Práctica 3

Procesadores vectoriales (II)

1 Introducción

En la práctica anterior se introdujo la arquitectura vectorial DLXV. Para ello se analizaron diversas rutinas que mostraban detenciones del cauce segmentado por diversos motivos. Finalmente se propuso al estudiante que implementara el bucle DAXPY para vectores de una longitud mayor que MVL, de esta manera se inició al estudiante en las técnicas de programación en máquinas vectoriales.

En la presente práctica se revisarán diversas estrategias de programación en procesadores vectoriales con el objetivo de aumentar el rendimiento del procesador.

2 Técnicas de programación en DLXV. Revisión

En el caso del Bucle DAXPY propuesto en la práctica anterior se introdujo el caso en que el tamaño del vector a tratar difería del tamaño de los vectores de la máquina vectorial. Este problema introducía la técnica de seccionamiento o *strip mining*, para aumentar el rendimiento del procesador. Hay más casos en los que el rendimiento de la máquina vectorial se degrada. Veamos algunos de ellos.

2.1 Separación entre los elementos de un vector.

Un problema que rápidamente surge es el del acceso a los elementos de un vector que no ocupan posiciones consecutivas de memoria. Veamos el siguiente código para la multiplicación de matrices:

```
      do 10  i = 1, 100
      do 10  j = 1, 100
         A(i,j) = 0.0
      do 10  k= 1, 100
10       A(i,j) = A(i,j) + B(i,k)*C(k,j)
```

La multiplicación de cada fila de B con cada columna de C puede ser vectorizada y aplicar la técnica de seccionamiento al bucle interior, con `k` como el índice variable. La distribución por columnas, usada por el FORTRAN, hace que los elementos `B(i,j)` y `B(i+1,j)` sean adyacentes, es decir, en cada iteración se accederá a un elemento separado del anterior por una fila completa de la matriz. En este caso, los elementos de `B` que son accedidos por las iteraciones del bucle interior están separados por 8 veces (el número de bytes por elemento) el tamaño de la fila, resultando un total de 800 bytes.

En el ejemplo anterior, utilizando una distribución por columnas para las matrices, la matriz C tiene una separación o *stride* de 1 (1 doble palabra, es decir, 8 bytes) y la matriz B tiene una separación o *stride* de 100 (100 dobles palabras, es decir, 800 bytes).

Una vez que los elementos deseados han sido cargados en el registro vectorial, se tiene un nuevo vector con sus elementos lógicamente adyacentes. Esto permite a una máquina vector-registro manejar separaciones mayores que uno, llamadas separaciones no unitarias, haciendo que las operaciones de carga y almacenamiento de vectores sean más generales. Si se almacenan por filas, los elementos consecutivos de las filas son adyacentes en memoria, mientras que si se almacenan por columnas, los elementos consecutivos en las columnas son adyacentes. Esto es extensible a matrices con `n` dimensiones.

La separación en un vector, igual que la dirección de comienzo del vector, puede ser colocado en un registro de propósito general, donde es utilizado por la operación vectorial. La instrucción `lvws` (*load vector with stride* del DLXV es la utilizada para cargar un vector utilizando una separación (*stride*). Igualmente, cuando un vector con separación no unitaria va a ser almacenado, puede utilizarse la instrucción `svws` (*store vector with stride*).

En la unidad de memoria pueden aparecer complicaciones al soportar separaciones mayores de uno, por tanto será posible solicitar accesos al mismo banco a una velocidad superior al tiempo de acceso de la memoria. Esta situación recibe el nombre de **conflicto de bancos de memoria** y provoca que cada carga o almacenamiento sea penalizada en mayor medida por el tiempo de acceso de la memoria.

Un conflicto en los bancos de memoria se produce siempre que se pide al mismo banco un acceso antes de que haya completado el anterior. En los bancos de memoria no ocurrirán conflictos si los valores de la separación y del número de bancos son primos entre sí y existen suficientes bancos de memoria para evitar conflictos en el caso de separación unitaria.

