DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico II

Procesamiento Vectorial de Bajo Nivel

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

Resumen

Se han implementado en este trabajo 4 filtros de imágenes: Cropflip, Sierpinski, Bandas y Motion Blur en C y assembler, luego se trabajó sobre estos aplicando distintas optimizaciones (y comparando tiempos con las implementaciones originales), finalmente se llevaron a cabo experimentos (indicados por el enunciado) con distintas versiones del programa sobre la imagen 'lena.512x512.bmp' y se intentó sacar conclusiones de estos, para confirmar y probar las ventajas que conlleva trabajar con SIMD.

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:: Recorta y voltea la imagen.
Sierpinski: Oscurece píxeles de forma tal que se formen triángulos de Sierpinski.
Bandas: Transforma la imagen a tonos de grises, según la suma de sus componentes R, G y B.
Motion Blur: Crea un efecto de desenfoque en la imagen.

Se ha buscado aprovechar los beneficios de SSE en:

- Procesar conjuntos de datos de manera eficiente, ejecutando de manera paralela (al mismo tiempo) la misma instrucción sobre distintos datos.
- El uso de los registros XMM, los cuales proveen una gran versatilidad con la opción de ejecutar operaciones de punto flotante y enteras con distintas precisiones (sobre datos empaquetados o escalares).
- Reducir los accesos a memoria aun mínimo indispensable.

Luego de implementar en C y asm (procesando de manera escalar y vectorial los datos, respectivamente) los filtros de imágenes se ha comparado la performance de ambos modelos de programación para determinar de manera aproximada la ventaja que se gana al trabajar sobre muchos datos de manera simultánea.

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 *r11* para cuando es modificado.

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

Para el Motion Blur el ciclo consta de:

- Al principio del principio se checkea si el puntero de escritura apunta a algún borde. Si es así se salta para escribir píxeles negros, si no, se procede al procesamiento.

- Levantamos de memoria 20 píxeles, en *xmm0, xmm2, xmm4, xmm6 y xmm8*. Haciendo copias en *xmm10, xmm11, xmm12, xmm13, xmm14* respectivamente.

- Se desempaquetan a word los dos píxeles mas bajos, para poder sumar sin que se genere overflow.

- Se suman dos píxeles, luego se desempaquetan y se convierten a floats para dividirlos por 5. Por último se empaquetan a words nuevamente.

- Luego se desempaquetan los dos píxeles altos del backup hecho inicialmente y se prosigue a procesarlos de igual manera.

- Por último se empaquetan los 4 píxeles a Bytes y se guardan en la imagen destino.

El filtro Motion Blur, aunque al comienzo parecía ser el más complicado, resultó ser más simple de lo que pensábamos.

2.2. Experimentos

Nota: Las mediciones de tiempo (en ciclos) de ejecución de las diversas implementaciones fueron 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 .comment y .^{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 .^{a}bs", 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.

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) Midiendo el Cropflip 10 veces los resultados fueron:

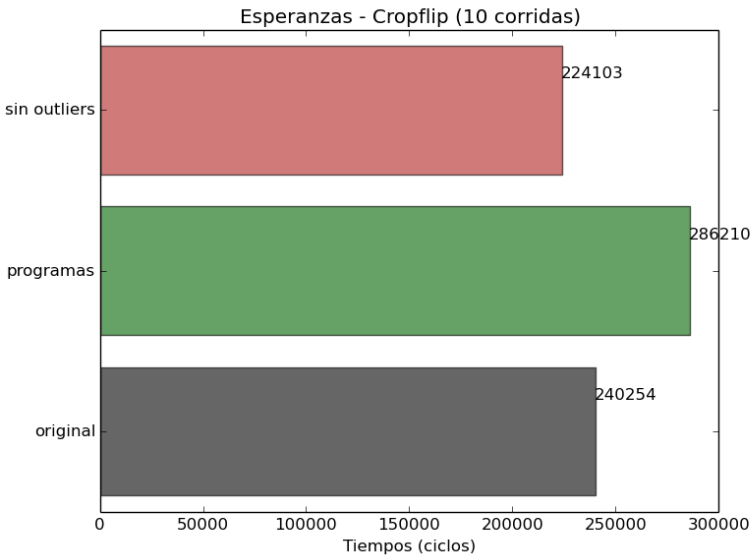
422861, 242136, 186857,
194709, 234097, 221979,
247912, 227976, 223245,
200770.

