Departamento de Computación,
Facultad de Ciencias Exactas y Naturales,
Universidad de Buenos Aires

RTP2

Organización del Computador II

Segundo Cuatrimestre de 2013

Grupo: **Frambuesa a la Crema**

Apellido y Nombre	LU	E-mail
Bálsamo, Facundo	874/10	facundobalsamo@gmail.com
Lasso, Nicolás	763/10	lasso.nico@gmail.com
Rodríguez, Agustín	120/10	agustinrodriguez90@hotmail.com

Índice

1. Introducción

El lenguaje C es uno de los más eficientes en cuestión de performance, pero esto no quiere decir que sea óptimo para todos los casos o que no haya campos en los que pueda utilizarse una opción mejor. Para comprobar esto, experimentamos con el set de instrucciones SIMD de la arquitectura Intel. Vamos a procesar imágenes mediante la aplicación de ciertos filtros y estudiaremos la posible ventaja que puede tener un código en Assembler[1] con respecto a uno en C. Implementaremos los filtros en ambos lenguajes para luego poder comparar la performance de cada uno y evaluar las ventajas y/o desventajas de cada uno.

[1] Durante el desarrollo del informe puede verse referido a Assembler como ASM haciendo un abuso de notación siendo que este es solo el nombre de la extensión.

2. Desarrollo y Resultados

2.1. Filtro Color

Este filtro consiste básicamente en dados un color y una distancia pasados como parámetros, procesa cada pixel de una imagen a color evaluando si el color del mismo se "aleja" más de la distancia del parámetro, y si eso pasa, el píxel se transforma a escala de grises, sino lo mantiene igual. Esto logra el efecto de resaltar un color en una imagen.

2.1.1. Implementación en C

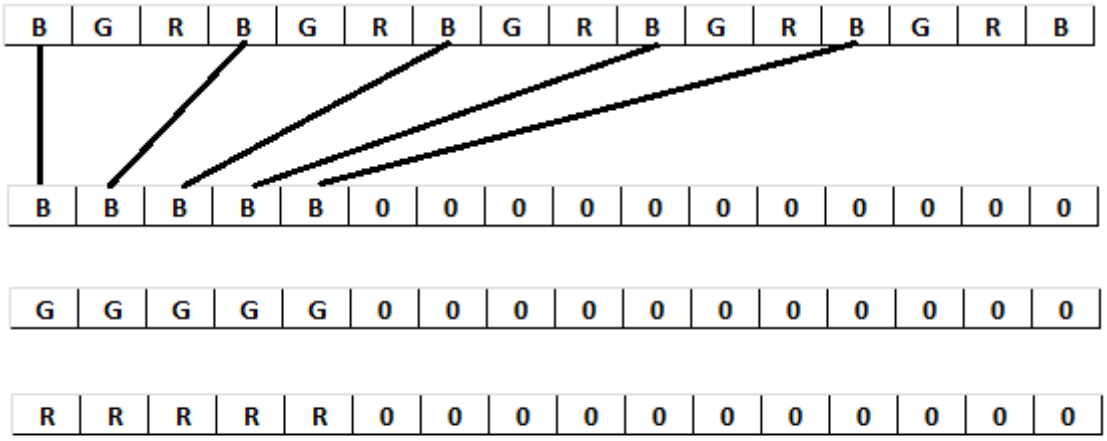
Mediante dos ciclos anidados, se recorre la imagen por cada componente de color de cada pixel. Por cada pixel se levantan sus 3 colores RGB para calcular la distancia a los 3 colores RGB pasados por parámetro. Si el color de la imagen supera esa distancia, entonces en el píxel que se está procesando quedan sus 3 colores iguales unos con otros, logrando convertirse a blanco y negro. Si el color no supera la distancia, debe mantenerse tal cual, logrando así ser resaltado.
Se define la distancia entre colores como:

$$distancia((r,g,b),(rc,gc,bc)) = \sqrt{(r-rc)^2+(g-gc)^2+(b-bc)^2}$$

Para pasar a blanco y negro el píxel, ponemos en cada canal de color (rgb), el mismo valor: $\frac{r+g+b}{3}$.

2.1.2. Implementación en Assembler

Haciendo uso de los registros *XMM* por cada acceso a memoria se pueden traer 16 bytes con lo cual la cantidad de accesos a memoria decrementa considerablemente. Esto provoca que en comparación con el código en C este sea más óptimo ya que el acceso a memoria es de las operaciones que más consumen en cantidad de ciclos haciendo que disminuya la performance. Esto podrá apreciarse en la sección de resultados.[2]
Al levantar los 16 bytes, lo primero que se realiza es copiar estos 16 bytes en otros 2 registros *XMM* para luego ordernarlos mediante la instrucción de Shuffle para que queden de la siguiente forma:



Luego le restamos a cada uno de los datos dentro de nuestros registros, un valor rc, bc o gc que es pasado por parámatro, según se explica al comienzo de esta sección. Una consideración que vale la pena aclarar es que la resta es una diferencia absoluta. Este recaudo se tuvo que tomar dado que los bytes de la matriz son *unsigned char* con lo cual si le restabamos a un valor más chico, un valor más grande, esto podía confundirse y dejarnos un valor que de ser negativo no sería válido pero para nosotros ya que

[2] Ver sección 2.1.3 Item 3 o sección 2.1.4

necesitamos el módulo. Siendo esto, se compara el valor más grande de cada dato dentro del registro y el más chico y se separan ambos en dos registros distintos. Luego se hace una resta del modo MAX - MIN dejando la resta sin signo como queríamos. A modo de ejemplo dejamos la siguiente imagen:

