DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Procesamiento de imágenes/SIMD/Lo de Charly/Bondiola

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Federico De Rocco	403/13	fede.183@hotmail.com
Fernando Abelini	544/09	ferabelini@outlook.com
Manuel Casanova	355/05	casanova_manuel@yahoo.com

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

Índice

1. Introducción

En este trabajo práctico se pide realizar un conjunto de filtros para imágenes en dos lenguajes de programación, C y Assembler. Este último debe hacerse considerando la paralelización de los algoritmos para lograr que se puedan procesar dos píxeles o más en un solo ciclo. Para esto se deberán utilizar las instrucciones SSE. También se pide analizar la performance de los filtros comparando las versiones de C y de Assembler. En el siguiente informe se harán descripciones de los filtros pedidos y se dará un análisis de como se implementaron en los ya mencionados lenguajes. Se incluirá, por cada uno, una sección experimentos en la cual se realizaran las pruebas solicitadas en el enunciado.

2. Cropflip

2.1. Descripción

Este filtro básicamente consiste en tomar la imagen fuente, recortarla y girarla . Para esto el programa comprende como datos de entrada: src la matriz entrada(con la imagen original), dst la matriz destino, cols que contiene la cantidad de columnas, filas la cantidad de filas, src_row_size que tiene el número de columnas que se debe desplazar de un elemento de la matriz para posicionarse justo en el elemento de abajo(simplemente tiene el ancho de una fila), dst_row_size ídem anterior pero de la matriz destino, tamx la cantidad de columnas en píxeles a recortar, tamy la cantidad de filas en píxeles a recortar, offsetx la columna, en píxeles, a partir de la cual debe comenzar a recortarse, offsetxy la fila, en píxeles, a partir de la cual debe comenzar a recortarse. Lo siguiente es el pseudocódigo de la versión C:

for $i = 0...tamy - 1$ **do**
  for $j = 0...tamx - 1$ **do**
    $dst\_matriz_{i,j,rojo} = src\_matriz_{tamy+offsety-i-1,offsetx+j,rojo}$
    $dst\_matriz_{i,j,verde} = src\_matriz_{tamy+offsety-i-1,offsetx+j,verde}$
    $dst\_matriz_{i,j,azul} = src\_matriz_{tamy+offsety-i-1,offsetx+j,azul}$
  end
end

Algorithm 1: Algoritmo de Cropflip en lenguaje C

Como explicación a este pseudocódigo hay que agregar unos cuantos detalles. El i y el j hacen mención al desplazamiento en la matriz en píxeles. Esta es la razón por la cual aparece el 4* todo el tiempo junto al índice j. Ya que a la hora de modificar algo tenemos que hacerlo mediante los datos de los píxeles los cuales son r, g, b y a. En las columnas avanzamos de a 4 para poder siempre estar posicionados en el principio de un píxel. También al principio se pasa src y dst, que originalmente son un vector, a matrices para facilitar la explicación. La versión en assembler de este algoritmo es la siguiente:

for $i = 0...tamy - 1$ **do**
  for $j = 0...tamx - 1$ **do**
    $Pixel1 = src[src\_row\_size * (tamy + offsety - i - 1) + offsetx + j]$
    $Pixel2 = src[src\_row\_size * (tamy + offsety - i - 1) + offsetx + j + 1]$
    $Pixel3 = src[src\_row\_size * (tamy + offsety - i - 1) + offsetx + j + 2]$
    $Pixel4 = src[src\_row\_size * (tamy + offsety - i - 1) + offsetx + j + 3]$
    $xmm0 = [Pixel1|Pixel2|Pixel3|Pixel4]$
    $dst[(i * dst\_row\_size) + j] = Pixel1$
    $dst[(i * dst\_row\_size) + j + 1] = Pixel2$
    $dst[(i * dst\_row\_size) + j + 2] = Pixel3$
    $dst[(i * dst\_row\_size) + j + 3] = Pixel4$
  end
end

Algorithm 2: Algoritmo de Cropflip en lenguaje ensamblador

Como ya no se nos facilita tanto tomar el vector con la imagen y verlo como una matriz lo que

hacemos es crearnos un i y j e ir recorriendo la matriz con ellos como si fueran índices y hacer las operaciones adecuadas utilizándolos. Como se trata de vectores, y más concretamente posiciones de memoria de registros, estamos ante el problema de pensar en una forma de referirnos a una posición de una estructura tipo matriz pero siendo el elemento real un simple vector de pixeles. Cuando hacemos esta cuenta:$src\_row\_size * (tamy + offsety - i - 1) + offsetx + j$ notamos que el índice de fila esta multiplicado por src_row_size, que es el tamaño de una fila en la matriz origen, y se le suma $offsetx + j$ que es el índice columna. Nos damos cuenta que, cuando la fila sea cero, esto no afectara y sencillamente recorrerá la primera fila de columna a columna. Cuando sea distinto a cero le sumara a la posición de la columna el valor que lo ubicará en el principio de la fila que denota el índice correspondiente a esta. Además como se pide paralelizar en assembler, se puede ver que en el algoritmo se pasan 4 píxeles en un solo ciclo al registro xmm0. Después se lo escribe de una sola pasada en memoria. Con esto se logra en un ciclo hacer 4 veces la operación que en el de C solo se realizaba una vez. El i avanza de a uno pero el j avanza de a 4 por las pretensiones de paralelización expuestas anteriormente.

2.2. Experimentos

2.3. Experimento 1.1

a) Las siguientes son las secciones, que no son del programa, y para qué creemos que funciona cada una: secciones .debug*: Es información para que utilicen los debuggers. Funciones que necesitan los debuggers para realizar ”debugueo simbólico”.

sección .comment: Esta sección contiene informacion que describe un objeto, pero que no se requiere para la ejecución del mismo. Sirve para almacenar información que no es parte del código y no se carga en memoria durante la ejecución.

sección eh_frame: Esta sección contiene información necesaria para desentrañar el frame durante el manejo de excepciones

b) Pone todos los parámetros en la pila. Coloca en la pila las variables locales y cada vez que las necesita modificar, las modifica en la porción de memoria.

c) Claramente no es necesario poner tantas cosas a la pila, de esa manera se evitan acceso a memoria innecesarios. Las comparaciones de los índices de los ”for”, se hacen en una sección apartada de código, y cada vez que se desea ver si un ciclo debe seguir o terminar se realizan 2 saltos. En cada iteración se calcula la posición de memoria a ser copiada. Eso se podría evitar calculando previamente los saltos, siempre y cuando, estos sean siempre iguales.

2.4. Experimento 1.2

a) La primera optimización que se observa es que no tira los parámetros a la pila y eso le ahorra accesos a memoria al principio, mejor velocidad. También genera menos cantidad de código para la función. El orden del código es mejor(no hace dos saltos por cada ciclo).

b) Estos son los distintos niveles de optimización que ofrece GCC:

-O0: Este nivel (la letra .ºseguida de un cero) desconecta por completo la optimización y es el predeterminado si no se especifica ningún nivel -O en CFLAGS o CXXFLAGS. El código no se optimizará. Esto, normalmente, no es lo que se desea.

-O1: Este es el nivel de optimización más básico. El compilador intentará producir un código rápido y pequeño sin tomar mucho tiempo de compilación. Es bastante básico, pero conseguirá realizar correctamente el trabajo.

-O2: Un paso delante de -O1. Este es el nivel recomendado de optimización, a no ser que tenga necesidades especiales. -O2 activará algunas opciones añadidas a las que se activan con -O1. Con -O2, el compilador intentará aumentar el rendimiento del código sin comprometer el tamaño y sin tomar mucho más tiempo de compilación.

-O3: Este es el nivel más alto de optimización posible. Activa opitimizaciones que son caras en términos de tiempo de compilación y uso de memoria. El hecho de compilar con -O3 no garantiza una forma de mejorar el rendimiento y de hecho, en muchos casos puede ralentizar un sistema debido al uso de binarios de gran tamaño y mucho uso de la memoria. También se sabe que -O3 puede romper algunos paquetes. Por ello no se recomienda usar -O3.

-Os: Este nivel optimizará su código para reducir el tamaño. Activa todas las opciones de -O2 que no aumenten el tamaño del código generado. Es útil para máquinas con capacidad limitada de disco o con CPUs con poca caché.

-Og: En GCC 4.8 aparece un nuevo nivel del optimización general: -Og. Trata de solucionar la necesidad de realizar compilaciones más rápidas y obtener una experiencia superior en la depuración a la vez que ofrece un nivel razonable de rendimiento en la ejecución. La experiencia global en el desarrollo debería ser mejor que para el nivel de optimización -O0. Observe que -Og no implica -g, éste simplemente deshabilita optimizaciones que podrían interferir con la depuración.

-Ofast: Nuevo en GCC 4.7. Consiste en el ajuste -O3 más las opciones -ffast-math, -fno-protect-parens y -fstack-arrays. Esta opción rompe el cumplimiento de estándares estrictos y no se recomienda su utilización.

c) **-pipe:** Hace que el proceso de compilación sea más rápido. Indica al compilador que use tuberías en lugar de archivos temporales durante los diferentes estados de compilación, lo cual utiliza más memoria. **-msse, -msse2, -msse3, -mmmx, -m3dnow:** Estas opciones activa instrucciones SSE, SSE2, SSE3, MMX y 3DNow para arquitecturas x86-64.

