB`

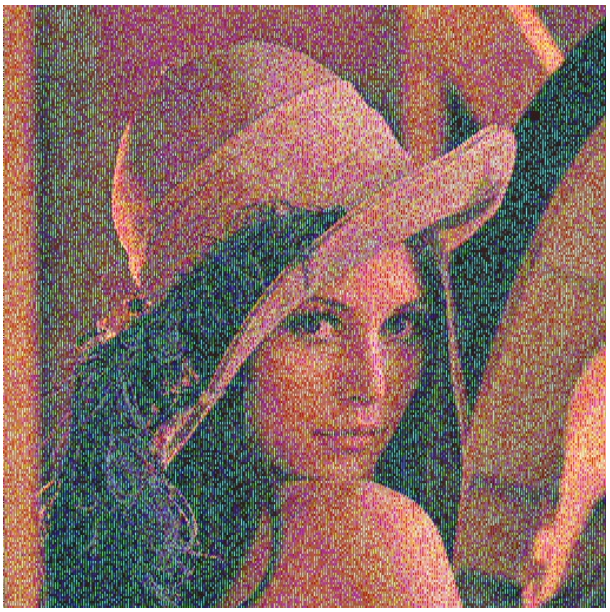
2.2.4. Comparación con la implementación C

Nuevamente compararemos nuestros algoritmos para este filtro. Como se ve en el gráfico, la implementación de ASM produce una gran mejora en la implementación de C. En esta implementación, además de la mejora de poder procesar más pixeles por iteración , contribuye la utilización de instrucciones SIMD y máscaras que agilizan el algoritmo.



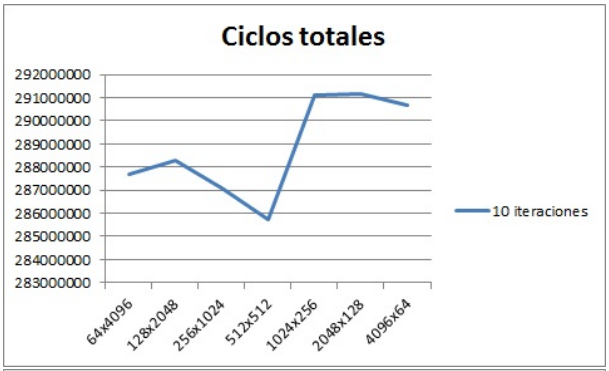
2.2.5. Comparación de precisión

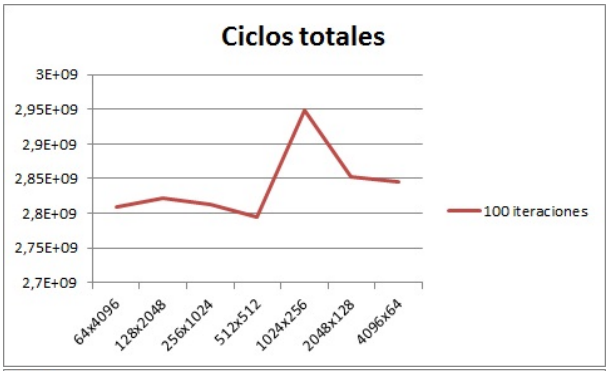
Aprovechando que esta implementación en ASM utiliza la división de valores tipo float para observar la perdida de precisión generada por la división sin realizar este pasaje. Para ello generamos el mismo filtro en ASM pero obviando el pasaje a float y las imágenes resultantes fueron las siguientes:

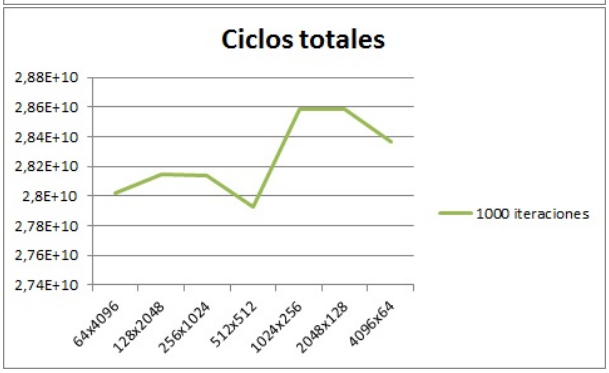


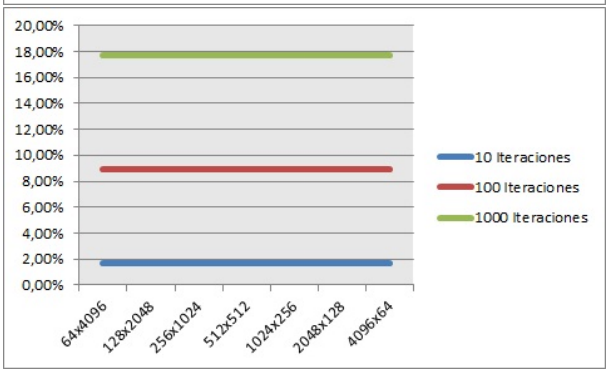
2.2.6. Comparación diferentes tamaños

Al igual que con Smalltiles se realizo una comparacion del rendimiento del algoritmo utilizando como entrada imagenes de igual cantidad de pixeles pero distribuidos de forma distinta. En el grafico de miss rate de este filtro se puede observar que para los tres casos de iteraciones se tiene un miss rate constante, por lo que se ve con mas claridad que la variacion de ciclos segun el tamaño de la imagen no depende solo del miss rate en nuestro algoritmo.









2.3. Combinar

El filtro combinar consiste en evaluar cada pixel de la imagen y asignarle un nuevo valor según una cuenta en la que participan el pixel en cuestión, su pixel simetrico correspondiente (tomando el reflejo vertical) y el alfa pasado como parametro. A continuación vamos a describir el funcionamiento del algoritmo.

2.3.1. Descripción de implementación en ensablador:

Esta implementación se compone de dos ciclos anidados. Uno de los ciclos va iterando sobre las filas y el otro itera sobre las columnas. Debido a esto, la complejidad del filtro es de O(w*h). En las siguientes secciones explicaremos el funcionamiento del ciclo principal del filtro.

2.3.2. Antes del ciclo

Antes del ciclo, configuramos 2 registros que vamos a utilizar a lo largo de todas las iteraciones:

- `XMM8` ← Este registro lo vamos a utilizar para realizar la division. Esta dividido en dwords y cada una de ella contiene el valor 255.0
- `XMM0` ← Este es el registro por el cual viene el parametro alpha. Lo vamos a configurar de forma tal que quede dividido de dwords para poder realizar las cuentas pertinentes mediante la funcion `PSHUFD`

2.3.3. Descripción del ciclo

El ciclo consta de dos etapas. La primera en la cual escribimos sobre la primera mitad de la imagen destino y la segunda en la cual escribimos sobre la segunda mitad de la misma. A continuación pasamos a detallar los pasos a seguir para una etapa:

- Lo primero que hacemos es obtener de la imagen los proximos 4 pixeles de la fila actual en el registro `XMM1` y los correspondientes 4 pixeles simetricos en `XMM2`
- Para poder realizar las operaciones sobre estos registros, reacomodamos los bytes de `XMM2` para que a cada pixel de `XMM1` le corresponda su pixel simetrico en `XMM2` utilizando la funcion `PSHUFD`
- Con el objetivo de hacer posibles las operaciones que se deben realizar para aplicar el filtro, realizamos desempaquetamientos con la idea de que cada pixel quede guardado en un registro. Para ello debemos desempaquetar dos veces de la siguiente manera: primero desempaquetamos de bytes a words utilizando las instrucciones `PUNPCKLBW` y `PUNPCKLBW` como se puede ver explicado en las figuras 5 y 6. Luego debemos desempaquetar de word a dword utilizando las instrucciones `PUNPCKLBWD` y `PUNPCKLWD`. Esto va a quedar mas claro con las siguientes figuras donde vamos a ver graficamente lo que sucede con las siguientes instrucciones:

  `PUNPCKLBWD XMM3,XMM6 :`

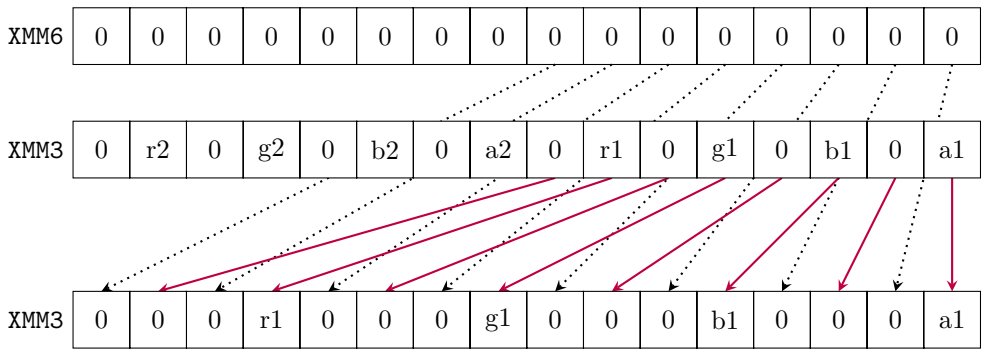


Figura 11: Desempaquetado de words a dwords, parte baja(`PUNPCKLWD`).

