Trabajo Práctico 2
Procesamiento de imágenes

Organización del Computador II

Primer Cuatrimestre 2011

1. Introdución teórica

La detección de bordes en una imagen reduce significativamente la cantidad de información de la misma, filtrando lo innecesario y preservando sus características fundamentales. De allí que los métodos para detectar bordes sean una herramienta muy poderosa para el procesamiento de imágenes. Podemos definir como un borde a los píxeles donde la intensidad de la imagen cambia de forma abrupta. Si consideramos a la imagen como una función de intensidad, $I(x, y)$, entonces lo que buscamos son saltos en dicha función. Para simplificar, estudiaremos el caso de imágenes en escala de grises.

El gradiente de una imagen en la posición (x, y) viene dado por el vector:

$$\nabla I = \begin{bmatrix} I_x \\ \\ I_y \end{bmatrix} = \begin{bmatrix} \frac{\partial I}{\partial x} \\ \\ \frac{\partial I}{\partial y} \end{bmatrix} \tag{1}$$

El vector gradiente siempre apunta en la dirección de la máxima variación de la imagen I en el punto (x, y). Para los métodos de gradiente nos interesa conocer la magnitud de este vector, denominado simplemente como gradiente de la imagen, denotado por $||\nabla I||$ y dado por:

$$||\nabla I|| = \sqrt{I_x^2 + I_y^2} \tag{2}$$

Esta cantidad representa la variación de la imagen $I(x, y)$ por unidad de distancia en la dirección del vector ∇I. En general, el gradiente de la imagen se suele aproximar mediante la expresión

$$||\nabla I|| \approx |I_x| + |I_y| \tag{3}$$

que es mucho más simple de implementar computacionalmente.

Para el cálculo del gradiente de la imagen en cada píxel tenemos que obtener las derivadas parciales. Las derivadas se pueden implementar digitalmente de varias formas. Una forma

es mediante máscaras de tamaño 3×3 o incluso más grandes. Por otro lado, las máscaras normalmente tienen tamaños impares, de forma que los operadores se encuentran centrados sobre el píxel en el que se calcula el gradiente.

Los operadores de **Roberts**, **Prewitt**, **Sobel** y **Frei-Chen** son operadores dobles o de dos etapas. La detección de bordes se realiza en dos pasos. En el primero se buscan bordes horizontales utilizando la máscara que corresponde a la derivada parcial en x. En el segundo paso se buscan los bordes verticales utilizando la máscara que corresponde a la derivada parcial en y. Finalmente, se suman ambos para obtener el gradiente de la imagen en cada pixel. En la figura 1 se pueden ver los operadores de **Roberts**, **Prewitt**, **Sobel** y F**rei-Chen** para determinar bordes horizontales (izquierda) y verticales (derecha).

1	0
0	-1

0	1
-1	0

(a)

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

(b)

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

(c)

-1	0	1
$-\sqrt{2}$	0	$\sqrt{2}$
-1	0	1

-1	$-\sqrt{2}$	-1
0	0	0
1	$\sqrt{2}$	1

(d)

Figura 1: Operadores de derivación: (a) de Roberts, (b) de Prewitt, (c) de Sobel y (d) de Frei-Chen.

Por último, dijimos que íbamos a simplificar el problema tratando sólo con imágenes en escala de grises. Para eso, debemos contar con una función que sea capaz de monocromatizar una imagen a color. La función más sencilla para hacer esto es:

$$f(p) = \sqrt[\epsilon]{\alpha R^\epsilon + \beta G^\epsilon + \gamma B^\epsilon} \tag{4}$$

La función f se aplica a cada píxel p de la imagen; R, G, B son sus componentes de color, α, β y γ son coeficientes entre 0 y 1 y ϵ es un exponente entre 1 e ∞.

Los resultados de aplicar los filtros y de pasar a escala de grises son los siguientes:

Figura 2: Imagen original

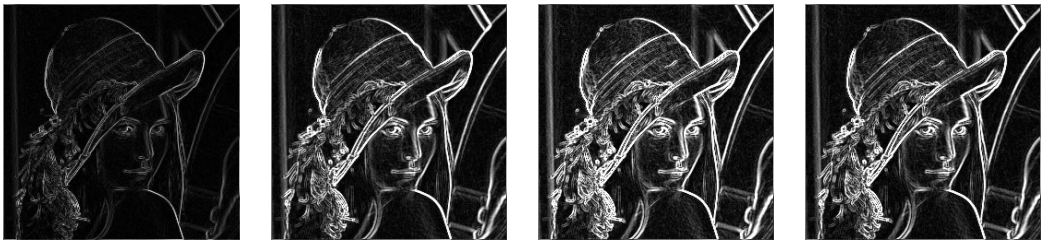

Figura 3: Imagen filtrada. Filtros: Roberts, Prewitt, Sobel, Frei-Chen (de izquierda a derecha).

