DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

9 de mayo de 2017

Organización del Computador II
Primer Cuatrimestre de 2017

InsiSto gEnio zen acaba

Integrante	LU	Correo electrónico
Bonggio, Enzo	074/15	ebonggio@dc.uba.ar
Szperling, Sebastián Ariel	763/15	sszperling@dc.uba.ar
Tarrío, Ignacio	363/15	itarrio@dc.uba.ar

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Convertir entre RGB e YUV

Los primeros filtros se tratan de conversores entre los espacios de colores RGB e YUV. Los mismos toman los valores de cada pixel y aplican transformaciones matriciales, tomando cada pixel individual como una matriz de sus componentes.

Esta transformación normalmente implica decimales, pero la misma está simplificada para funcionar con enteros entre 0 y 255 (que son los valores posibles de cada componente de un pixel).

Para la conversión RGB a YUV se aplican la siguiente transformación:

$$\begin{bmatrix} Y = sature(((66 * R + 129 * G + 25 * B + 128) >> 8) + 16) \\ U = sature(((-38 * R - 74 * G + 112 * B + 128) >> 8) + 128) \\ V = sature(((112 * R - 94 * G - 18 * B + 128) >> 8) + 128) \end{bmatrix}$$

donde *sature* representa una saturación sin signo, es decir, los valores fuera del rango [0,255] son acotados a los extremos del mismo.

Para la transformación inversa se utiliza la siguiente transformación:

$$\begin{bmatrix} R = sature((298 * (Y - 16) + 409 * (V - 128) + 128) >> 8) \\ G = sature((298 * (Y - 16) - 100 * (U - 128) - 208 * (V - 128) + 128) >> 8) \\ B = sature((298 * (Y - 16) + 516 * (U - 128) + 128) >> 8) \end{bmatrix}$$

Cabe destacar que si bien los valores son correctos, los visores de imágenes siguen renderizando los colores como si correspondiesen a RGB, por lo que se genera el efecto visual que se muestra a continuación:

1.1. Implementación

La solución implica recorrer la imagen completa aplicando la transformación a cada pixel de manera individual.

La implementación en C es relativamente trivial: se aplica la transformación correspondiente a cada componente de manera individual. En el caso de la transformación YUV a RGB existen algunos valores que podemos reutilizar, pero los demás cálculos se realizan como se indica en el problema.

Para la implementación en ASM definimos un par de constantes. La idea es que cada una represente una fila de la matriz de transformación. De esta manera, podemos calcular los 3 componentes fuente de cada componente destino en simultaneo:

$\texttt{XMM}_1$	R	G	B	0
	$\texttt{XMM}_{14} \leftarrow \texttt{XMM}_1$			
	$\texttt{XMM}_{15} \leftarrow \texttt{XMM}_1$			
$\texttt{XMM}_9$	66	129	25	0
$\texttt{XMM}_{10}$	-38	-74	112	0
$\texttt{XMM}_{11}$	112	-94	-18	0
	$\texttt{PMULLD XMM}_{14}$, $\texttt{XMM}_9$			
$\texttt{XMM}_{14}$	Y'_R	Y'_G	Y'_B	0
	$\texttt{PMULLD XMM}_{15}$, $\texttt{XMM}_{10}$			

$\texttt{XMM}_{15}$	U'_R	U'_G	U'_B	0
$\texttt{XMM}_1$	R	G	B	0

$\texttt{PMULLD XMM}_1\ \texttt{, XMM}_{11}$

$\texttt{XMM}_1$	V'_R	V'_G	V'_B	0

Estos valores parciales luego son sumados en paralelo

$\texttt{PHADDD XMM}_{15}\ \texttt{, XMM}_{14}$

$\texttt{XMM}_{15}$	Y'_{R+G}	Y'_B	U'_{R+G}	U'_B

$\texttt{PHADDD XMM}_1\ \texttt{, XMM}_1$

$\texttt{XMM}_1$	V'_{R+G}	V'_B	V'_{R+G}	V'_B

$\texttt{PHADDD XMM}_1\ \texttt{, XMM}_{15}$

$\texttt{XMM}_1$	Y'	U'	V'	V'

Como se puede ver, al sumar se genera un valor duplicado al final. El mismo no la correción del filtro (ya que no estamos trabajando con el componente alfa), y se genera porque las imagenes tienen 3 componentes.

Por último, estos componentes se denotan con $'$ porqe los mismos no son los valores finales: debemos aplicar 2 sumas y un shift a todos los componentes para finalizar la conversión. Para las sumas utilizamos nuevamente constantes precargadas en un registro XMM.

Ya que contamos con registros de 128 bits y cada pixel mide 32 bits de ancho (RGBA), podemos cargar 4 pixeles de la memoria por iteración. Los mismos son empaquetados y desempaquetados, y la lógica se encuentra cuadruplicada para procesar todos. Esto se hace para reducir el impacto de los accesos a memoria y los saltos condicionales.

Un detalle importante que aplica a este filtro es la independencia entre todos los pixeles, y la independencia de los mismos con respecto a su posición. Este detalle nos permite recorrer la imagen no como una matriz de pixeles sino como una lista continua de tamaño $w \times h$.

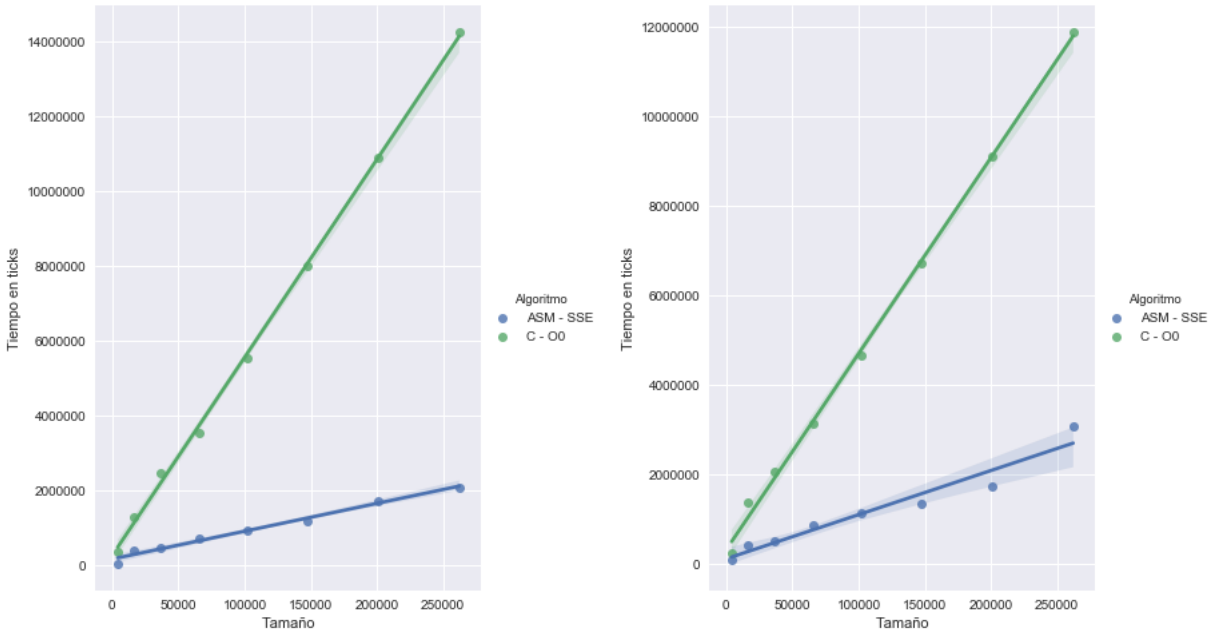
1.2. Análisis preeliminar

Comparación de rendimiento de ASM vs C

Para comparar el rendimiento de los algoritmos implementados, utilizamos los macros provistos por la cátedra para leer los valores del timestamp counter. Los valores de tiempo están expresados en ticks.