-fomit-frame-pointer: Esta es una opción muy común diseñada para reducir el tamaño del código generado. Está activada para todos los niveles de -O (excepto -O0) en arquitecturas donde no interfiera con la depuración (como x86-64), pero puede que haga falta activarla añadiéndola a sus opciones. Aunque el manual de GNU GCC no especifica todas las arquitecturas en las que está activada al usar -O, hay que activarla explícitamente en un x86. Sin embargo, al usar esta opción la depuración se convierta en difícil o incluso imposible.

En particular, provoca que la localización de problemas en aplicaciones escritas en Java sea mucho más complicada, aunque Java no es el único código afectado al usar esta opción. Así, aunque esta opción puede ayudar, la depuración será complicada. En particular, las trazas de ejecución (backtraces) no servirán de mucho. Sin embargo, si no planea hacer muchas depuraciones y no tiene añadida ninguna otra CFLAG relacionada con la depuración como -ggdb (y no está instalando paquetes con la variable USE debug), entonces intente usar -fomit-frame-pointer.

2.5. Experimento 1.3

a) Tiempo de 10 cropflips (medidos en ticks de clock):

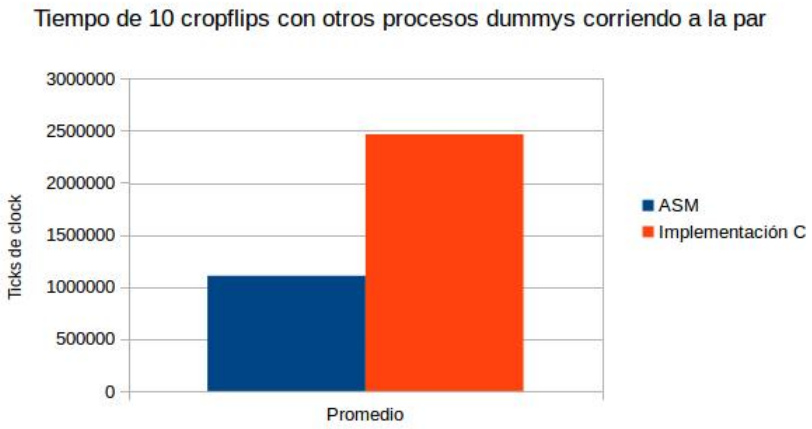
Implementación ASM	Implementación C
136442	689230
136492	709210
137432	711922
139064	716366
139164	716742
140314	719720
144945	757579
145538	882450
181304	974696
190728	1000265

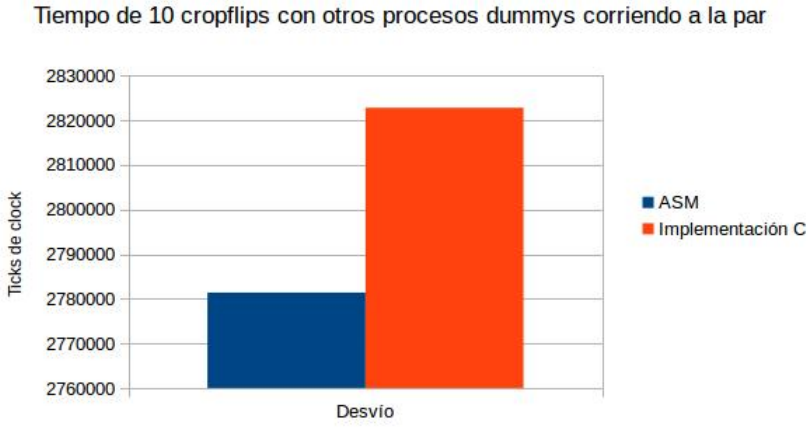
	Implementación ASM	Implementación C
Promedio	149142,3	787818
Desvío	19809,902	118500,007

b) Tiempo de 10 cropflips con otros procesos dummys corriendo a la par(medidos en ticks de clock):

Implementación asm	Implementación C
220151	1524515
221788	1528350
221867	1528868
225978	1546967
225984	1548969
227241	1553093
227273	1559291
229922	1642916
238021	1727886
9021929	10497577

	Implementación ASM	Implementación C
Promedio	1106015,4	2465843,2
Desvío	2781373,123	2822792,042





En los datos obtenidos se observa la existencia de números que se alejan mucho del promedio de tiempo de ejecución. Estos números no son consecuencia de la implementación, ya que ni la maquina donde se obtuvieron los datos, ni la imagen a ser procesada fueron cambiados. Esto se debe al funcionamiento del procesador cuando realiza un desalojo del programa y los ticks extras son el tiempo que tarda en desalojarlo y volverlo a cargar. Los números grandes generados por el funcionamiento del procesador, hacen que las conclusiones sobre las estadísticas que saquemos sean erróneas. De los datos obtenidos se puede concluir que:

- Los números grandes no son representativos de la realidad del funcionamiento de los filtros
- Cualquier conclusión sacada de los datos que contenga a estos números (outliers) se alejará mucho de la realidad.
- Los datos de las varianzas son números que aunque muestren información real, son números muy grandes para poder analizarlos, con lo que vamos a utilizar los desvíos estándar.

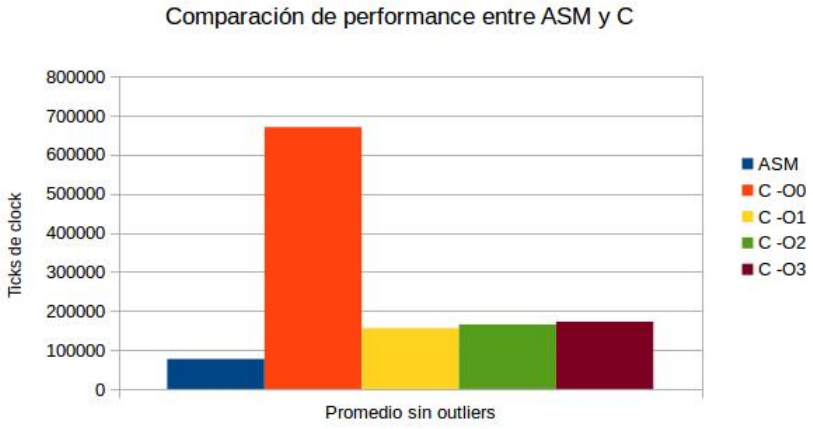
2.6. Experimento 1.4

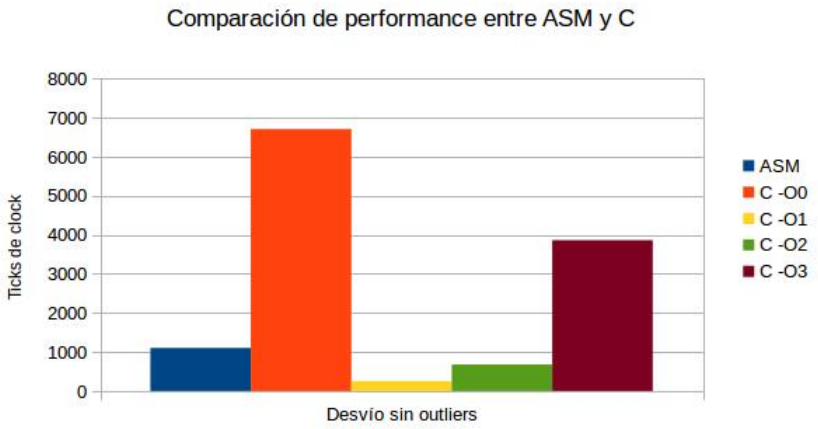
En la siguiente tabla se muestran los resultados de las versiones de ASM y C (con optimización), para comparar diferencia de performance. Los datos se encuentran ordenados de menor a mayor. Para luego calcular promedios y desvíos quitando outliers(consideramos menor y mayor valor)

ASM	C -O0 C	-O1	C -O2	C -O3
69148	663908	155642	164481	167122
75211	664376	155694	164501	167981
76054	664908	155701	164521	168318
76079	665380	155706	164655	169269
76241	665408	155771	165484	171927
76569	670776	155807	165644	172435
77480	674664	155991	165768	172949
78064	677812	156000	165860	173595
78377	681596	156437	166239	179969
103541	708340	223221	203650	192540

Calculemos ahora los promedios y desvíos estandards(sin outliers):

	ASM	C -O0 C	-O1	C -O2	C -O3
Promedio	76759.375	670615	155888.375	165334	172055.375
Desvío	1102.007	6706.207	253.696	678.080	3859.819





2.7. Experimento 1.5

En el siguiente experimento deseamos conocer cuál es el factor limitante de la implementación en ASM. Para esto vamos a comparar los resultados del algoritmo en ASM, contra el mismo, pero agregando operaciones aritméticas en un caso, y operaciones de acceso a memoria en otros. La idea es verificar si lo que limita a la función son los accesos a memoria o la intensidad de cómputo.

En el caso de operaciones aritméticas se agregan al ciclo cierta cantidad de ADD rax, rbx(4, 8, 16).

En el caso de accesos a memoria se agregar 2 operaciones que leen de la pila y 2 que escriben en la pila.

