

Multiprocesadores

Ejercicio 2: Programar en ensamblador DLXV la suma de dos vectores

García Esteban, Sergio

19-febrero-2017

Tiempo dedicado (aproximado): 0,5 h

Resumen

Se trata de programar en ensamblador DLXV la suma de dos vectores cuyo número de elementos es múltiplo de 64.

Ejercicio

Codificar en ensamblador DLVX el siguiente código:

```
integer i, max
parameter (max = multiplo de 64)
real*8 C(max), A(max), B(max)

do i= 1,max
    C(i) = A(i) + B(i)
enddo
```

Suponemos que los siguientes registros enteros están inicializados con los valores indicados:

- $R_A = \&A(0)$
- $R_B = \&B(0)$
- $R_C = \&C(0)$
- $R_{cont} = max$
- $R_{64} = 64$

Solución

<i>bu :</i>	<code>movi2s VLR, R64</code>	<code>set 64 ops en cada instrucción vectorial</code>
	<code>lv V0, [RA]</code>	<code>lee 64 elementos de vector 1</code>
	<code>lv V1, [RB]</code>	<code>lee 64 elementos de vector 2</code>
	<code>addv V2, V0, V1</code>	<code>ejecuta suma de 64 elementos</code>
	<code>sv [RC], V2</code>	<code>guarda 64 elementos en vector 3</code>
	<code>add RA, RA, R64</code>	<code>actualización punteros</code>
	<code>add RB, RB, R64</code>	
	<code>add RC, RC, R64</code>	
	<code>sub Rcont, Rcont, R64</code>	<code>comparación final de bucle</code>
	<code>bne bu</code>	

Multiprocesadores

Ejercicio 4: Análisis de dependencias. Ejemplo 12

García Esteban, Sergio

6-abril-2019

Tiempo dedicado (aproximado): 30 mins

Resumen

Se trata de analizar las dependencias de un código.

Notas generales

El ejercicio 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**. Hacednos llegar vuestros ejercicios **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 (ej4_NIP.pdf).

Plazo límite de entrega: martes 23 de abril, 23h59m59s.

Ejercicio

Analizar las dependencias del siguiente código:

```
do i= 1,100
S1:    C(i) = A(3i+1)
S2:    A(2i+7)= B(i)-3
enddo
```

1. ¿Puede ejecutarse el bucle de forma vectorial? ¿Y paralela?

1

iteración	1	2	3	4	5	6	7	8	9	10	11	12	...
elem de A que se lee	4	7	10	13\$	16	19€	22	25€	28	31€	34	37	...
elem de A que se escribe	9	11	13\$	15	17	19€	21	23	25€	27	29	31€	...

Dependencias -> \$ Antidependencias -> €

En la tabla podemos observar que existe una sola dependencia en el espacio de iteraciones, entre la iteración 3 y la iteración 4, por lo tanto NO ES VECTORIZABLE, aunque se podría hacer una vectorización parcial ejecutando las primeras 3 iteraciones en modo escalar y el resto en modo vectorial. ✓

Observamos que en la iteración 6 existe una antidependencia, esta no impediría la paralelización, pero tanto las dependencias entre iteraciones como las antidependencias entre iteraciones, hacen el bucle NO PARALELIZABLE. ✓

+1
MUY BIEN

- FALTA ANÁLISIS MATEMÁTICO - 3

- FALTA GRAFO DEPENDENCIAS: - 2)

195

Evaluación de la práctica 1: Fundamentos de Vectorización en x86
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores Universidad de Zaragoza

Sergio García Esteban

3-marzo-2020

Resumen

Para la evaluación de la práctica 1 vais a resolver algunas cuestiones correspondientes a los puntos 1.4, 1.6 y 2.2 del guión de prácticas. Los tiempos y métricas deberán obtenerse para las máquinas del laboratorio L0.04. 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 (pl_NIP.pdf).
Plazo límite de entrega: domingo 8 de marzo, 23h59m59s.

Parte 1. Vectorización automática

4. ¿Cuántas instrucciones se ejecutan en el bucle interno (esc.avx, vec.avx, vec.avxfma y vec.avx512)?

26/27

```
for (int i = 0; i < LEN; i++)  
    x[i] = alpha*x[i] + beta
```

Calcula la reducción en el número de instrucciones respecto la versión esc.avx.

versión	icount	reducción(%)	reducción(factor)
esc.avx	6144	0	1.0
vec.avx	768	87,5	8.0
vec.avxfma	768	87,5	8.0
vec.avx512	384	93,75	16.0

Indica muy brevemente cómo has calculado los anteriores valores.

