Trabajo Práctico 2

Lunes 21 de Julio de 2014 — Organización del computador II

SIMD

Grupo: Grecia/Gyros

Integrante	LU	Correo electrónico
Guido Rajngewerc	379/12	guido.raj@hotmail.com
Martín Caravario	470/12	martin.caravario@gmail.com
Federico Hosen	825/12	fhosen@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellon I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autonoma de Buenos Aires - Rep. Argentina
Tel/Fax: (54 11) 4576-3359
http://www.fcen.uba.ar

Índice

1. Tiles

Explicación del algoritmo :
El filtro tiles básicamente consta en leer 16 bytes de un pedazo de la imágen fuente, y escribirla en la imágen destino, sin tener que hacer ningun tipo de cálculo con valores "vecinos", o del mismo dato leído. Teniendo ésto en claro, simplificamos el tratamiento del dato, pensando a la matriz como una sucesión de filas, obviamente teniendo cuidado cómo avanzamos nuestros contadores de fila y columna actual.
Como tenemos siempre que leer y escribir de a 16 bytes, tenemos que tener cuidado *de dónde* vamos a leer, y *en dónde* vamos a escribir. Pues si estoy en la última de las filas estaría escribiendo (y quizás también leyendo) fuera de la imágen. Al analizarlo de esta forma, encontramos cuatro posibles casos a la hora de tener que leer y escribir datos. El más fácil de resolver de los tres es cuando tengo suficientes datos para leer y para escribir, en cuya situación simplemente hay que copiar los 16 bytes, sin mayores cuidados. El segundo caso es cuando estoy por leer cerca del final de la fila en la imagen fuente, pero tengo suficiente espacio para escribir en la fila destino, con lo cual, retrocedo en la fila fuente, leo 16 bytes, y los escribo (retrocediendo la misma cantidad de bytes en la fila destino) en el destino.
El tercer caso se produce cuando estoy en el fin de la fila destino, pero puedo leer con tranquilidad en la fila fuente. Esto quiere decir que tengo que retroceder en destino (hasta tener 16 bytes por escribir) y leer de fuente 16 bytes, pero solo escribir unos pocos, preservando el valor de los que se encontraban detrás de donde estaba parado en destino. Ésto es, estoy llegando al borde de la imágen destino, y tengo que escribir ün poquito"de la imágen fuente.
El último caso es el más complicado, y se preduce cuando estoy cerca del final de ámbas filas. En esta situación tengo que evaluar quién está más cerca del borde de su fila. El que esté mas cerca, determinará cuánto tengo que escribir, y cuánto tengo que leer.

1.1. Experimento 1

Resultados :
Realizamos el experimento usando el comando objdump con los parámetros *-dM intel* para que el mismo desensamble la parte ejecutable y la interprete como código de intel.
En primera instancia se ve que el compilador prepara la pila para el nuevo trabajo pusheando *rbp* y hacendo un *mov rbp, rsp.*
Lo siguiente que notamos fue que el código desensamblado al principio ejecuta las siguientes líneas:

```
 0: 55                    push   rbp
 1: 48 89 e5              mov    rbp,rsp
 4: 48 89 7d b8           mov    QWORD PTR [rbp-0x48],rdi
 8: 48 89 75 b0           mov    QWORD PTR [rbp-0x50],rsi
 c: 89 55 ac              mov    DWORD PTR [rbp-0x54],edx
 f: 89 4d a8              mov    DWORD PTR [rbp-0x58],ecx
12: 44 89 45 a4           mov    DWORD PTR [rbp-0x5c],r8d
16: 44 89 4d a0           mov    DWORD PTR [rbp-0x60],r9d
```

Conclusiones :
Pudimos concluir que todas las variables que llegan a la función son pusheadas a la pila. A medida que se va ejecutando el programa se ve como los accesos a memoria se repiten dado que debe alcanzar las variables almacenadas en el stack. Consideramos que esto es optimizable pues el uso de registros para almacenar las variables reduciría el alto costo que implica el acceso a memoria.

1.2. Experimento 2

Nuevamente corrímos el comando *objdump* con los parámetros anteriores y nos resultó claro a simple vista que el compilador ahora no recurre a la pila para almacenar variables, podemos ver como en primera instancia pushea varios registros de la siguiente forma:

```
 0: 41 57                 push   r15
 2: 41 56                 push   r14
 4: 41 55                 push   r13
 6: 41 54                 push   r12
 8: 55                    push   rbp
 9: 53                    push   rbx
```

Luego hace unos *movs* para traer variables de la pila a registros de propósito general, esto se puede apreciar en estas líneas:

```
14: 44 8b 54 24 38        mov    r10d,DWORD PTR [rsp+0x38]
19: 8b 6c 24 48           mov    ebp,DWORD PTR [rsp+0x48]
```

Conclusiones :
Al principio veíamos como el código optimizado genera un stack frame, pusheando los registros que utilizará luego. Concluímos entonces que estos registros son utilizados para mover y procesar las distintas variables. Así al ejecutar código veremos beneficios, pues se soluciona el problema planteado en el experimento anterior, ya que se limitan los accesos a memoria los cuales son operaciones muy caras en cuanto a tiempo se refiere. También puede verse que el tamaño total del código generado en el segundo caso es bastante menor que el primero. Eso implica menos operaciones realizadas.
NOTA: No adjuntamos los *objdump* completos en el anexo por que no creemos que sean un aporte significativo al análisis presentado.

Optimización :
Investigando sobre los flags de optimización encontramos que algunas de las acciones que realiza el flag -01 son:

- -fmerge-constants: Unifica las constantes de mismo valor (ints con ints, floats con floats)
- -fauto-inc-dec: Combina los aumentos y decrementos de las posiciones de acceso a memoria
- -fdce: Elimina codigo no utilizado (en el manual llamado *dead code*)
- -fif-conversion: Optimiza los saltos condicionales
- -fcombine-stack-adjustments: Intenta combinar *push* y *pop* de variables para limitar los accesos al stack

NOTA: La lista completa de los flags que se activan al activar -O1 se puede ver en el anexo de este TP.

GCC incluye una variada lista de opciones de optimización a traves de la compilación con las opciones **-O**.

- **-O1** Es la primer opción, busca hacer un código más pequeño y compacto sin condicionar de sobremanera el tiempo de compilación.
- **-O2** Ademas de las optimizaciones de -O1 realiza varias más que mejoran el rendimiento, pero tienen un costo apreciable en el tiempo de compilación.
- **-O3** Los tiempos de compilación y el uso de memoria se ven fuertemente afectados. El compilador intenta hacer toda la optimización de código que le es posible. Debe notarse que hemos encontrado casos de gente en foros comentando que su código no funciona con este flag.
- **-Og** Activa todos los flags de optimización que no afectan de ninguna manera la capacidad del debugger de encontrar los errores del código.
- **-O0** No hay optimización. Es la opción por defecto.
- **-Ofast** Deja de lado los estandares e intenta realizar todas las optimizaciones que le es posible al compilador.
- **-Os** Optimiza el tamaño usando los flags de -O2 que no incrementan el tamaño del código. Agrega ademas algunos otros flags.

1.3. Experimento 3

Dado la naturaleza de este experimento debemos primero aclarar las especificaciones de la computadora donde los realizamos:

```
CPU: Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz
Number of cores: 4
Arquitectura: 64 Bit
Memoria RAM: 8 GB

Maquina número 11 del laboratorio 4, Departamento de Computación, UBA.
```

Para disminuir el impacto de los outliers durante el proceso de experimentación ejecutamos el comando *htop* y analizamos que ningun proceso que se estuviera ejecutando pudiera tener un impacto significativo en nuestro experimento.
También consideramos que deberíamos obtener datos representativos por lo que cada filtro se ejecutó un total de 1000 veces. No tomamos nunca en cuenta los tiempos del opencv para cargar y guardar las imagenes, pues influiría en nuestros datos y no es lo que buscamos.
Luego de las 1000 pasadas obtuvimos la cantidad promedio de los ciclos que demora cada implementación.
De ahora en más puede asumirse que el filtro tiles fue llamado con los parámetros:

```
tamx= 55
tamy= 55
offsetx= 150
offsety= 150
```

Resultados :

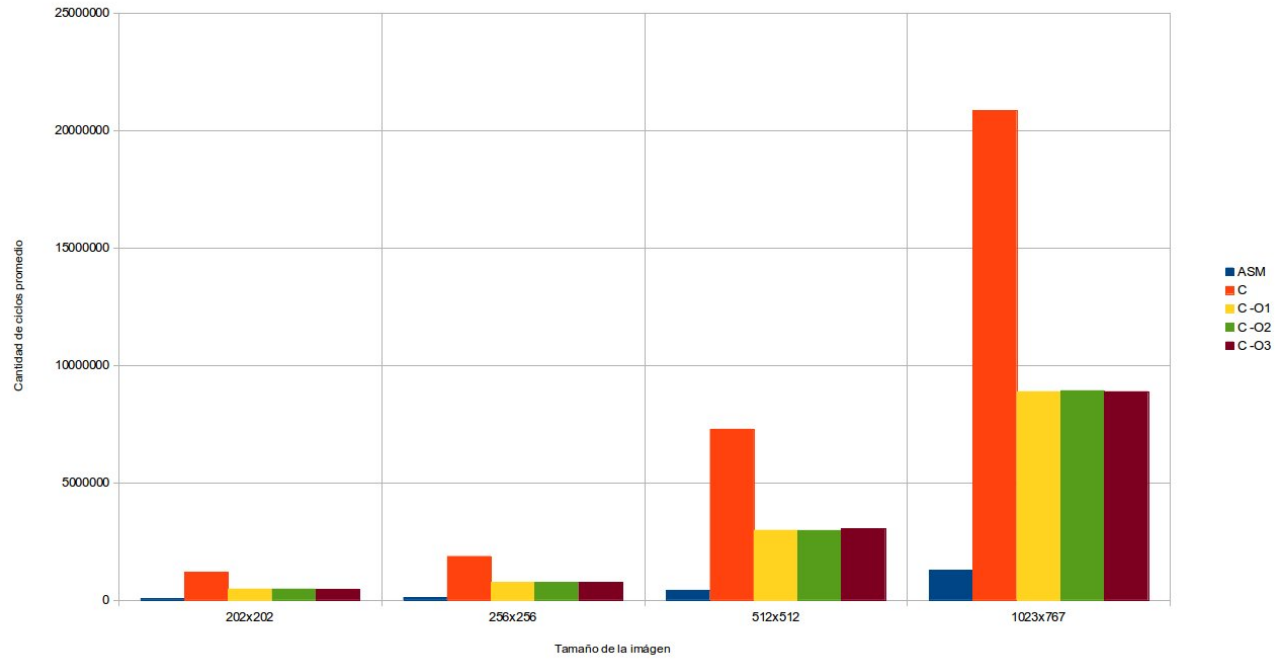


Gráfico variando tamaños e implementación sobre Lena

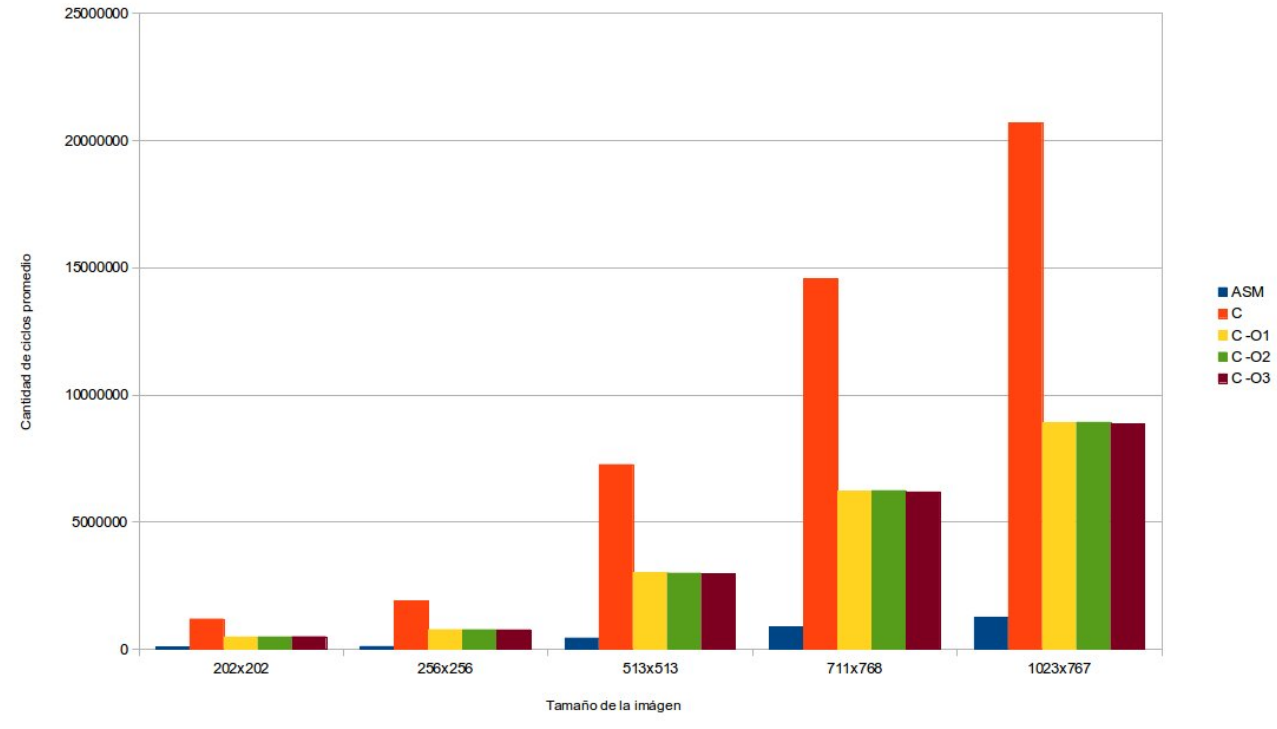


Gráfico variando tamaños e implementacion sobre Marilyn

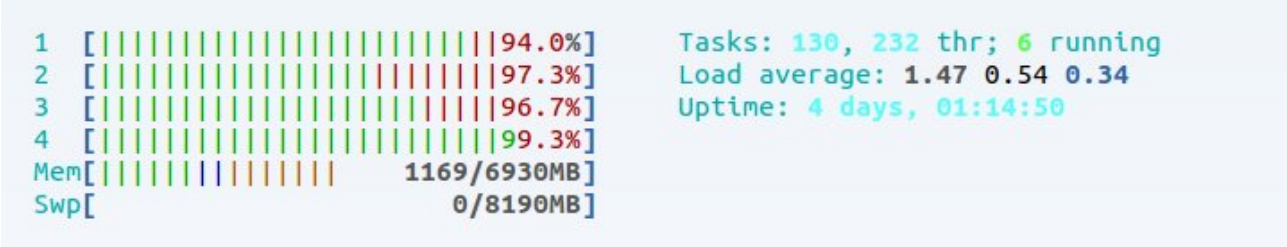
Conclusiones :
En ambos casos nos parece importante remarcar la gran diferencia que hay entre la cantidad de ciclos que se requieren para el procesamiento utilizando la implementación de ASM con respecto a las demas. El modelo de SIMD muestra la enorme ventaja que presenta procesar datos en paralelo.
Además se puede apreciar la incidencia de la optimización del compilador sobre el código en C. Aunque igualmente nos sorprendió como los disntitos niveles de optimización tienen un impacto similar en el rendimiento del algoritmo.
Finalmente vemos como a medida que el tamaño de las imagenes se agranda las diferencias son cada vez más notorias. Esto tiene sentido ya que la cantidad de ciclos que se necesitarán para procesar una imágen es directamente proporcional al tamaño de la misma.
Nos parece que en este filtro evidencia de mayor manera que en los demás la ventaja del modelo de programacion *SIMD*

sobre el modelo tradicional, ya que el filtro consiste simplemente en copiar un pedazo de imágen sobre otra, con escasos accesos a memoria (mas allá del obligatorio para leer el dato) por tener variables locales en la pila.

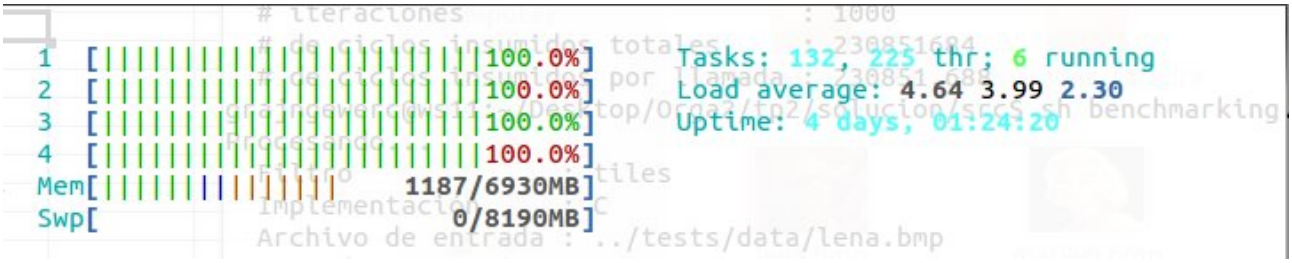
Nota:
En el anexo se pueden ver las tablas utilizadas para la creación de los gráficos anteriores.

En el experimento se nos solicitó comprobar como afecta el uso del procesador a la performance de las implementaciones. Para hacer esto ejecutamos programas que hicieran un fuerte uso de los procesadores del sistema y nos aseguramos que al procesar las imágenes no se matase ningun proceso, asegurando así que el experimento funcione tal como se espera.

