Evaluación de la práctica 3: Vectorización aplicada a un problema real: procesado de imagen 30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores Universidad de Zaragoza

Sergio García Esteban

14-marzo-2020

Resumen

Los tiempos y métricas deberán obtenerse para máquinas de los laboratorios L0.04 o L1.02. Sed concisos en las respuestas. Se valorarán las referencias utilizadas.

Notas generales

El trabajo puede presentarse de forma individual o en grupos de máximo dos personas. Podéis trabajar en grupos mayores, pero **cada grupo debe elaborar el material a entregar de forma independiente**. Hacedme llegar vuestros trabajos **en formato pdf** a través de la entrega habilitada en la web de la asignatura (moodle). Incluid vuestro nombre y apellidos en la cabecera del documento y vuestro NIP en el nombre del fichero (p3_NIP.pdf).
Plazo límite de entrega: martes 31 de marzo, 23h59m59s.

Parte 1. Conversión de formato RGB a YCbCr

1. Analiza el fichero que contiene el ensamblador y busca las instrucciones correspondientes al bucle interno en `RGB2YCbCr_cast1()`.
   ¿Cuántas instrucciones corresponden al cuerpo del bucle interno?
   ¿Cuántas de dichas instrucciones son vectoriales?

El bucle en C es traducido a ensamblador de la siguiente forma:
-Prólogo de 28 instrucciones de las cuales 2 son vectoriales
-BUCLE VECTORIAL de 250 INSTRUCCIONES de las cuales 246 son VECTORIALES
-Epílogo de 18 instrucciones de las cuales 0 son vectoriales
-Bucle escalar de 47 instrucciones de las cuales 0 son vectoriales

(Se interpretan como instrucciones vectoriales aquellas que tienen el prefijo 'vp' y aquellas que contengan 'ps' o '128')

2. Compara las prestaciones en términos de gigapíxeles procesados por segundo (Gpixels/s) para las siguientes versiones (se ha omitido el prefijo común `RGB2YCbCr_`):

función	Gpixels/s
`roundf0()`	0.05
`roundf1()`	0.06
`cast0()`	0.37
`cast1()`	0.69
`cast2()`	0.67
`cast_esc()`	0.16

Analiza brevemente los resultados.

`roundf0()` no es vectorizado.
`roundf1()` no es vectorizado, ahorra algo de tiempo al no tener que comprobar el solapamiento en tiempo de ejecución.
`cast0()` es vectorizado, suma 0.5 y hace cast a unsigned char.
`cast1()` es vectorizado, suma 0.5 en float y hace cast a unsigned char.
`cast2()` es vectorizado, no suma y hace cast a unsigned char.
`cast_esc()` no es vectorizado, suma 0.5 en float y hace cast a unsigned char.

Parte 2. Transformación en la disposición de datos

1. Lista el código fuente de la función `RGB2YCbCr_SOA0()`.

```
void
RGB2YCbCr_SOA0(image_t * restrict image_in, image_t * restrict image_out)
{
    double start_t, end_t;
    const int height = image_in->height;
```

```
const int width =  image_in->width;
unsigned char *Rpixels, *Gpixels, *Bpixels, *Ypixels, *Cbpixels, *Crpixels;

if (image_in->bytes_per_pixel != 3)
{
    printf("ERROR: input image has to be RGB\n");
    exit(-1);
}

/* fill struct fields */
image_out->width  = width;
image_out->height = height;
image_out->bytes_per_pixel = 3;
image_out->color_space = JCS_YCbCr;

/* transform data layout */
Rpixels = (unsigned char *) aligned_alloc(SIMD_ALIGN, 3*width*height);
Gpixels = Rpixels + 1*width*height;
Bpixels = Rpixels + 2*width*height;

Ypixels = (unsigned char *) aligned_alloc(SIMD_ALIGN, 3*width*height);
Cbpixels = Ypixels + 1*width*height;
Crpixels = Ypixels + 2*width*height;

/* transformación AoS -> SoA */
/* COMPLETAR ... */
#pragma GCC ivdep
for (int i = 0; i < height*width; i++)
{
    Rpixels[i] = image_in->pixels[3*i + 0];
    Gpixels[i] = image_in->pixels[3*i + 1];
    Bpixels[i] = image_in->pixels[3*i + 2];
}

start_t = get_wall_time();
for (int it=0; it < NITER; it++)
{
    /* COMPLETAR ... */
    #pragma GCC ivdep
    for (int i = 0; i < height*width; i++)
    {
        Ypixels[i] = (unsigned char) (0.5f +
            RGB2YCbCr_offset[0] +
            RGB2YCbCr[0][0]*Rpixels[i] +
            RGB2YCbCr[0][1]*Gpixels[i] +
            RGB2YCbCr[0][2]*Bpixels[i]);
```

```
            Cbpixels[i] = (unsigned char) (0.5f +
                RGB2YCbCr_offset[1] +
                RGB2YCbCr[1][0]*Rpixels[i] +
                RGB2YCbCr[1][1]*Gpixels[i] +
                RGB2YCbCr[1][2]*Bpixels[i]);
            Crpixels[i] = (unsigned char) (0.5f +
                RGB2YCbCr_offset[2] +
                RGB2YCbCr[2][0]*Rpixels[i] +
                RGB2YCbCr[2][1]*Gpixels[i] +
                RGB2YCbCr[2][2]*Bpixels[i]);
        }
        dummy(image_in, image_out);
    }
    end_t = get_wall_time();
    results(end_t - start_t, height*width, "RGB2YCbCrSOA0");

    /* transformación SoA -> AoS */
    /* COMPLETAR ... */
    #pragma GCC ivdep
    for (int i=0; i < height*width; i++)
    {
        image_out->pixels[3*i + 0] = Ypixels[i];
        image_out->pixels[3*i + 1] = Cbpixels[i];
        image_out->pixels[3*i + 2] = Crpixels[i];
    }

    free(Rpixels); free(Ypixels);
}
```

2. (OPTATIVO) Analiza el fichero que contiene el ensamblador y busca las instrucciones correspondientes al bucle interno en `RGB2YCbCr_SOA0()`.
   ¿Cuántas instrucciones corresponden al cuerpo del bucle interno?
   ¿Cuántas de dichas instrucciones son vectoriales?