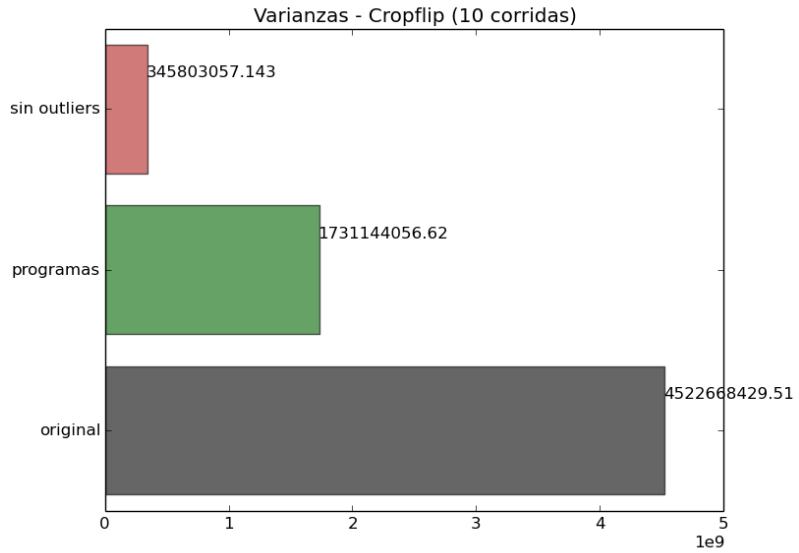
b) Midiendo el Cropflip con programas en simultáneo los resultados fueron:

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) Analizamos las esperanzas y varianzas del filtro sin outliers, con outliers (La medición original) y con los otros programas simultáneos:



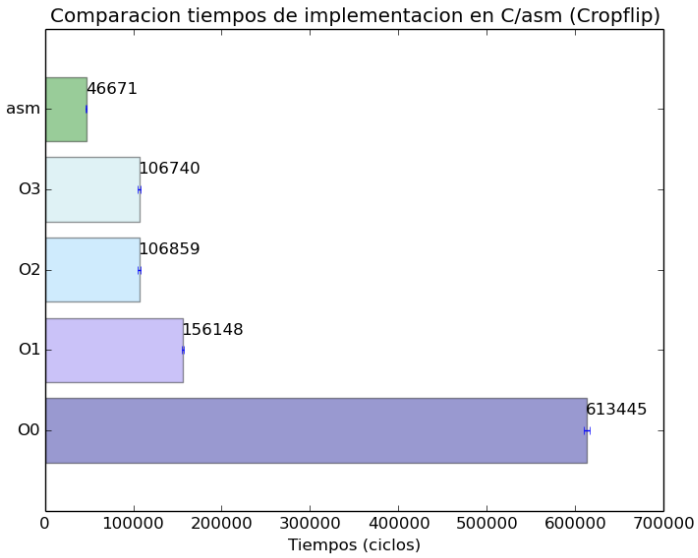


Luego de observar estas mediciones hemos notado una varianza demasiado alta (y, en consecuencia, una falta de confiabilidad en las mediciones), por lo tanto, luego de probar con distintos métodos de medición decidimos tomar el promedio de 1000 mediciones 10 veces, eliminar outliers y calcular luego nuevamente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