A modo de reducir el ruido, repetimos cada medición varias veces y tomamos el menor valor de cada medición al graficar (ya que cualquier valor mayor representa ruido introducido por el scheduler y otros factores).

Como ambos filtros son muy similares, los podemos comparar lado a lado:

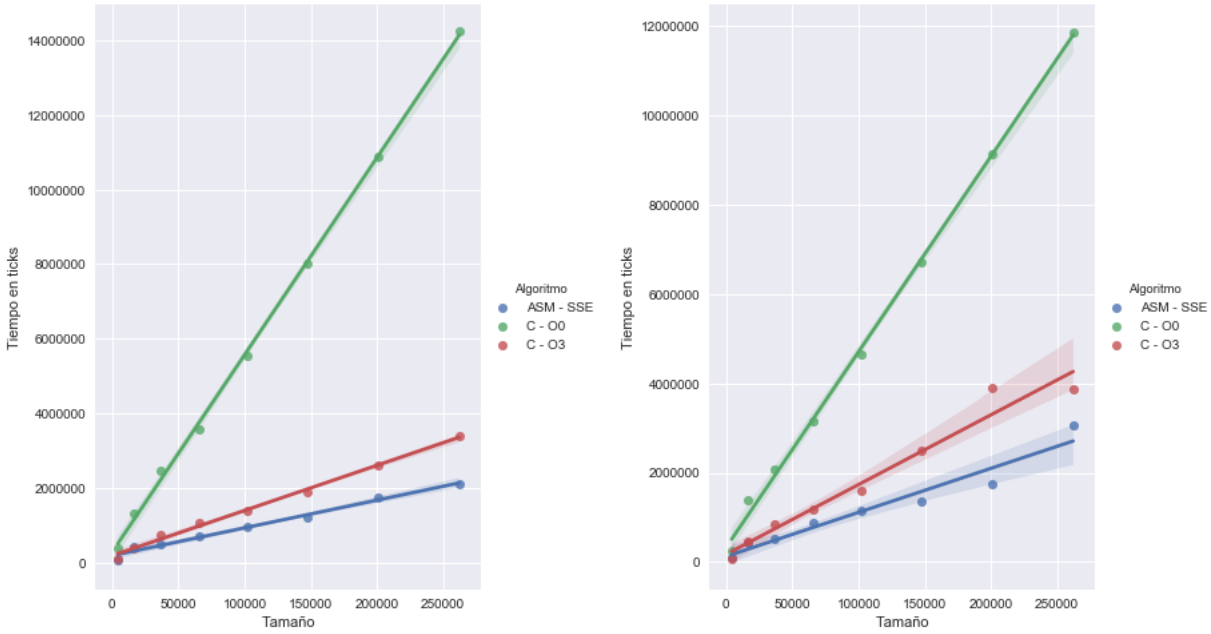


Por un lado, se puede notar una diferencia enorme en la performance del algoritmo escrito en C contra el escrito en ASM.

Por otro lado, se puede ver que la escala del conversor YUV a RGB toma menores valores para el algoritmo en C. Esto de debe a ciertas operaciones que como mencionamos se repiten, y por ende podemos pre-calcular y reutilizar valores intermedios.

1.3. Experimentación

Si bien la primera comparación es interesante, no estabamos seguros si se trataba pobre uso de recusos por parte del compilador. Por esto, decidimos comparar contra el mayor valor de optimización de GCC (`-O3`):



Al compilar con optimizaciones, la brecha de performance baja drásticamente. Sospechamos que esto se debe a un uso más exhaustivo de los registros en lugar de acceder constantemente a memrio. Sin embargo, en ciertos casos se puede ver que el algoritmo escrito con instrucciones SIMD sigue siendo más óptimo (hasta casi 60 % del tiempo, 2090676 en ASM vs 3377772 en C con `-O3`). Si bien la diferencia es visiblemente menor, y nuestras mediciones contaron con un poco de ruido, la diferencia porcentual es visible y relevante.

2. Combinar

Este filtro consiste en cambiar la distribución de píxeles de una imagen tal que queden ordenados en cuatro cuadrantes diferentes. Se obtendrá mediante la aplicación de este filtro cuatro imágenes distintas de menor tamaño a la original, pero en donde los píxeles de la original se encuentran aun presentes en la imagen resultante. Es decir :

$$\forall\ pixel,\ cantidadPixeles(pixel, imagenOriginal) == cantidadPixeles(pixel, imagenResultante) \wedge imagenOriginal.length == imagenResultante.length$$

$$pixel \in imagenOriginal$$

El resultado visual que provoca la aplicación de este filtro es la sensación de que la imagen se dividió en cuatro pequeñas imágenes cuando verdaderamente ninguna de ella es igual a la otra. Se puede ver en el ejemplo de la figura 1 como es la distribución que va teniendo la aplicación de nuestro filtro allí podemos distinguir que los píxeles antes y luego de la aplicación del filtro son los mismos.

Figura 1: Distribución de píxeles tras aplicar el filtro de combinar

También podemos ver en la figura 2 cual es el impacto de nuestro filtro en una imagen real, podremos notar aquí lo parecidas que son imágenes resultantes en cada uno de los cuadrantes, a nuestro ojo es casi imperceptible la diferencia.

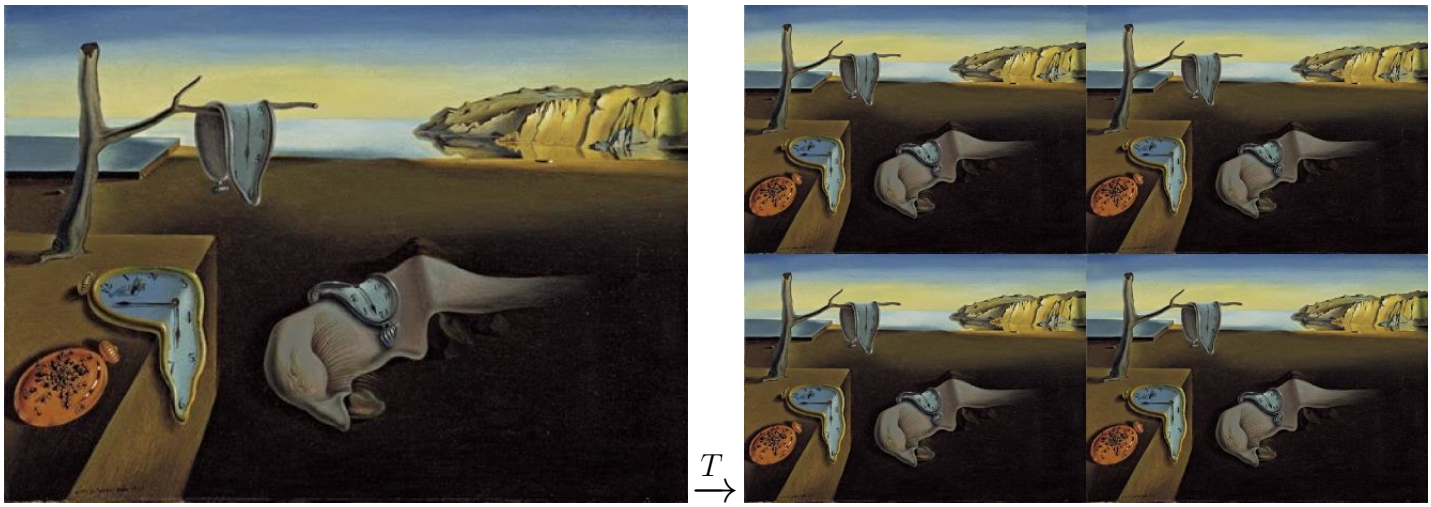


Figura 2: Imagen real antes y luego de la aplicación del filtro

2.1. Implementación

Explicación general de la solución

Solución en código C La solución que fue planteada en c consiste en recorrer la matriz asociada a la imagen de entrada una sola vez píxel por píxel. Cada uno de estos píxeles posee una coordenada y por medio de esta se calcula la coordenada en la matriz asociada a la imagen de salida.

