DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

subtitulo del trabajo

Organización del Computador II

Segundo Cuatrimestre de 2014

Integrante	LU	Correo electrónico
Fosco, Martin Esteban	449/13	mfosco2005@yahoo.com.ar
Palladino, Julián	231/13	julianpalladino@hotmail.com
De Carli, Nicolás	164/13	nikodecarli@gmail.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. Objetivos generales

Este Trabajo Práctico se ha centrado en explorar el modelo de programación **SIMD**, usándolo para una aplicación popular del set de instrucciones SIMD de intel (SSE), procesamiento de imágenes y videos. En particular, se implementaron 4 filtros de imágenes: Cropflip, Sierpinski, Bandas y Motion Blur.

2. Desarrollo

2.1. Explicación de Implementaciones en assembler

2.1.1. Cropflip

En el filtro Cropflip utilizamos aritmética de punteros para empezar a escribir en la útima fila de la imagen destino mientras se lee de la primer fila de la parte a recortar de la imagen fuente. Al cambiar de fila, el puntero de escritura (*rsi*) decrece, mientras que el de lectura crece (*rdi*). A su vez, utilizamos 3 contadores: De Bytes horizontales (*r11*), de Bytes verticales (*EAX*) y *r10* como backup de *rbx*, para cuando es modificado. Éste último sirve. El SIMD es aprovechado al máximo, ya el movimiento es de a 4 píxeles.

2.1.2. Sierpinski

El filtro Sierpisnki es procesado de la siguiente manera:

- Cargamos de la memoria 4 píxeles, los desempaquetamos a dwords y luego los convertimos a floats, para poder calcular el *coef* de cada uno. Quedan entonces en *xmm0*, *xmm1*, *xmm2* y *xmm3*, con sus componentes como floats.

- Luego, para calcular el coef de cada píxel, utilizamos valores ya cargados antes del ciclo (Por ejemplo *255/cols* en *xmm12* y *255/filas* en *xmm11*).

- Se multiplica el coef por todos los componentes de cada pixel al mismo tiempo utilizando SIMD.

- Por último se convierten y empaquetan estos 4 píxeles a Bytes, guardados en *xmm0*, y desde allí son almacenados al destino.

2.1.3. Bandas

El filtro Bandas lo intentamos implementar tres veces. En esas tres fuimos variando la forma de pensar el filtro, sobretodo el 'branching' que generaba la parte de pasar de la suma de R, G y B al número que teníamos que guardar en el destino.

- Con condicionales:

  La primer manera (la más ineficiente de las 3) fue intentar replicar el código C y hacer los condicionales correspondientes. Esta forma no sólo resultó ser la más díficil de implementar (El código se tornó muy largo y confuso), sino que nos dimos cuenta rápidamente que era extremadamente ineficiente y no llegamos a terminarla.

- Con cálculos:

  La segunda forma consistió en aplicar una cuenta matemática para pasar de la suma (de R, G y

B) al resultado final (La cuenta en sí era res=[([suma/96]+1)/2]*64). Esta nueva manera de pensar el ejercicio consigió eliminar el enorme branching que generaban los condicionales de la implementación anterior, pero todavía podría ser optimizado aún más.

- Con máscaras:

  Finalmente, por consejo de algunos docentes, decidimos implementar el ejercicio con el uso de máscaras. La idea es, en cada iteración:

  - Levantar cuatro píxeles en un XMM.
  - Luego en otros dos XMM hacer rotaciones para que queden alineados verticalmente los R, G y B de los cuatro píxeles.
  - Haciendo un pand con *xmm10* pasamos los 4 píxeles por una máscara tal que hayan ceros en todo el registro, excepto en los R, G y B alineados
  - Con una suma vertical (padd) obtenemos un XMM con las sumas R+G+B de los 4 píxeles
  - Luego, en lugar de hacer el branching innecesario, utilizamos '*pcmpgtw*' para compararlos con 4 máscaras que contengan 95, 287, 479 y 671. De esta forma, obtenemos máscaras que contienen unos en aquellos dwords que son mayores, y ceros donde son menores o iguales.
  - La idea es que, como nosotros ya sabemos el número que tiene el registro antes de aplicar cada máscara, podemos hacer máscaras específicas para llevar ese número al resultado deseado. Antes de aplicarla, le hacemos *and* con el resultado de la comparación. Si éste es 0, entonces la máscara no afectará el resultado.
  - Finalmente utilizamos *pshufb* para hacer convertir cada dword en cuatro Bytes iguales, tal que R=G=B=A=Suma.