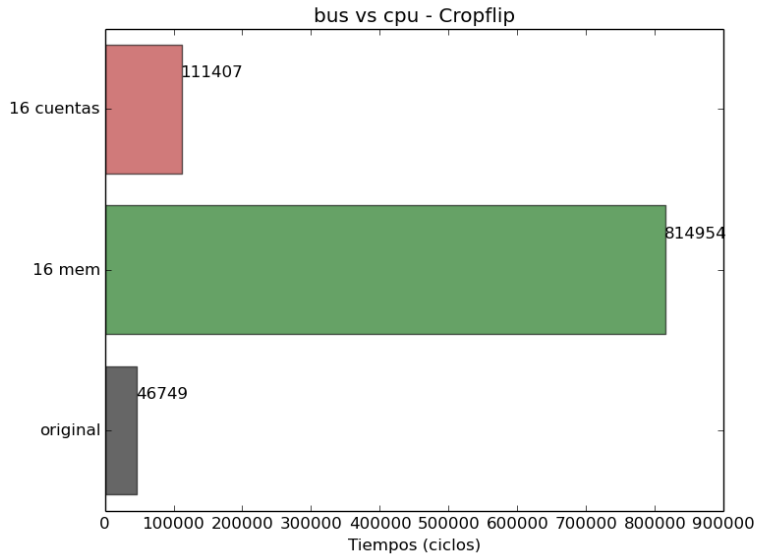
NOTA: las líneas pequeñas azules en el tope de las barras representan el desvío estándar del promedio de las mediciones que estamos mostrando.



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 debido a la limitación que supone operar siguiendo una lógica secuencial en casos en los que resulta mucho más ventajoso trabajar con varios datos al mismo tiempo.

Experimento 1.5) Cropflip - CPU vs bus de memoria

El experimento consistía en agregarle al filtro Cropflip en cada ciclo de la función 4, 8 y 16 cuentas aritméticas, luego 4, 8 y 16 operaciones de memoria y ver como afecta cada una la performance.
Como la función ejecuta pocas instrucciones en cada iteración, con cualquier cosa que le añadimos se pudo ver una diferencia en la cantidad de ciclos de cpu que necesita para completarse.
Para cada test corrimos 10 veces 100 iteraciones del programa obteniendo 10 promedios de tiempo consumido, la función original tarda aproximadamente 46 000 ciclos de procesador. Cuando insertamos 4 operaciones aritméticas el tiempo de procesamiento aumento a 55 000 ciclos aproximadamente, es decir, un 20 % mas. Análogamente, después de añadir 8 operaciones aritméticas, la cantidad de ciclos subió a 65 000 aproximadamente, o sea un 41 %. Finalmente al agregar 16 operaciones aritméticas la función tardó mas de lo esperado, 110 000 ciclos aproximadamente, lo que corresponde a un aumento del 139 % con respecto al filtro original.