Estado del procesador :
Los resultados obtenidos fueron los siguientes, en donde 1, 2, 3 y 4 representan a los núcleos del procesador



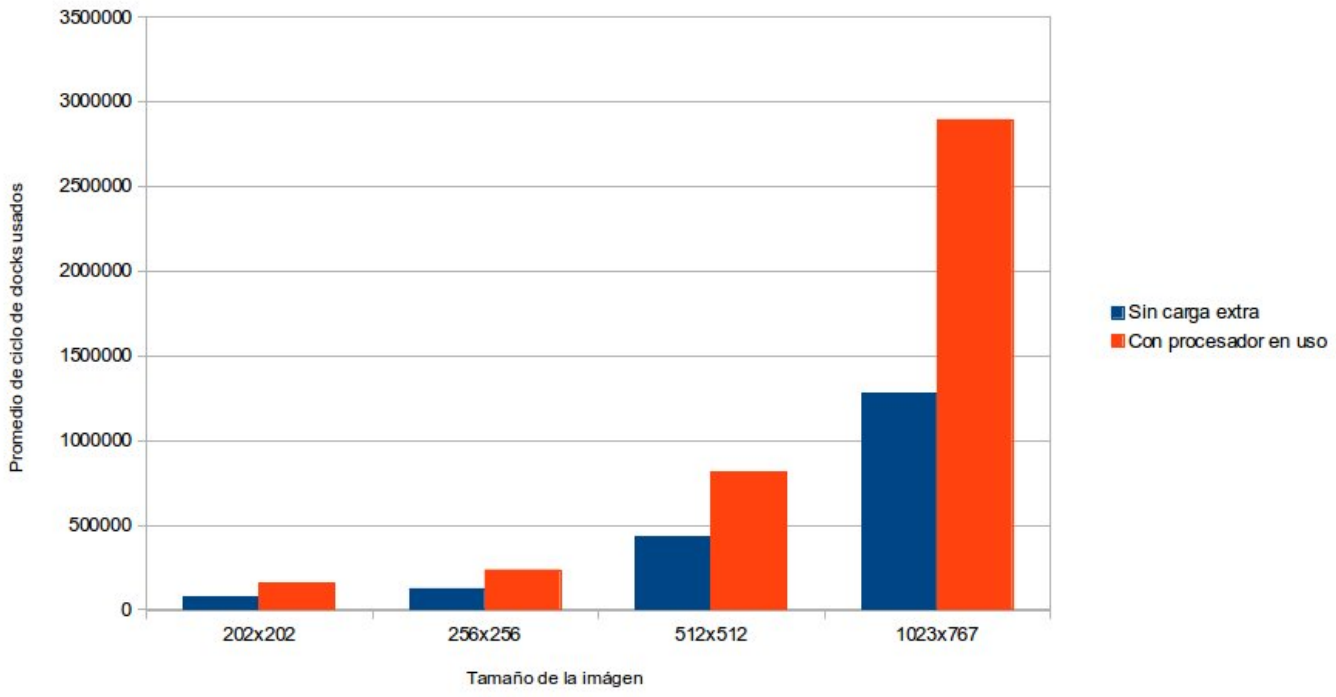
Aquí se observa el estado del procesador al exigirlo con procesos sin correr el experimento



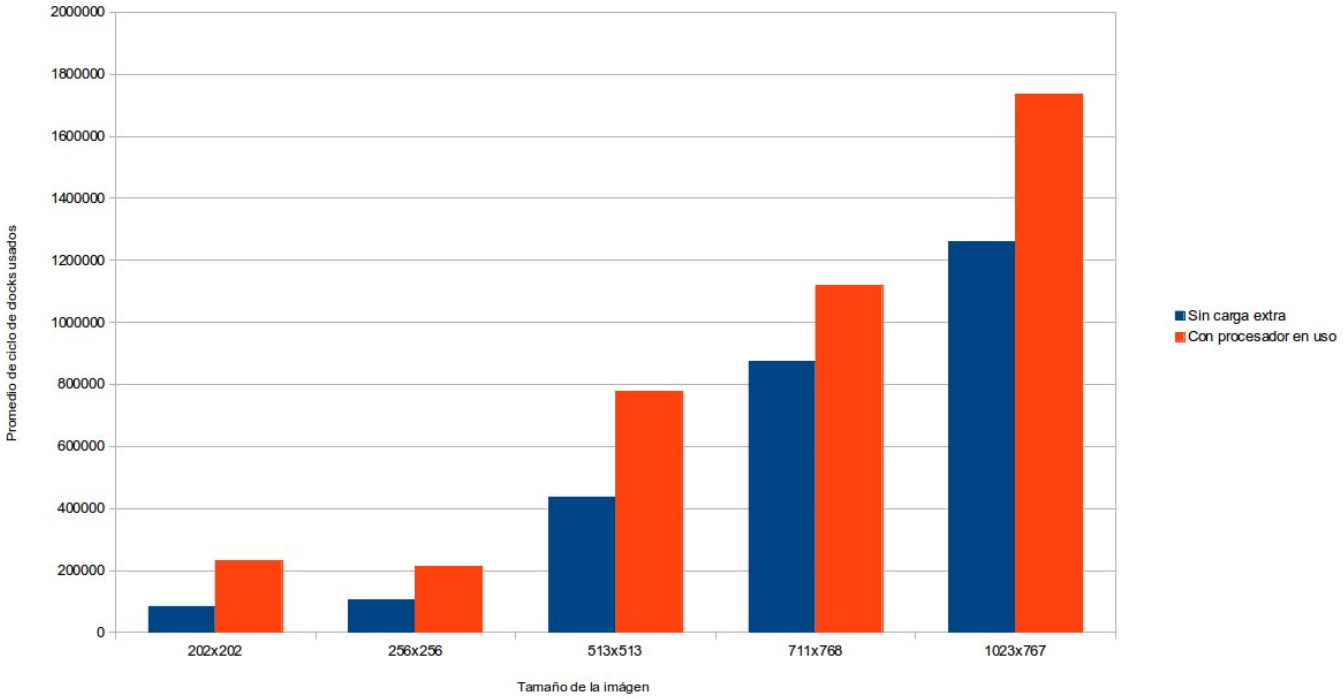
Aquí se aprecia como el procesador funciona mientras realizamos el experimento

Teniendo ya en claro que cumplíamos con la carga al procesador pasamos a realizar los análisis correspondientes nuevamente. Para esto repetimos el experimento 1000 veces y tomamos los promedios de las corridas.

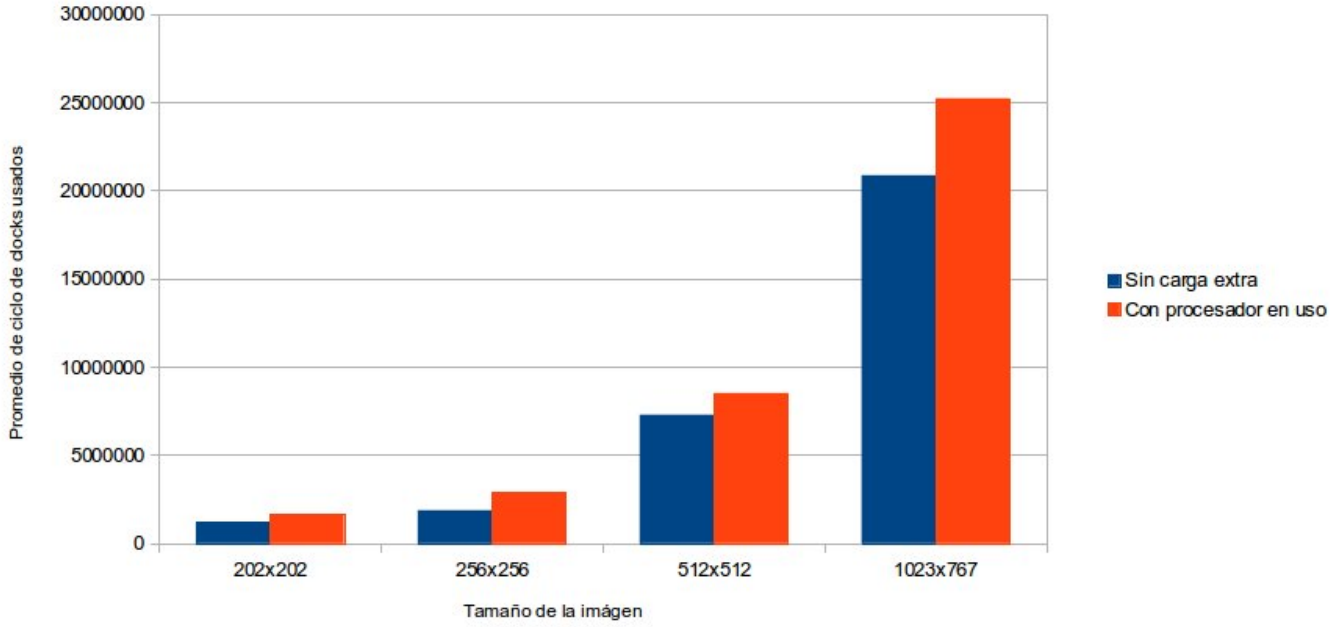
Resultados :



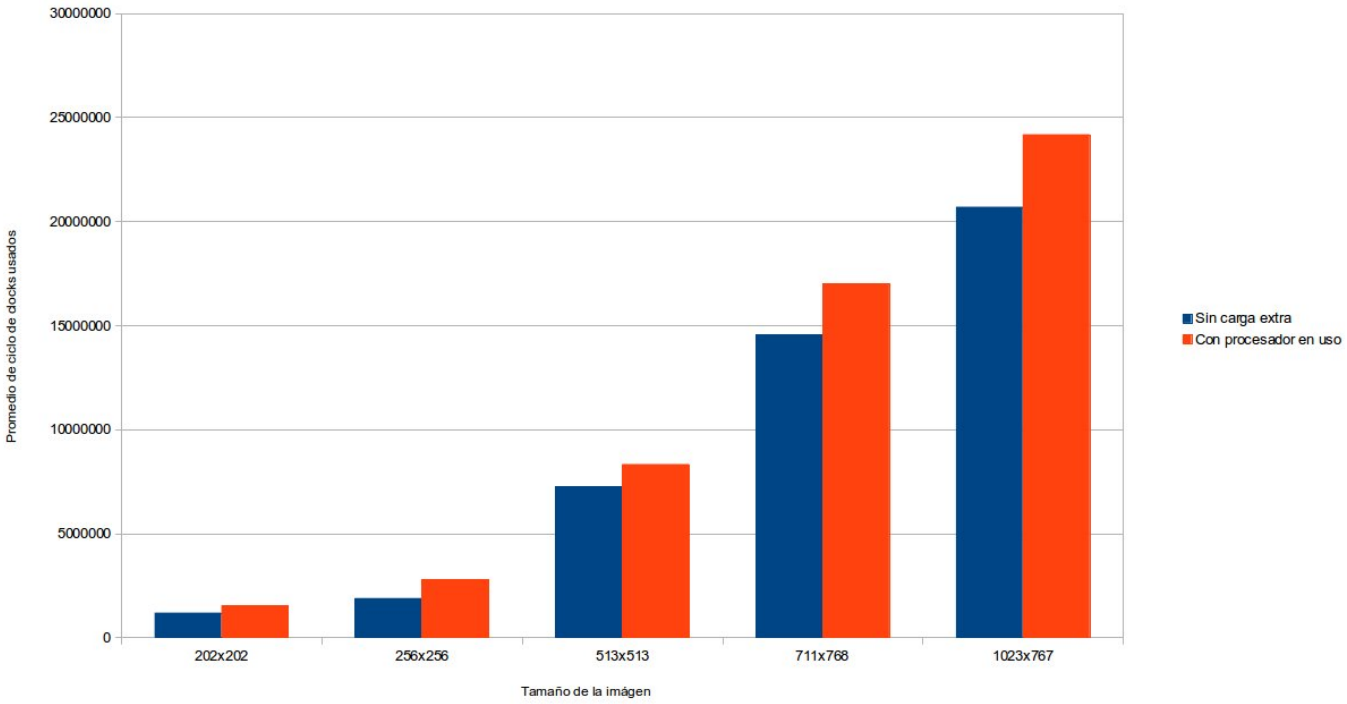
Diferencia de tiempo en la implementación de ASM en la imágen lena.bmp



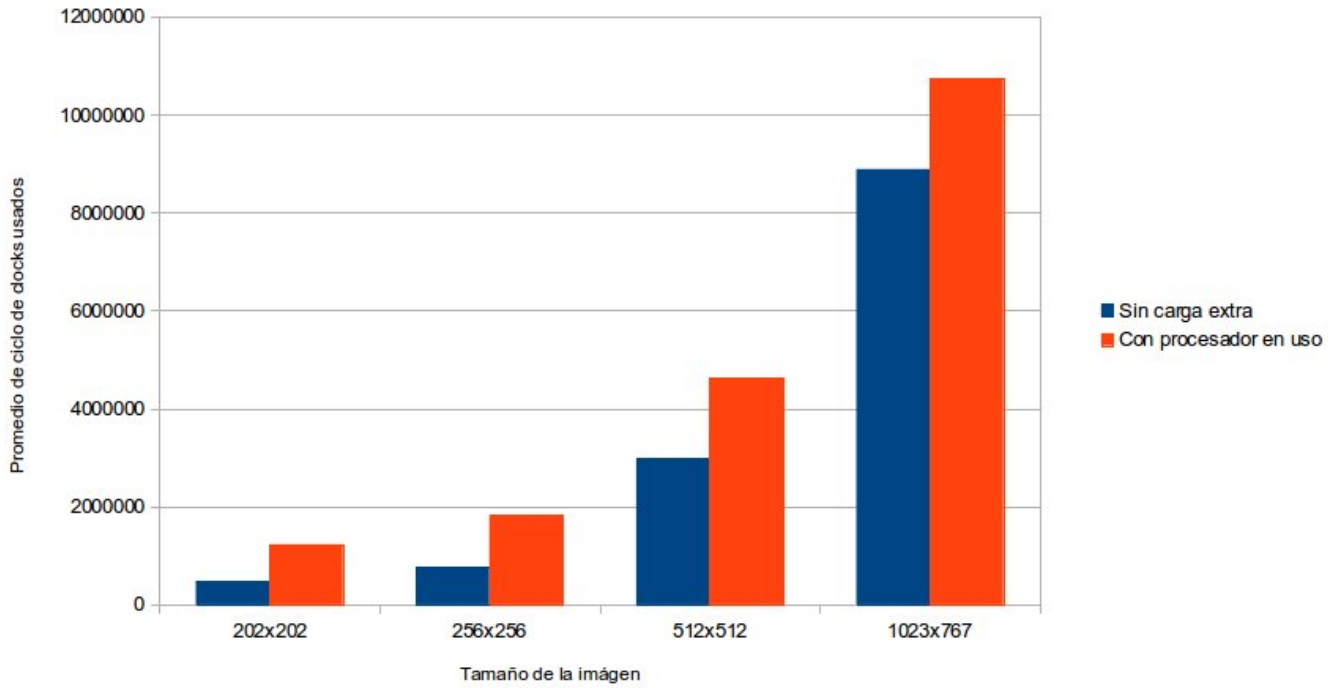
Diferencia de tiempo en la implementación de ASM en la imágen marilyn.bmp



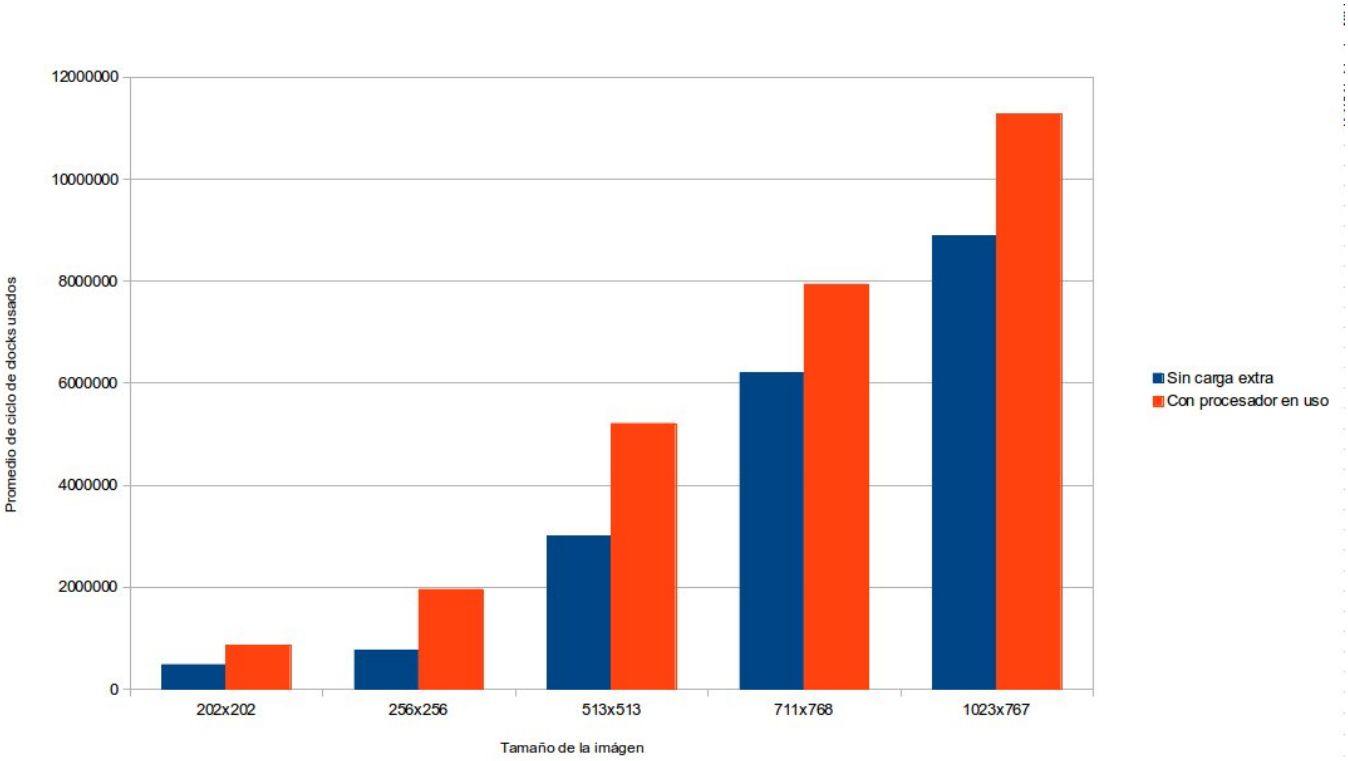
Diferencia de tiempo en la implementación de C en la imágen lena.bmp



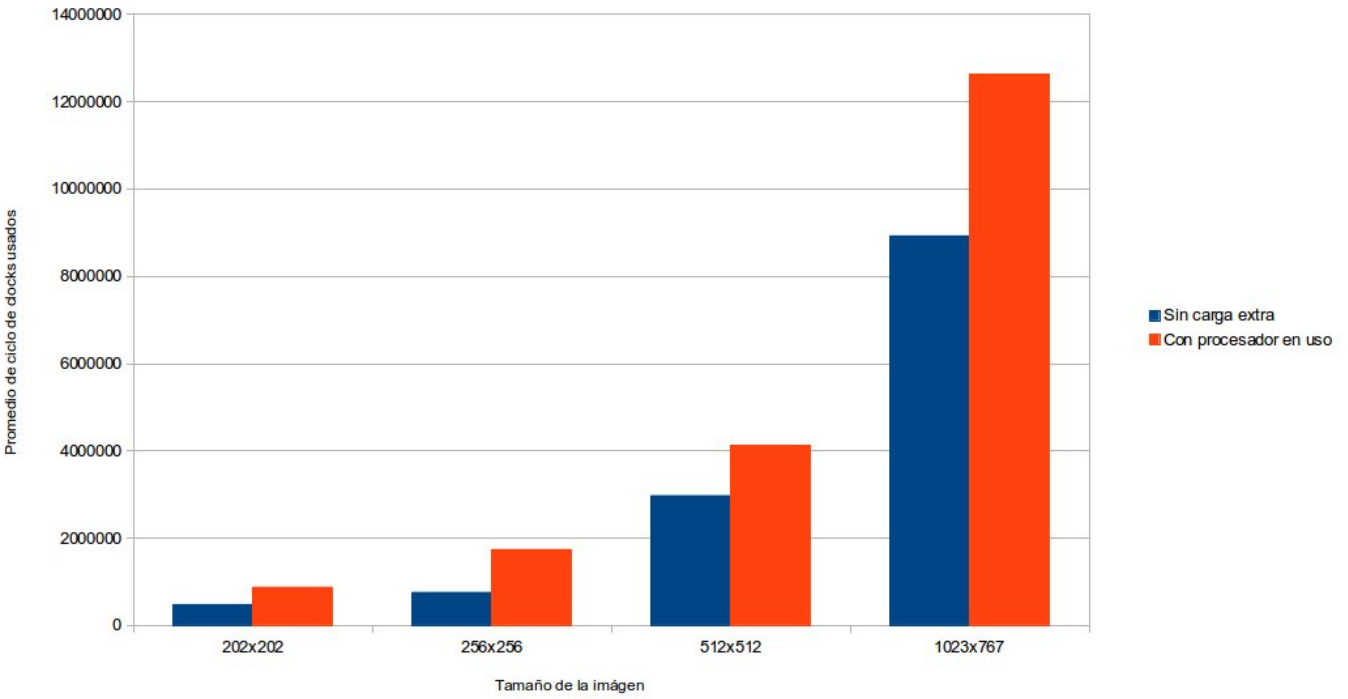
Diferencia de tiempo en la implementación de C en la imágen marilyn.bmp