   ```
   El bucle en C es traducido a ensamblador de la siguiente forma:
       -Prólogo de 5 instrucciones de las cuales 0 son vectoriales
       -BUCLE VECTORIAL de 171 INSTRUCCIONES de las cuales 168 son VECTORIALES
       -Epílogo de 13 instrucciones de las cuales 0 son vectoriales
       -Bucle escalar de 34 instrucciones de las cuales 0 son vectoriales

   (Se interpretan como instrucciones vectoriales aquellas que tienen el prefijo 'vp'
    y aquellas que contengan 'ps' o '128')
   ```

3. Lista el código fuente de la función `RGB2YCbCr_block()`.

```
void
```

```
RGB2YCbCr_block(image_t * restrict image_in, image_t * restrict image_out)
{
    /* Indicamos que no compruebe solapamiento en el campo pixels del struct */
    unsigned char * restrict pin;
    unsigned char * restrict pout;

    double start_t, end_t;
    const int height = image_in->height;
    const int width =  image_in->width;
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Rpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Gpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Bpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Ypixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Cbpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Crpixels[BLOCK];

    if (image_in->bytes_per_pixel != 3)
    {
        printf("ERROR: input image has to be RGB\n");
        exit(-1);
    }

    /* fill struct fields */
    image_out->width  = width;
    image_out->height = height;
    image_out->bytes_per_pixel = 3;
    image_out->color_space = JCS_YCbCr;

    start_t = get_wall_time();
    for (int it = 0; it < NITER; it++)
    {
        for (int i = 0; i < height*width; i+= BLOCK)
        {
            pin=&image_in->pixels[3*i];
            pout=&image_out->pixels[3*i];

            /* transformación AoS -> SoA */
            #pragma GCC ivdep
            for (int j = 0; j < BLOCK; j++)
            {
                Rpixels[j] = pin[3*j + 0];
                Gpixels[j] = pin[3*j + 1];
                Bpixels[j] = pin[3*j + 2];
            }
            /* conversión RGB -> YbCrCb */
            #pragma GCC ivdep
```

```
            for (int j = 0; j < BLOCK; j++)
            {
                Ypixels[j] = (unsigned char) (0.5f +
                    RGB2YCbCr_offset[0] +
                    RGB2YCbCr[0][0]*Rpixels[j] +
                    RGB2YCbCr[0][1]*Gpixels[j] +
                    RGB2YCbCr[0][2]*Bpixels[j]);
                Cbpixels[j] = (unsigned char) (0.5f +
                    RGB2YCbCr_offset[1] +
                    RGB2YCbCr[1][0]*Rpixels[j] +
                    RGB2YCbCr[1][1]*Gpixels[j] +
                    RGB2YCbCr[1][2]*Bpixels[j]);
                Crpixels[j] = (unsigned char) (0.5f +
                    RGB2YCbCr_offset[2] +
                    RGB2YCbCr[2][0]*Rpixels[j] +
                    RGB2YCbCr[2][1]*Gpixels[j] +
                    RGB2YCbCr[2][2]*Bpixels[j]);
            }
            /* transformación SoA -> AoS */
            #pragma GCC ivdep
            for (int j = 0; j < BLOCK; j++)
            {
                pout[3*j + 0] = Ypixels[j];
                pout[3*j + 1] = Cbpixels[j];
                pout[3*j + 2] = Crpixels[j];
            }
        }
        dummy(image_in, image_out);
    }
    end_t = get_wall_time();
    results(end_t - start_t, height*width, "RGB2YCbCr_block");
}
```

4. Compara las prestaciones en términos de gigapíxeles procesados por segundo (Gpixels/s) para las siguientes versiones (se ha omitido el prefijo común `RGB2YCbCr_`):

función	Gpixels/s
`roundf1()`	0.06
`cast1()`	0.69
`SOA0()`	1.20
`block()`	0.70

Analiza brevemente los resultados.

```
roundf1() no es vectorizado.
cast1() es vectorizado.
SOA0() es vectorizado y es más eficiente ya que los accesos a memoria son consecutivos
    Pero si contabilizamos el tiempo de transformación a SOA, el rendimiento es de 0.34
block() igual que el anterior, pero se realizan transformaciones AOS-SOA y conversión d
    de esta manera la memoria utilizada es la memoria necesaria para un bloque.
    (Contabiliza tiempo de transformación)
```

Ten presente que el tiempo de ejecución de `RGB2YCbCr_SOA0()` no incluye la transformación de datos, mientras que el tiempo de ejecución de `RGB2YCbCr_block()` **sí** lo hace.

5. **OPTATIVO**. Trata de reducir el tiempo de ejecución de `RGB2YCbCr_block()` cambiando el valor de `BLOCK`.

BLOCK	Gpixels/s
4	0.14
8	0.15
16	0.16
32	0.66
64	0.70
128	0.69
256	0.68
512	0.66