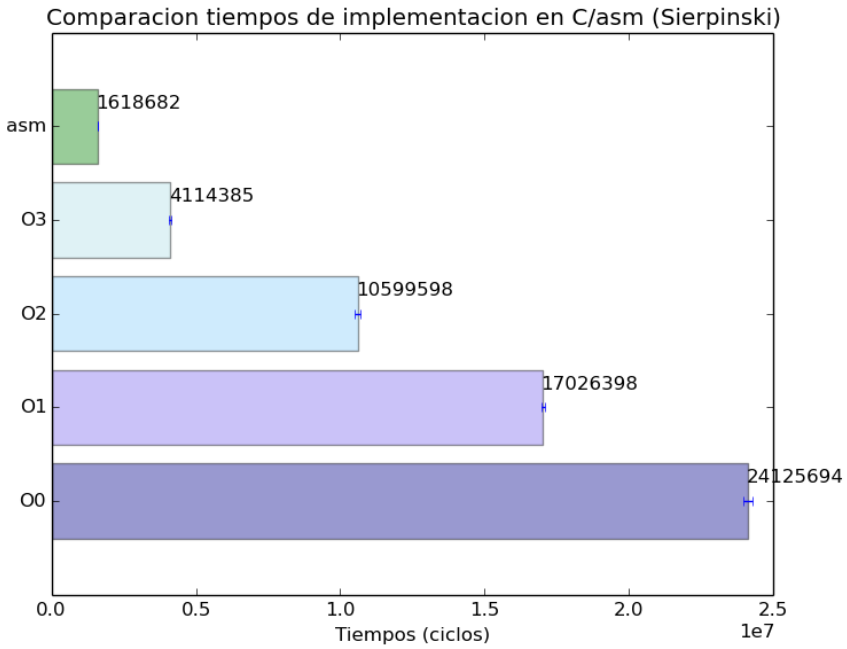
En una primera impresión nos podríamos dejar engañar por los grandes números y porcentajes, sin embrago luego de lembrar que la función cropflip_asm original no hace muchas operaciones, que el filtro itera 1 vez cada 4 píxeles (es decir 65536 veces), y que el procesador sobre el cual fueron hechas las pruebas genera 2 500 000 000 ciclos por segundo, podemos concluir que las cuentas aritméticas de enteros no generan un cuello de botella para el procesador.
A la hora de probar accesos a memoria la cosa fue muy distinta, cuando agregamos 4 el filtro necesitó aproximadamente 220 000 ciclos de reloj para terminar (Es decir un 378 % más).
Con 8 accesos a memoria mas comparado con la función original, la cantidad de ciclos consumidos aumento a 417 000, generando así una suba del 807 %.
Aquí paramos un momento para remarcar algo: Al contrario de lo que esperábamos, la caché no pudo hacer que el porcentaje de incremento sea un poco menor a estrictamente lineal en función a la cantidad de accesos a memoria agregados, de hecho con el doble de accesos a memoria el porcentaje de aumento fue mas que dos veces lo anterior.
Por último cuando probamos el filtro con 16 accesos a memoria los resultados fueron los siguientes, la función requirió de aproximadamente 815 000 ciclos de cpu para finalizar, esto implica una increíble suba del 1672 % (volvemos a notar que el porcentaje incrementado es mas del doble que el anterior).
Luego de ver la suba que implico el agregado de accesos a memoria y después de realizar el mismo test con operaciones de tipo “mov” con registros e inmediatos y obtener casi los mismos resultados (incluso un poquito mejores) que con las operaciones aritméticas, podemos concluir que si bien el procesador es muy rápido para distribuir datos que ya posee, la obtención de datos de la memoria implica un cuello de botella en el procesamiento del filtro, ya que esta acción tarda mucho mas que la mayoría de operaciones que no requieren datos que se encuentren fuera del procesador o de la instrucción.
Retomando el tema del orden de crecimiento de los tests, una posible teoría a este aumento “que no respeta la regla de 3 simple”, es que al usar en todas las mediciones los mismos 2 registros para todas las instrucciones extra, pudo haberse complicado la ejecución fuera de orden del procesador. Luego de notar esto volvimos a realizar las pruebas con mas diversidad de registros en las instrucciones aritméticas agregadas, y notamos un decremento en la tasa de aumento en relación a la cantidad de operaciones añadidas.