En escalar, 6 instrucciones por iteración y 1024 elementos, 6144 instrucciones. Las versiones avx y avxfma calculan el resultado de 8 elementos del vector en cada iteración del bucle y avx512 16 elementos, necesitarán 8 y 16 veces menos de iteraciones del bucle para calcular todo el vector.

5. A partir de los tiempos de ejecución obtenidos [...], calcula las siguientes métricas para todas las versiones ejecutadas:

- Aceleraciones (speedups) de las versiones vectoriales sobre sus escalares (vec.avx y vec.avxfma respecto esc.avx).
- Rendimiento (R) en GFLOPS.
- Rendimiento pico (R_{pico}) teórico de un núcleo (core), en GFLOPS. Para las versiones escalares, considerar que las unidades funcionales trabajan en modo escalar. Considerar asimismo la capacidad FMA de las unidades funcionales solamente para las versiones compiladas con soporte FMA.
- Velocidad de ejecución de instrucciones (V_i), en Giinstrucciones por segundo (GIPS).

Indica brevemente cómo has realizado los cálculos.

versión	tiempo(ns)	speed-up	R(GFLOPS)	R _{pico} (GFLOPS)	V _i (GIPS)
esc.avx	466.9	✓ 1.0	✓ 4.386	✓ 7,4	✓ 13,15
vec.avx	77.1	✓ 6.05	✓ 26,562	✓ 59,2	✓ 9,79
vec.avxfma	70.5	✓ 6.62	✓ 29,049	✓ 118,4	✓ 10,89

-4

51/55

Notas: GFLOPS = 10^9 FLOPS. GIPS = 10^9 IPS.

tiempo → medido en ejecución.
 speed-up → tiempo base / nuevo tiempo
 R → FLOPS / tiempo (ns) 312.443
 R-pico → UF * ops/UF * CPU-freq (2 UF y 3.7 GHz)
 V-I → icount / tiempo

¿La velocidad de ejecución de instrucciones es un buen indicador de rendimiento?

✓ No, ya que todas las instrucciones no son igual de productivas ni igual de costosas.

Parte 2. Vectorización manual mediante intrínsecos

2. Escribe una nueva versión del bucle, `ss_intr_AVX()`, vectorizando de forma manual con intrínsecos AVX. Lista el código correspondiente a la función `ss_intr_AVX()`.

```

__attribute__((noinline))
int ss_intr_AVX()
{
    double start_t, end_t;

    init();
    start_t = get_vall_time();

    if PRECISION==0
        __m256 vX, valpha, vbeta;
    for (unsigned int nl = 0; nl < NTIMES; nl++)
    {
        valpha = _mm256_set1_ps(alpha); // valpha = _mm_load1_ps(&alpha);
        vbeta = _mm256_set1_ps(beta);
        for (unsigned int i = 0; i < LEN; i += AVX_LEN)
        {
            vX = _mm256_load_ps(&x[i]);
            vX = _mm256_mul_ps(valpha, vX);
            vX = _mm256_add_ps(vX, vbeta);
            _mm256_store_ps(&x[i], vX);
        }
        dummy(x, alpha, beta);
    }
}

#else
    __m256d vX, valpha, vbeta;

```

3

```

for (unsigned int nl = 0; nl < NTIMES; nl++)
{
    valpha = _mm256_set1_pd(alpha); // valpha = _mm_load1_pd(&alpha);
    vbeta = _mm256_set1_pd(beta);
    for (unsigned int i = 0; i < LEN; i += AVX_LEN)
    {
        vX = _mm256_load_pd(&x[i]);
        vX = _mm256_mul_pd(valpha, vX);
        vA = _mm256_add_pd(vX, vbeta);
        _mm256_store_pd(&x[i], vA);
    }
    dummy(x, alpha, beta);
}
#endif

end_t = get_vall_time();
results[end_t - start_t, "ss_intr_AVX"];
check(x);
return 0;
}

```

Analiza el fichero que contiene el ensamblador de dicha función y busca las instrucciones correspondientes al bucle en `'ss_intr_AVX()'`.

```

scale_shift()
400740: c5 e4 59 00    vmulps (%rax),%ymm3,%ymm0
400744: 48 83 c0 20    add $0x20,%rax
400748: c5 fc 58 c2    vaddps %ymm2,%ymm0,%ymm0
40074c: c5 fc 29 40 e0 vmovaps %ymm0,-0x20(%rax)
400751: 48 39 c3       cmp %rax,%rbx
400754: 75 ea       jne 400740 <scale_shift+0x50>

ss_intr_AVX()
4008c0: c5 e4 59 00    vmulps (%rax),%ymm3,%ymm0
4008c4: 48 83 c0 20    add $0x20,%rax
4008c8: c5 fc 58 c2    vaddps %ymm2,%ymm0,%ymm0
4008cc: c5 fc 29 40 e0 vmovaps %ymm0,-0x20(%rax)
4008d1: 48 39 c3       cmp %rax,%rbx
4008d4: 75 ea       jne 4008c0 <ss_intr_AVX+0x50>

```

¿Hay alguna diferencia con las instrucciones correspondientes al bucle en `'scale_shift()'` (versión `vec_avx`)?
 Las instrucciones del bucle son las mismas.