`PUNPCKLBWD XMM5,XMM6 :`

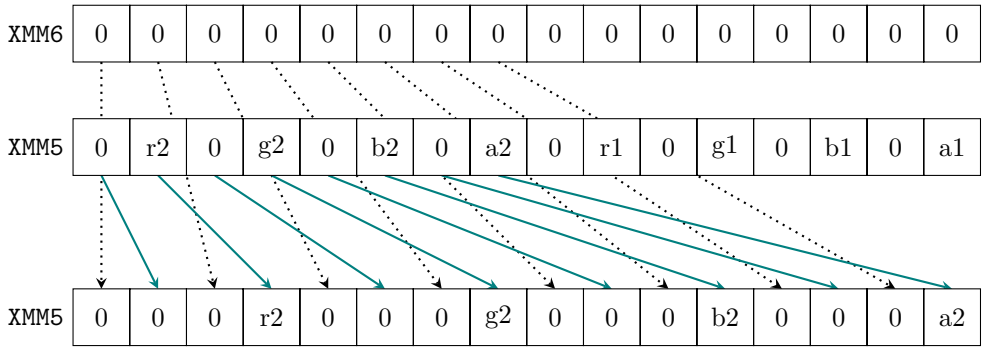


Figura 12: Desempaquetado de words a dwords, parte alta(`PUNPCKHWD`).

- El siguiente paso a realizar es la resta para la cual utilizamos la instruccion PSUBD ya que con lo previamente realizado nuestros pixeles se encuentran cada uno en un registro y este mismo esta dividido en DWORDS.
- Luego debemos efectuar la multiplicacion por alpha. Como alpha es un float, debemos convertir a floats los registros para poder realizarla. Para ello utilizamos la funcion `CVTDQ2PS`
- Continuamos realizando la division por 255.0 mediante la instrucción `DIVPS`
- El ultimo paso de la cuenta es la suma con IsrcB. Lo realizamos mediante la instruccion `PADDD`
- Ahora debemos hacer que los 4 pixeles que obtuvimos queden en un XMM para poder escribirlos en la imagen destino. Para ello debemos empaquetar. Hasta ahora tenemos cada pixel en un XMM divido en dwords. Para volver a tenerlos en bytes, primero debemos pasarlos a word utilizando la instruccion `PACKUSDW` y luego pasarlos a byte mediante la instruccion `PACKUSWB`. Veamos esto graficamente para que quede mas claro:

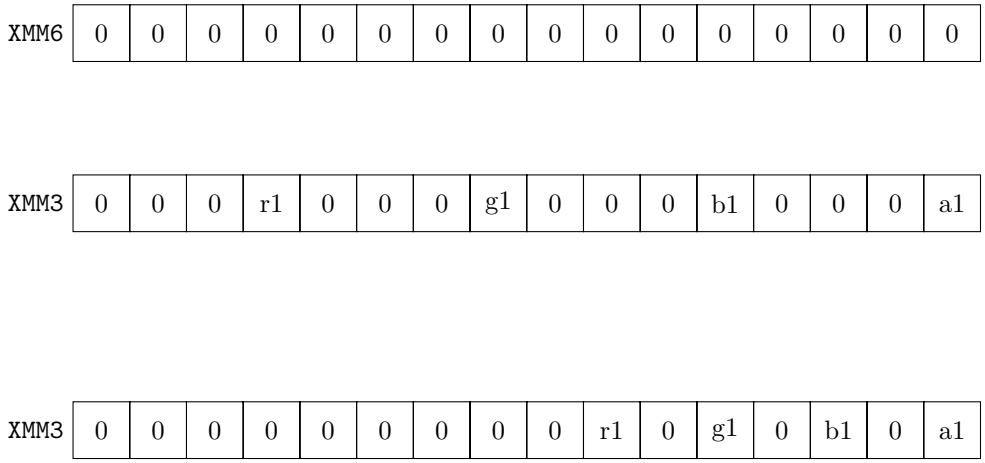


Figura 13: `PACKUSDW XMM3,XMM6`

XMM6	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

XMM3	0	0	0	0	0	0	0	0	0	r1	0	g1	0	b1	0	a1

XMM3	0	0	0	0	0	0	0	0	0	0	0	0	r1	g1	b1	a1

Figura 14: `PACKUSWB XMM3,XMM6`

- Hasta aca tenemos los 4 pixeles en 4 registros distintos. Debemos juntarlos todos en el mismo registro. Para ello acomodamos cada pixel en su registro de manera conveniente para luego realizar sumas entre registros y llegar a lo que queriamos. Para ubicar el pixel donde queremos en el registro utilizamos la instruccion `PSLLDQ` la cual realiza un shift logico. Esta instruccion toma como parametro la cantidad de bytes que se quiere desplazar. Veamoslo con el siguiente ejemplo en el cual quiero mover el cuarto pixel a su lugar correspondiente:

XMM7	0	0	0	0	0	0	0	0	0	0	0	0	r4	g4	b4	a4

XMM7	r4	g4	b4	a4	0	0	0	0	0	0	0	0	0	0	0	0

Figura 15: `PSLLDQ XMM7,12`

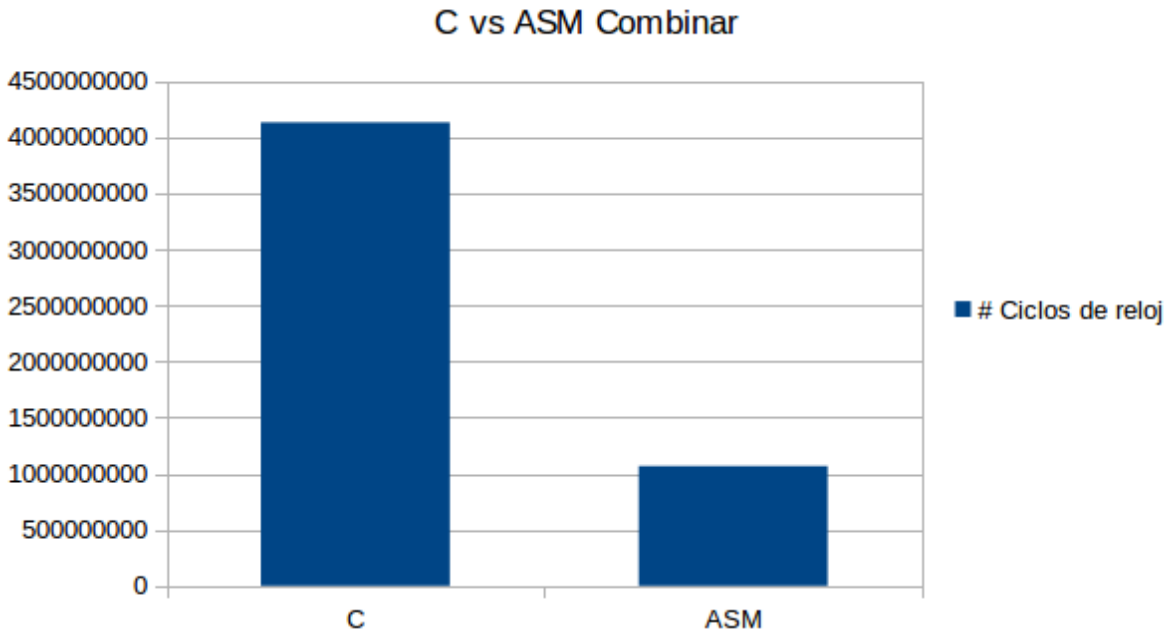
- Luego de tener a cada pixel en un XMM ubicado de forma conveniente, procedemos a realizar una serie de sumas , mediante la instruccion PADDD, con el objetivo de que los cuatro pixeles queden en un XMM.
- Por ultimo, ya teniendo los 4 pixeles en un XMM, procedemos a la escritura en la imagen destino.