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

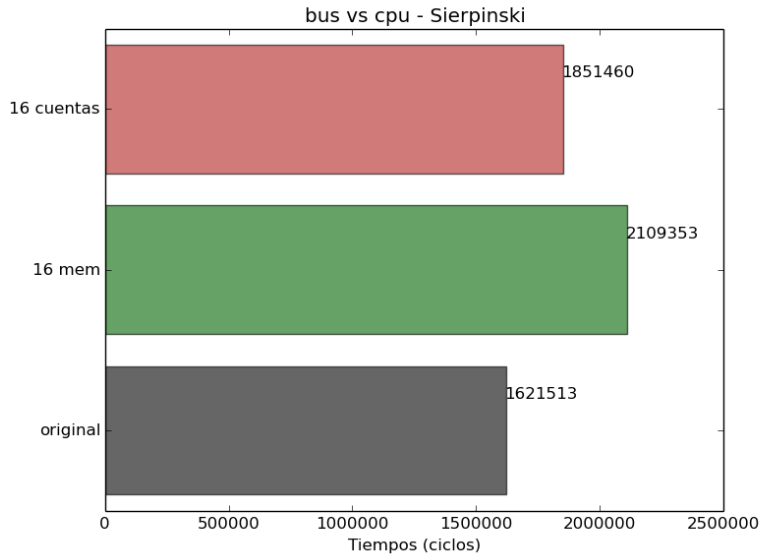


En este experimento la ventaja de usar SIMD siguió notándose con fuerza, mostrando una gran diferencia cuando es contrastado su tiempo de ejecución con el de las distintas optimizaciones de la implementación en C, si bien en este caso se notó que la optimización O3 tiene un gran impacto comparada con la O2, y lo mismo ocurrió al contrastar la O2 con la O1, y la O1 con la O0, suponemos que esto ocurre decido a que las optimizaciones del compilador son muy eficientes al trabajar sobre reducir el peso de operaciones aritméticas (operaciones de las que sierpinski se vale mucho para calcular el coeficiente).

Experimento 2.1) Sierpinski - CPU vs bus de memoria

Este experimento consiste en volver a realizar los tests del item 1.5 pero con la función del filtro Sierpinski. Para realizarlo utilizamos la misma metodología que empleamos al momento de realizar estas pruebas con Cropflip. La función original que computa el filtro de este experimento arroja como resultado un uso

