DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Filtros implementados en SIMD

Organización del Computador II
Primer Cuatrimestre de 2020

Integrante	LU	Correo electrónico
Gonzalo Consoli	595/18	gonzaloconsoli@hotmail.com
Julián Recalde Campos	502/17	recaldej@hotmail.com.ar

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 **3**

2. Descripción de Filtros **3**
2.1. Filtro Ocultar ASM 3
2.2. Filtro Descubrir ASM 5
2.3. Filtro Zigzag ASM 6

3. Hipótesis y Experimentación del Rendimiento **10**
3.1. Observaciones y detalles de la experimentación 10
3.2. Filtro Zigzag ASM con división alternativa con enteros 11
3.3. Hipótesis 11
3.4. Experimentación y Comparación 12

4. Comparación Assembler vs C **12**
4.1. Filtro Ocultar ASM vs Filtro Ocultar en C 12
4.2. Conclusión comparación ASM V C en Ocultar 13
4.3. Filtro Descubrir ASM vs Descubrir Ocultar en C 13
4.4. Conclusión comparación ASM V C en Descubrir 14
4.5. Filtro Zigzag ASM vs Filtro Zigzag en C 14
4.6. Conclusión comparación ASM V C en Zigzag 15

5. Experimentos **15**
5.1. ZIGZAG implementado con enteros vs Zigzag implementado con float 15
5.2. Conclusión experimento ZIGZAG implementado con enteros 16
5.3. Leer mascaras de memoria vs registros 17
5.4. Conclusión experimento Ocultar y Descubrir con mascaras de memoria vs registros 17

6. Conclusión Global **18**

1. Introducción

En este informe vamos a presentar implementaciones de filtros para imágenes, estos filtros que vamos a estar implementado son Zigzag, Ocultar, Descubrir, donde el objetivo de cada filtro es el siguiente, Zigzag genera un efecto ondulado (como un zigzag) en la imagen, mientras que ocultar esconde una imagen dentro de otra y descubrir permite recuperar la imagen escondida.

Para estos filtros nosotros vamos a comparar distintas implementaciones, con el fin de ver cual implementación logra una mejor performance, si lográsemos una implementación lo suficientemente performante, estas podrían incluso calcularse en tiempo real, en algunos contextos como los videojuegos donde las imágenes deben calcularse a tiempo real esto podría ser útil si alguien por ejemplo utilizando zigzag, quiere simular una sensación de mareo, o algún otro efecto implementado con las mismas tecnologías o en algún otro contexto que requiera una ejecución muy eficiente.

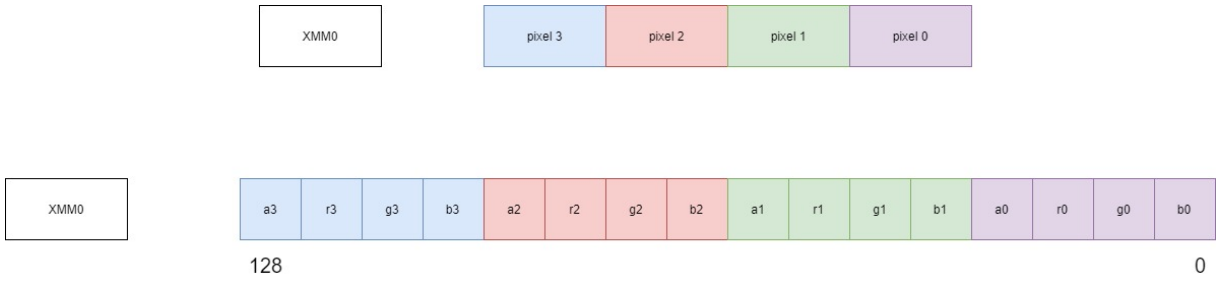
Ahora presentaremos la idea de como están implementados estos filtros en su versión de Assembler utilizando operaciones de SIMD

2. Descripción de Filtros

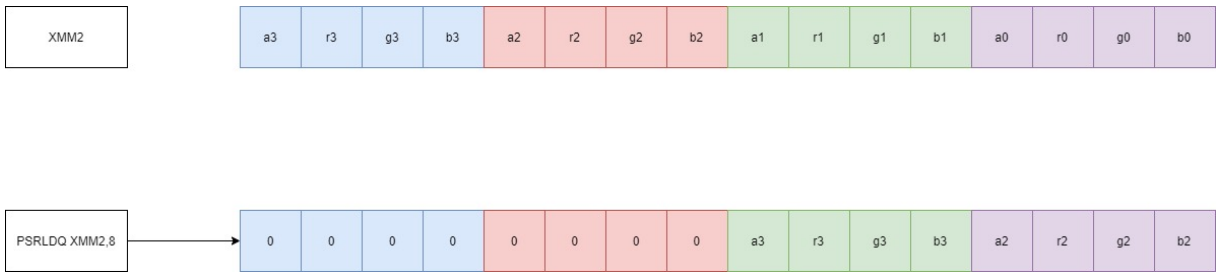
2.1. Filtro Ocultar ASM

Implementación del filtro Ocultar en Assembler Procederemos a explicar nuestra implementación del filtro ocultar.

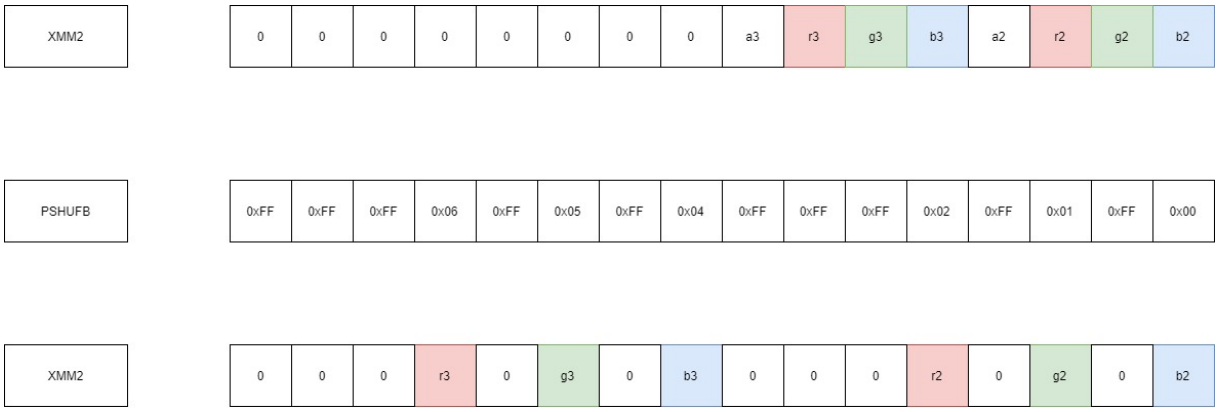
Nuestra implementación en assembler consiste en lo siguiente: Tenemos un ciclo donde vamos a leer 16 bytes de memoria de src2, o sea estamos tomando 4 píxeles y los guardamos en xmm0. Con estos 4 píxeles que vamos a ir levantando periódicamente lo que queremos hacer es transformar estos píxeles que están en formato RGB a blanco y negro. Esto lo realizamos sumando la componente blue, con 2 veces la green mas la red. Una vez que tenemos el píxel transformado a blanco y negro podemos continuar el proceso de ocultar la imagen, ya que lo que vamos a hacer es agarrar bits del byte de la imagen en blanco y negro y hacer un xor con src-mirror para luego guardarlo en los últimos bits mas significativos de la imagen destino.