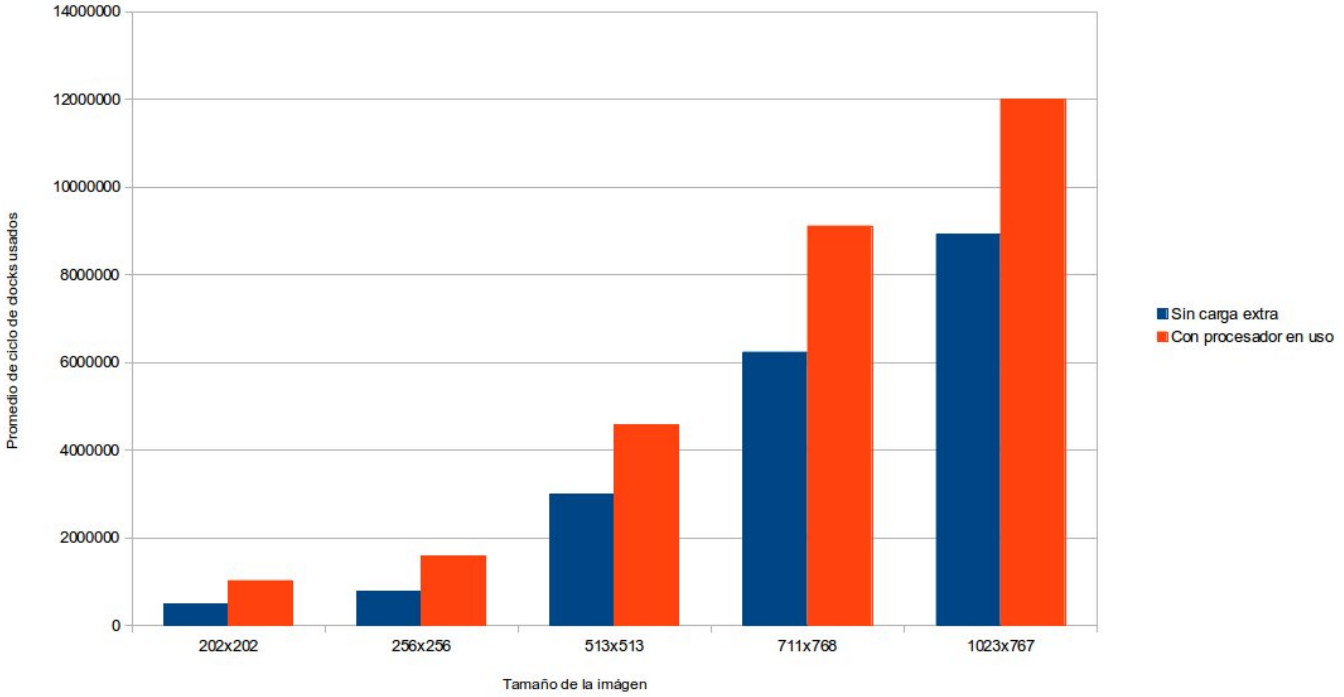
Diferencia de tiempo en la implementación de C optimizada con el flag -O1 en la imágen lena.bmp



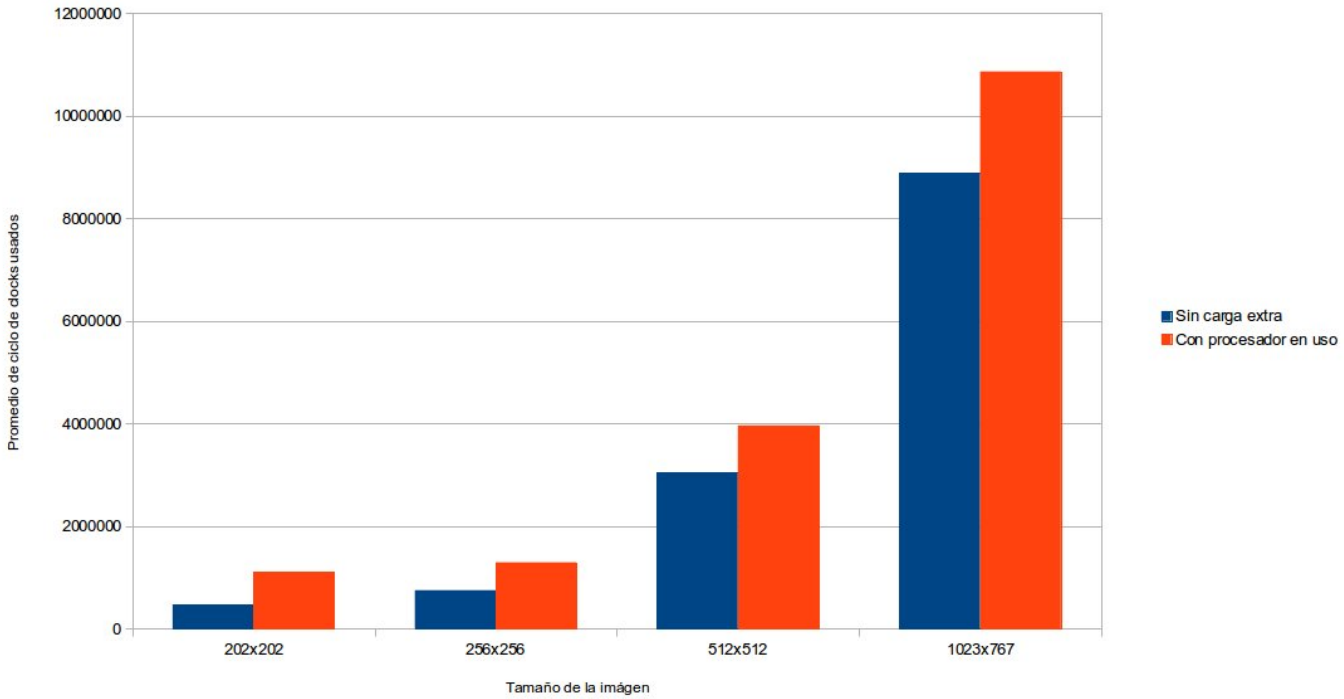
Diferencia de tiempo en la implementación de C optimizada con el flag -O1 en la imágen marilyn.bmp



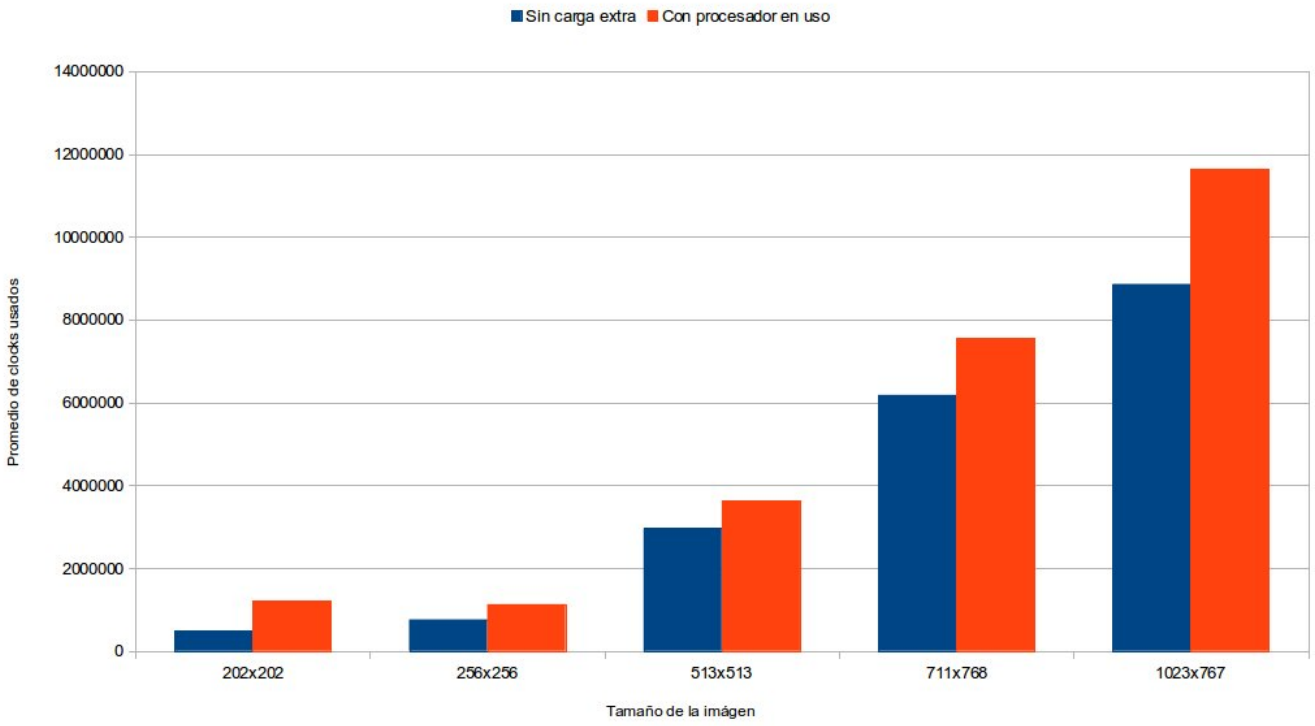
Diferencia de tiempo en la implementación de C optimizada con el flag -O2 en la imágen lena.bmp



Diferencia de tiempo en la implementación de C optimizada con el flag -O2 en la imágen marilyn.bmp



Diferencia de tiempo en la implementación de C optimizada con el flag -O3 en la imágen lena.bmp



Diferencia de tiempo en la implementación de C optimizada con el flag -O3 en la imágen lena.bmp

Teniendo en cuenta la gran diferencia que hay en el desempeño en la mayoría de los casos nos pareció pertinente mostrar las siguientes tablas donde se ve el porcentaje en que empeora la perfomance cuando el procesador esta siendo exigido por otros procesos.

Lena	202x202	256x256	512x512	1023x767
ASM	110 %	89 %	89 %	126 %
C	36 %	55 %	17 %	21 %
C -O1	154 %	138 %	55 %	21 %
C -O2	82 %	130 %	38 %	42 %
C -O3	134 %	72 %	30 %	22 %

Marilyn	202x202	256x256	513x513	711x768	1023x767
ASM	177 %	103 %	78 %	28 %	38 %
C	30 %	48 %	15 %	17 %	17 %
C -O1	77 %	155 %	73 %	28 %	27 %
C -O2	111 %	107 %	53 %	46 %	35 %
C -O3	147 %	48 %	23 %	22 %	31 %

Conclusiones :
Tal como esperabamos se vio una fuerte diferencia de velocidad yendo desde un 22 % más lento hasta a un 177 % lo cual implica un proceso casi 3 veces más lento.
Por otro lado la implementación de C sin optimizar es la que menos se vio afectada por la sobrecarga del procesador. Como analizamos anteriormente las optimizaciones realizadas por el compilador tienen un fuerte acento en limitar los accesos a memoria. Dado que los accesos a memoria no se ven tan afectados por la carga del procesador y la mayoria de los ciclos de la implementación de C se dedican a ello la performance no se ve tan afectada.
Todos estos cambios en la performance evidentemente se explican porque el procesador no puede dedicar a nuestro algoritmo la misma cantidad de clocks que hacía anteriormente pues debe atender los otros procesos que se están ejecutando a la vez.

1.4. Experimento 4

Para este experimento decidimos que para testear la velocidad del algoritmo con más accesos a memoria agregaríamos el siguiente código dentro del ciclo (en este filtro son 2 ciclos distintos) del algoritmo:

```
movdqu xmm6, [R13]
movdqu [R13], xmm6
movdqu xmm6, [R13]
```

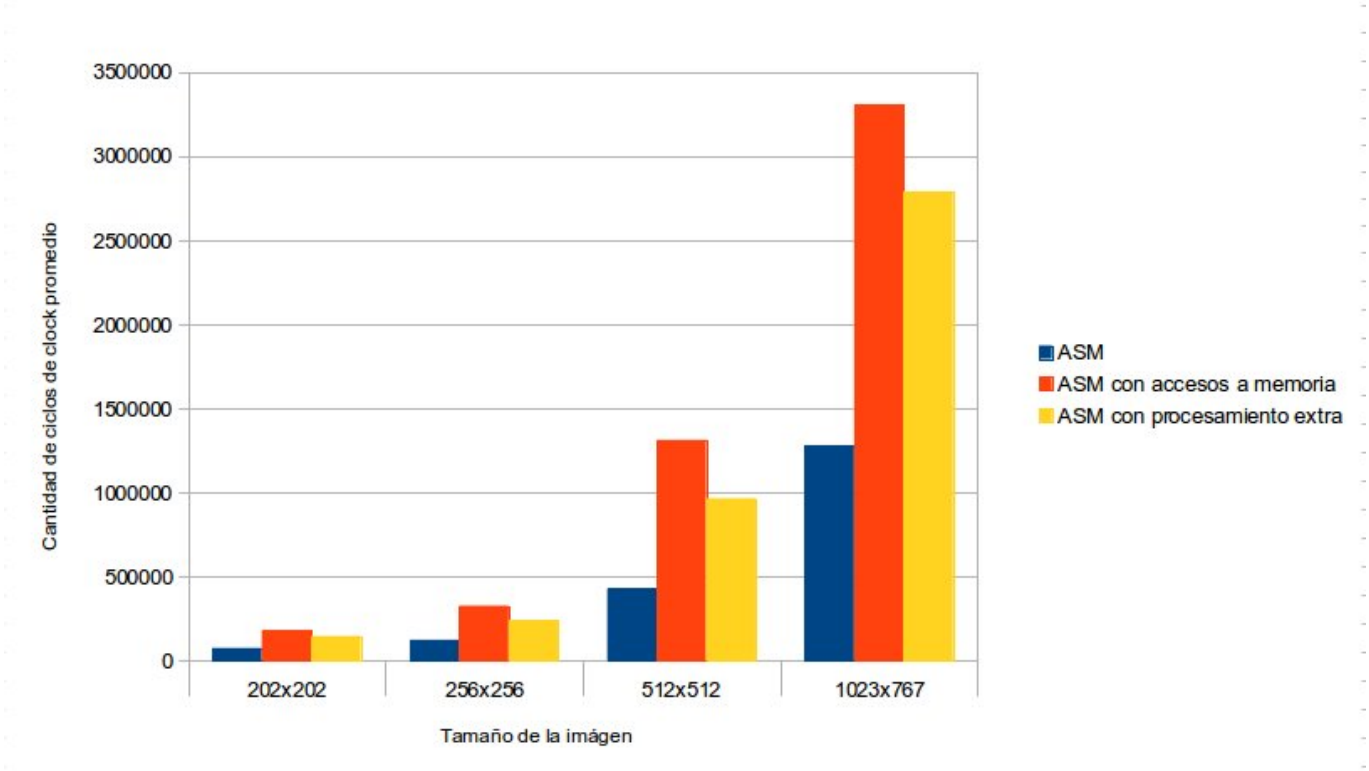
```
movdqu [R13], xmm6
movdqu xmm6, [R13]
movdqu [R13], xmm6
movdqu xmm6, [R13]
movdqu [R13], xmm6
movdqu xmm6, [R13]
movdqu [R13], xmm6
movdqu xmm6, [R13]
movdqu [R13], xmm6
movdqu [R13], xmm6
movdqu [R13], xmm6
movdqu [R13], xmm6
```

Por otro lado para exigirle al procesador más intensidad de cómputo decidimos agregar el siguiente segmento de código:

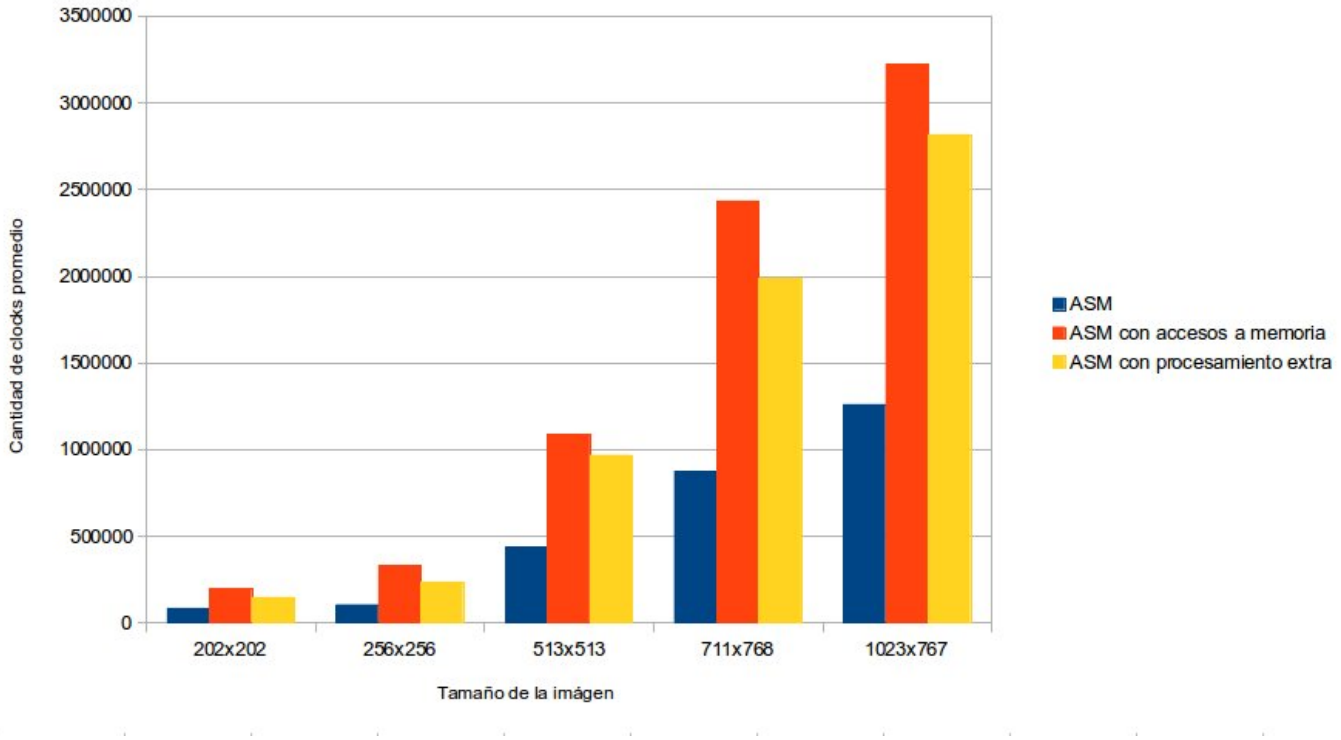
```
movdqu xmm6, [R13]
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
cvtps2pd xmm7, xmm6
```

Compilamos nuestro nuevo código de assembler y ejecutamos las pruebas pertinentes, nuevamente utilizando los mismos parámetros de siempre y tomando promedios sobre 1000 iteraciones.

Resultados :



Diferencia de tiempo en las distintas implementaciones de ASM en la imágen lena.bmp con distintos tamaños



Diferencia de tiempo en las distintas implementaciones de ASM en la imágen marilyn.bmp con distintos tamaños

Conclusiones :
Como pudimos observar en los gráficos anteriores los nuevos segmentos de código afectan fuertemente a la performance. Obviamente este comportamiento se ve acentuado al agrandar el tamaño de la imágen, ya que esto implica más ciclos donde se ejecutará este codigo agregado, por lo tanto más accesos a memoria y más conversiones
Por otro lado se puede apreciar a simple vista el peso que tienen los accesos a memoria en el tiempo total de ejecución del filtro. Ambos agregados de código tienen un impacto sustancial, pero pudimos concluir que son los accesos a memoria los que limitan la performance del filtro implementado en el mdoelo *SIMD*.

2. Popart

Explicación del algoritmo :
La idea del algoritmo es tener un ciclo que procese de a 5 pixels, osea levantar de a 16 bytes para procesar 15. Antes de comenzar el ciclo se calcula la cantidad de iteraciones que se van a realizar. Cómo en los demás algoritmos, el caso borde a tratar es el del fin de la fila, en cuyo caso, lo que debo hacer es retroceder para poder levantar los ultimos 16 bytes, salvando el byte menos significativo, y trabajando con los 15 bytes más significativos (los últimos 5 pixels).
El procesamiento de datos no tiene mayores dificultades, es simplemente realizar la suma para cada pixel, y realizar las comparaciones. Para poder hacerlas contamos con máscaras en memoria que tienen las constantes contra las cuales vamos a comparar. Teniendo los resultados de las comparaciones, generamos resultados "disjuntos"para después poder simplemente realizar un *por* y unificar los resultados.
Algo importante a destacar, es que, a la hora de realizar las sumas, éstas se realizan en *words*, dado que un *byte* no alcanza para representar numéricamente la suma de los 3 componentes.