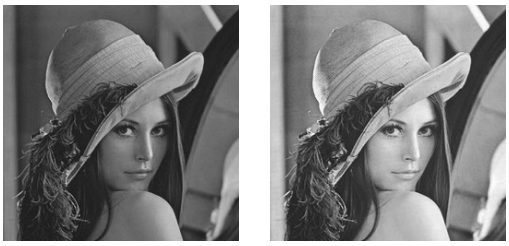

Figura 4: Imagen en esacala de grises. Tomando: $\epsilon = 1$, $\epsilon = \infty$ (de izquierda a derecha).

2. Enunciado

El objetivo de este trabajo práctico es analizar la performance de un procesador al hacer uso de las operaciones **SIMD** para el procesamiento de imágenes. Se implementarán 6 funciones en dos versiones. En lenguaje **C** y siendo compilado con el **GCC** utilizando procesamiento general y una versión en **Assembler**, haciendo uso de las operaciones de **SSE**. Finalmente, se compararán ambas versiones analizando las mejoras de performance obtenidas.

2.1. Código

2.1.1. Filtros

Implementar los filtros descriptos anteriormente para imágenes en escala de grises, tanto en lenguaje **C** como en **Assembler**. Precisamente, deberán implementar las siguientes funciones:

a) *void roberts_c (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

b) *void prewitt_c (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

c) *void sobel_c (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

d) *void freichen_c (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

e) *void roberts_asm (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

f) *void prewitt_asm (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

g) *void sobel_asm (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

h) *void freichen_asm (unsigned char* src, unsigned char* dst, int h, int w, int row_size)*

Donde:

- *src*: Es el puntero al inicio de la matriz de elementos de 8 bits sin signo que representa a la imagen de entrada. Como la imagen está en escala de grises, cada píxel se corresponde con un elemento de la matriz.
- *dst*: Es el puntero al inicio de la matriz de elementos de 8 bits sin signo que representa a la imagen de salida. Como la imagen está en escala de grises, cada píxel se corresponde con un elemento de la matriz.
- *h*: Representa el alto en píxeles de la imagen, es decir, la cantidad de filas de las matrices de entrada y salida.
- *w*: Representa el ancho en píxeles de la imagen, es decir, la cantidad de columnas de las matrices de entrada y salida.
- *row_size*: Representa la cantidad de bytes que ocupa una fila de, tanto la matriz de entrada como la matriz de salida. Dado que cada píxel está representado por un byte se esperaría que el tamaño en bytes de cada fila de las matrices de entrada y salida sea igual al ancho de la imagen. Esto no es cierto en muchos casos y tiene que ver con el formato en cual

está almacenada la imagen (i.e. *bmp*, *jpg*, *png*, etc). Algunos formatos extienden el tamaño (en bytes) de la filas de la imagen para que sea múltiplo de un valor conveniente para su posterior acceso, por ejemplo, 4. Es decir, si una imagen tiene 10 pixeles de ancho ocuparía 10 bytes de ancho pero se la extiende a 12 bytes para que sea más cómodo para procesar. Esto no afecta a la imagen en sí, mas bien es la representación interna de la misma.

Concretamente la aplicación de un filtro se resume en el siguiente cálculo:

$$dst_{ij} = saturar(abs(filtro_x) + abs(filtro_y)) \tag{5}$$

$$\begin{aligned} filtro_x = & \quad src_{i-1j-1} * f_{x00} + src_{i-1j} * f_{x01} + src_{i-1j+1} * f_{x02} + \\ & src_{ij-1} * f_{x10} + src_{ij} * f_{x11} + src_{ij+1} * f_{x12} + \\ & src_{i+1j-1} * f_{x20} + src_{i+1j} * f_{x21} + src_{i+1j+1} * f_{x22} \end{aligned} \tag{6}$$

$$\begin{aligned} filtro_y = & \quad src_{i-1j-1} * f_{y00} + src_{i-1j} * f_{y01} + src_{i-1j+1} * f_{y02} + \\ & src_{ij-1} * f_{y10} + src_{ij} * f_{y11} + src_{ij+1} * f_{y12} + \\ & src_{i+1j-1} * f_{y20} + src_{i+1j} * f_{y21} + src_{i+1j+1} * f_{y22} \end{aligned} \tag{7}$$

Donde f_x y f_y son los filtros horizontal y vertical respectivamente (para el caso del filtro de 2x2 la aplicación es análoga a la expuesta).

