DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico 2

16 de junio 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.

Para los siguientes ejemplos se utiliza el filtro RGB2YUV, pero la implementación del filtro inverso es muy similar.

Primero, se definen 3 máscaras. Cada una se utilizará para crear una de las componentes finales:

$\texttt{XMM}_9$	66	129	25	0
$\texttt{XMM}_{10}$	-38	-74	112	0
$\texttt{XMM}_{11}$	112	-94	-18	0

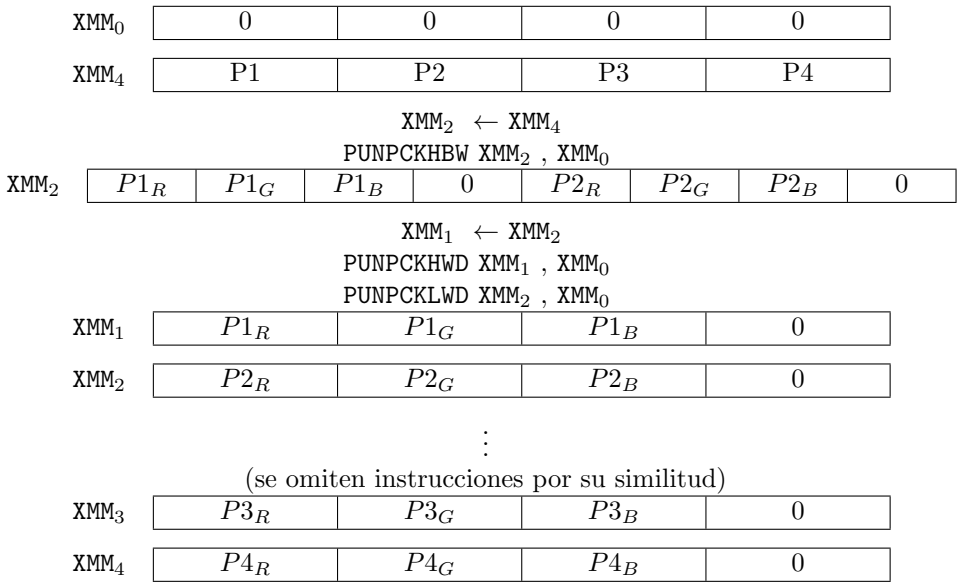
Ya que contamos con registros de 128 bits y cada pixel mide 32 bits de ancho (RGBA), podemos cargar 4 pixeles de la memoria por iteración. Esto permite reducir el impacto de los accesos a memoria y los saltos condicionales.

No obstante, los mismos son desempaquetados para ocupar los registros XMM de manera individual, con cada una de sus componentes en tamaño doubleword (32 bits). Esto es porque los valores que multiplicamos y sumamos pueden exceder el límite de una word con signo, y perderíamos precisión o retornaríamos valores inválidos. Por ejemplo, para el caso RGB2YUV, si un pixel fuese (255,255,255), la componenete Y sería

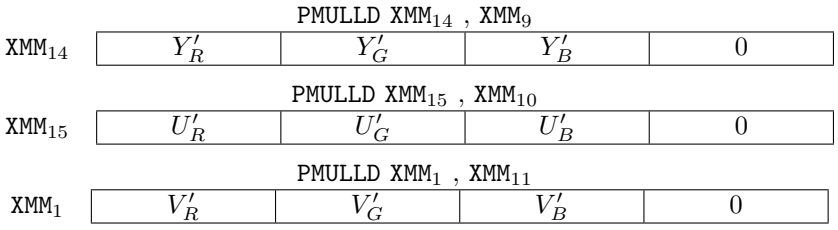
$$Y = sature(((66 * 255 + 129 * 255 + 25 * 255 + 128) >> 8) + 16)$$
$$Y = sature((220 * 255 >> 8) + 16)$$
$$220 * 255 = 56100 > 32768 = 2^{15}$$

Esto también se refleja en la implementación C, donde consideramos utilizar `short`, de 16 bits, pero esto generaba demasiados errores y nos vimos forzados a usar `int`, de 32 bits.

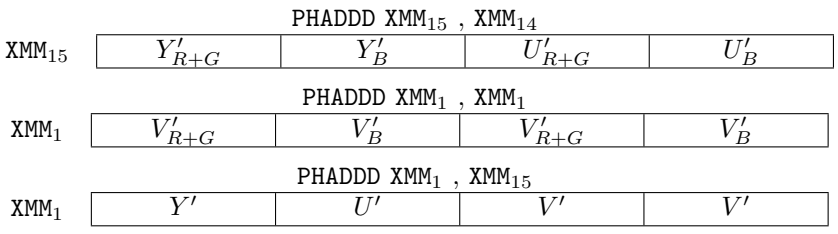
Por lo tanto, la lógica de procesamiento de cada pixel se ve repetida 4 veces, una por pixel por iteración. Sin embargo, tenemos la garantía que la imagen tendrá como ancho un múltiplo de 4, por lo que esto no presenta un problema.



Cada pixel es copiado 3 veces (en su registro original y a los registros XMM14 y XMM15), una por componente destino (en este caso, Y, U y V). Luego, se multiplican estas copias del pixel por la máscara correspondiente:



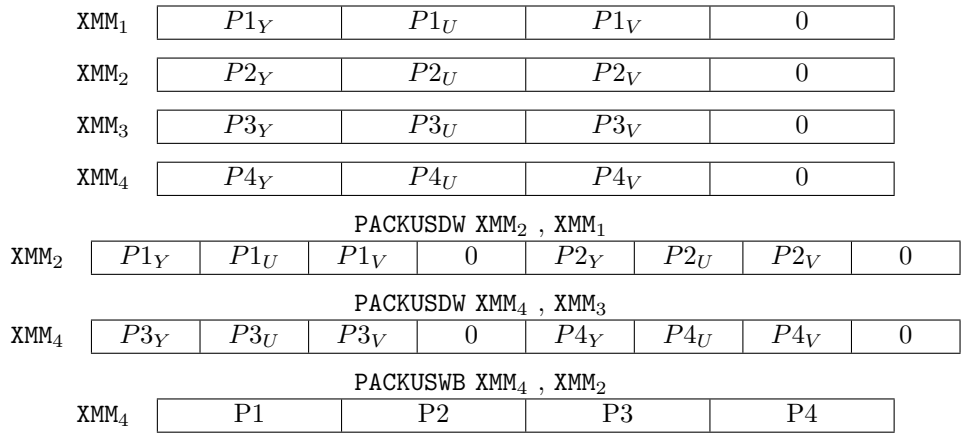
Por último, estos valores intermedios se suman horizontalmente para crear las componentes finales:



Como se puede ver, al sumar se genera un valor duplicado al final. El mismo se genera porque las imagenes tienen 3 componentes y `PHADDD` toma siempre 2 parámetros. Esto no afecta la correción del filtro, ya que no estamos trabajando con el componente alfa y el mismo puede ser descartado.

Por último, estos componentes se denotan con $'$ porque 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.

Al finalizar la conversión, los pixeles son reempaquetados para ser guardados en una sola operación. El empaquetado es sin signo, de manera que cualquier valor por encima del máximo es acotado al máximo de dicha componente (`0xFF`):



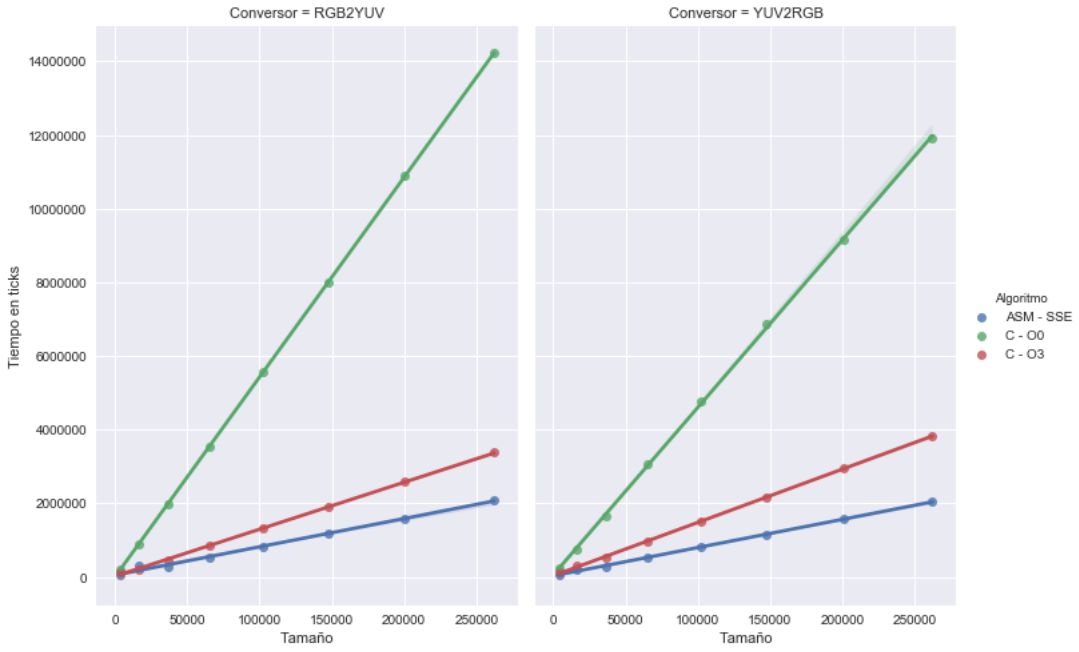
(notese que en este contexto Pi no representa el mismo pixel de entrada sino su correspondiente post-conversión)