Solución en ASM En cuanto a la implementación en código assembler se tuvo que pensar una solución totalmente diferente ya que al tener que hacerlo con registros de 128 bits nuestra solución de agarrar los píxeles uno por uno y calcular su lugar correspondiente no seria posible. Se tomaron las siguientes decisiones de modelado Decisiones de modelado:

- Decidir con cuanta información a la vez íbamos a trabajar , se llego a la conclusión que lo mejor seria trabajar con 8 píxeles a la vez. Esta decisión se tomo puesto que eligiendo los 8 píxeles del lugar correcto podíamos llegar a armar 8 píxeles que iban a ser puestos en la imagen de salida. Esto nos debería dar un ahorro en la cantidad de veces que vamos a pegarle a memoria.
- Decidir entre la posibilidad de agarrar 16 píxeles que este en la misma fila o que estén la misma columna. Nos pareció lo mas fácil de implementar y lo mas efectivo a la hora de usar la cache era que tomáramos 8 píxeles contiguos. Se vera mas adelante en un experimento la diferencia entre ambos
- Ya que íbamos a tomar de a 8 píxeles contiguos pero el enunciado del trabajo practico solo aseguraba que la imagen a lo ancho iba a ser múltiplo de 4 , teníamos que hacer algo con respecto a los casos donde la imagen no era múltiplo de 8. En este caso tratamos de emular un poco la practica que realiza muchas veces el compilador de intel donde este separa el código en casos especiales para poder ahorrarse de preguntar en el medio del código . Por ende en nuestro código ASM solo preguntamos una vez si el ancho es múltiplo de 4 o de 8 y dependiendo de eso el código se disfurca en dos

Ahora hablemos un poco mas de la iteración dentro del un ciclo de nuestro código ASM , podemos separar lo que hace dentro de una columna de lo que hace al cambiar de fila. Dentro de una columna nuestro objetivo es agarrar 8 píxeles llámense Pi con $1 \leq i \leq 8$ y trabajarlos de forma tal que queden listos para ser pegados en la memoria de la imagen destino. El siguiente gráfico nos mostrara como ocurre la transformación de los 8 píxeles desde que son extraídos desde la imagen fuente hasta que están listos para ser puestos en la imagen destino :

P8	P7	P6	P5	P4	P3	P2	P1

Separo en pares e impares

0	P7	0	P5	0	P3	0	P1

P8	0	P6	0	P4	0	P2	0

Shifteo pre juntar

P7	0	P5	0	0	P4	0	P2

Los uno cruzados con un or

P7	P3	P5	P1	P8	P4	P6	P2

Hago shuffle

P7	P5	P3	P1	P8	P6	P4	P2

Una vez que tenemos los 8 píxeles ordenados según los quiero procedo a guardarlos en la posiciones de memoria a las que apuntan el registro $R8$ y $R9$ sin preocuparme en este caso del cuadrante donde estos dos están apuntando.

En cuando a que pasa cuando la iteración sobre la columna se termina y se pasa a una nueva columna lo vamos a explicar a continuación: Primero movemos $R8$ y $R9$ hacia su próxima fila recordando que la próxima fila sea cual sea el cuadrante donde se encuentro se realiza sumándole una fila a cada uno.

El tema viene ahora en que hay que ir switcheando los cuadrantes donde voy insertando los píxeles resultantes cada uno fila. Siendo esta parte del código la que se encarga de swtichear $R8$ y $R9$ entre los distintos cuadrantes se utiliza para ello una variable puente para no sobrescribir ni pisar ningún valor.

Snippet del código :

```
mov rax, r8
mov r8, r10
mov r10, rax
mov rax, r9
mov r9, r11
mov r11, rax
```

Lo único que cambia de esta explicación con respecto a cuando el ancho no es multiplico de 8 es en este punto donde tenemos que cambiar de fila, ya que nos quedaron 4 píxeles sin procesar y como seria demasiado trabajoso

holdearlos para poder trabajarlos en otra iteración posterior, lo mejor que nos ocurrió fue trabajarlos manualmente e insertarlos en el lugar donde les corresponde. Luego de esto se sigue con el paso recién explicado.

2.2. Análisis preliminar

Comparar para distintos tamaños, relaciones entre implementaciones

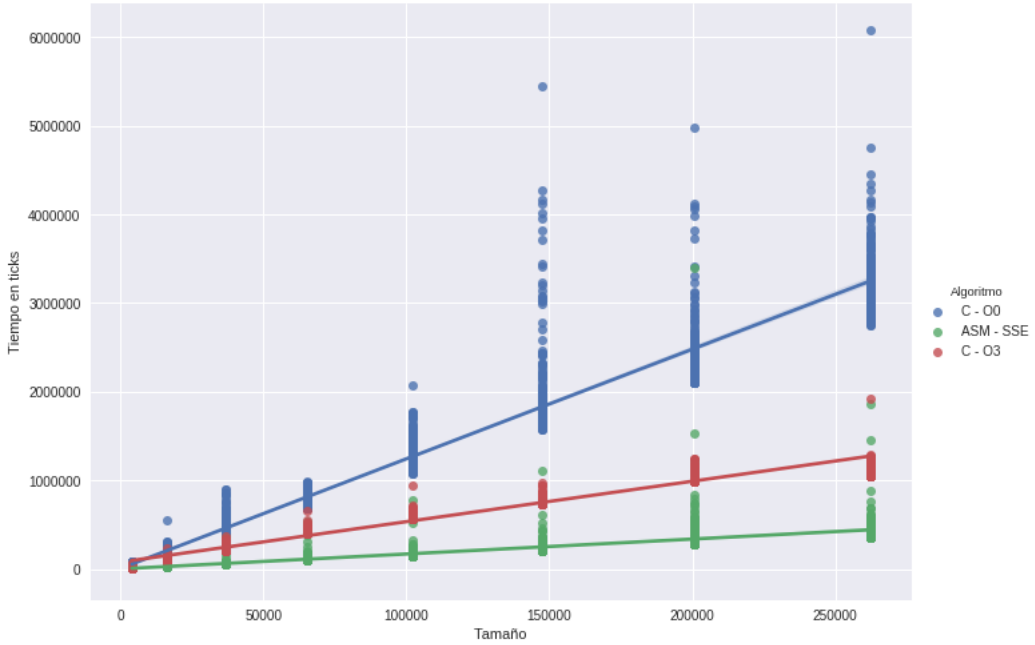


Figura 3: Comparación llana de las implementaciones en C contra la implementación en ASM

Comparación de rendimiento de ASM vs C

Luego de sacar manualmente un poco del ruido podemos ver una clara diferencia entre la implementación de C y la de ASM. Dato curioso que podemos agregar entre la diferencia de O0 y O3 es que al momento de compilar con gcc (gracias a la herramienta https://godbolt.org/) se puede apreciar como una linea de código se vuelve ASM dependiendo el flag.