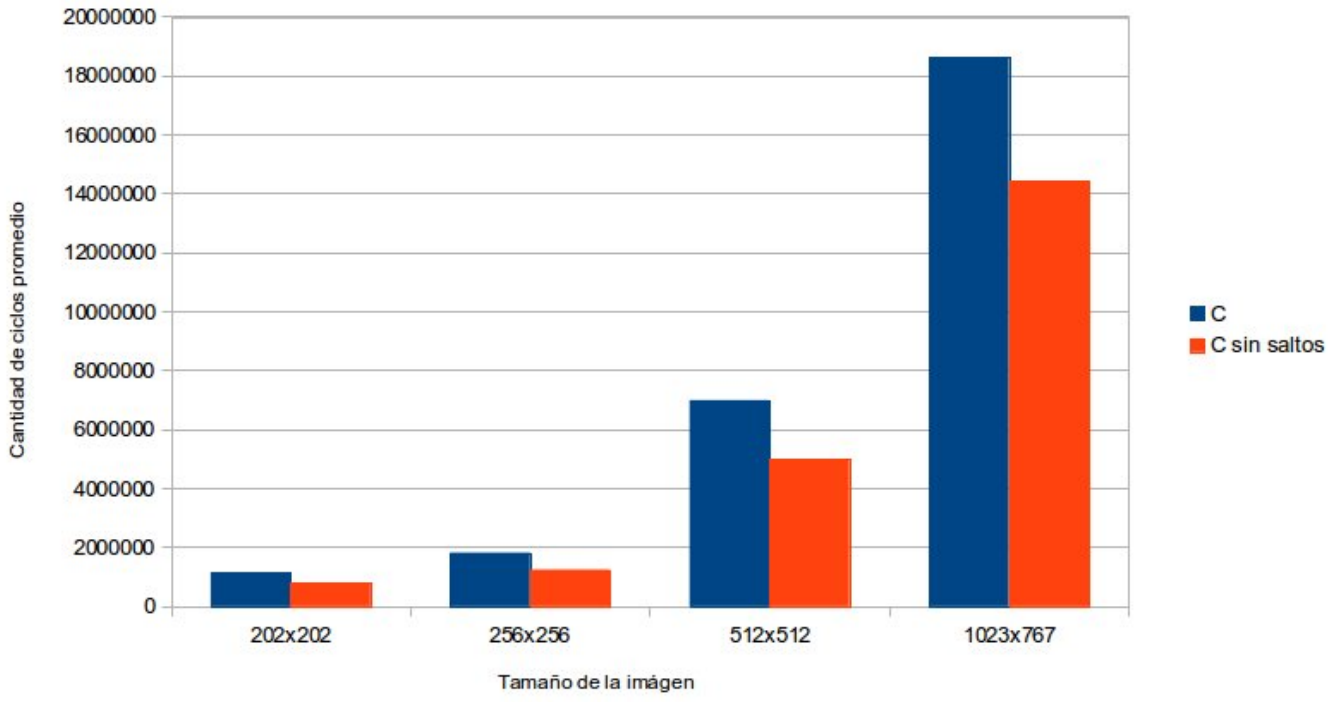
2.1. Experimento 1

Para este experimento comentamos los saltos condicionales del código de C. Aunque la imagen generada no sea la misma, el tiempo de cómputo del código que realiza el procesamiento es el mismo.
Dado la naturaleza de este experimento debemos primero aclarar las especificaciones de la terminal donde realizamos las mediciones:

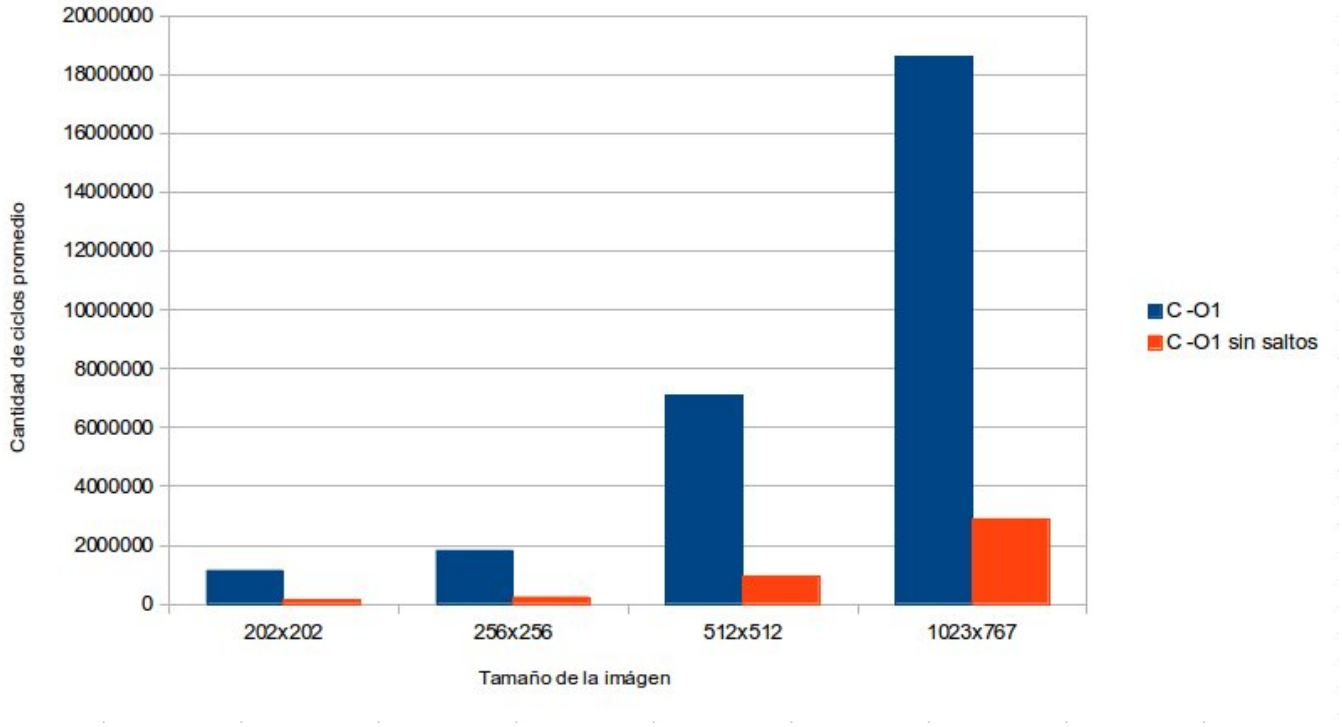
```
CPU: Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz
Number of cores: 4
Arquitectura: 64 Bit
Memoria RAM: 8 GB

Maquina número 13 del laboratorio 4, Departamento de Computación, UBA.
```

Para eliminar la posibilidad de que outliers modifiquen el resultado del experimento realizamos siempre 1000 ejecuciones de cada implementación, medimos el tiempo total y tomamos un promedio. La medición en ningún momento toma en cuenta los tiempos de apertura y cierre de las imágenes.
Se puede asumir que utilizamos la misma computadora y el mismo método de medición en todos los experimentos de este filtro.
Nota: En el anexo se pueden ver las tablas utilizadas para la creación de los gráficos anteriores.

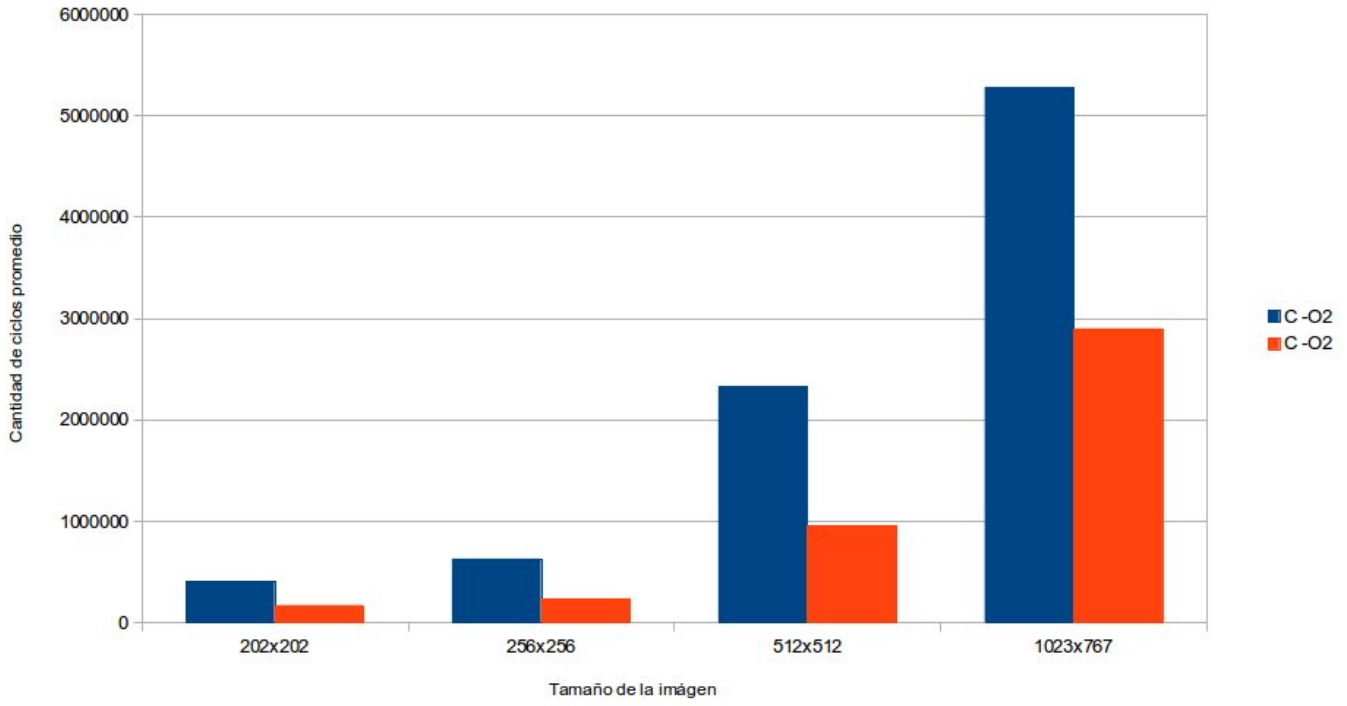
Resultados :



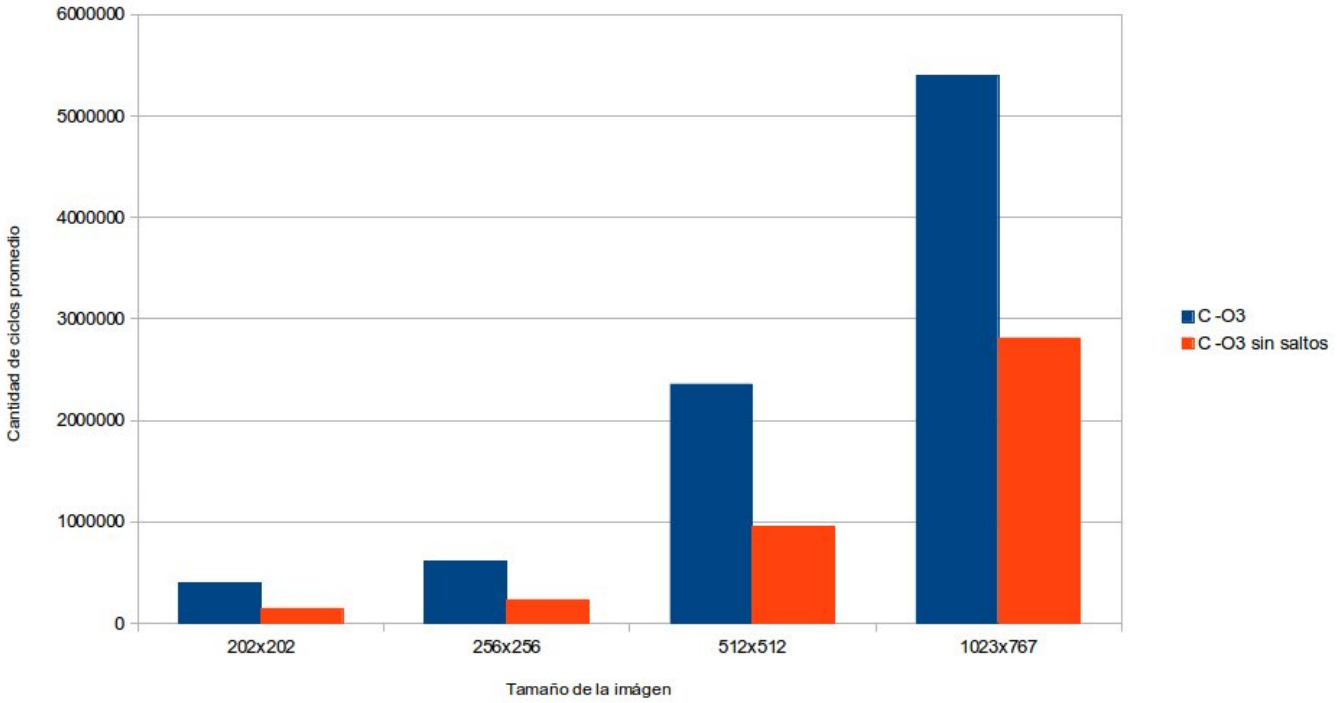
Diferencia de tiempo en las distintas implementaciones de C en la imágen lena.bmp con distintos tamaños



Diferencia de tiempo en las distintas implementaciones de C con flag de optimización -O1 en la imágen lena.bmp con distintos tamaños



Diferencia de tiempo en las distintas implementaciones de C con flag de optimización -O2 en la imágen lena.bmp con distintos tamaños



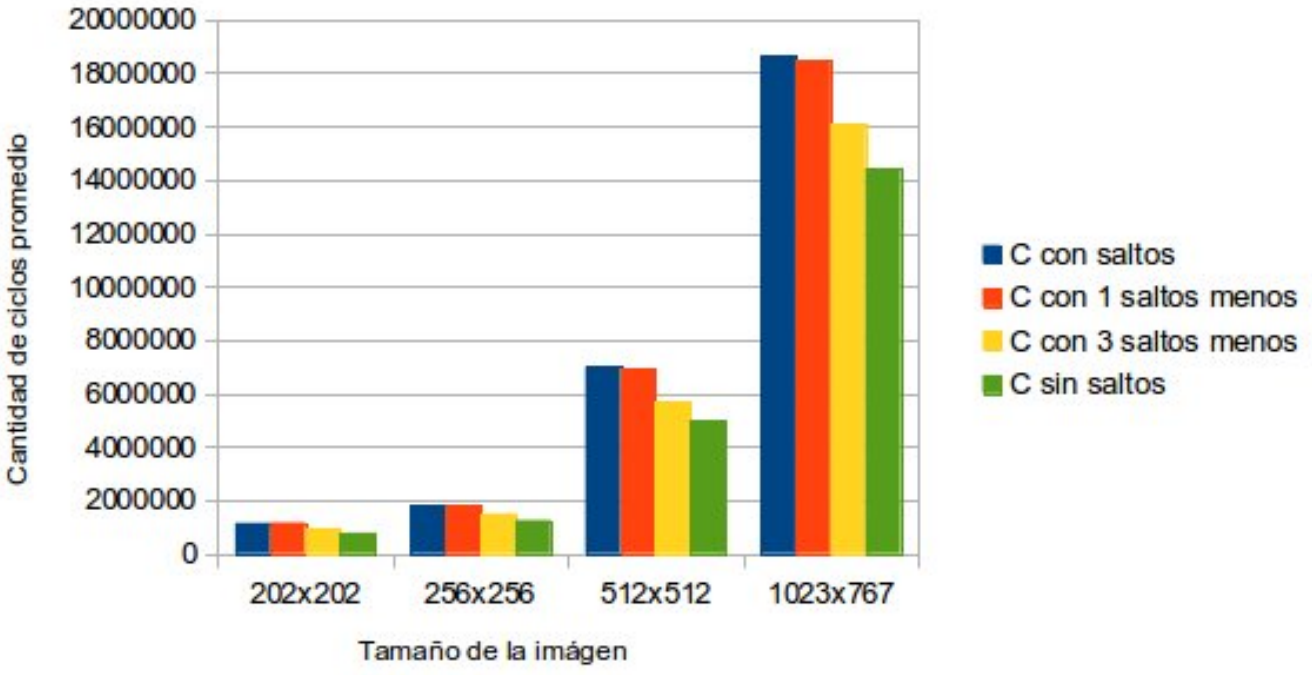
Diferencia de tiempo en las distintas implementaciones de C con flag de optimización -O3 en la imágen lena.bmp con distintos tamaños

Conclusión :
En los gráficos anteriores se puede apreciar el peso que tienen los saltos condicionales a la hora de la ejecución. A pesar de ser un factor fundamental en los tiempos de ejecución, a medida que se optimiza el código los saltos tienen cada vez menos incidencia en los tiempos totales. Creemos que esto se debe a que de por si la optimización permite evitar en parte el costo de los saltos condicionales.

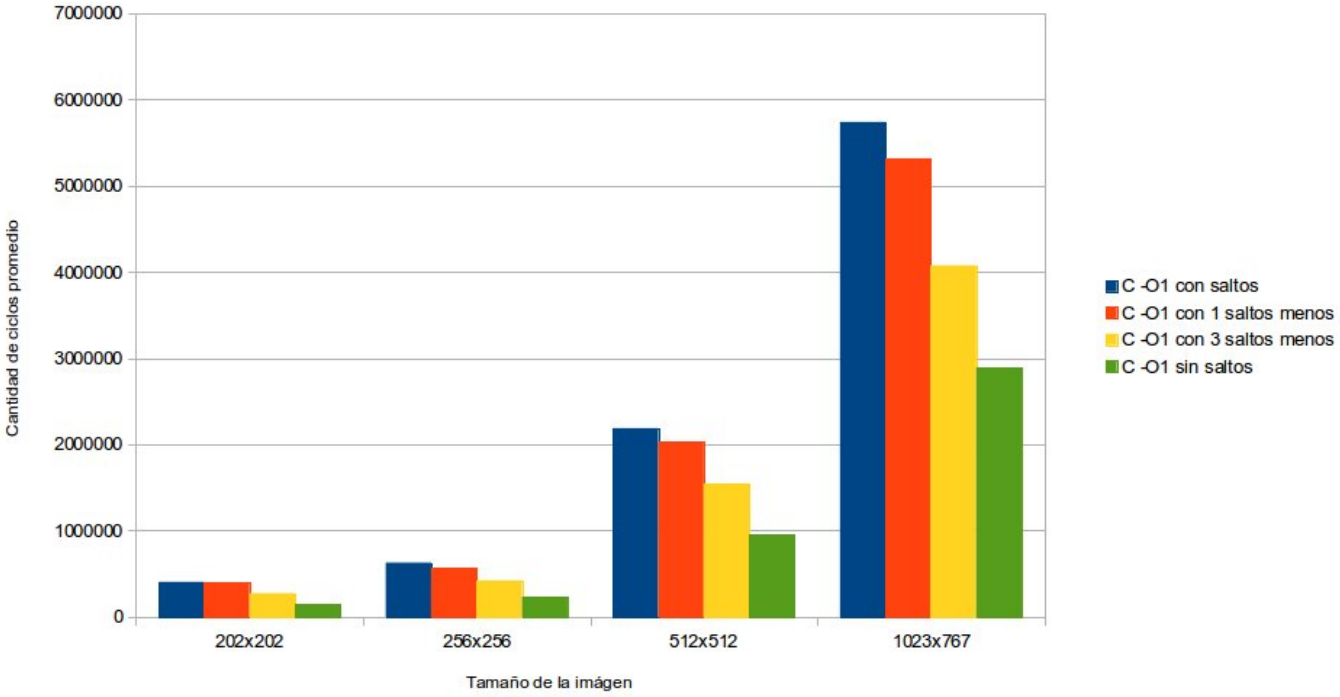
2.2. Experimento Agregado

Para profundizar aún más en el experimento anterior decidimos probar como funcionaria el código con diversas cantidades de saltos condicionales. Los gráficos a continuación demuestran los resultados. Notese que obviamos el análisis del caso de ASM por lo aclarado al comienzo del experimento 2.