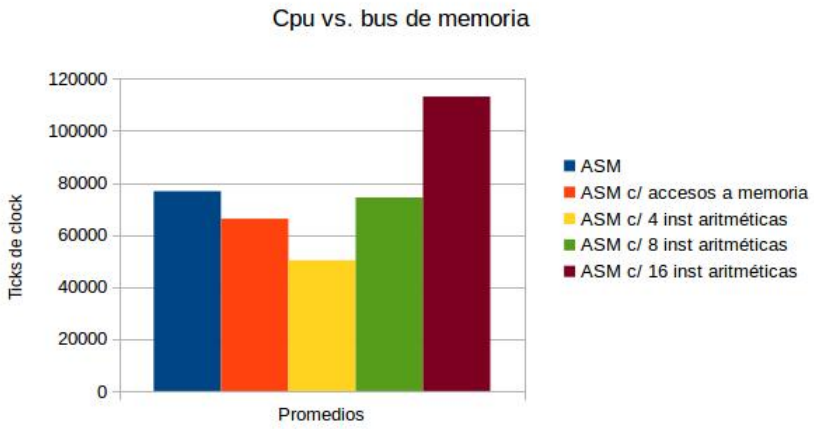
La siguiente tabla muestra los resultados de 10 muestras:

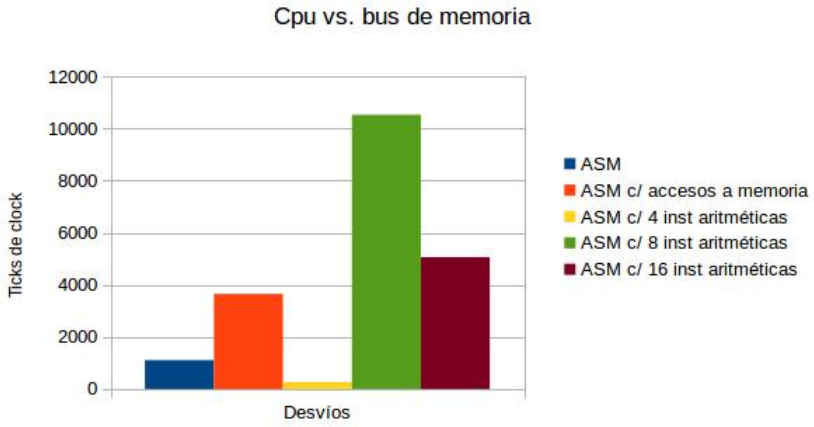
ASM	ASM c/ accesos a memoria	ASM c/ 4 inst aritméticas	ASM c/ 8 inst aritméticas	ASM c/ 16 inst aritméticas
69148	61281	49963	68062	105315
75211	62412	50006	68325	105355
76054	63893	50045	68869	109601
76079	64306	50143	68894	110420
76241	64945	50177	70784	111987
76569	65438	50189	71283	113378
77480	66135	50208	72280	115122
78064	67821	50287	74418	117049
78377	74275	50853	99932	121998
103541	172337	161096	155036	139228

La siguiente tabla muestra promedios y desvíos sin outliers:

	ASM	ASM c/ accesos a memoria	ASM c/ 4 inst aritméticas	ASM c/ 8 inst aritméticas	ASM c/ 16 inst aritméticas
Promedio	76759.375	66153.125	50238.5	74348.125	113113.75
Desvío	1102.007	3649.540	263.965	10535.389	5065.040

Como vemos en el gráfico del promedio, la implementación con accesos a memoria no significa diferencia comparada con la implementación sin modificar. En cambio podemos ver que al agregar 16 instrucciones aritméticas los tiempos de ejecución suben considerablemente.
Con lo cual podemos concluir que el limitante es la intensidad de cómputo.





3. Sierpinski

3.1. Descripción

El filtro de Sierpinski consiste en tomar la imagen fuente y aplicarle un efecto fractálico encima, simulando los triángulos de Sierpinki pero con cuadrados. Para esto, se recorre toda la matriz de píxeles, se toma la posición actual y se genera un coeficiente que se multiplica por cada color en ese lugar.

El siguiente es el algoritmo en C que implementamos para este ciclo:

for $i = 0...tamy - 1$ **do**
  for $j = 0...tamx - 1$ **do**
    $dst\_matriz_{i,j} = src\_matriz_{i,j,rojo} * coef(i,j)$
    $dst\_matriz_{i,j+1} = src\_matriz_{i,j,verde} * coef(i,j)$
    $dst\_matriz_{i,j+2} = src\_matriz_{i,j,azul} * coef(i,j)$
  end
end

Algorithm 3: Algoritmo de Sierpinki en lenguaje C

Este es el algoritmo para calcular el coeficiente en C:

$$coef(i,j) = (1/255{,}0) * ((255{,}0 * i/CantFilas) \oplus (255{,}0 * j/CantColumnas))$$

Esto, por otro lado, es el que usamos en assembler considerando que tenemos que garantizar una cierta paralelización de los datos:

for $i = 0...tamy - 1$ **do**
  for $j = 0...tamx - 1$ **do**
    $xmm1 = [Pixel4|Pixel3|Pixel2|Pixel1]$
    $UnpackdeByteaWord$
    $xmm1 = [Pixel2|Pixel1]$
    $xmm2 = [Pixel4|Pixel3]$
    $UnpackdeWordaDWord$;
    $xmm3 = [Pixel2]$
    $xmm1 = [Pixel1]$
    $xmm2 = [Pixel3]$
    $xmm4 = [Pixel4]$;

    $Coef = CalcularCoeficientes$
    $xmm0 = |coeficiente4|coeficiente3|cociente2|coeficiente1|$
    $broadcastdeloscoeficientesen4registrosparamultiplicaracadapixel$

    $xmm15 = |coeficiente1|coeficiente1|cociente1|coeficiente1|$
    $xmm14 = |coeficiente2|coeficiente2|cociente2|coeficiente2|$
    $xmm13 = |coeficiente3|coeficiente3|cociente3|coeficiente3|$
    $xmm12 = |coeficiente4|coeficiente4|cociente4|coeficiente4|$
    $ConvertimosAPresicionSimple(xmm1)$
    $ConvertimosAPresicionSimple(xmm2)$
    $ConvertimosAPresicionSimple(xmm3)$
    $ConvertimosAPresicionSimple(xmm4)$

    $xmm1 = xmm1 * xmm15$
    $xmm2 = xmm2 * xmm14$
    $xmm3 = xmm3 * xmm13$
    $xmm4 = xmm4 * xmm12$

    $ConvertimosAEnteros(xmm1)$
    $ConvertimosAEnteros(xmm2)$
    $ConvertimosAEnteros(xmm3)$
    $ConvertimosAEnteros(xmm4)$

    $PackDWordaWord$
    $xmm2 = [Pixel4, Pixel3]$
    $xmm1 = [Pixel2, Pixel1]$
    $PackWordaByte$
    $xmm1 = [Pixel4|Pixel3|Pixel2|Pixel1]$
    $return\ xmm1$
  end
end

Algorithm 4: Algoritmo de Sierpinki en lenguaje ensamblador

3.2. Experimentos

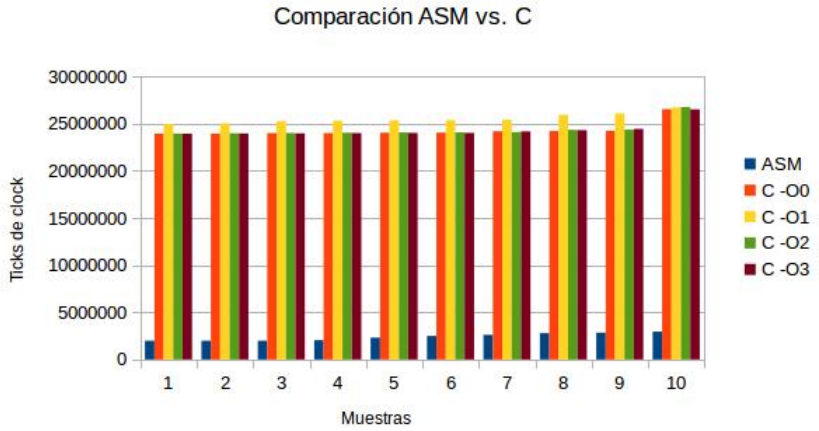
3.3. Experimento 2.1

Secuencial vs Vectorial

Se realizó la siguiente medición para comparar la performance de las versiones de C y ASM. Para esto se midió el tiempo de el algoritmo Sierpinski en ASM y en C compilado con las diferentes opciones de optimización(-O0 -O1 -O2 -O3)

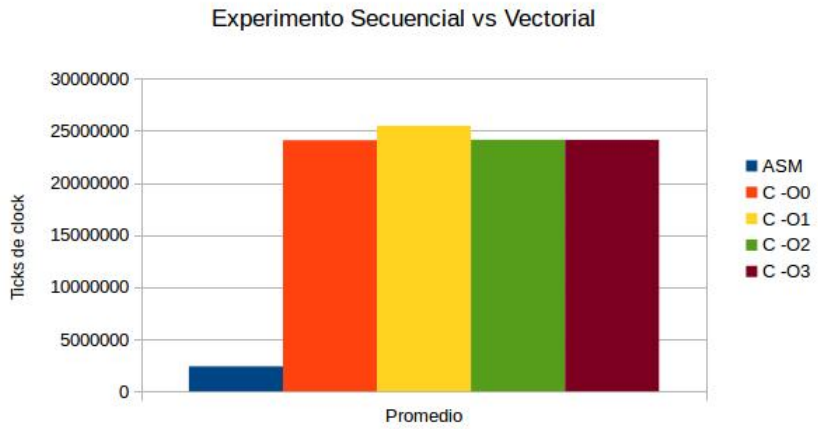
Se obtiene la siguiente tabla de resultados:

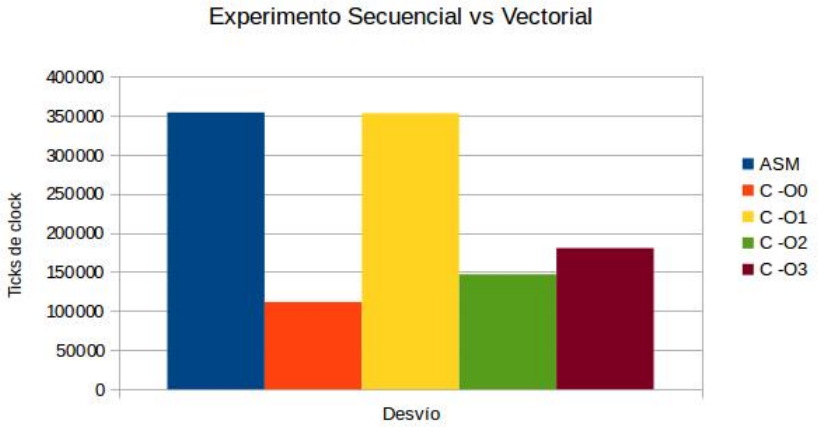
ASM	C -O0	C -O1	C -O2	C -O3
1979149	23950764	24978220	23956842	23945653
1979591	23950966	25046817	23984700	23961380
1983075	24002597	25246046	24016093	23971489
2046425	24003697	25316344	24039011	24006034
2311542	24045930	25349957	24069702	24019744
2493975	24056933	25373885	24077508	24022409
2605145	24203169	25411694	24105201	24180257
2792657	24216998	25936224	24347966	24314764
2838663	24233878	26082798	24363335	24446882
2948310	26538594	26733527	26753905	26513368



Se puede ver que los tiempos de ASM con respecto a C con cualquier nivel de optimización son superiores. Calculemos Promedios y desvíos estándar de los datos descartando outliers(consideramos outliers al menor y mayor valor)

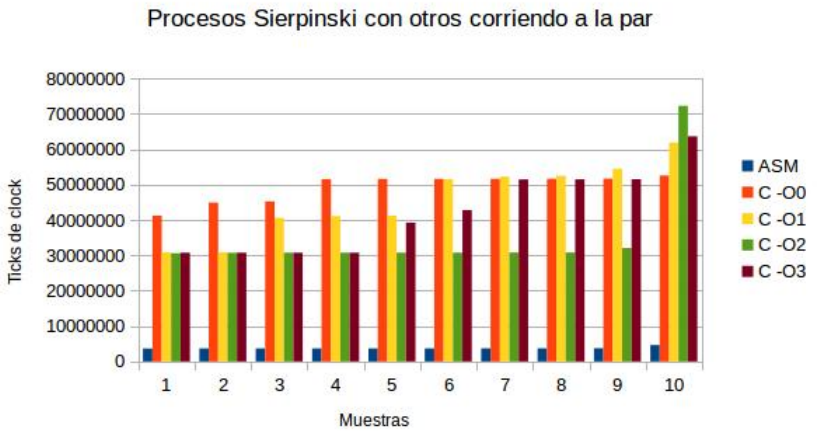
	ASM	C -O0	C -O1	C -O2	C -O3
Promedio	2381384.125	24089271	25470470.625	24125439.5	24115369.875
Desvío estándar	354190	111538	353104	146950	180484



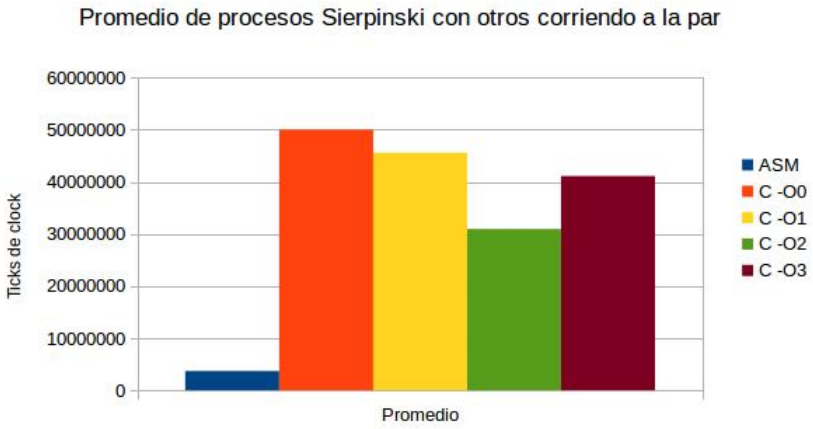


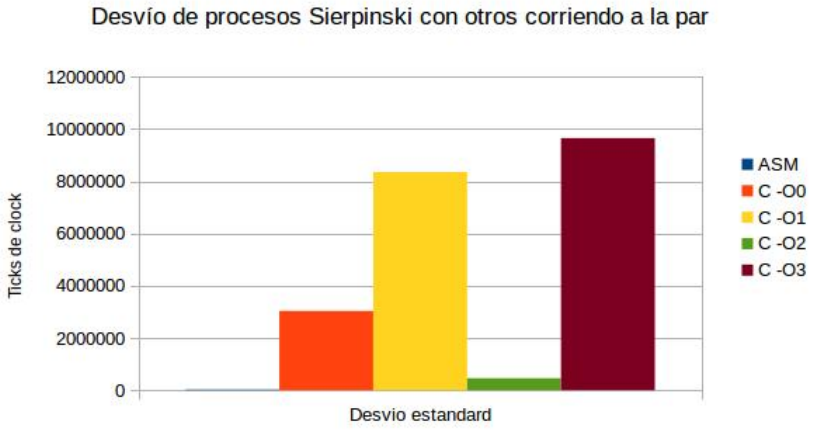
Verifiquemos ahora la performance del filtro con procesos corriendo a la par. Para esto ejecutamos un programa por cada core lógico que cicla infinitamente sumando 1 a una variable.

ASM	C -O0	C -O1	C -O2	C -O3
3670735	41244828	30835911	30596371	30743988
3672513	44918524	30861208	30738494	30755299
3673100	45245188	40615205	30757275	30756271
3677214	51538693	41101907	30764515	30759578
3684686	51626548	41226423	30772374	39228198
3685366	51634223	51560508	30772609	42790292
3700843	51645082	52257507	30801802	51483986
3713232	51652554	52499858	30818767	51504201
3736790	51704539	54523487	32069388	51533906
4624072	52573857	61887910	72290370	63653873



	ASM	C -O0	C -O1	C -O2	C -O3
Promedio	3692968	49995668.875	45580762.875	30936903	41101466.375
Desvío estándar	22617	3034470	8353997	458279	9652768





3.4. Experimento 2.2

CPU vs bus de memoria

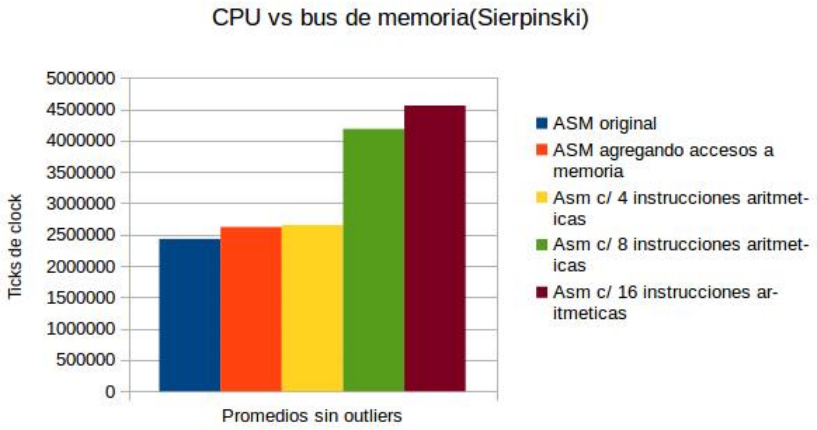
Realizamos experimento para verificar qué factor es el mayor limitante de la performance la versión ASM de Sierpinski. Para esto comparamos los tiempos del código original contra uno modificado, agregando instrucciones aritméticas o instrucciones que realicen accesos a memoria.
Se obtienen los siguientes resultados:

ASM original	ASM agregando accesos a memoria
2940985	2886240
2115091	2228486
2113430	2280717
2450323	2582231
2806890	2818483
2847948	2617010
2830241	3999342
2116241	2309797
2152194	2111518
2044160	2104327

ASM original	Asm c/ 4 instrucciones aritméticas	Asm c/ 8 instrucciones aritméticas	Asm c/ 16 instrucciones aritméticas
2940985	3382576	2829676	2378403
2115091	3219106	2839452	2448264
2113430	2184028	2840240	2452696
2450323	2195801	2842240	4327147
2806890	2419465	2855848	4507762
2847948	2932075	4444074	4841270
2830241	3396019	5630711	5592174
2116241	2831709	5967198	6113666
2152194	2030719	6050832	6165354
2044160	2268153	7446114	6189698

Calculamos promedios, descartando outliers.