  Luego de rotar y pasar por la máscara, quedaría de esta forma:

	0															127
XMM0	B_1	0	0	0	B_2	0	0	0	B_3	0	0	0	B_4	0	0	0
XMM1	G_1	0	0	0	G_2	0	0	0	G_3	0	0	0	G_4	0	0	0
XMM2	R_1	0	0	0	R_2	0	0	0	R_3	0	0	0	R_4	0	0	0

  Luego de sumar verticalmente:

	0															127
XMM0	S_1	0	0	0	S_2	0	0	0	S_3	0	0	0	S_4	0	0	0

  Y al aplicar *pshufb*:

	0															127
XMM0	S_1	S_1	S_1	S_1	S_2	S_2	S_2	S_2	S_3	S_3	S_3	S_3	S_4	S_4	S_4	S_4

  El uso de máscaras no sólo resultó ser más claro, sino que también demostró ser altamente eficiente en comparación con el branching excesivo y las cuentas de punto flotante.

2.1.4. Motion Blur

2.2. Experimentos

Nota: Las mediciones de tiempo (en ciclos) de ejecución de las diversas implementaciones fuerna hechas en una Intel Core i5-3210M (2.50 Ghz)

2.2.1. Cropflip

Experimento 1.1)

a)Al hacer el .ºbjdump"no solo se imprime la función çropflip_c.c", sino que también se imprimen varias funciones que utiliza GDB para hacer el debugging, tales como ".debug_info", ".debug_abbrev", ".debug_aranges", etc. También imprime ".commentz .eh_header", que contienen información sobre el Linker y el compilador.

b) Las variables locales las almacena en memoria, haciendo movs manualmente en el stack frame. (Por ej, poniendo las variables en [rbp-0x58], [rbp-0x60], [rbp-0x64]).

c) Se podría optimizar el almacenamiento de las variables locales. Como el acceso a memoria es mucho más ineficiente que el acceso a los registros, sería más óptimo almacenar las variables en ellos.

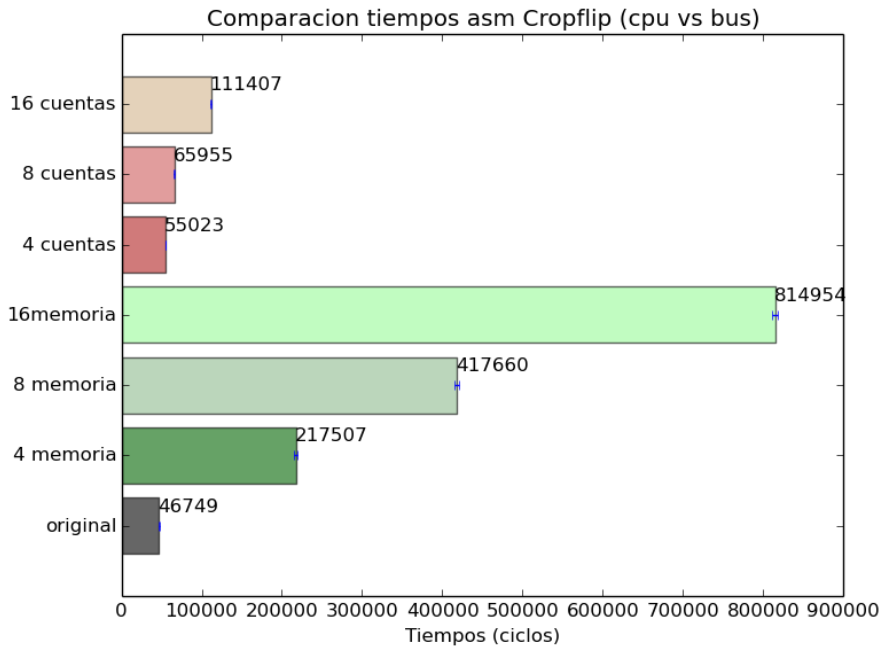
Experimento 1.2)

a) -O1 no reduce mucho el tiempo de compilación, y ejecuta una optimización "moderada". Se ve claramente que se reducen los accesos a memoria para las variables locales, y se reducen mucho las líneas de código.