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 preliminar

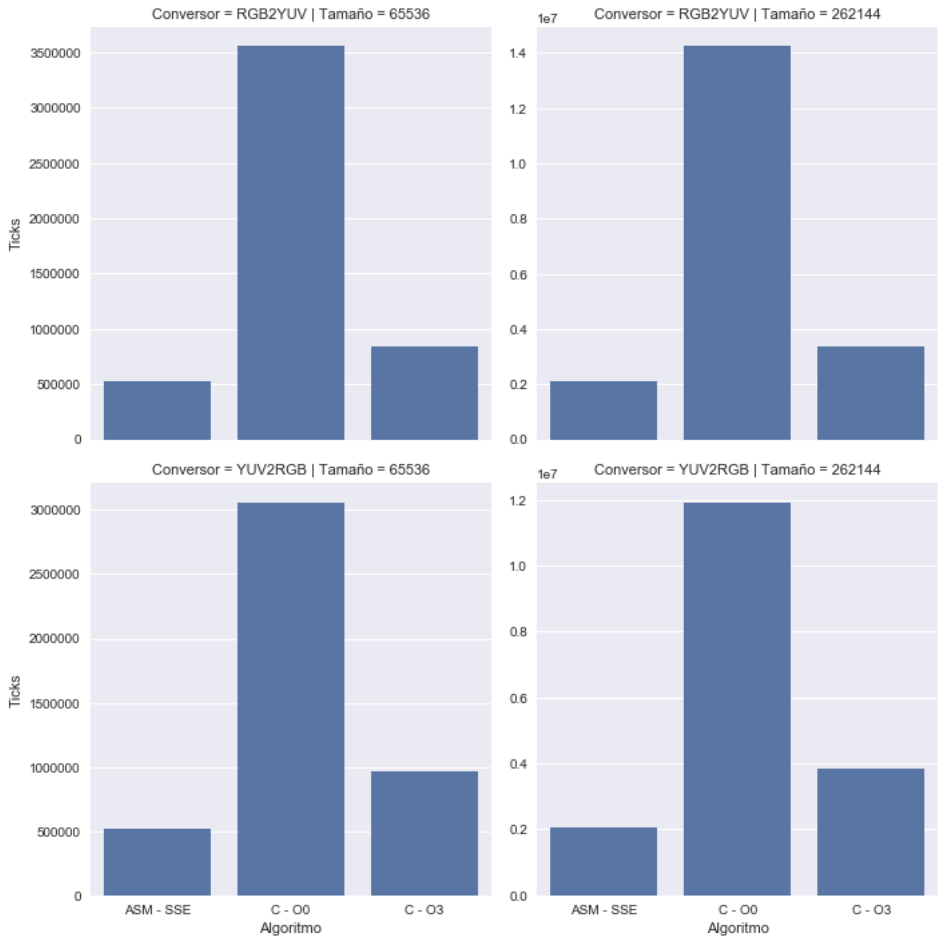
Para realizar un análisis preliminar del rendimiento de los algoritmos, debimos medir los tiempos de ejecución de los mismos. Por claridad, unificamos el criterio utilizados para las mediciones a lo largo de todo el trabajo práctico. Los detalles de dicho critero se encuentran en el apéndice correspondiente.

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



Lo primero que se puede apreciar en este gráfico es que, más allá de la implementación utilizada, el tiempo de ejecución del filtro aumenta de forma lineal con el tamaño de la imagen, en este caso medido en pixeles. Cabe destacar que en nuestras mediciones, los casos de imágenes pequeñas dieron resultados dispares, pero para el resto de los tamaños esta tendencia se mantiene.

Para continuar, se puede notar una diferencia enorme en la performance del algoritmo escrito en C contra el escrito en ASM. 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 memoria. Sin embargo, se puede ver que el algoritmo escrito con instrucciones SIMD sigue siendo más óptimo. Decidimos analizar esto más de cerca:



Este gráfico nos muestra de forma clara que el filtro escrito en ASM se ejecuta en entre 50 % y 70 % del tiempo al compararlo con el filtro en C compilado con el flag `-O3`. Si bien ambos filtros se ejecutan varias veces más rápido que la versión sin optimizaciones de compilador (`-O0`), la diferencia porcentual entre ellos es visible y relevante.

Por otro lado, se puede ver que en O0 el conversor YUV a RGB performa mejor que su contraparte RGB a YUV (con una diferencia de más de 10 %). Esto posiblemente se deba a ciertas operaciones que como mencionamos se repiten, y por ende podemos pre-calcular y reutilizar valores intermedios.

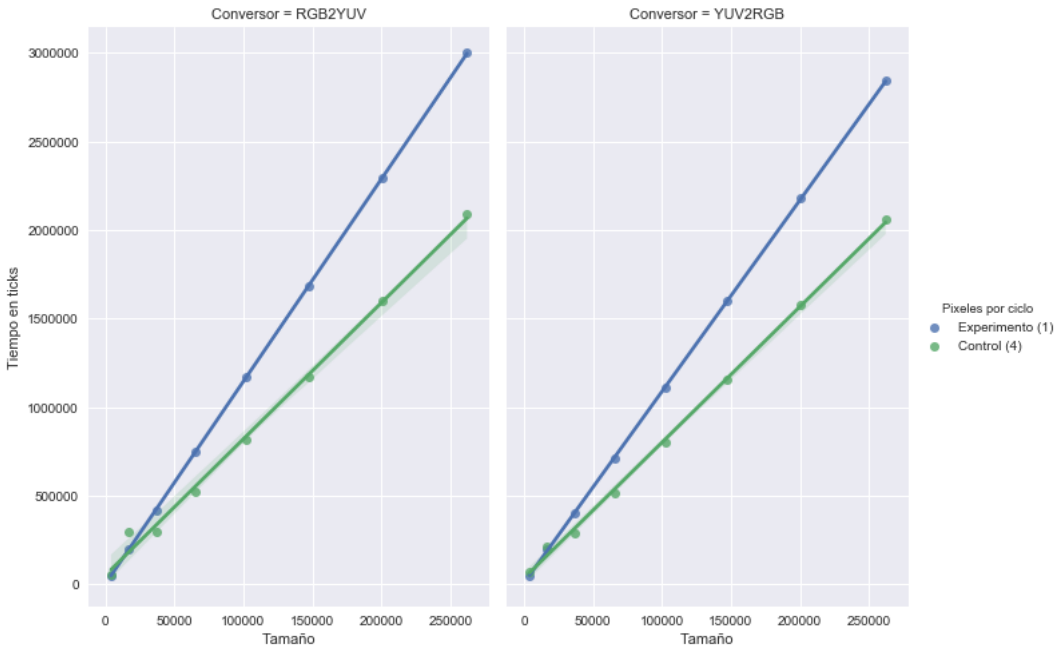
1.3. Experimentación

A modo de experimentación, decidimos probar 2 hipótesis distintas:

Hipótesis 1: cargar 4 pixeles a la vez mejora la performance

Como se mencionó en la sección de implementación, en nuestra implementación cargamos 4 pixeles de la memoria a los registros XMM, para luego desempaquetarlos y procesarlos de manera individual. Nosotros hicimos esto bajo la hipótesis que reducir los saltos y los accesos a memoria mejoraría el rendimiento de los filtros.

Decidimos comprobar esta teoría implementando los mismos filtros pero cargando y procesando 1 único pixel por iteración:



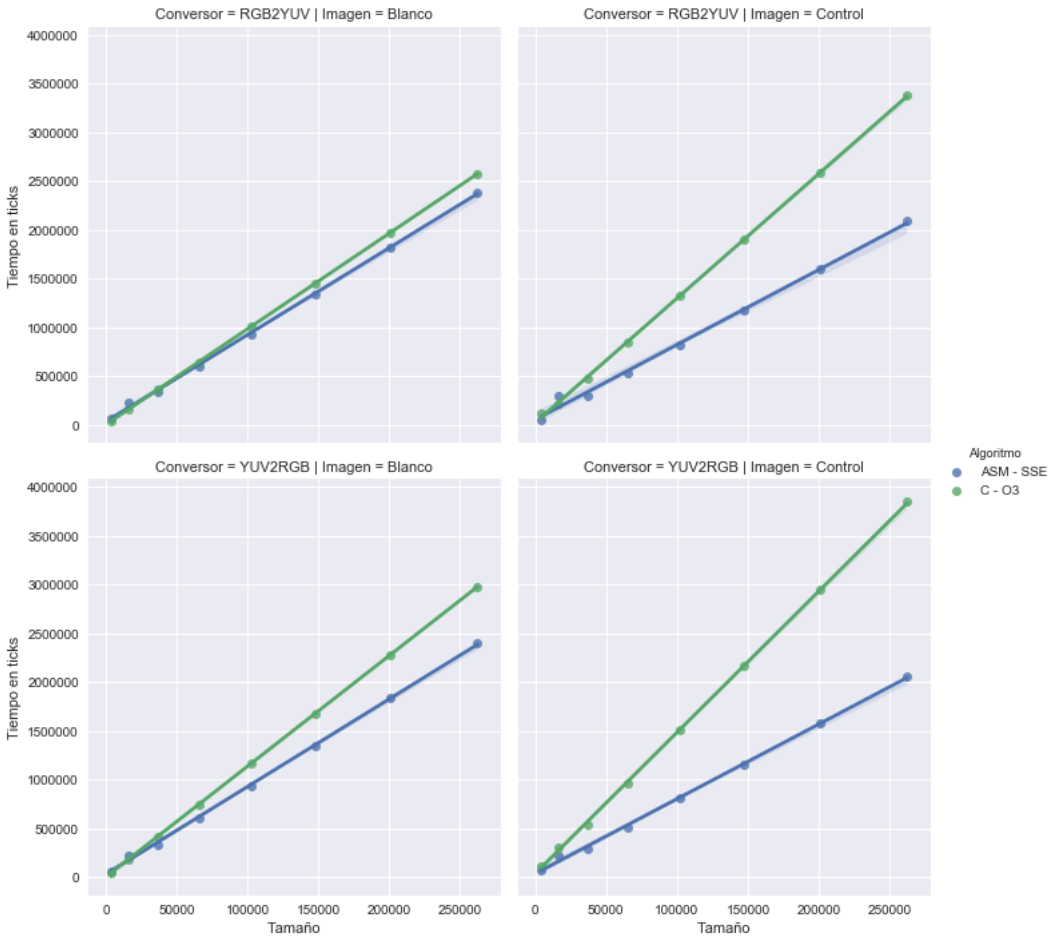
Efectivamente, las implementaciones más veloces son aquellas que aprovechan la instrucción `movdqu` para cargar 4 pixeles con un solo acceso a memoria, y que consecuentemente procesan 4 pixeles sin hacer saltos.

Por otro lado, al acceder de manera lineal a la memoria, el caché debería mitigar una gran candidad de esos accesos, así que es posible que el impacto mayor no provenga necesariamente de los accesos a memoria, sino de los saltos condicionales.

Hipótesis 2: los filtros corren en tiempos uniformes para toda imagen de mismo tamaño

De cara a la experimentación, esperábamos que los algoritmos fuesen agnósticos a la composición de la imagen de entrada, es decir, de los pixeles particulares que la componen. Si bien sabemos que el tamaño de la imagen si afecta de forma lineal el tiempo de ejecución, supusimos que los colores que componen las imágenes no modificarían dicho tiempo.

Para comprobar esta teoría, probamos correr el conversor sobre utilizando un bitmap blanco. Corrimos los mismos conversores, en ambas implementaciones:



Para nuestra sorpresa, podemos ver que la saturación de las componentes tiene un efecto visible en el tiempo de ejecución de los filtros. No solo eso, el bitmap afecta de manera positiva a la implementación en C, mientras que impacta de forma negativa a su contraparte SSE.

Este dato nos tomó por sorpresa, y aunque intentamos analizar el por qué de este impacto, no pudimos dilucidar la causa. De un modo u otro, esto muestra que la imagen de entrada pueden afectar notoriamente la performance de los filtros, negando nuestra hipótesis inicial.

2. Combinar

Este filtro consiste en cambiar la distribución de pixeles 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 pixeles de la original se encuentran aun presentes en la imagen resultante.

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 pixeles antes y luego de la aplicación del filtro son los mismos.

