Capítulo 3

Procesadores Vectoriales

3.1 Introducción

Aunque la utilización de procesadores superescalares y supersegmentados permite mejorar el rendimiento de un procesador, existen límites al aumento de rendimiento que se puede conseguir en un cauce. Estos límites se deben a dos factores principales:

- *La frecuencia de funcionamiento del reloj*. La frecuencia del reloj puede incrementarse como vimos en el caso de procesadores supersegmentados. Sin embargo, a partir de cierto punto un incremento en el número de etapas del cauce puede ralentizar el procesador debido, sobre todo, a las dependencias entre las instrucciones y a la sobrecarga de los *latchs* necesarios entre cada etapa.

- *Frecuencia de adquisición y decodificación de la instrucción*. Este obstáculo, a veces llamado el *cuello de botella de Flynn*, hace difícil la adquisición y decodificación de muchas instrucciones por ciclo de reloj. Este obstáculo es el que dificulta construir procesadores con frecuencias de reloj altas y con ratios de emisión de instrucciones muy altos.

Los procesadores encauzados de alta velocidad son particularmente útiles en las grandes aplicaciones científicas y de ingeniería. Este tipo de procesadores utilizan una caché para evitar la alta latencia que supone acceder a la memoria principal. Sin embargo, los programas científicos suelen tener un conjunto de datos muy activo a los que se accede con poca localidad lo que empobrece el rendimiento de la jerarquía de memoria. Este problema se puede evitar no introduciendo en la caché estas estructuras si es posible determinar los patrones de acceso a la memoria y encauzar los accesos a la misma de forma eficiente. Las últimas arquitecturas caché y el uso del compilador a través de instrucciones de bloque y precarga han reducido estos problemas en la jerarquía de la memoria, pero continúan siendo un serio problema en algunas aplicaciones.

Los *procesadores vectoriales* permiten operaciones de alto nivel que trabajan sobre *vectores*. Una operación vectorial típica podría sumar dos vectores de punto flotante de 64 elementos para obtener un vector resultado de 64 elementos. Esta instrucción vectorial es equivalente a un bucle en donde cada iteración calcula uno de los 64 elementos del resultado, actualiza los índices y vuelve al comienzo del bucle.

Las instrucciones vectoriales tienen varias propiedades importantes que resuelven la mayor parte de los problemas mencionados arriba:

- El cálculo de cada resultado es independiente del cálculo de los resultados previos, permitiendo un encauzamiento con más etapas *sin* generación de riesgos entre los datos. Esencialmente, la ausencia de riesgos se determina en el proceso de compilación o mediante la ayuda del programador al decidir que se puede utilizar una instrucción vectorial.

- Una instrucción vectorial especifica una gran carga de trabajo — es equivalente a la ejecución de un bucle entero. Así, el ancho de banda necesario para las instrucciones es menor, mitigando el cuello de botella de Flynn.

- Las instrucciones vectoriales que acceden a memoria tienen un patrón de acceso conocido. Si los elementos del vector están adyacentes, la adquisición del vector se puede conseguir a través de bancos de memoria entrelazados. La alta latencia de iniciar un acceso a memoria principal en comparación con un acceso a caché se amortiza ya que se necesita un único acceso para acceder a todo el vector, en vez de un acceso para cada palabra. Así, el coste de la latencia de acceso a memoria principal se observa una única vez para todo el vector, en vez de una vez por cada palabra del vector.

- Dado que se sustituye un bucle por una instrucción vectorial cuyo comportamiento está predeterminado, los riesgos de control que aparecen en la bifurcación del bucle no existen.

Por estas razones, las operaciones vectoriales pueden realizarse más rápidamente que una secuencia de operaciones escalares sobre el mismo número de datos, dando a los diseñadores una alternativa a incluir este tipo de unidades cuando el dominio de aplicación pueda usar unidades vectoriales frecuentemente.

Como se mencionó anteriormente, los proceadores vectoriales segmentan las operaciones sobre los elementos individuales de un vector. Esta segmentación incluye el acceso a la memoria y el cálculo de las direcciones efectivas. Además, la mayoría de los procesadores vectoriales permite la ejecución simultánea de varias operaciones vectoriales.

3.2 Arquitectura vectorial básica

Un procesador vectorial consta típicamente de una unidad escalar segmentada más una unidad vectorial. Todas las unidades funcionales dentro de la unidad vectorial tienen una latencia de varios ciclos de reloj. Existen dos tipos de arquitecturas en los procesadores vectoriales:

- *Procesadores con registros vectoriales*. En esta arquitectura todas las operaciones vectoriales, excepto la lectura y escritura, se realizan entre registros vectoriales. Desde finales de los 80 todos los fabricantes de computadores vectoriales; lo que incluye a Cray Research (CRAY-1, CRAY-2, X-MP, Y-MP y C-90), los supercomputadores japoneses (NEC SX/2 y SX/3, las Fujitsu VP200 y VP400 y la Hitachi S820), utlilizan este modelo.

- *Procesadores vectoriales memoria-memoria*. En esta arquitectura todas las operaciones vectoriales son del tipo memoria-memoria. Los primeros procesadores vectoriales fueron de este tipo, al igual que los procesadores vectoriales de CDC.

La figura 3.1 muestra la arquitectura básica de un procesadores con registros vectoriales. Este procesador, basado ligeramente en el CRAY-1, nos servirá como base a la largo de este capítulo. Le denominaremos DLXV; su parte entera es el DLX, y su parte vectorial es la extensión vectorial del DLX.

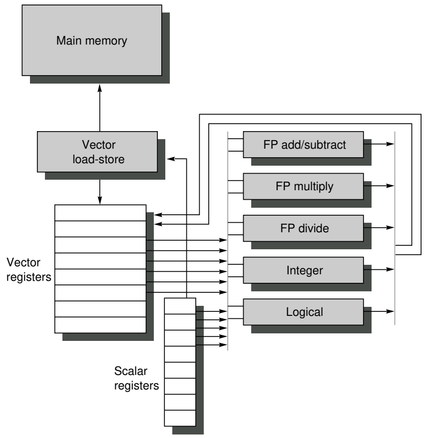

Figura 3.1: Estructura básica de una arquitectura registro vectorial, DLXV.

Los principales componentes del DLXV son

- *Registros vectoriales*. Cada registro vectorial es un banco de longitud fija que permite almacenar un único vector. El DLXV tiene ocho registros vectoriales, cada uno de los cuales almacena 64 elementos.
- *Unidades funcionales vectoriales*. Cada unidad está totalmente segmentada y puede comenzar una nueva instrucción cada ciclo de reloj. El DLXV tiene cinco unidades funcionales como se muestra en la figura 3.1.
- *Unidad de carga/almacenamiento*. Esta es la unidad de memoria vectorial que se encarga de cargar y almacenar vectores desde o en la memoria. Tanto la carga como el almacenamiento está totalmente segmentado, de tal manera que es posible mover palabras entre los registros vectoriales y la memoria con un ancho de banda de una palabra por ciclo de reloj, después de una latencia inicial. Esta unidad se encargará también de manejar las lecturas/escrituras de escalares.
- *Un conjunto de registros escalares*. Los registros escalares también proporcionarán datos como entradas de las unidades vectoriales. El número de registros existentes es el mismo que en el DLX: 32 registros de propósito general y 32 registros de punto flotante.