¿Hay diferencia en el rendimiento de las funciones `'scale_shift()'` (versión `vec_avx`) y `'ss_intr_AVX()'`?

4

El tiempo obtenido en la ejecución es 75,1 ns en `scale_shift()` y 73,3 ns en `ss_intr_AVX()`, tienen rendimientos muy similares.

104

Evaluación de la práctica 2: Limitaciones a la Vectorización

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. Ser 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 (p2_NIP.pdf).

Plazo límite de entrega: jueves, 19 de marzo, 23h59m59s.

Parte 1. Efecto del alineamiento de los vectores en memoria

La función `ss_align_v1()` calcula el *kernel scale and shift*. El vector `x[]` está alineado con el tamaño vectorial de AVX, es decir, su dirección inicial es múltiplo de 32 bytes (256 bits).

```
for (unsigned int i = 0; i < LEN; i++)
    x[i] = alpha*x[i] + beta;
```

La función `ss_align_v2()` hace el mismo cálculo pero con el vector `x[]` NO alineado, ya que `ss_processa` desde el elemento con índice 1:

```
for (unsigned int i = 0; i < LEN; i++)
    x[i+1] = alpha*x[i+1] + beta;
```

Las funciones `ss_align_v1_intr()` y `ss_align_v2_intr()` implementan con intrínsecos los bucles de las funciones `ss_align_v1()` y `ss_align_v2()` respectivamente. En el primer caso los accesos a memoria son alineados y en el segundo son no alineados. Todas estas funciones las compilamos en la sesión de prácticas con las versiones 9.2 y 7.2 de gcc.

- 1. Para cada instrucción de escritura en memoria ejecutada, indica su tipo -escalar(E)/vectorial (V)- y la dirección del dato al que accede. En caso de instrucción vectorial, especifica solamente la dirección del primer elemento. Supón que el vector `x[]` tiene 32 elementos de tipo float, y que está almacenado a partir de la dirección `0x6020c0`. Añade las filas que sean necesarias en las tablas.

Notación: para cada instrucción vectorial, indica el número de elementos en el vector, y si dichos elementos deben estar alineados (A) o no (U).

align_v1()	tipo inst.	dirección
<code>movaps %ymm0, -0x20(%rax)</code>	V8A	0x6020c0
<code>movaps %ymm0, -0x20(%rax)</code>	V8A	0x6020e0
<code>movaps %ymm0, -0x20(%rax)</code>	V8A	0x602100
<code>movaps %ymm0, -0x20(%rax)</code>	V8A	0x602120

align_v2_gcc7()	tipo inst.	dirección
<code>vmovups %xmm0, 0x2017cf(%rip)</code>	V4U	0x6020c4
<code>vmovss %xmm0, 0x2017cb(%rip)</code>	E1	0x6020d4
<code>vmovss %xmm0, 0x2017bb(%rip)</code>	E1	0x6020d8
<code>vmovss %xmm0, 0x2017ab(%rip)</code>	E1	0x6020dc
<code>vmovaps %ymm0, -0x20(%rax)</code>	V8A	0x6020e0
<code>vmovaps %ymm0, -0x20(%rax)</code>	V8A	0x602100
<code>vmovaps %ymm0, -0x20(%rax)</code>	V8A	0x602120
<code>vmovss %xmm0, 0x20273e(%rip)</code>	E1	0x602140

30/30

align_v2_intr()	tipo inst.	dirección
vmovups %ymm0, -0x20(%rax)	V8U	0x6020c4
vmovups %ymm0, -0x20(%rax)	V8U	0x6020e4
vmovups %ymm0, -0x20(%rax)	V8U	0x602104
vmovups %ymm0, -0x20(%rax)	V8U	0x602124

Parte 2. Efecto del solapamiento de las variables en memoria

1. Escribe los tiempos de ejecución en ns de los bucles ejecutados en las siguientes llamadas a funciones. Describe muy brevemente en la tabla las tareas realizadas u obviadas en tiempo de ejecución. Notación:

- [n]S: [no] comprobar solapamiento
- [n]A: [no] comprobar alineamiento
- E/V: ejecución escalar/vectorial En caso de que no se efectúe alguna tarea (nX), indicar la razón.