R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC	R - RC

Una vez hecha la resta, se procede a convertir a float ya que las siguientes operaciones son multiplicación, suma y raiz cuadrada para la cual necesitamos convertir nuestros datos a Float. La cuenta realizada por cada píxel es la siguiente:

$$\sqrt{(r - rc)^2 + (g - gc)^2 + (b - bc)^2}$$

Al final de toda la operación dejamos en 2 registros los 5 floats con cada una de estas operaciones en cada pixel. La siguiente imagen queda a forma de entendimiento:

CUENTA	CUENTA	CUENTA	CUENTA

CUENTA	0	0	0

Una vez que tenemos estos valores y luego de convertirlos a INT de tamaño DoubleWord. Mediante el uso de la instrucción PCMPGTD, comparamos que datos son mayores y cuales son menores o iguales al valor pasado por parámetro, *threshold*, generando una máscara con valores 0xffffffff si el resultado de la operación dio *true* y 0x0 en otro caso.
Ya con esta máscara generada, copiamos en 2 registros los valores originales de la matriz, y dejamos, en uno, los valores que queremos procesar usando la instrucción PAND y en otro los valores que queremos dejar como están en el original usando PXOR de la máscara y un PAND contra el registro. De esta manera negamos la máscara y nos quedamos con los otros.
Para terminar, procesamos los datos en los que debemos realizar la siguiente cuenta, para cada píxel:

$$\frac{b+g+r}{3}$$

Por último solo le agregamos los resultados de los bits procesados al registro que tiene los datos originales y lo guardamos en la matriz de destino.

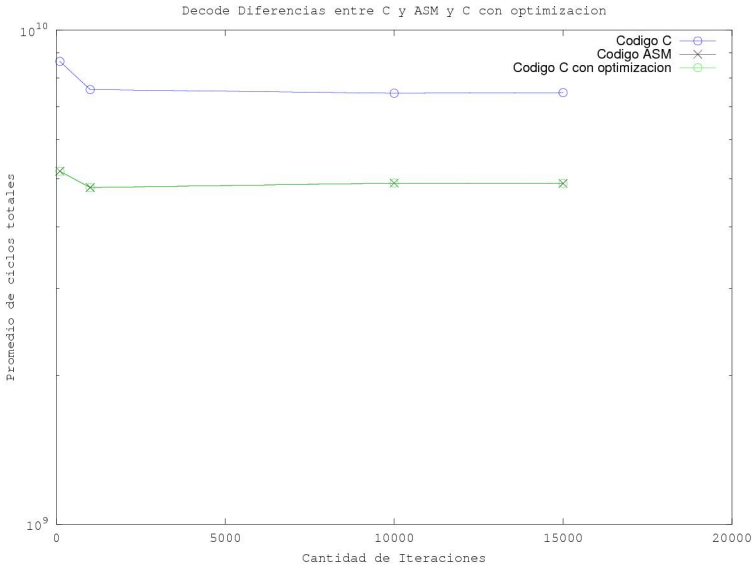
2.1.3. Consideraciones según optimizaciones

- Usando la herramienta *objdump* desensamblamos el archivo de extensión .o de código del filtro de Color. Al observar este código, lo primero que notamos es que el compilador no usó instrucciónes de SIMD a pesar de que el procesador tenga esa característica. Esto se debe a que al escribir en lenguaje C no se puede hacer uso de estas instrucciones a menos que usemos una librería aparte que haga uso de estas como puede ser *libSIMDx86*.
  En cuanto a como se manipulan las variables locales, se respetan las convenciones de pushear registros como r15 - r12 y rbx. Pero en casi todo el código se hace uso de las variables por parámetros y se usa mucho la pila moviendo el registro rbp para recorrerla.

- Existen algunas optimizaciones que se pueden realizar a la hora de compilar el código en C. Estas son -O1, -O2 o -O3, las cuales, siendo agregadas como flags a la hora de compilar, mi código ensamblado queda mucho más óptimo. En particular el flag -O1 hace que el tamaño del código ensamblado sea mucho menor que el código ensablado sin optimización. En particular al mirar el objdump de código con -O1 se pudo apreciar que el código era mucho menor en cantidad de lineas y que la cantidad de registros pusheados a la pila fue mayor.
  El flag -O2 hace todas las optimizaciones que pueda en el código que no estén involucradas con optimizaciones de espacio-tiempo. Y por último el flag -O3 abre todas las opmitizaciones de -O2 más algunas extras con el fin de hacer aun más optimo el código.[3]

[3]Para mas información revisar el http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html

- Haciendo uso de las optimizaciones mencionadas anteriormente, se realizaron experimentos de performance usando el código en ASM, en C y en C compilado con optimizacion -O1, -O2 y -O3. El siguiente gráfico muestra como el código en C optimizado y el ASM son casi idénticos.



Esta medición fue realizada cambiando la cantidad de iteraciones y midiendo cuantos ticks de procesador consume procesar el video completo. El problema de realizar las mediciones de esta manera es que el procesador switchea entre distintos procesos todo el tiempo haciendo que mi contador aumente al ejecutar procesos que no pertenecen a mi función y se cuenta en intervalos muy grandes provocando que la probabilidad de contar ticks de procesos exteriores sea mayor. Es por esta razón que nuestra medición no es lo suficientemente precisa. Una forma de hacerla mas precisa sería evaluando un promedio de la cantidad de Ticks que consume por cada frame de cada iteración. Pero al trabajar en ordenes tan grandes deberíamos procesar demasiada información que no viene al caso de lo que se queire mostrar.

