



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

1. Introducción	3
2. Desarrollo y Resultados	3
2.1. Filtro Color	3
2.1.1. Implementación en C	3
2.1.2. Implementación en Assembler	3
2.1.3. Resultados	4
2.2. Filtro Miniature	4
2.2.1. Implementación en C	5
2.2.2. Implementación en Assembler	5
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	9
3. Conclusión	9

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.

2.1.1. Implementación en C

2.1.2. Implementación en Assembler

2.1.3. Resultados

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

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

Analizando los resultados de las implementaciones de los 6 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 (como se comprobó en el filtro recortar).
- 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.