Llamada a función	tiempo (ns)	tareas
ss_alias_v1(&x[i], y)	3954.1	S -> E
ss_alias_v1(y, x)	78.8	S A -> V
ss_alias_v2(&y[i], &x[i])	92.3	nS(restrict) A -> V
ss_alias_v2(y, x)	78.3	nS(restrict) A -> V
ss_alias_v3(&y[i], &x[i])	92.3	nS(pragmas ivdep) A -> V
ss_alias_v3(y, x)	78.3	nS(pragmas ivdep) A -> V
ss_alias_v4(y, x)	78.3	nS(restrict) nA(builtin_assume_aligned) -> V

Parte 3. Efecto de los accesos no secuenciales (stride) a memoria

1. Lista el código ensamblador correspondiente al bucle interno de la función ss_stride_vec().
¿Cuántas instrucciones vectoriales hay en el cuerpo del bucle?
Ayuda: utiliza las etiquetas al final de cada línea para identificarlas.

```

vmovaps (%rax), %ymm5
vshufps $0x88, 0x20(%rax), %ymm5, %ymm1
vperm2f128 $0x3, %ymm1, %ymm1, %ymm2
add $0x40, %rax
vshufps $0x44, %ymm2, %ymm1, %ymm0
vshufps $0xee, %ymm2, %ymm1, %ymm2
vinsrtf128 $0x1, %xmm2, %ymm0, %ymm0
vmulps %ymm4, %ymm0, %ymm0
vaddps %ymm3, %ymm0, %ymm0
vmovss %xmm0, -0x40(%rax)
vextractps $0x1, %xmm0, -0x38(%rax)
vextractps $0x2, %xmm0, -0x30(%rax)
vextractps $0x3, %xmm0, -0x28(%rax)
vextractf128 $0x1, %ymm0, %xmm0
vmovss %xmm0, -0x20(%rax)
vextractps $0x1, %xmm0, -0x18(%rax)
vextractps $0x2, %xmm0, -0x10(%rax)
vextractps $0x3, %xmm0, -0x8(%rax)
cmp $0x6030c0, %rax
jne 4007d8 <ss_stride_vec+0x48>

```

20 instrucciones, de las cuales 15 son vectoriales.

****OPTATIV0**.** Detalla las operaciones realizadas por las instrucciones vectoriales del bucle interno en 'ss_stride_vec()'.

```

vmovaps (%rax), %ymm5 -> lee de memoria 8 elementos del vector x[]
vshufps $0x88, 0x20(%rax), %ymm5, %ymm1 -> mezcla los 8 elementos leídos en la instr anterior y los siguientes 8 elementos en memoria, para obtener los 8 elementos de índice par.
vperm2f128 $0x3, %ymm1, %ymm1, %ymm2 -> permutación para ordenar los elementos vshufps $0x44, %ymm2, %ymm1, %ymm0 -> mezcla 1 para ordenar los elementos vshufps $0xee, %ymm2, %ymm1, %ymm2 -> mezcla 2 para ordenar los elementos vinsrtf128 $0x1, %xmm2, %ymm0, %ymm0 -> inserta 128 bits (xmm) de la mezcla 2 en el registro resultante de la mezcla 1 para obtener los 8 elementos ordenados
vmulps %ymm4, %ymm0, %ymm0 -> 8 elementos resultantes * alpha

```

vaddps %ymm3,%ymm0,%ymm0 -> 8 elementos resultantes + beta
vextractps \$0x1,%xmm0,-0x38(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 2
vextractps \$0x2,%xmm0,-0x30(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 4
vextractps \$0x3,%xmm0,-0x28(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 6
vextractf128 \$0x1,%ymm0,%xmm0 -> extrae de ymm0 a xmm0 los siguientes 4 elementos
vextractps \$0x1,%xmm0,-0x18(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 10
vextractps \$0x2,%xmm0,-0x10(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 12
vextractps \$0x3,%xmm0,-0x8(%rax) -> extrae de xmm0 y escribe en memoria elemento índice 14

2. Calcula la aceleración (*speedup*) de la versión icc sobre la gcc.

versión escalar: 470.1/470.2 -> 100% versión vectorial: 608.0/770.2 -> 78%

Parte 4. Efecto de las sentencias condicionales en el cuerpo del bucle

1. Lista el código ensamblador correspondiente al bucle interno de la función `cond_vec()`.
¿Cuántas instrucciones vectoriales hay en el cuerpo del bucle?

```

movaps 0x6020c0(%rax),%ymm2
movaps 0x6030c0(%rax),%ymm3
add $0x20,%rax
vcmpltps %ymm1,%ymm2,%ymm0
vblendvps %ymm0,%ymm2,%ymm3,%ymm0
movaps %ymm0,0x6010a0(%rax)
cmp $0x1000,%rax
jne 400940 <cond_vec+0x40>

```

8 instrucciones, de las cuales 5 son vectoriales.

