



DEPARTAMENTO  
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA



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

<b>1. Introducción</b>	<b>3</b>
<b>2. Desarrollo y Resultados</b>	<b>4</b>
2.1. Filtro Color . . . . .	4
2.1.1. Implementación en C . . . . .	4
2.1.2. Implementación en Assembler . . . . .	4
2.1.3. Consideraciones según optimizaciones . . . . .	5
2.2. Filtro Miniature . . . . .	7
2.2.1. Implementación en C . . . . .	7
2.2.2. Implementación en Assembler . . . . .	7
2.2.3. Resultados . . . . .	7
2.3. Decodificación Esteganográfica . . . . .	8
2.3.1. Implementación en C . . . . .	8
2.3.2. Implementación en Assembler . . . . .	8
2.3.3. Resultados . . . . .	8
<b>3. Conclusión</b>	<b>9</b>

## 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 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.

## 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 pixel se transforma a escala de grises, sino lo mantiene igual. Esto logra el efecto de resaltar un color en una imagen

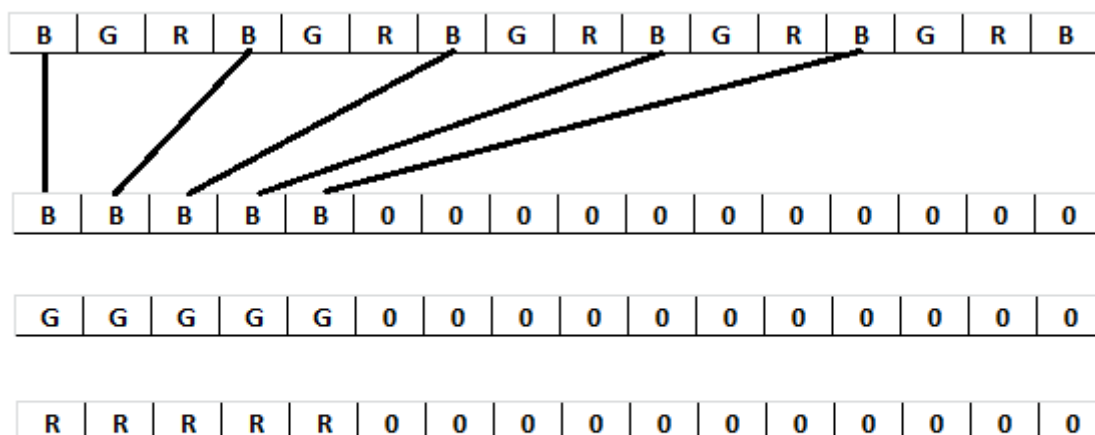
#### 2.1.1. Implementación en C

Mediante dos ciclos anidados, se va recorriendo 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 pixel que se está procesando quedan sus 3 colores iguales, logrando una escala de grises. Si el color no supera la distancia, debe mantenerse tal cual, logrando así ser resaltado.

#### 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, haciendo que en comparación con el código en C este sea mas optimo ya que el acceso a memoria es de las operaciones que mas consumen en cantidad de ciclos haciendo que disminuya la performance. Esto podrá apreciarse en la sección de resultados.

Al levantar los 16 bytes, lo primero que realizo es copiar estos 16 bytes en otros 2 registros *XMM* para luego ordenarlos 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 parámetro, 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 restabamos a un valor mas chico un valor mas grande esto podía confundirse y dejarnos un valor que de ser una resta con sino sería válido pero para nosotros no era util. Siendo esto, se toma compara el valor más grande de cada dato dentro del registro y el mas 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	R - RC
--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------	--------

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

<b>CUENTA</b>	<b>CUENTA</b>	<b>CUENTA</b>	<b>CUENTA</b>
<b>CUENTA</b>	<b>0</b>	<b>0</b>	<b>0</b>

Un vez que tenemos estos valores y luego de convertirlos a INT de tamaño Double word. Mediante el uso de la instrucción PCMPGTD, comparamos que datos son mayores y cuales son menores o iguales al valor pasado por parametro *threshold* generando una mascara 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 que debemos realizar la siguiente cuenta en para cada pixel:

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

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.