La figura 3.2 muestra las características de algunos procesadores vectoriales, incluyendo el tamaño y el número de registros, el número y tipo de unidades funcionales, y el número de unidades de carga/almacenamiento.

Processor	Year announced	Clock rate (MHz)	Registers	Elements per register (64-bit elements)	Functional units	Load-store units
CRAY-1	1976	80	8	64	6: add, multiply, reciprocal, integer add, logical, shift	1
CRAY X-MP CRAY Y-MP	1983 1988	120 166	8	64	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	2 loads 1 store
CRAY-2	1985	166	8	64	5: FP add, FP multiply, FP reciprocal/sqrt, integer (add shift, population count), logical	1
Fujitsu VP100/200	1982	133	8-256	32-1024	3: FP or integer add/logical, multiply, divide	2
Hitachi S810/820	1983	71	32	256	4: 2 integer add/logical, 1 multiply-add, and 1 multiply/divide-add unit	4
Convex C-1	1985	10	8	128	4: multiply, add, divide, integer/logical	1
NEC SX/2	1984	160	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
DLXV	1990	200	8	64	5: multiply, divide, add, integer add, logical	1
Cray C-90	1991	240	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4
Convex C-4	1994	135	16	128	3: each is full integer, logical, and FP (including multiply-add)	
NEC SX/4	1995	400	8 + 8192	256 variable	16: 4 integer add/logical, 4 FP multiply/divide, 4 FP add, 4 shift	8
Cray J-90	1995	100	8	64	4: FP add, FP multiply, FP reciprocal, integer/logical	
Cray T-90	1996	~500	8	128	8: FP add, FP multiply, FP reciprocal, integer add, 2 logical, shift, population count/parity	4

Figura 3.2: Características de varias arquitecturas vectoriales.

En DLXV, las operaciones vectoriales usan los mismos nemotécnicos que las operaciones DLX, pero añadiendo la letra V (*ADDV*). Si una de las entradas es un escalar se indicará añadiendo el sufijo "SV" (*ADDSV*). La figura 3.3 muestra las instrucciones vectoriales del DLXV.

Instruction	Operands	Function
ADDV ADDSV	V1,V2,V3 V1,F0,V2	Add elements of V2 and V3, then put each result in V1. Add F0 to each element of V2, then put each result in V1.
SUBV SUBVS SUBSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Subtract elements of V3 from V2, then put each result in V1. Subtract F0 from elements of V2, then put each result in V1. Subtract elements of V2 from F0, then put each result in V1.
MULTV MULTSV	V1,V2,V3 V1,F0,V2	Multiply elements of V2 and V3, then put each result in V1. Multiply F0 by each element of V2, then put each result in V1.
DIVV DIVVS DIVSV	V1,V2,V3 V1,V2,F0 V1,F0,V2	Divide elements of V2 by V3, then put each result in V1. Divide elements of V2 by F0, then put each result in V1. Divide F0 by elements of V2, then put each result in V1.
LV	V1,R1	Load vector register V1 from memory starting at address R1.
SV	R1,V1	Store vector register V1 into memory starting at address R1.
LVWS	V1,(R1,R2)	Load V1 from address at R1 with stride in R2, i.e., R1+i × R2.
SVWS	(R1,R2),V1	Store V1 from address at R1 with stride in R2, i.e., R1+i × R2.
LVI	V1,(R1+V2)	Load V1 with vector whose elements are at R1+V2(i), i.e., V2 is an index.
SVI	(R1+V2),V1	Store V1 to vector whose elements are at R1+V2(i), i.e., V2 is an index.
CVI	V1,R1	Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1.
S--V S--SV	V1,V2 F0,V1	Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--SV performs the same compare but using a scalar value as one operand.
POP	R1,VM	Count the 1s in the vector-mask register and store count in R1.
CVM		Set the vector-mask register to all 1s.
MOVI2S MOVS2I	VLR,R1 R1,VLR	Move contents of R1 to the vector-length register. Move the contents of the vector-length register to R1.
MOVF2S MOVS2F	VM,F0 F0,VM	Move contents of F0 to the vector-mask register. Move contents of vector-mask register to F0.

Figura 3.3: Instrucciones vectoriales del DLXV.

Para entender mejor el funcionamiento de un procesador vectorial, nos centraremos en la ejecución de un bucle típico de álgebra lineal en el DLXV. Supongamos que tenemos que calcular

$$Y = a \times X + y$$

donde X e Y son vectores, inicialmente residentes en memoria, y a es un escalar. A este problema se le denomina bucle SAXPY o DAXPY y forma el bucle interno del benchmark Linpack. (SAXPY viene de *single-precision* *a*×*X* *plus* *Y*; DAXPY viene de *double-precision* *a*×*X* *plus* *Y*.) Linpack es un conjunto de rutinas de álgebra lineal, y rutinas para realizar el método de eliminación de Gauss.

Por ahora, supongamos que el número de elementos, o longitud, de un registro (64) vectorial coincide con el número de elementos de los vectores X e Y.

Ejemplo Mostrar el código del bucle DAXPY en DLX y DLXV. Suponer que la dirección de inicio de X e Y están en Rx y Ry, respectivamente.

Respuesta El la figura se muestra el código para ambos casos.

Si comparamos ambos códigos, podemos extraer una serie de conclusiones. La mas importante es que el procesador vectorial reduce ampliamente el número de instrucciones a captar, ejecutando únicamente 6 instrucciones frente a casi 600 en el caso de DLX. Esta reducción ocurre por dos motivos:


```
      LD     F0,a
      ADDI   R4,Rx,#512   ;última dirección a cargar
loop: LD     F2,0(Rx)     ;carga X(i)
      MULTD  F2,F0,F2     ;a x X(i)
      LD     F4,0(Ry)     ;carga Y(i)
      ADDD   F4,F2,F4     ;a x X(i) + Y(i)
      SD     F4,0(Ry)     ;guarda en Y(i)
      ADDI   Rx,Rx,#8     ;incrementa índice de X
      ADDI   Ry,Ry,#8     ;incrementa índice de Y
      SUB    R20,R4,Rx    ;calcula finalización
      BNZ    R20,loop     ;salta si no se ha terminado
```

(a)

```
LD       F0,A        ;carga escalar a
LV       V1,Rx       ;carga vector X
MULTSV   V2,F0,V1    ;multiplicación vector-escalar
LV       V3,Ry       ;carga vector Y
ADDV     V4,V2,V3    ;suma
SV       Ry,V4       ;guarda el resultado
```

(b)

Figura 3.4: Código del bucle DAXPY para los procesadores: (a) DLX; (b) DLXV.

porque el DLXV trabaja con vectores de 64 elementos y porque la sobrecarga de instrucciones que constituyen casi la mitad del bucle en el DLX no están presentes en el código del DLXV.