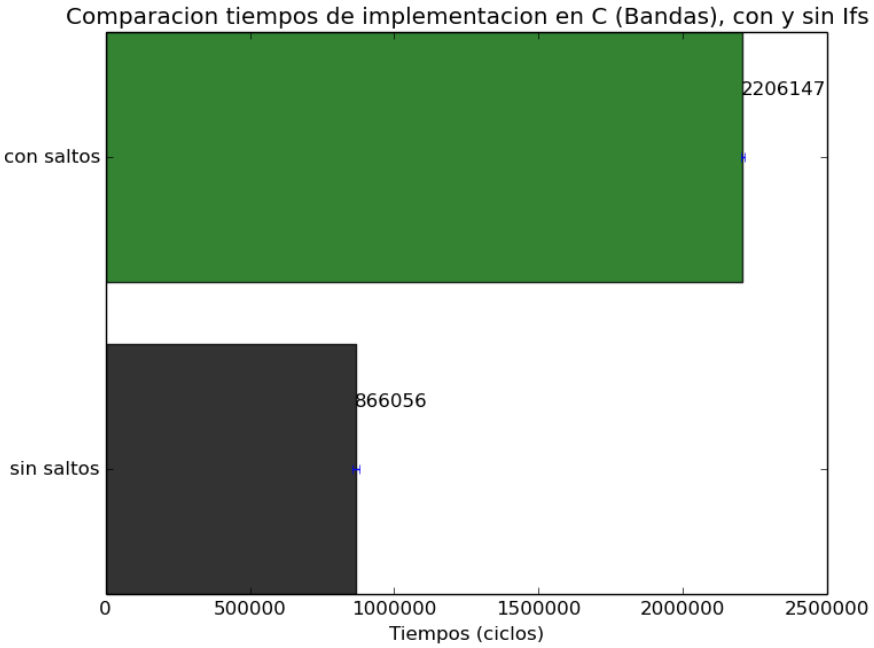
de 1.62M ciclos de cpu (De ahora en adelante utilizamos la letra “M” para simbolizar millones y “K” para indicar miles).
Antes de empezar con los resultados podemos observar que la función a evaluar realiza una cantidad notable mas de operaciones (muchas de ellas de punto flotante) en cada iteración que la previamente testeada, por lo que esperamos obtener resultados diferentes, ya que agregar 4 instrucciones a un programa que posee 4 significa un aumento en la cantidad de operaciones del 100 %, sin embargo, añadirle la misma cantidad a una funcion que tiene 80 implica solo un aumento del 5 %.
Ahora sí, empecemos. Primero realizamos un *benchmark* del filtro habiéndole agregado 4 instrucciones aritmeteicas y la diferencia de performance fue inapreciable, en promedio solo 40K ciclos más que la original, o sea tan solo un 2.47 % mas. Luego probando con 8 operaciones aritmeticas vimos que la función requirió, en promedio, de 1.70M ciclos de cpu, es decir apenas un 4.94 % arriba del filtro original (exactamente el doble que lo anterior).
Por último hicimos el experimento agregando 16 instrucciones de la misma índole que las anteriores añadidas y el test arrojó un resultado de 1.84M de ciclos necesarios para completar la ejecución, esto es un incremento del 13.58 %(Como en el item 1.5, no siempre el doble de operaciones insertadas implica el doble de crecimiento). Luego de realizar todos los tests aritmeticos podemos ver que este tipo de operaciones casi no afectan la performance de la función, luego observando que la misma posee igual cantidad de accesos a memoria que Cropflip, podemos concluir que la performance en promedio no se deteriora mucho luego de agregar instrucciones aritméticas de enteros por la gran cantidad de operaciones de punto flotante y SIMD que tiene el filtro.
Los experimentos con accesos a memoria fueron parecidos, cuando le agregamos 4 a la función esta aumentó su cantidad de ciclos a 1.64M, es decir, tan solo un 1 % más. Después añadimos 8 instrucciones de memoria al filtro original y el número de ciclos de CPU subió a 1.82M, esto implica una suba del 12 %, sorpresivamente 12 veces más que el anterior. Por último, con 16 operaciones de memoria la cantidad de ciclos consumidos fue de 2.10M, un 30 % más que el original. Notamos que el porcentaje de suba no era directamente proporcional con la cantidad de instrucciones agregadas. Nuestra hipótesis sobre este suceso es que como pusimos todas las operaciones extra juntas, el procesador no podia hacer ninguna otra instruccion mientras esperaba que le lleguen los datos de memoria. Para confirmar esta teoría volvimos a realizar el experimento esparciendo las operaciones de memoria entre las instrucciones aritmeticas de la funcion original, y el resultado fue positivo, en todas las pruebas el porcentaje de aumento varió entre 1 % y 2 %.
Finalmente podemos concluir que si bien las operaciones de memoria consumen muchos ciclos esperando los datos, el procesador es capaz de aprovechar ese tiempo realizando instrucciones aritmeticas entre ellas.



2.2.3. Bandas

Experimento 3.1) Bandas - saltos condicionales

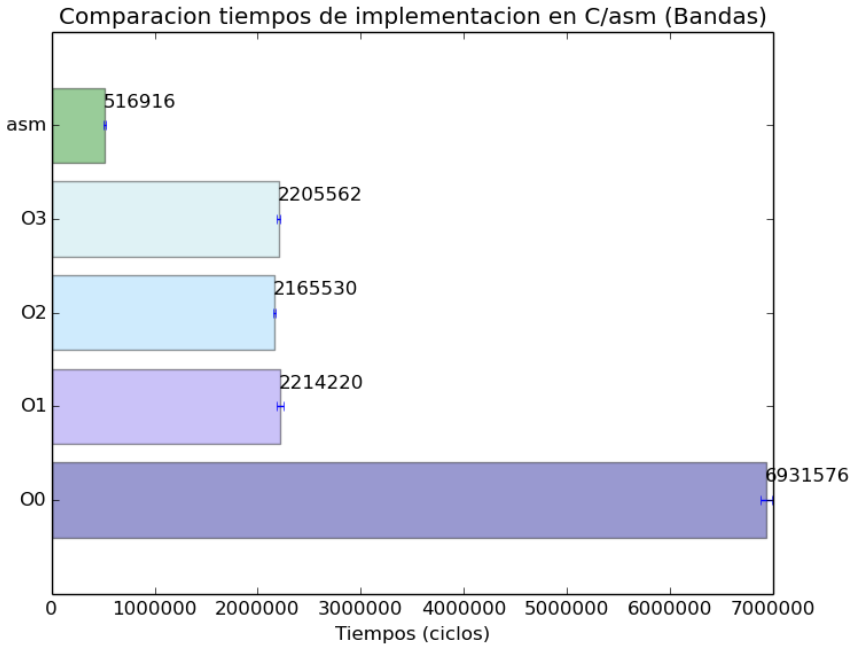
En este experimento corrimos el programa con el flag -O1, en comparación con el mismo programa sin los condicionales. Como explicamos en el punto 1.2, -O1 ofrece ciertas optimizaciones del código que reducen sustancialmente el 'branching', reemplazándolo en lo posible por instrucciones aritméticas. Por 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