Resultados :

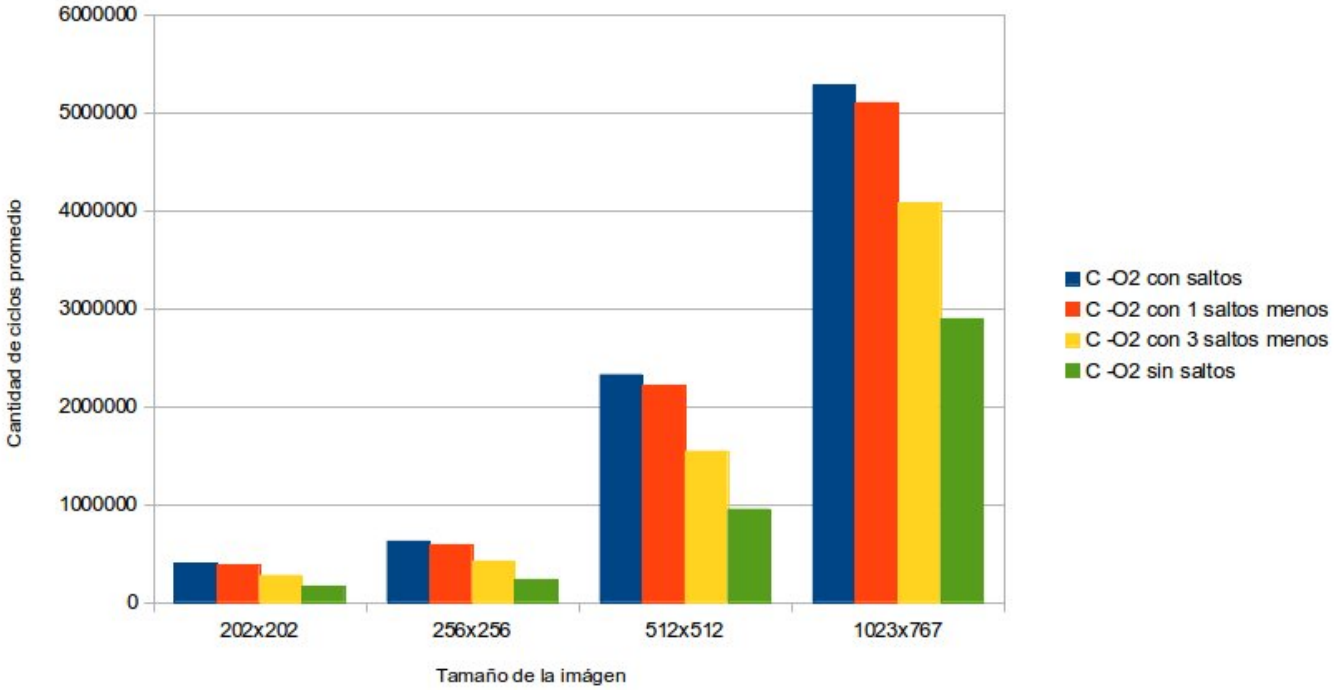


Diferencia de tiempo en el código de C ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen lena.bmp

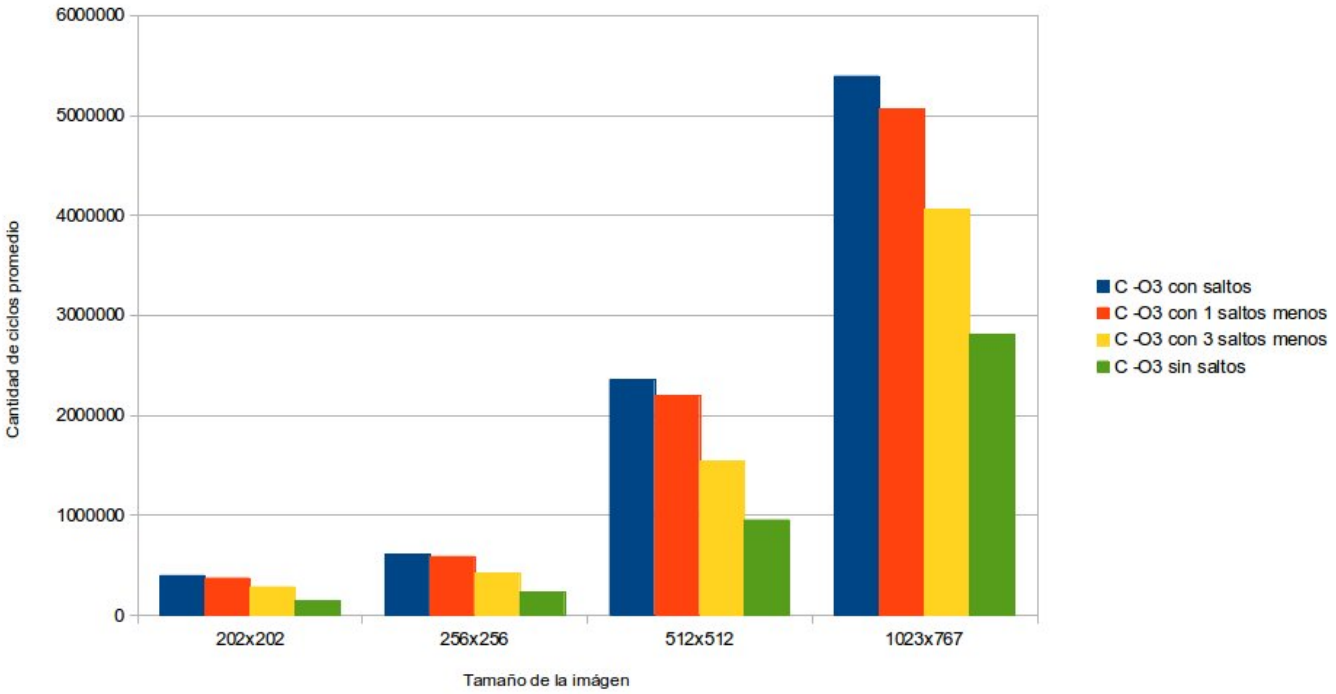


Diferencia de tiempo en el código de C con flag de optimización -O1 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen lena.bmp

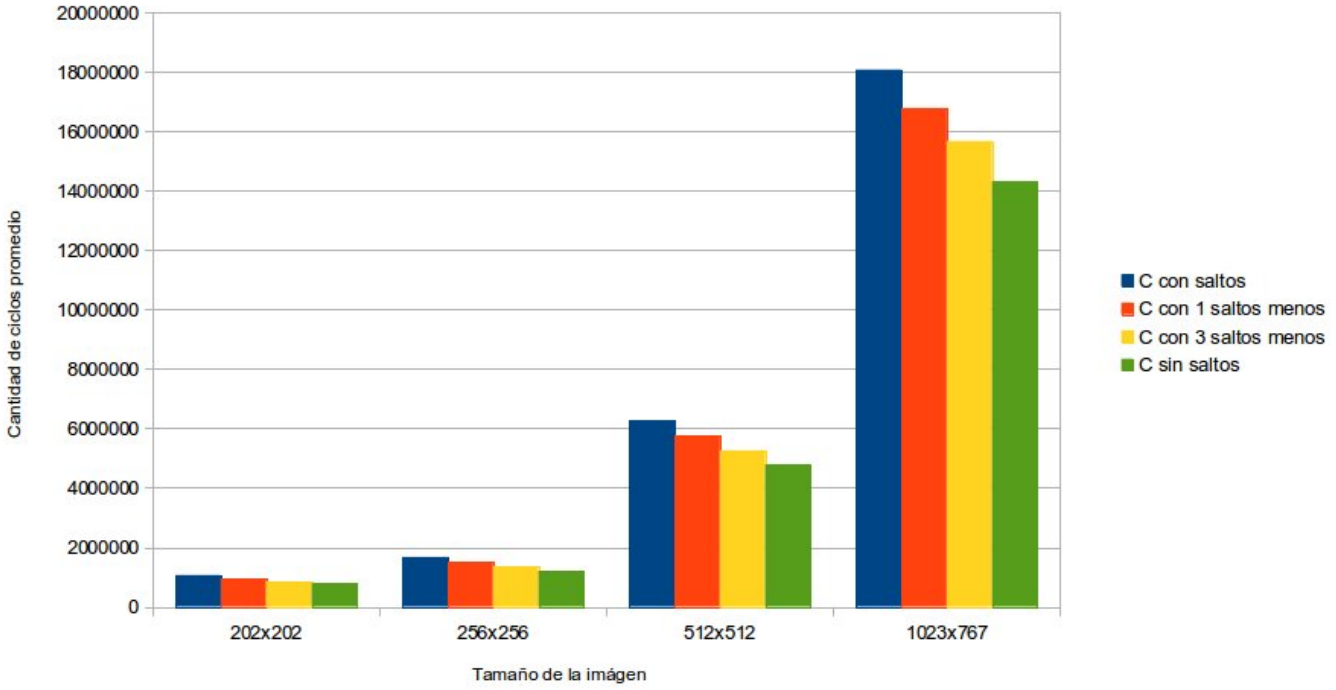
Diferencia de tiempo en el código de C con flag de optimización -O1 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen lena.bmp



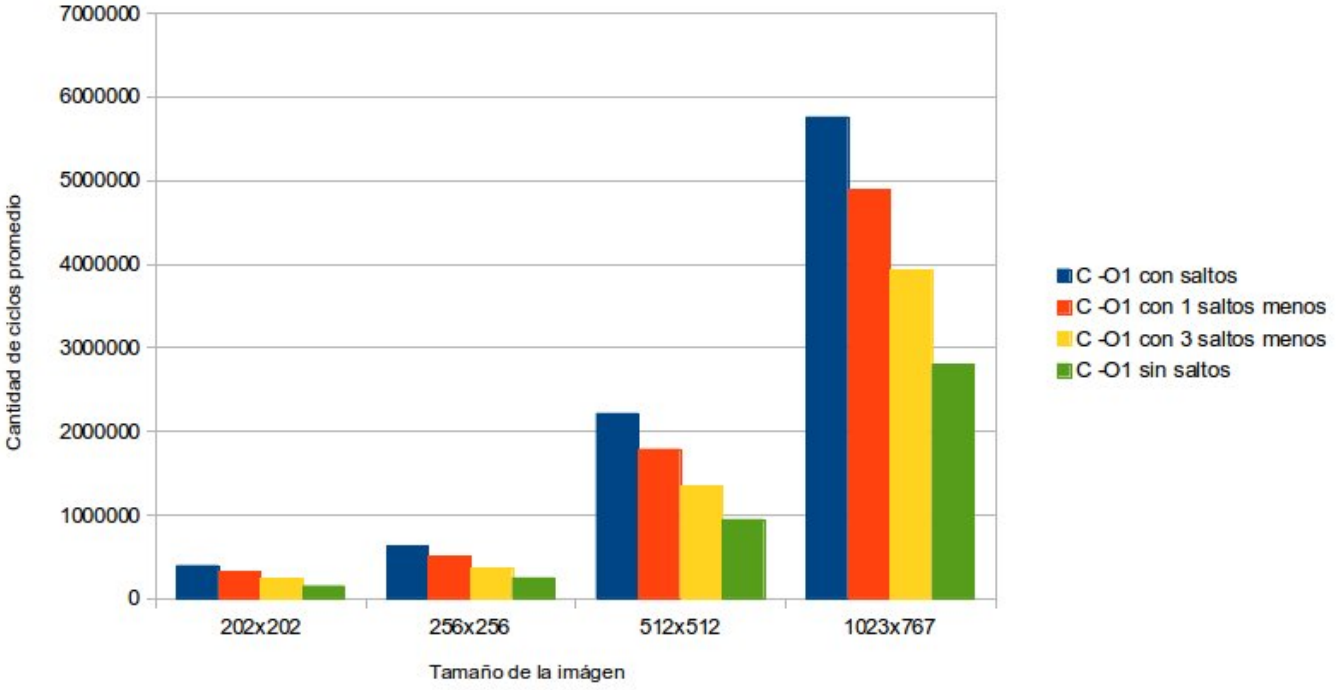
Diferencia de tiempo en el código de C con flag de optimización -O2 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen lena.bmp



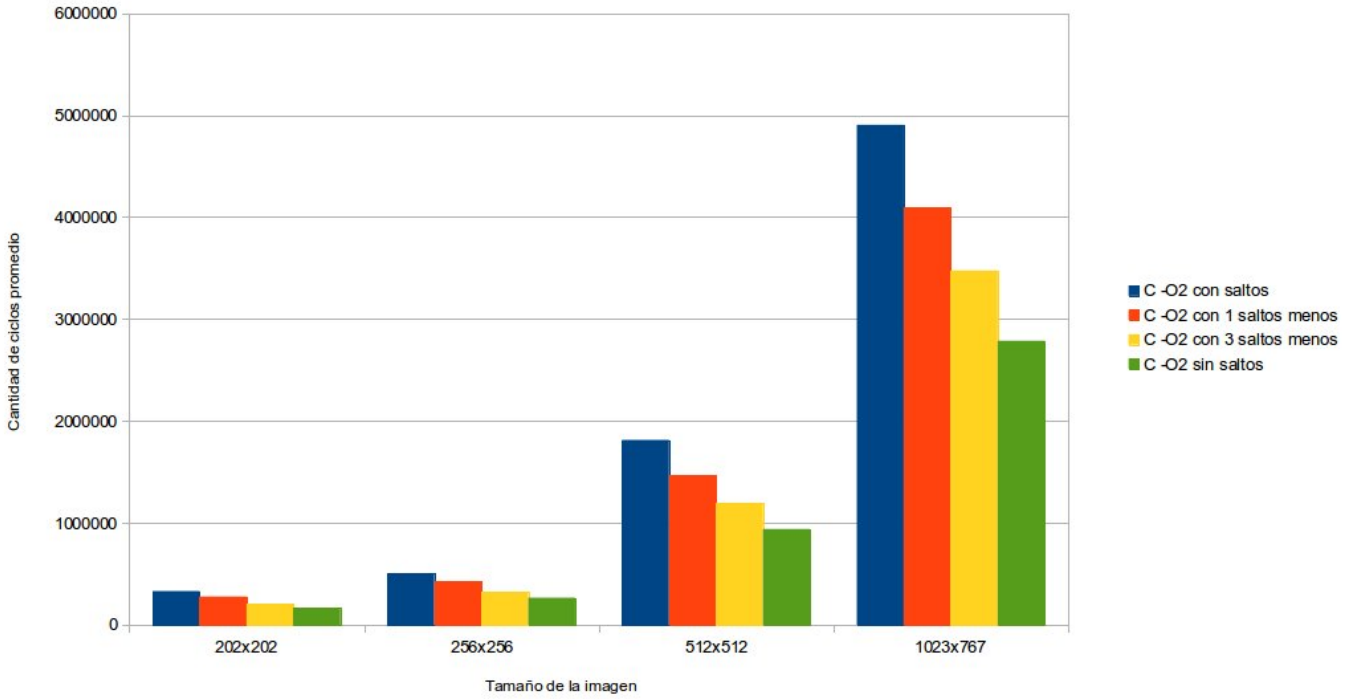
Diferencia de tiempo en el código de C con flag de optimización -O3 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen lena.bmp



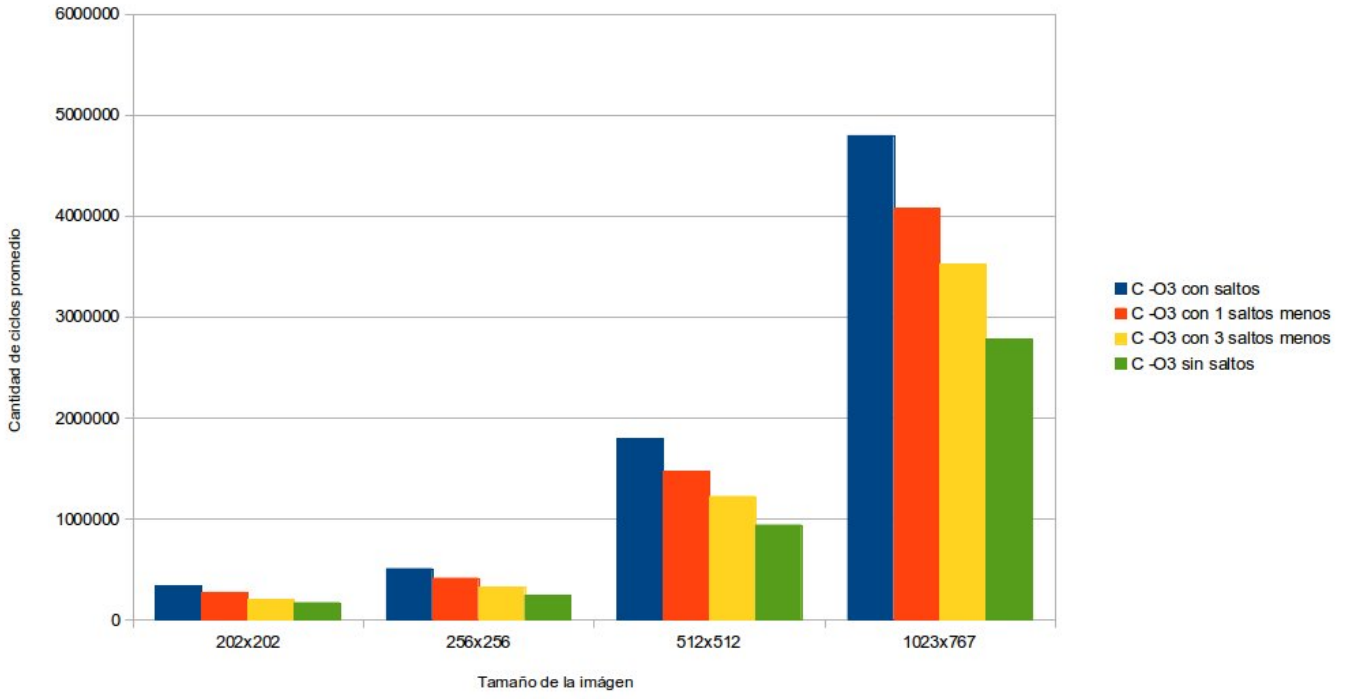
Diferencia de tiempo en el código de C ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen marilyn.bmp



Diferencia de tiempo en el código de C con flag de optimización -O1 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen marilyn.bmp



Diferencia de tiempo en el código de C con flag de optimización -O2 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen marilyn.bmp



Diferencia de tiempo en el código de C con flag de optimización -O3 ejecutando distinta cantidad de saltos condicionales. Este gráfico representa el proceso sobre la imágen marilyn.bmp

Conclusiones :
Como se puede ver en lo gráficos anteriores los saltos condicionales son instrucciones que le toman bastante tiempo al procesador, tanto es asi que la performance en ocasiones mejora más de un 50 % al quitar por completo este tipo de comparaciones.
Ademas, detallando el experimento como lo hicimos se ve mejor el peso de cada salto por si mismo. Se puede apreciar que quitando un salto la performance mejora entre un entre un 2 % y un 9 %.
Nos hubiera gustado poder comparar los costos de un salto condicional de C contra uno de ASM, pero por lo explicado anteriormente sobre nuestro código de ASM no nos fue posible.