Otra diferencia importante es la frecuencia de los interbloqueos en el cauce. En el código DLX cada *ADD* debe esperar a una *MULTD*, y cada *SD* debe esperar a la *ADDD*. En el procesador vectorial, cada instrucción vectorial opera sobre todos los elementos del vector independientemente. Así, la parada del cauce sólo es necesaria una vez por operación vectorial. En este ejemplo, la frecuencia de detención del cauce en el DLX sería 64 veces superior al del DLXV. Las paradas en el cauce del DLX pueden resolverse usando segmentación software o desenrollando el bucle. Sin embargo, la gran diferencia en cuanto a instrucciones no puede reducirse.

Tiempo de ejecución vectorial

El tiempo de ejecución de una secuencia de operaciones vectoriales depende principalmente de tres factores: la longitud de los vectores sobre los cuales se opera, los riesgos estructurales y las dependencias existentes entre los datos. Dada la longitud del vector y la *frecuencia o ratio de inicio*, que es la frecuencia a la cual una unidad vectorial consume nuevos operandos o produce nuevos resultados, podemos calcular el tiempo de una instrucción vectorial. El ratio de inicio suele ser de uno por ciclo de reloj para operaciones individuales. Sin embargo, algunos supercomputadores son capaces de tener instrucciones vectoriales capaces de producir dos o más resultados por ciclos, y otros tienen unidades que pueden no estar totalmente segmentadas. Para simplificar los cálculos, supondremos en adelante que el ratio de inicio es de uno en el resto del capítulo. Por tanto, el tiempo de ejecución para una instrucción vectorial es, aproximadamente, la longitud del vector.

Para simplificar la discusión de la ejecución vectorial y su temporización, usaremos la noción de *convoy*, que es el conjunto de instrucciones vectoriales que pueden ejecutarse de forma conjunta en un periodo de reloj. (Aunque el concepto de convoy se usa en los compiladores vectoriales, no existe una terminología estándar.) Las instrucciones de un convoy *no deben* contener ningún riesgo estructural o de datos (aunque esta restricción se relajará posteriormente). Para facilitar el análisis, supondremos que un convoy de instrucciones debe haber completado su ejecución antes de que otro conjunto de instrucciones (escalares o vectoriales) pueda ejecutarse. Esta restricción se relajará posteriormente en la sección 3.6.

Acompañando a la noción de convoy existe una métrica temporal, denominada *chime*, que puede utilizarse para estimar el rendimiento de una secuencia vectorial formada por convoys. Un *chime* es una medida aproximada del tiempo de ejecución de un secuencia de vectores; esta medida es independiente del la longitud del vector. Así, un secuencia vectorial formada por m convoys se ejecuta en m *chimes*, para un vector de longitud n, esto supone aproximadamente $m \times n$ ciclos de reloj. Un *chime* ignora algunas de las sobrecargas específicas del procesador, muchas de las cuales son dependientes de la longitud del vector.

Ejemplo Mostrar en la siguiente secuencia de código los convoys de instrucciones existentes suponiendo una única copia de cada unidad funcional vectorial. ¿Cuántos *chimes* necesitará la ejecución de esta secuencia vectorial?. ¿Cuántos *chimes* por se necesitan por FLOP (operación en punto flotante)?.

```
LV      V1,Rx      ;carga vector X
MULTSV  V2,F0,V1   ;multiplicación vector-escalar
LV      V3,Ry      ;carga vector Y
ADDV    V4,V2,V3   ;suma
SV      Ry,V4      ;guarda el resultado
```

Respuesta El primer convoy estaría formado por la primera instrucción *LV*. La multiplicación *MULTSV* depende de esta primera instrucción, así que no puede estar en el mismo convoy. La segunda *LV* puede estar en el mismo convoy que la *MULTSV*. La *ADDV* depende de la segunda carga, por lo que debe ir en un tercer convoy. Finalmente, la instrucción *SV* depende de la suma, y debe ir en el siguiente convoy. Por lo tanto, serán necesarios los siguientes convoys:

1. *LV*

2. *MULTSV* *LV*

3. *ADDV*

4. *SV*

Dado que la secuencia requiere cuatro convoys, durará cuatro *chimes*. Dado que la secuencia necesita cuatro *chimes* y existen dos operaciones en punto flotante por resultado, el número de *chimes* por FLOP es de dos.

Otra fuente importante de sobrecarga es el tiempo de inicio (*start-up time*). Este tiempo se debe a la latencia del cauce de la operación vectorial y viene determinado principalmente por el número de etapas de cauce de la unidad funcional usada. Para el DLXV, supondremos las mismas etapas que el CRAY-1, aunque procesadores más modernos pueden tener unidades con menor latencia. Todas la unidades funcionales están totalmente segmentadas. La figura 3.5 muestra la sobrecarga debida al tiempo de inicio en el DLXV.

Ejemplo Mostrar el instante en el que empieza cada convoy y el número de ciclos necesarios para el ejemplo anterior. Comparar el tiempo obtenido con la aproximación en *chimes*. Utilizar los tiempos de inicio indicados en la figura 3.5.

Respuesta La figura 3.6 muestra el resultado para cada convoy, suponiendo que la longitud del vector es n.

Operation	**Start-up penalty**
Vector add	6
Vector multiply	7
Vector divide	20
Vector load	12

Figura 3.5: Penalización por el tiempo de inicio en el DLXV.

Convoy	**Starting time**	**First-result time**	**Last-result time**
1. `LV`	0	12	$11 + n$
2. `MULTSV LV`	$12 + n$	$12 + n + 12$	$23 + 2n$
3. `ADDV`	$24 + 2n$	$24 + 2n + 6$	$29 + 3n$
4. `SV`	$30 + 3n$	$30 + 3n + 12$	$41 + 4n$

Figura 3.6: Tiempos de inicio y del primer y último resultado para los convoys 1-4.

Unidades vectoriales de carga-almacenamiento y el sistema de memoria vectorial

El comportamiento de la unidad vectorial de carga-almacenamiento es mucho más complejo que el de las unidades funcionales aritméticas. El tiempo de inicio de una lectura es el tiempo que tarda en obtenerse la primera palabra desde la memoria. Si el resto del vector puede proporcionarse sin paradas, entonces el ratio de inicio es igual al ratio al cual se pueden captar o almacenar nuevas palabras. Al contrario que en el caso de las unidades funcionales más simples, este ratio no tiene por qué ser de un ciclo de reloj.

Normalmente, las penalizaciones de inicio para las unidades de carga-almacenamiento son mayores que las de las unidades aritméticas (hasta 50 ciclos de reloj para algunos procesadores). Para el DLXV supondremos un tiempo de inicio de 12 ciclos de reloj; por comparación, el CRAY-1 y el CRAY X-MP tienen unos tiempos de inicio entre nueve y 17 ciclos de reloj.