2.1.4. Loop unrolling

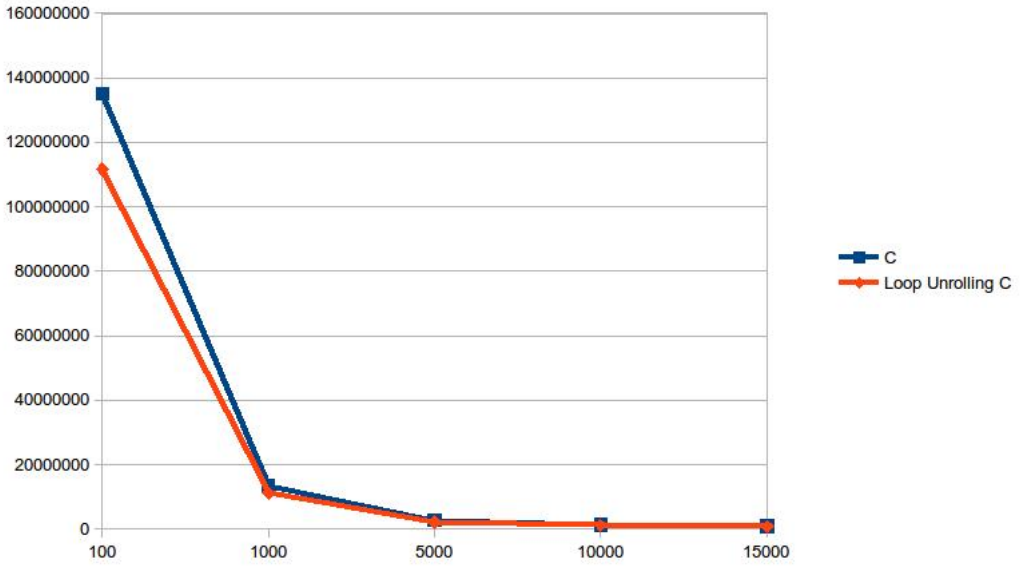
Loop unrolling, o desenrollado de ciclo en español, es una técnica de optimización que consiste en transformar un bucle para mejorar la velocidad de ejecución de un programa. La transformación puede llevarse a cabo por el programador o por un compilador de optimización.
La técnica se basa en eliminar las instrucciones que controlan el bucle, por ejemplo, reduciendo la cantidad de iteraciones que tiene que pasar un ciclo, evitando comparar menos veces si se terminó de iterar o no. Por ejemplo:

Normal loop	After loop unrolling

`int x;`
`for (x = 0; x < 100; x++)`
`{`
`  delete(x);`
`}`
	`int x;`
	`for (x = 0; x < 100; x+=5)`
	`{`
	`  delete(x);`
	`  delete(x+1);`
	`  delete(x+2);`
	`  delete(x+3);`
	`  delete(x+4);`
	`}`

Como podemos ver, el ciclo desenrrollado recorre menos veces, por lo tanto hace menos comparaciones. Pero corre con la gran desventaja de que se genera mucho más código.

Una forma de aplicarlo a nuestro filtro de color en C es procesar más cantidad de píxels. En nuestro filtro C común procesamos de a 1 píxel. Probamos procesar de a 2 píxels, donde el procesamiento de columnas, o de ancho de la imágen se reduce a la mitad. Sin embargo, midiendo velocidades, no detectamos grandes diferencias, ya que esa optimización no es muy significativa o el compilador de C tal vez haga esta optimización.
Luego probamos optimizando más, procesando de a 6 píxels en el C y ahorrando ciertos accesos a memoria. Los resultados fueron más notorios:



El eje x son la cantidad de iteraciones y el eje y es el tiempo que tardó para cierta cantidad de iteraciones.
En velocidad vemos la mejora del loop unrolling, pero en código incrementa considerablemente.

En cuanto al código ASM resulta más complejo hacer esta optimización ya que procesamos de a 5 píxels, lo que entra en un registro XMM.
Lo que probamos fue copiar ese procesamiento de 5 píxels, para procesar idénticamente la misma cantidad dentro del ciclo. Es decir que dentro de un mismo bucle procesamos 5 píxels, y luego otros 5, procesando de a 10 en un mismo bucle. De esta manera nos ahorramos que se ejecuten las dos instrucciones de control del ciclo la mitad de las veces:

```
.ciclo:
CMP rcx, 0
JLE .fin
```

Midiendo tiempos de ejecución nos damos cuenta que no hay grandes diferencias con el ASM implementado sin aplicar esta técnica. Al parecer ahorrar 2 instrucciones básicas de assembler para el tamaño de imágen que probamos (960 x 540), no es significativo.
Vale aclarar que la cantidad de píxels en ancho o en alto debe ser múltiplo de 10, sino habría que tener ciertas consideraciones.

2.1.5. Conversiones y extensiones de datos

En varias secciones del código en assembler de los filtros fue necesario extender o convertir los datos de entrada ya sea para realizar operaciones con mayor precisión o para poder operar con ellos.

Por ejemplo, en el filtro de color la extensión y conversión del dato fue necesaria a la hora de calcular la máscara[4]. La diferencia absoluta pudo calcularse sin ninguna operación extra pero cuando tuvimos

[4] Ver sección 2.1.2

que multiplicar y sumar datos era posible que se superara el número 255, que es el mayor número representable con 8 bits. Por esta razón se extendieron los char de entrada en DoubleWords. Podría haber bastado con extender a Word pero luego necesitamos hacer una operación de raiz cuadrada con lo cual fue más óptimo extender a DoubleWord y luego convertir a Float ya que de esta manera no perdemos precisión en el último cálculo. A continuación, resultó necesario convertir de nuevo a dato Int Double Word. Para esto se usó la instrucción CVTTPS2DQ ya que tiene la particularidad de truncar el número en lugar de redondear al más cercano como hacen otras instrucciones por default y está seteado en el flag 13 y 14 del registro MXCSR[5]. Vale aclararlo ya que a la hora de buscar diferencias entre filtros esto produjo una reducción muy grande de diferencias. Por último en la máscara se comprimió el dato de DoubleWord a Byte. Para la operación real con los datos que son guardados en la salida también fue necesario realizar estas operaciones, se extendió a Int de 32 bits y luego se convirtió a Float todo de una vez para que resulte más óptimo en lugar de primero desempaquetar a Word y luego a DoubleWord. Luego se procedió con las operaciones de suma y división y se empaquetó de nuevo a Byte para insertarlo en la posición de salida.

