Práctica 3: Vectorización aplicada a un problema real: procesado de imagen
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores

Jesús Alastruey Benedé y Víctor Viñals Yúfera
Área Arquitectura y Tecnología de Computadores
Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

27-marzo-2017

Figure 1: Annapurna I (8.091 m) desde el campo base (Nepal)

Departamento de Informática e Ingeniería de Sistemas
Universidad Zaragoza
Escuela de Ingeniería y Arquitectura
Universidad Zaragoza

Resumen

El objetivo de esta práctica es aplicar los conocimientos adquiridos en las dos sesiones anteriores a una aplicación de procesado de imagen. Vamos a abordar la conversión de RGB a YCbCr.

Ficheros y directorios de trabajo

Se proporciona un paquete `.tar.gz` en el que se encuentran los siguientes ficheros:

- `p3.md`: fichero de texto en formato *markdown* con el guión de la práctica.
- `jpeg_handler.c`: funciones para lectura y escritura de imágenes en formato JPEG.
- `convertRGB2YCbCr.c`: funciones para convertir una imagen de formato RGB a YCbCr.
- `dummy.c`: su objetivo es forzar que el compilador genere código que ejecute el bucle de trabajo un número especificado de repeticiones.
- `lib/libjpeg.a`: funciones de biblioteca para comprimir y descomprimir imágenes en formato JPEG.
- `include/jconfig.h`, `include/jerror.h`, `include/jmorecfg.h`, `include/jpeglib.h`: ficheros de cabecera necesarios para utilizar las funciones en la biblioteca anterior.
- `Makefile` y `Makefile.icc`: para compilar los ficheros fuente.
- `images/`: contiene una fotografía en formato JPEG.

Consideraciones previas

1. Requerimientos hardware y software:
   - CPU con soporte de la extensión vectorial AVX
   - SO linux

   Los equipos del laboratorio L0.04 y L1.02 cumplen los requisitos indicados. Puede trabajarse en dichos equipos de forma presencial y también de forma remota si hay alguno arrancado con Linux. En el guión de la práctica 1 se explica cómo descubrir qué máquinas de un laboratorio están accesibles de forma remota.

Parte 0. Biblioteca de funciones JPEG (libjpeg)

En el fichero `jpeg_handler.c` se encuentran las funciones que vamos a utilizar para leer y escribir imágenes en formato JPEG:

```
int  read_jpeg_file(char *filename, image_t *image);
/* char *filename: nombre del fichero que contiene la imagen a leer */
/* puntero a una estructura con información sobre la imagen leída
   (altura, anchura, número de canales) y
   el valor de los píxeles de la misma
   (3 bytes por pixel para imágenes RGB,
    1 byte por pixel para imágenes en escala de grises)
   Imagen almacenada por filas desde la esquina superior izquierda
   En caso de imagen RGB, los valores RGB se almacenan entrelazados:
       RGB_pixel0  RGB_pixel1 RGB_pixel2 ...
 */

int write_jpeg_file(char *filename, image_t *image);
/* char *filename: nombre del fichero que contiene la imagen a escribir */
/* puntero a una estructura con información sobre la imagen a escribir */
```

Ambas funciones hacen uso de la biblioteca `libjpeg`, software que implementa compresión y descompresión de imágenes en formato JPEG. En este enlace puedes obtener información sobre `libjpeg`:

http://www.ijg.org/

Puedes usar directamente la biblioteca proporcionada o mejor aún, compilarla desde las fuentes, que están disponibles en el siguiente enlace:

http://www.ijg.org/files/jpegsrc.v9b.tar.gz

Parte 1. Conversión de formato RGB a YCbCr

Vamos a trabajar con el fichero `convertRGB2YCbCr.c`.

1. Completa la función `convertRGB2YCbCr_v1()` de forma que convierta una imagen en formato RGB a formato YCbCr. Los componentes YCbCr de cada píxel se calculan de la siguiente forma:

   ```
   Y  =         0.299*R   +  0.587*G   +  0.114*B
   Cb = 128 - 0.168736*R - 0.331264*G +  0.500*B
   Cr = 128 -   0.5*R    - 0.418688*G - 0.081312*B
   ```

   https://en.wikipedia.org/wiki/YCbCr#JPEG_conversion

   Los valores de los píxeles están almacenados en forma de vector lineal, no como una matriz.