2.3.4. Experimentacion

En esta seccion vamos a realizar distintos experimentos que nos van a permitir ver como se comporta nuestro filtro ante varias situaciones y poder sacar conclusiones. En todos los experimentos tuvimos la precaucion de borrar los valores atipicos (outliers) que nos puedan interferir en nuestros resultados.

Experimento 1: Comparación C vs ASM

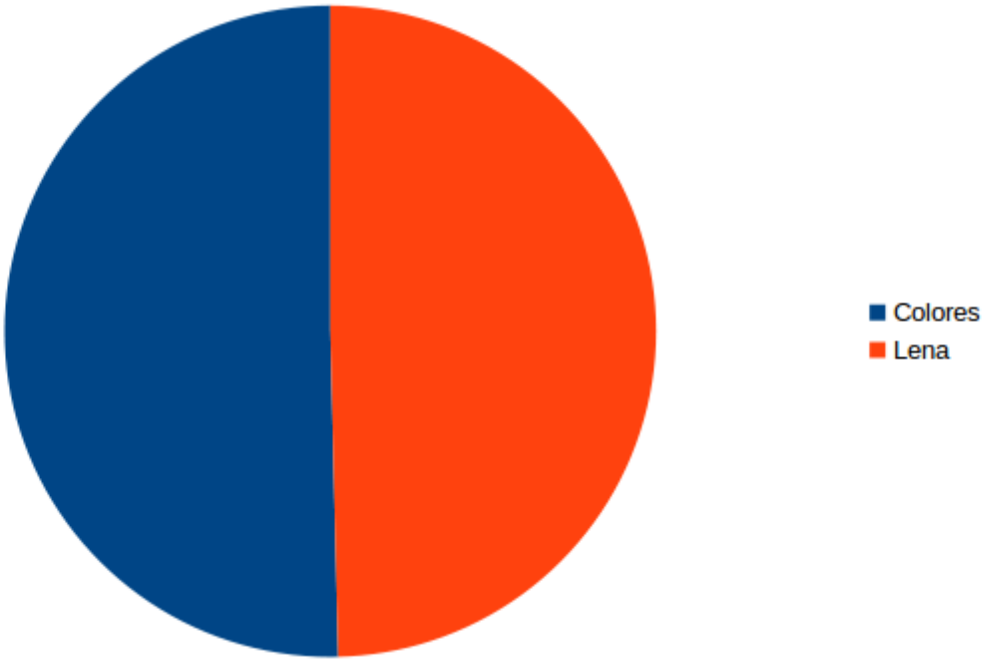
En este primer experimento vamos a comparar el rendimiento de la version C vs el rendimento de la versión ASM. Esperamos que ésta última sea mejor debido a que trabaja con más pixeles a la vez. Para realizar este experimento utilizamos la imagen de Lena de 1024x768 con 100 iteraciones. El resultado fue el siguiente :



Podemos ver en este grafico que claramente nuestra version de ASM tiene un rendimiento mucho mejor que la version de C como creiamos que iba a suceder. Para ser mas precisos, los datos nos arrojan que la version de ASM tardó un 25.84? % de lo que tardó la de C.

Experimento 2: Probar con otra imagen

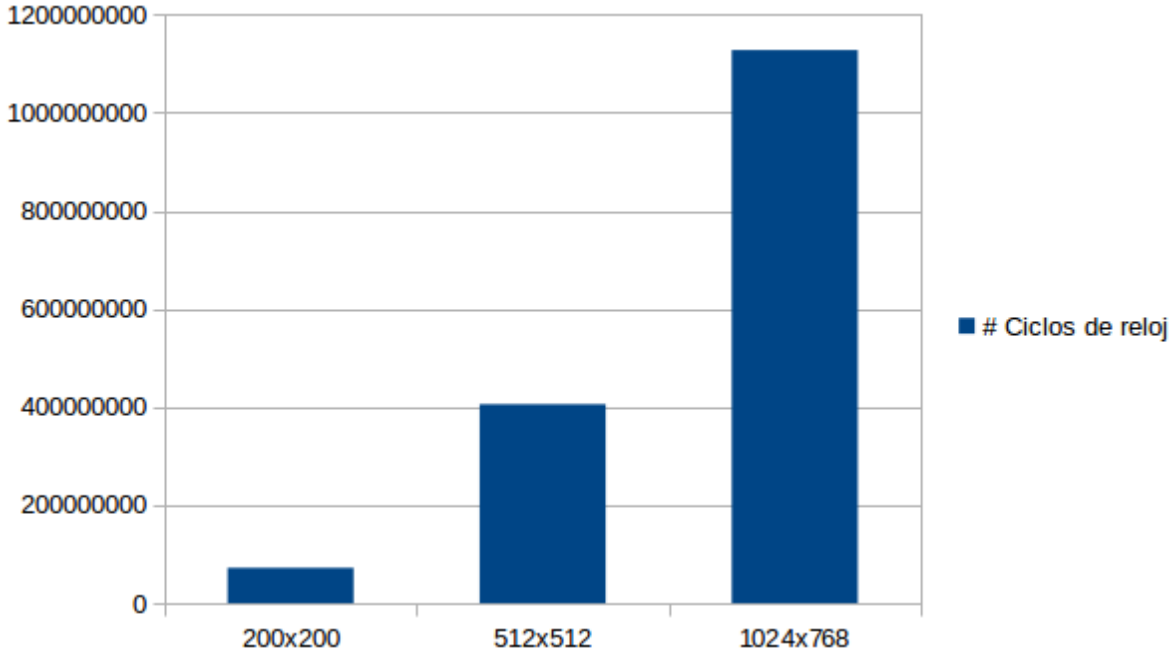
Durante todo el tp programamos usando la imagen de Lena. En este segundo experimento que proponemos vamos a ver que sucede si corremos nuestro filtro sobre una imagen completamente diferente a la de Lena. Usaremos la imagen Colores que es mucho mas çolorida"que la de Lena. Nuestra hipotesis es que el rendimiento no va a variar a pesar de correr el filtro con dos imagenes distintas. Nuestra experimentacion arrojó el siguiente resultado:



Este grafico representa la suma de los ciclos de reloj luego de correr ambas imagenes y podemos ver que la proporcion entre los tiempos insumidos por ambas imagenes es la misma y por lo tanto esto confirma lo que habiamos planteado como hipotesis. Creemos que esto sucede ya que nuestro algoritmo para este filtro no toma diferentes acciones segun los valores de los pixeles de la imagen pasada como parametro, es decir nuestro algoritmo realiza las mismas instrucciones independientemente de la imagen. Cabe aclarar que las 2 imagenes con las que realizamos este experimento tienen el mismo tamaño.

Experimento 3: Cambio de tamaño

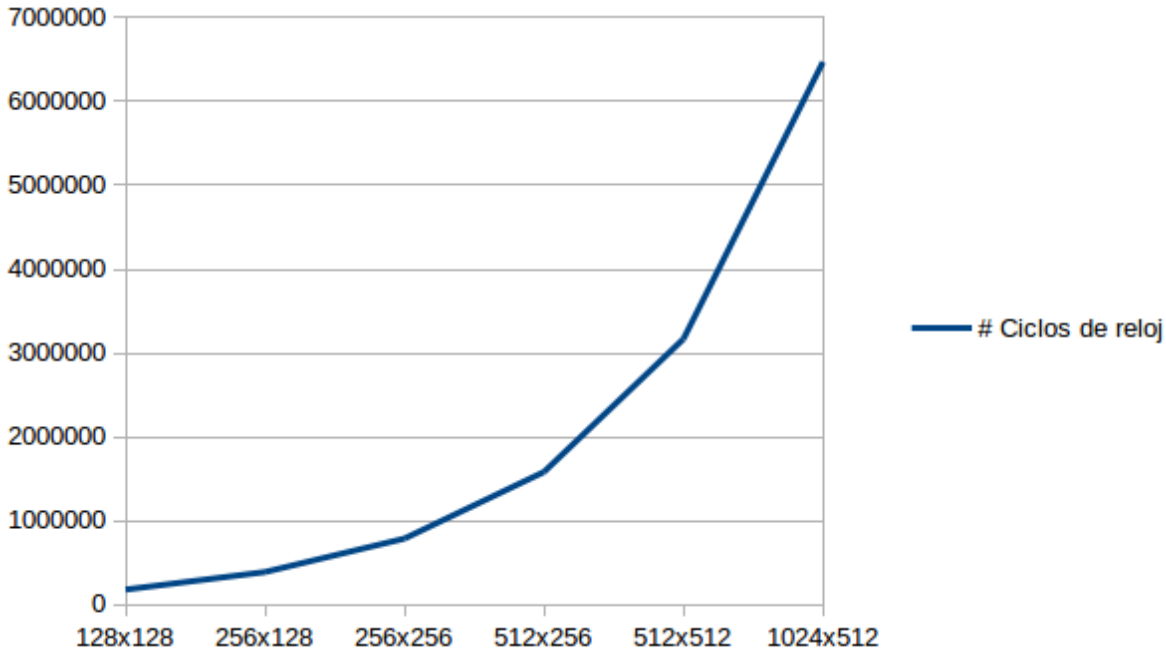
Ya vimos en el experimento 2 que cambiando la imagen el rendimiento no se ve afectado. Veamos que sucede si el tamaño de la imagen varía. Tomamos como hipotesis el filtro al aplicarse sobre la imagen de mayor tamaño va a tardar mas que la de menor dimension. Tomamos 3 tamaños diferentes de la imagen de Lena y obtuvimos el siguiente resultado:



Como era de esperarse, al aumentar el tamaño de la imagen, se ve afectado el tiempo que tarda en aplicarse el filtro. Esto se debe a que al ser mayor la dimension de la imagen, ésta misma posee una mayor cantidad de pixeles a procesar.

Experimento 4: Como varía el rendimiento segun el tamaño de la imagen

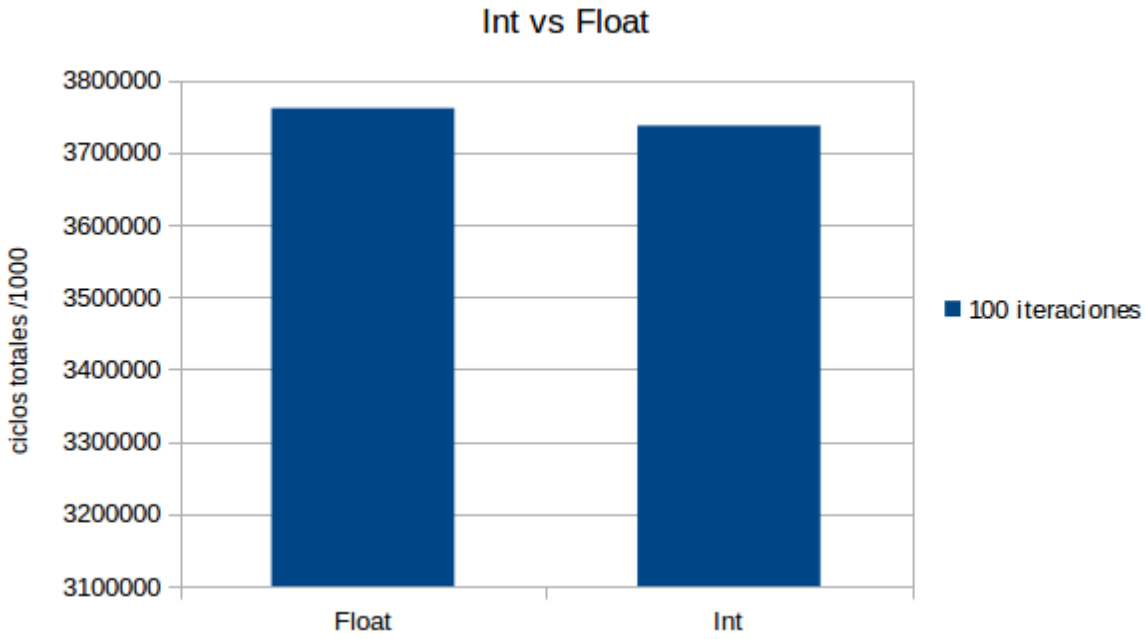
Del experimento anterior concluimos que el tamaño de la imagen sobre la cual vamos a trabajar afecta al rendimiento del filtro. La motivacion de este nuevo experimento es ver si hay alguna relacion entre la dimension de la imagen y el rendimento. Vamos a tomar como hipotesis que el rendimento se comporta de igual manera que la dimension de la imagen, es decir, por ejemplo si aumentamos al doble el tamaño de la imagen, aumenta al doble tambien la cantidad de ciclos. Para este experimento vamos a realizar varias corridas del filtro sobre imagenes que van a ir aumentando su tamaño de manera cuadratica. Obtuvimos el siguiente grafico:

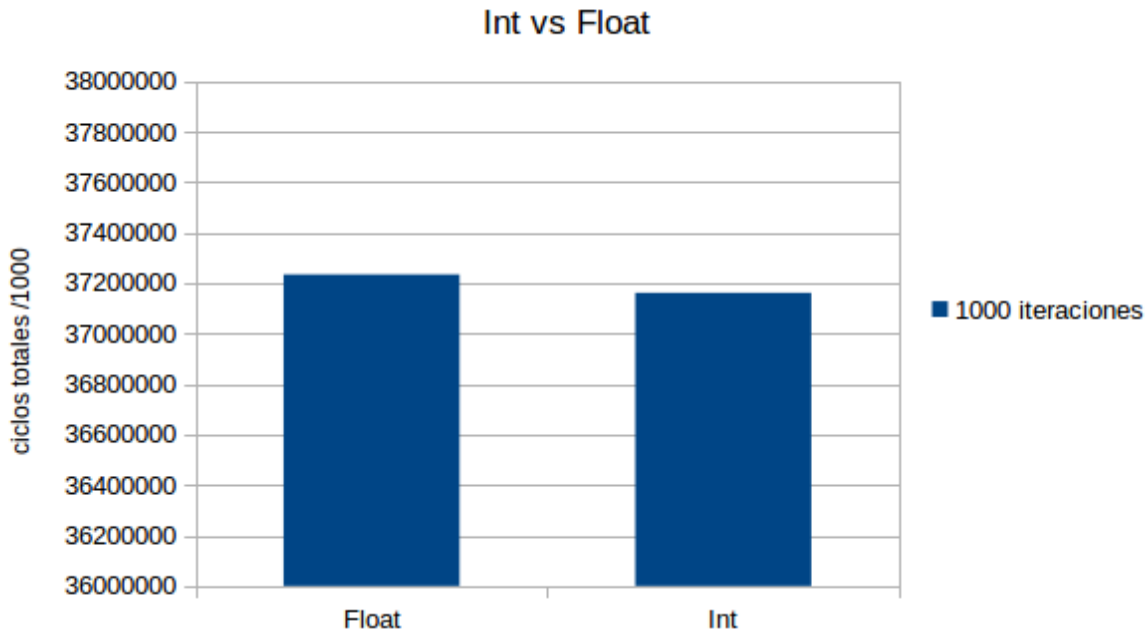


Como se puede ver en el grafico, el rendimiento varía cuadraticamente al igual que la dimension de la imagen. El resultado de este experimento reafirma nuestra hipotesis: el rendimiento del filtro depende de la dimension de la imagen. Esto se condice con lo visto anteriormente en la descripcion: la complejidad del algoritmo es O(w*h)

Experimento 5 : Int vs Float

En este experimento vamos a comparar el rendimiento del filtro en la version de C utlizando enteros y floats. Esperamos que el filtro se ejecute mas rapidamente en su implementacion con enteros. Los resultados fueron volcados sobre los siguientes graficos