2.3. Experimento 2

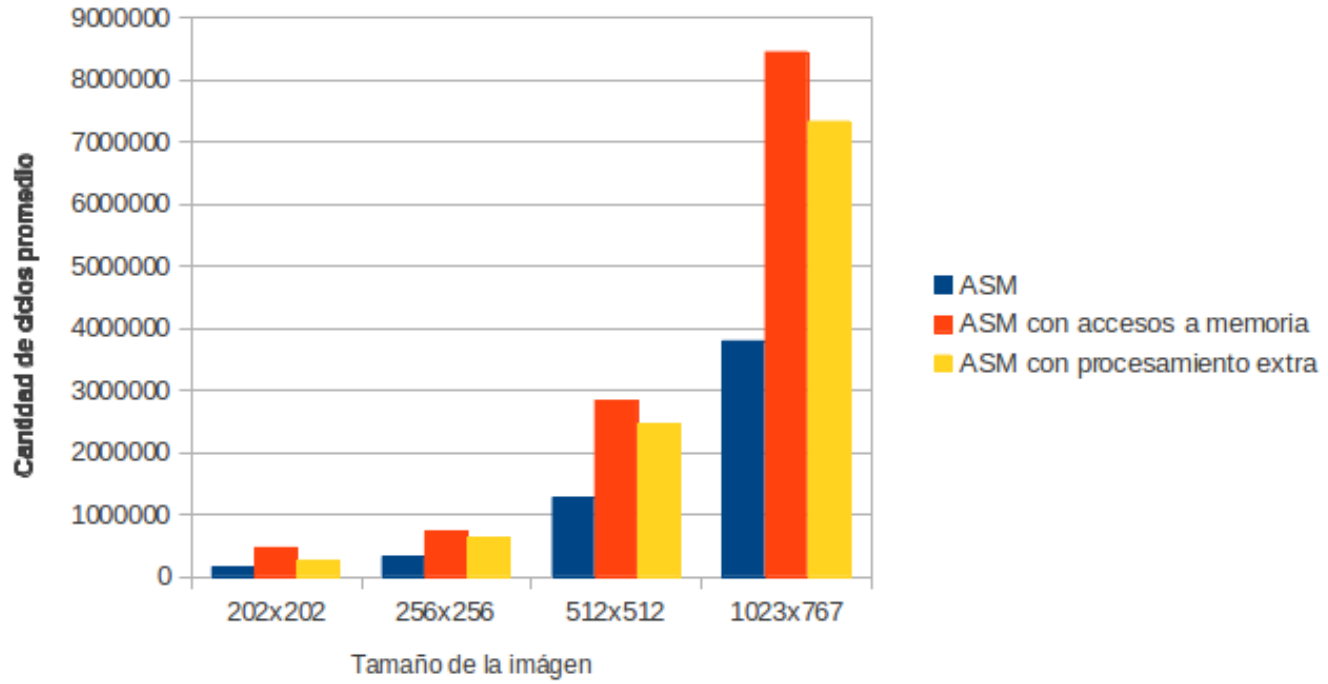
Para este experimento debimos agregar código en los ciclos del procesamiento de la imágen. En esos códigos hicimos accesos a memoria y conversiones para poder comparar su incidencia y su peso en el tiempo total del procesamiento. A a continuación se detallan los códigos agregados.
Para acceder a memoria:

```
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu xmm0, [RDI]
movdqu [RDI], xmm0
movdqu [RDI], xmm0
movdqu [RDI], xmm0
movdqu [RDI], xmm0
```

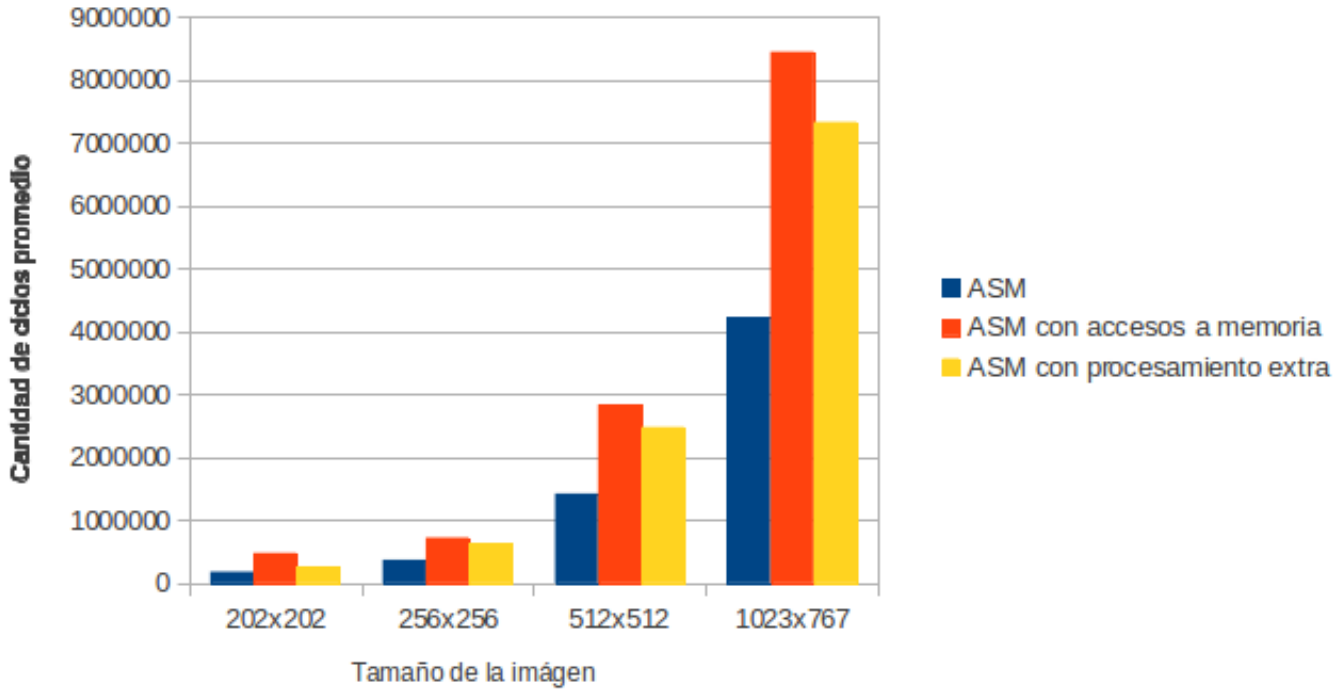
Para exigir más procesamiento de datos:

```
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
cvtps2pd xmm15, xmm0
```

Resultados :
A continuación se muestran los gráficos que muestran la relación entre los tiempos de las distintas implementaciones de ASM.



Diferencia de tiempo en la implementación de ASM dependiendo si se agregó código. En este caso se utilizó la imágen lena.bmp



Diferencia de tiempo en la implementación de ASM dependiendo si se agregó código. En este caso se utilizó la imágen marilyn.bmp

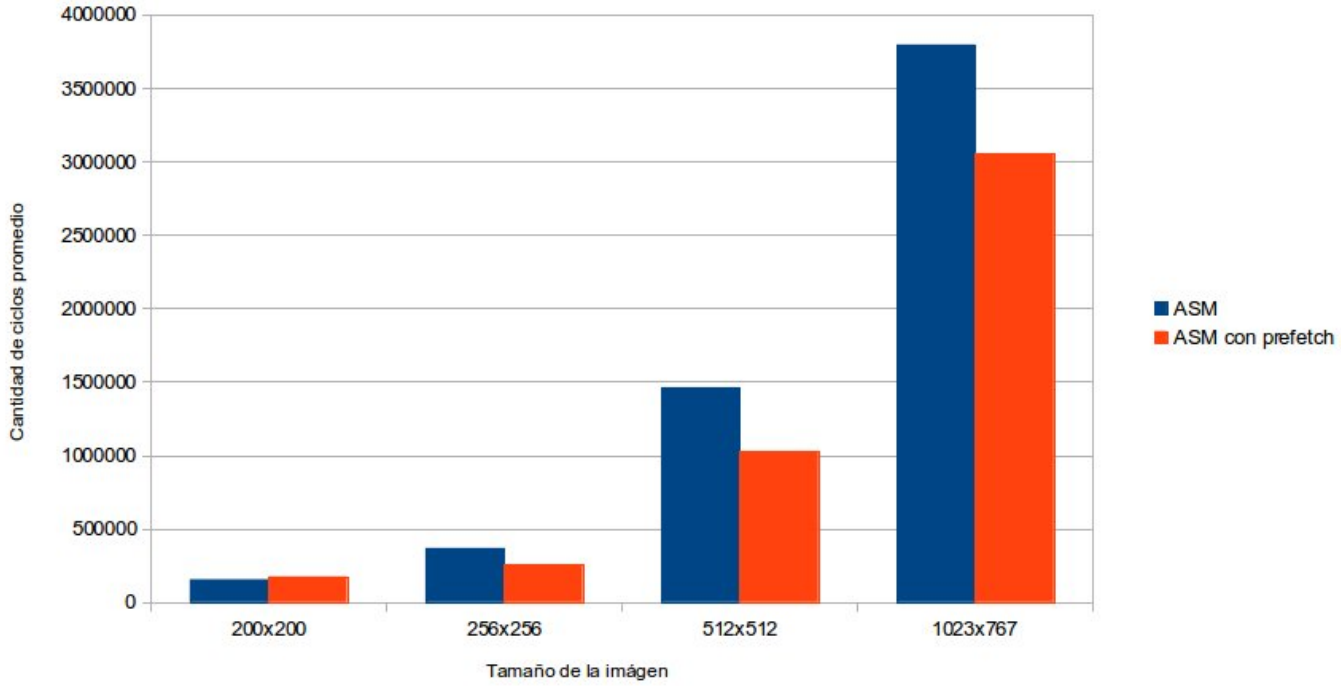
Conclusiones :
Como en el filtro anterior, se puede apreciar como estas modificaciones implican un fuerte aumento en la cantidad de ciclos necesarios para la ejecución del algoritmo.
Como se puede ver en los resultados obtenidos, ambos agregados tienen un impacto sobre la performance, aunque como esperábamos los accesos a memoria influyen más, es decir, son más *caros* temporalmente hablando.
Luego de la realización del experimento, concluímos que podría haber casos aún peores que el realizado, ya que al realizar los accesos a memoria siempre levantabamos las mismas posiciones sin tener en cuenta la optimizacion de la cache, la cual guarda esa sección de memoria para poder ser accedida con mayor velocidad en un futuro.

2.4. Experimento 3

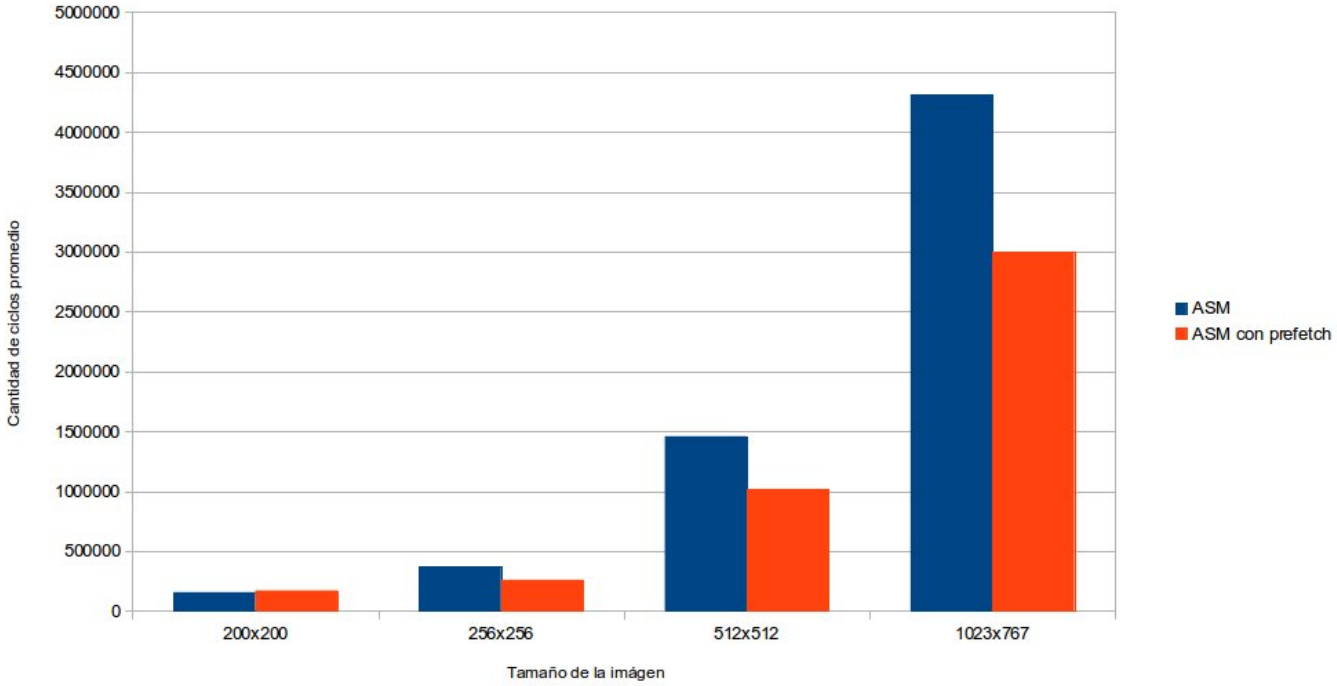
En el presente experimento se nos solicitó indagar sobre la técnica de *prefetch*. Para ello en el ciclo principal vamos a levevantar los datos necesarios para el procesamiento del ciclo inmediatamente posterior. Para evitar posibles casos bordes mal resueltos decidimos limitar el *prefetch* solo al ciclo principal, dejando de lado los casos del fin de la fila.

Nota: El código utilizado para el *prefetch* puede verse en el anexo.

Resultados :



Diferencia de tiempo en la implementación de ASM con y sin prefetch. En este caso se utilizó la imágen lena.bmp



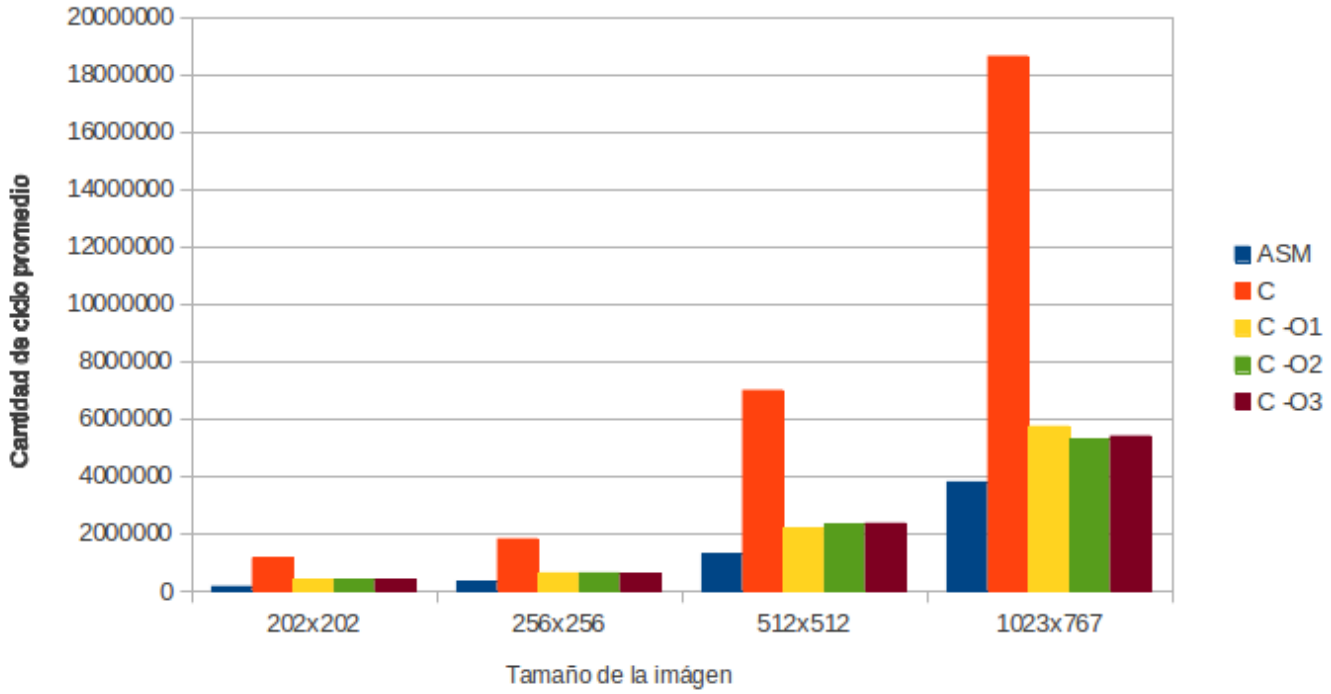
Diferencia de tiempo en la implementación de ASM con y sin prefetch. En este caso se utilizó la imágen marilyn.bmp

Conclusiones :
En casi todos los casos se puede observar como el agregado del código necesario para hacer prefetch tiene una incidencia positiva en los tiempos de ejecución. Para nuestra sorpresa en imágenes pequeñas el agregado de código tiene un

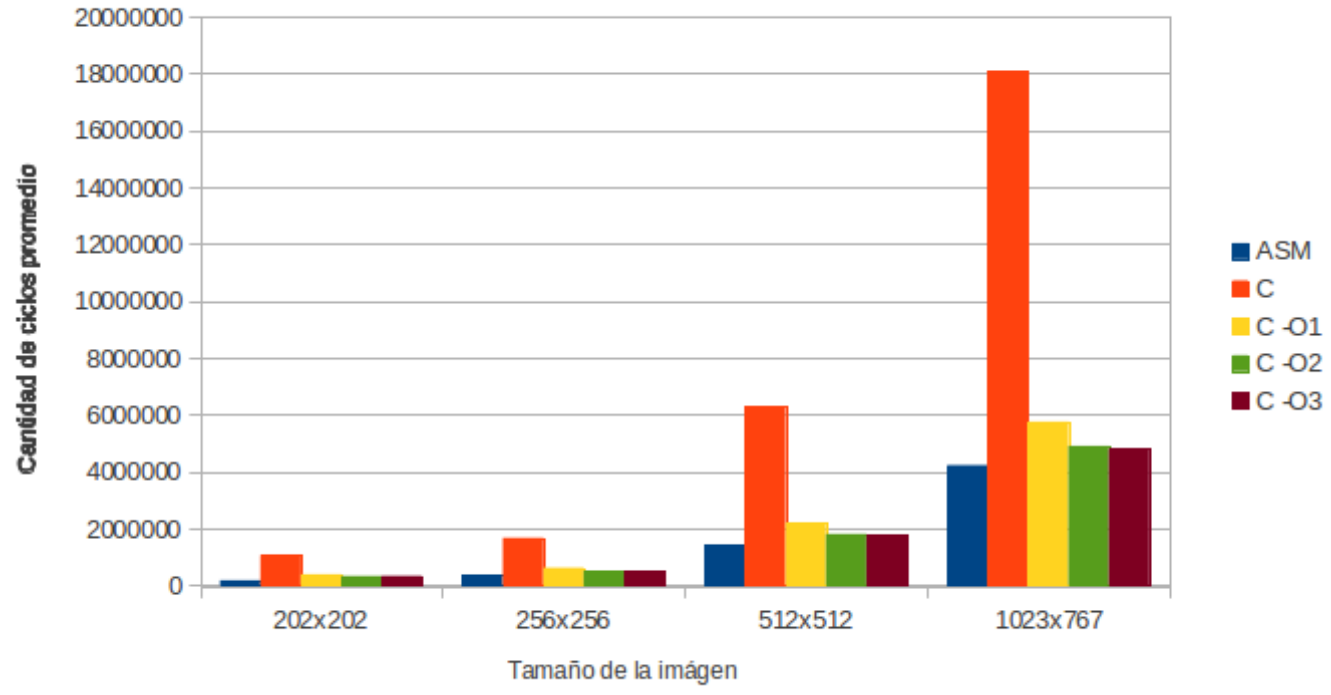
impacto levemente negativo. Esto puede deberse a que el prefetch que realiza el procesador es limitado y entorpecido por nuestra implementación, logrando un efecto negativo en la performance
Como aclaramos en el anexo debimos agregar al código un acceso a memoria, pero no nos parece que eso explique esta inconsistencia de las imágenes pequeñas.
Por otro lado sabemos el que procesador realiza automáticamente optimizaciones de prefetch y puede que nuestro código sea menos eficiente en ciertos casos que el realizado por el procesador sobre el código sin prefetch.
Igualmente puede verse en las tablas anexadas que la diferencia temporal es menor a los 20000 ciclos lo cual si bien no es exactamente despreciable es una diferencia pequeña.

2.5. Experimento 4

Resultados :



Diferencia de tiempo en las distintas implementaciónes del filtro. En este caso se utilizó la imágen lena.bmp



Diferencia de tiempo en las distintas implementaciónes del filtro. En este caso se utilizó la imágen marilyn.bmp

Conclusiones :
Como era de esperar la implementación del filtro en ASM con la utilización de SIMD requiere menos ciclos para finalizar. La diferencia con C es sorprendentemente grande, aunque con las implementaciones de C optimizadas pudimos observar que no habia tanta diferencia. Creemos que esto puede deberse a que en ASM debemos realizar algunos accesos a memoria para trabajar con las máscaras definidas y esto en C puede ser fuertemente optimizado por el compilador.
Es evidente que el uso de instrucciones que permitan utilizar la técnica de SIMD mejora mucho la performance del filtro.

3. Temperature

Explicación del algoritmo :
El algoritmo que produce el efecto *temperature* sobre la imágen es muy parecido al de popart. La forma de recorrer la imágen es la misma, el caso borde también. En lo único que se diferencian es en que éste necesita realizar una división (el promedio de los 3 componentes del pixel). Para realizar dicha operación es necesario hacer un pasaje a punto flotante y luego realizar la división utilizando aritmética de punto flotante.

3.1. Experimento 1

Para este experimento decidimos comparar las implementaciones de c y assembler con el objetivo de analizar las distintas performances. Para eso, variamos la implementación de c, optimizandola con distintos flags proporcionados por la cátedra.
Las especificaciones de la máquina en la cual realizamos las mediciones son las siguientes:

```
CPU: Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz
Number of cores: 4
Arquitectura: 64 Bit
Memoria RAM: 8 GB

Maquina número 13 del laboratorio 4, Departamento de Computación, UBA.
```

Cabe destacar que para eliminar la posibilidad de que outliers modifiquen el resultado del experimento, realizamos siempre 1000 ejecuciones de cada implementación, medimos el tiempo total y tomamos un promedio. La medición en ningún momento toma en cuenta los tiempos de apertura y cierre de las imágenes.