2. En primer lugar, vamos a verificar que la función de conversión funciona correctamente. Para ello, vamos a utilizar un programa que convierte una imagen de formato RGB a YCbCr y después realiza la conversión inversa, de YCbCr a RGB. Las dos imágenes generadas (RGB->YCbCr, YCbCr->RGB) se escriben en fichero.

   Compila el programa `test_convertRGB2YCbCr.c`:

   ```
   $ make test_convert_YCbCr
   ```

   Ejecútalo:

   ```
   $ ./test_convert_YCbCr
   ```

   Comprueba que las dos imágenes generadas se corresponden con la imagen original.

   (OPTATIVO) Si comparamos con detenimiento la imagen orginal y las generadas, se observan unas ligeras diferencias en la parte inferior derecha.
   ¿Cuál es el origen de dichas diferencias?

3. Observar el informe del compilador. ¿Ha vectorizado el bucle interno en `convertRGB2YCbCr_v1()`?

4. Modifica el código de la función `convertRGB2YCbCr_v2()` para que el compilador pueda vectorizarla. Pueden ser de utilidad las técnicas que vimos en la sesión 2. Ayuda: busca cómo indicar al compilador que un puntero en una estructura no apunta a la misma zona de memoria que otro puntero.

5. Ejecuta la versión modificada:

   ```
   $ ./test_convertRGB2YCbCr -c 1
   ```

   Calcula la aceleración obtenida respecto la la versión `convertRGB2YCbCr_v1()`.

6. Analiza el fichero que contiene el ensamblador y busca las instrucciones vectoriales correspondientes al bucle interno en `convertRGB2YCbCr_v2()`.

Parte 2. Transformación en la disposición de datos

En esta parte vamos a modificar la transformación de la disposición (layout) de los datos de la imagen para mejorar la eficiencia de los cálculos. En el caso de una imagen RGB, podemos cambiar de una organización de datos en formato vector de estructuras (*Array of Structures*, AoS):

```
R0 G0 B0 R1 G1 B1 ... Rn-1 Gn-1 Bn-1
```

a otra en formato estructura de vectores (*Structure of Arrays*, SoA):

```
R0 R1 ... Rn-1 G0 G1 ... Gn-1 B0 B1 ... Bn-1
```

Hay otras disposiciones posibles, como por ejemplo, una híbrida:

```
R0 R1 ... Rk-1 G0 G1 ... Gk-1 B0 B1 ... Bk-1 Rk Rk+1 ...
```

En el siguiente enlace se describen transformaciones AOS->SOA y SOA->AOS para vectorizar cálculos de procesado geométrico:

https://software.intel.com/en-us/articles/3d-vector-normalization-using-256-bit-intel-advanced-vector-extensions-intel-avx

1. Completa la función `convertRGB2YCbCr_SOA1()` que, antes de realizar los cálculos, transforme la disposición de los datos de la imagen RGB de formato AOS a SOA.
2. Verifica que la función de conversión funciona correctamente. Para ello, compila el programa `test_convertRGB2YCbCr.c`:

   ```
   $ make test_convertRGB2YCbCr
   ```

   Y ejecútalo:

   ```
   $ ./test_convertRGB2YCbCr -c 2
   ```

   Comprueba que las imágenes generadas se corresponden con la imagen original.
3. Analiza el fichero que contiene el ensamblador y busca las instrucciones vectoriales correspondientes al bucle interno en `convertRGB2YCbCr_SOA1()`.
4. Escribe una función `convertRGB2Gray_SOA_block()` que entrelace la transformación de los datos con los cálculos a realizar. De esta forma, en lugar de necesitar nuevas variables con **todos** los valores RGB de la imagen en formato SOA, solamente serán necesarias variables que almacenen **parte** de los valores RGB (en concreto, BLOCK píxeles).
5. Verifica que la función de conversión funciona correctamente.

   ```
   $ make test_convertRGB2YCbCr
   $ ./test_convertRGB2YCbCr -c 5
   ```
6. Compara el tiempo de ejecución de las funciones
   - `convertRGB2YCbCr_v2()`
   - `convertRGB2YCbCr_SOA1()`
   - `convertRGB2YCbCr_block()`

   Ten presente que el tiempo de ejecución de `convertRGB2YCbCr_SOA1()` no incluye la transformación de datos, mientras que el tiempo de ejecución de `convertRGB2YCbCr_block()` **sí** lo hace.
7. (OPTATIVO) Trata de reducir el tiempo de ejecución de `convertRGB2YCbCr_block()` cambiando el valor de BLOCK.

Optativo

Realizar los puntos anteriores con el compilador `icc`.