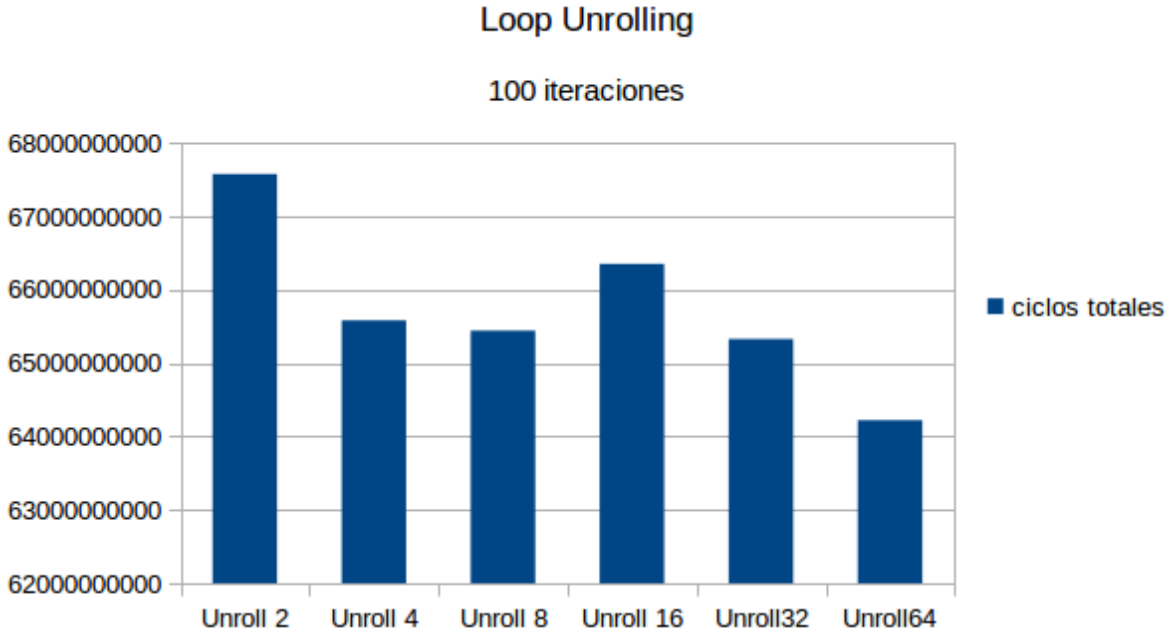


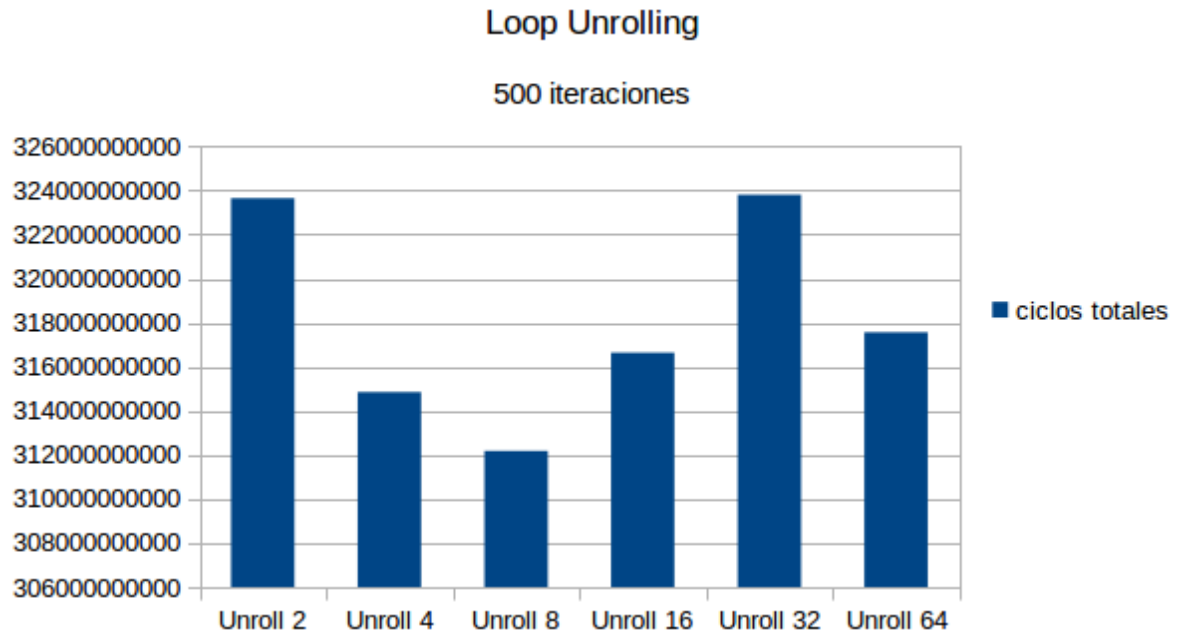


Como se puede ver en los graficos, la version implementada con enteros tiene un leve mejor rendimiento comparada con la implementada con floats. La imagen generada es muy similar a la buscada (de hecho pasa los tests). Por lo tanto, al tener un mejor rendimiento y no tener diferencias significativas con la imagen buscada, la implementacion con enteros es mas favorable.

Experimento 6 : Loop unrolling

El proposito de este experimento es ver como funciona nuestro filtro al aplicarle la tecnica de loop unrolling. El objetivo de esta tecnica es mejorar el rendimiento de un programa pero con la condicion de aumentar su tamaño binario. Se trata de desenrollar los bucles realizando así menor cantidad de iteraciones de los mismos evitando las comparaciones de fin de ciclo. Realizaremos mediciones con distintas cantidades de iteraciones desenroscadas y esperaremos que el rendimiento vaya aumentando a medida que crecen las iteraciones desenroscadas. Los resultados obtenidos fueron los siguientes :





En ambos graficos podemos ver que hasta desenrollar 8 iteraciones, el resultados fue el esperado : el rendimiento fue mejorando. Para mayor cantidad de iteraciones desenrolladas, el comportamiento no es el esperado. Creemos que esto puede ocurrir debido a las instrucciones en vuelo. Luego de desenrollar 8 iteraciones, es probable que se tenga que desechar instrucciones en vuelo para volver a cargar nuevas.

2.4. Colorizar

El filtro colorizar modifica cada pixel de una imagen color de acuerdo a una ecuación que depende de los máximos valores de los canales R, G y B entre los píxeles del cuadrado de lado 3 formado por el pixel original y sus 8 vecinos inmediatos.

El problema principal a resolver es minimizar la cantidad de accesos a memoria y paralelizar lo más posible las comparaciones necesarias para hallar el máximo valor de cada canal en el cuadrado de lado 3 correspondiente a cada pixel.

2.4.1. Descripción del ciclo

Para cada pixel, dividimos el ciclo en cuatro etapas:

1. **Lectura de píxeles vecinos:** Cargamos los píxeles de cada fila del cuadrado en `XMM1`, `XMM2` y `XMM3`.
2. **Búsqueda de máximos por canal:** Separamos cada pixel por canal, y comparamos sucesivamente los valores de cada canal entre sí para hallar los máximos.
3. **Hallar $\Phi_{R,G,B}$:** Hallamos los valores de Φ_R, Φ_G y Φ_B utilizando los máximos obtenidos en la etapa anterior.
4. **Escritura de pixel destino:** Computamos el valor del pixel destino y lo escribimos en la imagen.

Veamos a continuación cada etapa en detalle.

2.4.2. Lectura de píxeles vecinos

Es importante aclarar que en la implementación de ASM, para cada iteración del ciclo vamos procesar y luego escribir en memoria dos pixeles. Para hacerlo traeremos de memoria, de la imagen fuente, en cada iteración a estos dos pixeles a procesar y a los 8 pixeles que los rodean.

Como los dos pixeles con los que vamos a trabajar son continuos, los que los rodean se repiten, entonces en total traeremos doce pixeles en tres registros que se ordenaran de la siguiente forma para cada pixel $x_{[i][j]}$ a procesar.

`MOVDQU XMM1`, $src[i-1][filaInferior]$
`MOVDQU XMM2`, $src[i-1][filaActual]$
`MOVDQU XMM3`, $src[i-1][filaSuperior]$

Contenido de los registros `XMM1`, `XMM2` y `XMM3` en este instante:

$T4$	$R4$	$G4$	$B4$	$T3$	$R3$	$G3$	$B3$	$T2$	$R2$	$G2$	$B2$	$T1$	$R1$	$G1$	$B1$

Figura 16: Contenido de cada registro luego de leer los píxeles vecinos.

2.4.3. Búsqueda de máximos por canal

El siguiente paso es comparar los valores de cada canal entre sí para hallar los máximos.

Lo hacemos buscando los máximos byte a byte entre `XMM1` y `XMM2` y luego entre `XMM1` y `XMM3` de manera de obtener los máximos por canal "columna a columna":

```
PMAXUB XMM1, XMM2
PMAXUB XMM1, XMM3
```

Así, en `XMM1` quedan contenidos los máximos por columnas.

$MaxColumna4$	$MaxColumna3$	$MaxColumna2$	$MaxColumna1$

Figura 17: Contenido de xmm1.

El paso siguiente es obtener los máximos que rodean a los pixeles 1 y 2 a procesar. Estos se obtienen con las instrucciones `PMAXUB` aplicadas a `XMM1` shifteado de forma tal que primero comparemos los tres primeros pixeles (maximos que rodean a pixel1) y luego los tres ultimos pixeles (maximos que rodean a pixel2).

```
MOVDQU XMM2, XMM1                                          {XMM2 = XMM1}
PSRLDQ XMM2, 4                          {XMM2 = | * |MaxColumna4|MaxColumna3|MaxColumna2|}
PMAXUB XMM1, XMM2                                          {Maximosparciales}
PSRLDQ XMM2, 4                                {XMM2 = | * | * |MaxColumna4|MaxColumna3|}
PMAXUB XMM1, XMM2
```

Luego de estas instrucciones, quedan en el registro `XMM1` guardados los máximos de los pixeles 1 y 2 respectivamente

*	*	$MaximosPixel2$	$MaximosPixel1$

Figura 18: Contenido de xmm1.