	ASM	ASM c/ accesos a memoria	Asm c/ 4 instrucciones aritméticas	Asm c/ 8 instrucciones aritméticas	Asm c/ 16 instrucciones aritméticas
Promedios sin outliers	2429044	2618448	2651115	4183824	4556041



De la misma forma que en el filtro Cropflip. Podemos ver por el gráfico anterior que el limitante es la intensidad de cómputo. Ya que el tiempo de ejecución aumenta a medida que se agregar operaciones aritméticas. En cambio con accesos a memoria no hay tanta diferencia.

4. Bandas

4.1. Descripción

El filtro bandas sirve para, en términos simples, pasar una imagen de color a una en varios tonos de gris(blanco y negro). La forma de realizar esto es revisar píxel a píxel de la imagen y analizar el valor de la suma de los componentes R, G y B:
$b_{i,j} = src.r_{i,j} + src.g_{i,j} + src.b_{i,j}$
A los píxeles de la imagen destino se les da valores basado en la siguiente ecuación.

$$dsp(i,j) < r,g,b >= \begin{cases} < 0,0,0 > & b < 96 \\ < 64,64,64 > & 96 <= b < 288 \\ < 128,128,128 > & 288 <= b < 480 \\ < 192,192,192 > & 480 <= b < 672 \\ < 255,255,255 > & \text{si no} \end{cases}$$

El siguiente es el pseudocódigo de bandas en C:

```
for i = 0...columns − 1 do
    for j = 0...rows − 1 do
        suma = src_matriz_{i,j}.r + src_{i,j}.g + src_{i,j}.b;
        if suma < 96 then
            dst_matriz_{i,j}.r = 0;
            dst_matriz_{i,j}.g = 0;
            dst_matriz_{i,j}.b = 0;
        end
        if 96 <= suma < 288 then
            dst_matriz_{i,j}.r = 64;
            dst_matriz_{i,j}.g = 64;
            dst_matriz_{i,j}.b = 64;
        end
        if 288 <= suma < 480 then
            dst_matriz_{i,j}.r = 128;
            dst_matriz_{i,j}.g = 128;
            dst_matriz_{i,j}.b = 128;
        end
        if 480 <= suma < 672 then
            dst_matriz_{i,j}.r = 192;
            dst_matriz_{i,j}.g = 192;
            dst_matriz_{i,j}.b = 192;
        else
            dst_matriz_{i,j}.r = 255;
            dst_matriz_{i,j}.g = 255;
            dst_matriz_{i,j}.b = 255;
        end
    end
end
```

Algorithm 5: Algoritmo de Bandas en lenguaje C

En el caso del Asm, para poder paralelizar se toma una muestra de cuatro píxeles. Después se la divide en tres registros de 128 bits para poder tener en cada uno un solo color. Se los suma cada cual con el corre spondiente y se ve a que banda pertenece cada uno. Este pseudocódigo lo explica:

for $i = 0...columns - 1$ **do**
  for $j = 0...rows - 1$ **do**
    $xmm0 = [Pixel1|Pixel12|Pixel3|Pixel4]$;
    $xmm1 = [Pixel1.r|Pixel12.r|Pixel3.r|Pixel4.r]$;
    $xmm2 = [Pixel1.g|Pixel12.g|Pixel3.g|Pixel4.g]$;
    $xmm3 = [Pixel1.b|Pixel12.b|Pixel3.b|Pixel4.b]$;
    $xmm0 = [Pixel1.r + Pixel1.g + Pixel1.b|Pixel2.r + Pixel2.g + Pixel2.b|Pixel3.r +$
    $Pixel3.g + Pixel3.b|Pixel4.r + Pixel4.g + Pixel4.b]$;
    $xmm1 = [0, 0, 0, 0]$;
    $xmm2 = mask96\_288(xmm0)$;
    $xmm2 = xmm2 \wedge [64, 64, 64, 64]$;
    $xmm1 = xmm1 + xmm2$;

    $xmm2 = mask288\_480(xmm0)$;
    $xmm2 = xmm2 \wedge [128, 128, 128, 128]$;
    $xmm1 = xmm1 + xmm2$;

    $xmm2 = mask480\_672(xmm0)$;
    $xmm2 = xmm2 \wedge [192, 192, 192, 192]$;
    $xmm1 = xmm1 + xmm2$;

    $xmm2 = mask672(xmm0)$;
    $xmm2 = xmm2 \wedge [255, 255, 255, 255]$;
    $xmm1 = xmm1 + xmm2$;

    $dst_{i,j} = xmm1[1]$;
    $dst_{i,j+1} = xmm1[2]$;
    $dst_{i,j+2} = xmm1[3]$;
    $dst_{i,j+3} = xmm1[4]$;
  end
end

Algorithm 6: Algoritmo de Bandas en lenguaje ensamblador

Para remplazar los condicionales, utilizaremos máscaras y instrucciones pcmp. Estas están representadas con las funciones mask del pseudocódigo. Crearemos las mascaras que nos indicaran cuales valores cumplen las condiciones y hacemos un and con estas y el valor apropiado. Despues pasamos esto a la imagen resultado.

4.2. Experimentos

4.3. Experimento 3.1

Iniciamos el experimento evaluando la performande del código en C utilizando el flag -O1. En este caso no cambiamos la implementación.
Las diez mediciones de tiempo:

Común
12797809
11358186
11456151
11293769
11472111
11471974
11450355
11362942
11477382
11391629

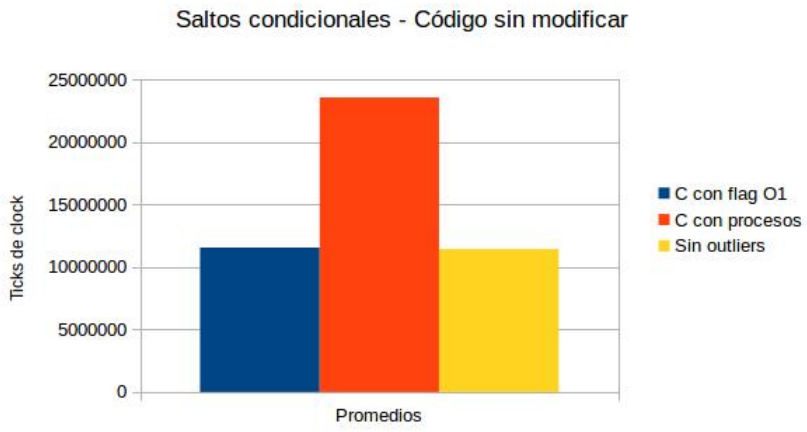
La esperanza para estas mediciones es de 11553230,8 y su varianza de 441664,378.

Con los procesos corriendo al mismo tiempo
23654158
16879559
27527850
34991082
25223929
20965508
31951311
22588902
16661662
15221451

La esperanza para estas mediciones es de 23566541,2 y su varianza de 6574579,376.

Sin los outliers
11456151
11472111
11471974
11450355
11362942
11391629

La esperanza para estas mediciones es de 11434193,666 y su varianza de 45819,082.



Ahora veamos que pasa en el caso de que modifiquemos el código y solo dejemos una banda.

Común con código cambiado
12436809
11426194
11446638
11500629
11287762
11280833
11780128
11297695
11351046
11389654

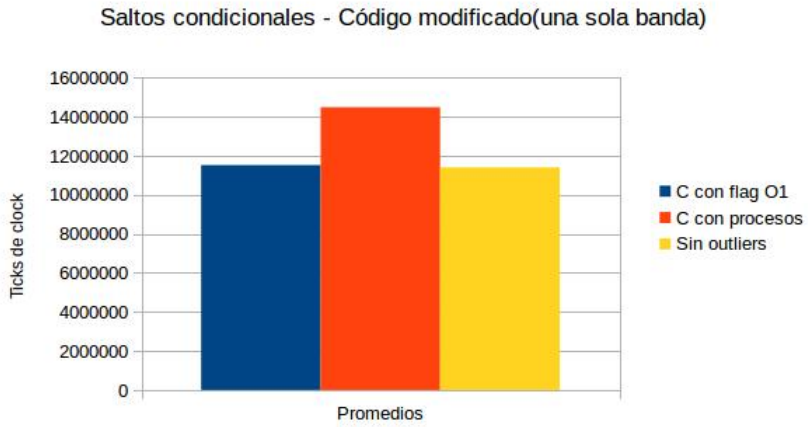
La esperanza para estas mediciones es de 11519738,8 y su varianza de 354158,127.