[5] Ver Manual de Intel volumen 1, sección 10.2.3

2.2. Filtro Miniature

Este filtro consiste en procesar una imagen para lograr un efecto miniatura. Consiste en que los objetos se vean pequeños o como de juguetes. Para esto se "desenfoca" la parte superior y la inferior de la imágen, quedando así sólo el foco en la parte del medio, logrando tal efecto.

2.2.1. Implementación en C

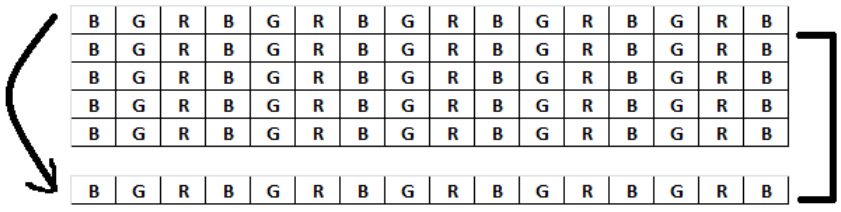
Para esta implementación procesamos la imagen por bandas. Para eso calculamos el límite de las bandas según los parámetros de entrada. Luego recorremos mediante dos ciclos la banda del medio y la dejamos igual, no hacemos ninguna transformación.
Luego dentro de un ciclo que cuenta las iteraciones, hacemos el procesamiento de "desenfoque" de la banda superior primero y luego la inferior.
Para desenfocar una banda, se recorre mediante dos ciclos que me permiten levantar cada color de cada pixel. Teniendo un componente de color de un pixel, se calcula el nuevo valor que debe tener el mismo. Para este cálculo se procesa la subimagen que hay alrededor del píxel, haciendo el cálculo por color, la multiplicamos por la matriz M, y vamos acumulando ese producto.
Para evitar la saturación, al resultado lo dividimos por 6, que es la suma de las componentes de la matriz M. Con el nuevo valor obtenido, lo pasamos a la imágen resultante. Seguimos procesando hasta finalizar la banda. Para incrementar el efecto, lo que hacemos es, por iteración, guardar el píxel desenfocado, para que en la siguiente iteración lo procese otra vez.

$$M = \begin{pmatrix} 0{,}01 & 0{,}05 & 0{,}18 & 0{,}05 & 0{,}01 \\ 0{,}05 & 0{,}32 & 0{,}64 & 0{,}32 & 0{,}05 \\ 0{,}18 & 0{,}64 & 1 & 0{,}64 & 0{,}18 \\ 0{,}05 & 0{,}32 & 0{,}64 & 0{,}32 & 0{,}05 \\ 0{,}01 & 0{,}05 & 0{,}18 & 0{,}05 & 0{,}01 \end{pmatrix}$$

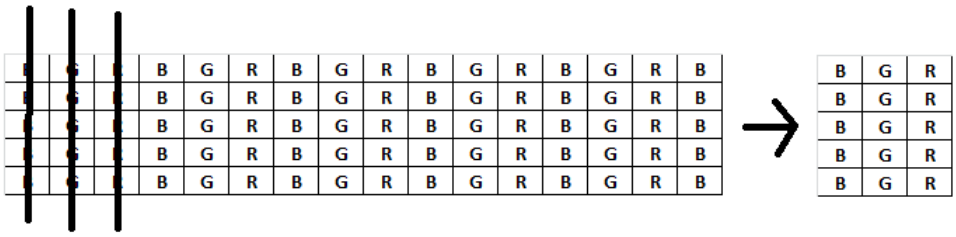
2.2.2. Implementación en Assembler

Para la implementación en assembler de este filtro, la diferencia mas sifnificativa contra la implementación en C, fue la forma de levantar los registros. Usando los registros *XMM* se levantaron 5 registros de 16 bytes cada uno de forma que al procesar la matriz se pudo reducir la cantidad de accesos a memoria y así mejorar la performance. Se consideraron dos opciones a la hora de diseñar este filtro:

- Levantar de a 5 registros y avanzar bajando de a una fila y utilizando los 5 registros anteriores dejando 4 iguales y ahorrandome 4 accesos y pisando el primer registro con la nueva fila de abajo. La imagen muestra la idea:



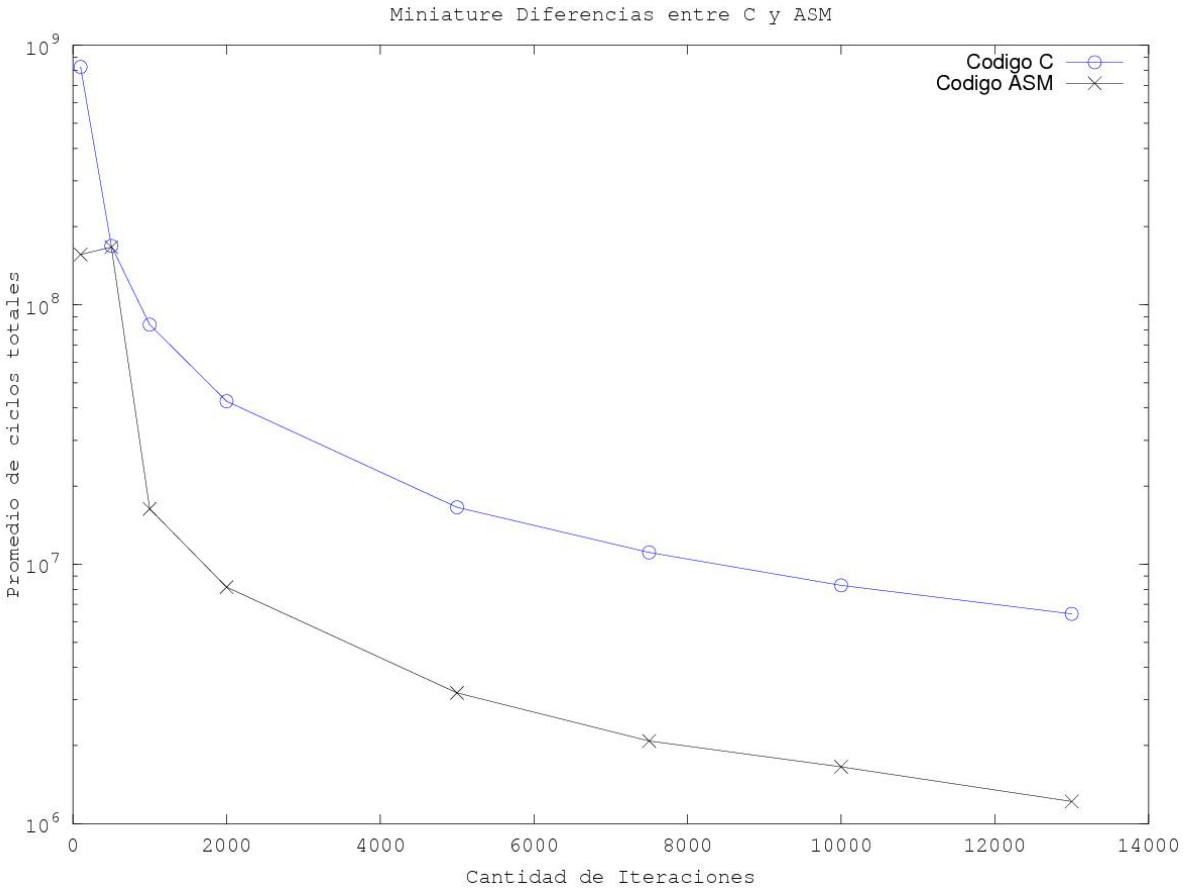
- El otro propuesto fue, levantar esos 5 registros pero en lugar de mover 1 registro hacia abajo, aprovechar la cache del procesador moviendonos hacia la derecha ya que cuando copiamos de memoria un registro de 16 bytes traemos con el mucha mas información que queda en cache con lo cual al movernos hacia de esta manera estamos aprovechando los HIT de la cache en lugar de un MISS por cada iteración del ciclo. A continuación una imagen de lo propuesto.



Finalmente se opto por la segunda opción por cuestiones de performance. El resto de las cuestiones estructurales como ciclos y operaciones son muy similares al C.

2.2.3. Resultados

Comparando ambas implementaciones, notamos que estructuralmente no cambian mucho. En ambas tenemos la misma idea de ir recorriendo las bandas por un lado e ir haciendo la cuenta para el nuevo píxel por otro. De todas formas, lo que se destaca es que en ASM tenemos menos accesos a memoria, ya que procesamos de a 5 píxels, mientras que en C lo hacemos de a uno.
Además se puede apreciar que el ensamblado del compilador de C, al no usar SIMD se producen más comparaciones que el ASM para terminar de recorrer la imágen. En cuanto a la performance confirmamos lo que habíamos supuesto, el ASM resultó ser más rápido que el C.



Como podemos ver en el gráfico, lo que tarda la ejecución en promedio, el C se mantiene a la misma distancia del ASM a medida que aumentan las iteraciones, siempre tardando más tiempo.

Experimentamos con varios tamaños de imágenes y pudimos ver que la diferencia en velocidad no cambia mucho. El C sigue tardando más que el ASM, y con imágenes más grandes hasta puede tardar

más que lo que vimos en el gráfico anterior.
Consideramos que a la hora de hacer este tipo de filtros, lo más prioritario para optimizar es evitar los accesos a memoria lo más posible, ya que como sabemos es una de las cosas más costosas en cuanto a performance.

2.3. Decodificación Esteganográfica

El filtro Decode, obtiene un mensaje a partir de los bytes de la entrada, procesándolos primero y luego quedándose con los bits relevantes para la decodificación.