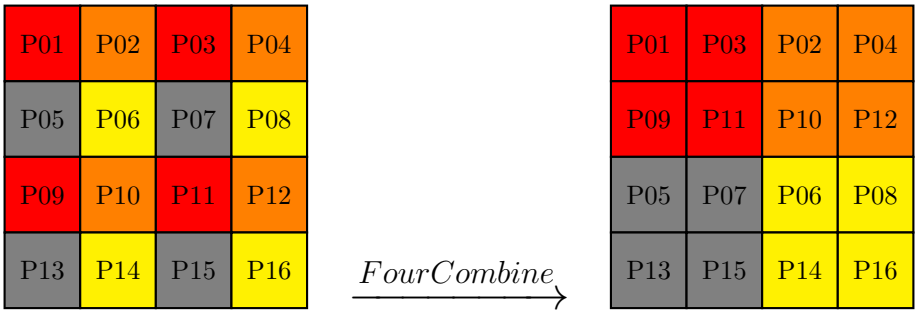


Figura 1: Distribución de pixeles tras aplicar el filtro de combinar

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.

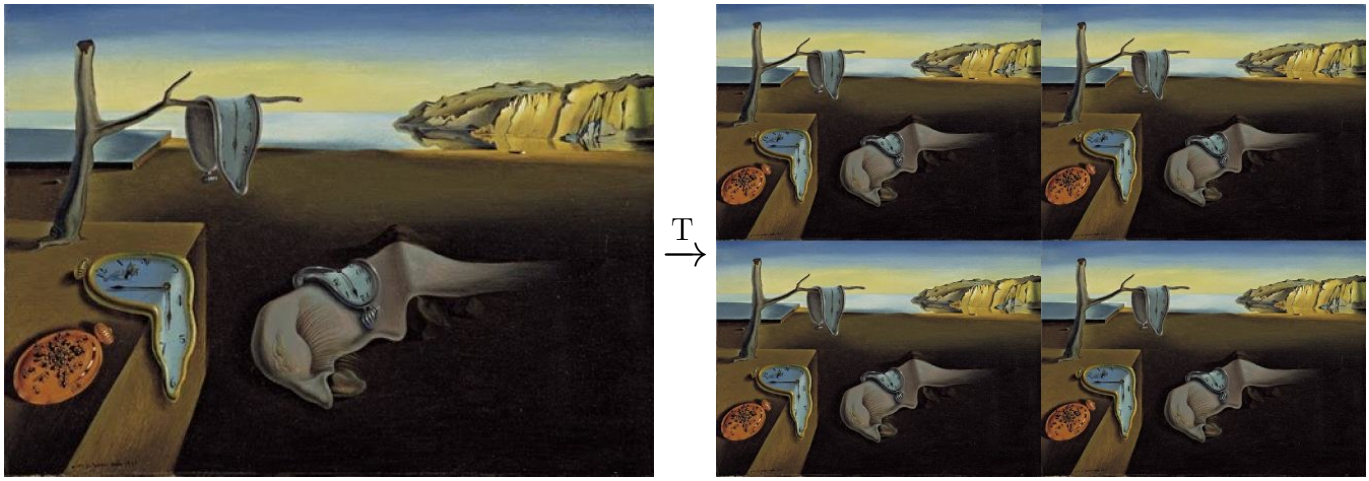


Figura 2: Imagen real antes y luego de la aplicación del filtro

2.1. Implementación

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 pixel por pixel. Cada uno de estos pixeles 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 pixeles uno por uno y calcular su lugar correspondiente no seria posible. Se tomaron las siguientes decisiones de modelado:

- Al decidir con cuanta información a la vez íbamos a trabajar, se llegó a la conclusión que lo mejor sería trabajar con 8 pixeles a la vez. Esta decisión se tomo puesto que eligiendo los 8 pixeles del lugar correcto podíamos llegar a armar 8 pixeles que iban a ser puestos en la imagen de salida. Esto nos debería dar un ahorro en la cantidad de veces que vamos a acceder a memoria.

- Al decidir entre la posibilidad de agarrar 16 pixeles que este en la misma fila o que estén la misma columna, nos pareció lo más fácil de implementar y lo más efectivo a la hora de usar la cache era que tomáramos 8 pixeles contiguos. Se vera más adelante en un experimento la diferencia entre ambos.
- Ya que íbamos a tomar de a 8 pixeles 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 bifurca

Ahora hablemos un poco más 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 pixeles desde el puntero RDI, agarrando primero cuatro y guardarlo en $XMM2, XMM4$, moverse y agarrar cuatro más y guardarlo en $XMM3, XMM5$ 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 pixeles 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 pixeles 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. Ya que podrían estar apuntando a un lugar del cuadrante 1, 2 o bien al 3, 4.

Entre el comienzo de una fila y la siguiente, se realiza el siguiente cambio en los registros $R8, R9, R10, R11$:

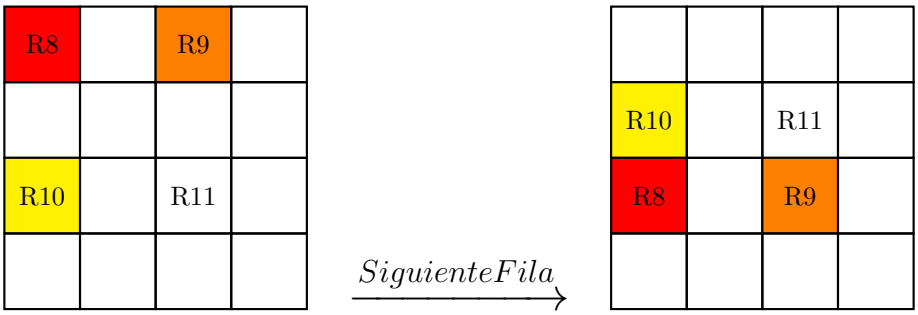


Figura 3: Lugar que ocupan los registros $R8, R9, R10, R11$ antes y luego del cambio de fila

- El $R8$ toma el valor de $R9$

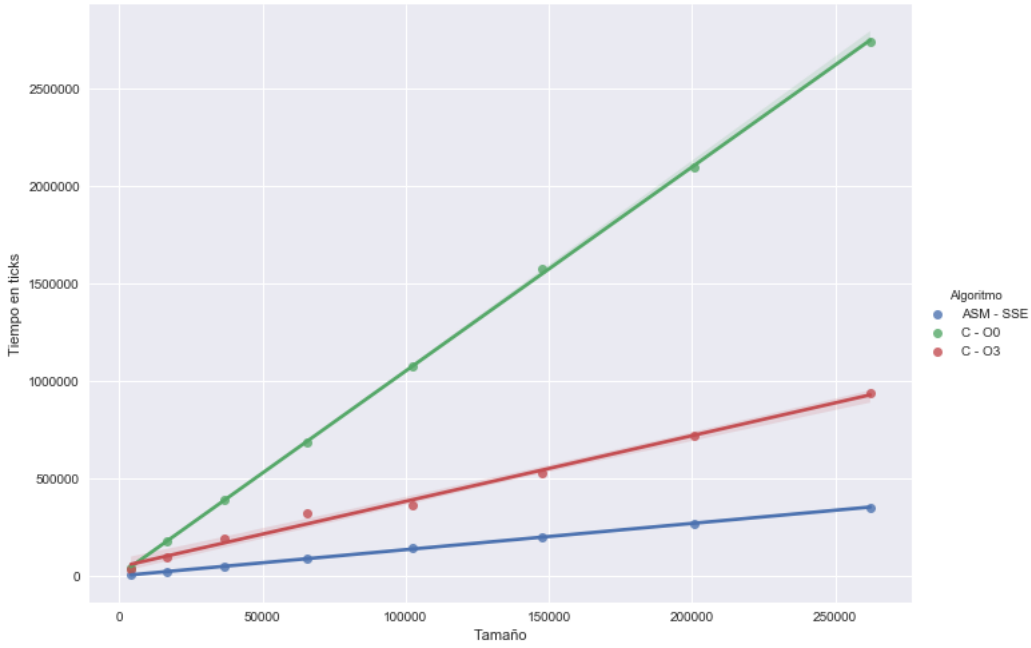
- El valor de $R9$ se el valor de la mitad de una fila
- Se switchea $R8R9$ por $R10R11$ respectivamente

Snippet del código del switch:

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

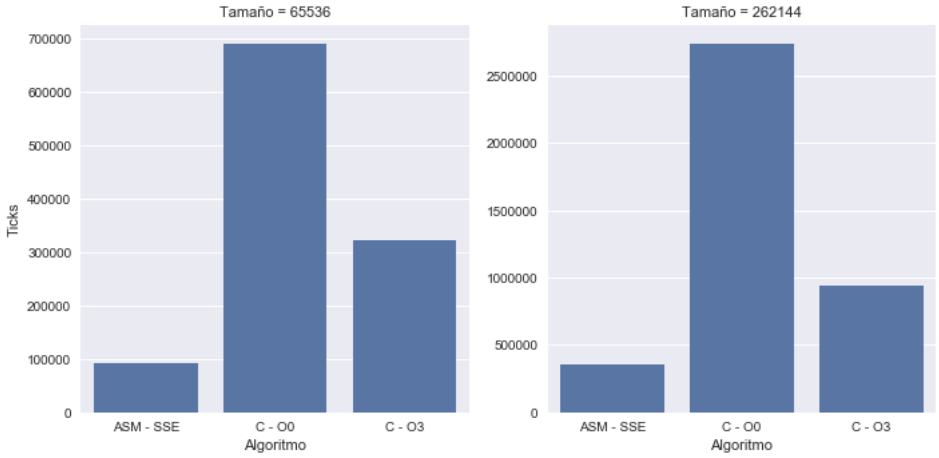
Lo único que cambia de esta explicación con respecto a cuando el ancho no es múltiplo de 8 es en este punto donde tenemos que cambiar de fila, ya que nos quedaron 4 pixeles sin procesar, se procesan manualmente por separado antes de pasar a la siguiente fila.

2.2. Análisis preliminar



Al igual que los conversores, el tiempo de ejecución del filtro es una función lineal del tamaño de la imagen en pixeles. Este comportamiento era esperado, ya que los mismos solo deben moverse a la memoria destino, lo cual es una operación de tiempo constante.

Nuevamente, algunas mediciones con imágenes más pequeñas tienen ruido visible, pero la pendiente de crecimiento del gráfico se mantiene.



En términos de performance, podemos ver que la versión implementada con SSE opera en aproximadamente 45 % del tiempo que la versión optimizada de C.

Para este caso particular, decidimos analizar el ensamblador generado por `gcc` tanto para `-O0` como para `-O3`. Para esto usamos la herramienta Compiler Explorer (https://godbolt.org/), que nos permitió analizar el código y sus cambios en vivo, asociando cada linea de código C con su correspondiente en ASM.

Una linea en particular que nos llamó la atención fue la siguiente

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

Cuando no se activa ninguna optimización, $h >> 1$ se calcula cada vez que se necesita, en cada iteración del ciclo. En cambio, al activar las optimizaciones, ese valor se calcula y almacena antes de comenzar el ciclo. Existen otras mejoras pero el nivel del código no nos permitió apreciar grandes diferencias.

