


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 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 imágen 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 imágen

### 2.1.1. Implementación en C

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

### 2.1.2. Implementación en Assembler

### 2.1.3. Resultados

## 2.2. Filtro Miniature

Este filtro consiste en procesar una imágen para lograr un efecto miniatura. Consiste en que que los objetos se vean pequeños o como de juguetes. Para esto se "desenfocaüna parte superior y otra 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. 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 "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, 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 saturaciión, 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 pixel desenfocado, para que en la siguiente iteración lo procese otra vez.

### 2.2.2. Implementación en Assembler

### 2.2.3. Resultados

## 2.3. Decodificación Esteganográfica

Explicacion decode.

#### 2.3.1. Implementación en C

#### 2.3.2. Implementación en Assembler

#### 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 prerequisito 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.