2.3.1. Implementación en C

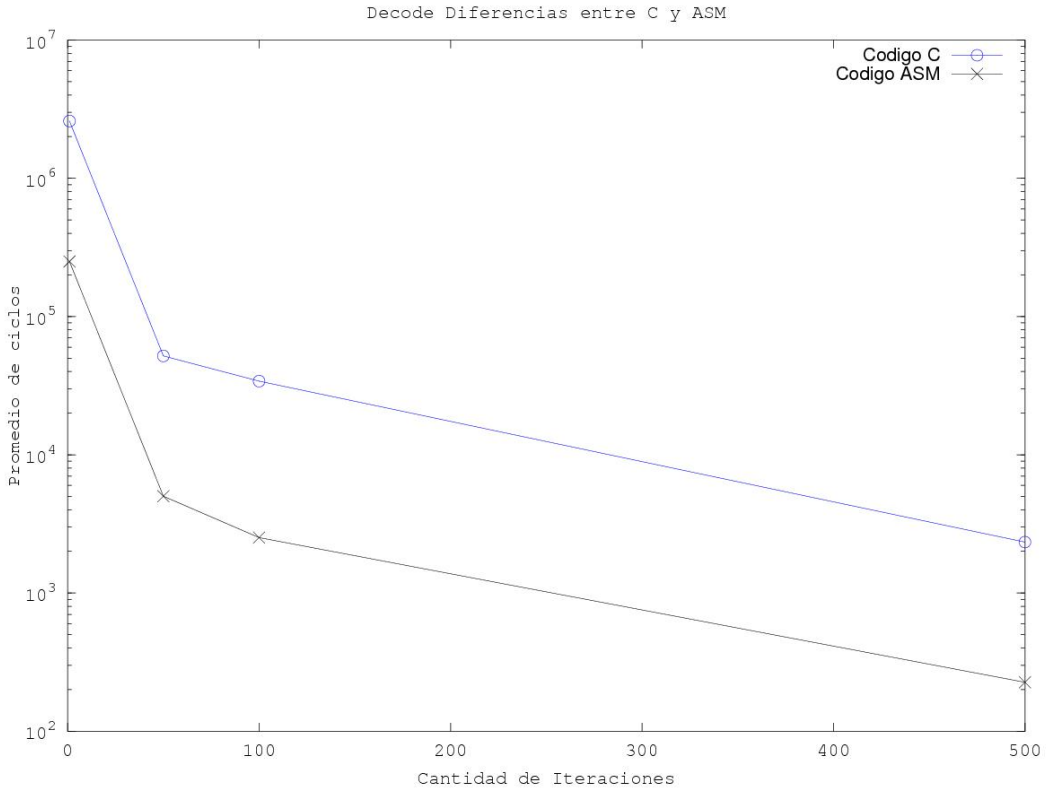
En un ciclo principal, se recorren los bytes de la fuente, hasta terminar de recorrer la imagen o alcanzar el tamaño recibido por parámetro. Dentro de este ciclo hay otro, que recorre de a 4 bytes. Los obtiene primero el código de la operación a aplicar, luego los bits que serán procesados y luego se realiza el algoritmo correspondiente. En cada iteración del ciclo interior, en una variable que contiene el byte decodificado, inserta los bits procesados en la posición que les corresponde. Al terminar con el cuarto byte, pega ese valor en la salida y avanza los punteros y variables para continuar con el siguiente grupo de bytes.

2.3.2. Implementación en Assembler

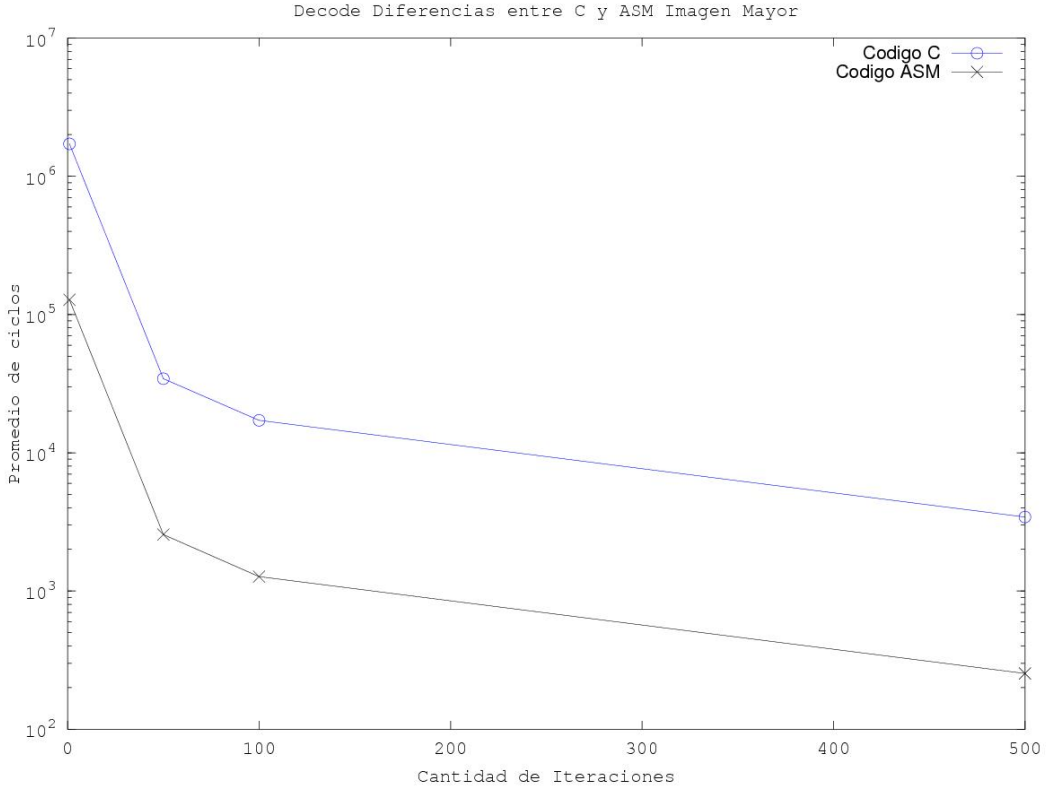
El ciclo del algoritmo obtiene en primer lugar 16 bytes de la fuente. Luego, con una máscara obtengo los bits 2 y 3 de cada byte en un registro. Luego con otras máscaras, se obtienen los bytes a los que se les debe sumar 1 en un registro y a los que se debe restar 1 en otro. El próximo paso, niega los bits de los bytes cuyo código de operación es 3. A ese resultado, se le suma 1 al registro usando la máscara obtenida anteriormente, y lo mismo se hace para restar 1 usando la otra máscara. Después de eso, se borran todos los bits que no interesan, dejando solo los bits 0 y 1 de cada byte.
Para juntarlos y armar los bytes decodificados, en diferentes registros, se dejan solo los pares de bits que van en las posiciones 0 y 1, 2 y 3, 4 y 5 , 6 y 7. Luego, con PSHUFB y las máscaras correspondientes, llevo esos bits manteniendo sus posiciones, pero a los primeros 3 bytes del registro. Y finalmente con la instrucción POR se guardan en un solo registro todos los bits en su posición correspondiente. Finalmente se guarda ese registro en la posición del output. Como solo los primeros 3 bytes eran reelevantes para el output, el puntero se avanza solo 3 lugares. En la fuente en cambio, aumento 12 lugares la posición. Esto se hace así porque cada píxel ocupa 3 bytes, y para formar un byte de la salida se necesitan 4 bytes. Avanzar de a 12 la fuente y de a 3 el destino entonces, es conveniente para avanzar de forma más ordenada y segura.