b) Usando otros parámetros como -O2 u -O3 se hace la compilación con más optimizaciones cuanto más grande sea el número. También están -Os que optimiza el tamaño del código y -Og que optimiza el debugging.

c)

- **FDCE:** Hace eliminación de "dead code"(Código muerto), es decir, borra el código que consume recursos pero sus resultados nunca son utilizados.
- **FDSE:** Hace eliminación de "dead store"(Almacenamiento muerto), o sea, ignora aquellas variables que después no llegan a ser utilizadas.
- **-fif-conversion y -fif-conversion2:** Reemplazan condicionales por equivalentes aritméticos. Esto incluye funciones como movimientos condicionales, mínimos, máximos, función .ªbs", entre otros. Luego de ver los resultados del experimento 3.1 (Saltos condicionales en el filtro Bandas) queda claro que esta optimización es extremadamente útil.



En el gráfico se nota la clara diferencia entre las corridas con optimizaciones y sin ellas. El criterio fue el mismo que aplicamos en los experimentos de más adelante (Haciendo 1000 iteraciones).

Experimento 1.3)

En este experimento consideraremos diferentes criterios realizando siempre 10 mediciones, con outliers, sin ellos, y agregando programas en C++ que trabajen en simultáneo.

a) Resultados:

422861, 242136, 186857,
194709, 234097, 221979,
247912, 227976, 223245,
200770.

b) Resultados:

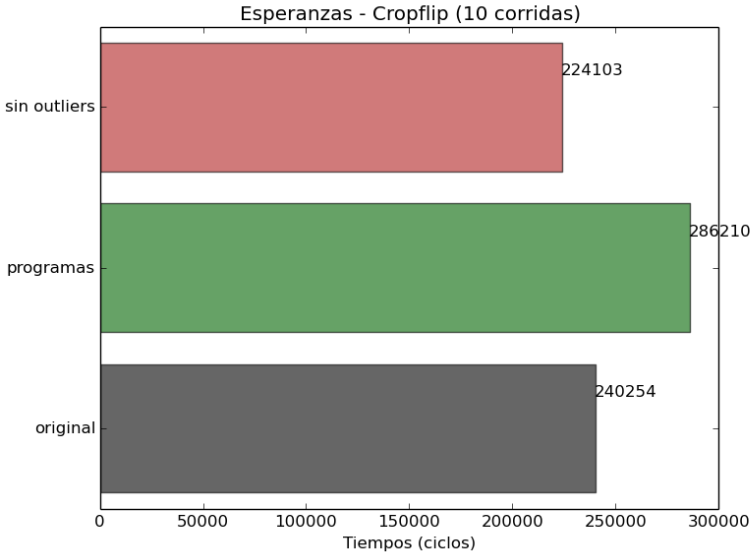
289209, 302654, 258757,
254428, 248527, 254606,
260391, 338981, 373612,
280937

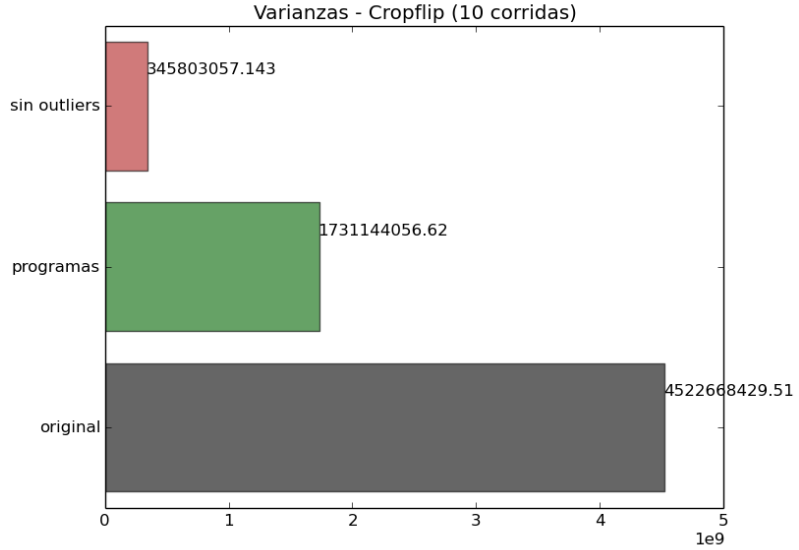
c)
10 mediciones
Esperanza: 240254.2
Desvío standard: 67250.78758
Varianza: 4522668429.51111