2.2 Matrices dispersas.

Una matriz dispersa es una matriz en la que la mayoría de los elementos es cero. Estas matrices se suelen almacenar de una manera inteligente (con vectores que almacenen los índices que no son cero). De esta manera una suma de vectores dispersos se podría hacer de la siguiente manera:

```
      do 100  i = 1, n
100      A(K(i)) = A(K(i)) + C(M(i))
```

Este código implementa la suma de `A` y `C`, usando los vectores de índices `K` y `M` para designar los elementos que no son cero de `A` y de `C`. (`A` y `C` deben tener el mismo número de elementos que no son cero, en este caso `n`.)

Un mecanismo para soportar las matrices dispersas se basa en las operaciones de **expansión-comprensión** (*scatter-gather*) utilizando un **vector de índices**. Una operación de compresión (*gather*) toma un vector de índices y busca el vector cuyos elementos están en las direcciones dadas por la dirección base más los desplazamientos presentes en el vector de índices. El resultado es un vector no disperso en el registro vectorial. Después de operar sobre esos elementos de una forma densa, el vector disperso puede ser almacenado en forma expandida mediante un almacenamiento distribuido (*gather*) utilizando el mismo vector de índices.

Esta técnica permite que el código con matrices dispersas pueda ser ejecutado de forma vectorial. El código fuente descrito anteriormente no podría ser vectorizado automáticamente por un compilador debido a que el compilador no puede saber que los elementos de `K` son valores distintos y que, por tanto, no existen dependencias. En su lugar, una directiva del ensamblador podría comunicar al compilador que puede vectorizar el bucle.

Las instrucciones `lvi` (Cargar vector indexado) y `svi` (Almacenar vector indexado) proporcionan esas operaciones en el DLXV. La instrucción `cvi` (Crear vector de índices) del DLXV crea un vector de índices a partir de una separación (*m*) donde los valores del vector de índices serán: 0,*m*,2*m*,..., 63*m*.

3 Experimentando con el simulador xdlxvsim

Ejercicio 1:

Realizar un programa de multiplicación de matrices **almacenadas por columnas en memoria** (como en Fortran). Las matrices serán de 6x6 números de punto flotante de doble precisión (8 bytes) con estos valores:

$$\begin{pmatrix} 10 & 11 & 12 & 13 & 14 & 15 \\ 20 & 21 & 22 & 23 & 24 & 25 \\ 30 & 31 & 32 & 33 & 34 & 35 \\ 40 & 41 & 42 & 43 & 44 & 45 \\ 50 & 51 & 52 & 53 & 54 & 55 \\ 60 & 61 & 62 & 63 & 64 & 65 \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 \\ 7 & 8 & 9 & 1 & 2 & 3 \\ 4 & 5 & 6 & 7 & 8 & 9 \\ 9 & 8 & 7 & 6 & 5 & 4 \\ 3 & 2 & 1 & 9 & 8 & 7 \\ 6 & 5 & 4 & 3 & 2 & 1 \end{pmatrix} = \begin{pmatrix} 384 & 375 & 366 & 384 & 375 & 366 \\ 684 & 675 & 666 & 684 & 675 & 666 \\ 984 & 975 & 966 & 984 & 975 & 966 \\ 1284 & 1275 & 1266 & 1284 & 1275 & 1266 \\ 1584 & 1575 & 1566 & 1584 & 1575 & 1566 \\ 1884 & 1875 & 1866 & 1884 & 1875 & 1866 \end{pmatrix}$$

Se recuerda que el DLXV no tiene instrucciones para sumar todos los elementos de un mismo vector. También se recuerda que acceder a elementos consecutivos en memoria es más eficiente que si los elementos están separados.

