Práctica 1. Lenguajes de alto nivel y ensamblador

Índice general

1.1. Objetivos

En esta práctica comenzaremos afianzando nuestros conocimientos sobre la programación en ensamblador sobre ARM adquiridos el curso pasado. Para ello trabajaremos en un proyecto de procesamiento de imágenes en el que tengamos que combinar código escrito en un lenguaje de alto nivel como C con código escrito directamente en lenguaje ensamblador y alguna biblioteca (que no sabremos a partir de qué ficheros fuente se generó). Los principales objetivos son:

- Recordar el convenio de paso de parámetros a funciones.
- Recordar los distintos ámbitos de variables, local y global.
- Comprender los tipos estructurados propios de los lenguajes de alto nivel.
- Comprender el código generado por el compilador *gcc*.

1.2. Representación digital de imágenes

Una imagen se puede representar como una matriz de *pixels*, en la que cada elemento de la matriz expresa el valor, en una determinada escala, de un *pixel*. En RGB, cada pixel se caracteriza por tres valores: cantidad de rojo (R), verde (G) y azul (B). Por tanto, cada elemento de una matriz que representa una imagen en color será un vector de tres elementos (R, G, B).

En concreto, para la presente práctica, usaremos la siguiente definición para el tipo *pixel* RGB:

```
typedef struct _pixel_RGB_t {
        unsigned char R;
        unsigned char G;
        unsigned char B;
} pixelRGB;
```

Como vemos en la definición del tipo, cada canal de color se representa con 8 bits. Esto quiere decir que podemos distinguir 256 tonos o niveles diferentes en cada canal, con un total de 24 bits por pixel (24bpp), que es lo habitual en muchos formatos de imagen actuales.

Por otro lado, para representar imágenes en *escala de grises* basta con un único valor para indicar la luminosidad de cada *pixel*. En esta práctica, utilizaremos nuevamente 8 bits para representar cada *pixel* en una imagen en escala de grises.

Por ejemplo, para definir dos imágenes, de 128 filas y 64 columnas de pixels cada una, la primera en color y la segunda en escala de grises, bastaría con realizar la siguiente declaración:

```
#define NFILAS 128
#define NCOLS 64

pixelRGB imagenColor[NFILAS*NCOLS];
unsigned char imagenEscalaGrises[NFILAS*NCOLS];
```

Si bien podríamos haber usado un tipo diferente (`pixelRGB imagenColor[128][64]`), hemos optado por una definición más sencilla que simplifica la comprensión del código ensamblador generado por el compilador. De este modo, y asumiendo que organizamos nuestras matrices por filas en memoria, para acceder al *pixel* de la fila 13, columna 24 de nuestras dos imágenes bastará con indexar como en este ejemplo:

```
imagenEscalaGrises[13*NCOLS + 24] = ... imagenColor[13*NCOLS + 24].R .... ;
```

1.3. Transformación RGB a escala de grises

La transformación entre ambos espacios de color se realiza mediante una sencilla función lineal. Como se observa en la Figura 1.1 es necesario multiplicar el valor de cada canal de un pixel por una constante, y la suma de los productos será el valor en escala de grises:

```
destino[x*NC+ y]= 0.2126*origen[x*NC+y].R + 0.7152*origen[x*NC+y].G + 0.0722*
    origen[x*NC+y].B;
```

Sin embargo, para poder usar esas constantes necesitaríamos poder operar con números reales. Pero en nuestro procesador ARM **no disponemos de representación en punto flotante** y por tanto no podemos operar con números reales. Para solventarlo, haremos una aproximación usando aritmética entera:

```
dest =  (3483*orig.R + 11718*orig.G + 1183*orig.B) /16384;
```

Aún nos encontramos con otra dificultad al utilizar ese código, ya que hemos decidido que representaremos, tanto los valores de cada canal de RGB como los de luminancia (grises) con tan solo 8 bits. Si bien el resultado de la operación anterior estará en el rango [0,255], representable con 8 bits, **las operaciones intermedias no lo están**. Cuando usamos el lenguaje C para codificar la sentencia anterior, se producen una serie de *castings* automá-

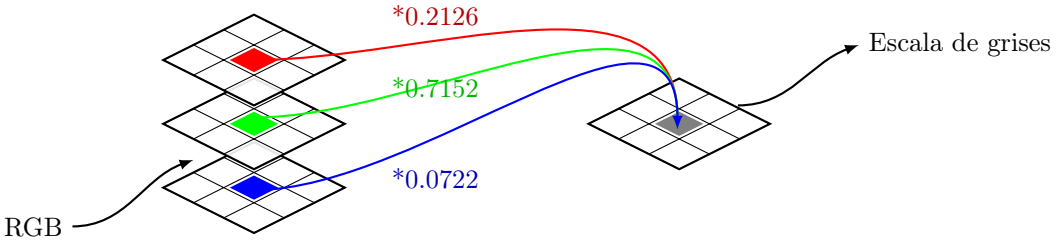

Figura 1.1: Transformación de `RGB` a escala de grises

ticos (cambios de tipo) para poder realizar las operaciones. Pero si hacemos cada una por separado, como en el siguiente ejemplo, el resultado sería **incorrecto**:

```
unsigned char aux_r, aux_g, aux_b;

aux_r =  3483*orig.R; // valor no codificable con 8 bits
aux_g = 11718*orig.G; // valor no codificable con 8 bits
aux_b =  1183*orig.B; // valor no codificable con 8 bits
dest = (aux_r + aux_g + aux_b)/16384;
```