Resultados

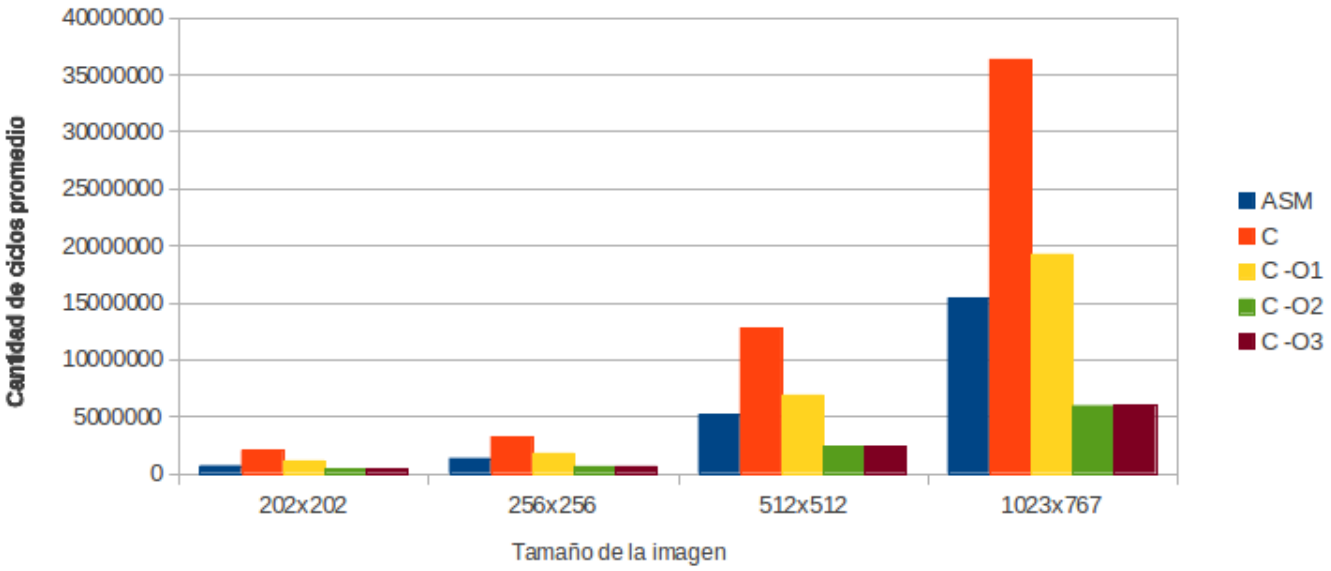


Gráfico variando tamaños e implementaciones de lena

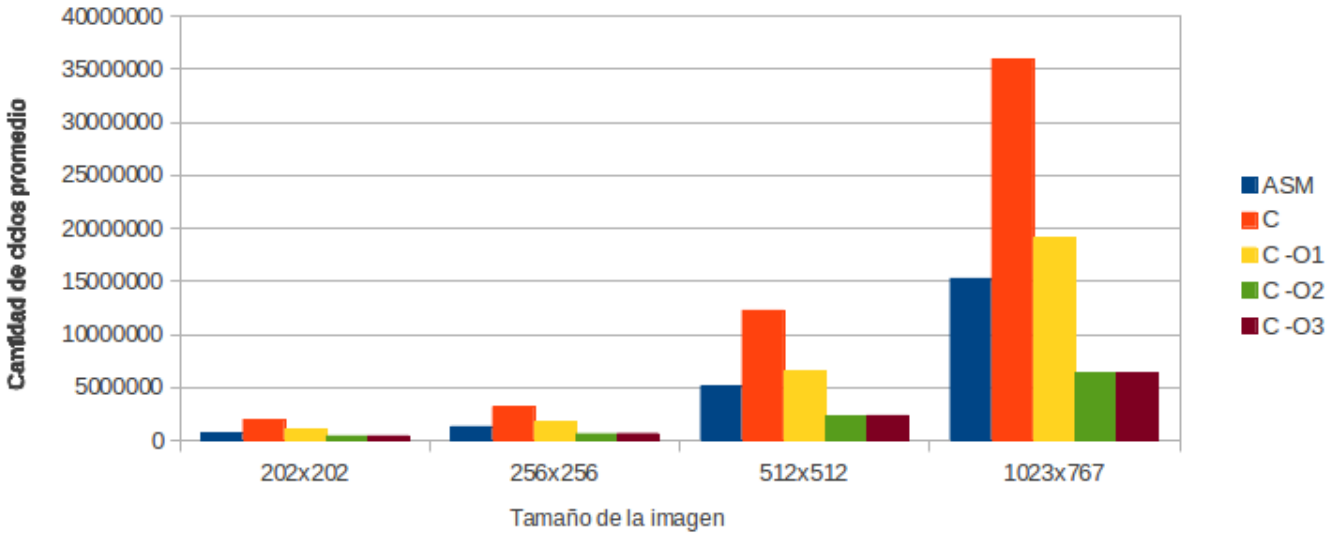


Gráfico variando tamaños e implementaciones de marilyn

Conclusiones :
En ambos gráficos pudimos observar que las implementaciones de C con optimizaciones de tipo -O2 y -O3 eran más rápidas que las de assembler. Esto se debe a que en la implementación de assembler, dentro del ciclo principal, hay demasiados accesos a memoria con el objetivo de levantar máscaras.Esta variación de performance, puede ser atribuida a los saltos condicionales utilizados dentro del ciclo para hacer las comparaciones que, como ya fue comprobado en la experimentación de popart, ralentizan considerablemente la velocidad. Otra observación importante es que C no utiliza aritmética de punto flotante para realizar las operaciones de división entera, mientras que en assembler no existen instrucciones de la tecnología SSE que no requieran aritmética de punto flotante.
Con respecto a la comparacíon entre los tamaños, observamos una disminución de la velocidad a medida que aumentamos los tamaños de las imágenes. También notamos que la versión de C optimizada con -O2 y -O3 es mas rápida que la de assembler, sin importar el tamaño de la imágen que utilizemos.
Finalmente, cabe destacar que no observamos grandes diferencias de peformance al comparar las imágenes de lena y marilyn, más bien sus tiempos de cómputo son bastante similares, como puede observarse en los cuadros expuestos en el anexo.

4. LDR

Explicación del algoritmo :
La idea del algoritmo es, primero con un ciclo procesar el *interior* de la imágen, y luego escribir el *recuadro*.
El primer ciclo procesa de a 4 pixels, con lo cual necesita levantar 8 pixels de la imágen fuente. Se usan dos registros para leer los 24 bytes de cada fila, y se van sumando a 4 registros, que van acumulando las sumas *verticales* correspondientes a cada componente de cada pixel. Una vez obtenidas las sumas parciales, se realizan las multiplicaciones con *alpha* y *src* en enteros. Luego se realiza el pasaje a *double* para hacer las divisiones con presición doble. Para el pasaje a entero se toma la parte entera, truncando el resultado. Por último, luego de sumar *src* se empaquetan los resultados y se escribe sobre la matriz destino.
Una vez terminado esto, queda escribir las filas y columnas originales sobre la imágen destino. Para hacer esto se realizaron tres ciclos. Uno para copiar las primeras dos filas, otro para copiar las últimas dos, y un tercero para escribir las 4 columnas (las dos primeras, y las dos últimas).

4.1. Experimento 1

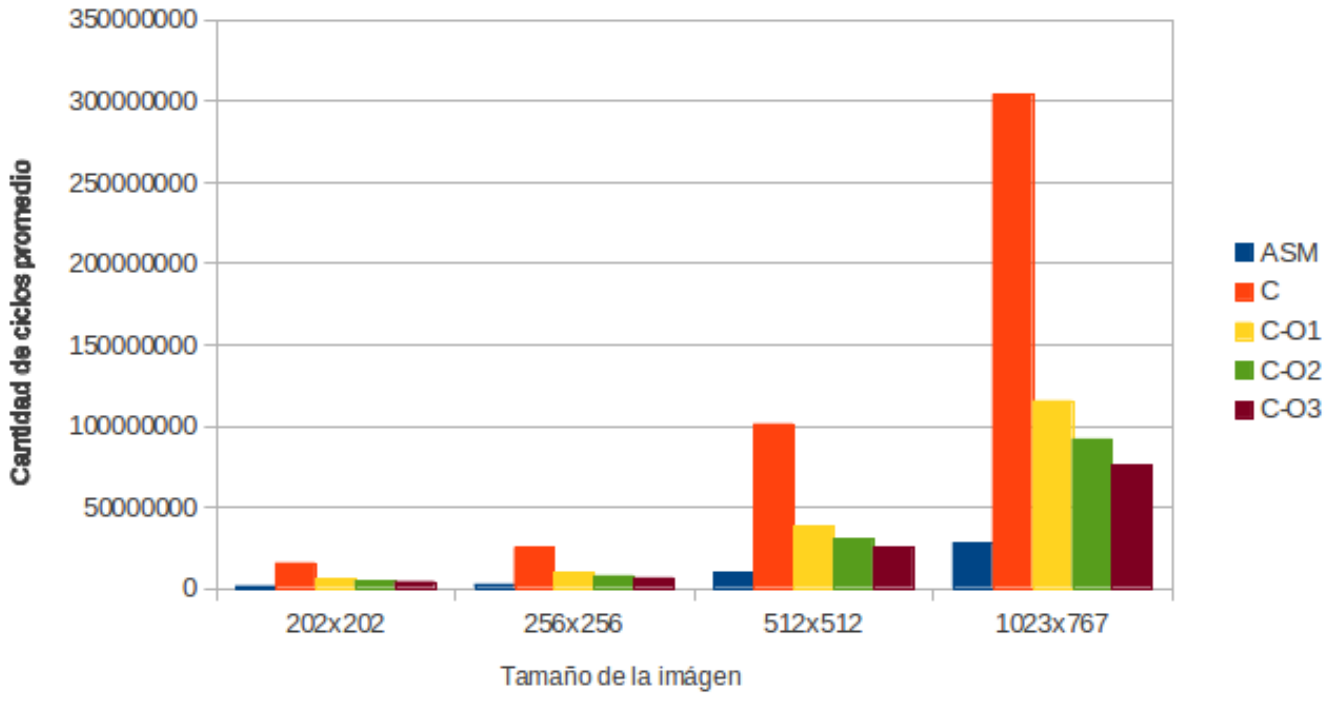
En este experimento se nos solicitó realizar mediciones temporales de la aplicación del filtro ldr en distintas implementaciones y con distintas opciones de optimización. Para intentar que el experimento fuera lo más representativo posible utilizamos distintas variables de Alpha a la hora de hacer los llamados al filtro.
Dado la naturaleza de este experimento debemos primero aclarar las especificaciones de la terminal donde los realizamos:

```
CPU: Intel(R) Core(TM) i5-3330 CPU @ 3.00GHz
Number of cores: 4
Arquitectura: 64 Bit
Memoria RAM: 8 GB

Maquina número 13 del laboratorio 4, Departamento de Computación, UBA.
```

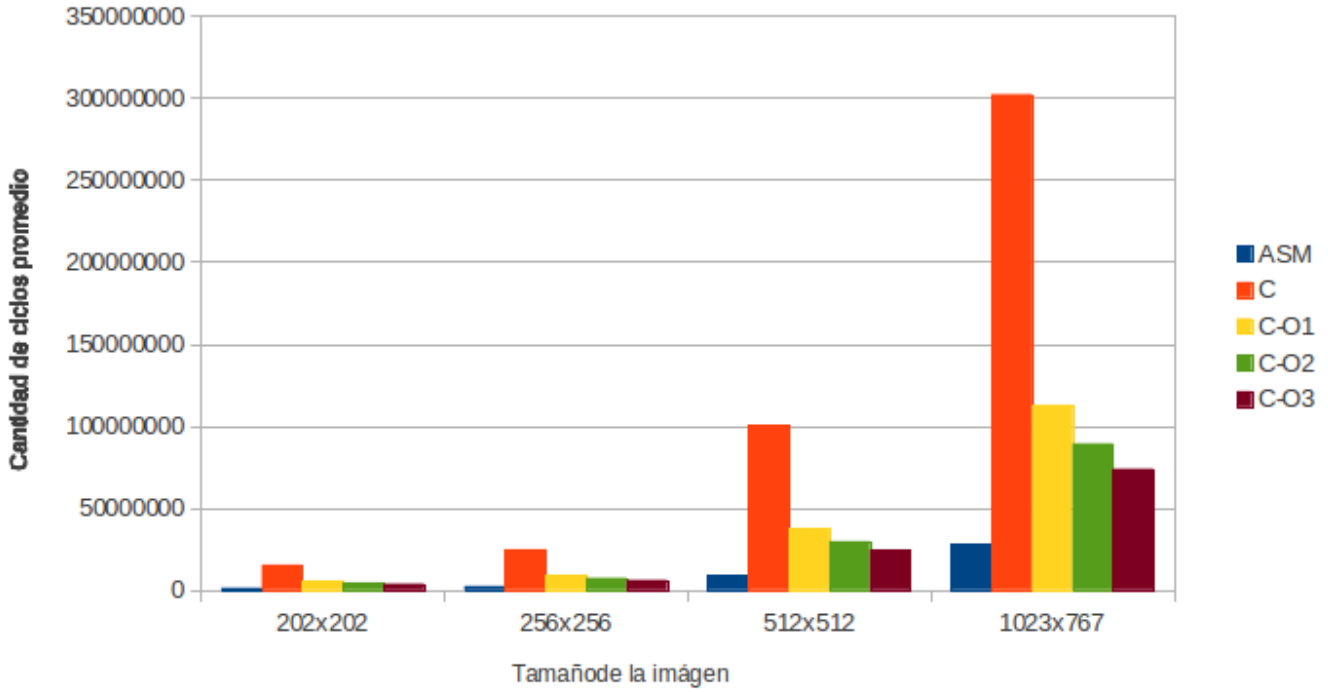
Para eliminar la posibilidad de que outliers modifiquen el resultado del experimento realizamos siempre 1000 ejecuciones de cada implementación, medimos el tiempo total y tomamos un promedio. La medición en ningún momento toma en cuenta los tiempos de apertura y cierre de las imágenes.
Se puede asumir que utilizamos la misma terminal y el mismo método de medición en todos los experimentos de este filtro.

Resultados :

Alpha = 0

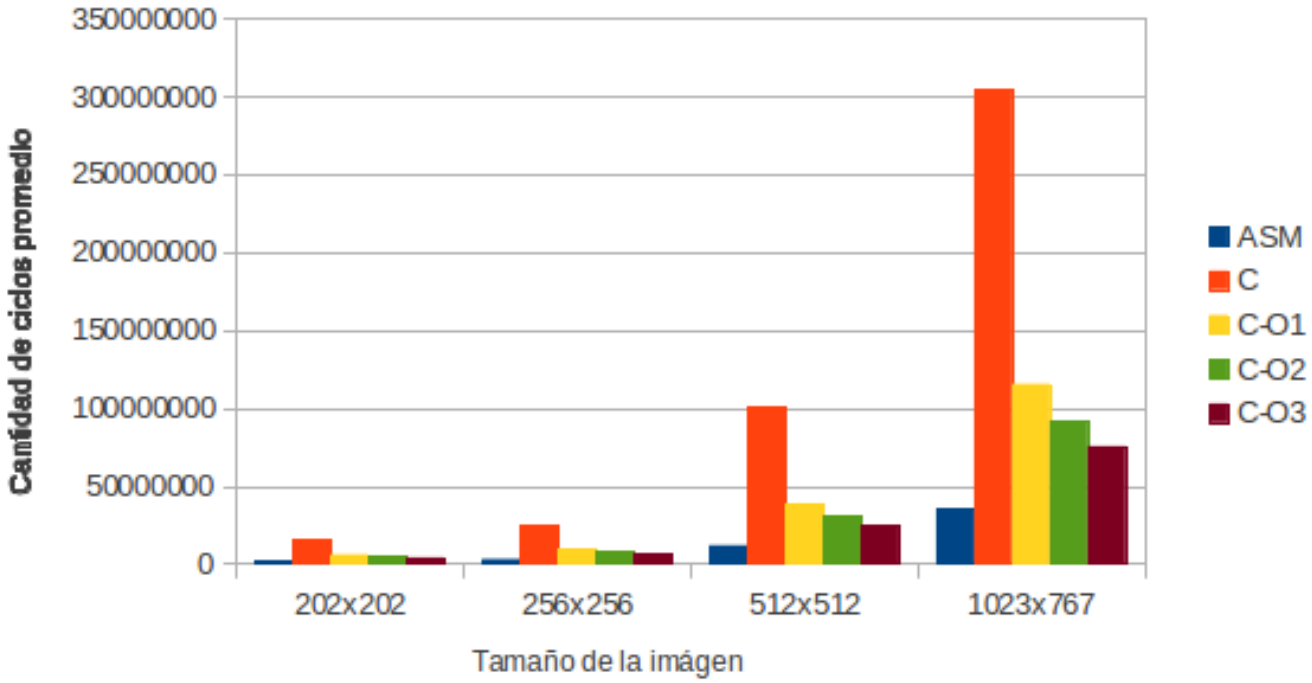


Diferencia de tiempo en las distintas implementaciones de ldr con alpha 0 sobre la imágen lena.bmp

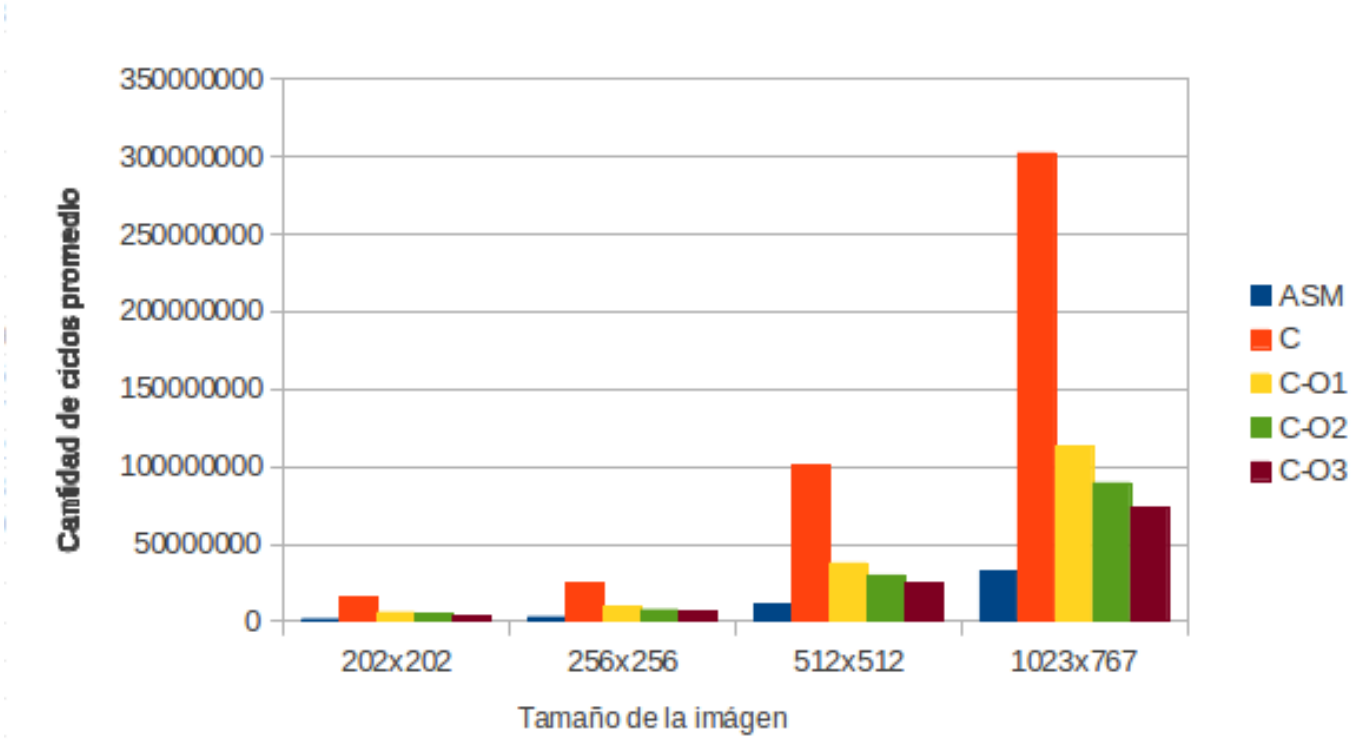


Diferencia de tiempo en las distintas implementaciones de ldr con alpha 0 sobre la imágen marilyn.bmp