2.3. Experimentación

Hipótesis 1: cargar 16 pixeles a la vez mejora la performance

Tratamos de llevar a cabo en este conjunto de experimentación la idea de Loop unrolling, con alguna pequeña modificación. Esta técnica se suele utilizar cuando la cantidad de ciclos es estática y conocida, lo cual no aplica a este caso (depende del ancho y alto de la imagen). Lo que realizamos fue el unroll de un ciclo, es decir cada ciclo opera sobre el doble de datos, logrando asi reducir la cantidad de ciclos a la mitad. Pasamos a procesar 16 pixeles por ciclo distribuidos en dos filas; de esta manera solo habría que tener en cuenta que las filas sean pares para poder aplicar esta mejora al algoritmo, sin modificar las consideraciones a tener sobre el ancho ya explicadas.

A la hora de ver que tipo de experimentos hacíamos evaluamos dos aspectos:

- 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 pixeles 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 pixel a la vez.

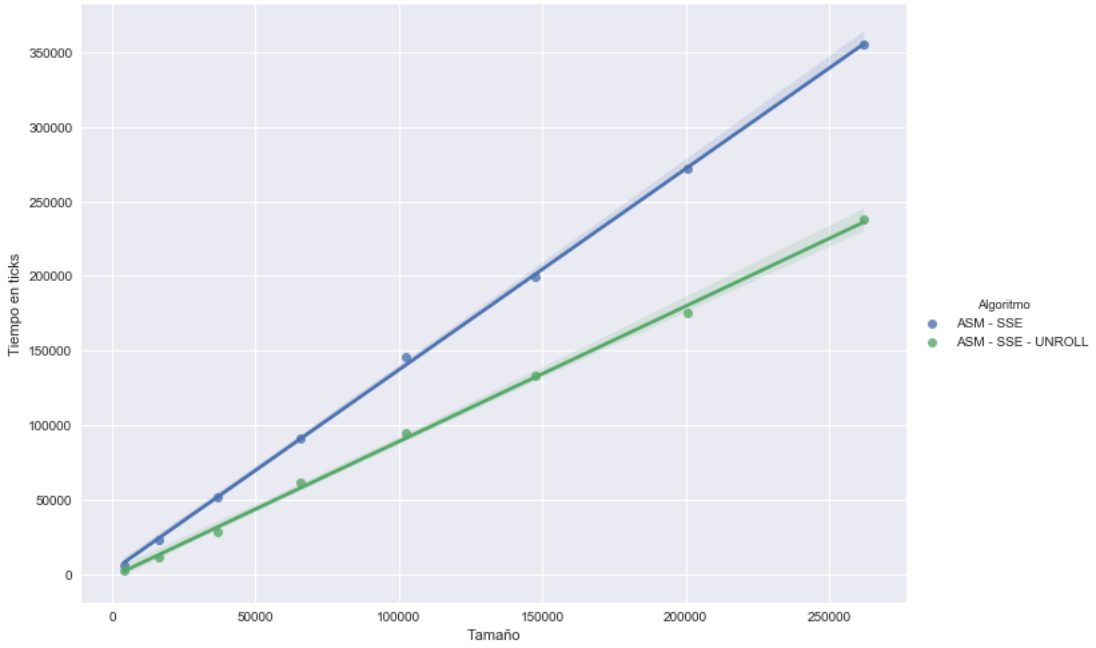


Figura 4: Comparación de las versiones de control y con Loop Unrolling

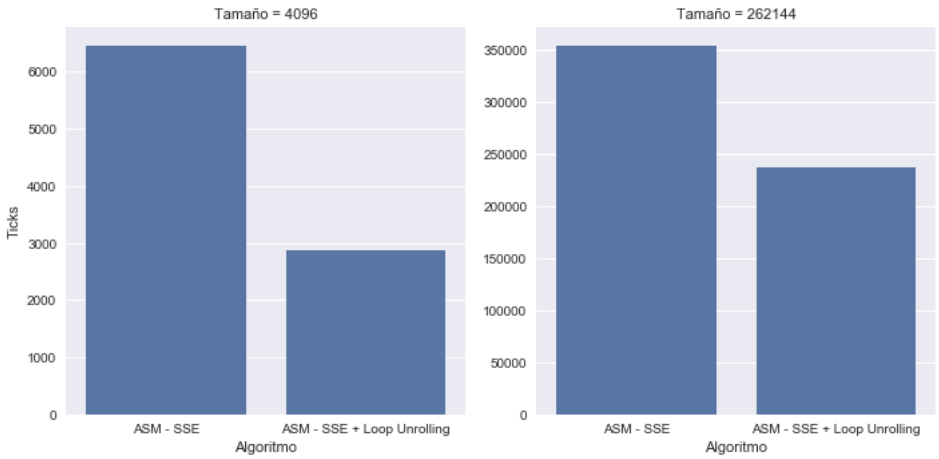


Figura 5: Comparación de casos borde de ambas implementaciones

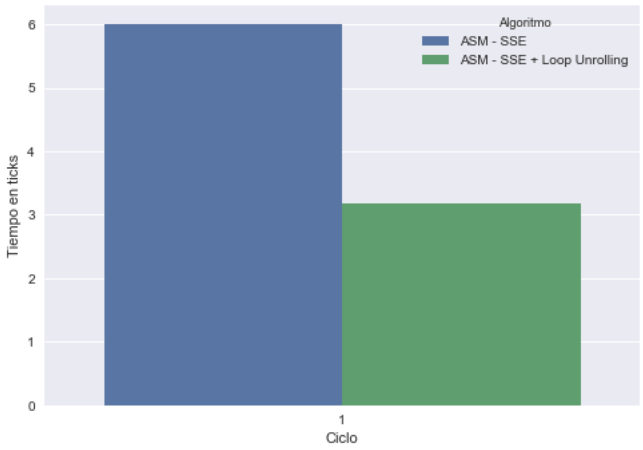


Figura 6: Ticks en un ciclo comparación de las 3 implementaciones (ajustados a los pixeles procesados)

Podemos distinguir de la figura 4 que efectivamente el desenrollado del ciclo en 2 resultó beneficioso para el filtro. Esto es similar a lo que observamos en el caso de los conversores, donde se realizó el experimento opuesto, con la salvedad que aquí no contamos con una cantidad conocida de ciclos para enrollar/desenrollar. Si miramos más de cerca en la figura 5, podemos ver que las imagenes pequeñas también se benefician de esta modificación, aunque las mismas cuenten con menos saltos condicionales, que es lo que este método optimiza.

Por último, decidimos analizar el tiempo que toma cada implementación en promedio para procesar la misma cantidad de pixeles. Para esto, medimos el tiempo de cada ciclo en ambas implementaciones, y luego dividimos por 2 los tiempos en el caso del algoritmo desenrollado, para ajustarlo al algoritmo de control. Los resultados se presentan en la figura 6. En la misma se puede observar que el tiempo que el ciclo desenrollado utiliza para la cantidad de trabajo equivalente a la versión de control es sustancialmente menor, casi la mitad del tiempo. Este resultado nos resulto poco intuitivo: si bien nos esperabamos una mejoría por haber evitado saltos condicionales, no nos esperabamos que fuese tan distinto. Consideramos que también sea más performante por el trabajo que implica el cambio de fila, que en esta versión se evita al procesar 2 filas a la vez. De un modo u otro, este resultado nos muestra definitivamente que el Loop Unrolling es una herramienta muy poderosa, mejorando ampliamente el tiempo de procesamiento por pixel.

Hipótesis 2: usar registro AVX mejora la performance

Otra idea que había surgido para probar en un experimento fue la de usar registros más grandes (AVX) para poder lograr así una reducción a la hora de mezclar los elementos. Se había pensado que teniendo muchos más pixeles juntos iba a resultar más 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 más performante que el propuesto como solución.

Hipótesis 3: el código presentado hace buen uso de la memoria cache

Nos propusimos además cambiar el código para lograr así demostrar que nuestro algoritmo presentado al leer fila por fila logra una mejora en cuando a la administración de la memoria cache.

Los cambios en el código para lograr esto fueron:

- En vez de procesar los pixeles fila por fila lo hicimos columna por columna. De esta forma esperamos que en imágenes grande la cantidad de misses en la memoria cache
- Procesar la misma cantidad de pixeles por ciclo en ambas soluciones. Asi podemos evitar dudar de los resultados y pensar en la mejora de uno por sobre la otra por la cantidad de pixeles que esta procesando.
- Hay un incremento en la cantidad de veces que se pide a la memoria información ya que cada vez que procesamos una columna queremos volver a un estado donde mis punteros estén al principio. Esto puede ayudar a que en casos donde la imagen en su totalidad entra en la cache, se mejore la performance.

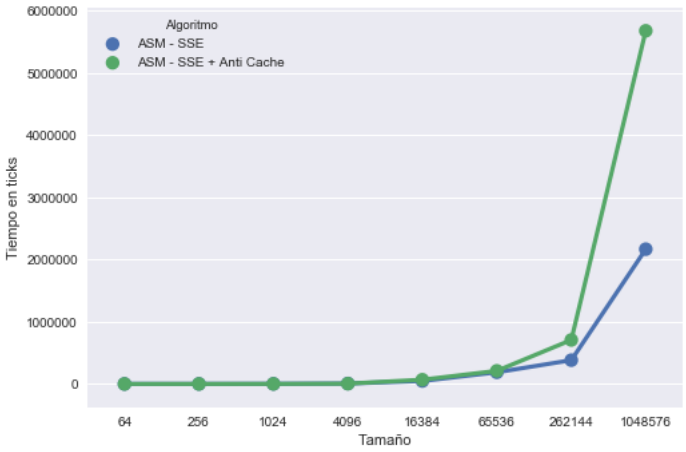


Figura 7: Comparación entre procesamiento de la imagen por filas y por columnas

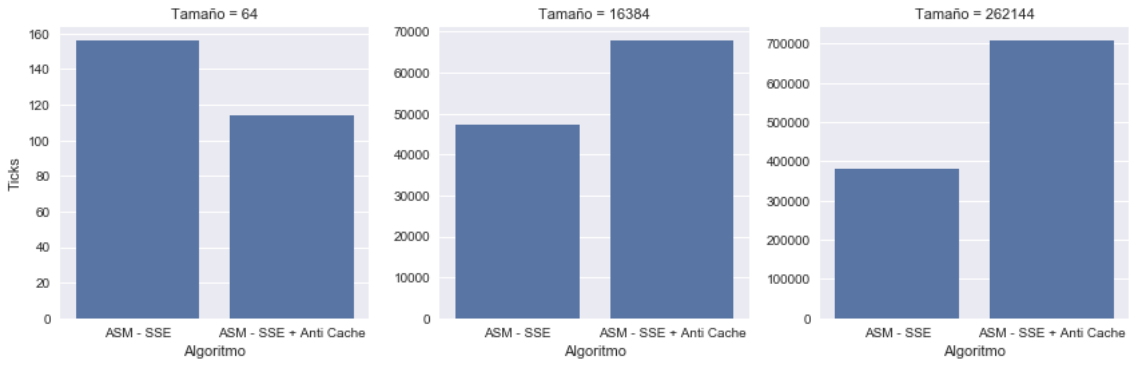


Figura 8: Comparación de impacto de caché en casos puntuales

Se pueda apreciar que hasta cierto tamaño de imagen (en particular, menor a 128x128 pixeles), la caché cumple no un rol significativo y nuestros resultados se parecen bastante. Esto es esperable, ya que es posible que para imagenes pequeñas, múltiples filas correspondan a la misma linea de caché. En cambio, en imágenes más grandes la caché empieza tomar un roll más importante y se empieza a notar como el algoritmo que presentamos es mucho más eficiente que el usamos para nuestra experimentación.