Para mantener un ratio de inicio de una palabra por ciclo de reloj, el sistema de memoria debe ser capaz de producir o aceptar esta cantidad de datos. Esto se consigue mediante la utilización de varios bancos de memoria. La razón de utilizar varios bancos de memoria en vez de memoria entrelazada se debe a varias razones:

1. Muchos procesadores vectoriales soportan varias cargas o almacenamientos por ciclo de reloj. Para soportar múltiples accesos de forma simultánea, el sistema de memoria necesita tener varios bancos de memoria para posibilitar el control del direccionamiento a los bancos independientemente.

2. Muchos procesadores vectoriales soportan la carga y almacenamiento de datos que no son secuenciales. En estos casos, el direccionamiento independiente de bancos se hace necesario.

Ejemplo Supongamos que queremos captar un vector de 64 elementos que empiezan en la dirección 136, y que un acceso a memoria supone 6 ciclos de reloj. ¿Cuántos bancos de memoria debemos tener?. ¿Con que dirección se accede a estos bancos?. ¿Cuándo llegarán los elementos a la CPU?.

Respuesta Con seis ciclos por acceso, necesitamos al menos seis bancos de memoria, pero como queremos que el número de bancos sea potencia de dos, elegiremos ocho bancos. La figura 3.1 muestra las direcciones a las que se accede en cada banco en cada periodo de tiempo.

Beginning at clock no.	**Bank**							
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0	192	136	144	152	160	168	176	184
6	256	200	208	216	224	232	240	248
14	320	264	272	280	288	296	304	312
22	384	328	336	344	352	360	368	376

Tabla 3.1: Direcciones de memoria (en bytes) y momento en el cual comienza el acceso a cada banco.

La figura 3.7 muestra la temporización de los primeros accesos a un sistema de ocho bancos con una latencia de acceso de seis ciclos de reloj. Existen dos observaciones importantes con respecto a las figuras 3.1 y 3.7: La primera es que la dirección exacta proporcionada por un banco está muy determinada por los bits de menor orden; sin embargo, el acceso inicial a un banco está siempre entre las ocho primeras dobles palabras de la dirección inicial. La segunda es que una vez se ha producido la latencia inicial (seis ciclos en este caso), el patrón es acceder a un banco cada n ciclos, donde n es el número total de bancos ($n = 8$ en este caso).

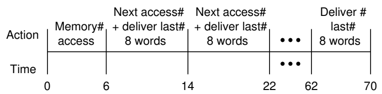

Figura 3.7: Tiempo de acceso para las primeras 64 palabras de doble precisión en una lectura.

3.3 Problemas y soluciones frecuentes

En esta sección trataremos dos temas que aparecen en los programas reales: ¿Qué ocurre cuando la longitud del vector en un programa no coincide con el tamaño de los registros vectoriales?. ¿Cómo se manejan elementos que no son adyacentes en la memoria?.

3.3.1 Control de la longitud del vector

Un procesador vectorial tiene una tamaño de vector natural determinado por el número de elementos de cada registro vectorial. Esta longitud, que es 64 para DLXV, no coincidirá normalmente con la longitud del vector real en un programa. Además, en un programa real la longitud de una operación vectorial particular no se suele saber tiempo de compilación . De hecho, un trozo de código puede requerir longitudes de vector diferentes. Por ejemplo, consideremos el siguiente código:

```
      do 10 i = 1, n
10    Y(i) = a * X(i) + Y(i)
```

El tamaño de la operación vectorial depende de *n*, que no se conoce hasta el tiempo de ejecución del programa. El valor de *n* puede ser incluso un parámetro de un procedimiento en el cual se encuentra el bucle anterior y que está, por tanto, sujeto a cambios durante la ejecución.

La solución a estos problemas es crear un *registro de longitud del vector* (VLR). El VLR controla la longitud de cualquier operación vectorial, incluyendo la carga y el almacenamiento de un vector. Sin embargo, el valor de VLR no puede ser mayor que la longitud de los registros vectoriales. Esto soluciona nuestro problema siempre que la longitud real sea menor que la *longitud máxima del vector* (MVL) definida por el procesador.

¿Qué ocurre si el valor de *n* no se conoce en tiempo de compilación y, por tanto, puede ser mayor que MVL?. Para resolver este segundo problema en donde la longitud del vector es mayor que la longitud máxima, se utiliza una técnica llamada *strip mining*. Esta técnica consiste en la generación de código de tal manera que cada operación vectorial se haga para un tamaño menor o igual que MVL La idea es crear un bucle que maneje cualquier número de iteraciones que sea un múltiplo de MVL y otro bucle que maneje el resto de iteraciones. En la práctica, los compiladores crean un único bloque que se parametriza para manejar ambas porciones cambiando la longitud. La versión modificada de código anterior escrito en FORTRAN, el lenguaje más utilizado en las aplicaciones científicas, con comentarios en estilo C es:

```
      low = 1
      VL = (n mod MVL) /* determinamos el tamaño del resto */
      do 1 j = 0, (n / MVL) /* bucle exterior */
            do 10 i = low, low+VL-1 /* ejecutar para la longitud VL */
                  Y(i) = a*X(i) + Y(i) /* operación principal */
10          continue
            low = low+VL /* empieza el siguiente vector */
            VL = MVL /* se actualiza la longitud a la máxima */
1     continue
```

El efecto del bucle exterior es dividir el vector en segmentos que son procesados en el bucle interior. La longitud del primer segmento es *(n mod MVL)* y todos los demás segmentos son de longitud *MVL*, como se muestra en la figura 3.8.

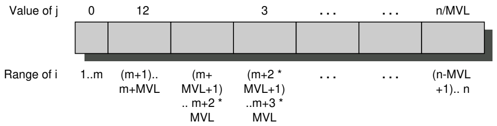

Figura 3.8: Un vector de longitud arbitraria procesado con el *strip mining*. Todos los bloques menos el primero son de longitud MVL. En esta figura, la variable *m* se usa en lugar de la expresión *(n mod MVL)*.

El bucle interior del código de arriba es vectorizable con longitud *VL*, que es igual a *(n mod MVL)* o MVL. El registro VLR debe actualizarse dos veces (una por cada lugar en donde aparece una asignación a la variable *VL*). Con varias operaciones vectoriales ejecutándose en paralelo, el hardware debe copiar el valor de VLR cuando se emite una operación vectorial.

Aparte de la sobrecarga de inicio, es necesario tener en cuenta la sobrecarga de ejecutar el nuevo bucle. Esta sobrecarga, que aparece de la necesidad de reiniciar la secuencia del vector y actualizar VLR, se añade al tiempo de inicio efectivo de una operación vectorial, suponiendo el no solapamiento con otras instrucciones. Si la sobrecarga para cada convoy es de 10 ciclos, la sobrecarga efectiva para 64 elementos se incrementa en 10 ciclos, ó 0.15 ciclos por elemento.

Existen dos factores clave que contribuyen al tiempo de ejecución de un bucle *strip-mined* consistente en una secuencia de convoys:

1. El número de convoys en el bucle, que determina el número de *chimes*. Usaremos la notación T_{chime} para el tiempo de ejecución en *chimes*.
2. La sobrecarga de cada secuencia de convoys. Esta sobrecarga consiste en el coste de ejecución del código escalar de la técnica *strip mining*, T_{loop}, más el coste de inicio de cada convoy, T_{start}.