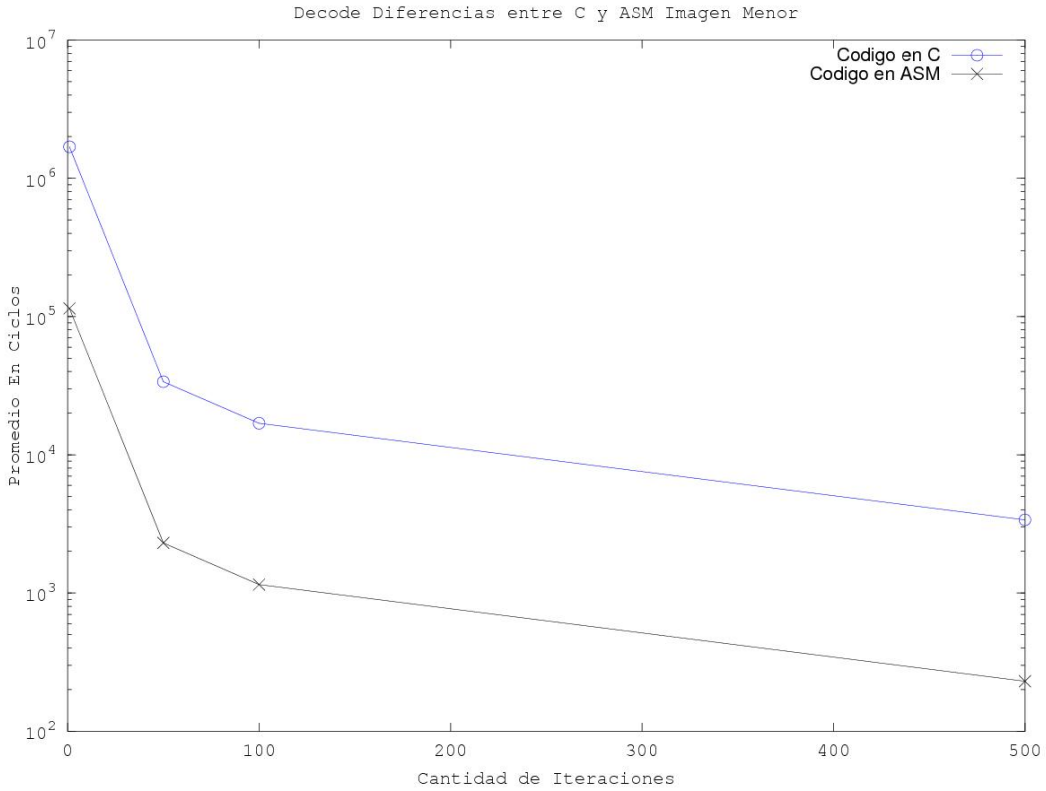
2.3.3. Resultados

La diferencia principal entre el código escrito en lenguaje C y el código ASM, es la cantidad de accesos a memoria realizados en cada iteración. Otra diferencia puede verse en la cantidad de bytes que pueden ser procesados simultaneamente durante el mismo ciclo. Mientras que en C leemos y procesamos un byte por vez, el código ASM nos permite acceder y procesar 16 bytes en simultaneos. De todas maneras, en este caso trabajamos con 12 bytes que es un numero mas útil ya que es múltiplo de 3 (cantidad de bytes por píxel) y de 4 (cantidad de bytes por cada caracter).
Pudiendo levantar de a 16 bytes en memoria, podría esperarse que el código asm demorase una diesiseiava parte de la cantidad de ciclos que demora el código en C. Pero debemos tener en cuenta que en nuestro caso estamos aprovechando solo 12 de los 16 bytes que leemos en cada iteración por lo tanto es más acertado evaluar como si solo leyeramos 12 bytes. Además, hay que tener en cuenta que las instrucciones SSE pueden necesitar más ciclos que las operaciones comunes que usamos en el C, eso disminuye un poco la performance en la comparación. Viendo el promedio de ciclos necesitados por cada código, estamos en la mayoría de los casos en cifras cercanas a 10.35, por lo tanto, concluimos que para la mayoría de los casos, esa es la relación de performance entre los códigos.



Probando con imágenes de distintos tamaños, una de 1920 X 1080 píxeles y otra de 640 x 480 píxeles, sucedió en ambos casos que hubo un aumento en la diferencia entre los códigos, pero se mantuvo, para las dos imágenes y, para distinta cantidad de iteraciones, un promedio de aproximadamente, por lo que no presentan grandes cambios con la imagen original.