2.4. Trabajos Futuros

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 más 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.

Algo interesante para poder medir mejor la diferencia entre las implementaciones y su función con el cache podría ser medir la cantidad de miss/hits que tienen cada una de las implementaciones así ver en medidas más tangibles como funciona cada uno de ellos.

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

En la implementación en C, un detalle con el que nos cruzamos y que nos generó errores fue que la imagen se encuentra alamcenada con las filas inferiores en primer lugar. Esto es, si representamos la imagen como un arreglo bidimensonal de pixeles `RGBA** img`, esto significa que `img[0][0]` contiene el primer pixel de la fila inferior de la imagen.

Por como dividimos el problema (se detalla más adelante), esto generó confusión al implementar los casos especiales del filtro. No obstante, esto nos sirvió para estructurar mejor la implementación en ASM, que por la naturaleza de ese lenguaje resulta más compleja.

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).

Nuestra implementación en particular procesa los pixeles en el siguiente orden (donde w es el ancho y h la altura en pixeles):

w	w-1	$\cdots$	3	2	1
2*w	2*w-1	$\cdots$	w+3	w+2	w+1
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$
w*h	w*h-1	$\cdots$	w*(h-1)+3	w*(h-1)+2	w*(h-1)+1

Este orden fue escogido en su momento por resultar más sencillo de implementar, pero es posible que el orden de operación resulte en menos comparaciones y saltos condicionales.

El filtro puede dividirse en 4 secciones, dependiendo de la posición en la imagen del pixel a procesar:

1. Caso general: la gran mayoría de los pixeles entran en esta categoría. En estos casos, se deben cargar 4 pixeles de la memoria para procesar y crear 3 pixeles interpolados (como se muestra en la tabla anterior).

   Los 4 pixeles a procesar (que llamaremos A, B, C y D como figuran en la tabla anterior) son cargados en 2 accesos a memoria, ya que pertenecen a 2 filas distintas de la imagen de origen:

$\texttt{XMM}_0$	0	0	B	A
$\texttt{XMM}_1$	0	0	D	C

   Los mismos son desempaquetados para ocupar los registros enteros en esa misma disposición.

   Por un lado, calculamos los dos pixeles que resultan de interpolar 2 pixeles de origen:

$\texttt{XMM}_3$	B	A

$\texttt{MOVLHPS XMM}_3\texttt{ , XMM}_3$

$\texttt{XMM}_3$	A	A

XMM$_4$	0	0	B	A
XMM$_5$	0	0	D	C

PSRLDQ XMM$_4$, 4

PSLLDQ XMM$_5$, 4

XMM$_4$	0	0	0	B
XMM$_5$	0	D	C	0

PADDB XMM$_4$, XMM$_5$

XMM$_4$	0	D	C	B

PUNPCKLBW XMM$_4$, CEROS

XMM$_4$	C	B

PADDW XMM$_3$, XMM$_4$

XMM$_3$	A+C	A+B

PSRLW XMM$_4$, 1

XMM$_3$	(A+C)/2	(A+B)/2

Por el otro, calculamos el pixel que proviene de interpolar 4 pixeles a la vez:

XMM$_0$	B	A
XMM$_1$	D	C

PADDW XMM$_1$, XMM$_0$

XMM$_1$	B+D	A+C

XMM$_0$ $\leftarrow$ XMM$_1$

PSRLDQ XMM$_1$, 8

XMM$_1$	0	B+D

PADDW XMM$_1$, XMM$_0$

XMM$_1$	B+D	A+B+C+D

PSRLW XMM$_1$, 2

XMM$_1$	(B+D)/4	(A+B+C+D)/4

Por último, empaquetamos y ordenamos los pixeles de la siguiente forma:

XMM$_2$	(A+B+C+D)/4	(A+C)/2	(A+B)/2	A

Al igual que en la carga, los resultados empaquetados son guardados a memoria de a 2, ya que pertenecen a distintas filas de la imagen de destino. Los dos pixeles de la parte baja del registro corresponden a la fila superior de la imagen, mientras que los dos pixeles alta del registro corresponden a la fila interior de la imagen.

2. Última columna: en estos casos los pixeles B y D del diagrama no existen, por lo que solo se calcula 1 pixel interpolado entre A y C, y tanto A como el nuevo pixel generado son duplicados a la derecha.

   El proceso es similar al caso general, obteniendo como resultado:

XMM$_2$	0	0	(A+C)/2	(A+C)/2

Sin embargo, se omiten las operaciones que utilizan B y D cuando se puede, y se asumen ceros en sus lugares. Además, el pixel original se copia duplicado por separado.

3. Última fila: similar al caso de última columna, los pixeles C y D no existen. A es interpolado con B, y tanto A como el nuevo pixel generado son duplicados hacia abajo.

   El proceso es similar al caso general, obteniendo como resultado:

$\texttt{XMM}_2$	0	0	(A+B)/2	A

   Sin embargo, se omiten las operaciones que utilizan B y D cuando se puede, y se asumen ceros en sus lugares. Ambos pixeles son duplicados en la última fila de la imagen destino.

4. Último pixel: este caso refiere a un pixel ubicado en última fila y columna. Aqui no hay pixel para interpolar, por lo que se hace un cuadruplicado de A.

$\texttt{XMM}_{15}$	0	0	0	A

$\texttt{XMM}_{14} \leftarrow \texttt{XMM}_{15}$

$\texttt{PSLLDQ XMM}_{14}\texttt{ , 4}$

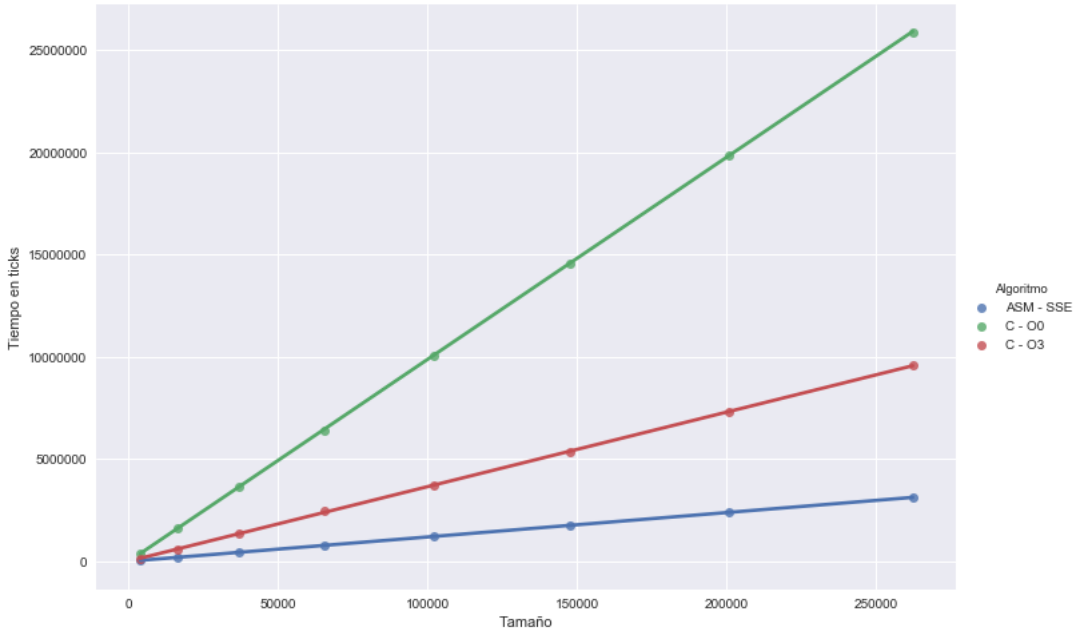
$\texttt{XMM}_{14}$	0	0	A	0

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

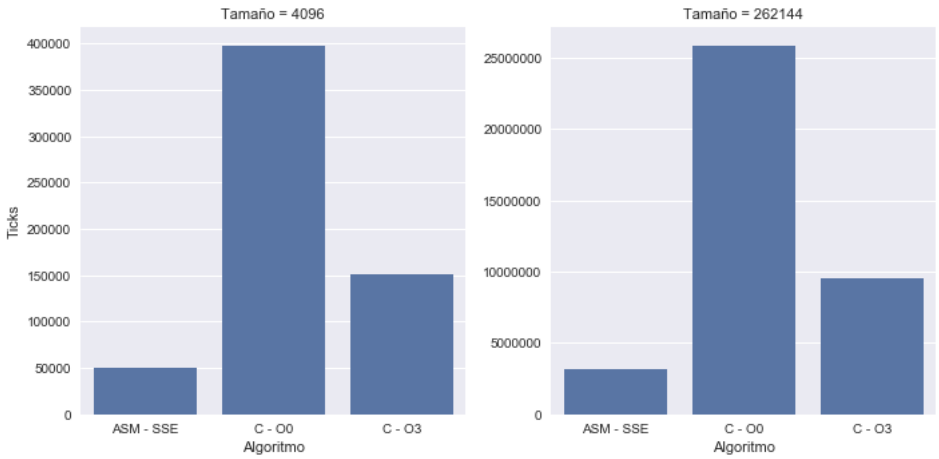
$\texttt{XMM}_{15}$	0	0	A	A

3.2. Análisis preliminar

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.



Por un lado, podemos observar que, como el resto de los filtros, el tiempo de ejecución de los filtros crece de forma lineal respecto a la cantidad de pixeles. Esto es esperable, ya que por cada pixel del caso general se deben leer y escribir 4 pixeles. Eso no es así para los otros pixeles, pero en nuestros tests de control estos representan la menor parte. Es posible que en imágenes pequeñas con distintas proporciones (por ejemplo, 200x50 en lugar de 100x100) esta proporción no se respete, pero el impacto de los casos borde debería ser menor a medida que crece la imagen.