Ejercicio 2:

A continuación se pretende repasar los conceptos expuestos al principio de la memoria y observar las paradas y rendimientos de bucles que utilicen estas técnicas de programación en máquinas vectoriales. En primer lugar ejecuta en el modo paso a paso el bucle siguiente:

```
; Ejemplo Vectorial 4
; Acceso a matrices dispersas
        .data 0
        .global a
a:      .double 1, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 2, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 3, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 4, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 5, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 6, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 7, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 8, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 9, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .double 10, 0, 0, 0, 0, 0, 0, 0, 0, 0
        .global k
k:      .double 0, 80, 160, 240, 320, 400, 480, 560, 640, 720
        .global c
c:      .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .double 0, 10, 0, 0, 0, 0, 0, 0, 0, 0
        .global m
m:      .double 8, 88, 168, 248, 328, 408, 488, 568, 648, 728

inicio: add     r10, r0, k      ;r10 contiene la dirección del vector K
        add     r11, r0, a      ;r11 contiene la dirección del vector A
        add     r12, r0, m      ;r12 contiene la dirección del vector M
        add     r13, r0, c      ;r13 contiene la dirección del vector C
        add     r1, r0, #10     ;10 -> r1
        movi2s  vlr, r1         ;10 -> vlr
        lv      v1, r10         ;Carga vector K en V1
```

```
        lvi     v2, r11, v1     ;Carga vector A(K(i))
        lv      v3, r12         ;Carga vector M
        lvi     v4, r13, v3     ;Carga vector C(M(i))
        addv    v2, v2, v4      ;Suma vectorial
        svi     r11, v1, v2     ;Almacena A(K(i))
        trap    #0              ;Fin del programa vectorial
```

En el presente ejemplo se utiliza el acceso a matrices dispersas comentado anteriormente. Observa las detenciones que se producen y justifica el por qué de estas detenciones. ¿Cómo se podrían evitar? Estudia la utilización del vector de índices para el agrupamiento/dispersión de los elementos correspondientes.

Ejercicio 3:

A continuación se propone que el estudiante ejecute el siguiente código vectorial donde se usan técnicas de ejecución condicional mediante agrupamiento y dispersión.

```
; Ejemplo Vectorial 5
; Ejecución condicional de instrucciones mediante scatter/gather

        .data 0
        .global a
a:      .double 10, 0, 10, 0, 10, 0, 10, 0, 10, 0
        .double 10, 0, 10, 0, 10, 0, 10, 0, 10, 0
        .global b
b:      .double 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
        .double 11, 12, 13, 14, 15, 16, 17, 18, 19, 20
        .global cero
cero:   .double 0

inicio: add     r10, r0, a      ;r10 contiene la dirección del vector A
        add     r11, r0, b      ;r11 contiene la dirección del vector B
        add     r1, r0, #20     ;20 -> r1
        movi2s  vlr, r1         ;20 -> vlr
        add     r2, r0, #8      ;8 -> r2
        lv      v1, r10         ;Carga vector A en V1
        ld      f0, cero        ;Carga un cero en coma flotante en F0
        snesv   f0, v1          ;Pone VM a 1 sí V1(i) distinto de F0
        cvi     v2, r2          ;Crea vector de índices en V2
        pop     r1, vm          ;Cuenta el numero de unos de VM
        movi2s  vlr, r1         ;Carga el vector de longitud vectorial
        cvm                     ;Pone VM todo a 1
        lvi     v3, r10, v2     ;Carga los elementos de A distintos de cero
        lvi     v4, r11, v2     ;Carga los elementos de B correspondientes
        subv    v3, v3, v4      ;Resta vectorial
        svi     r10, v2, v3     ;Almacena el resultado en A
        trap    #0              ;Fin del programa vectorial
```

Estudia de nuevo el código con cuidado. Justifica de nuevo las detenciones que se producen y propón un método para evitar las detenciones de memoria.

4 Trabajo a realizar

En la memoria sobre procesadores vectoriales, tanto de la práctica 3 como de la 4, se adjuntará un análisis de tiempos de ejecución de las rutinas de ejemplo y de las realizadas por el estudiante, así como un análisis de riesgos y rendimientos de la arquitectura vectorial del DLXV para cada caso.