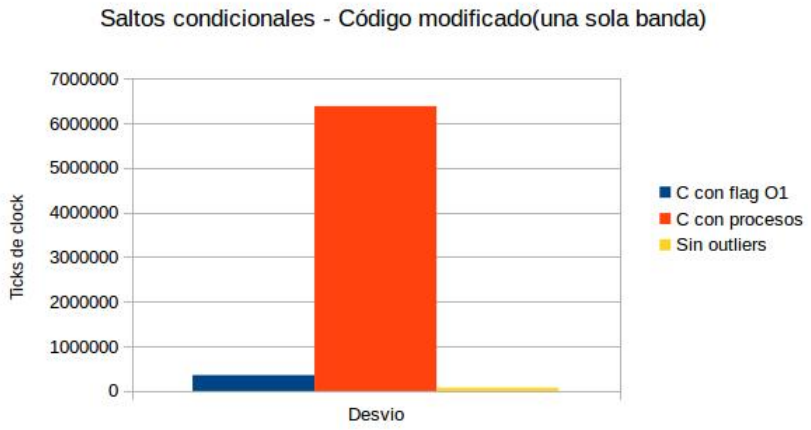
Con los procesos corriendo al mismo tiempo
20866272
11749017
11300268
11354311
11275667
11349775
11321804
13787571
11277630
30533381

La esperanza para estas mediciones es de 14481569,6 y su varianza de 6382347,830.

Común con código cambiado
11426194
11446638
11500629
11297695
11351046
11389654

La esperanza para estas mediciones es de 11401976 y su varianza de 72019,224.





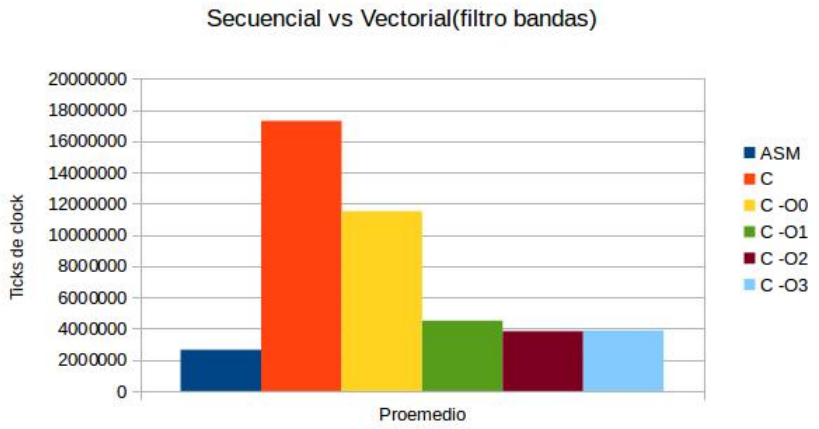
Como conclusión podemos observar que hay diferencia en todos los promedios. Esto muestra que, al haber menos saltos condicionales se puede obtener mejor performance.

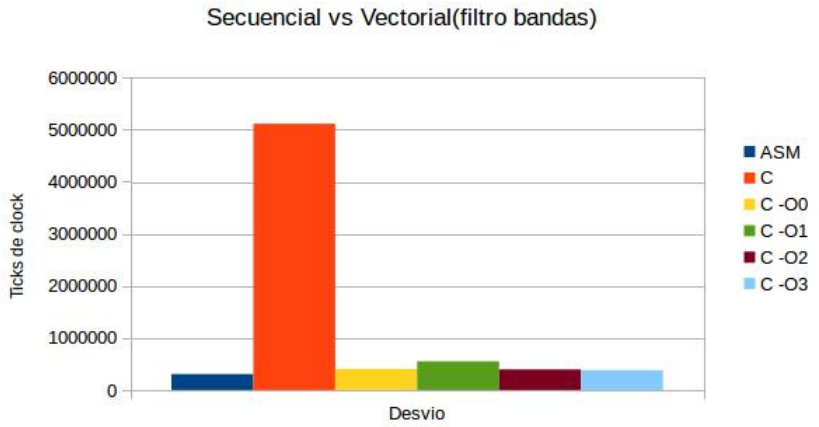
4.4. Experimento 3.2

Se toma, al igual que en Sierpinski, una muestra de 10 tiempos como se muestra en el siguiente cuadro.

ASM	C Común	C -O0	C -O1	C -O2	C -O3
3505918	22135543	12641517	6063488	4942045	4954026
2589290	20564901	11374010	4370152	3735952	3706931
2542470	14731584	11354143	4356261	3675945	3761121
2534196	12061801	11537515	4376631	3712537	3716506
2532852	11766793	11393466	4304202	3728077	3734503
2532789	14631057	11318969	4307278	3653412	3823449
2557916	15051960	11323085	4329139	3650608	3754663
2533167	18488820	11380404	4302995	3678528	3751923
2533650	28220955	11461653	4319343	3768387	3796926
2534175	15310701	11364076	4301430	3690456	3757289

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	2639642,3	17296411,5	11514883,8	4503091,9	3823594,7	3875733,7
Desvío estándar	304909,096	5115429,788	401257,823	549008,248	394767,849	380439,069

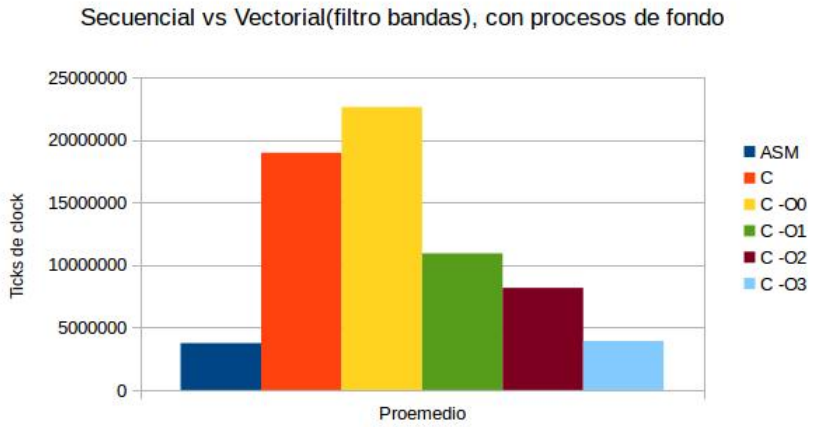


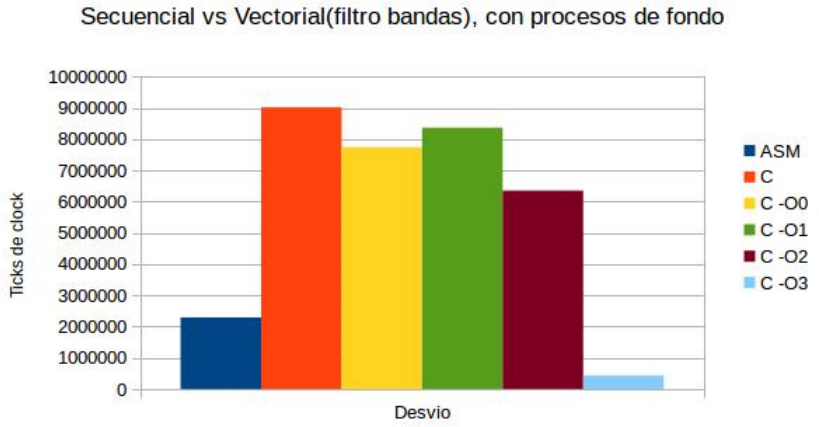


Con los procesos de los cores lógicos:

ASM	C Común	C -O0	C -O1	C -O2	C -O3
3520388	40116269	37608428	28102158	8617277	5064644
2593217	22731334	22614501	4672825	3659492	3700715
5670735	13091673	24022404	16448796	3740572	3698320
2658001	11822485	11310001	13676554	3657665	3716433
2575713	26133355	22537746	9039156	15063752	3703833
2538091	11627679	24434225	19962600	5914944	3704757
9665964	17933191	20460184	4367391	6998639	3698814
2639889	20779059	22565466	4385083	23254434	4195769
2672597	12737771	11301497	4397085	3698772	3899291
2933228	12931149	29606566	4304727	7139958	3749571

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	3746782,3	18990396,5	22646101,8	10935637,5	8174550,5	3923080,5
Desvío estándar	2289566,028	9017447,683	7741773,966	8364446,914	6351725,293	429928,854

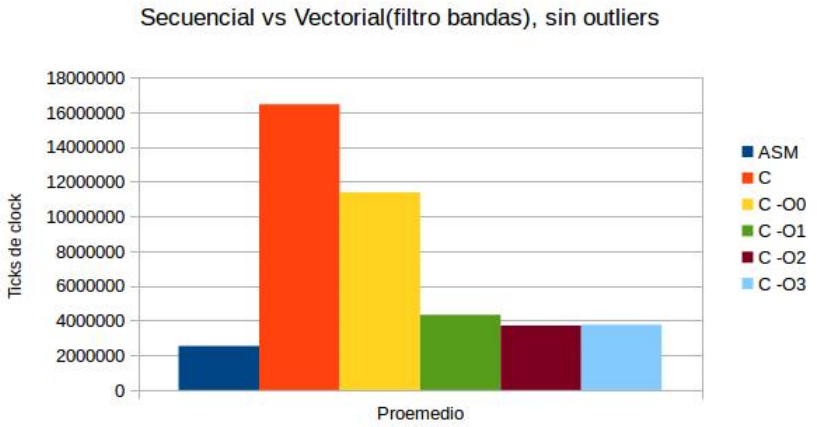


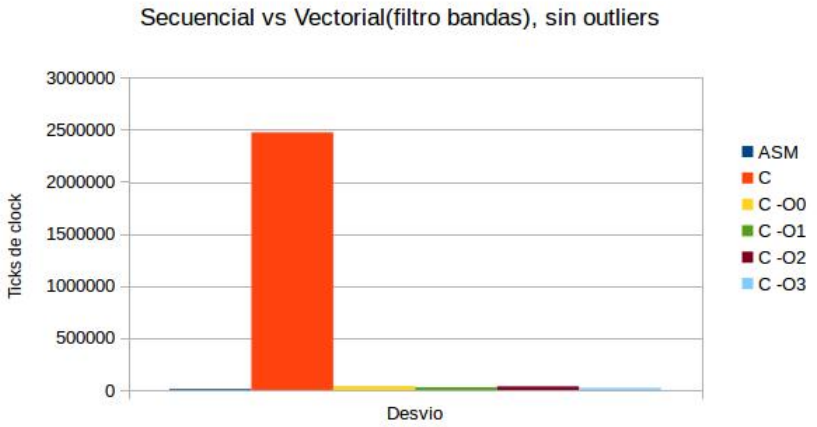


Sin los outliers:

ASM	C Común	C -O0	C -O1	C -O2	C -O3
2542470	20564901	11374010	4370152	3675945	3761121
2534196	14731584	11354143	4356261	3712537	3734503
2557916	15310701	11393466	4319343	3728077	3754663
2533167	14631057	11380404	4304202	3678528	3751923
2533650	15051960	11461653	4307278	3768387	3796926
2534175	18488820	11364076	4329139	3690456	3757289

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	2539262,3336	16463170,5	11387958,666	4331062,5	63708988,333	3759404,166
Desvío estándar	9782,110	2473951,563	38540,121	26799,958	35407,098	20561,245





Como podemos ver en cada una de las mediciones tenemos que el código en ensamblador presenta siempre mejor desempeño respecto al de C con cualquiera de los flags. En cuanto al de C tenemos que la mejora de los flags parece ser ascendente en función de los promedios de tiempos. A medida que se aumenta el grado de optimización mejora el tiempo de ejecución.

5. Motion Blur

5.1. Descripción

Este filtro toma una imagen de entrada y le aplica un efecto de desenfoque de movimiento. Para lograr esto se tiene que aplicar la siguiente función de cada píxel:
$dst_{i,j} = 0,2 * src_{i-2,j-2} + 0,2 * src_{i-1,j-1} + 0,2 * src_{i,j} + 0,2 * src_{i+1,j+1} + 0,2 * src_{i+2,j+2}$
La función toma partes del valor de sus vecinos. Los bordes se pintan de negro por no ser posible calcular la anterior ecuación.

$dst_{i,j} = 0sii < 2 \vee j < 2 \vee i + 2 >= tamy \vee j + 2 >= tamx$

El siguiente es el seudocódigo asociado al filtro motion blur en la implementación en C:

for $i = 0...filas - 1$ **do**
  for $j = 0...cols - 1$ **do**
    if $i < 2$ *or* $j < 2$ *or* $i + 2 >= tamy$ *or* $j + 2 >= tamx$ **then**
      $dst\_matriz_{i,j} = 0$
    else
      $dst\_matriz_{i,j}.r = 0,2 * src\_matriz_{i-2,j-2}.r + 0,2 * src\_matriz_{i-1,j-1}.r + 0,2 *$
      $src\_matriz_{i,j}.r + 0,2 * src\_matriz_{i+1,j+1}.r + 0,2 * src\_matriz_{i+2,j+2}.r$
      $dst\_matriz_{i,j}.g = 0,2 * src\_matriz_{i-2,j-2}.g + 0,2 * src\_matriz_{i-1,j-1}.g + 0,2 *$
      $src\_matriz_{i,j}.g + 0,2 * src\_matriz_{i+1,j+1}.g + 0,2 * src\_matriz_{i+2,j+2}.g$
      $dst\_matriz_{i,j}.b = 0,2 * src\_matriz_{i-2,j-2}.b + 0,2 * src\_matriz_{i-1,j-1}.b + 0,2 *$
      $src\_matriz_{i,j}.b + 0,2 * src\_matriz_{i+1,j+1}.b + 0,2 * src - matriz_{i+2,j+2}.b$
    end
  end
end

Algorithm 7: Algoritmo de Motion Blur en lenguaje C

Para poder realizarlo en assembler con las paralelizaciones pedidas utilizamos el código representado por el siguiente pseudocódigo:

/* escribo primeras lineas negras */
$CantidadDeNegros$ =ancho de la imagen en píxeles$*2/16$
/* quiero escribir 2 lineas y las escribo de a 16 bytes */
$xmm0 = [0, 0, ..., 0]$
$punteroImagenDestino = dst$
for $i = 0...CantidadDeNegros - 1$ **do**
  $[punteroImagenDestino] = xmm0$
  $punteroImagenDestino+ = 16$
end
/* Pego la imagen procesada */
$punteroImagenDestino =$ ancho de la imagen en píxeles$*2 + 8 + dst$
$punteroImagenOrigen =$ancho de la imagen en píxeles$*2 + 8 + src$
/* me paro en la tercera fila corrido del borde en ambas imágenes */
$dezplazamientosimple =$ancho de la imagen en píxeles$+4$
for $i = 2..filas - 3$ **do**
  for $j = 2..cols - 3$ **do**
    $calcularBlur(punteroImagenOrigen, dezplazamientosimple)$
    $[punteroImagenDestino] = xmm2\ punteroImagenDestino+ = 16$
    $punteroImagenOrigen+ = 16$
  end
  $punteroImagenDestino+ = 16$
  $punteroImagenOrigen+ = 16$
end
/* escribo ultimas lineas negras */
$CantidadDeNegros$=ancho de la imagen en píxeles$*2/16$
/* quiero escribir 2 lineas y las escribo de a 16 bytes */
$xmm0 = [0, 0, ..., 0]$
$punteroImagenDestino = dst+$((ancho de la imagen en píxeles)$*(filas - 4)$)
for $i = 0...CantidadDeNegros - 1$ **do**
  $[punteroImagenDestino] = xmm0$
  $punteroImagenDestino+ = 16$
end
/* Dibujo los Bordes */
$bordeIzq = dst+$ ((ancho de la imagen en píxeles)$*2$)
$bordeDer = bordeIzq+$(ancho de la imagen en píxeles)-8)
/* quiero escribir los primeros 8 bytes y los últimos 8 bytes*/
$punteroImagenDestino = dst$
for $i = 2..filas - 4$ **do**
  $[bordeDer] = mm0$
  $[bordeIzq] = mm0$
  $bordeDer+ = (anchodelaimagenenpixeles)$
  $bordeIzq+ = (anchodelaimagenenpixeles)$
end

Algorithm 8: Algoritmo de Motion Blur en Lenguaje Ensamblador

$xmm2 = [0, 0, ..., 0]$
for $p = -2.,2$ **do**
  $calcularpunto2(punteroImagen + (dezplazamientosimple + p))$
  $xmm2+ = xmm1$
end

Algorithm 9: Algoritmo de calcularBlur

$xmm1 = [puntero]$
$xmm3 = rojos(xmm1)$
$xmm4 = verdes(xmm1)$
$xmm5 = azules(xmm1)$
$xmm3 = multiplicarPorPunto2(xmm3)$
$xmm4 = multiplicarPorPunto2(xmm4)$
$xmm5 = multiplicarPorPunto2(xmm5)$
$xmm1 = reacomodar(xmm3, xmm4, xmm5)$

Algorithm 10: Algoritmo de calcularPunto2

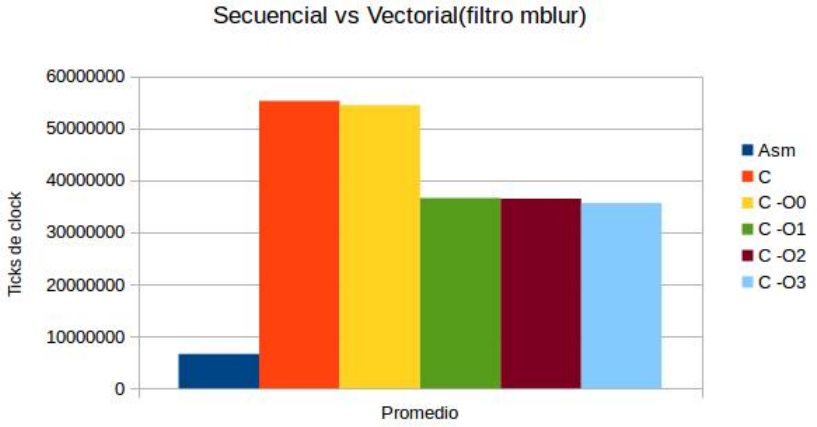
5.2. Experimentos

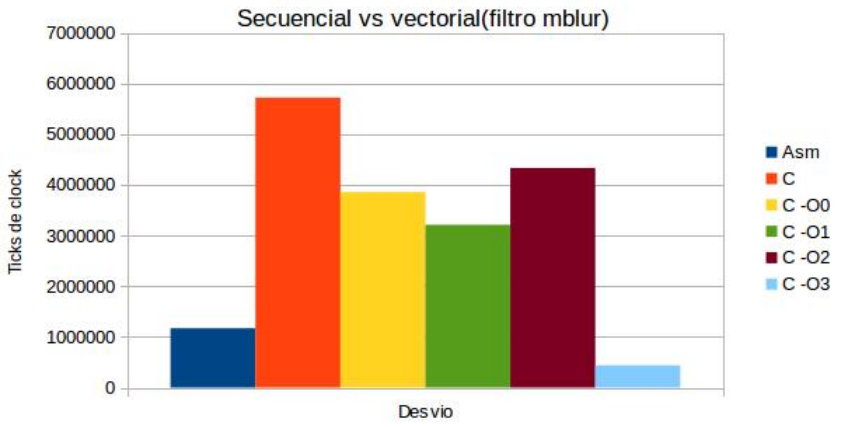
5.3. Experimento 4.1

Ejecuciones normales:

ASM	C Común	C -O0	C -O1	C -O2	C -O3
9811133	71481047	65367066	45690152	48757938	36848987
6475657	53292102	53092896	35605701	35310343	35482460
6082872	53386860	53290886	35641333	35305795	35453542
6073548	53402474	52988651	35375138	35035853	35446445
6083263	53423766	53328311	35735266	35047948	35457402
6191587	53383341	53020204	35480725	34928285	35454715
7105584	53380246	53130117	35603857	35042228	35458166
6077194	53403868	53386978	35426223	35009035	35451349
6138411	53459756	53055045	35478923	35029172	35463802
6073581	53402321	53331958	35570851	34907520	35449573

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	6611283	55201578.1	54399211.2	36560816.9	36437411.7	35596644.1
Desvío estándar	1170141.364	5720179.373	3856421.478	3209534.859	4331142.437	440143.252

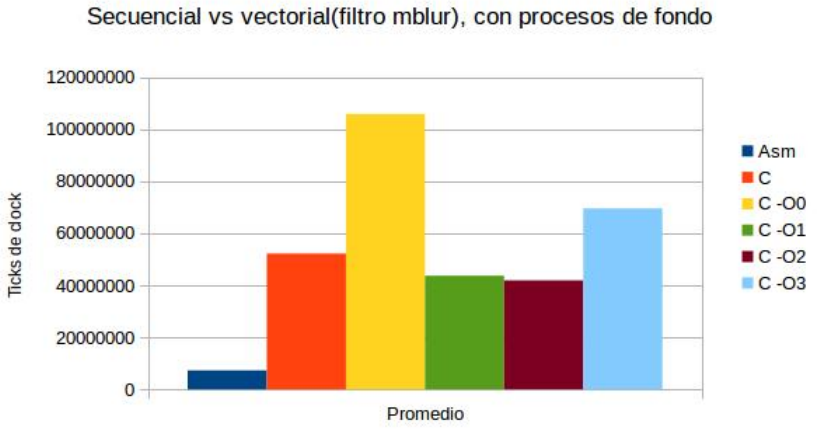


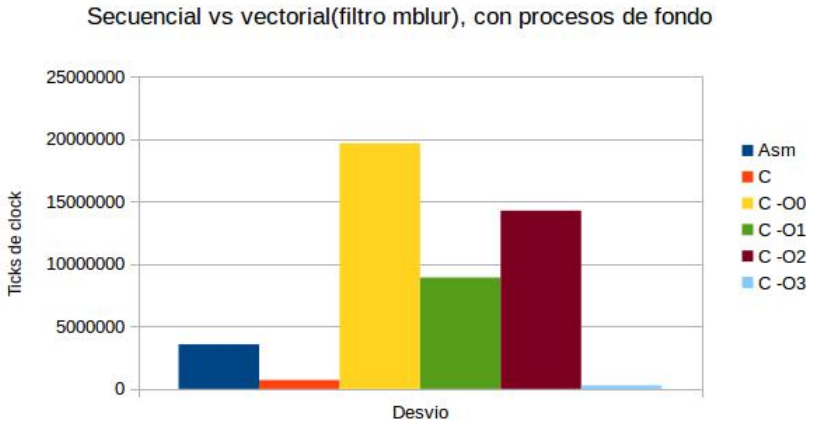


Ejecuciones con procesos de fondo:

ASM	C Común	C -O0	C -O1	C -O2	C -O3
7028607	54165561	108283600	55304774	36276538	70333140
6067921	51980211	137908488	35431544	34912347	69450712
17428782	51927103	110948366	53267979	34897387	69503896
6056054	51939743	106124251	36090813	34899538	69440427
6055162	51918722	97325536	52284248	34897226	69452395
6171578	51925591	139943107	35441175	34917320	69454792
6057270	51958043	97851481	45701347	68901527	69469463
6100773	51940695	86538067	35319999	68898909	69450636
6077780	51920440	86534939	52266627	34897625	69456824
6060194	51922148	86521687	35191326	34898730	69459960

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	7310412.1	52159825.7	105797952.2	43629983.2	41839714.7	69547224.5
Desvío estándar	3567860.188	705010.590	19659036.034	8909090.312	14268569.116	276678.882

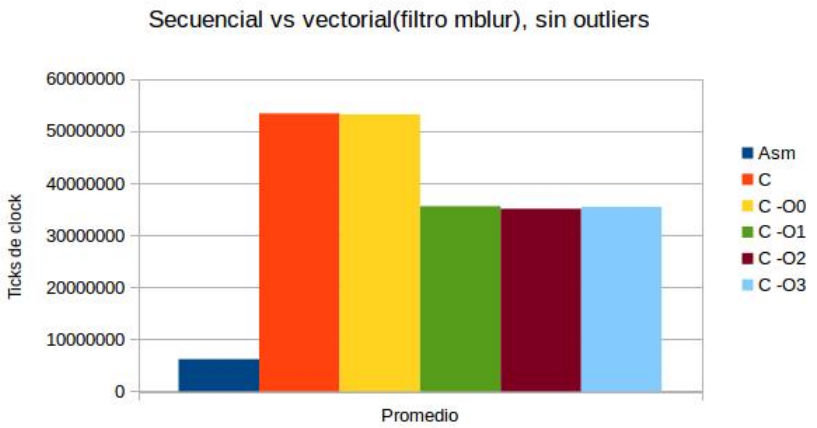


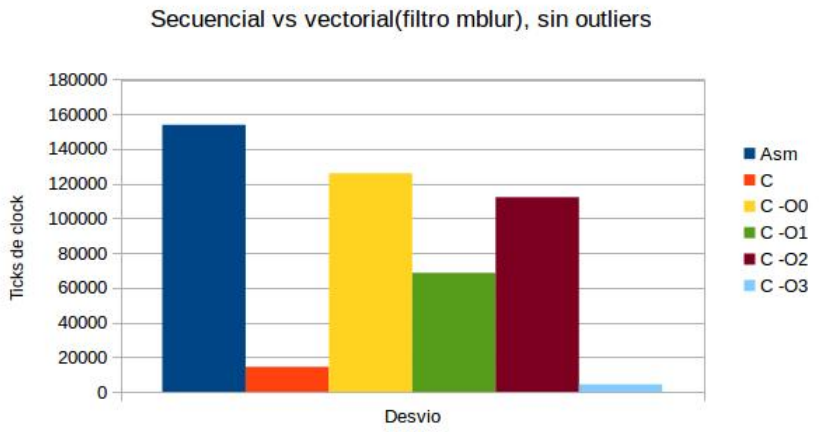


Ejecuciones sin los outliers:

ASM	C Común	C -O0	C -O1	C -O2	C -O3
6475657	53386860	53092896	35605701	35305795	35457402
6082872	53402474	53290886	35641333	35035853	35454715
6083263	53423766	53328311	35480725	35047948	35458166
6191587	53383341	53130117	35603857	35042228	35451349
6077194	53402321	53055045	35478923	35009035	35463802
6138411	53403868	53331958	35570851	35029172	35453542

	ASM	C Común	C -O0	C -O1	C -O2	C -O3
Promedio	6174830.666	53400438.333	53204868.833	35563565	35078338.5	35456496
Desvío estándar	153933.493	14424.430	125985.325	68595.290	112240.231	4367.540





Podemos ver que la versión de assembler se ve aventajada, en cuanto a tiempo se refiere, respecto a la de C para cualquier flag de optimización. Y en cuanto al código en C O1, O2 y O3 presentan tiempos parecidos y mejores a C sin optimización. En el caso de los procesos de fondo, el C sin optimizar muestras ser más lento que con las optimizaciones 01 y 02. Pero es peor con 03.

6. Conclusión

Para realizar este trabajo práctico utilizamos el set de instrucciones SSE. La idea principal del trabajo fue realizar la implementación de 4 filtros de imágenes. Primero en C y luego en Assembler, usando en este último el concepto de paralelización de procesos SIMD(Single instruction multiple data).

Las dificultades que se observaron fueron principalmente en la versión de lenguaje ensamblador. Los requerimientos de enunciado complicaron la realización del código. Como por ejemplo los condicionales del bandas los cuales tuvimos que implementar usando máscaras e instrucciones de comparación ó en Sierpinski las conversiones.

En cuanto a los experimentos se pudo observar que la implementación en asm superaba considerablemente la de C sin importar los flags de optimización que se utilicen. Esto se debe principalmente a la posibilidad de SIMD de procesar más de un píxel a la vez. La forma en la que se realizan los ciclos en ensamblador es un poco más complicado ya que se debe ver la matriz como una tira de valores. Descartando la posibilidad de verla como un arreglo bidimensional. Es más cómodo además porque en C nos tomamos la libertad de pasarlo a forma de matriz.