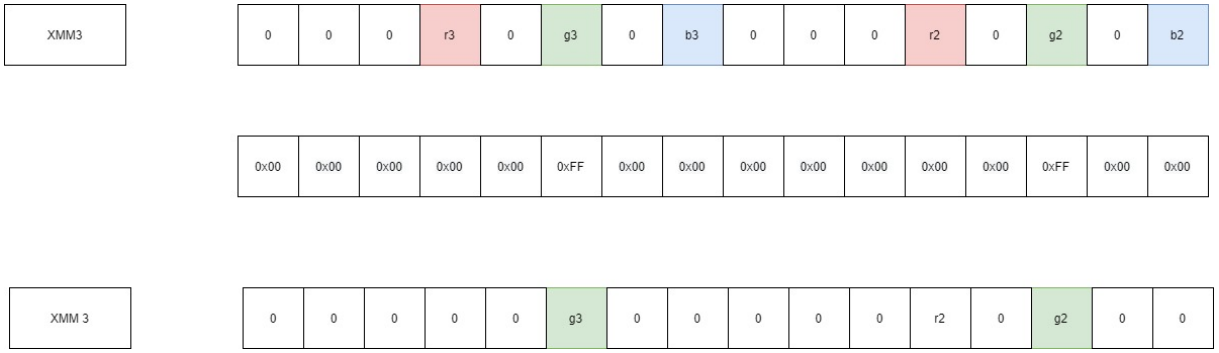
Luego hacemos una copia de ese registro a xmm2 y shifteamos 8 bytes a la derecha.



Luego hacemos un pshufb del registro xmm1 y una mascara. Lo que hace esto es extender cada componente del píxel con un byte en 0 y poner la componente a en 0 también.

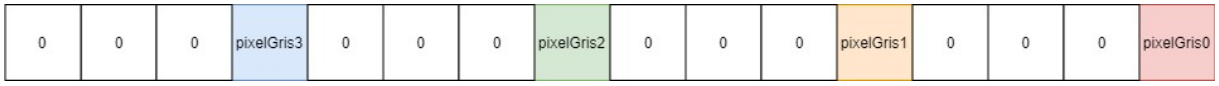


Luego a cada uno de esos registros les hacemos un AND con una mascara para solo quedarnos con la componente Green y poder luego sumar estos registros.



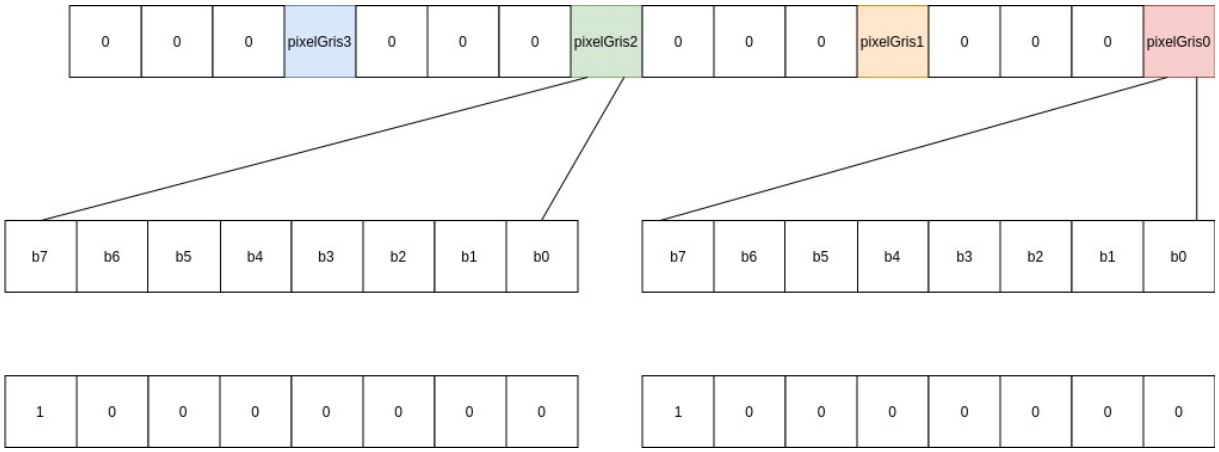
Una vez que ya le sumamos la componente Green a cada registro, procedemos a realizar una suma horizontal dos veces.Con esto vamos a tener 8 words en el registro en donde tenemos blue+red+2*green por cada pixel y la parte alta del registro coincide con la parte baja.

Dividimos este registro por 4 haciendo un psrlw 2 y luego hacemos un pshufb para reorganizar los datos.*5*



Ahora leemos los píxeles de src1 y realizamos un PAND con una mascara para limpiar los 2 bits menos significativos de cada componente RGB.Después realizamos la cuenta k= TAMAÑO TOTAL-16-ÍNDICE con índice igual al numero de iteración que estamos en el ciclo, necesitamos restar 16 para que coincida, y leemos en la posición [src+k] para traernos los píxeles inversos a los que habíamos leído. Luego aplicamos un PSHUFB con otra mascara que invierte el orden de cada píxel, ya que queremos obtener el mirror de la imagen.

Ahora nos resta realizar un xor entre los bits del píxel gris y el mirror.Esto lo realizamos haciendo shifts de a bits del registro donde teníamos los píxeles gris acomodados como queríamos hasta que lo hacemos coincidir con unas mascaras que lo que hacen es "$captar$" el bit que nos interesa, cada vez que captamos un bit este lo guardamos en otro registro que tendrá nuestro resultado con un or.



Por ejemplo en el gráfico mostramos como seria para quedarnos con el bit 7 de cada píxel que se encuentra en el registro xmm. Este procedimiento lo realizaremos 6 veces hasta que consigamos todos los bits que necesitamos.

Una vez finalizado este proceso, procedemos a realizar un xor con los píxeles mirror y luego un and con los píxeles que habíamos levantado y luego limpiado los 2 bits menos significativos.

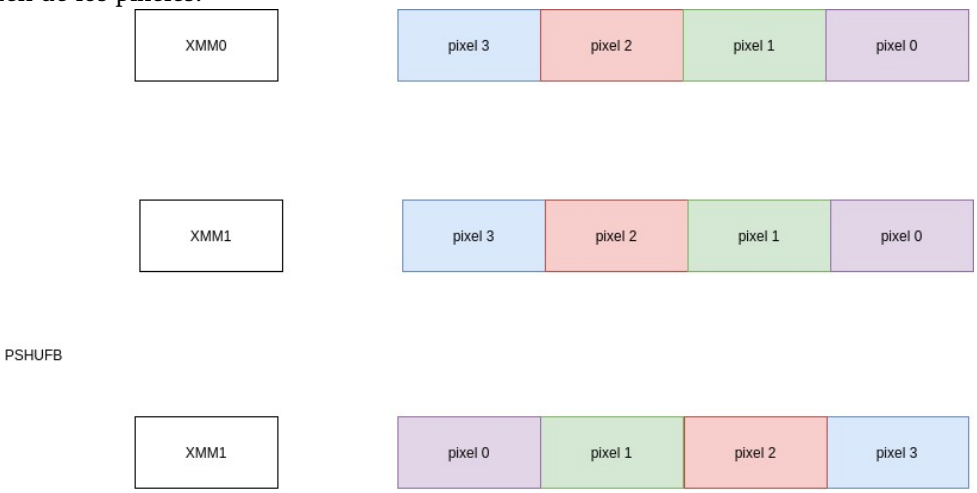
Luego solo nos resta escribir este registro y realizar otra iteración del ciclo.

Aclaración: Todas las mascaras fueron cargadas en memoria alineada y luego las cargamos en registros XMM al inicio de nuestra función para no tener que realizar reiteradas lecturas a memoria.