1.4. Extracción de bordes

En esta práctica nos proponemos realizar una extracción de bordes de una imagen en color, como ilustra la Figura 1.2.

(a) Imagen RGB (b) Bordes extraídos

Figura 1.2: Ejemplo de extracción de bordes en la imagen Lenna.

La extracción de bordes puede realizarse de varias maneras. En esta práctica aplicaremos una técnica básica, consistente en:

1. Pasar la imagen en color a una imagen en escala de grises
2. Aplicar un filtrado gaussiano sobre la imagen en escala de grises para suavizarla (eliminar ruido)
3. Aplicar un filtrado de sobel (nos da una aproximación de la norma del gradiente en un punto) sobre la imagen suavizada para realzar los bordes

Para realizar estos filtrados uitilizaremos una pequeña librería de procesado de imagenes (libimgarm.a) que se suministra con el guion de la práctica. Esta biblioteca tiene dos funciones que implementan los dos filtros que necesitamos:

- `int gaussian(unsigned char* im1, int width, int height, int i, int j)`
- `int sobel(unsigned char* im1, int width, int height, int i, int j)`

Estas funciones nos calculan el valor del filtrado correspondiente al colocar el filtro sobre el elemento en la fila i y la columna j de la imagen que se pasa como primer argumento. Los argumentos `width` y `height` indican respectivamente el número de columnas y de filas de la imagen.

Por lo tanto, para aplicar alguno de estos filtrados sólo tenemos que recorrer los píxeles de la imagen, invocando una de estas funciones cada vez, pasándole las coordenadas del píxel correspondiente. Por ejemplo, para obtener la imagen suavizada en el array im2 aplicando el filtro gaussiano sobre la imagen en el array im1, haríamos:

```
int i,j;

for (i=0 ; i < height; ++i)
        for (j=0 ; j < width; ++j)
                im2[i * width + j] = gaussian(im1, width, height, i, j);
```

1.5. Trabajo a desarrollar (parte obligatoria)

El objetivo de esta práctica es realizar las transformaciones anteriores empleando código C y ensamblador, y ejecutarlas en la placa (no usando el simulador). El proyecto se desarrollará sobre los ficheros que se enumeran a continuación:

- `init.S` Este fichero, en lenguaje ensamblador, contendrá el símbolo global `start` e invocará la función `main()` que estará en el fichero `main.c`. **Este fichero se entregará completo** y no es necesario modificarlo.
- `main.c`. Este fichero define la función principal del programa (`main`), que se encargará de inicializar la matriz de color (RGB) e invocar a las funciones que implementan las transformaciones. Si se define la constante `LENA` (entregado así por defecto), la matriz RGB inicial contendrá los valores de una imagen de 512x512 muy popular en los trabajos de edición de imagen. Si no se compila con la macro LENA definida se creará una imagen sintética, y podremos elegir el ancho y el alto para facilitar el desarrollo y depuración.

  Además, se incluye una implementación de un timer que nos permite medir el tiempo que tardan cada una de las funciones. El código que se suministra utiliza dicho timer. **Este fichero se entregará completo** y no es necesario modificarlo. A continuación se muestra la implementación de la función `main()` para ilustrar el flujo de ejecución del programa completo:

  ```
  int main() {
  ```

```
    short int time = 0;
    timer_init()
    initRGB(N,M);
    timer_start();
    RGB2GrayMatrix(imagenRGB, imagenGris,N,M);
    apply_gaussian(imagenGris, imagenGauss, N, M);
    apply_sobel(imagenGauss, imagenSobel, N, M);
    time = timer_stop();
    return 0;
}
```