10 mediciones con el programa en simultáneo:
Esperanza: 286210.2
Desvío standard: 41607.01932
Varianza: 1731144056.62222

d) 10 mediciones sin 2 outliers:
Esperanza: 224103
Desvío standard: 18595.78063
Varianza: 345803057.14286

e)





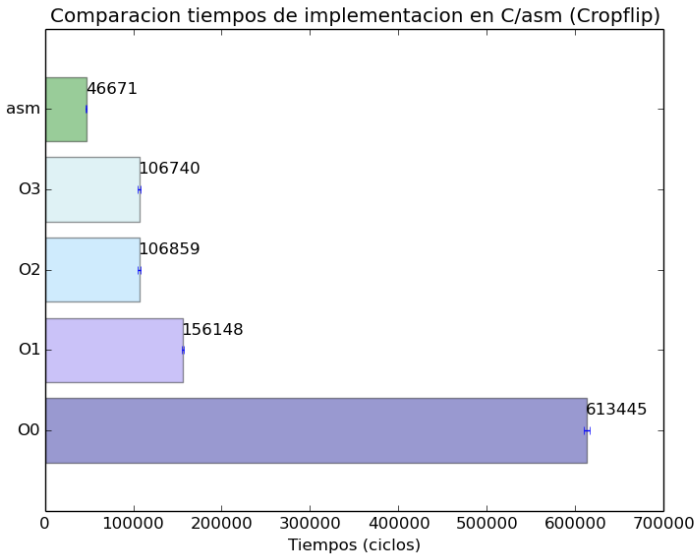
Luego de observar estas mediciones hemos notado una varianza demasiado alta (y, en consecuencia, una falta de confiabilidad en las mediciones), por lo tanto, luego de probar con distintos métodos de medición decidimos tomar el promedio de 1000 mediciones 10 veces, eliminar outliers y calcular luego nuevamente el promedio de las mediciones restantes.

Experimento 1.4)

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro cropflip (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

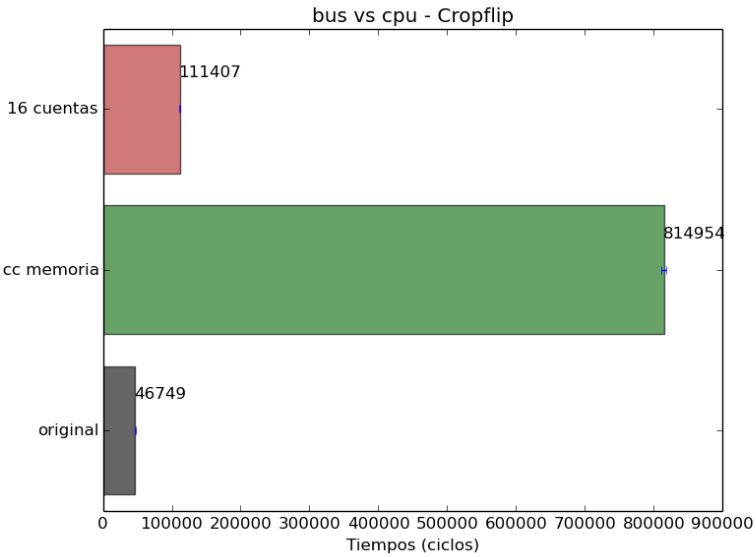
- **asm-** 46671 ciclos
- **C O3-** 106740 ciclos
- **C O2-** 106859 ciclos
- **C O1-** 156148 ciclos
- **C O0-** 613445 ciclos

NOTA: las líneas pequeñas azules en el tope de las barras representan el desvío estándar del promedio de las mediciones que estamos mostrando.