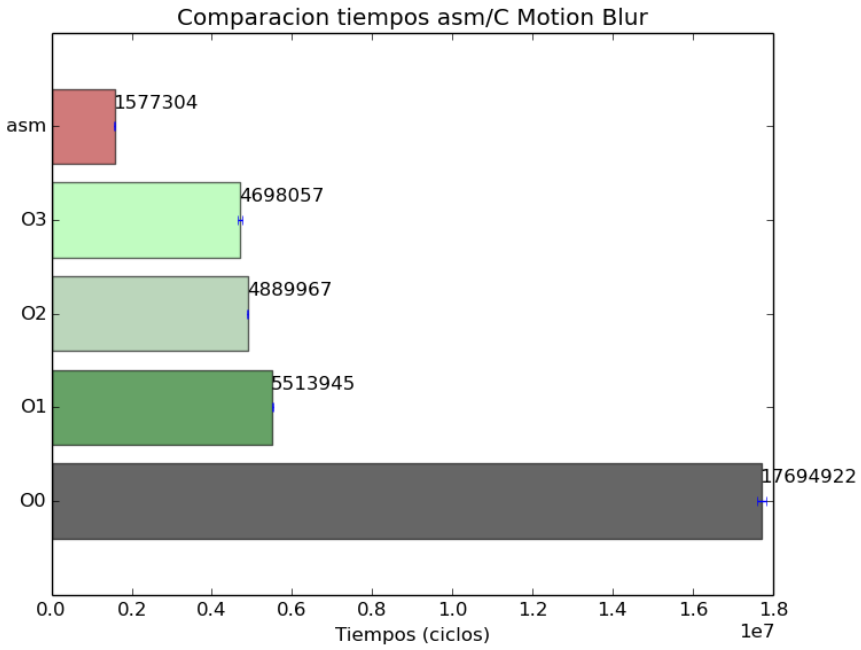
Se puede notar en este gráfico la ineficiencia de la implementación en C, que es en parte remediada cuando se aplican las optimizaciones del compilador, si bien no se consigue obtener resultados cercanos al de la implementación en assembler, ni avanzar mcuho más en las optimizaciones, ya que se ve cómo luego del flag de optimización O1 las siguiente optimizacones parecen haberse estancado, quedando todas en el mismo nivel relativo (suponemos que el O3 supera en tiempo al O2 por poco debido a procesos que puede haber estado corriendo el sistema en el momento de tomar las mediciones, si bien se entiende que todas parecen estar en la misma línea aproximadamente), podríamos arriesgarnos incluso a decir que no creemos que las optimizaciones con los siguientes flags (O4 en adelante) no harán mucho más por mejorar la eficiencia temporal del procesamiento secuencial.

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, evidenciando una clara desventaja (nuevamente) de la implementación en C en comparación con la de assembler, lo que contribuye a afianzar la idea de que el procesamiento vectorial supera con mucho al secuencial (en los casos en que se pueden aprovechar sus ventajas).