En este momento tenemos los maximos correspondientes al pixel 1 y 2 en byte. Luego de esto, podemos extender los contenidos de las componentes a dword por medio de dos `PUNPCKLBW` para realizar las comparaciones necesarias y obtener los Φ_{color}.

2.4.4. Hallar $\Phi_{R,G,B}$

Procedemos a computar Φ_R, Φ_G y Φ_B de forma secuencial, por medio de condicionales y comparaciones entre registros de 32bits. Finalmente convertimos a Floats y empaquetamos en `XMM1` los valores hallados como se indica a continuación:

`XMM1`	*	Φ_R	Φ_G	Φ_B

Figura 19: Empaquetado de los valores $\Phi_{R,G,B}$ computados.

Empaquetamos $\Phi_{R,G,B}$ de esta manera para facilitar las comparaciones que realizaremos en la siguiente y última etapa.

2.4.5. Escritura de pixel destino

Antes de escribir el pixel destino, debemos hallar el valor de cada canal. Recordemos las ecuaciones:

$$I_{dst_R}(i,j) = min(255, \Phi_R * I_{src_R}(i,j))$$
$$I_{dst_G}(i,j) = min(255, \Phi_G * I_{src_G}(i,j))$$
$$I_{dst_B}(i,j) = min(255, \Phi_B * I_{src_B}(i,j))$$

Obviando las instrucciones en las que muevo y convierto las componentes en bytes de los dos pixeles a procesar en `XMM3` a dwords, podemos proceder a hallar $I_{src1_{R,G,B}}(i,j)$ y $I_{src2_{R,G,B}}(i,j)$:

```
MOVDQU XMM3, src[j][i]                                     {leo el pixel original}
CVTDQ2PS XMM3, xmm3                                        {convierto cada canal a float}
```

Para facilitar la lectura, voy a explicar lo que sucede en sólo uno de los dos registros que tienen los Φ . En este instante `XMM3` contiene el valor de cada canal del pixel original convertido a punto flotante como se ilustra a continuación:

`XMM3`	*	I_{src1_R}	I_{src1_G}	I_{src1_B}

Figura 20: Canales R, G, B convertidos a punto flotante.

Multiplicamos cada canal por el valor Φ correspondiente y buscamos el mínimo entre este producto y 255,. Utilizamos una comparacion entre un registro con 255 cuatro veces y el que contiene la multiplicación de $\Phi * color$. Con la máscara que se genera, sabemos en que lugar poner un 255 y en cual otro, $\Phi * color$. Tras las comparaciones, hacemos una conversion de las componentes a entero y luego empaquetamos las componentes de ambos registros (recordemos que estabamos procesando dos pixeles) en `XMM1`. Primero la conversión sería de 4Floats a 4Enteros; Luego empaquetamos dos veces y el contenido de de `XMM1` es ahora:

`XMM1`	*	*	I_{dst2}	I_{dst2}

Figura 21: Valores del pixel destino por canal.

Para terminar, extraemos la parte baja de XMM1, I_{dst1} y I_{dst2} en la imagen destino, procesando así en un ciclo, dos pixeles.

2.4.6. Comparación con la implementación C

Escencialmente, la versión C de este filtro realiza la misma secuencia de operaciones, pero accediendo a memoria de a un byte por vez, y realizando las comparaciones sin paralelismo. Esto significa que por cada iteración se realizan:

- 9 accesos a memoria para obtener cada píxeles vecinos y original,
- 1 accesos a memoria para escribir el pixel destino, y
- 24 comparaciones para hallar $max_{R,G,B}$.

Esto es un total de 10 (20) accesos a memoria y 24 (48) instrucciones de comparacion para procesar 1 (2) pixel.

En contraste, la implementación assembler realiza:

- 3 accesos a memoria para obtener los píxeles vecinos y original,
- 1 acceso a memoria para escribir el pixel destino, y
- 4 comparaciones para hallar $max_{R,G,B}$.

Es decir, un total de 4 accesos a memoria y 4 comparaciones para procesar 2 pixeles.

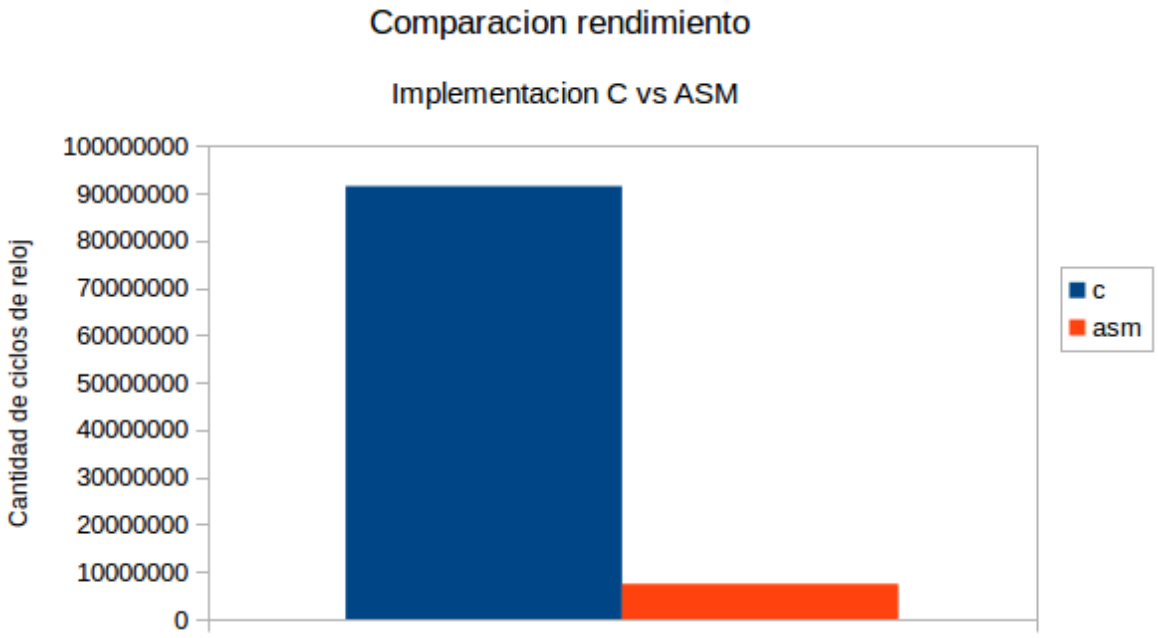
Expresado de forma relativa, la implementación assembler realiza el 20 % de accesos a memoria y el 8,3 % de comparaciones que su contraparte en lenguaje C.

2.4.7. Rendimiento

Observamos las siguientes cantidades promedio de ciclos al realizar 1000 iteraciones de ambas implementaciones con una imagen cuadrada de lado 512 y parámetro $\alpha = 0{,}4$:

Medición	Implementación C	Implementación assembler	Relación
Ciclos	91502288	7426924	8,11 %