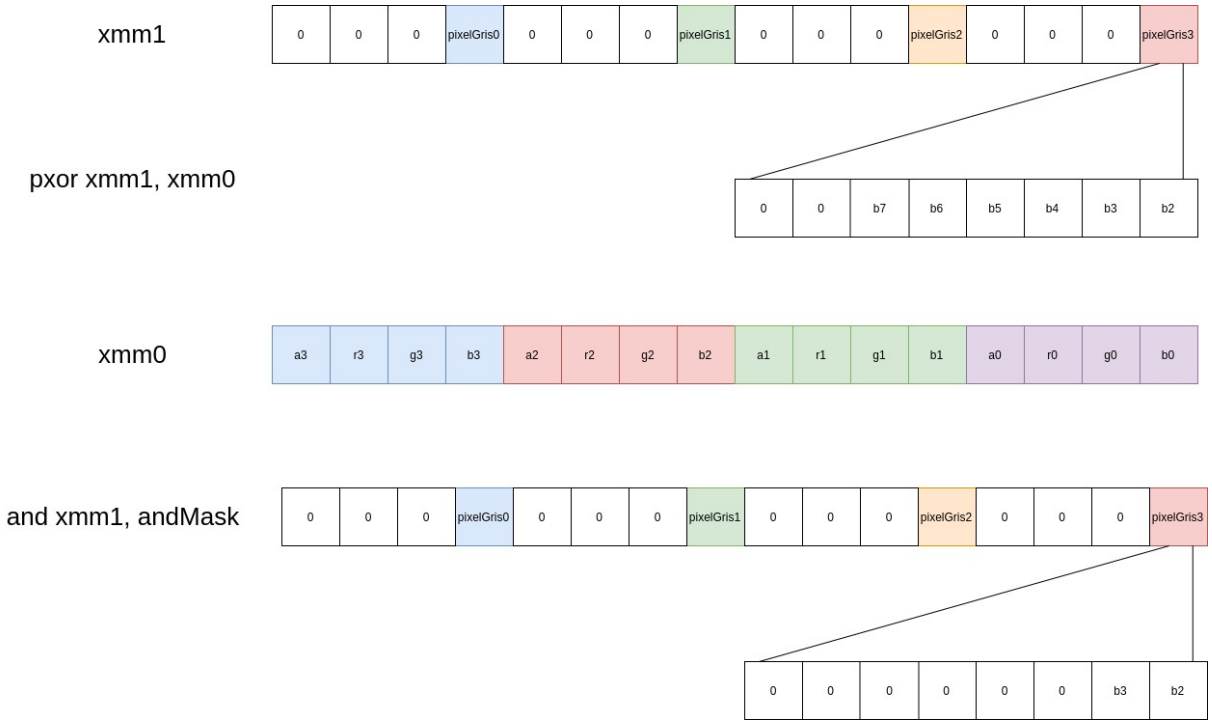
2.2. Filtro Descubrir ASM

Implementación del filtro Descubrir en Assembler Procederemos a explicar nuestra implementación del filtro descubrir.

En la implementación del filtro Descubrir en Assembler, lo que hacemos es leer 4 píxeles de src a xmm0. Luego leemos en xmm1 en la posición [src+k] con k= TAMAÑO TOTAL-16-ÍNDICE con índice igual al numero de iteración en el que estamos, esto nos permite levantar los píxeles correspondientes al mirror. En xmm1 aplicamos un pshufb con la misma mascara de Ocultar que nos permite invertir el orden de los píxeles.



Shifteamos a derecha cada dword 2 bits en xmm1 para luego poder hacer un pxor entre xmm1 y xmm0 para luego realizar un and y quedarnos con los 2 bits menos significativos de cada byte.Que estos representan determinados bits del píxel en gris.



Luego por cada bit que queremos extraer lo que hacemos es hacer una copia de xmm1 en xmm4 y shiftear xmm4 dependiendo que bit queremos conseguir para poder hacerlo coincidir con cada una de nuestras mascaras para cada bit. Luego cuando tenemos el bit que queremos alineado con nuestra mascara realizamos un and, y guardamos los bits en xmm2 que va a ser donde iremos guardando todos nuestros bits. Finalmente cuando ya realizamos los 6 shifts para extraer cada bit de cada píxel hacemos un pshufb para repetir cada píxel gris 3 veces y luego hacemos un un or con una mascara para setear el byte de transparencia en 255.Por ultimo escribimos los datos en memoria.

2.3. Filtro Zigzag ASM

Implementación del filtro Zigzag en Assembler Procedemos a explicar la idea del funcionamiento de esta función implementada en assembler

Nuestra función Zigzag es básicamente un filtro que genera un efecto de zigzag en la imagen, para lograr esto operamos diferentes cuentas en cada fila de píxeles de la imagen original, dependiendo en que fila nos encontremos utilizaremos una operación u otra:

Si el numero de la fila contando como cero a la fila de pixeles superior de la imagen es :
si fila = 0 (mod4) hacer: pixel(n) = (pixel(n-1) + pixel(n-2) +pixel(n+1) + pixel (n-2)) /5
si fila = 1 (mod4) hacer: pixel(n) = pixel(n-2)
si fila = 2 (mod4) hacer: pixel(n) = (pixel(n-1) + pixel(n-2) +pixel(n+1) + pixel (n-2)) /5
si fila = 3 (mod4) hacer: pixel(n) = pixel(n+2)
todo esto considerando que se deja un marco blanco de 2 pixeles

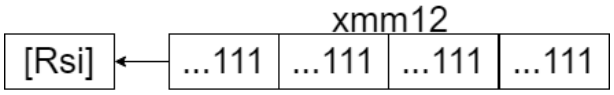
```
PSEUDOCÓDIGO:

Zigzag(ImagenINPUT,N filas,M columnas){

agregarFilaBlanca(ImagenINPUT,fila(0))
agregarFilaBlanca(ImagenINPUT,fila(1))
FOR EACH fila(I) IN ImagenINPUT  DO
    agregar2PixelesBlancosPrincipio(ImagenINPUT,fila(I))
    if( fila(I)==0 (mod4 ) do: Zigzager(ImagenINPUT,fila(I))
    if( fila(I)==1 (mod4)) do: Shift2PixelesDer(ImagenINPUTT,fila(I))
    if( fila(I)==2 (mod4 ) do: Zigzager(ImagenOINPUT,fila(I))
    if( fila(I)==3 (mod4 ) do: Shift2PixelesIzq(ImagenINPUT,fila(I))
    agregar2PixelesBlancosFin(ImagenINPUT,fila(I))
END FOR
agregarFilaBlanca(ImagenINPUT,fila(N-1))
agregarFilaBlanca(ImagenINPUT,fila(N))
return ImagenOUTPUT
}
```

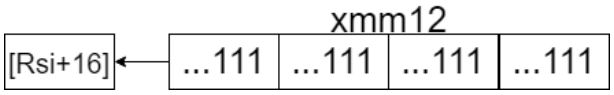
Explicación de cada función en assembler Para cada función en rsi se encuentra un puntero a la imagenOUTPUT y en rdi una a la imagenINPUT, antes de llamar a cada función de los IF, se tiene en cuenta que rsi apunta al 3er píxel de la imagen a escribir y rdi apunta al primer píxel de la imagen de lectura.

agregarFilaBlanca: Esta función es bastante simple, lo único que hace es agregar a cada componente de un píxel el numero 255,

y hace esto para todos los píxeles de cada fila, lo hacemos usando el registro xmm7 que se le cargo una imagen con 1 en todos sus bits

while(no recorri toda la fila) do
movdqu [rsi], xmm7
movdqu [rsi+16], xmm7
rsi = rsi+32
rdi = rdi+32

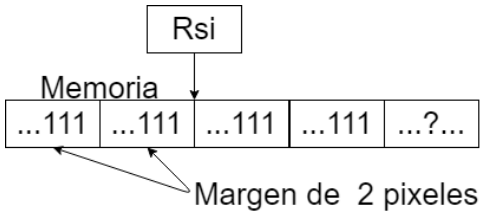




Esto agrega 8 píxeles blancos por ciclo, ya que el ancho de imagen es mayor a 16 y múltiplo de 8 podremos recorrer fácilmente toda la fila sin ningún caso borde, al finalizar la función tanto como rdi rsi apuntara al comienzo de la siguiente fila y otras variables que se utilizan el código para indicar fila y columna.

