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

Marzo-2025

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

Resumen

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

Ficheros y directorios de trabajo

- `jpeg_handler.c`: funciones para lectura y escritura de imágenes en formato JPEG.
- `rgb2gray.c`: contiene varias funciones para convertir una imagen de formato RGB a escala de grises.
- `misc.c`: funciones diversas (medida de tiempos, comparación de imágenes, lectura y escritura de imágenes en formato PPM, PGM ...).
- `dummy.c`: su objetivo es forzar que el compilador genere código que ejecute el bucle de trabajo un número especificado de repeticiones.
- `test_rgb2gray.c`: programa que permite ejecutar y verificar las distintas funciones en `rgb2gray.c`.
- `Makefile` y `Makefile.icc`: para compilar los ficheros fuente.
- `images/`: contiene una fotografía en formato JPEG, que es la imagen a procesar. Este directorio almacenará las imágenes generadas así como la referencia para la verificación de resultados.

Consideraciones previas

Requerimientos hardware y software:

- CPU con soporte de la extensión vectorial AVX
- SO Linux

Los equipos de los laboratorios del DIIS cumplen los requisitos indicados. Puede trabajarse en dichos equipos de forma presencial y en los del L1.02 también de forma remota. En el repositorio de prácticas de la asignatura hay un documento que explica cómo descubrir equipos disponibles o encenderlos 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 de 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/

Usaremos la versión de la biblioteca `libjpeg` instalada en el sistema:

```
$ rpm -q libjpeg-turbo
```

Parte 1. Conversión de formato RGB a escala de grises

En esta parte vamos a trabajar con el fichero `rgb2gray.c`.

1. La función `rgb2gray_roundf0()` convierte una imagen en formato RGB a formato escala de grises. El valor de cada píxel de la imagen de salida se calcula de la siguiente forma:

   ```
   gray = roundf(0.299*R + 0.587*G + 0.114*B);
   ```

Siendo (R,G,B) los componentes RGB de cada píxel de la imagen sobre la que se aplica el filtro.

Analiza el código que realiza la conversión. Ayuda: los valores RGB de los píxeles están almacenados en forma de vector lineal con la siguiente disposición:

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

2. Compila el programa `test_rgb2gray.c`:

   ```
   $ make test_rgb2gray
   ```

   Ejecuta la función `rgb2gray_roundf0()` desde el programa `test_rgb2gray`:

   ```
   $ ./test_rgb2gray -c0 -r
   ```

   Verifica que se han generado dos imágenes en escala de grises a partir de la imagen entrada: una en formato JPEG y otra en formato PGM. Esta última la usaremos como referencia para validar los resultados de otros códigos.

3. Observar el informe del compilador. ¿Ha vectorizado el bucle interno en `rgb2gray_roundf0()`?
   Analiza el fichero que contiene el ensamblador y busca las instrucciones vectoriales (si existen) correspondientes al bucle en `rgb2gray_roundf0()`.

4. La función `rgb2gray_roundf1()` incluye cambios dirigidos a la vectorización de su bucle de cálculo. Las técnicas aplicadas fueron vistas en la práctica anterior.
   Observar el informe del compilador. ¿Ha vectorizado el bucle interno en `rgb2gray_roundf1()`?
   Analiza el fichero ensamblador y confirma tu respuesta anterior.

   Ejecuta la función `rgb2gray_roundf1()` desde el programa `test_rgb2gray`:

   ```
   $ ./test_rgb2gray -c 1
   ```

5. La función `rgb2gray_cast0()` sustituye la función de redondeo `roundf()` por un cast. Observar el informe del compilador. ¿Ha vectorizado el bucle interno en `rgb2gray_cast0()`?
   Analiza el fichero que contiene el ensamblador y busca el código asociado a dicho bucle.

   Ejecuta la función `rgb2gray_cast0()` desde el programa `test_rgb2gray`:

   ```
   $ ./test_rgb2gray -c 2
   ```

6. La función `rgb2gray_cast1()` utiliza en la conversión constantes de tipo `float` en lugar de `double`. Para ello se añade el sufijo `f`, por ejemplo: `0.5f`. Observa el informe del compilador. ¿Ha vectorizado el bucle interno en `rgb2gray_cast1()`?
   Analiza el fichero que contiene el ensamblador y busca el código asociado a dicho bucle.