2. Detalla las operaciones realizadas por las instrucciones vectoriales del bucle.
Por ejemplo:

```

movaps 0x6020c0(%rax),%ymm2 -> lee de memoria 8 elementos del vector y[]
movaps 0x6030c0(%rax),%ymm3 -> lee de memoria 8 elementos del vector x[]

```

vcmpltps %ymm1,%ymm2,%ymm0 -> genera una máscara comparando los 8 elementos leídos del vector y, si el elemento es menor al umbral escribe 1, else 0
vblendvps %ymm0,%ymm2,%ymm3,%ymm0 -> selecciona los elementos leídos de ambos vectores, si la máscara es 1 escribe elemento del vector y, else del x
vnoovaps %ymm0,0x6010a0(%rax) -> escribe en memoria los 8 elementos resultantes de la instr anterior en el vector z[]

3. Calcula la aceleración (*speedup*) de la versión vectorial sobre la escalar.

573.0/89.6 -> 639%

Speedup = 64

110

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_S0A0().

```
void  
RGB2YCbCr_S0A0(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 *Cpixels, *Bpixels, *Ypixels, *Cbpixels, *Crbpixels;
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 */
Bpixels = (unsigned char *) aligned_alloc(SIMD_ALIGN, 3*width*height);
Cpixels = Bpixels + 1*width*height;
Ypixels = Cpixels + 2*width*height;

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

/* transformación AoS -> SoA */
#pragma GCC ivdep
for (int i = 0; i < height*width; i++)
{
    Bpixels[i] = image_in->pixels[3*i + 0];
    Cpixels[i] = image_in->pixels[3*i + 1];
    Ypixels[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]*Bpixels[i] +
            RGB2YCbCr[0][1]*Cpixels[i] +
            RGB2YCbCr[0][2]*Ypixels[i]);
    }
}

```

3

```

Cbpixels[i] = (unsigned char) (0.5f +
    RGB2YCbCr_offset[1] +
    RGB2YCbCr[1][0]*Bpixels[i] +
    RGB2YCbCr[1][1]*Cpixels[i] +
    RGB2YCbCr[1][2]*Ypixels[i]);
Crbpixels[i] = (unsigned char) (0.5f +
    RGB2YCbCr_offset[2] +
    RGB2YCbCr[2][0]*Bpixels[i] +
    RGB2YCbCr[2][1]*Cpixels[i] +
    RGB2YCbCr[2][2]*Ypixels[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(Bpixels); 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

4

```

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))) Bpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Gpixels[BLOCK];
    unsigned char __attribute__((aligned(SIMD_ALIGN))) Rpixels[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++)
            {
                Bpixels[j] = pin[3*j + 0];
                Gpixels[j] = pin[3*j + 1];
                Rpixels[j] = pin[3*j + 2];

                /* conversión RGB -> YCbCr */
                #pragma GCC ivdep

```

```

                for (int j = 0; j < BLOCK; j++)
                {
                    Ypixels[j] = (unsigned char) (0.5f +
                        RGB2YCbCr_offset[0] +
                        RGB2YCbCr[0][0]*Bpixels[j] +
                        RGB2YCbCr[0][1]*Gpixels[j] +
                        RGB2YCbCr[0][2]*Rpixels[j]);
                    Cbpixels[j] = (unsigned char) (0.5f +
                        RGB2YCbCr_offset[1] +
                        RGB2YCbCr[1][0]*Bpixels[j] +
                        RGB2YCbCr[1][1]*Gpixels[j] +
                        RGB2YCbCr[1][2]*Rpixels[j]);
                    Crpixels[j] = (unsigned char) (0.5f +
                        RGB2YCbCr_offset[2] +
                        RGB2YCbCr[2][0]*Bpixels[j] +
                        RGB2YCbCr[2][1]*Gpixels[j] +
                        RGB2YCbCr[2][2]*Rpixels[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.
casti() es vectorizado.

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 c 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 RGE2YCbCr_SOA0() no incluye la transformación de datos, mientras que el tiempo de ejecución de RGE2YCbCr_block() sí lo hace.

5. **OPTATIVO.** Trata de reducir el tiempo de ejecución de RGE2YCbCr_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