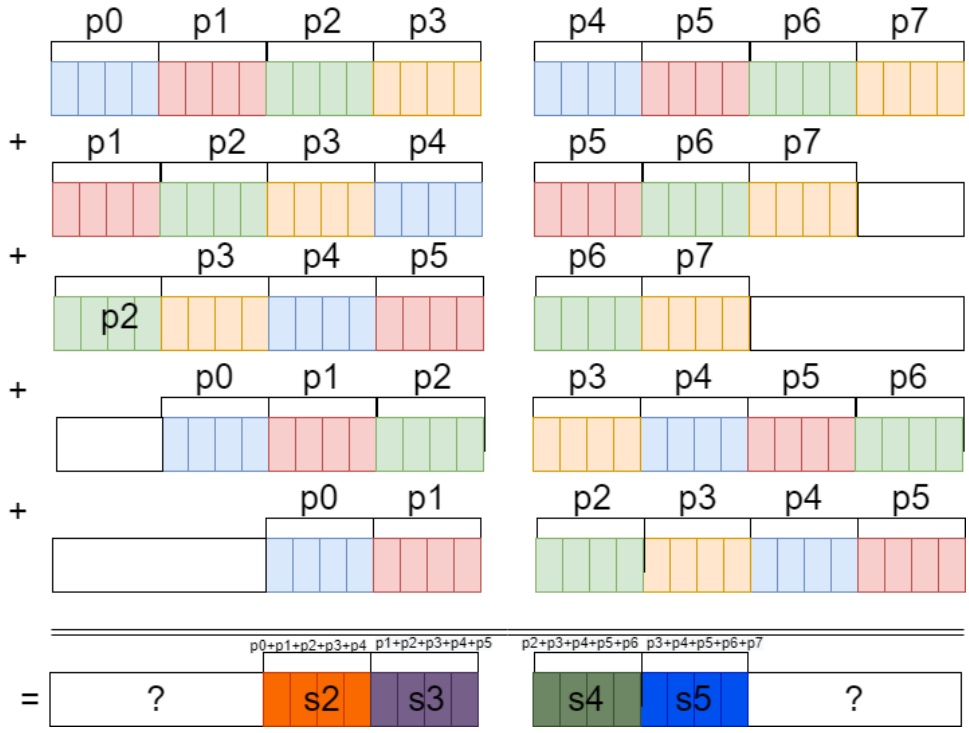
agregar2PixelesBlancosPrincipio: Similar a lo anterior solo que hacemos una sola iteración y con un solo registro

movdqu [rsi], xmm7
rsi+8

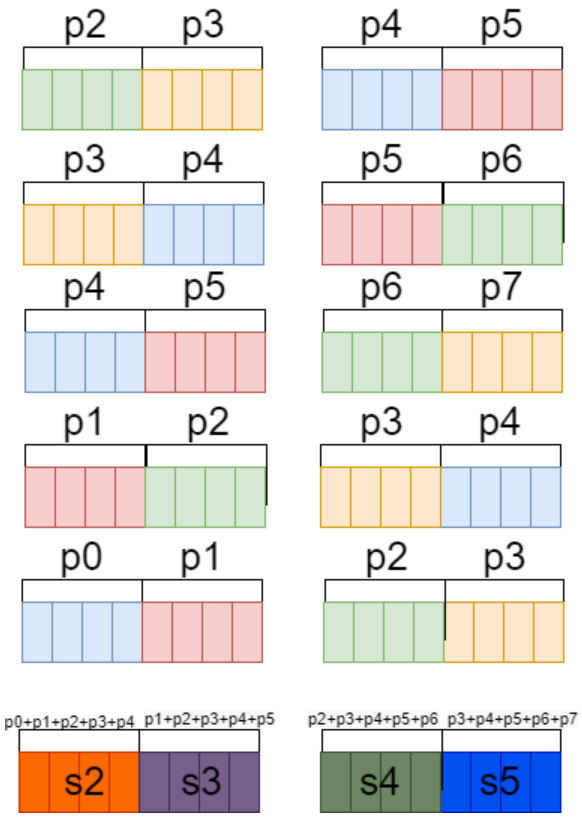


Actualizamos sumando rsi=rsi+8 para que escribamos a partir del tercer píxel, ya que debemos dejar 2 píxeles de margen (se actualizan también variables de columna) para al utilizar las siguientes funciones tener bien alineados los punteros de lectura y escritura.

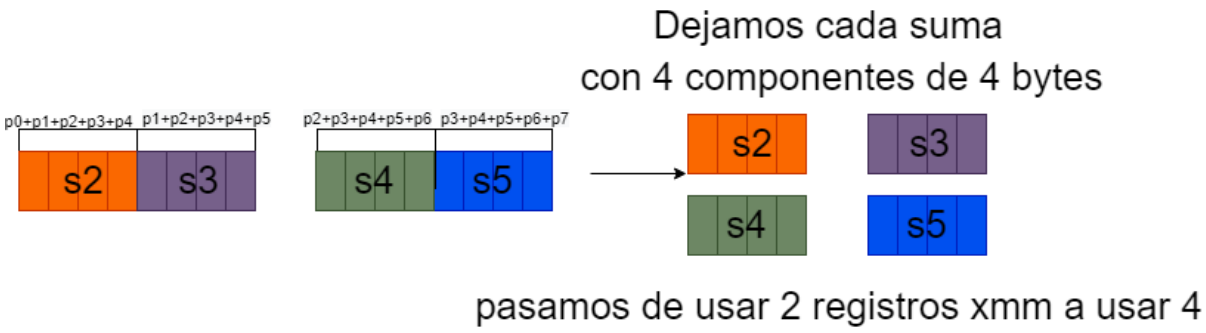
Zigzager: La idea principal de Zigzager es para cada píxel sumar los 2 valores adyacentes a su izq y los 2 valores adyacentes a su derecha, dividir por 5 y así tener un promedio del color de esos 5 píxeles, primero vemos que datos queremos sumar (logramos conseguir cada registro copiando los 2 registros de lectura (xmm0 y xmm1 les cargo sus valores desde rdi y rdi +16) y copiándolos a otros registros y shifteando), notar que leemos desde el píxel 0 hasta el píxel 7 , pero terminaos escribiendo nada mas que entre los píxeles p2 y p5



Para sumar estos valores primero extendemos cada componente del píxel desde Byte a Word, cuando expandamos cada píxel pasara de ocupar 4 bytes a ocupar 4 words y obtenemos estos datos donde cada píxel ocupa 4 words, los obtenemos utilizando los anteriores datos que teníamos expandiendo la parte baja o shifteando para obtener la parte alta de los registros necesarios del anterior gráfico para conseguir el siguiente gráfico.



Luego al resultado necesitamos dividirlo, para esto pasamos cada word a double word, para luego transformarla a punto flotante,



Para esto usamos cvtdq2ps xmm, xmm

Finalmente dividimos , y volvemos a comprimir al resultado a como lo teníamos originalmente, utilizando

divps xmm, xmm y cvtps2dq xmm, xmm

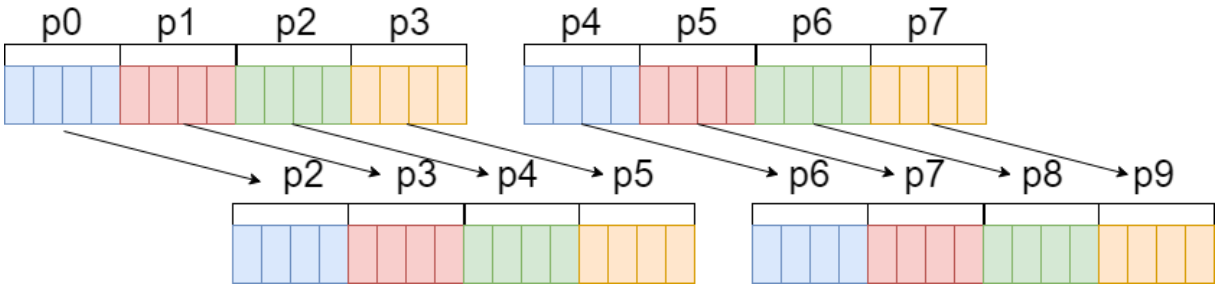
Luego volvemos los datos de double word a byte haciendo varias instrucciones, consiguiendo que cada píxel ocupe 4 bytes(cada componente de píxel igual a un byte) y escribimos el resultado en rsi, notar que si estamos al principio de la fila rsi apunta al píxel 2 ya que dejamos un marco, para conservar esta alineación actualizamos rsi y rdi solamente de a 4 píxeles ya que escribimos solamente 4 píxeles