Alpha = -150

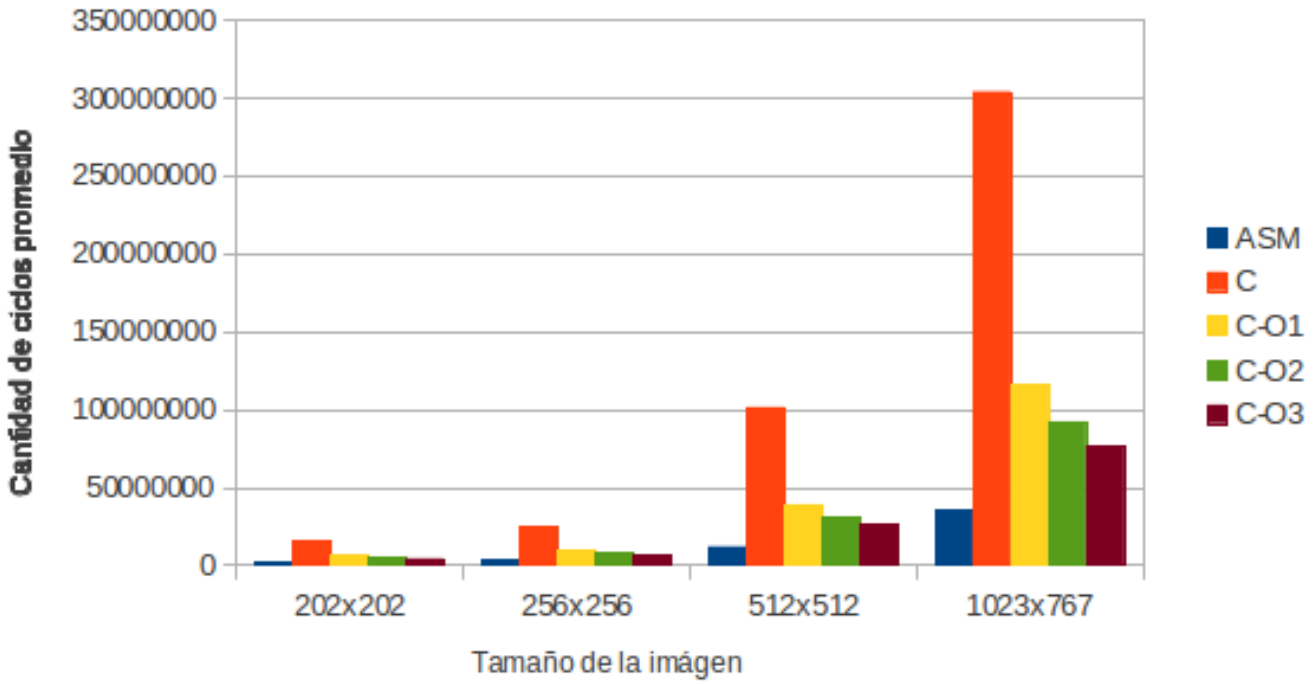


Diferencia de tiempo en las distintas implementaciones de ldr con alpha -150 sobre la imágen lena.bmp

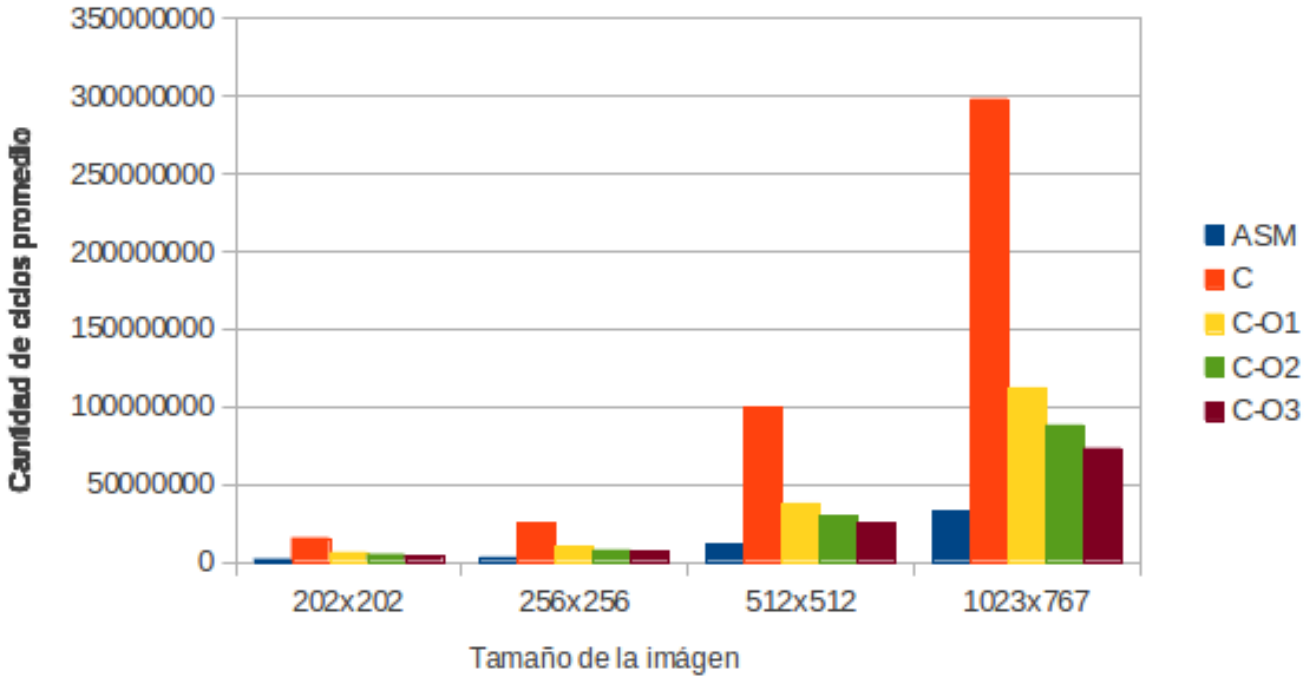


Diferencia de tiempo en las distintas implementaciones de ldr con alpha -150 sobre la imágen marilyn.bmp

Alpha = 200



Diferencia de tiempo en las distintas implementaciones de ldr con alpha 200 sobre la imágen lena.bmp



Diferencia de tiempo en las distintas implementaciones de ldr con alpha 200 sobre la imágen marilyn.bmp

Conclusiones :
Este filtro es particular por la gran cantidad de pixels que son necesarios tratar para obtener el resultado final. Se necesitan un total de 25 pixeles para obtener el resultado de un solo pixel a escribir. Aquí el hecho de obtener varios bytes de información con un sólo acceso a memoria es lo que marca la diferencia, pues son estos accesos los que más influencia tienen sobre los tiempos totales de cómputo.
Por más que el compilador optimize el rendimiento del algoritmo en *C*, nunca va a poder obtener el mismo resultado que el algoritmo *SIMD*. Ya que a pesar de que nuestro algoritmo realiza 10 accesos a memoria, conversiones a presición doble y divisiones en presición doble,el algoritmo en *C* también tiene que realizar dichas conversión y cálculos, e inclusive realizar más accesos a memoria por pixel procesado.
Es por ésto que aquí se nota de manera contundente la ventaja del modelo de programación *SIMD*, no sólo por su menor cantidad de accesos a memoria si no por la paralelización de sus calculos.

5. Conclusion Final

Este trabajo nos introdujo en los usos y ventajas del modelo de SIMD. A lo largo del mismo pudimos ver la incidencia de la paralelización en relación al código en C con diversas optimizaciones y en casi todos los casos los tiempos del código SIMD son mejores.
Consideramos que es una herramienta muy útil, que no solo aprovecha tiempo de cómputo al paralelizar datos, si no que también ayuda a reducir el cuello de botella que implica ir a buscar datos en memoria La idea del procesamiento de imágenes utilizando SIMD no solo la consideramos excelente sino que no se nos ocurren otras maneras de mejorarla, con excepción tal vez de incluir a la par técnicas de prefetching como la que utilizamos en el experimento 3 del filtro popart. Bajo nuestro punto de vista se podría trabajar sobre la combinación de estas técnicas para obtener resultados óptimos. Por otro lado creemos que sería interesante investigar a futuro sobre las posibilidades de trabajar con librerias de C que permitan el uso de modelos SIMD.

6. Anexo

6.1. Tiles

Experimento 2 Los siguientes son los flags del compilador que se activan al compilar con **-O1**

-fauto-inc-dec
-fcompare-elim
-fcprop-registers
-fdce
-fdefer-pop
-fdelayed-branch
-fdse
-fguess-branch-probability
-fif-conversion2
-fif-conversion
-fipa-pure-const
-fipa-profile
-fipa-reference
-fmerge-constants
-fsplit-wide-types
-ftree-bit-ccp
-ftree-builtin-call-dce
-ftree-ccp
-fssa-phiopt
-ftree-ch
-ftree-copyrename
-ftree-dce
-ftree-dominator-opts
-ftree-dse
-ftree-forwprop
-ftree-fre
-ftree-phiprop
-ftree-slsr
-ftree-sra
-ftree-pta
-ftree-ter
-funit-at-a-time

Fuente: https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

Tablas de gráficos

Experimento 3

Lena				
Implementacion	202x202	256x256	512x512	1023x767
ASM	75388	123997	428379	1277280
C	1192504	1867935	7284579	20857194
C -O1	482174	766752	2983577	8887534
C -O2	480184	757947	2974592	8928209
C -O3	471276	752455	3049398	8884119

Marylin					
Implementacion	202x202	256x256	513x513	711x768	1023x767
ASM	83500	104020	436491	873202	1259169
C	1186442	1887907	7255042	14566845	20693776
C -O1	479879	766641	3002008	6203812	8891075
C -O2	483785	765281	2988575	6234333	8922553
C -O3	489666	759208	2963692	6181656	8860856

El gráfico presenta la cantidad de ciclos de clock usados por cada implementación en cada uno de los foratos de imágen presentado

Experimento 4

Lena				
Implementacion	202x202	256x256	512x512	1023x767
ASM	75388	123997	428379	1277280
ASM con accesos a memoria	180294	327070	1312274	3306227
ASM con procesamiento extra	146648	237734	965665	2790585

Marylin					
Implementacion	202x202	256x256	513x513	711x768	1023x767
ASM	83500	104020	436491	873202	1259169
ASM con accesos a memoria	198442	332243	1085000	2432667	3222566
ASM con procesamiento extra	147406	235683	967057	1987867	2815649

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación de ASM según cada uno de los formatos de imágen presentado

6.2. Popart

Tablas de gráficos

Experimento 1

Lena	202x202	256x256	512x512	1023x767
C con saltos	1143087	1812791	6988284	18628804
C con 1 saltos menos	1156697	1777060	6896489	18451978
C con 3 saltos menos	945268	1492647	5658926	16099632
C sin saltos	777720	1241648	4985094	14424474

Lena	202x202	256x256	512x512	1023x767
C -O1 con saltos	406957	629364	2187798	5735027
C -O1 con 1 saltos menos	399478	569699	2028426	5313200
C -O1 con 3 saltos menos	275833	423195	1544985	4073329
C -O1 sin saltos	144555	236581	953089	2892494

Lena	202x202	256x256	512x512	1023x767
C -O2 con saltos	401139	626439	2324926	5279355
C -O2 con 1 saltos menos	377954	590801	2217774	5102172
C -O2 con 3 saltos menos	275833	423195	1544985	4073329
C -O2 sin saltos	167597	234406	951575	2897037

Lena	202x202	256x256	512x512	1023x767
C -O3 con saltos	396281	609264	2354951	5393349
C -O3 con 1 saltos menos	371553	586811	2195694	5068043
C -O3 con 3 saltos menos	283198	421029	1538136	4059907
C -O3 sin saltos	145082	230806	952991	2811187

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación de C según cada uno de los formatos de imágen presentado en base a la cantidad de saltos condicionales. Estás tablas reflejan el procesamiento de lena.bmp

Marilyn	202x202	256x256	512x512	1023x767
C con saltos	1063462	1671673	6281355	18078820
C con 1 saltos menos	931952	1495750	5754964	16788354
C con 3 saltos menos	847142	1335499	5262007	15674181
C sin saltos	790822	1203297	4786338	14326621

Marilyn	202x202	256x256	512x512	1023x767
C -O1 con saltos	393319	624893	2212643	5755820
C -O1 con 1 saltos menos	321837	507696	1785118	4886206
C -O1 con 3 saltos menos	243467	368318	1343582	3931394
C -O1 sin saltos	150819	247938	945751	2793338

Marilyn	202x202	256x256	512x512	1023x767
C -O2 con saltos	330812	501740	1813578	4904778
C -O2 con 1 saltos menos	277347	425904	1465169	4089876
C -O2 con 3 saltos menos	204139	322578	1197079	3475575
C -O2 sin saltos	166631	266256	939620	2786044

Marilyn	202x202	256x256	512x512	1023x767
C -O3 con saltos	337376	507069	1792359	4797980
C -O3 con 1 saltos menos	272733	413253	1469620	4074090
C -O3 con 3 saltos menos	200897	325632	1223724	3522743
C -O3 sin saltos	169995	244202	940377	2782180

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación de C según cada uno de los formatos de imágen presentado en base a la cantidad de saltos condicionales. Estás tablas reflejan el procesamiento de lena.bmp

Experimento 2

Lena	202x202	256x256	512x512	1023x767
ASM	155239	324289	1275782	3796299
ASM con accesos a memoria	470169	723553	2841649	8447304
ASM con procesamiento extra	267291	627531	2462221	7329529

Marilyn	202x202	256x256	512x512	1023x767
ASM	179798	369976	1422257	4228925
ASM con accesos a memoria	477135	722095	2836583	8443482
ASM con procesamiento extra	256014	628219	2479804	7328600

Estas tablas presentan la cantidad de ciclos de clock utilizadas por cada implementación de ASM dependiendo si se le agregó o no código innecesario y que tipo de instrucciones contenía ese código.

Experimento 3

Lena	200x200	256x256	512x512	1023x767
ASM	154788	368108	1462234	3796299
ASM con prefetch	174252	256910	1028769	3054427

Marilyn	200x200	256x256	512x512	1023x767
ASM	156407	368088	1458196	4313031
ASM con prefetch	170476	259418	1018766	2994057

Estas tablas presentan la cantidad de ciclos de clocks utilizadas en la implementación de ASM con y sin prefetch sobre las imágenes lena.bmp y marilyn.bmp

Experimento 4

Lena	202x202	256x256	512x512	1023x767
ASM	155239	324289	1275782	3796299
C	1143087	1812791	6988284	18628804
C -O1	406957	629364	2187798	5735027
C -O2	401139	626439	2324926	5279355
C -O3	396281	609264	2354951	5393349

Marilyn	202x202	256x256	512x512	1023x767
ASM	179798	369976	1422257	4228925
C	1063462	1671673	6281355	18078820
C -O1	393319	624893	2212643	5755820
C -O2	330812	501740	1813578	4904778
C -O3	337376	507069	1792359	4797980

Las tablas presentas la cantidad de ciclos de clock promedio usados por cada implementación en cada uno de los fomratos de imágen presentado

Otros

Experimento 3 Código utilizado para el *prefetch* (lineas 111 a 128)

```
movdqu xmm13,[rdi]
    .ciclo:
        cmp EBX, EAX
        je .fin
        cmp ECX, R12D
        je .procesamiento_fin_fila


        movdqu xmm0, xmm13
        add RDI, 15
        movdqu xmm13,[rdi]
```

Notese que debimos liberar el registro xmm13, pero a costa de hacer un acceso a memoria extra dentro del ciclo principal.

6.3. Temperature

Tablas de gráficos

Experimento 1

Lena	202x202	256x256	512x512	1023x767
ASM	679200	1319942	5162642	15391566
C	2020095	3226367	12765367	36331100
C -O1	1113909	1768372	6878336	19234646
C -O2	380776	617748	2369219	5946792
C -O3	399295	624962	2398419	5965785

Marilyn	202x202	256x256	512x512	1023x767
ASM	669418	1293922	5103069	15240908
C	1965552	3144202	12238667	35925600
C -O1	1081041	1701790	6547066	19064632
C -O2	406940	610597	2305452	6336834
C -O3	409459	637775	2328194	6355376

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación del filtro Temperature sobre las imágenes de lena y marilyn

6.4. Low dynamic range

Tablas de gráficos

Experimento 1

Alpha = 0

Lena	202x202	256x256	512x512	1023x767
ASM	1166080	2288388	9225660	27880012
C	15379671	24894818	101003544	304141280
C-O1	5843192	9438724	38281320	115282784
C-O2	4648209	7520207	30472004	91746288
C-O3	3825440	6185683	25067610	75512952

Marilyn	202x202	256x256	512x512	1023x767
ASM	1165523	2287458	9234665	27882244
C	15289225	24751692	100334872	302060032
C-O1	5744034	9269425	37497460	112768400
C-O2	4537903	7332371	29671628	89209672
C-O3	3772228	6065842	24511930	73752104

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación del filtro LDR sobre las imágenes lena.bmp y marilyn.bmp con alpha valiendo 0

Alpha = -150

Lena	202x202	256x256	512x512	1023x767
ASM	1548220	2882468	11638900	35188468
C	15376594	24896076	101013296	304106432
C-O1	5831326	9455642	38256864	115186624
C-O2	4626566	7534527	30469736	91791896
C-O3	3834070	6185983	25027606	75350752

Marilyn	202x202	256x256	512x512	1023x767
ASM	1421891	2666284	10759818	32476810
C	15285646	24727216	100322696	301949056
C-O1	5724670	9250726	37441532	112647136
C-O2	4512518	7330982	29640704	89125048
C-O3	3750818	6068686	24448104	73528392

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación del filtro LDR sobre las imágenes lena.bmp y marilyn.bmp con alpha valiendo -150

Alpha = 200

Lena	202x202	256x256	512x512	1023x767
ASM	1537558	2886052	11667044	35169600
C	15422698	24928888	101128624	303739808
C-O1	5940403	9572697	38727876	115623168
C-O2	4701071	7616287	30773550	91889432
C-O3	3935289	6387645	25741110	76330760

Marilyn	202x202	256x256	512x512	1023x767
ASM	1395859	2673072	10767060	32466370
C	15167054	24500116	99136704	297859392
C-O1	5792324	9322972	37417492	111920856
C-O2	4552404	7303384	29371708	87792528
C-O3	3833282	6163086	24458236	72782104

Estas tablas presentan la cantidad de ciclos de clock usados por cada implementación del filtro LDR sobre las imágenes lena.bmp y marilyn.bmp con alpha valiendo 200

7. Forma de medición de tiempos

Considerando la importancia de los métodos de medición en un trabajo que busca analizar tiempos nos pareció pertinente aclarar la serie de pasos que realizamos para realizar las mediciones, así como también que archivos debimos modificar.

- **benchmarking.sh** Este script simplemente corre el tp con el filtro y la cantidad de iteraciones pasadas como parámetro
- **tp2.h** A la estructura de configuración (*configuracion_t*) se le agregó un componente iteraciones
- **tp2.c** Se le agregó un ciclo for que aplica el filtro deseado tantas veces como indique el parámetro *iteraciones*. Se mide el tiempo antes y después de terminar las aplicaciones. Se debe notar que los llamados a las funciones que abren y cierran las imágenes quedan por fuera de la medición.
- **cli.c** Se le agregó como opción por defecto *config->iteraciones = 1;*

Con las modificaciones anteriores podíamos correr el comando *./benchmarking.sh* ***filtro iteraciones*** *>> ejemplo.txt* que creaba un archivo *ejemplo.txt* y en el imprimía los datos de imágen corrida, implementación utilizada, tiempo de comienzo, tiempo total, cantidad de iteraciones y promedio de cada aplicación del filtro.
Ese ultimo promedio es el que tomamos para realizar los gráficos y los análisis de este trabajos.
En la entrega final de los archivos hacemos entrega de los archivos sin modificar.