Es decir, para generar el píxel dst_{ij} de la imagen destino (filtrada), primero se aplica la máscara en x, luego en y (ambas máscaras centradas en el píxel src_{ij} de la imagen de origen) y por último se suman los valores absolutos de ambos resultados. Notar que el resultado de esta operación puede ser mayor que 255 por lo que hay que saturar el valor.

2.1.2. Monocromatización

Se pide también, implementar la función de conversión a escala de grises, tanto en lenguaje **C** como en **Assembler**. Para esta versión analizaremos los casos extremos de ϵ en la función ($\epsilon = 1$ y $\epsilon = \infty$) y fijaremos $\alpha = 1/4$, $\beta = 1/2$, $\gamma = 1/4$. Entonces nos quedan dos posibles funciones para monocromatizar:

$$f_{uno}(p) = \frac{(R + 2G + B)}{4}(\epsilon = 1) \tag{8}$$

$$f_{infinito}(p) = max(R, G, B)(\epsilon = \infty) \tag{9}$$

Precisamente, deberán implementar las siguientes funciones:

a) *void gris_epsilon_uno_c (unsigned char* src, unsigned char* dst, int h, int w, int src_row_size, int dst_row_size)*

b) *void gris_epsilon_inf_c (unsigned char* src, unsigned char* dst, int h, int w, int src_row_size, int dst_row_size)*

c) *void gris_epsilon_uno_asm (unsigned char* src, unsigned char* dst, int h, int w, int src_row_size, int dst_row_size)*

d) *void gris_epsilon_inf_asm (unsigned char* src, unsigned char* dst, int h, int w, int src_row_size, int dst_row_size)*

Donde:

- *src*: Es el puntero al inicio de la matriz de elementos de 24 bits sin signo (el primer byte representa el canal azul de la imagen (B), el segundo el verde (G) y el tercero el rojo (R)) que representa a la imagen de entrada. Es decir, como la imagen está en color, cada píxel está representado por 3 bytes.
- *dst*: Es el puntero al inicio de la matriz de elementos de 8 bits sin signo que representa a la imagen de salida. Como la imagen está en escala de grises cada píxel se corresponde con un elemento de la matriz.
- *h*: Representa el alto en píxeles de la imagen, es decir, la cantidad de filas de las matrices de entrada y salida.
- *w*: Representa el ancho en píxeles de la imagen, es decir, la cantidad de columnas de las matrices de entrada y salida.
- *src_row_size*: Idem *row_size* pero para la imagen fuente.
- *dst_row_size*: Idem *row_size* pero para la imagen destino.

2.1.3. Consideraciones

Las funciones que deban implementar en **Assembler** deben utilizar el juego de instrucciones **SSE** para optimizar la performance de las mismas.

Tener en cuenta lo siguiente:

- El ancho de las imagenes es siempre mayor a 16 píxeles.
- No se debe perder precisión en ninguno de los cálculos.
- La implementación de cada filtro deberá estar optimizada para el filtro que se está implementando. Es decir, no vale hacer una función que aplique un filtro genérico y después usarla para implementar los que se piden.
- Para el caso de las funciones implementadas en **Assembler** deberán procesar **al menos 8 pixeles** simultámeamente.

2.2. Desarrollo

Para facilitar el desarrollo del trabajo práctico, cuentan con un **.zip** (`tp2-codigo.zip`) que tiene lo necesario para poder compilar y probar las funciones que vayan a implementar. Entre las cosas más importantes se encuentra el programa principal (de línea de comandos), **tpcopados**, que se ocupa de parsear las opciones ingresadas por el usuario y ejecutar el filtro seleccionado sobre la imagen ingresada.

Para la manipulación de las imagenes (cargar, grabar, etc.) el programa hace uso de la libreria **OpenCV**. Para instalar esta librería en las distribuciones basadas en **Debian** sólo basta con ejecutar lo siguiente:

```
$ sudo apt-get install libcv-dev libhighgui-dev libcvaux-dev
```

La descripción completa del **.zip** es la siguiente:

- *bin*: Contiene el ejecutable del TP
- *data*: Contiene imágenes de prueba
- *enunciado*: Contiene este enunciado
- *src*: Contiene los fuentes del programa principal. Entre los archivos hay un **Makefile** que permite compilar el programa (entre otras cosas).
- *test*: Contiene algunos scripts para realizar tests sobre los filtros y sobre el uso de la memoria.