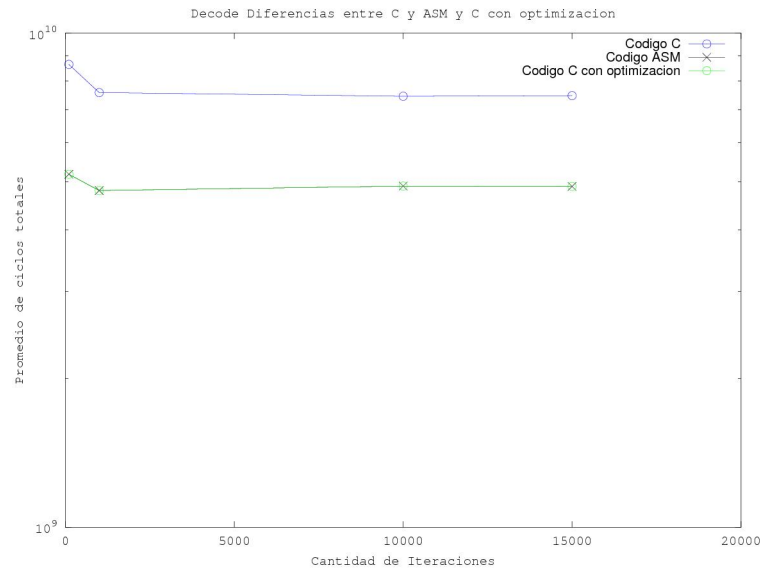
### 2.1.3. Consideraciones según optimizaciones

- Usando la herramienta *objdump* desensamblamos el archivo .o de código del filtro de Color. Al observar este código, lo primero que notamos es que el compilador no usó instrucciones 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 operaciones 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 parametros 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 ensamblado sin optimización. En particular al mirar el *objdump* de código con -O1 se pudo apreciar que el código era mucho menor en cantidad de líneas 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 optimizaciones de -O2 mas algunas extras con el fin de hacer aun más óptimo el código.<sup>1</sup>
- Haciendo uso de las optimizaciones mencionadas anteriormente, se realizaron experimentos de performance usando el código en ASM, en C y en C compilado con optimización -O1, -O2 y -O3. El siguiente gráfico muestra como el código en C optimizado y el ASM son casi idénticos.

<sup>1</sup>Para mas información revisar el <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>



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 quiere mostrar.

## 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" una parte superior y otra inferior de la imagen, quedando así sólo el foco en la parte del medio, logrando tal efecto.

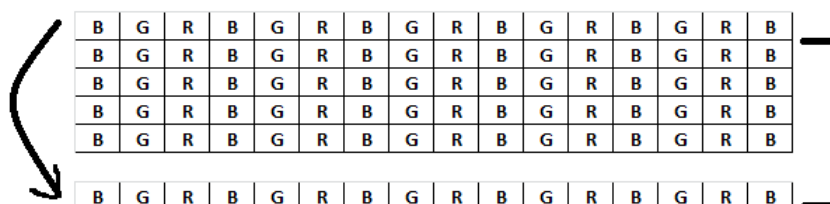
### 2.2.1. Implementación en C

Para esta implementación procesamos la imagen por bandas. Para eso calculamos el límite de las bandas. 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, hago el procesamiento de "desenfoco" 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, calculo 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 imagen resultante. Seguimos procesando hasta finalizar la banda. Para incrementar el efecto, lo que hacemos es, por iteración, guardar el pixel desenfocado, para que en la siguiente iteración lo procese otra vez.

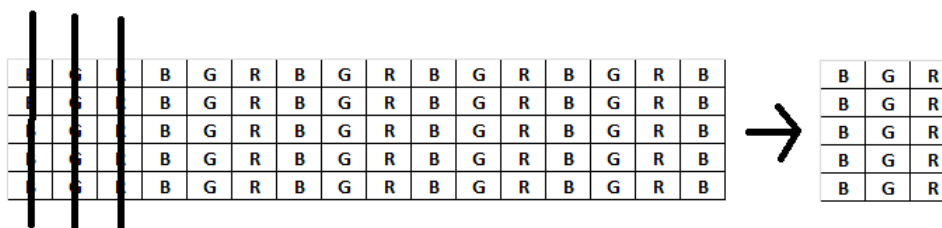
### 2.2.2. Implementación en Assembler

Para la implementación en assembler de este filtro, la diferencia mas significativa 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 ahorrando 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.



### 2.2.3. Resultados

## 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 relevantes 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 pixel 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



### 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 del 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 preproceso 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 prerequisite que los datos estén contiguos en memoria. Como descubrimos con el filtro Rotar, los datos dispersos nos obligan a hacer múltiples lecturas a memoria y perder tiempo reordenándolos dentro de los registros antes de poder procesarlos.

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. Ninguno de los filtros implementados demoró más de 1 segundo en ejecutarse completamente. 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.