rsi = rsi+16

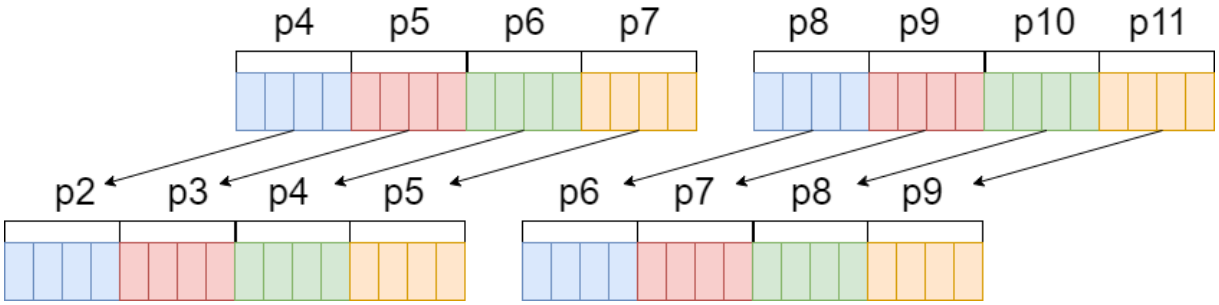
rdi = rdi+16

Cuando lleguemos al final de la fila lo único que hay que hacer es imprimir 2 píxeles blancos y que rsi = rsi+16 para que vuelva a apuntar al principio de la fila (ya que estaba desalineado 2 píxeles a la derecha y venia sumando de a 4 píxeles, en la ultima iteración escribiríamos en los píxeles ”m-5,m-4;m-3,m-2z deberíamos poner en blanco m-1, m.

Shift2pixelesDer y Shift2pixelesizq : Para ambas funciones el comportamiento es muy similar, lo mas relevante en cuanto a complicaciones es ver el alineamiento de la direcciones que leemos,en ambos casos estamos siempre escribiendo 2 píxeles desalineados a la derecha, en shift2pixelesDer leemos 2 píxeles a la izquierda relativos a los que escribimos, por lo tanto estamos leyendo alineados con el inicio de la fila,



Mientras que en Shift2pixelesizq estamos leyendo 2 píxeles relativos a la derecha de donde escribimos, por lo tanto leemos 4 píxeles a la derecha (si estuviésemos al principio de la fila seria leer a partir del 4to píxel, seria análogo a saltear se leer los primeros 4 pixeles con un registro entero)



Por lo tanto el único cuidado que hay que tener es al finalizar la fila recordar en cada caso rsi y rdi para alinearlos al comienzo de la siguiente fila, en ambos casos mientras que iteramos en la fila aumentamos de a 8 píxeles

rsi = rsi +32

rdi = rdi+32

3. Hipótesis y Experimentación del Rendimiento

3.1. Observaciones y detalles de la experimentación

Vamos a experimentar utilizando las distintas implementaciones de cada filtro que mencionamos, vamos a comparar el rendimiento de los filtros de Asembler contra los que tenemos implementados en C, a esto vamos a medir y comparar el tiempo de ejecución para así poder observar cual implementación logra mas performance.

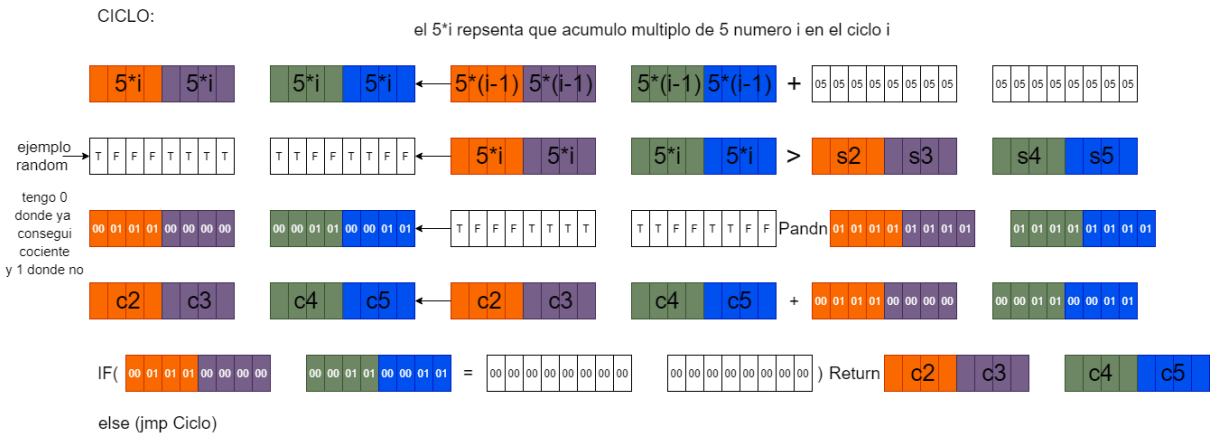
Además vamos a hacer experimentos dentro de las implementaciones de Assembler para comparar distintas variantes, las variantes que vamos a comparar es implementar la división de Zigzag utilizando el algoritmo de la división clásica de los enteros, contra dividir usando las operaciones de punto flotante

por otro lado vamos a comparar en el rendimiento de leer mas acceso a memoria o menos en cada ciclo en nuestras funciones de Ocultar y Descubrir, para esto vamos a probar el rendimiento de estas implementaciones leyendo las mascaras en cada ciclo, contra tener cargadas las mascaras en registros antes de comenzar el ciclo.

3.2. Filtro Zigzag ASM con división alternativa con enteros

Todo el algoritmo es similar tanto antes de dividir y después de dividir, la modificación mas interesante es como tratamos los datos que habíamos conseguido sumar, estos datos una ves que los tenemos sumados en word, no los pasaremos a dword sino los utilizamos en word para conseguir su divisor, para esto utilizamos el algoritmo de la división clásica de los enteros, tenemos que contar cuantas veces entra el divisor en el dividendo y luego esta cantidad contada sera el cociente de la división.

Para lograr esto cargamos en mascaras el numero 5 en cada word y el numero 1 en cada word, además vamos a necesitar un registro que sume 1 en cada word que aun no tengo el cociente y sume cero en cada word que ya conseguí cociente, cuando consigo todos los cocientes ya podre salir del ciclo, una forma de saber si puedo salir es ver si mi registro suma1 tiene cero en todas sus posiciones.



Las sumas y comparaciones son de a word, los colores indican a que píxel corresponde cada sección de el registro, s2,s3,s4,s5 son los obtenidos de la suma anterior a dividir, el registro con los números (00,01,01,01,00,00,00,00) y (00,00,01,01,00,00,01,01) son los registros que se utilizan para modificar el cociente y al mismo tiempo para saber cuando poder salir del ciclo (si todos los números marcan cero se debe a que ya conseguimos cociente), este cociente esta en los 2 registros con (c2,c3) y (c4,c5) estos 2 registros guardan los cocientes de dividir los 4 píxel (cada píxel tiene 4 componentes por lo tanto se almacenan 16 cocientes distintos).

Al finalizar esto el resultado se pasa de word a byte y se almacena en memoria de la misma manera que en la anterior implementación de Zigzag.

3.3. Hipótesis

Hipótesis Experimento ZIGZAG implementado con enteros

Dividir con enteros debería dar mejores resultados que dividir con punto flotante ya que las operaciones de punto flotante son costosas y además de esto tenemos que transformar un dato que teníamos en word a dword y luego a flotante de precisión simple, esta conversión tampoco es muy económica.

Hipótesis Experimento Ocultar y Descubrir con mascaras de memoria vs registros

Para las implementaciones de Ocultar y Descubrir se debería notar una mejora de rendimiento leyendo las mascaras una sola vez en memoria cargándolas en registro antes del ciclo que cargándolas de memoria en todos los ciclos.

Hipótesis global

A Las implementaciones de Assembler deberían tener un mejor desempeño que las implementaciones de c sin optimizar.

B Las implementaciones de Assembler deberían tener un mejor desempeño o similar que las implementaciones de c con optimizaciones.

C Los tiempos de ejecución de Assembler podrían ser lo suficientemente bajos como para implementarse en tiempo real.