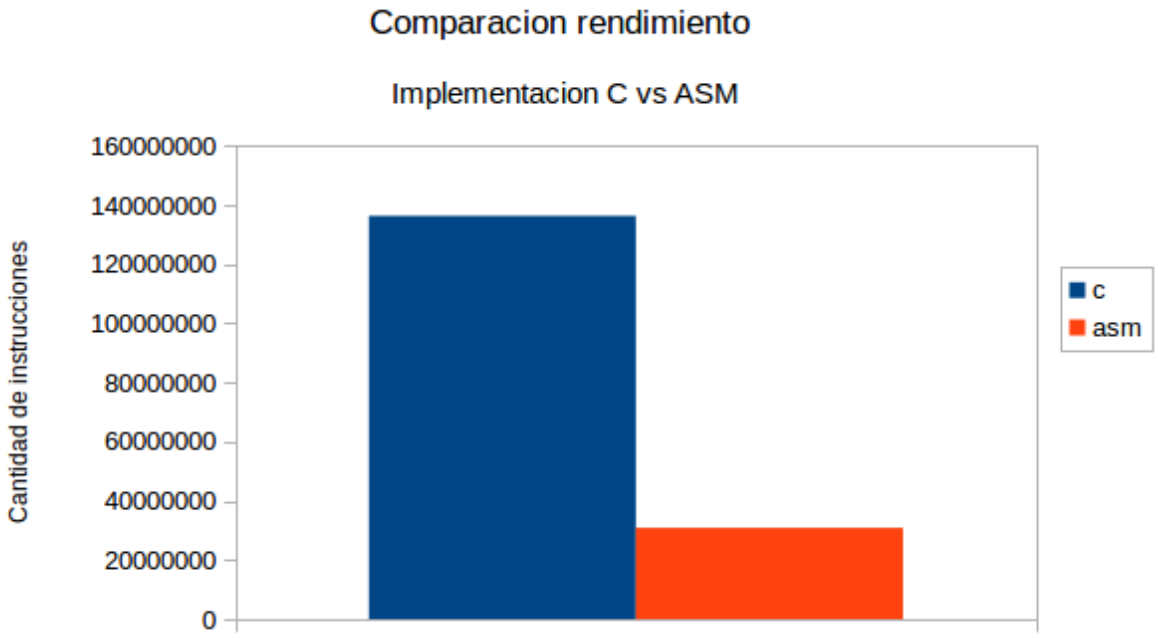
Veamoslo gráficamente:



Podemos ver que la versión de asm procesa mucho más rápido la imagen.
Utilizando la herramienta cachegrind de valgrind, vemos el total de instrucciones ejecutadas:

Medición	Implementación C	Implementación assembler	Relación
Instrucciones	136,355,680	30,878,898	22, 6 %

Gráficamente:

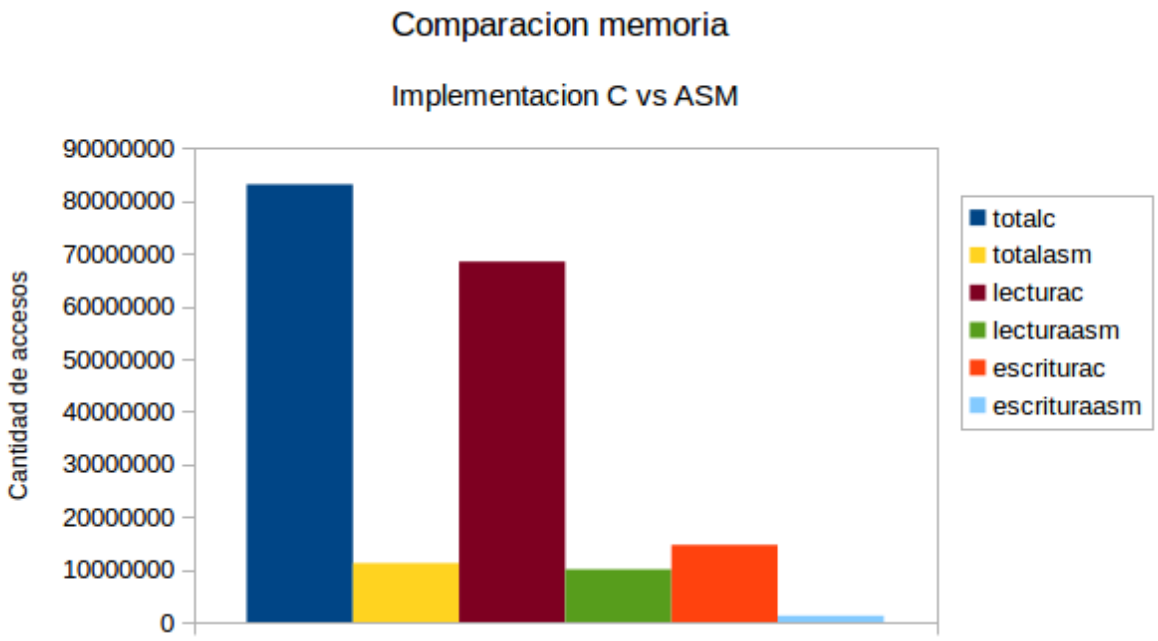


Notamos que ASM ejecuta menos instrucciones además de terminar más rápido, procesando la misma imagen.

Veamos ahora los accesos a memoria, totales y por lectura o escritura:

Medición	Implementación C	Implementación assembler	Relación
Lectura	68,475,770	10,006,753	14,6 %
Escritura	14,644,443	1,196,220	8,2 %
Accesos totales	83,120,213	11,202,973	13,5 %
Relacion L/E	21,3 %	11,9 %	*

Veamoslo gráficamente:



Analicemos estos números:

ASM accede un 13,5 % de veces que C a memoria, contra un 20 % de veces que teníamos estimado, suponemos que esto tiene que ver con los datos que guarda el procesador en la caché, lo que optimiza los accesos.

ASM escribe casi un 10 % de veces que las que C escribe y lee de memoria casi un 15 % de su contraparte, volviendolo casi orden de magnitud más eficiente.

Finalmente, la relación Lectura/Escritura es de un 21,3 % vs un 11,9 %. Creemos que esto se debe a que nuestra implementación de ASM procesa el doble de pixeles por acceso a memoria que su contraparte en C, lo que hace que la cantidad de accesos por escritura sea de casi la mitad.

2.5. Rotar

El filtro rotar consiste en reubicar las componentes de cada pixel en la imagen destino de la siguiente forma:

Pixel	T	R	G	B

Figura 22: Pixel i previo a aplicacion de Rotar.

Pixel	T	B	R	G

Figura 23: Pixel i posterior a aplicacion de Rotar.

Al ser un filtro bastante simple no se presta mucho para el analisis, ya que solo hay que reubicar bytes mediante un shuffle.

2.5.1. Descripción del ciclo

Lo único interesante de la implementacion en ASM es que podemos procesar 4 pixeles por iteracion, ya que por cada lectura de memoria, obtenemos en cada registro XMM 4 de los mismos.

`XMM1`	$T4$	$R4$	$G4$	$B4$	$T3$	$R3$	$G3$	$B3$	$T2$	$R2$	$G2$	$B2$	$T1$	$R1$	$G1$	$B1$

Figura 24: XMM1 previo a aplicacion de Rotar.

`XMM1`	$T4$	$B4$	$R4$	$G4$	$T3$	$B3$	$R3$	$G3$	$T2$	$B2$	$R2$	$G2$	$T1$	$B1$	$R1$	$G1$

Figura 25: XMM1 posterior a aplicacion de Rotar.

2.5.2. Comparación con la implementación C

Nuevamente, la única diferencia está en la cantidad de pixeles procesador por ciclo
Operaciones realizadas en la implementación en C para 16 bytes:

- Se realizan $4 * 2 = 8$ accesos a memoria(en cuatro iteraciones), 4 para la lectura de un pixel determinado en la imagen fuente y otros 4 para la escritura en la imagen destino.
- Se realizan 3 reubicaciones de datos

Operaciones realizadas en la implementación en ASM para 16 bytes:

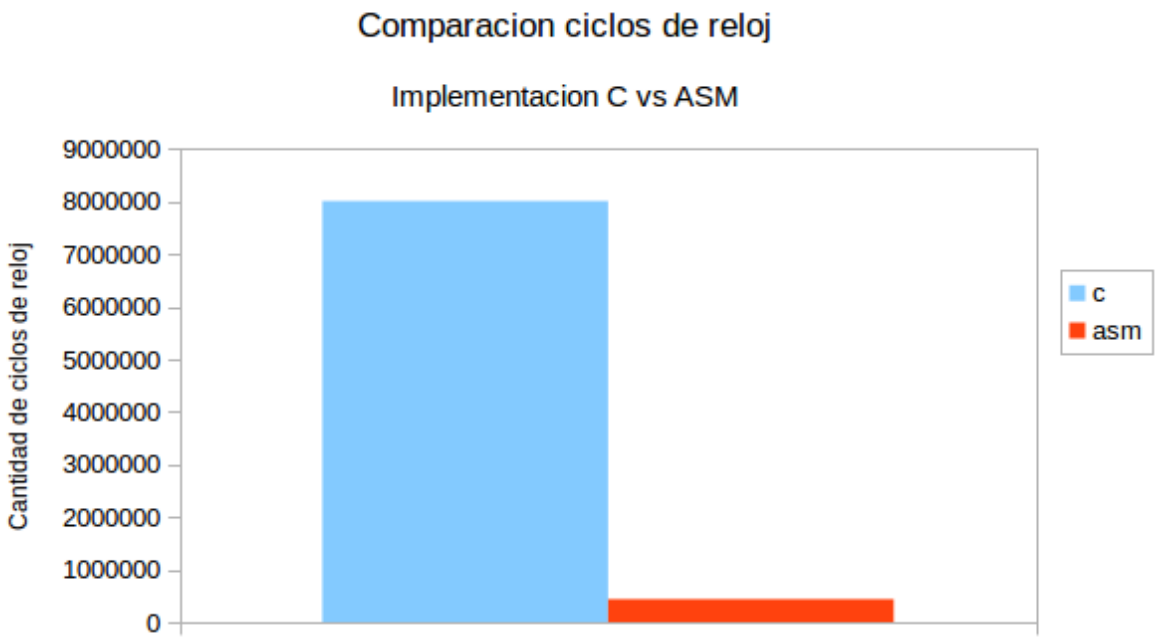
- Se realiza 1 acceso a memoria para la lectura de 4 pixeles determinados en la imagen fuente y 1 acceso utilizando para la escritura de los mismos 4 en la imagen destino.
- Se realiza 1 operacion para reubicar datos.