Se observa en este experimento una notoria ventaja (en eficiencia temporal) del procesamiento vectorial de datos sobre el secuencial, incluso luego de haber aplicado las optimizaciones sobre el secuencial (que redujeron mucho el tiempo de procesamiento) la implementación de assembler pudo aplicar el filtro mucho más rápido y de manera más efectiva, mientras que entre las optimizaciones O2-O3 no hubo diferencia casi, indicando que no parece ser posible optimizar mucho más la implementación en C.

Experimento 1.5) Cropflip - cpu vs bus de memoria



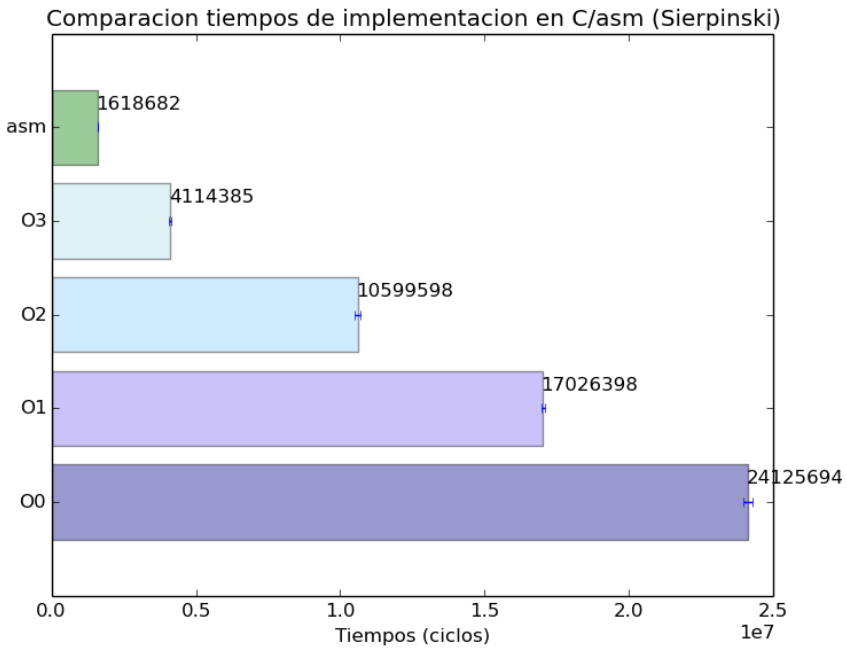
blablalbla pregrafico blablabla postgrafico

2.2.2. Sierpinski

Experimento 2.1) Sierpinski - secuencial vs vectorial

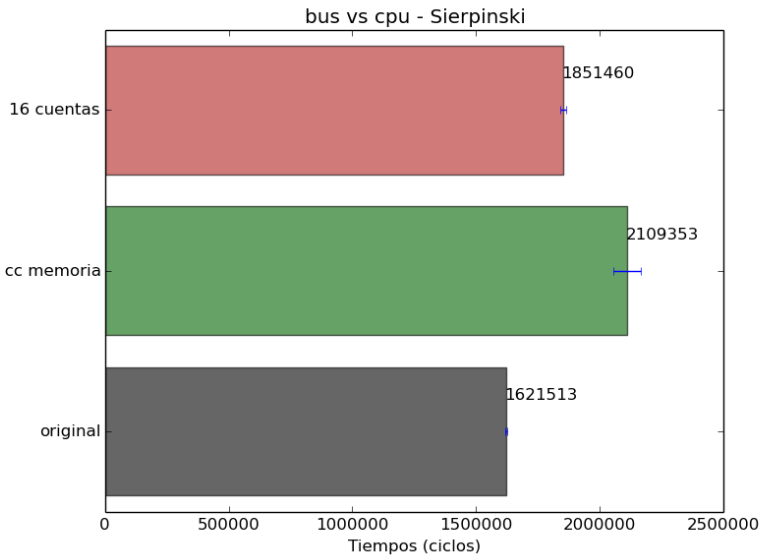
Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro sierpinski (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 1618682 ciclos
- **C O3-** 4114385 ciclos
- **C O2-** 10599598 ciclos
- **C O1-** 17026398 ciclos
- **C O0-** 24125694 ciclos