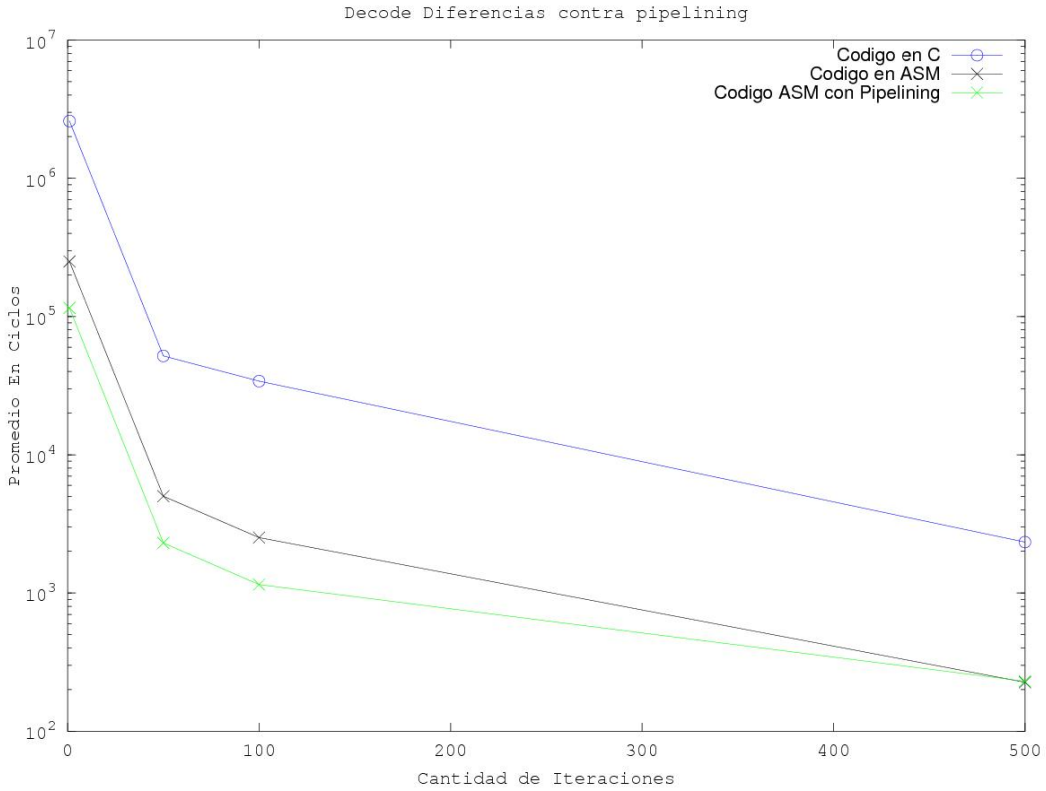




2.3.4. Decode con Pipelining

Para desarrollar el código con la tecnica de pipelining, lo primero que modificamos fue el uso de los regitros. Intentando hacer, cada sección del código, con registros distintos para disminuir la dependencia entre las instrucciones. Una vez logrado esto, el aumento de performance fue muy significativo. Realizar una iteración del código nos costaba antes aproximadamente 250200 ciclos del clock y, con la nueva técnica logramos reducir el tiempo a aproximadamente 115362 ciclos.
Luego intentamos intercalar algunas instrucciones para reducir aun más la cantidad de ciclos necesarios, pero no se lograron cambios apreciables.

En cuestion de performance, se puede apreciar en el siguiente gráfico, que se ha logrado un salto en la cantidad de ciclos bastante importante. En el código C, en comparacion con el ASM sin pipelining, demoraba aproximadamente 10.36 veces más. La diferencia se lograba gracias a la posibilidad de leer de la memoria de a 16 bytes por cada acceso en lugar del único que podiamos leer en C. Luego, una vez utilizada la técnica de pipelining, podemos apreciar que la diferencia es aun mayor, logrando que en C se demore aproximadamente 22.48 veces más que utilizando pipelining en ASM. Y el ASM sin pipelining demora aproximadamente 2.17 veces más que el que si utiliza esta técnica.


Decode Diferencias contra pipelining
Codigo en C
Codigo en ASM
Codigo ASM con Pipelining
Promedio En Ciclos
Cantidad de Iteraciones
10^7
10^6
10^5
10^4
10^3
10^2
0
100
200
300
400
500

3. Conclusión

Las instrucciones SIMD (Single Instruction Multiple Data) proveen al programador de una herramienta más efectiva para realizar el mismo conjunto de operaciones a una gran cantidad de datos.

La aplicación de filtros a imágenes era un ejemplo perfecto para probar su eficiencia.

Analizando los resultados de las implementaciones de los 3 filtros, podemos notar:

- Las operaciones básicas (padd, psub, pmul, pdiv, shifts, etc.) SIMD tienen un costo similar a sus correspondientes operaciones unitarias, pero generalmente requieren algún tipo de pre-proceso para poder trabajar con los 16 bytes (pack, unpack, shifts) en una sola iteración, por lo tanto, aunque más eficientes, no lo son en una relación directamente proporcional.
- En el caso que sí hay una relación directamente proporcional es en el acceso a memoria.
- Además, el acceso a memoria es, por lejos, la operación más costosa de las que implementamos en cada filtro.
- Por consecuencia directa del ítem anterior, las llamadas a otras funciones (que a su vez, probablemente contengan variable locales) dentro de una iteración provocan estragos en la efectividad de las implementaciones en C.
- Para poder aprovechar las instrucciones SIMD es un prerequisito que los datos estén contiguos en memoria.

Concluimos que, definitivamente, las instrucciones SIMD, cuando pueden aprovecharse, demuestran una gran eficiencia. Sin embargo, hay que tener algunas consideraciones:

Aunque las imágenes, video y sonido son los primeros candidatos a ser optimizados por paralelización, no todos los procesos pueden ser efectivos y se requiere un análisis profundo de los datos para ver si vale el esfuerzo.

Además, aunque se pueda lograr una gran optimización, no siempre es lo más importante. La optimización seguramente es indispensable en transmisiones de video en vivo, pero baja en importancia si tuviese que ser aplicado una sola vez en una aplicación tipo MS Paint.

Las desventajas que podrían opacar a la optimización son:

El código no es portable, únicamente funciona en procesadores que implementan el set de instrucciones AMD64, requiriendo reescrituras para otras plataformas. Sin embargo el código C debería funcionar perfectamente en IA-32, ARM y cualquier otro procesador que tenga un compilador de lenguaje C.

El código es mucho más largo y difícil de entender (por lo tanto mayor posibilidad de tener bugs) que en un lenguaje de más alto nivel como C. Y en pos de la optimización, se llegan a eliminar funciones (poniéndolas inline), lo que genera código repetido, largo y confuso.