Además existe un coste fijo asociado con la creación de la primera secuencia vectorial. En los procesadores vectoriales recientes esta sobrecarga se ha reducido sustancialmente, así que la ignoraremos.

Estos componentes se pueden utilizar para obtener una expresión del tiempo de ejecución de una operación vectorial sobre un vector de longitud *n*, que llamaremos T_n:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

Los valores de T_{start}, T_{loop} y T_{chime} son dependientes del compilador y del procesador. La asignación de registros y la reordenación de las instrucciones afectan a la sobrecarga de las operaciones vectoriales y qué instrucciones aparecen en cada una de ellas.

Por simplicidad, usaremos un valor constante para T_{loop}. Basándonos en varias medidas sobre la máquina CRAY-1, el valor elegido es 15 ciclos para T_{loop}. A primera vista este valor puede parecer muy pequeño. La sobrecarga en cada bucle requiere calcular la dirección de comienzo del vector y la longitud del mismo, incrementar los contadores y ejecutar el bucle. En la práctica, estas instrucciones escalares pueden solaparse total o parcialmente con las instrucciones vectoriales, minimizando el

tiempo consumido en estas funciones. El valor de T_{loop} depende, por supuesto, de la estructura del bucle, pero esta dependencia es pequeña en comparación con la conexión entre el código vectorial y los valores de T_{chime} y T_{start}.

Ejemplo ¿Cuál es el tiempo de ejecución en el procesador vectorial DLXV para la operación vectorial $A = B \times s$, donde s es un escalar y la longitud de los vectores A y B es 200?.

Respuesta Supongamos que la direcciones de A y B se encuentran inicialmente en Ra y Rb, y s en Fs, y recordemos que para DLX (y para DLXV) R0 siempre contiene un 0. Dado que $(200\ mod\ 64) = 8$, la primera iteración del bucle ya modificado se ejecutará para una longitud de vector de 8 elementos, y las siguientes iteraciones se ejecutarán para una longitud de vector de 64 elementos. El byte de la dirección de inicio para el siguiente segmento de cada vector es 8 veces la longitud del vector. Dado que la longitud del vector es 8 ó 64, incrementaremos los registros de dirección en $8 \times 8 = 64$ después del primer segmento y $8 \times 64 = 512$ para los siguientes segmentos. El número total de bytes en el vector es $8 \times 200 = 1600$, y testearemos la finalización comparando la dirección del siguiente segmento con la dirección inicial más 1600. En la figura 3.9 se muestra el código real.

```
       ADDI     R2,R0,#1600   ;no. bytes en el vector
       ADD      R2,R2,Ra      ;fin de vector
       ADDI     R1,R0,#8      ;longitud del primer segmento
       MVI2S    VLR,R1        ;longitud del vector
       ADDI     R1,R0,#64     ;longitud en bytes
       ADDI     R3,R0,#64     ;longitud del vector del resto de segmentos
LOOP:  LV       V1,Rb         ;carga B
       MULTVSI  V2,V2,Fs      ;vector * escalar
       SV       Ra,V2         ;almacena A
       ADD      Ra,Ra,R1      ;siguiente segmento de A
       ADD      Rb,Rb,R1      ;siguiente segmento de B
       ADDI     R1,R0,#512    ;longitud del vector (en bytes)
       MOVI2S   VLR,R3        ;inicializa longitud a 64
       SUB      R4,R2,Ra      ;¿Hemos recorrido A?
       BNZ      R4,LOOP       ;si no, volver al bucle
```

Figura 3.9: Código generado para la operación vectorial $A = B \times s$.

Las tres instrucciones vectoriales del bucle son dependientes y deben ir separadas, por lo que $T_{chime} = 3$. Usando nuestra fórmula

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

$$T_{200} = 4 \times (15 + T_{start}) + 200 \times 3$$

$$T_{200} = 60 + (4 \times T_{start}) + 600 = 660 + (4 \times T_{start})$$

El valor de T_{start} es la suma del tiempo de inicio de la carga del vector, 12 ciclos de reloj, el tiempo de inicio de la multiplicación, 7 ciclos de reloj, y el tiempo de inicio del almacenamiento, 12 ciclos de reloj. Así, el valor de T_{start} viene dado por

$$T_{start=12+7+12=31}$$

Así, el valor total de T_n será

$$T_{200} = 660 + 4 \times 31 = 784$$

El tiempo de ejecución por elemento incluyendo el coste de inicio es de $784/200 = 3.9$.

La figura 3.10 muestra la sobrecarga total por elemento y el tiempo de ejecución por elemento para el ejemplo anterior ($A = B \times s$) para varias longitudes del vector. Como se puede observar, para vectores cortos el tiempo de inicio es casi más de la mitad del tiempo total, mientras que para vectores grandes se reduce a cerca de un tercio del tiempo total. Los picos que muestran las gráficas se producen cuando la longitud del vector cruza un múltiplo de 64, forzando a otra iteración del bucle. Estas operaciones incrementa T_n en $T_{loop} + T_{start}$. Los valores asintóticos de ambas curvas son 3.9 ciclos de reloj como tiempo de ejecución total por elemento y 0.9 como sobrecarga total por elemento.

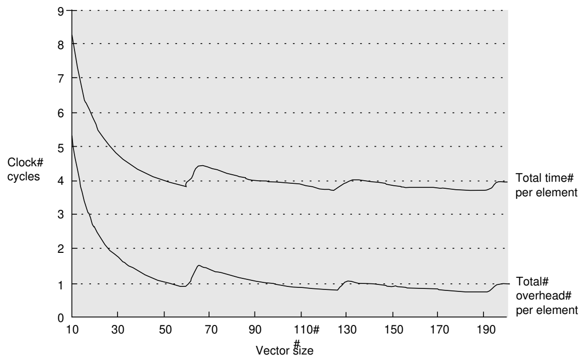

Figura 3.10: Tiempo total de ejecución por elemento y sobrecarga total por elemento, en función de la longitud del vector para el ejemplo $A = B \times s$.