Después de analizarlo los resultados tanto en O0 como en O3 podemos ver que esta linea de código :

```
mDst[offsetH + (h >> 1)][offsetW + (w >> 1)] = mSrc[h][w];
```

Cuando se encuentra el flag de O0 activado $h >> 1$ se calcula todas las veces, en cambio cuando el flag de O3 esta activo esto no sucede y se calcula antes de ciclar. Existen otras mejoras pero el nivel del código no nos permitió encontrarlas. Nos pareció interesante en este gráfico dejar todos tiempos que fuimos midiendo para ver que tal se comporta el ordenador cuando medimos tiempos aunque hay un poco de outliners la gran mayoría de los tiempos rondan en lo mismo.

2.3. Hipótesis de trabajo

Conjunto de ideas de experimentos

Una de las primeras ideas a la hora de experimentar con nuestra implementación de ASM es poder correrla en contra de la de implementación que se realizo en C , teniendo en cuenta que en el caso de este filtro juega mas el echo de guardar en memoria los datos que de procesarlos.

También tratamos de llevar a cabo en este conjunto de experimentación la idea de Loop unrolling , con alguna pequeña modificación. Si vamos a lo que se entiende como Loop unrolling y lo aplicamos al algoritmo de ASM lo que lograríamos seria leer de a 16 píxeles a lo largo a la vez en de a 8 píxeles logrando así una reducción en los saltos del loop de columna. Pero esta implementación nos llevaría a tener que abrir mas los casos en los cuales anteriormente tratábamos ahora tendríamos imágenes que deberían ser congruentes a 0 modulo 16 para poder ir por el camino del unrolling pero caso contrario es decir ser congruente 4, 8 , 12 deberían tratarse en otra parte del programa y así solo

logrando un incremento en la performance para 0.25 de los casos. Para poder solventar este pequeño numero de casos donde el unrolling podría ser efectivo se decidió procesar 16 píxeles a la vez pero distribuidos en dos filas , de esta manera solo habría que tener en cuenta que las filas sean múltiplos de 2 es decir pares para poder aplicar esta mejora al algoritmo.

Otra idea que había surgido para probar en un experimento fue la de usar registros mas grandes (AVX) para poder lograr así una reducción a la hora de mezclar los elementos, se había pensado que teniendo muchos mas píxeles juntos iba a resultar mas fácil poder ordenarlos para su posterior puesta en en memoria. El problema con este experimento fue que encontramos que las instrucciones AVX (que funcionan en 256 bits) lo único que hacen es replicar comportamientos en la parte baja y la alta del registro es decir los trata como dos registros de 128 bits que solo logran comportamientos de este tipo de registro. Desistimos de hacer este experimento por no encontrar las instrucciones necesarias para poder realizar un código mas performante que el propuesto como solución.

Afirmaciones que buscan probar verdaderas

Buscamos mediante distintos sistemas de medidas mostrar que desenrollar el loop aumenta la performance de nuestro algoritmo ya que el sistema de predicción debería predecir menos saltos.

2.4. Diseño experimental

Explicación de como y que van a medir

Queremos ver bien de cerca que es lo que pasa con nuestras dos implementaciones porque si tomamos las medidas en una escala muy chica vamos a perder información sobre que lo que esta pasando.

También nos gustaría saber cual es el tiempo que se demoran ambos algoritmos en desarrollar un ciclo. Sabemos que la diferencia esta en que ambos resuelven diferente cantidad de píxeles a la vez, por ende el tiempo que demoren en resolver un ciclo va a estar dividido por la cantidad de elementos que trabaja a la vez. Vamos a poner también en la misma linea el tiempo de un ciclo del algoritmo de c que solo procesa un píxel a la vez.

Un experimento interesante que debería estar incluido en el tp sería poder medir que porcentaje de un ciclo del algoritmo presentado como respuesta se utiliza cargando datos, que porcentaje en procesarlos y que porcentaje en cargarlos nuevamente a memoria, tenemos la impresión que la gran mayoría del tiempo se la pasa cargando y poniendo cosas en memoria por ende si quisiéramos mejorar aun mas nuestro algoritmo podríamos solamente trabajar sobre el porcentaje de procesamiento de los datos. Estaríamos en presencia de un cuello de botella representado por la memoria.

El roll del cache en imágenes grandes aquí vamos a probar cual es la diferencia entre la implementación presentada como respuesta al trabajo practico y la implementación del unrolling. Aunque anteriormente vimos que en ejemplos chicos la implementación del unrolling era mejor, en el caso de imágenes obscenamente grandes (mayores a la capacidad del cache) esto no ocurre en imágenes grandes.

Explicación del conjunto de datos de entrada

En el único momento que vamos a utilizar datos por fuera de la cátedra va a ser al momento de querer ver el funcionamiento de la cache en nuestro algoritmo, en este caso utilizamos una imagen que excede la cache de nuestra CPU , la imagen pesa 200mb y por obvios motivos no va a ser adjuntada junto al tp.

2.5. Resultados y Análisis

Resultados obtenidos, gráficos y tablas

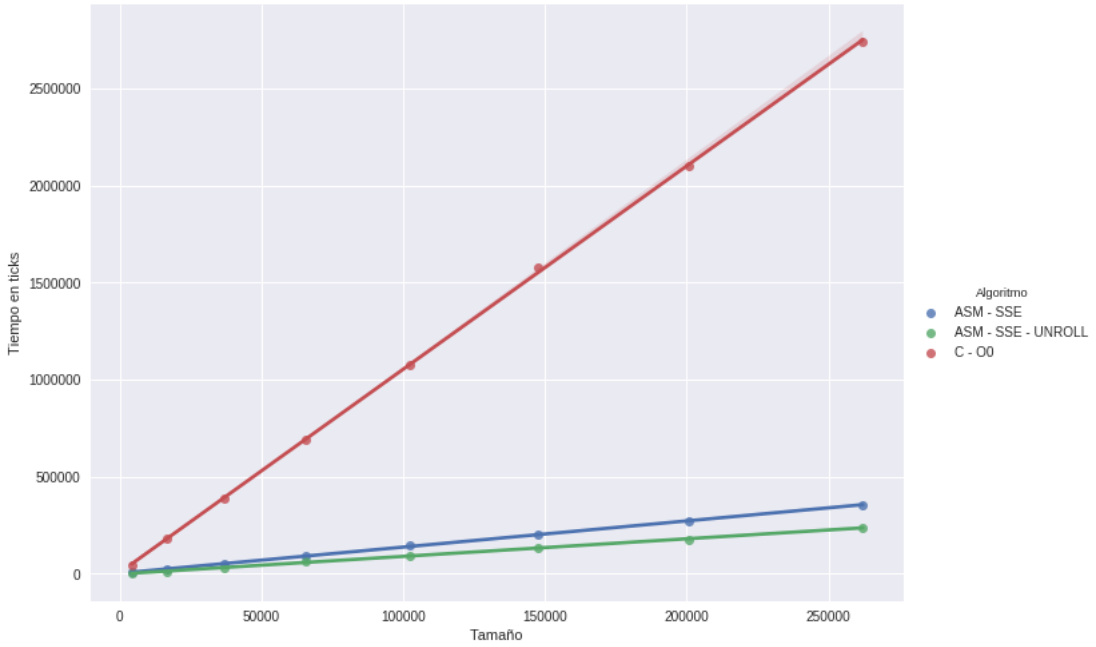


Figura 4: Versión Unroll del algoritmo puesta en comparación del entregado como respuesta del tp