3.4. Experimentación y Comparación

Para realizar las experimentaciones usamos como input todas la imágenes que nos otorga la cátedra, estas las utilizamos modificando el tamaño como en los test dados por la cátedra, en los siguientes tamaños:

32x16 64x32 128x64 200x100 256x128 400x200 512x256 800x400 1600x800

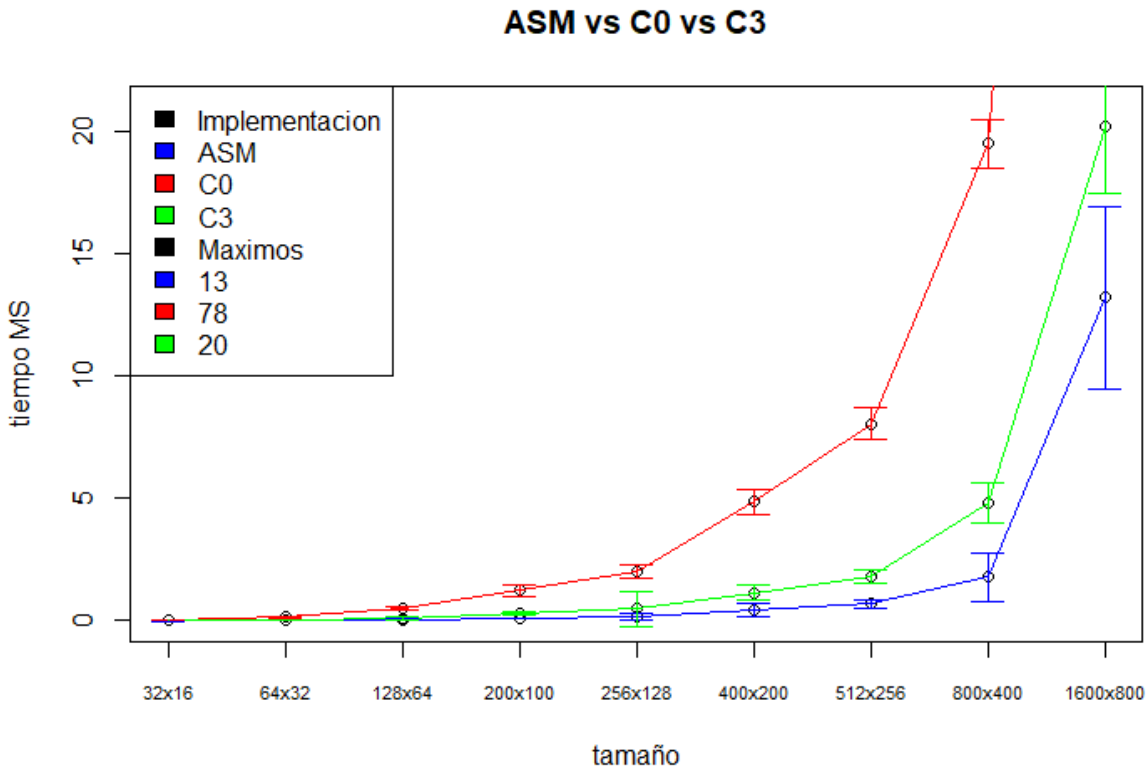
Estas imágenes les aplicamos cada filtro utilizando el test de la cátedra para medir tiempo, este test modificamos el texto de output para poder almacenar los datos en formato csv, corrimos cada filtro 30 veces en cada imagen y por cada filtro distinto armamos un archivo csv que almacena el tiempo de ejecución de cada foto y el tamaño de cada foto.

Con esto obtenemos un archivo.csv para cada implementación distinta de cada filtro, por ejemplo: para el filtro Ocultar tenemos 3 implementaciones, asm, c0 (implementación en código c sin optimizar) y c3 (implementación en código c con optimizaciones), en OcultarData/dataAsm.csv cargamos esto en R, para cada tamaño distinto de imagen calculamos el promedio de tiempo de ejecución, la desviación estándar del tiempo de ejecución y esto da como resultado los gráficos que veremos a continuación, hacemos lo mismo para cada implementación de cada filtro y comparamos los tiempos promediados y ordenados por tamaño de distintas implementaciones de mismos filtros. En los gráficos el eje X es el índice de tamaño, 1 corresponde a 32x16, 2 a 64x32 ... 9 a 1600x800.

4. Comparación Assembler vs C

4.1. Filtro Ocultar ASM vs Filtro Ocultar en C

En Nuestra comparación entre implementación de Assembler contra la implementación en C con el nivel de optimización por default podemos notar que nuestra función de Assembler tiene un rendimiento notablemente superior (hablamos de 13ms contra 78ms) .En la comparación entre la implementación de C con máxima optimización contra nuestra implementación en Assembler, ambas tienen un rendimiento similar, pero la de assembler le saca ventaja a la de C. Esto se puede observar mejor en la siguiente imagen donde graficamos el tiempo que tarda cada implementación en función del tamaño de la imagen.



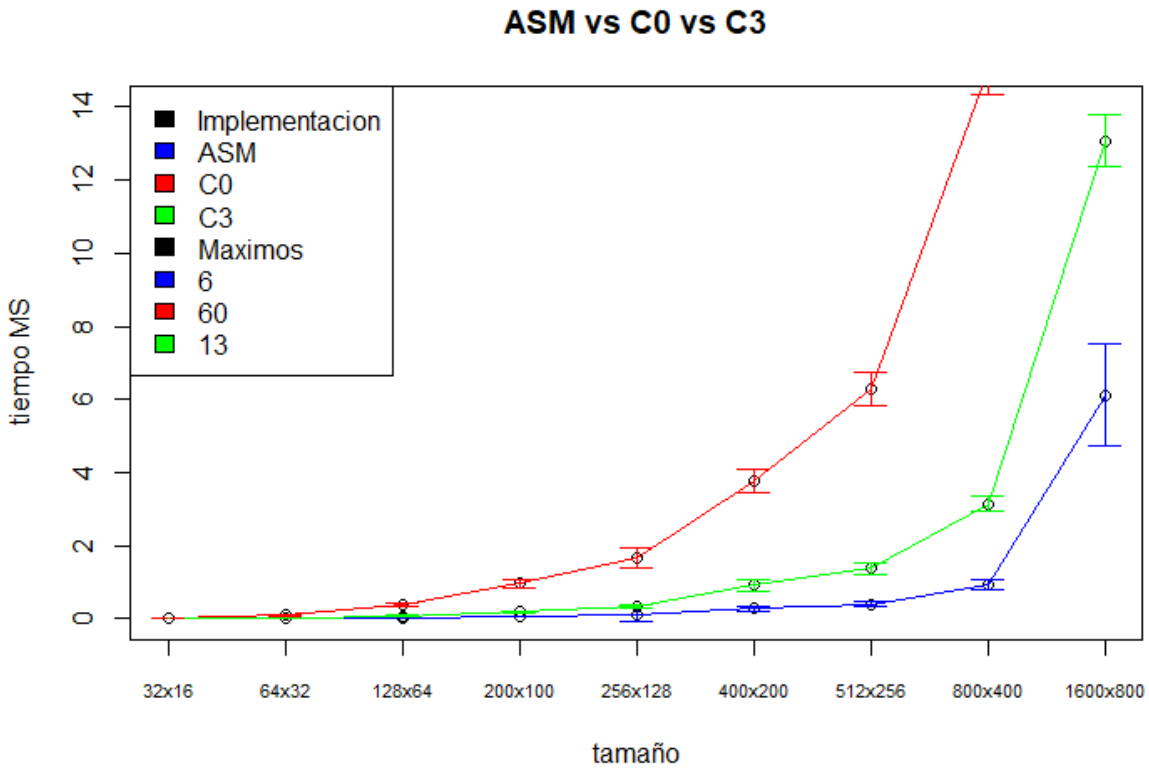
Podríamos decir que la implementación de C es mas lenta porque esta trabaja de a un píxel, y la de Assembler trabaja de a 4 píxeles simultáneamente, la implementación de C0 tardo 6 veces mas que la de Assembler y la implementación de c3 tardo 1.5 veces mas que la de Assembler en el tamaño máximo de 1600x800.