3.3.2 Separación vectorial (Vector Stride)

El segundo problema es que elementos adyacentes en un vector pueden no serlo en la memoria. Consideremos el siguiente código para multiplicar 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)
```


En la sentencia etiquetada como 10 podríamos vectorizar la multiplicación de cada fila de B con cada columna de C y sacar del bucle interno k como variable índice.

Para hacer eso debemos considerar como se direccionan los elementos adyacentes en B y en C. A la hora de guardar un array en memoria, este se encuentra linealizado. Esta linealización significa que los elementos en la fila o los elementos en la columna no son adyacentes en la memoria, dependiendo si la linealización empieza por columnas o filas, respectivamente. Por ejemplo, si el bucle anterior se ha escrito en FORTRAN, que almacena los array por columnas, los elementos de B que son accedidos en las distintas iteraciones del bucle interior están separados por ocho veces el tamaño de una fila dando un total de 800 bytes. Esta distancia de separación entre elementos se guarda en un registro llamado *stride* (*salto*). En este ejemplo, usando almacenamiento por columnas, el salto para C será de 1 (8 bytes) y para B será de 100 (800 bytes).

Una vez que se ha cargado el vector en un registro vectorial, éste actúa como si los elementos fueran lógicamente adyacentes. Por tanto los registros vectoriales deben poder manejar saltos mayores que uno usando únicamente operaciones vectoriales de lectura y escritura en memoria. Esta habilidad de acceder a direcciones de memoria no secuencial y reorganizarlas en una estructura densa es una de las mayores ventajas de un procesador vectorial sobre un procesador basado en caché.

En DLXV, donde la unidad de dirección es el byte, el salto para nuestro ejemplo sería 800. El valor debe ser calculado dinámicamente, dado que el tamaño de la matriz no se conoce en tiempo de compilación, o puede cambiar en diferentes ejecuciones de la misma sentencia. El salto del vector, como la dirección del comienzo del vector, puede ponerse en un registro de propósito general. Después la instrucción *LVWS* (cargar vector con salto) puede consultar este registro en el momento de cargar el vector. En algunos procesadores vectoriales las cargas y los almacenamientos tienen siempre almacenado este valor (*stride*) en un registro, con lo que se pueden utilizar instrucciones de carga y almacenamiento normales.

En el caso de vectores con saltos distintos de 1, pueden aparecer problemas con el sistema de memoria. En el caso de accesos escalares y vectoriales con valor de salto igual a 1, es posible conseguir un acceso continuo a la memoria si el número de bancos de memoria es al menos tan grande como el tiempo de acceso en ciclos de reloj. Cuando se introducen accesos vectoriales con valor de separación distinto de 1, es posible que aparezcan accesos a un mismo banco con un ratio mayor que el tiempo de acceso a la memoria. En este caso se producirá un fenómeno de contención en dicho banco, con lo que uno de los accesos deberá detenerse. Un conflicto a un banco de memoria, y por tanto una parada, se produce si

$$\frac{m.c.m(separación, n^o\ de\ bancos)}{separación} < latencia\ de\ acceso\ a\ la\ memoria$$

Ejemplo Supongamos que tenemos 16 bancos de memoria con una latencia de lectura de 12 ciclos. ¿Cuánto tiempo tardará en completarse una lectura de un vector de 64 elementos con un valor de separación de 1?.¿Y con un valor de separación de 32?.

Respuesta Dado que el número de bancos es mayor que la latencia de lectura, para un valor de salto de 1, la carga tardará 12+64=76 ciclos de reloj, o 1.2 ciclos por elemento. El peor valor de separación es un valor que sea un múltiplo del número de bancos de memoria, como es el caso de un valor de salto de 32 con 16 bancos de memoria. Cualquier acceso a la memoria colisiona con el acceso previo. Esto da lugar a una latencia de 12 ciclos por elemento y un tiempo total de 768 ciclos de reloj.

Los conflictos de acceso a la memoria no ocurren si el número de bancos y el valor de salto son primos relativos y existen suficientes bancos para evitar el caso de *stride* igual a 1. Al incrementar el número de bancos de memoria a un número mayor que el mínimo para evitar paradas para un valor de salto de 1, también se eliminan paradas para valores mayores que uno. Por ejemplo, con 64 bancos, un valor de salto de 32 se bloqueará en uno de cada dos casos, en vez de en cada acceso. En 1995, la mayoría de los supercomputadores vectoriales tiene al menos 64 bancos, y algunos tienen hasta 1024 en su configuración máxima. Dado que los conflictos pueden ocurrir en el caso de valores de salto distinto de 1, muchos programadores favorecen el caso de acceso con valor de salto 1 siempre que sea posible.

3.4 Eficacia de la vectorización mediante compilador

Existen dos factores que afectan al rendimiento de un programa que se ejecuta en modo vectorial. El primer factor es la estructura del propio programa: ¿Tienen los bucles dependencias de datos verdaderas, o pueden reestructurarse para eliminar dichas dependencias?. Este factor viene dado por los algoritmos elegidos y por la codificación de los mismo. El segundo factor es la capacidad del compilador. Mientras que ningún compilador puede vectorizar un bucle donde no existe paralelismo entre las iteraciones del mismo, existe una tremenda variación en la habilidad de los compiladores para determinar si un bucle puede o no ser vectorizado.

La figura 3.2 muestra el porcentaje de operaciones en punto flotante para cada benchmark del Perfect Club y el porcentaje de operaciones ejecutadas en modo vectorial en el CRAY X-MP.

Benchmark name	**FP operations**	**FP operations executed in vector mode**
ADM	23%	68%
DYFESM	26%	95%
FLO52	41%	100%
MDG	28%	27%
MG3D	31%	86%
OCEAN	28%	58%
QCD	14%	1%
SPICE	16%	7%
TRACK	9%	23%
TRFD	22%	10%