- `time.c / time.h`. Ficheros con la implementación de las funciones que nos permiten medir tiempos. La función `timer_stop()` devuelve un entero (de 16 bits) que nos indicará el tiempo transcurrido expresado en unidades de 0.5ms (es decir, si obtenemos el número 10000, querrá decir que han transcurrido 5 segundos). **Estos ficheros se entregarán completos** y no es necesario modificarlos.
- `types.h`. Este fichero contiene la definición del tipo `imagenRGB`.
- `trafo.h`. Este fichero contiene el prototipo de las funciones utilizadas en la función `main()`, salvo `initRGB(...)` que se implementa en el fichero `main.c`. Incorpora también el prototipo de una función auxiliar `rgb2gray` que deberá implementarse en ensamblador.
- `trafo.c`. En este fichero **el alumno realizará la implementación** de las transformaciones explicadas anteriormente.
- `misc.S` En este fichero **el alumno deberá realizar la implementación, en ensamblador, de las funciones** `rgb2gray` y `apply_gaussian`.
- `lena512color.h` y `lena512color.c`, contienen una imagen en color muy utilizada como benchmark en procesamiento de imágenes.
- `imgarm.h` fichero que exporta la interfaz de la biblioteca de procesado de imagen que vamos a utilizar.
- `libimgarm.a` biblioteca estática que contiene la implementación de los filtros gaussiano y de sobel. Para que el proyecto compile correctamente hay que modificar la configuración (*Settings*) de enlazado para indicar que debe enlazarse con esta biblioteca.

El trabajo del alumno consistirá en:

1. Implementar en C (en el fichero `trafo.c`) todas las funciones cuyo prototipo se incluye en el fichero `trafo.h`
   - `unsigned char rgb2gray(pixelRGB* pixel)`.
     Deberá aplicar la transformación lineal indicada en la sección 1.3 a los tres valores del pixel recibido como parámetro[1] para obtener el correspondiente valor en escala de grises.

[1] Se recibe un puntero al pixel, esto es, la dirección de un pixel

- `void apply_gaussian(unsigned char im1[], unsigned char im2[], int width , int height)` y
  `void apply_sobel(unsigned char im1[], unsigned char im2[], int width, int height)`.
  Implementar estas funciones siguiendo las indicaciones de la Sección 1.4.

2. Implementar, en **ensamblador** las siguientes funciones:

   - `unsigned char rgb2gray(pixelRGB* pixel)` y

   - `void apply_gaussian(unsigned char imagenGris[],short int imagenGauss[], int N, int M)`

   Se implementarán en el fichero `misc.S` y se modificará el código C para que invoque éstas versiones en lugar de las correspondientes versiones C (para ello habrá que comentar o cambiar el nombre de la implementación C correspondiente).

   Téngase en cuenta que el repertorio ARM **no dispone de ninguna instrucción de división**. Por tanto, se deberá codificar una rutina `div16384` que reciba un argumento (el numerador de la división) y devuelve la parte entera del cociente de dividir dicho argumento entre 16384. Téngase en cuenta que **16384 es una potencia de dos**. Implementando esta funcionalidad como una rutina aparte convertiremos a la rutina `rgb2gray` en *no-hoja*.

3. Comparar el tiempo de ejecución de la aplicación para la imagen de entrada *Lena* (no comentar la línea que define la constante `LENA` y asegurarse de dejar los valores de `N=M=512`), en los siguientes casos:

   - Proyecto únicamente con código C, compilado con flags `-O0 -g` (esto es, la configuración por defecto)
   - Proyecto únicamente con código C, compilado con flags `-O2`[2] (dejamos al compilador optimizar y no generamos información de depuración).
   - Proyecto con código C y código ensamblador en las tres funciones solicitadas, compilado con flags `-O2` (dejamos al compilador optimizar y no generamos información de depuración).

1.5.1. Visualización de resultados

Una vez realizadas las transformaciones, pero antes de finalizar la ejecución (es decir, poniendo un breakpoint en la sentencia `return` de la función `main`) es posible volcar el contenido de regiones de memoria de la placa en ficheros del disco del PC del laboratorio. Para ello, usaremos la orden *dump* del depurador *GDB*:

- En la perspectiva de depuración, visualizar la ventana *Console*. Específicamente seleccionar la consola *[GDB Hardware Debugging] arm-none-eabi-gdb*.

[2]Para cambiar las opciones de optimización abrir las propiedades del proyecto y seguir los menús `C/C++ Build -> Settings -> ARM Sourcery Windows GCC C Compiler -> Optimization` y elegir `-O2` como `Optimization level`

- Escribir el comando:

  `dump value` *nombreFicheroSalida matrizDeImagen*.

  Por ejemplo, para crear un fichero con la información contenida en la matriz `imagenGris` en el fichero `C:\hlocal\grises.dat`, deberíamos escribir:

  ```
  dump value C:\hlocal\grises.dat imagenGris
  ```

- El fichero con el volcado de memoria no está en ningún formato habitual de imagen, es un volcado crudo de la memoria. Para pasarlo a un formato reconocible por cualquier visor de imágenes ppm utilizaremos el programa *dump2ppm.c* suministrado con el guion de la práctica.

  Ojo, este código NO hay que ejecutarlo en la placa ni compilarlo con el compilador cruzado de ARM. Simplemente debe generarse un ejecutable en Windows que lea el fichero volcado y creará uno nuevo en formato PPM. Si no se dispone de ninguna herramienta para visualizar una figura en ese formato, existen herramientas *online* para convertir una imagen PPM en BMP, formato propietario de Windows.