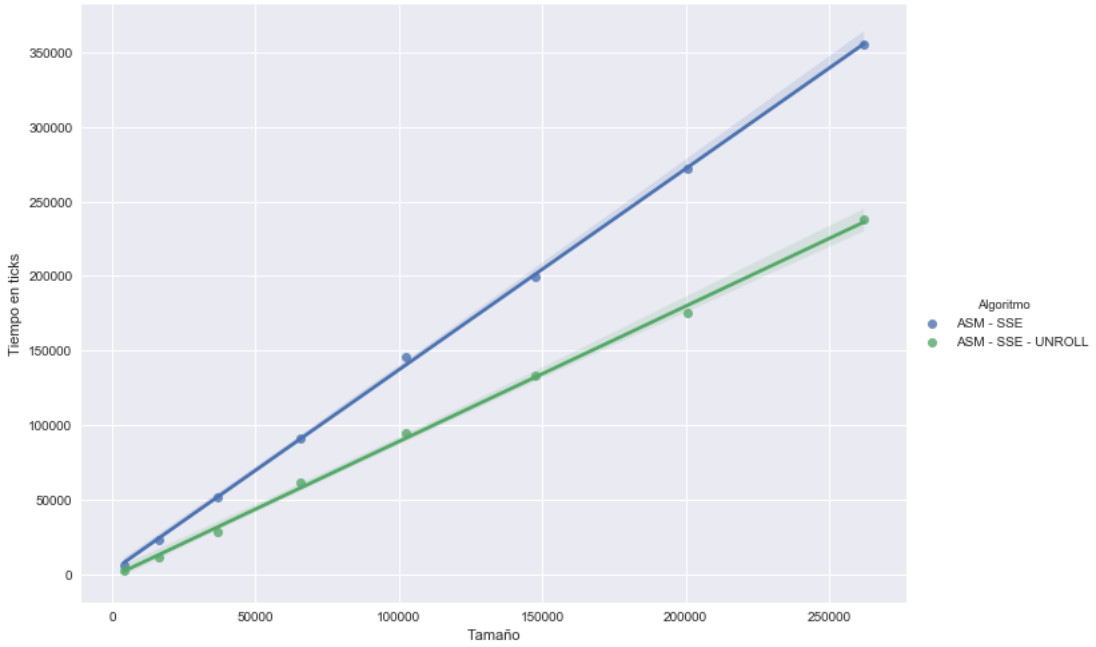


Figura 5: Comparación exclusiva de las implementaciones en ASM

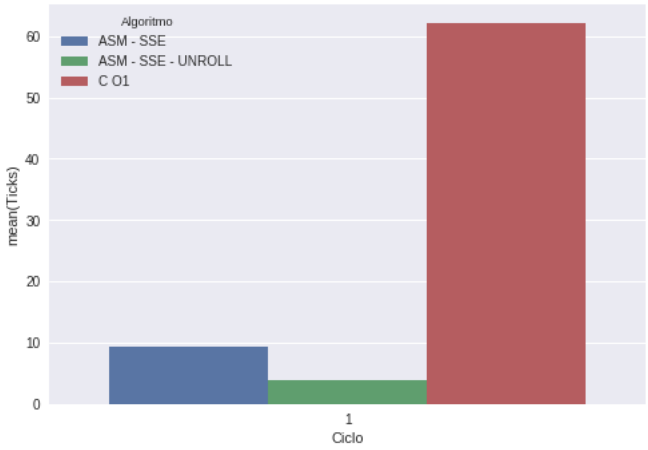


Figura 6: Ticks en un ciclo comparación de las 3 implementaciones

Explicación e interpretación de los resultados obtenidos

Podemos distinguir de la figura 3 que hay una mejora a medida que la imagen se va haciendo cada vez mas grande en el algoritmo de desarrollo del ciclo, esto nos estaría indicando a simple vista que el método funciona, podríamos tener alguna desventaja ya que nuestro programa estaría pesando mas de lo normal, pero esto no sería algo que tenga importancia en un programa como el desarrollado. También se puede hablar de la comparación en imágenes chicas viendo la figura 4. En este caso se trabajo con los resultados mas chicos del set de resultados para poder ver que tan distintos o iguales eran los algoritmos y se puede notar que en imágenes chicas no hay casi diferencia de performance, esto tiene sentido pensando que en imágenes chicas no existe una cantidad de saltos que pueda llegar a influir en la predicción de saltos del procesador.

Por ultimo una manera interesante de ver la diferencia de la performance que se nos ocurrió fue la de medir un ciclo de cada algoritmo y ver que tan performante era cada uno luego de hacer la modificación a esto como se indica en la sección de çomo y que van a medir"se puede observar otra manera como la implementación de UNROLL lograr obtener un tiempo que es al menos dos veces mejor que el método puesto en nuestro TP. Este resultado parece contradecir los anteriores dos ya que en estos no se observa que los resultados del UNROLL sean dos veces mejores. No fue testeado pero mi hipótesis recae a que el trabajo que se debe hacer cada vez que se cambia de fila es mayor cuando estamos haciendo el UNROLL , tal vez por la misma ineficiencia de la implementación. Pero este resultado nos muestra el potencial que tiene el UNROLL al poder tener un mejor tiempo de procesamiento por píxel.

Por ultimo voy a hablar de mi test sobre la cache sobre el cual solo quedan los resultados y no hay gráfico asociado . Temo que la hipótesis que teníamos no se pudo probar ya que los mejores tiempo tanto para el código ASM original como para el ASM - UNROLL son los siguientes : 243442734 ticks y 220905687 ticks respectivamente , hay una mejora de al menos 10 % en promedio , cuando se esperaba que al poder leer por filas en nuestro ASM original se podría conseguir una hit rate mas grande.

2.6. Conclusiones

En conclusión el unroll aplicado al algoritmo original logro aumentar la performance mas denotada en imágenes mas grandes. El poder procesar mas píxeles hace que el ciclo sea mas corto y por lo tanto es ahí donde gana tiempo. El algoritmo se podría mejorar para evitar perder tanto tiempo cuando se cambia de fila.

Relación entre las hipótesis de trabajo y resultados

3. Zoom

El filtro LinearZoom consiste en interpolar linealmente los pixeles de la imagen, duplicando su ancho y su alto, es decir aumentando su tamaño efectivo en 4.

La interpolación lineal de los pixeles consiste en calcular los promedios de cada componente entre los pixeles adyacentes al nuevo pixel interpolado. La misma se puede representar con la siguente matriz:

A	B
C	D

$\xrightarrow{LinearZoom}$

A	(A+B)/2	B	$\cdots$
(A+C)/2	(A+B+C+D)	(B+D)/2	$\cdots$
C	(C+D)/2	D	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

En el caso de la última fila y la última columna, como no existe información para interpolar, los elementos deben ser duplicados.

El filtro en sí no presenta un efecto visual muy notorio, pero el tamaño de la imagen crece.

3.1. Implementación

La implementación de este algoritmo requirió un enfoque distinto al que tomamos con los demás: ya que la ultima fila debe copiarse hacia abajo, nos resultó más facil recorrer la imagen de arriba hacia abajo, es decir, de los valores más altos de la matriz de pixeles. Se hace esta salvedad por el hecho que el formato proporcionado de imagen guarda las filas de inferior a superior. En los otros algoritmos esta distinción no se hizo ya que no fue necesaria.

En la implementación con SIMD, podemos aprovechar las operaciones vectoriales para operar no solo sobre las múltiples componentes a la ves, sino también sobre más de un pixel (como son sumas de hasta 4 bytes, solo requerimos tamaño word por componente).