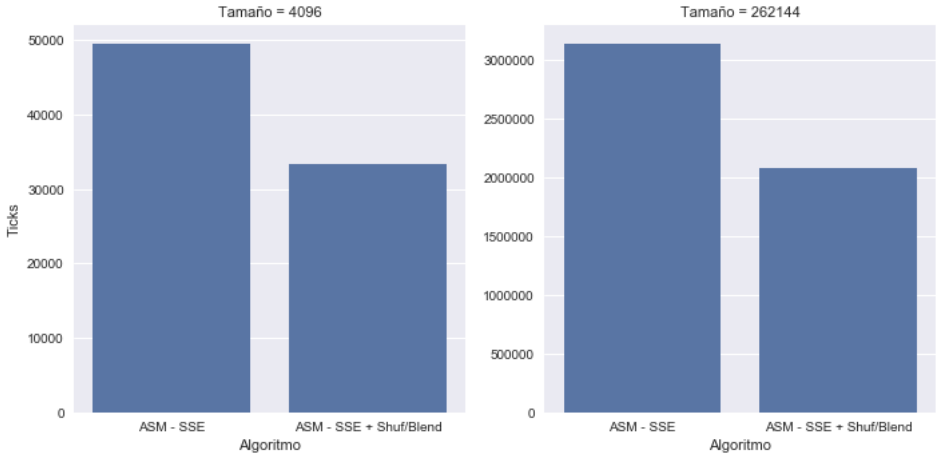


Por otro lado, podemos ver que efectivamente, incluso con las optimizaciones de GCC, la vectorización de la interpolación aumenta el rendimiento de manera dramática, ejecutando el filtro siempre en menos del 40 % del tiempo.

3.3. Experimentación

Como se mencionó en la sección de implementación, nosotros utilizamos `movs`, shifts y `add`s para acomodar los pixeles y realizar los promedios entre los pixeles. Sin embargo, existen instrucciones específicamente pensadas para estas tareas: los shuffles y los blends.

Con esto en mente, decidimos modificar nuestro filtro para que utilice dichas instrucciones:



Nuestras pruebas confirmaron esta hipótesis: en promedio pudimos reducir el tiempo de ejecución en 30 %. Nuestra idea es que usar instrucciones como `pshufd` y `pblendw` es más eficientes porque combinan la copia y el acomodado de los pixeles a los bits específicos de los registros que necesitamos. Al mismo tiempo evitamos utilizar la parte lógica del CPU, ya que no utilizamos add sino blend para mezclar los registros, y suponemos que requiere menos tiempo de espera que una suma de números empaquetados.

4. Maximo cercano

Este es un filtro tal que cada pixel de la imagen original se edita de la siguiente manera:

1. Obtenemos los pixeles de alrededor, que se encuentren a una distancia de 3 pixeles o menos. A esto lo denominamos kernel, en este caso, de 7x7.
2. Se calcula el máximo valor para cada componente entre estos pixeles obtenidos.
3. Se genera un nuevo pixel con estos 3 valores encontrados.
4. El pixel final se calcula mediante una combinación lineal entre el pixel original y el pixel que contiene las componentes máximas.
5. Aquellos pixeles para los cuales no se puede armar un kernel de 7x7 alrededor (las primeras y últimas 3 columas y 3 filas de la imagen) deben ser pintados de blanco.

La combinación lineal entre los pixeles se hace de la siguiente manera:

$$src * (1 - val) + max * (val)$$

Donde *src* es el pixel original, *max* es el pixel generado, y *val* es un número flotante entre 0 y 1, este mismo es un parámetro del filtro.

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: Kernel, en rojo el pixel que estamos editando y en amarillo los pixeles que forman parte del kernel

4.1. Implementación

La implementación en C se trató de hacer lo más simple posible. Iteramos las filas y las columnas, y en cada paso preguntamos si el pixel se encuentra en el margen de 3 pixeles. Si es así, lo pintamos de blanco; si no, nos generamos un pixel con cada componente inicializada en el mínimo valor(0) y recorremos el kernel y usamos este pixel que acabamos de generar para usarlo de máximo actual, modificando cada componente cuando encontramos un valor mayor al actual. Una vez iterado el kernel realizamos la combinación lineal sobre cada componente y guardamos el pixel en la imagen destino.

En cambio, en la implementación del filtro en lenguaje ensamblador, esta se ve complejizada ya que 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 pixeles 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.

En este caso primero recorremos las filas y diferenciamos 2 casos: el caso en que toda la fila se encuentra en el margen que se pinta de blanco y el que no.

Si es una fila blanca, vamos a intentar guardar la mayor cantidad de pixeles contiguos posible. Como ya vimos que en los registros XMM nos caben 4, nos generamos un registro tal que los 4 pixeles sean blancos y guardarlos en memoria a la vez, al ser imágenes con ancho múltiplo de 4. Podemos aplicar esto sobre toda la fila, iterando de a 4 columnas pero vez. Con esto logramos en las primeras filas y las últimas una reducción al iterar las columnas.

Si es una fila que no va a pertenecer en su completitud al margen, vamos a tener que calcular el máximo de cada color en el kernel y después hacer la combinación lineal. En la iteración del kernel, en diferencia a C, podemos guardarnos en un solo registro 4 pixeles tal que puedan ser los máximos. Para esto usamos la instrucción PMAXUB, que compara byte a byte entre los registros y guarda el máximo en el registro destino. Gracias a esta instrucción podemos ir comparando estos 4 posibles máximos contra 4 pixeles contiguos del kernel e ir manteniendo los máximos parciales. Cabe destacar que la comparación es byte a byte, así que va a comparar las componentes sin desempaquetarlas. Como

en el kernel tenemos nada más 7 pixeles contiguos, vamos a aplicar este método 2 veces por fila, repitiendo un pixel de la fila en cada aplicación sin que esto afecte el resultado final.

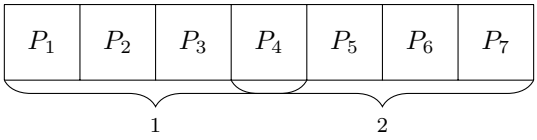
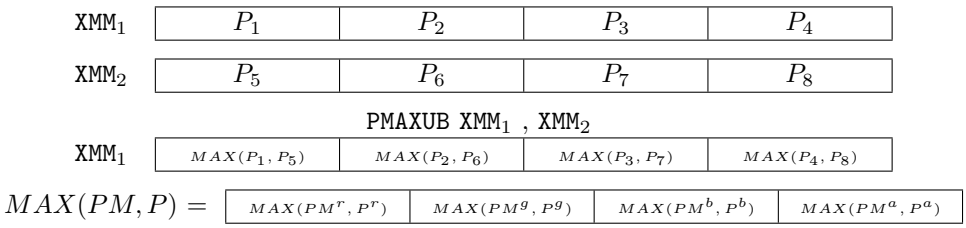
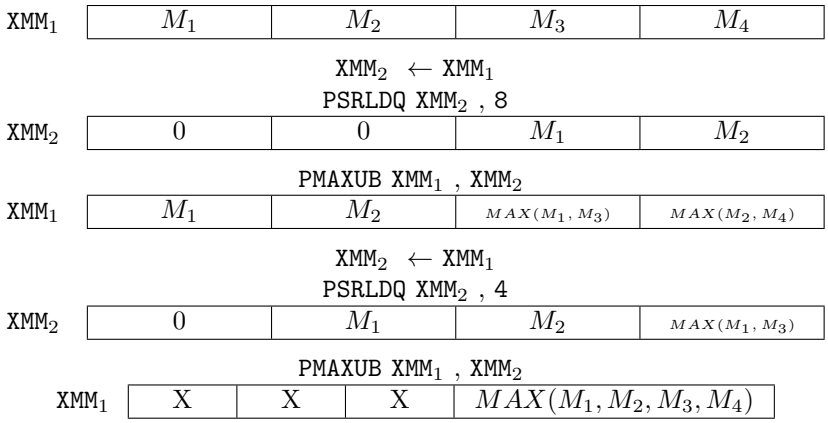


Figura 9: pixeles contiguos de una fila del kernel, las llaves indican que pixeles se toman en las 2 aplicaciones por fila

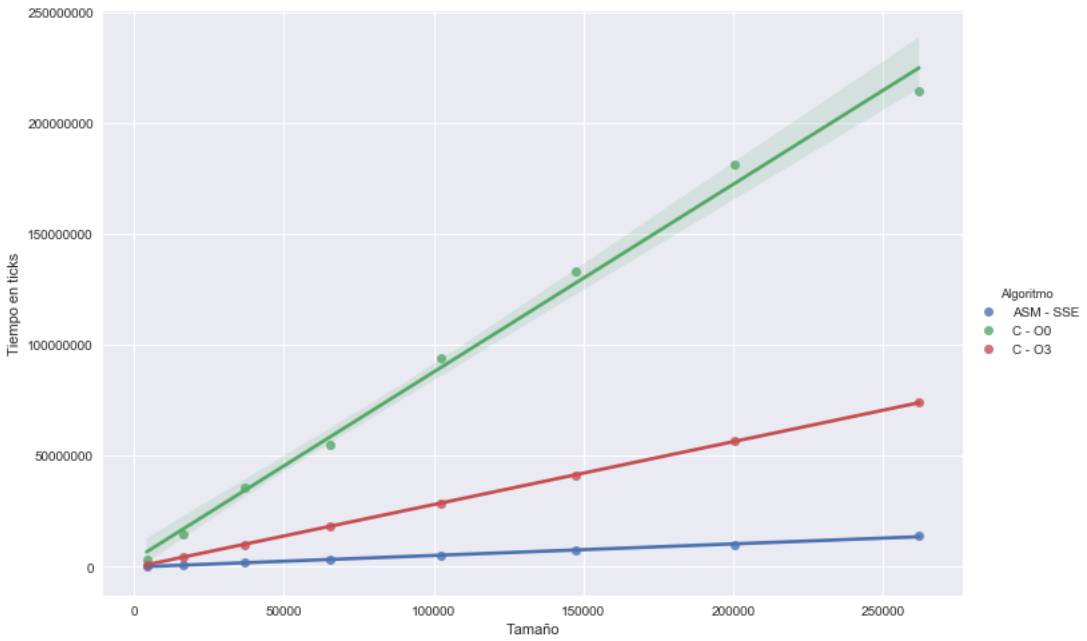


Una vez realizado esto sobre cada fila, vamos a tener en el registro los cuatro posibles pixeles máximos. Los mismos son comparados entre sí para conseguir el pixel definitivo que contenga el máximo de cada componente. Para esto podemos seguir usando PMAXUB, duplicamos el registro y shifteamos 8 bytes a la derecha uno de ellos, para que nos quede desplazado 2 pixeles y podamos comparar el primer par de pixeles del registro contra el segundo par. Repetimos este proceso desplazando 4 bytes para comparar los últimos 2 pixeles. Una vez realizado esto nos va a quedar en la parte menos significativa del registro el pixel que estábamos buscando.