4.2. Conclusión comparación ASM V C en Ocultar

En rendimiento en Assembler en SIMD de ocultar es notablemente mejor al de C sin optimizaciones y es ligeramente mejor (1.5 aproximadamente) que c3, (c con optimizaciones

4.3. Filtro Descubrir ASM vs Descubrir Ocultar en C

De igual manera que en Ocultar, en el análisis de rendimiento de Descubrir notamos que las comparaciones son las mismas, Assembler le gano a ambas, un detalle a notar el tiempo de ejecución de Descubrir de Assembler fue la mitad que el tiempo de ocultar y paso algo similar en la implementación C3 (C con optimizaciones), mientras que la implementación de C sin optimizar no hubo tanta diferencia entre descubrir y Ocultar.



Mirando ambos gráficos podemos notar que las implementaciones en C sin optimizar son muy lentas en comparación a con optimización y Assembler, en el tamaño mas grande podemos notar como,por ejemplo, En ocultar la implementación de asm tardo 10 veces menos de tiempo que la implementación de C0 y tardo la mitad que C3, usando el tamaño 1600x800.

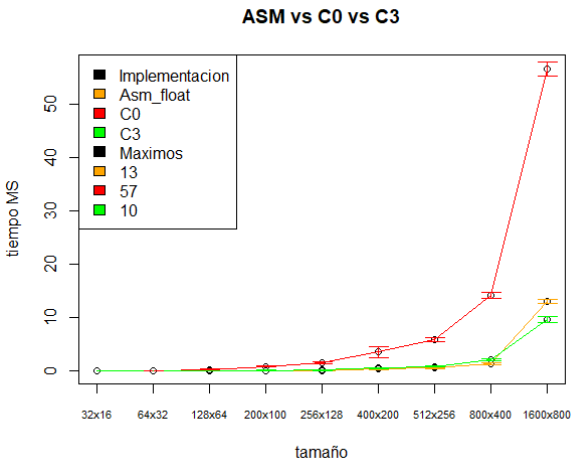
4.4. Conclusión comparación ASM V C en Descubrir

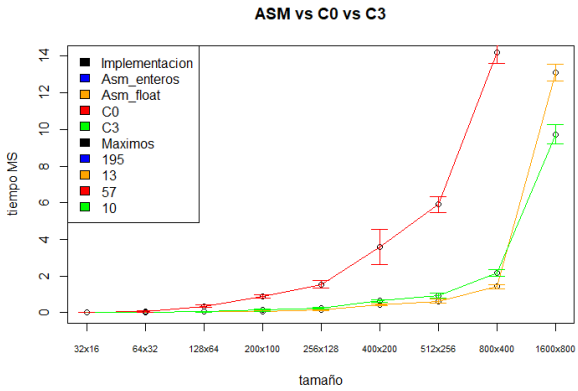
En rendimiento en Assembler en SIMD Descubrir tiene resultados similares a los de ocultar en comparaciones, fue considerablemente mejor que c sin optimizar y tardo aproximadamente la mitad de tiempo que c3.

4.5. Filtro Zigzag ASM vs Filtro Zigzag en C

Notamos que la implementación de Assembler con punto flotante fue mejor que la de C sin optimizar, tardo aproximadamente 4 veces menos que la de C, pero en la implementación de C3 y Assembler se nota que están muy similares,

Para distinguir mejor agregamos otro gráfico con mas aumento, y notamos que en los primeros tamaños van muy similares, en el tamaño 800x400 Assembler lograba mejor performance, pero a partir del tamaño 1600x800 notamos que Assembler con punto flotante logra un tiempo 1.4 mayor que el tiempo de C con implementaciones



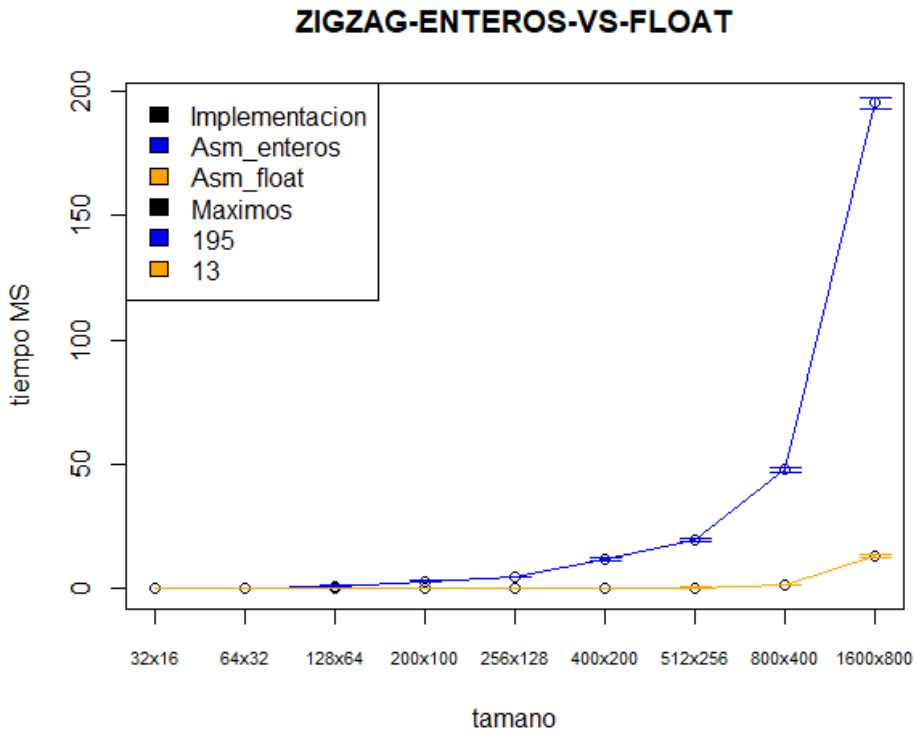


4.6. Conclusión comparación ASM V C en Zigzag

Se consiguió una considerable ventaja en assembler contra C sin optimizar pero contra C3 no esta del todo claro si se consiguió una mejora, en el tamaño 800x400 es donde mayor ventaja notamos pero a partir de 1600x800 logro un desempeño peor, assembler tardo 1.3 veces la implementación de C3, en principio parecerían tardar tiempos similares.

5. Experimentos

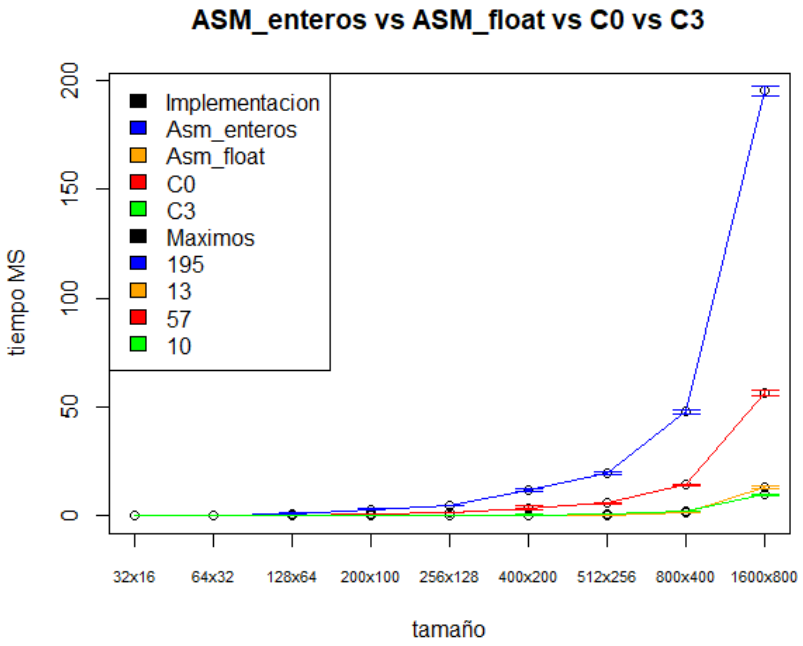
5.1. ZIGZAG implementado con enteros vs Zigzag implementado con float



Comparamos las distintas implementaciones de Assembler, podemos notar que la implementación con punto flotante logra un rendimiento completamente superior, la implementación de la división con

enteros tarda 15 veces mas en computar las imágenes de 1800x800.