$\texttt{XMM}_3$	A	B
$\texttt{XMM}_4$	C	D

`PADDW` $\texttt{XMM}_3$, $\texttt{XMM}_4$

$\texttt{XMM}_3$	A+C	B+D

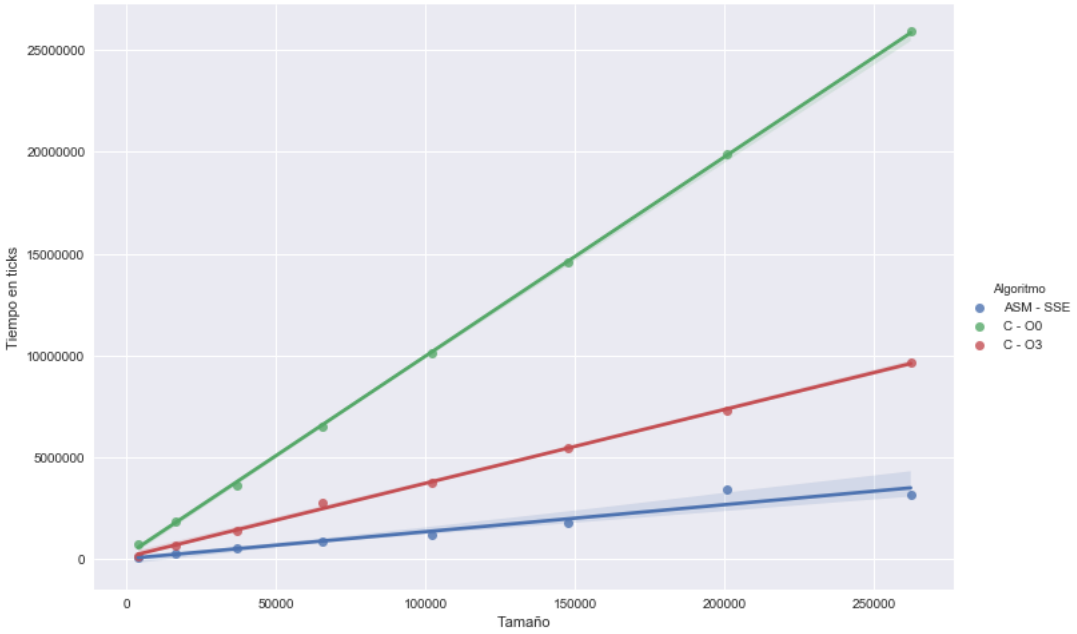
`PSRLW` $\texttt{XMM}_3$, 1

$\texttt{XMM}_3$	(A+C)/2	(B+D)/2

3.2. Análisis preeliminar

Comparación de rendimiento de ASM vs C

Dado que la interpolación lineal es un procesamiento muy conocido de imágenes, nos esperabamos que fuese uno de los filtros más beneficiados por la vectorización de los cálculos.



Efectivamente, nuestros análisis dieron que incluso con las optimizaciones de GCC, la vectorización de la interpolación aumenta el rendimiento de manera dramática, ejecutando el filtro siempre en menos del 50 % del tiempo.

4. Maximo cercano

Este es un filtro en el que para cada pixel de la imagen original, se genera un pixel nuevo buscando cual es el máximo valor de cada componente en los píxeles de alrededor, y con este nuevo pixel hacemos una combinación lineal con el original, y un parámetro que nos dan y lo ponemos en la imagen destino en la misma posición que el original. La búsqueda de las componentes máximas, se hace sobre un kernel de 7x7 píxeles centrado en el pixel que estamos interesados en cambiar.

	P_{00}	P_{01}	P_{02}	P_{03}	P_{04}	P_{05}	P_{06}
	P_{10}	P_{11}	P_{12}	P_{13}	P_{14}	P_{15}	P_{16}
	P_{20}	P_{21}	P_{22}	P_{23}	P_{24}	P_{25}	P_{26}
	P_{30}	P_{31}	P_{32}	P_{33}	P_{34}	P_{35}	P_{36}
	P_{40}	P_{41}	P_{42}	P_{43}	P_{44}	P_{45}	P_{46}
	P_{50}	P_{51}	P_{52}	P_{53}	P_{54}	P_{55}	P_{56}
	P_{60}	P_{61}	P_{62}	P_{63}	P_{64}	P_{65}	P_{66}

Cuadro 1: En rojo el pixel que estamos editando y en amarillo los píxeles que estan dentro del kernel

4.1. Implementación

Para la implementación de este filtro, recorremos la imagen original, iterando sobre sus filas y sus columnas, como hay píxeles que no tenemos un kernel de 7x7 alrededor, estos los pintamos de blanco, pero si podemos, iteramos sobre el kernel y nos vamos fijando cuáles son las componentes máximas y cuando recorrimos todo el kernel, para cada componente hacemos esta cuenta:
Componente Destino ← Componente Original * (1 - VAL) + Componente Máxima * VAL. (Donde VAL es el parámetro que nos pasan en la función).

Al implementar este filtro en lenguaje ensamblador, podemos aprovechar de las ventajas que nos brinda el modelo SIMD. En particular, los registros XMM son de 16 bytes, que los podemos utilizar para procesar 4 píxeles en paralelo. Para esta implementación, vamos a aprovechar estos registros para buscar el máximo sobre el kernel y hacer la combinación lineal sobre cada componente en paralelo.

Como dijimos recorremos las columnas y filas, primero nos fijamos si es una fila que tenemos que pintar de blanco, en caso afirmativo, sabemos que toda esa fila va a hacer blanca, entonces podemos aprovechar los registros XMM para guardar en memoria múltiples pixeles y como nos entran 4, podemos generarnos un registro XMM que contenga 4 píxeles blancos y guardarlos en memoria en la imagen destino a la vez.

Ahora sí es un píxel que debemos calcularlo, tenemos que iterar sobre el kernel y buscar el máximo de cada color. Lo que podemos hacer para aprovechar SIMD, es cargar 4 píxeles en un registro y otros 4 en otro, ahora si aplicamos la instrucción PMAXUB, que compara byte a byte entre los registros y guarda el máximo de los dos en el destino. Entonces lo que ganamos con esto es que en el registro destino nos quedó 4 píxeles que cada tiene el máximo de cada componente entre el pixel que estaban en la misma posición de los 2 registros.

$\texttt{XMM}_1$	P_1	P_2	P_3	P_4
$\texttt{XMM}_2$	P_5	P_6	P_7	P_8

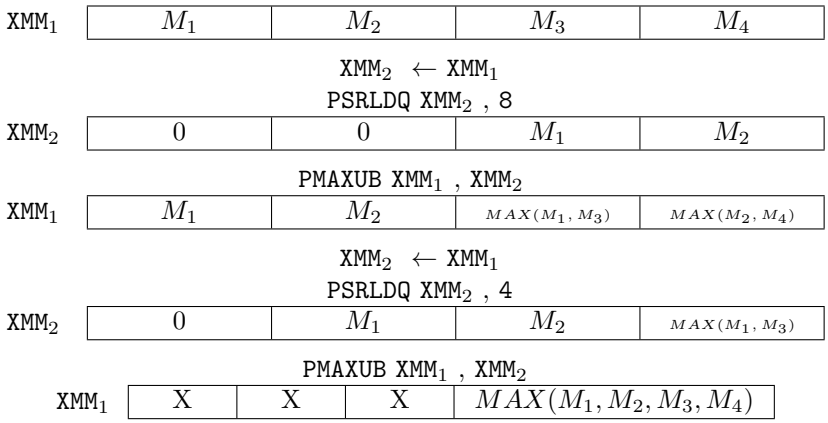
$\texttt{PMAXUB XMM}_1\ ,\ \texttt{XMM}_2$

$\texttt{XMM}_1$	$MAX(P_1, P_5)$	$MAX(P_2, P_6)$	$MAX(P_3, P_7)$	$MAX(P_4, P_8)$

$MAX(PM, P) =$

$MAX(PM^r, P^r)$	$MAX(PM^g, P^g)$	$MAX(PM^b, P^b)$	$MAX(PM^a, P^a)$

Usando esta metodología podemos mantener un XMM que contenga los píxeles máximos y lo comparamos contra los del kernel, actualizando este registro. Pero como cada fila del kernel tiene 7 píxeles contiguos, hacemos esta técnica dos veces, pero estaríamos comparando 8 pixeles, así que repetimos un píxel en cada paso asi comparamos todos los píxeles. Hacemos esto para cada fila del kernel y nos termina quedando un registro con 4 posibles máximos, luego debemos compararlos entre sí. Para ello copiamos el registro donde tenemos los posibles máximos, shifteamos a la derecha uno de ellos, 8 bytes shifteamos para que nos quede desplazado 2 pixeles. Y volvemos a comparar, ahora nos quedan solamente 2 y de vuelta desplazamos los pixeles pero ahora solamente 4 bytes, y comparamos de vuelta, quedandonos el píxel máximo.

