Evaluación de la práctica 2: Limitaciones a la vectorización
30237 Multiprocesadores - Grado Ingeniería Informática
Esp. en Ingeniería de Computadores

Jesús Alastruey Benedé y Víctor Viñals Yúfera
Área Arquitectura y Tecnología de Computadores
Universidad de Zaragoza

16-marzo-2017

Resumen

Los tiempos y métricas deberán obtenerse para las máquinas del laboratorio L0.04, L1.02 o lab000. 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** por correo electrónico. Incluid vuestro nombre y apellidos en la cabecera del documento y vuestro NIP en el nombre del fichero (p1_NIP.pdf).
Plazo límite de entrega: miércoles 22 de marzo, 23h59m59s.

Parte 1. Efecto del alineamiento de las variables en memoria

La función `axpy_align_v1()` calcula el kernel AXPY. Todos los vectores están alineados con el tamaño de AVX, es decir, su dirección inicial es múltiplo de 32 bytes (256 bits).

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

```
La función `axpy_align_v2()` hace el mismo cálculo pero con vectores **NO**
alineados (los vectores se procesan desde el elemento con índice 1):

    for (int i = 0; i < LEN; i++) {
        y[i+1] = alpha*x[i+1] + y[i+1];
    }
```

Las funciones `axpy_align_v4()` y `axpy_align_v5()` implementan con intrínsecos los bucles de las funciones `axpy_align_v1()` y `axpy_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 `gcc`.

1. **Indica el tipo instrucción -escalar(E)/vectorial(V)- y las direcciones de los datos a los que acceden las instrucciones de escritura en memoria.**

En caso de instrucción vectorial, especifica solamente la dirección del primer elemento.
Supón que el vector `y[]` tiene 32 elementos de tipo `float`,
y que está almacenado a partir de la dirección `0x606100`.

función	tipo inst.	dirección
`s000_align_v1()`	V	`0x606100`
	V	`0x606120`
	V	`0x606140`
	V	`0x606160`

La instrucción vectorial utilizada para escribir el resultado en memoria es `vmovaps`.

función	tipo inst.	dirección
`s000_align_v2()`	E	`0x606104`
	E	`0x606108`
	E	`0x60610C`
	E	`0x606110`
	E	`0x606114`
	E	`0x606118`
	E	`0x60611C`
	V	`0x606120`
	V	`0x606140`

Los primeros 7 elementos se escriben utilizando la instrucción `vmovss` y el resto se escriben en grupos de 8 elementos usando la instrucción `vmovaps`.

función	tipo inst.	dirección
`s000_align_v5()`	V	`0x606104`
	V	`0x606124`
	V	`0x606144`
	V	`0x606164`

Los elementos se escriben en memoria con la instrucción vectorial `vmovups`.

Parte 2. Efecto del solapamiento de las variables en memoria

1. **Escribe los tiempos de ejecución de los bucles ejecutados**
   en las siguientes llamadas a funciones.
   Describe muy brevemente (en la tabla)
   las tareas realizadas *en tiempo de ejecución*.

llamada a función	tiempo	tareas
axpy_alias_v1(x, y, &y[1])	26.71	comprobar solapamiento => **hay**

llamada a función	tiempo	tareas
		ejecutar versión escalar
s000_alias_v1(x, y, z)	0.92	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **32B** no efectuar peeling
s000_alias_v2(&x[1], &y[1], &z[1])	0.93	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **no alineados** efectuar peeling
s000_alias_v2(x, y, z)	0.92	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **32B** no efectuar peeling
s000_alias_v3(&x[1], &y[1], &z[1])	1.12	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **no alineados** efectuar peeling
s000_alias_v3(x, y, z);	1.11	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **32B** no efectuar peeling
s000_alias_v4(x, y, z);	0.82	comprobar solapamiento => **no hay** ejecutar versión vectorial comprobar alineamiento => **32B** no efectuar peeling

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

1. **Analiza el ensamblador que ha generado `icc` al compilar la función `s000_stride_v1()`.**
   ¿Cuántas instrucciones corresponden al cuerpo del bucle interno?
   ¿Cuántas de dichas instrucciones son vectoriales?
   Ayuda: utiliza las etiquetas al final de cada línea para identificarlas.

   (OPTATIVO) Detalla las operaciones realizadas por las instrucciones vectoriales del bucle interno en `s000_stride_v1()`.