En este experimento la ventaja de usar SIMD siguió notándose con fuerza, mostrando una gran diferencia cuando es contrastado su tiempo de ejecución con el de las distintas optimizaciones de la implementación en C, si bien en este caso se notó que la optimización O3 tiene un gran impacto comparada con la O2.

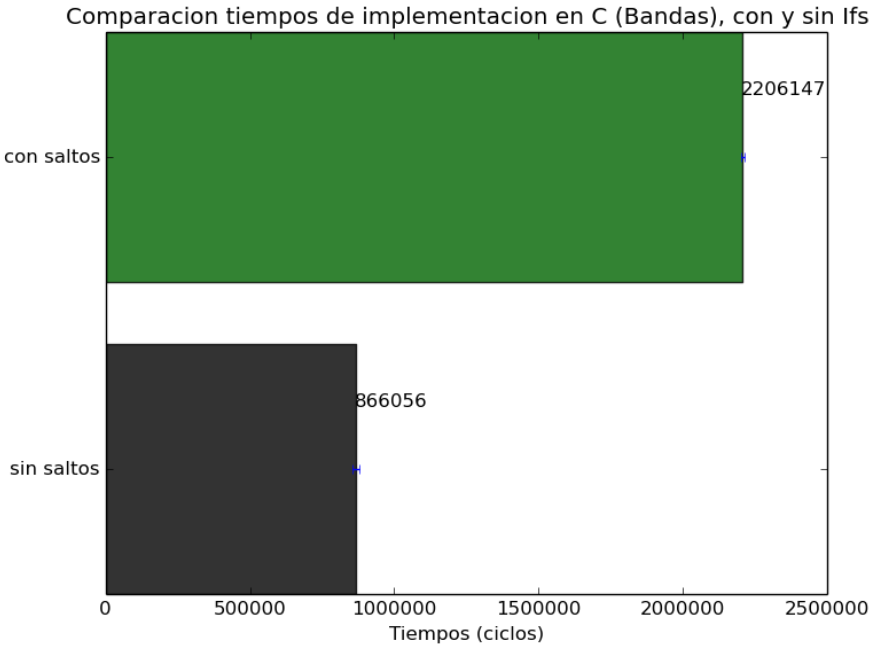
Experimento 2.1) Sierpinski - cpu vs bus de memoria



2.2.3. Bandas

Experimento 3.1) Bandas - saltos condicionales

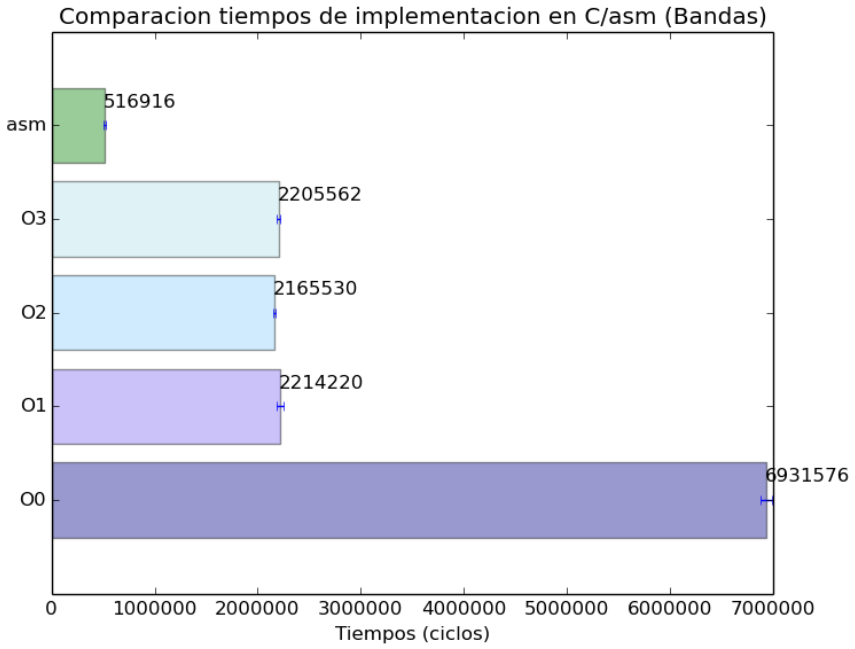
En este experimento corrimos el programa con el flag -O1, en comparación con el mismo programa sin los condicionales. Como explicamos en el punto 1.2, -O1 ofrece ciertas optimizaciones del código que reducen sustancialmente el 'branching', reemplazándolo en lo posible por instrucciones aritméticas. Por ello, ya esperábamos cierta diferencia a favor del programa sin condicionales. Sin embargo al correr los experimentos nos sorprendió que la diferencia era abismal: El código sin condicionales corre 3 veces más rápido que el optimizado. Esto deja en evidencia lo ineficiente que es utilizar saltos condicionales, incluso si lo optimizamos.