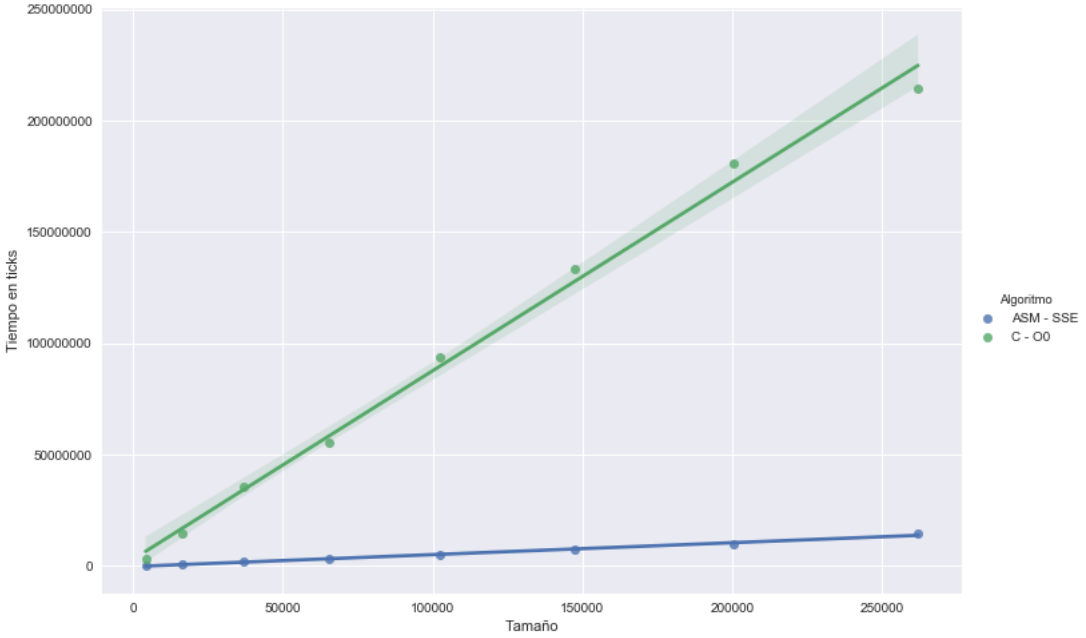


Ya conseguido el máximo, ahora tenemos que realizar la combinación lineal con el píxel original, el nuevo píxel que generamos y el parámetro. Lo que queremos es hacer la cuenta en paralelo, para eso, podemos usar instrucciones SIMD para multiplicar las componentes de los píxeles en paralelo por el parámetro, que es un float, entonces convertimos a float cada componente quedándonos todas las componentes en un registro XMM, y para multiplicarlas por el parámetro tenemos que hacer un registro XMM que lo contenga 4 veces y aplicamos MULPS, quedando de resultado la multiplicación por cada componente contra el parámetro cada una en floats. Luego hacemos lo mismo con el píxel original, y sumamos estos dos resultados. Ahora nos queda guardar este resultado en la imagen destino, pero antes, debemos convertir los floats a integers.

4.2. Análisis preeliminar

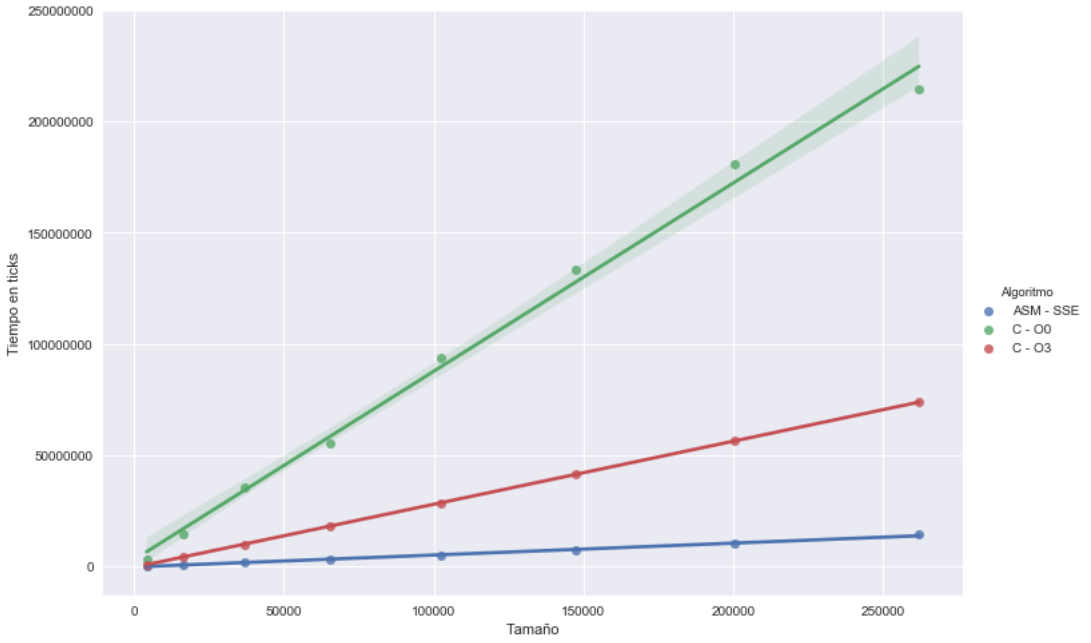
Comparación de rendimiento de ASM vs C

Como en los otros filtros, corrimos una serie de tests para las mismas imágenes en diferentes tamaños. Primero vamos a comparar la implementación de C contra la de ASM.



Podemos ver claramente como el filtro implementado en ASM corre más rápido que en C, de hecho, al incrementar el tamaño de la imagen la diferencia es aún más notable. Esto es un comportamiento esperable ya que en C estamos yendo a buscar el pixel a memoria por cada pixel del kernel, en cambio en ASM con las instrucciones SIMD cada 7 pixeles lo pedimos 2 veces. Además, en ASM aprovechamos y hacemos las cuentas para el pixel destino en paralelo.

A la hora de compilar el código en C, se pueden aplicar optimizaciones, el compilador interpreta el código y trata de mejorar nuestro codigo potencialmente. Una de estas es aplicando el flag O3 a la hora de compilar. Comparemos el mismo set de test anterior pero ahora con el flag O3 activado.