   El cuerpo del bucle está comprendido entre las etiquetas `..B7.5` y `..B7.6`:

```
..B7.5:
    lea       (,%rdx,8), %rcx
    vmovss    x(,%rdx,8), %xmm2
    vmovss    y(,%rdx,8), %xmm5
    vinsertps $16, 8+y(,%rdx,8), %xmm5, %xmm6
    vinsertps $16, 8+x(,%rdx,8), %xmm2, %xmm3
    vinsertps $32, 16+x(,%rdx,8), %xmm3, %xmm4
    vinsertps $32, 16+y(,%rdx,8), %xmm6, %xmm7
    vinsertps $48, 24+y(,%rdx,8), %xmm7, %xmm9
    vinsertps $48, 24+x(,%rdx,8), %xmm4, %xmm8
    addq      $4, %rdx
    vfmadd231ps %xmm1, %xmm8, %xmm9
    cmpq      $1024, %rdx
    vmovss    %xmm9, y(%rcx)
    vextractps $1, %xmm9, 8+y(%rcx)
    vextractps $2, %xmm9, 16+y(%rcx)
    vextractps $3, %xmm9, 24+y(%rcx)
    jb        ..B7.5
```

Este contiene 13 instrucciones escalares (`lea`, `vinsertps`, `addq`, `cmpq`, `vextractps` y `jb`) y 4 instrucciones vectoriales (`vmovss` y `vfmadd231ps`).

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

$$t_{ex\,GCC} = 3.78;\ t_{ex\,ICC} = 3.54$$

$$speedup_{ICC} = \frac{t_{ex\,GCC}}{t_{ex\,ICC}} = 1.07$$

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

1. **¿Cuántas instrucciones vectoriales corresponden al cuerpo del bucle vectorizado? Detalla las operaciones realizadas por las instrucciones vectoriales del bucle.**

```
400d0b:    vpcmpeqd %ymm3,%ymm3,%ymm3
400d0f:    vmovaps 0x289(%rip),%ymm4
400d17:    vmovsd %xmm0,-0x58(%rbp)
400d1c:    nopl   0x0(%rax)
400d20:    xor    %eax,%eax
400d22:    nopw   0x0(%rax,%rax,1)

400d28:    vmovaps 0x606100(%rax),%ymm1           # carga y[i]..y[i+7] en %ymm1
400d30:    add    $0x20,%rax
400d34:    vcmpltps %ymm4,%ymm1,%ymm2             # compara y[i]..y[i+7] con 1.0/1023.0
400d39:    vpxor  %ymm2,%ymm3,%ymm0               # crea máscara en %ymm0
400d3d:    vmaskmovps 0x6040e0(%rax),%ymm0,%ymm0  # carga en %ymm0 condicionalmente elems de y
400d46:    vblendvps %ymm2,%ymm1,%ymm0,%ymm0      # copia condicionalmente en %ymm0 elems de y (%ymm
400d4c:    vmovaps %ymm0,0x6020e0(%rax)           # almacena el resultado en z
400d54:    cmp    $0x2000,%rax
400d5a:    jne    400d28 <cond_vec+0x48>
```

El bucle interno contiene 3 instrucciones escalares (`add`, `cmp` y `jne`) y 6 vectoriales (`vmovaps`, `vcmpltps`, `vpxor`, `vmaskmovps`, `vblendvps`y `vmovaps`).

El funcionamiento es el siguientes:

1. **400d28:** `vmovaps` carga el vector en la dirección `0x606100+%rax` (`y[i]..y[i+7]`) en el registro `%ymm1`.

2. **400d30:** Se incrementa el índice `%rax` en `0x20`; `i += 8`.
3. **400d34:** `vcmpltps` almacena en `%ymm2` una máscara que indica los elementos de `%ymm1` que son menores que los de `%ymm4` (en el que previamente se ha cargado el resultado de la operación `1.0 / 1023.0`).
4. **400d39:** `vpxor` almacena en `%ymm0` el resultado de aplicar la operación XOR bit a bit a los elementos de `%ymm2` (resultados de la comparación anterior) e `%ymm3`, un vector con todos los elementos igual a 0. El resultado es que `%ymm0` contiene un vector de elementos `0x00000000` para los elementos que cumplen la condición y `0xFFFFFFFF` para los que no la cumplen.
5. **400d3d:** `vmaskmovps` carga condicionalmente `y[i]..y[i+7]` desde memoria (almacenados a partir de `0x6040e0+%rax`) en el registro `%ymm0`, para los elementos en los que `%ymm0` no sea `0x00..00`.
6. **400d46:** `vblendvps` copia condicionalmente al registro `%ymm0` los elementos de `%ymm2` o de `%ymm1`, en función de los bits que estén marcados en `%ymm0`; para elementos que contengan `0x00`, copiará de `%ymm1` y para elementos que contengan otra cosa, de `%ymm2`.
7. **400d4c:** `vmovaps` almacena en memoria los elementos del registro `%ymm0`, en las direcciones correspondientes a `z[i]..z[i+7]` (almacenados a partir de `0x6020e0+%rax`).

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

$$t_{ex\,esc} = 5.5;\ t_{ex\,vec} = 1.6$$

$$speedup_{vec} = \frac{t_{ex\,esc}}{t_{ex\,vec}} = 3.44$$