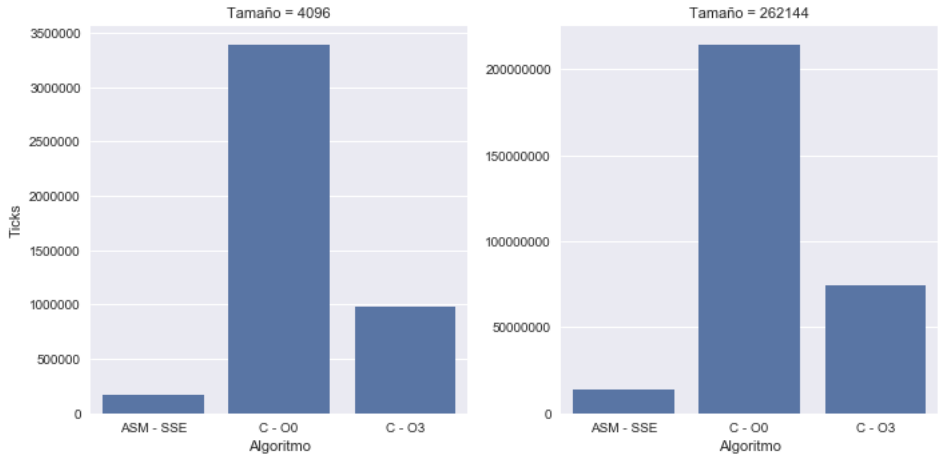
Ya conseguido el máximo, ahora tenemos que realizar la combinación lineal entre el pixel original, el nuevo pixel que generamos y el parámetro. Lo que queremos es hacer la cuenta para cada componente en paralelo: para esto podemos usar instrucciones SIMD para multiplicar las componentes de los pixeles en paralelo por el parámetro. Como el mismo es un valor decimal (`float`), convertimos cada componente `float` quedándonos todas las componentes en un registro XMM. Para multiplicarlas por el parámetro, armamos 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 realizamos un proceso análogo con el pixel original y $1 - val$, y sumamos estos dos resultados. Convertimos este resultado de vuelta a enteros y lo guardamos en la imagen destino, continuamos con el resto de los pixeles.

4.2. Análisis preliminar



Una vez más, este filtro se comporta de manera lineal con respecto al tamaño de la imagen en pixeles, independientemente de la implementación usada. Para cada pixel deben procesarse 49 pixeles (por el kernel de 7x7), pero esta cantidad es fija, por lo cual es lógico asumir que el filtro sea lineal. Por otro lado, para el caso del margen blanco (que implica unicamente procesar el pixel original), la cantidad de pixeles abarcada es muy poca para que estos tengan un impacto visible en la performance.

También 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. Si bien hay una amplia mejora al compilar con optimizaciones `-O3`, no llega a mejorar la versión de ASM.



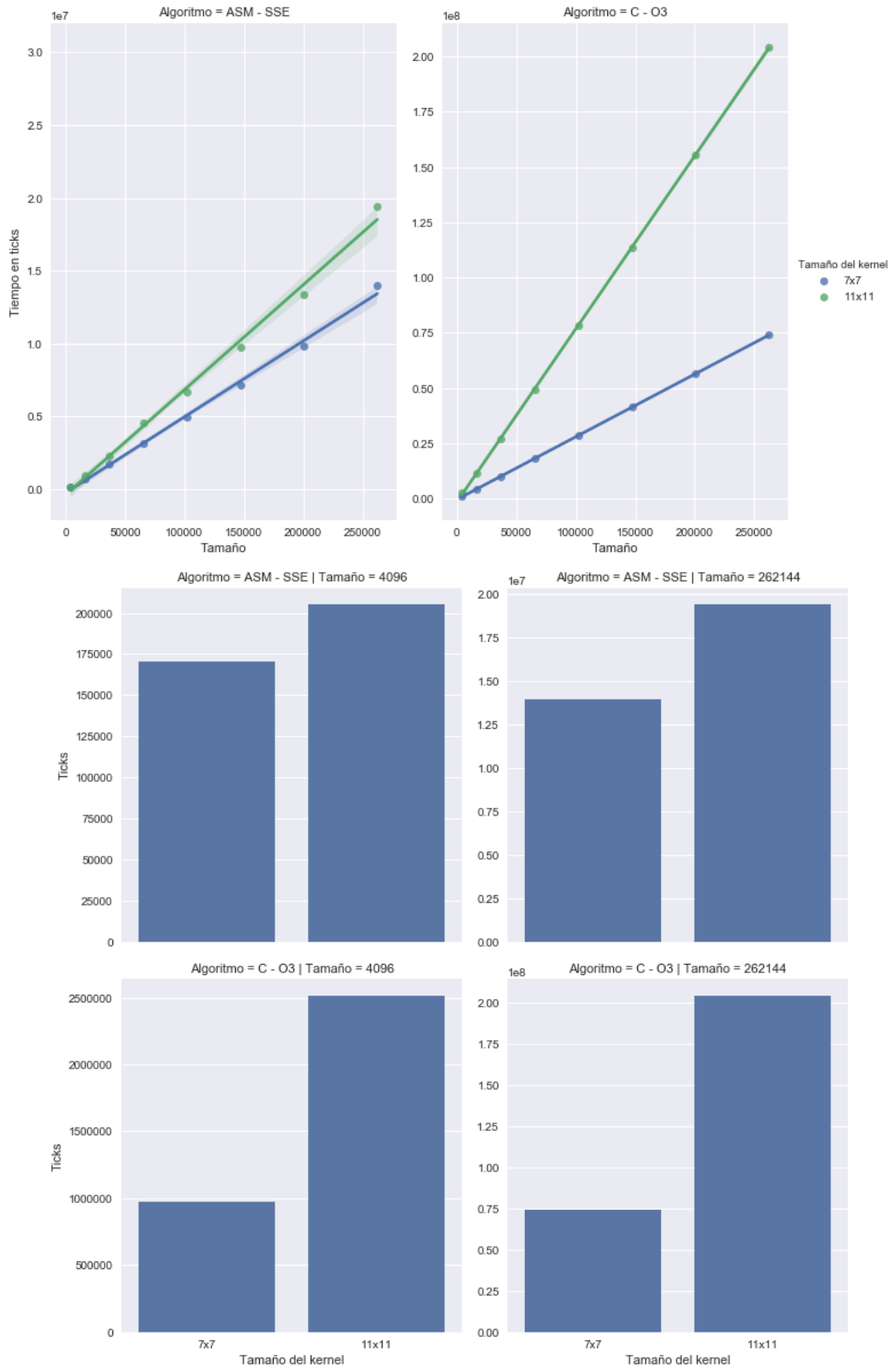
Analizando de cerca, podemos ver que la diferencia entre las distintas implementaciones es la mayor entre todos los filtros estudiados: la implementación en ASM corren en menos de 25 % del tiempo en promedio para una misma imagen.

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.

4.3. Experimantación

Como reveló el análisis preliminar, la cantidad de pixeles origen a procesar por pixel destino tuvo un gran impacto en la performance de nuesto filtro. Decidimos ver cómo impacta en ambas implementaciones si agrandamos el tamaño de kernel. Para esto corrimos los mismos tests que antes, pero ahora con un kernel de 11x11 en vez que de 7x7.

Nuestra hipótesis fue que esta medida afectaría 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 estamos incrementando el tamaño del kernel en casi 2.5 veces, esperamos una performance peor en este orden. Pero en ASM aprovechando la paralelización de datos, levantamos 11 pixeles con 3 llamadas a memoria nada más.



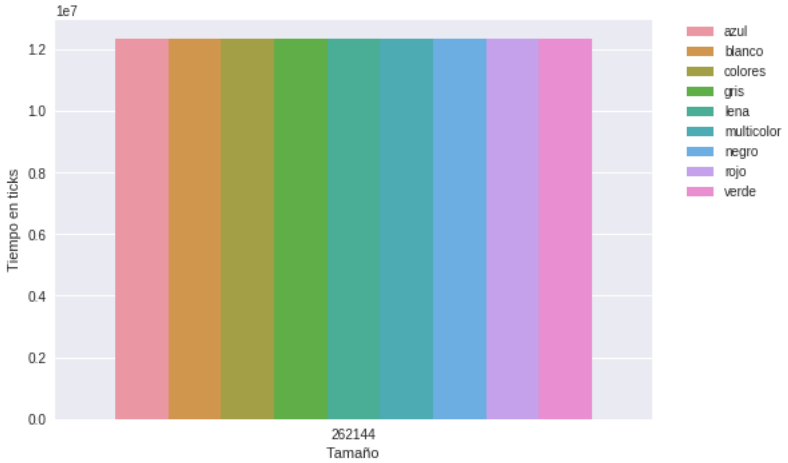
Si observamos los gráficos, podemos ver como en C con el kernel de 11x11 tarda casi 2.5 veces más, como la relación de diferencia de pixeles en los kernels, y para ASM hay nada mas una diferencia de 1.2 y 1.4 para instancias

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.

Análisis de las variables de entrada

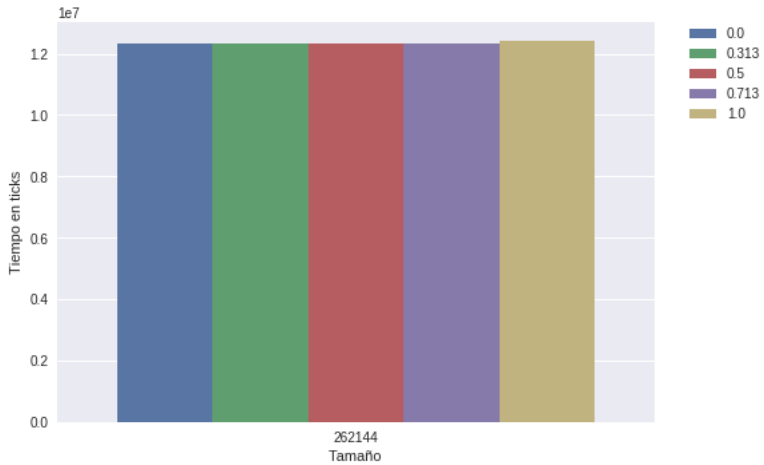
Queremos realizar un análisis sobre el tiempo en que tarda en terminar el filtro en base a los parámetros que recibe, distintos tipos de imagen, tamaños y el valor que afecta la combinación lineal. Nuestra hipótesis es que las pruebas que hacemos no van a tener un impacto significativo sobre el rendimiento del filtro, ya que no hay optimizaciones que hagan variar el tiempo final según los parámetros (más allá de la cantidad de pixeles).

Primero hicimos un análisis sobre distintas imágenes y cómo reaccionaba el algoritmo frente a ellas. Para este caso decidimos comparar las imagenes de control junto con imágenes de colores sólidos (azul, rojo, verde, gris, blanco y negro) y con una imagen generada aleatoriamente (en python), donde cada pixel es de un color elegido aleatoriamente con distribución uniformemente. Para esta experimentación se dejó fijo en todas las imágenes el tamaño (512x512) y el valor del parámetro de la combinación lineal (0.5).