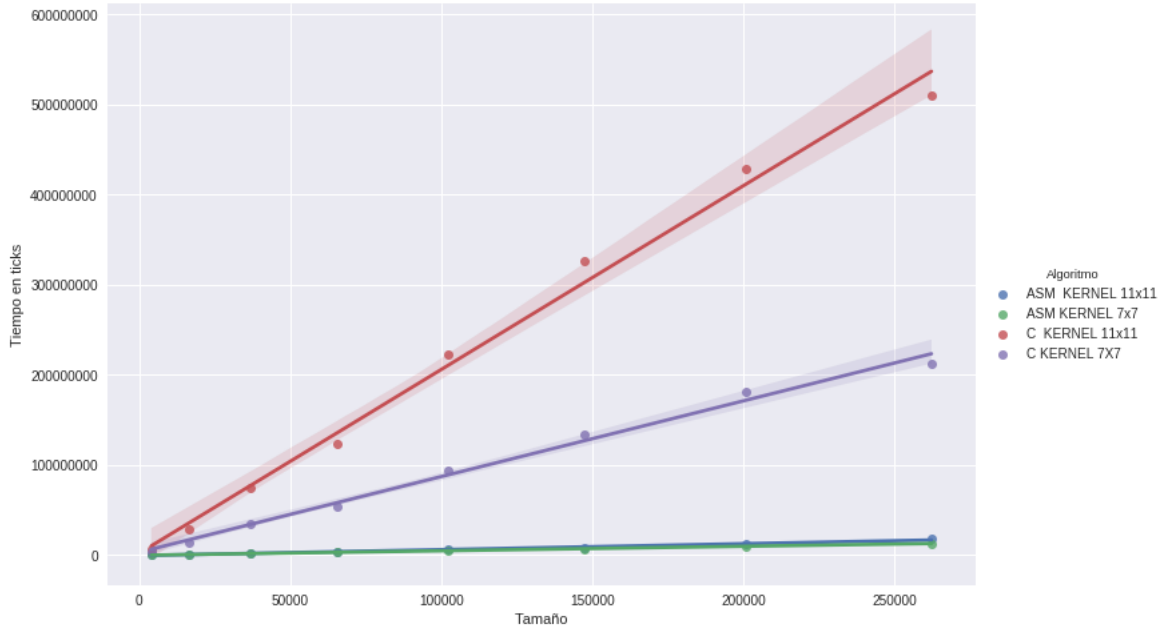


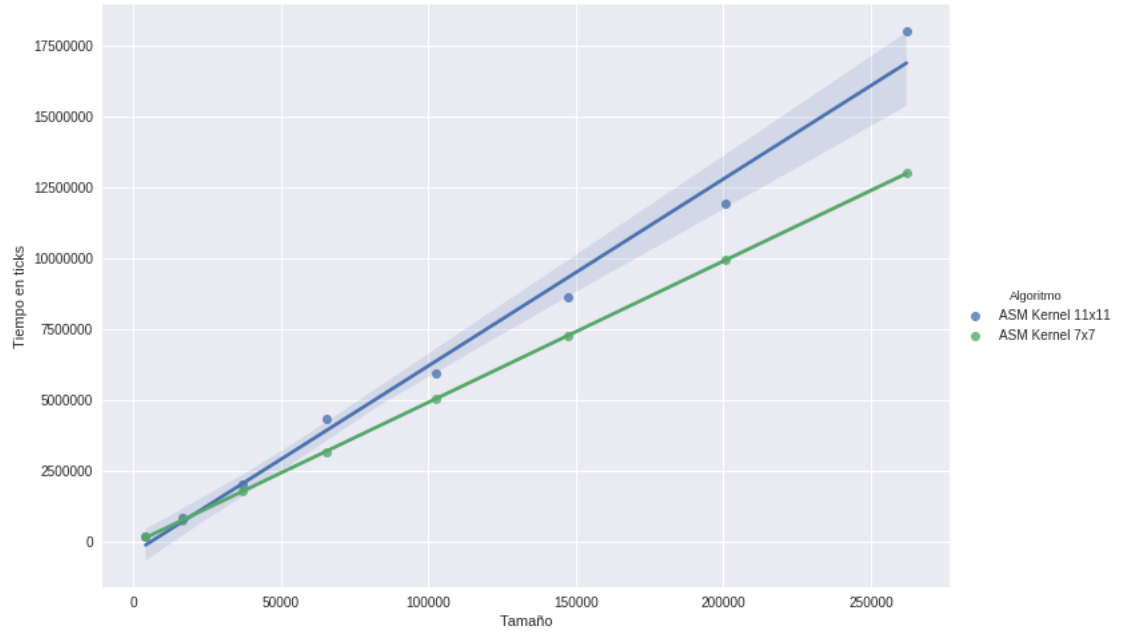
Si bien hay una amplia mejora en la implementación de C sin el flag O3 contra la que tiene el flag activado, no llega a mejorar la versión de ASM.

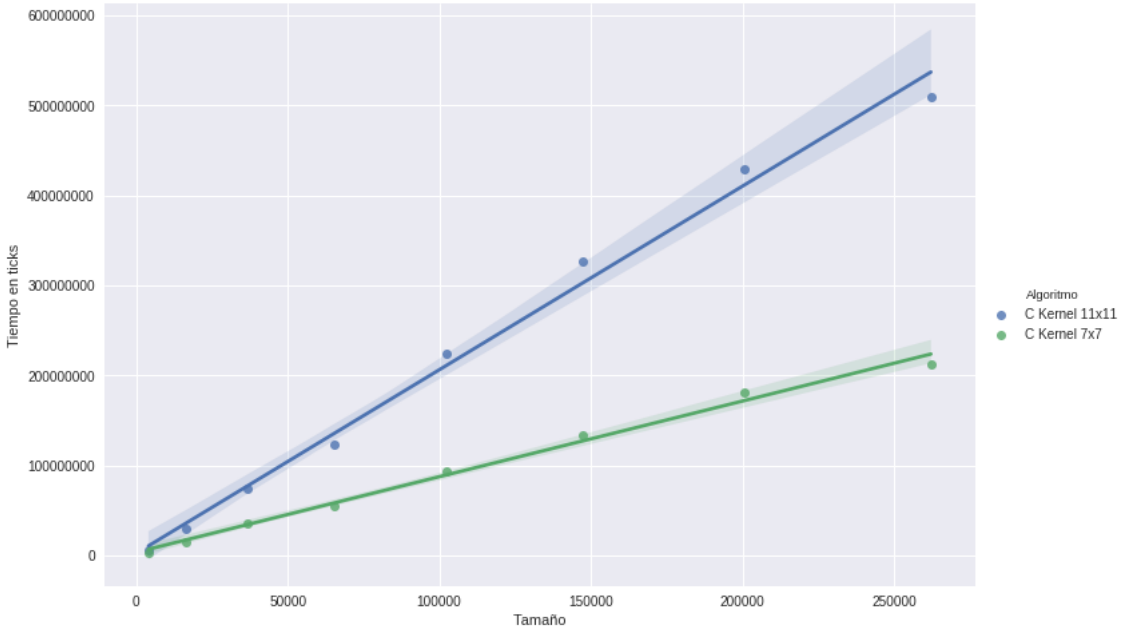
4.3. Experimantación

Queremos ver cómo impacta en ambas implementaciones si agrandamos el tamaño de kernel. Vamos a correr los mismos tests que antes, pero ahora en vez que del kernel sea 7x7 será de 11x11.

Nuestra hipótesis, es que esta medida va a afectar con más contundencia a la implementación de C, si bien en ASM va a tardar más, en relación al kernel mas chico, no le va afectar tanto. Esto es debido a que, en C trae cada pixel del kernel uno por vez, y como son casi 2.5 veces más, la cantidad de píxeles en el kerenel, esperamos una performance peor en este orden. Pero en ASM aprovechando la paralelización de datos, levantamos 11 píxeles con 3 llamadas a memoria nada más.







Si observamos los gráficos, podemos ver como en C con el kernel de 11x11 tarda casi 2,5 veces más, como la relación de diferencia de pixeles en los kernels, y para ASM hay nada mas una diferencia de 1,2 y 1,4 para instancias más grandes. Como habíamos predicho en la hipótesis, este cambio en el tamaño del kernel, afectó bastante más a la implementación de C que a la de ASM.