Si observamos también la implementación de división con enteros contra las implementaciones de C



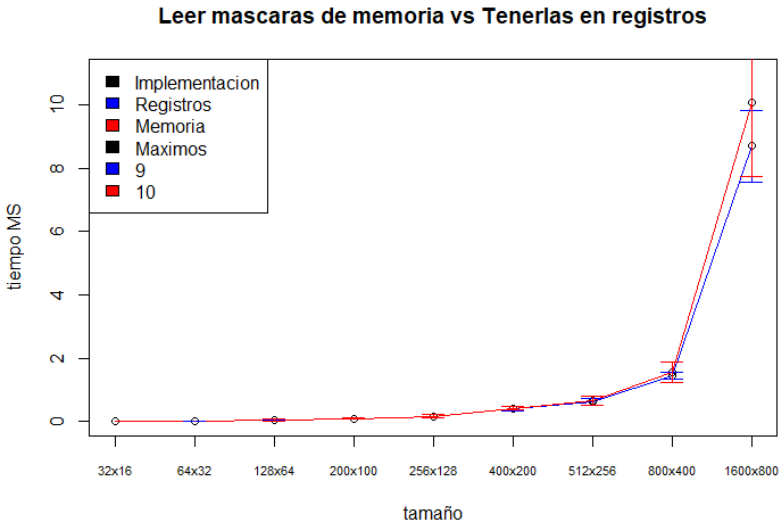
notamos que tiene un rendimiento incluso peor que el de las implementaciones de C.

5.2. Conclusión experimento ZIGZAG implementado con enteros

Al comparar las implementaciones de Zigzag dividiendo con punto flotante o con enteros, notamos que paso exactamente lo contrario a lo que estábamos esperando, el tiempo de dividir usando el algoritmo de división de los enteros resulto ser peor que la implementación de C, sospechamos que esto se debe a la cantidad de instrucciones agregadas, ya que por cada grupo de píxeles que analizamos tenemos que iterar a lo sumo $255/5 = 51$ iteraciones que no teníamos comparado con la versión de dividir con float, a esto le agregamos que tenemos 29 instrucciones que se ejecutan en el peor caso 51 veces mas que las 24 instrucciones que teníamos al dividir con punto flotante, tal vez tratando de lograr una mejor performance evitando algunas instrucciones especificas terminamos empeorando el tiempo de ejecución por agregar un ciclo con saltos condicionales, notar que el tiempo de ejecución de dividir con enteros fue 3.4 veces peor que la implementación de C sin ninguna optimización, por lo tanto en este caso no solo no ganamos nada sino que empeoramos el tiempo de ejecución.

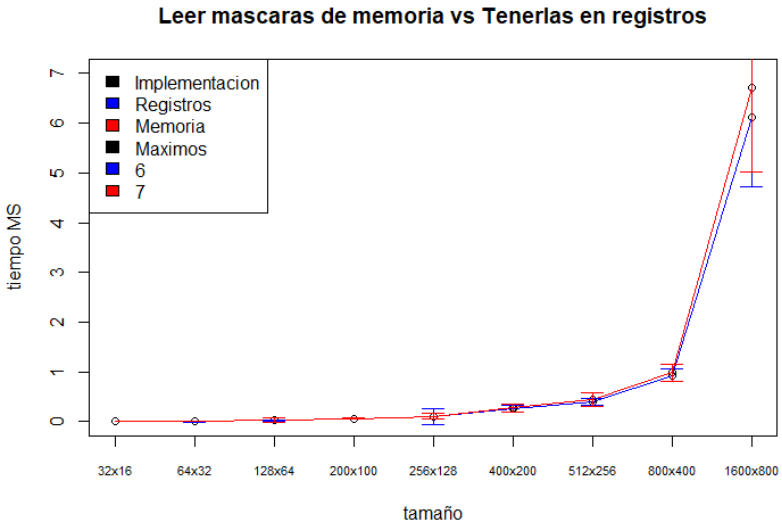
5.3. Leer mascaras de memoria vs registros

Ocultar



En el caso de la función Ocultar podemos notar que para imágenes chicas prácticamente no hay diferencia en el tiempo que tarda en ejecutarse la función, ya que el máximo que tardo la función teniendo que leer de memoria fueron 10ms y leyendo de registros fueron 9ms.

Descubrir



Podemos observar que al igual que la función Ocultar, en Descubrir obtuvimos resultados muy similares, con diferencia de 1ms en las imágenes de mayor tamaño, observando la desviación vemos que no hay diferencia notoria entre las implementaciones.

5.4. Conclusión experimento Ocultar y Descubrir con mascaras de memoria vs registros

Con los gráficos observados de las implementaciones de Ocultar y Descubrir leyendo mascaras de memoria vs guardándolas en registros previamente podemos notar que la diferencia de rendimiento es mínima.Es tan poco el incremento de rendimiento que obtenemos que no es suficiente para que podamos dar como valida nuestra hipótesis inicial. Podríamos pensar que no obtenemos el cambio de rendimiento que esperábamos debido a que las mascaras una vez leídas se quedan en la memoria cache del procesador y tiene un relativo fácil acceso a ellas.

6. Conclusión Global

Una vez finalizado los experimentos y analizados los datos podemos ver que algunas de nuestras conclusiones tenían sentido pero otras en otras no tanto:

A Observando los gráficos podemos notar que se cumple que las implementaciones de assembler lograron mejor performance que las implementaciones de C0, nuestras implementaciones aprovechan las instrucciones de SIMD y nos permiten computar en un mismo tiempo una mayor cantidad de datos, ya que nuestras implementaciones en Assembler son mucho mas rápidas que sus versiones en C sin optimización.Esto se debe a que cuando le indicamos -o0 a gcc este prácticamente no realiza ningún tipo de optimización y solo se ocupa de que el tiempo de compilación sea rápido.

B En los filtros de Ocultar y Descubrir la implementación de Assembler fue mejor que la implementación de C3, Creemos que esto se debe al aprovechar las instrucciones de SIMD y lograr computar mas datos en un mismo tiempo, la implementación de zigzag división con float si bien no fue mejor que la implementación Zigzag C3 consiguió un rendimiento muy parejo, en particular en la imagen mas grande fue un poco peor, creemos que se debe a las optimizaciones que corren en c3, nosotros desconocemos como están implementadas estas, por lo tanto no sabemos el motivo por el cual no logramos vencerlo, tal vez existe alguna otra forma de dividir sin utilizar la división de punto flotante y sin perder performance como en el algoritmo de la división que usamos en la otra implementación de assembler para vencer las optimizaciones de C.

C Si quisiéramos usar estos filtros en tiempo real, uno tendría que considerar el tiempo esperado de computo, si quisiéramos por ejemplo tener 60 imágenes por segundo necesitaríamos tardar menos que 1/60 *1000=16.7 Milisegundos en calcular el filtro, tanto para ocultar y Descubrir y Zigzag logramos estar por debajo de este tiempo, en el caso de Zigzag división con enteros no nos serviría ya que tardo aproximadamente 200ms en la resolución de 1600x800, algo que no tenemos en cuenta es en que contexto se correrían en tiempo real y no sabemos realmente si tendrían que ejecutarse en incluso menos tiempo si es que a la par de estos filtros se hacen otros cálculos que terminen alentando demasiado la velocidad de calculo de la imagen como podría pasar, por ejemplo, en los videojuegos que se calcula a tiempo real muchos elementos de una imagen, pero haciendo un análisis simple parecerían poder ser útiles para ejecutarse en tiempo real.

Datos de la experimentación: Los datos Utilizados en el experimento se encuentran en la carpeta tp-2-orga-2/src/DataExperimentos allí se encuentran los datos recolectados de los experimentos en formato csv y el código utilizado para generar las imágenes y los scripts en tp-2-orga-2/src utilizados para correr varias veces los test de la cátedra las imagenes utilizadas son las de la cátedra en el google drive:

TP2/imgExamples/800x600

TP2/imgExamples/1280x720