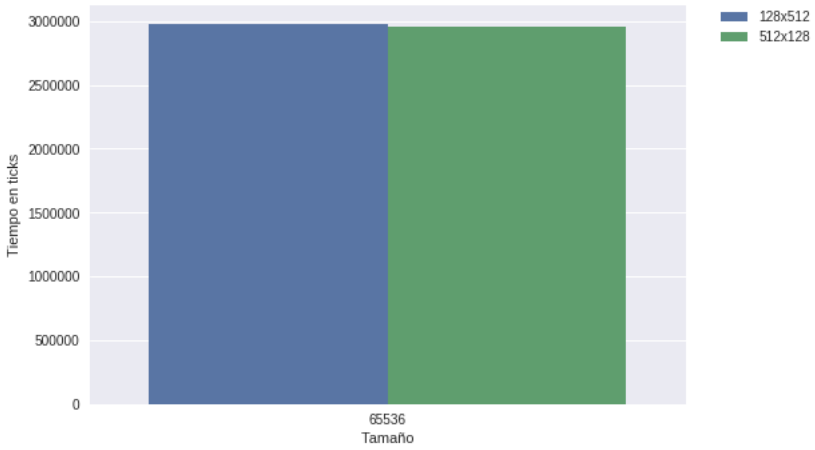
A simple vista se ve que varía muy poco, y posiblemente esto sea producto de la inexactitud de la experimentación. Este resultado se debe a que los cálculos se hacen de cualquier manera, siempre buscamos en el kernel el mayor de las componentes. Una cosa que se nos ocurrió es que la imagen aleatoria podría haber llegado a tardar un poco más debido a que cambia más el valor del máximo en la búsqueda del kernel, no como en los colores fijos que no cambia, pero suponemos que esto no tiene impacto y que el procesador no pierde más tiempo cuando por ejemplo en la instrucción PMAXUB tiene que cambiar los valores en los registros.

El siguiente experimento fue sobre el parámetro de entrada de punto flotante que se aplica sobre la combinación lineal. Notemos que acá si podríamos haber optimizado cuando el valor viene en 0 ya que nos podemos ahorrar la búsqueda del máximo, pero es una optimización simple que no es a lo que apuntamos en este trabajo. Corrimos este experimento como en el anterior pero usando solo las imágenes de control y utilizando los valores 0, 0.313, 0.5, 0.713 y 1.



De nuevo podemos observar la poca varianza entre los distintos valores. El que menos ticks dió fue el de 0.5 con 12335487 ticks y el que mayor fue el de 1 con 12416301 ticks, la diferencia entre estos dos representa el $0,65\,\%$ de este último. Con estos resultados podemos afirmar que no hay diferencia significativa.

Por último probamos cómo afectan las diferentes proporciones de tamaño, por ejemplo, si la imagen es más ancha que alta. En este caso probamos para las imágenes de control con el valor fijo en 0.5 y usando los tamaños 512x128 y 128x512 ya que tienen la misma cantidad de pixeles.

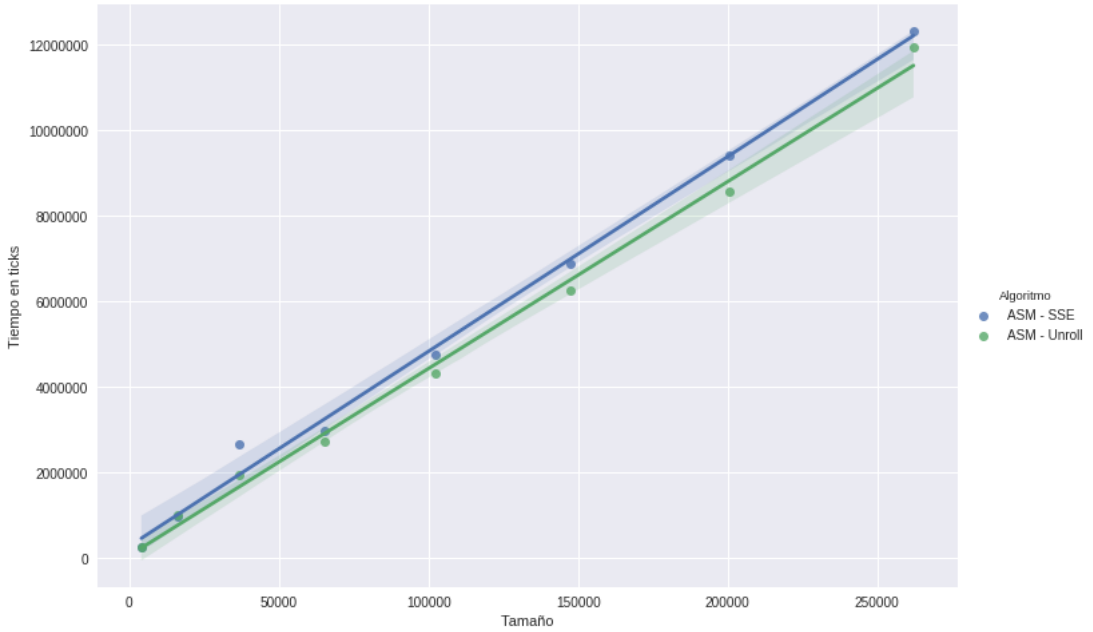


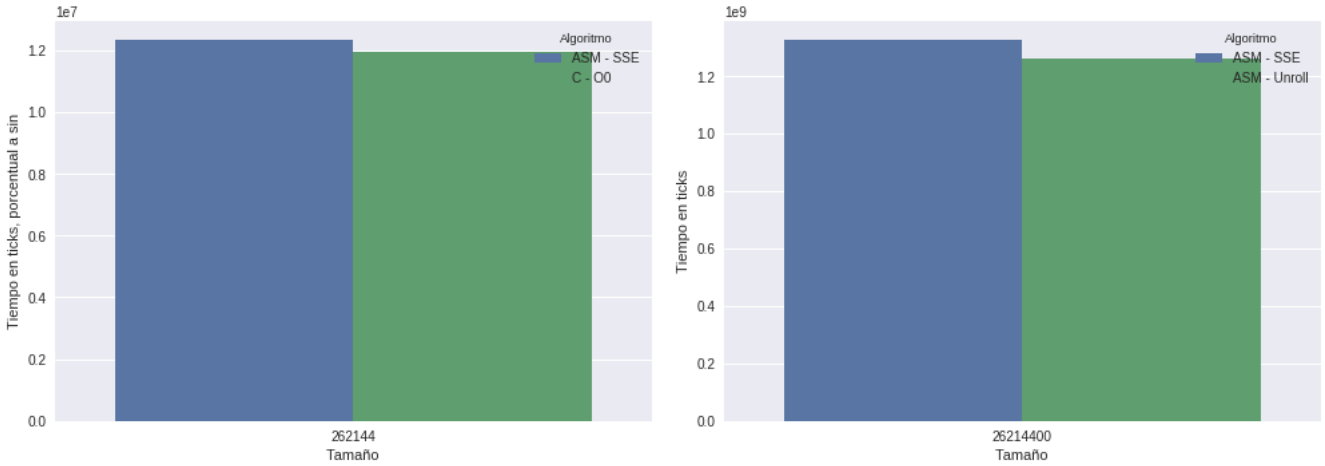
Una vez más, la diferencia es bastante menor, con la imagen ancha corriendo en $99{,}3\,\%$ del tiempo de la recíproca. Es una diferencia casi despreciable, y es posible que resulte de ruido de medición. Se puede mencionar que esto podría provenir del costo menor de dibujar las 3 primeras y últimas filas, ya que no se calcula nada en las mismas(se utiliza la constante blanca). Al ser más ancha, vamos a poner más pixeles dentro de estas. Si bien la cantidad de pixeles blancos es la misma, en este caso se concentran en el ciclo de una fila en lugar de estar dispersos en varias filas en las primeras y últimas columnas.

Con estos resultados, podemos concluir que el único factor determinante en el rendimiento es la cantidad de pixeles.

Unrolling

En este experimento nuevamente aplicamos la técnica de unrolling de ciclos. En este caso si contamos con casos finitos al crear el kernel, ya que el mismo tiene un tamaño fijo en filas y columnas. Actualmente ya procesamos una fila entera en un ciclo, por lo que desenrollamos el procesamiento las 7 filas. A diferencia de los casos anteriores, el kernel ocupa una parte menor del procesamiento de la imagen, y suponemos que el mayor impacto en performance se encuentra en los accesos a memoria, por lo que tenemos la hipótesis de que la mejora será muy leve o nula. Corrimos los mismos tests que en el analisis preliminar para la version original y la version con unrolling. Adicionalmente, decidimos medir el caso puntual de una imagen de tamaño 5120x5120, para estudiar el impacto que puede tener en imágenes más grandes.





Se puede percibir una mejora, pero en comparación a lo que tarda el algoritmo es muy leve: 4 % para la imagen de 512x512 y 5 % para la imagen de 5120x5120. La diferencia no es mucha, pero si estamos en una imagen muy grande o tenemos que optimizar mucho, podría llegar a valer la pena.

5. Conclusiones

A grandes rasgos, podemos concluir que el uso de operaciones SIMD es altamente beneficioso en muchos contextos dentro del procesamiento de imágenes. Si bien existen casos donde su uso no aporta grandes benficios (como pueden ser los casos de los conversores RGB-YUV o FourCombine), en la mayoría de los filtros la diferencia de performance era notoria.

Cabe destacar que la implementación de los algoritmos en ASM con operaciones SIMD fue mucho más costosa que la versión correspondiente en lenguajes de alto nivel, y de no ser realizada con ciudado puede llevar no a mejora sino a pérdida de performance. En particular, en una implementación inicial de uno de los filtros hallamos información que estaba siendo calculada más de una vez, anulando los beneficios de la paralelización. Incluso dentro de nuestra experimentación con LinearZoom podemos ver que obtener la implementación más eficiente posible no es trivial.

Los casos donde consideramos más útil la implementación con SIMD es en aquellos que utilizan un kernel de píxeles (como en MaxCloser). En dichos filtros cada pixel debe procesarse de manera distinta múltiples veces (para encontrar su reemplazo y como parte de un kernel), y resulta difícil (o imposible) reutilizar valores intermedios.

6. Apéndices

6.1. Apéndice I: recolección de datos

Para recolectar datos, tratamos de armar un criterio consistente y que genere el menor ruido posible. Para esto hicimos lo siguiente:

- Las mediciones de tiempo se realizan antes y después de la llamada al algoritmo del filtro (obviando el tiempo de carga y liberado de memoria de las imágenes).
- Las mediciones de tiempo se realizan utilizando el comando `RDTSC` a través del macro de C provisto por la cátedra.
- Para las mediciones de control se utilizaron los siguientes tamaños de imagen: 64x64, 128x128, 192x192, 256x256, 320x320, 384x384, 448x448 y 512x512. Se utilizó una sola proporción para las imágenes (1:1) para evitar introducir posibles discrepancias.
- Para las mediciones de control se utilizaron las imágenes provistas por la cátedra: `lena.bmp` y `colores.bmp`. Las mismas cuentan un una distribución relativamente heterogenea de colores y por ende podrían considerarse un "caso promedio".
- Cada medición se repitió 100 veces, conservando únicamente el valor menor, ya que consideramos al mismo el más representativo y con menos ruido de otros procesos en la computadora.
- Todas las mediciones se realizaron en equipos del laboratorio Turing, para evitar diferencias de hardware.

Los datos obtenidos fueron almacenados en formato CSV, y luego analizados y graficados utilizando pandas y matplotlib.