   Ejecuta la función `rgb2gray_cast1()` desde el programa `test_rgb2gray`:

   ```
   $ ./test_rgb2gray -c 3
   ```

   Calcula su aceleración respecto `rgb2gray_cast0()`.

7. La función `rgb2gray_cast2()` es una variante de la anterior a la que se ha eliminado la suma del valor 0.5. Ejecuta la función `rgb2gray_cast2()` desde el programa `test_rgb2gray`:

   ```
   $ ./test_rgb2gray -c4
   ```

   Observa que la función que compara la imagen de salida con la de referencia detecta diferencias en casi la mitad de los píxeles. Compara de forma visual las diferencias de la imagen generada respecto a la de referencia.

   Calcula la aceleración respecto `rgb2gray_cast0()`.

8. **Optativo**. Elimina los `pragmas` de la función `rgb2gray_cast1()`. Evalúa sus prestaciones.

9. **Optativo**. Elimina los `restricts` de la función `rgb2gray_cast1()`. Evalúa sus prestaciones.

10. La función `rgb2gray_cast_esc()` es una variante escalar de `rgb2gray_cast1()`. En su declaración está una directiva que impide la vectorización. Ejecuta la función `rgb2gray_cast_esc()`:

    ```
    $ ./rgb2gray_filter -c5
    ```

    Calcula la aceleración de `rgb2gray_cast1()` respecto `rgb2gray_cast_esc()`.

Parte 2. Transformación en la disposición de datos

En esta parte vamos a modificar 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 describe una transformación de AoS a SoA:

https://www.intel.com/content/www/us/en/developer/articles/technical/memory-layout-transformations.html

1. Completa la función `rgb2gray_SOA0()` para que transforme la disposición de los datos de la imagen RGB de AoS a SoA antes de realizar los cálculos correspondientes al filtrado. Efectúa el filtrado como en la función `rgb2gray_cast1()`.
2. Verifica que la conversión funciona correctamente. Para ello, recompila y ejecuta el programa `test_rgb2gray.c`:

   ```
   $ make test_rgb2gray
   $ ./test_rgb2gray -c6
   ```

   Comprueba que la imagen en escala de grises generada se corresponde con la imagen de referencia. Calcula la aceleración respecto `test_rgb2gray_cast1()`.
3. Analiza el fichero que contiene el ensamblador y busca las instrucciones vectoriales correspondientes al bucle interno en `rgb2gray_SOA0()`.
4. Implementa la función `rgb2gray_SOA1()` como una variante de `rgb2gray_SOA0()` en la que se cuenta el tiempo de la transformacion de datos. Evalúa sus prestaciones:

   ```
   $ ./test_rgb2gray -c7
   ```

   Calcula la aceleración respecto `rgb2gray_cast0()`.
5. Escribe una función `rgb2gray_block()` que entrelace la transformación de los datos con los cálculos a realizar. De esta forma, las variables auxiliares almacenarán en formato SoA **parte** de los valores RGB (en concreto, `BLOCK` píxeles) en lugar de **todos** los valores RGB de la imagen.
6. Verifica que la función de conversión funciona correctamente.

   ```
   $ make test_rgb2gray
   $ ./test_rgb2gray -c8
   ```

   Calcula la aceleración respecto `rgb2gray_cast1()`.
7. **Optativo**. Intenta reducir el tiempo de ejecución de `rgb2gray_block()` cambiando el valor de `BLOCK`.
8. Compara el tiempo de ejecución de las distintas funciones:

   ```
   $ ./test_rgb2gray -c9
   ```

   Ten presente que el tiempo de ejecución de `rgb2gray_SOA0()` no incluye la transformación de datos, mientras que el tiempo de ejecución de `rgb2gray_block()` **sí** lo hace.
9. Elimina el flag `-ffast-math` en el fichero `Makefile`. Recompila y evalúa las distintas funciones:

   ```
   $ make clean
   $ make test_rgb2gray
   $ ./test_rgb2gray -c9
   ```

Optativo

Repetir los puntos anteriores con el compilador icx.