3. Conclusiones

C y sus optimizaciones vs ASM

Luego de realizar este Trabajo Práctico podemos concluir con total certeza que, en cuanto a tiempos de ejecución, *Assembler* supera ampliamente a C; y queda en evidencia que, cuanto más cerca se trabaja del código máquina, más óptimo es el resultado.
En cuanto a las optimizaciones, queda claro que C presenta muchos mejores tiempos optimizado que sin optimizar; aunque en algunos casos no se notan grandes diferencias entre O1, O2 y O3 (Incluso llegamos a ver casos en donde O3 estaba en el mismo nivel que O2).
Nuestra conclusión al respecto es que, a pesar de que *ASM* pierde la claridad y el parecido con el lenguaje *"humano"* que puede llegar a tener C, vale la pena conocer los grandes beneficios que tiene programar en el más bajo nivel.
También esto nos genera curiosidad de que tan poco eficientes serán lenguajes de más alto nivel, ya que sacrifican optimizaciones que se podrían aplicar a códigos (desechando procesos que sabemos que no vamos a necesitar en un programa en particular, por ejemplo) y se pierde versatilidad para poder proveer al programador un entorno más amigable y fácil de asimilar.
Esto se debería a que en un punto, al proveer instrucciones de alto nivel que encompasan varias instrucciones de un lenguaje de más bajo nivel, estamos dejando de lado todas las infinitas combinaciones de instrucciones que se podrían utilizar para ejecutar programas de manera distinta, es decir que limitamos las opciones para resolver un determinado problema que tenemos a unas pocas combinaciones de instrucciones determinadas por el lenguaje de alto nivel.

Optimizaciones en el ASM

Luego de programar durante la carrera utilizando variables y condicionales de forma constante y sin pensarlo dos veces, nos encontramos con ciertas realidades bastante duras: Los accesos a memoria y el *branching* son **altamente ineficientes**. Como vimos previamente (Ver experimentos **1.5** y **2.1**), agregar operaciones aritméticas empeoran el tiempo de ejecución, pero los accesos a memoria son extremadamente peores si no son amortiguados como vimos en el experimento **2.1**. Esto nos sirvió bastante para ver por nosotros mismos lo que nos enseñan en teoría desde Organización del Computador 1: La memoria **es realmente lenta** en comparación a los registros del CPU.
En cuanto al *branching*, lo tuvimos que comprobar por nosotros mismos (Ver implementación del Bandas en **2.1.3**): La utilización de saltos condicionales no parece ser gran problema en C, pero en ASM se nota la gran diferencia de performance y claridad en el código. Como ya vimos en el experimento **3.1**, por más que se optimice con O1 (Que reduce la ineficiencia de los saltos condicionales), el código sin saltos tarda tres veces menos que el optimizado.

Conclusiones personales

La idea de este Trabajo Práctico era expandir nuestros conocimientos de *Assembler* adquiridos en el **TP1**, entendiendo como funciona SIMD y de qué manera se puede programar manejando varios datos al mismo tiempo. A su vez, aprendimos a medir en tiempo real nuestros propios programas por primera vez en la carrera, y eso nos empujó a querer superar nuestro propio código varias veces: Una vez que tuvimos terminados los cuatro filtros comenzamos a optimizarlos, mirando la cantidad aproximada de ciclos que tardaban, y pensando de qué manera se podría llegar a un mejor resultado.
Gracias a ese desafío, comprendimos la potencia de algunas instrucciones tales como *pshuf* (Que nos terminó ahorrando varias líneas de código), y aprendimos nuevas maneras de pensar los problemas, como aprovechar el uso de *máscaras* (Que terminó siendo más limpio, claro y rápido que cualquier otra forma de encarar el filtro **Bandas**).
También ganamos una nueva perspectiva en cuanto a la programación en C, conociendo sus optimizaciones y sus debilidades en cuanto a eficiencia. Incluso, conversando con docentes, supimos que aún se puede llevar el asunto a otro nivel, con el uso de herramientas más avanzadas, como el *multithreading*.