2.5.3. Rendimiento

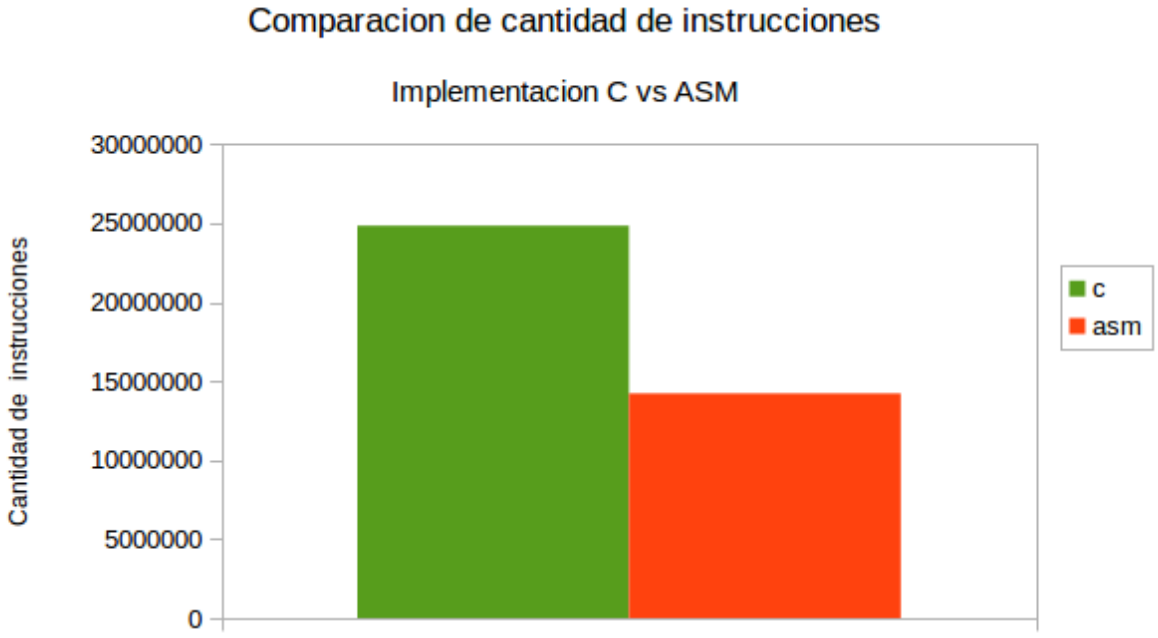
Observamos en promedio las siguientes cantidades de ciclos de reloj al realizar 1000 iteraciones de ambas implementaciones con una imagen cuadrada de lado 512.

Medición	Implementación C	Implementación assembler	Relación
Ciclos	8007735	434925	5,4 %

Veamoslo graficamente:



Vemos que la implementación de ASM finaliza en menos ciclos.

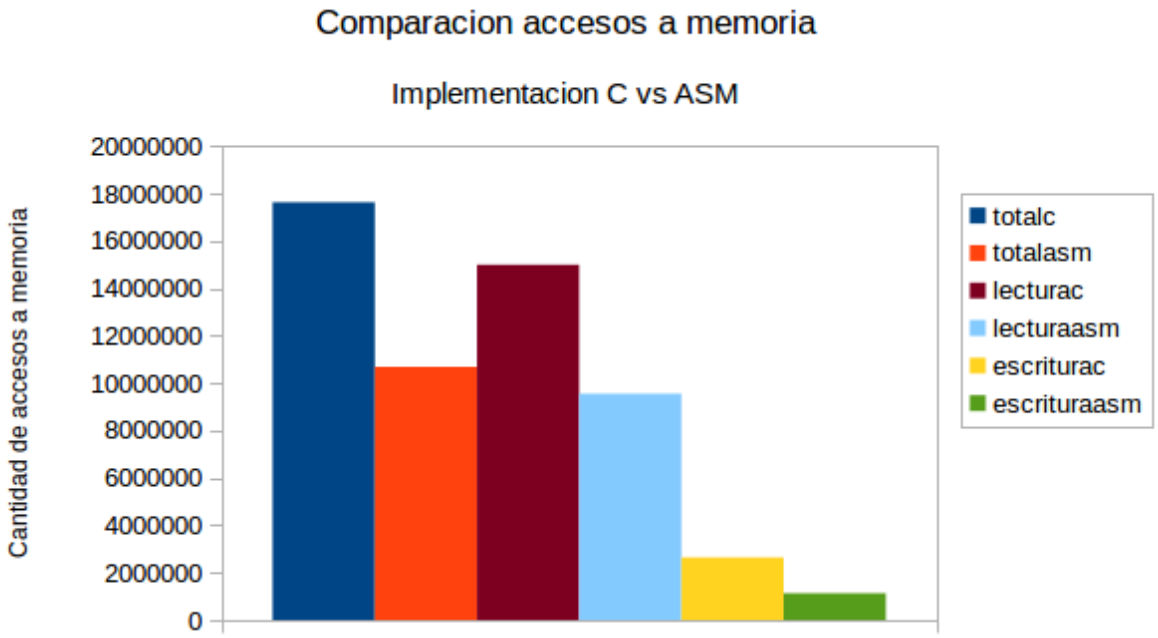


La cantidad de instrucciones que realiza la implementación en ASM es alrededor de un 60 % de la cantidad de C.

Ahora veamos los accesos a memoria:

Medición	Implementación C	Implementación assembler	Relación
Lectura	14,993,841	9,551,738	63,7 %
Escritura	2,639,391	1,131,501	42,8 %
Total	17,633,232	10,683,239	60,6 %
Relación L/E	5.7 %	8.4 %	*

Gráficamente:



Analizando los datos vemos que contra nuestra estimación de acceder un cuarto de las veces a memoria, ASM accede un 63.7 % del total en C veces. También ocurre algo raro con la escritura, creíamos que los accesos también serían un cuarto del total en C, pero no fue así. ¿Quizás esto se explique porque usamos una máscara para usar el shuffle en el código de ASM? Queda una duda sin resolver. Además, la relación de Escritura/ Lectura es más alta en ASM. Tiene sentido pues si bien los resultados no son los esperados, seguimos accediendo muchas menos veces en ASM para leer que en C. Lo que disminuye el numerador en la división y hace que el porcentaje de escrituras sea más alto en relacion a las lecturas.

3. Conclusión

A lo largo de este trabajo obtuvimos mejoras de rendimiento significativas de manera consistente al hacer uso de instrucciones SIMD.

Identificamos dos posibles focos de aplicación para este juego de instrucciones: la paralelización de cómputos y el acceso a memoria de a bloques de datos contiguos. Observamos que en los casos que se logró acceder a más de un dato en memoria a la vez, obtuvimos las ganancias de rendimiento más grandes (un ejemplo de esto es Colorizar).

En aquellos casos donde esto no fue posible (por ejemplo en la implementación de Rotar), aún así obtuvimos grandes mejoras de rendimiento al paralelizar el procesamiento de los datos leídos en forma secuencial, aunque más pequeñas en comparación con los casos en los que esto sí fue posible.

Además, observamos que el código assembler generado por el compilador gcc (con nivel de optimización por defecto) produce una cantidad adicional de accesos a memoria al guardar en la pila los parámetros de las funciones y variables locales, que producen la penalidad de rendimiento correspondiente. Esto es especialmente evidente cuando el código C hace muchas llamadas a funciones, como en el caso de los filtros que invocan funciones auxiliares en el ciclo que itera sobre todos los píxeles de la imagen.