Experimento 3.2) Bandas - secuencial vs vectorial

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro bandas (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 516916 ciclos
- **C O3-** 2205562 ciclos
- **C O2-** 2165530 ciclos
- **C O1-** 2214220 ciclos
- **C O0-** 6931576 ciclos

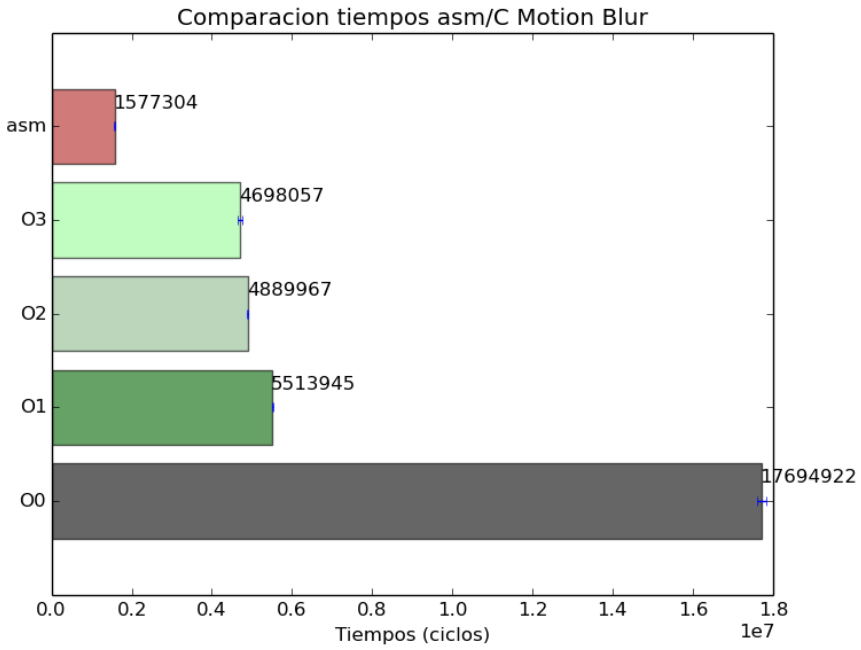


2.2.4. Motion Blur

Experimento 4.1) Motion Blur - secuencial vs vectorial

Este experimento comparará la performance de las implementaciones C y assembler que aplican el filtro motion blur (siguiendo el modelo de procesamiento secuencial y vectorial, respectivamente), probamos la implementación en C con 4 distintos flags de optimización.

- **asm-** 1577304 ciclos
- **C O3-** 4698057 ciclos
- **C O2-** 4889967 ciclos
- **C O1-** 5513945 ciclos
- **C O0-** 17694922 ciclos



En este experimento, nuevamente, la implementación de assembler mostró una clara diferencia (positiva) con las distintas optimizaciones de la de C, mientras que en C no se pudo optimizar el código mucho más allá de la primera optimización.

3. Conclusiones y trabajo futuro