El uso del programa principal es el siguiente:

```
$ ./tpcopados opciones nombre_archivo_entrada [nombre_archivo_salida]
```

soporta los tipos de imágenes mas comunes y acepta las siguientes opciones:

- *-f, –filtro NOMBRE_FILTRO*
  Aplica un filtro a la imagen. Los filtros disponibles son: roberts (r), prewitt (p), sobel (s), freichen (f)
- *-g, –escala-gris NOMBRE_GRIS*
  Convierte la imagen a escala de gris. Los métodos disponibles son: uno (u), infinito (i)
- *-h, –help*
  Imprime esta ayuda
- *-i, –implementacion NOMBRE_MODO*
  Implementación sobre la que se ejecutará el proceso seleccionado. Los implementaciones disponibles son: c, asm
- *-t, –tiempo CANT_ITERACIONES*
  Mide el tiempo que tarda en ejecutar el filtro sobre la imagen de entrada una cantidad de veces igual a CANT_ITERACIONES

- *-v, –verbose*
  Imprime información adicional

Por ejemplo:

```
$ ./tpcopados -i asm -f prewitt ../data/lena.bmp
```

Aplica el filtro **Prewitt** a la imagen seleccionada utilizando la implementación en **Assembler** del filtro.

Y si hacemos:

```
$ ./tpcopados -t 1000 -i asm -f prewitt ../data/lena.bmp
```

Realiza lo mismo que antes pero repite la aplicación del filtro dentro de un ciclo de **1000** iteraciones y devuelve la cantidad de **ticks** (ciclos de reloj del procesador) que insumió al aplicación del filtro. Esto lo van a necesitar para poder comparar la performance de las versiones de **C** y **Assembler**.

Nota: Para evitar arrastrar errores, la aplicación de los filtros no utiliza las funciones de conversión a escala de gris implementada por ustedes sino que utiliza la que provee la librería **OpenCV**.

2.2.1. Mediciones de tiempo

Utilizando la instrucción de assembler `rdtsc` podemos obtener el valor del Time Stamp Counter (TSC) del procesador. Este registro se incrementa en uno con cada ciclo del procesador. Restando el valor del registro antes de llamar a una función al valor del registro luego de la llamada, podemos obtener la duración en ciclos de esa ejecución de la función.

Esta cantidad de ciclos no es siempre igual entre invocaciones de la función, ya que este registro es global del procesador y si nuestro programa es interrumpido por el scheduler para realizar un cambio de contexto, contaremos muchos más ciclos que si nuestra función se ejecutara sin interrupción. Por esta razón el programa principal del TP permite especificar una cantidad de iteraciones para repetir el filtro, con el objetivo de suavizar este tipo de outliers.

2.3. Informe

El informe debe incluir las siguientes secciones:

a) Carátula: La carátula del informe con el **número/nombre del grupo**, los **nombres y apellidos** de cada uno de los integrantes junto con **número de libreta** y **email**.

b) Introducción: Describen qué es lo que realizaron en el trabajo práctico.

c) Desarrollo: Describen **en profundidad** cada una de las funciones que implementaron. Para la decripción de cada función deberan mostrar gráficamente (mostrando el contenido de los registros XMM) cómo es una iteración del ciclo de la función. Es decir, cómo bajan los datos de la imagen a los registros, cómo los reordenan para poder procesarlos, las operaciones que le aplican a los datos, etc.

d) Resultados: **Deberán analizar** y **comparar** las implementaciones de cada funciones en su versión **C** y **Assembler** y mostrar los resultados obtenidos a través de tablas y gráficos. También deberán comentar sobre los resultados que obtuvieron. En el caso de que sucediera que la versión en C anduviese más rápido que su versión en Assembler **justificar fuertemente** a que se debe esto.

e) Conclusión: Comentar sobre qué les dejó el trabajo práctico, la programación vectorial a bajo nivel, problemas que se encontraron en el camino, etc.

Importante: El informe se evalúa de manera independiente del código. Puede reprobarse el informe y en tal caso deberá ser reentregado para aprobar el trabajo práctico.

3. Entrega

Se deberá entregar un archivo comprimido con el mismo contenido que el dado para realizarlo, habiendo modificado solo los archivos que tienen como nombre las funciones a implementar.

La fecha de entrega de este trabajo es **Martes 10 de Mayo** y deberá ser entregado a través de la página web. El sistema sólo aceptará entregas de trabajos hasta las **23:59** del día de entrega.

Ante cualquier problema con la entrega, comunicarse por mail a la lista de docentes.