Tabla 3.2: Nivel de vectorización obtenido sobre el Cray X-MP para el benchmark *Perfect Club*.

Existe una tremenda variación en el comportamiento de los compiladores para vectorizar un programa. Como resumen de estado de los compiladores actuales, la figura 3.3 muestra el éxito que distintos compiladores tienen en la vectorización de un test formado por 100 kernels FORTRAN escritos a

mano. Estos kernels se diseñaron para comprobar la capacidad de vectorización y pueden vectorizarse en su totalidad.

Processor	**Compiler**	**Completely vectorized**	**Partially vectorized**	**Not vectorized**
CDC CYBER-205	VAST-2 V2.21	62	5	33
Convex C-series	FC5.0	69	5	26
CRAY X-MP	CFT77 V3.0	69	3	28
CRAY X-MP	CFT V1.15	50	1	49
CRAY-2	CFT2 V3.1a	27	1	72
ETA-10	FTN 77 V1.0	62	7	31
Hitachi S810/820	FORT77/HAP V20-2B	67	4	29
IBM 3090/VF	VS FORTRAN V2.4	52	4	44
NEC SX/2	FORTRAN77 / SX V.040	66	5	29

Tabla 3.3: Resultado de aplicar la vectorización mediante compilador a 100 programas de tests FORTRAN.

3.5 Mejora del rendimiento de los procesadores vectoriales

En esta sección estudiaremos tres técnicas para mejorar el rendimiento de los procesadores vectoriales. La primera de ellas intenta hacer que una secuencia de operaciones vectoriales dependientes se ejecute de forma más rápida. Las otras dos tratan de expandir el tipo de bucles que se pueden ejecutar vectorialmente. La primera técnica, en *encadenamiento*, apareció con la CRAY-1, pero en la actualidad la soporta la mayoría de los procesadores vectoriales. Las otras dos técnicas discuten la extensión del procesamiento vectorial a la ejecución condicional y a las matrices dispersas.

3.5.1 Encadenamiento de operaciones vectoriales

Consideremos la siguiente secuencia vectorial

MULTV V1, V2, V3
ADDV V4, V1, V5

En DLXV, como lo hemos visto hasta ahora, estas dos instrucciones se ejecutarían separadamente, dado que son dependientes. Por otra parte, si el registro vectorial, *V1* en este caso, no se trata como una única entidad sino como un grupo de registros individuales, la idea de adelantar la ejecución de las instrucciones puede, conceptualmente, extenderse para trabajar sobre elementos individuales de un vector. A este hecho, que permitiría que la instrucción *ADDV* empiece a ejecutarse de forma anticipada, se le denomina *encadenamiento*. El encadenamiento permite que las operaciones vectoriales empiecen tan pronto como los elementos individuales de sus vectores fuente estén disponibles: Los resultados de la primera unidad funcional en la cadena son enviados a la secunda unidad funcional. Las implementaciones actuales utilizan el *encadenamiento flexible*, que permite el encadenamiento de una instrucción vectorial con cualquier otra instrucción vectorial activa, suponiendo que no se generen riesgos estructurales. El encadenamiento flexible requiere más puertos de lectura y escritura en el banco de registros vectoriales.

Incluso en el caso de que un par de operaciones dependan una de otra, el encadenamiento permite que las operaciones procedan en paralelo sobre elementos separados del vector. Esto permite la emisión conjunta de varias operaciones, disminuyendo el número de *chimes*. Para la secuencia anterior, es posible conseguir un ratio sostenido (ignorando el tiempo de inicio) de dos operaciones vectoriales por ciclo de reloj incluso aunque las operaciones sean dependientes. El tiempo total de ejecución para la secuencia anterior será

$$Longitud\ del\ vector + Tiempo\ de\ inicio_{ADDV} + tiempo\ de\ inicio_{MULTV}$$

La figura 3.11 muestra la temporización de la versión encadenada y sin encadenar del par de instrucciones vectoriales mostradas arriba con una longitud de vector de 64. Este convoy requiere un *chime*: sin embargo, al usar encadenamiento, la sobrecarga de inicio se observará en el tiempo real del convoy. En la figura 3.11, el tiempo total para las operaciones encadenadas es de 77 ciclos, o 1.2 ciclos por resultado. Con 128 operaciones en punto flotante realizadas en ese tiempo, se obtiene 1.7 FLOPs por ciclo. Para la versión no encadenada, estos tiempos son 141 ciclos y 0.9 FLOPs por ciclo.

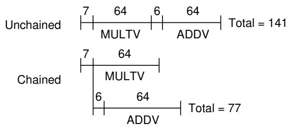

Figura 3.11: Temporización de una secuencia de operaciones vectoriales dependientes ADDV y MULTV, sin y con encadenamiento.

Aunque el encadenamiento nos permite reducir el componente *chime* del tiempo de ejecución poniendo dos instrucciones dependientes en el mismo convoy, no elimina la sobrecarga de inicio. Si queremos un tiempo de ejecución exacto, debemos de tener en cuenta el tiempo de inicio dentro y entre los convoys. Con el encadenamiento el número de *chimes* para una secuencia viene dado por el número de unidades vectoriales diferentes disponibles en el procesador y el número que necesita la aplicación. En particular, ningún convoy puede contener riesgos estructurales. Esto significa, por ejemplo, que una secuencia conteniendo dos instrucciones vectoriales de memoria necesita al menos dos convoys, y por tanto dos *chimes,* en un procesador como el DLXV con una única unidad de carga-almacenamiento.

3.5.2 Sentencias de ejecución condicional

En la sección anterior vimos que muchos programas sólo consiguen un nivel de vectorización de bajo a moderado. Dada la ley de Amdahl, el speedup para dichos programas será muy limitado. Las causas de este poco nivel de vectorización son la presencia de condicionales (sentencias *if*) dentro de los bucles y el uso de matrices dispersas. Los programas que contienen sentencias *if* en los bucles no pueden ejecutarse en modo vectorial usando las técnicas vistas ya que estas sentencias introducen dependencias de control en el bucle. De la misma manera, las matrices dispersas no pueden implementarse eficientemente con los mecanismos vistos hasta ahora; este es uno de los factores de la falta

de vectorización de SPICE. En lo que resta de sección discutiremos estrategias para manejar estas dos situaciones.

Consideremos el siguiente bucle:

```
          do 100 i = 1,64
                if (A(i) .ne. 0) then
                      A(i) = A(i) - B(i)
                endif
100       continue
```

Este bucle no podría ser vectorizado por la presencia de la sentencia condicional; sin embargo, si el bucle interno pudiera ser ejecutado para las iteraciones en las cuales $A(i) \neq 0$, entonces la substracción podría vectorizarse. Un forma para permitir la paralelización del vector es la utilización de un *vector máscara de control*. Este vector máscara de control es un vector booleano de longitud MVL que controla la ejecución de una instrucción vectorial de la misma manera que las instrucciones de ejecución condicional usan una condición booleana para determinar si debe ejecutarse. Cuando el *registro de máscara del vector* está activo, cualquier instrucción vectorial ejecuta la operación únicamente sobre aquellos elementos cuya entrada correspondiente en el registro de máscara esté a uno. Si el registro de máscara se pone a uno mediante el resultado de una condición, únicamente los elementos que satisfacen la condición se verán afectados. El siguiente código muestra la implementación del bucle anterior, suponiendo que las direcciones de comienzo de A y B están en *Ra* y *Rb*, respectivamente:

```
LV      V1,Ra       ;carga el vector A en V1
LV      V2,Rb       ;carga el vector B
LD      F0,#0       ;F0=0
SNESV   F0,V1       ;pone VM a 1 si V1(i)≠F0
SUBV    V1,V1,V2    ;resta utilizando la máscara
CVM                 ;pone el vector de máscara a 1
SV      Ra,V1       ;almacena el resultado en A
```

La mayoría de los recientes procesadores vectoriales proporcionan un control mediante un vector máscara. La capacidad del vector máscara descrita aquí está disponible en algunos procesadores, pero otros sólo permiten el uso del vector máscara para un subconjunto de las instrucciones vectoriales.

El uso de un registro de máscara tiene, sin embargo, sus desventajas. Al examinar la ejecución de instrucciones de ejecución condicional, observamos que estas instrucciones todavía requieren un tiempo de ejecución aún en el caso de que no se satisfaga la condición. Sin embargo, la eliminación del salto y las dependencias de control asociadas pueden hacer que la instrucción condicional se ejecute más rápidamente incluso si a veces se realiza trabajo innecesario. De forma similar, las instrucciones vectoriales ejecutadas con un vector máscara todavía supone un tiempo de ejecución, incluso en el caso donde la máscara es 0. Sin embargo, incluso con un número significativo de ceros en la máscara, el uso del control mediante máscara puede ser más rápido que el uso del modo escalar. De hecho, la gran diferencia en rendimiento potencial entre el modo escalar y vectorial hace que la inclusión de instrucciones de máscara de vector sea crítica.

En segundo lugar, en algunos procesadores vectoriales el vector de máscara sólo sirve para deshabilitar el almacenamiento del resultado en el registro destino, la operación sigue realizándose. Así, si la operación en el ejemplo anterior fuese una división en vez de una resta y el test se realizara sobre B en lugar de A, podría producirse una excepción de punto flotante.

3.5.3 Matrices dispersas

Existen técnicas que permiten la ejecución vectorial de programas que utilizan matrices dispersas. En una matriz dispersa, los elementos de un vector se almacenan en una forma compactada a la que se accede de forma indirecta. Para una representación típica de una matriz dispersa, podríamos ver un código como el que sigue:

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

Este código implementa una suma de un vector disperso sobre los arrays A y C, usando como vectores índices K y M para indicar los elementos no nulos de A y C.

Un primer mecanismo para soportar las matrices dispersas son las *operaciones scatter-gather* usando vectores índice. El objetivo de dichas operaciones es soportar la transición entre una representación densa y una representación normal.Una operación *gather* toma un *vector índice* y capta el vector cuyos elementos están en las direcciones obtenidas de añadir a la dirección base el desplazamiento dado por el vector índice. El resultado es un vector no disperso en un registro vectorial. Después de que los elementos se operan en su forma densa, el vector disperso puede almacenarse en su forma expandida mediante un almacenamiento *scatter*, usando el mismo vector índice. El soporte hardware para estas operaciones se denomina *scatter-gather* y aparece en varios procesadores. Las instrucciones *LVI* (carga de vector indexada) y *SVI* (almacenamiento de vector indexado) proporciona estas operaciones en el DLXV. Por ejemplo, suponiendo que Ra, Rc, Rk y Rm contienen las direcciones de comienzo de los vectores de la secuencia arriba indicada, el bucle interno de la secuencia puede codificarse con instrucciones vectoriales como sigue

```
LV     Vk,Rk         ;carga K
LVI    Va,(Ra+Vk)    ;carga A(K(I))
LV     Vm,Rm         ;carga M
LVI    Vc,(Rc+Vm)    ;carga C(M(I))
ADDV   Va,Va,Vc      ;los suma
SVI    (Ra+Vk),Va    ;almacena A(K(I))
```

Esta técnica permite que el código de las matrices dispersas puedan ejecutarse en modo vectorial. El código fuente descrito arriba *nunca* sería vectorizado por un compilador ya que el compilador no puede saber que los elementos de K son valores distintos, y que no existen dependencias. En su lugar, una directiva le indicaría al compilador que el bucle puede ejecutarse en modo vectorial; sin dicha directiva, programas tales como el SPICE no podrían vectorizarse incluso si existiera el soporte hardware.

La capacidad *scatter-gather* está incluida en muchos de los supercomputadores recientes. Estas operaciones rara vez alcanzan la velocidad de un elemento por ciclo, pero son mucho más rápidas que la

alternativa de utilizar un bucle escalar. Si la propiedad de dispersión de un matriz cambia, es necesario calcular un nuevo vector índice. Muchos procesadores proporcionan soporte para un cálculo rápido de dicho vector. La instrucción *CVI* (create vector index) del DLX crea un vector índice dado un valor de salto (m), cuyos valores son $0, m, 2 \times m, ..., 63 \times m$. Algunos procesadores proporcionan una instrucción para crear un vector índice comprimido cuyas entradas se corresponden con las posiciones a 1 en el registro máscara. En DLX, definimos la instrucción *CVI* para que cree un vector índice usando el vector máscara. Cuando el vector máscara tiene todas sus entradas a uno, se crea un vector índice estándar.

Las cargas y almacenamientos indexados y la instrucción *CVI* proporcionan un método alternativo para soportar la ejecución vectorial condicionada. A continuación se muestra la secuencia de instrucciones que implementa el bucle que vimos al estudiar este problema.

```
LV       V1,Ra          ;carga el vector A en V1
LV       F0,#0          ;F0=0
SNESV    F0,V1          ;pone VM a 1 si V1(i)≠0
CVI      V2,#8          ;genera los índices en V2
POP      R1,VM          ;determina el número de unos en VM
MOVI2S   VLR,R1         ;carga el registro longitud del vector
CVM                     ;pone a 1 la máscara
LVI      V3,(Ra+V2)     ;carga los elementos no nulos de A
LVI      V4,(Rb+V2)     ;carga los elementos correspondientes de B
SUBV     V3,V3,V4       ;realiza la resta
SVI      (Ra+V2),V3     ;almacena A
```

Si la implementación utilizando *scatter-gather* es mejor que la versión utilizando ejecución condicional depende de la frecuencia con la que se cumpla la condición y el coste de las operaciones. Ignorando el encadenamiento, el tiempo de ejecución para la primera versión es $5n + c_1$. El tiempo de ejecución de la segunda versión, utilizando cargas y almacenamiento indexados con un tiempo de ejecución de un elemento por ciclo, es $4n + 4 \times f \times n + c_2$, donde f es la fracción de elementos para la cual la condición es cierta (es decir, $A \neq 0$. Si suponemos que los valores c_1y $c_2$son comparables, y que son mucho más pequeños que n, entonces para que la segunda técnica sea mejor que la primera se tendrá que cumplir

$$5n \geq 4n + 4 \times f \times n$$

lo que ocurre cuando $\frac{1}{4} \geq f$.

Es decir, el segundo método es más rápido que el primero si menos de la cuarta parte de los elementos son no nulos. En muchos casos la frecuencia de ejecución es mucho menor.

3.6 Medición del rendimiento de los procesadores vectoriales

En esta sección describiremos las medidas de rendimiento utilizadas en los procesadores vectoriales. Para determinar el rendimiento de un procesador sobre un problema vectorial debemos observar tanto el coste de inicio como el ratio sostenido. La forma mejor y más sencilla de determinar el rendimiento de un procesador vectorial en un bucle es dar el tiempo de ejecución del bucle vectorial. Para bucles vectoriales se suele utilizar como medida el número de MFLOPS más que el tiempo de ejecución. Usaremos la notación R_n para el ratio de MFLOPS sobre un vector de longitud *n*.

En esta sección examinaremos el rendimiento del DLXV en el bucle DAXPY midiendo el rendimiento desde diferentes puntos de vista. A continuación calcularemos el tiempo de ejecución de un bucle vectorial. Al mismo tiempo, examinaremos diferentes formas de medir el rendimiento usando el tiempo calculado.

3.6.1 Medidas del rendimiento vectorial

Dado que la longitud del vector es tan importante en el establecimiento del rendimiento de un procesador, veremos las medidas relacionadas con la longitud además del tiempo de ejecución y los MFLOPS obtenidos. Estas medidas relacionadas con la longitud tienden a varias de forma muy importante dependiendo del procesador y que son importantes de comparar. (Recordar, sin embargo, que el *tiempo* es siempre la medida de interés cuando se compara la velocidad relativa de dos procesadores.) Las tres medidas más importantes relacionadas con la longitud son

- R_∞. El ratio de MFLOPS para un vector de longitud infinita. Aunque esta medida puede ser de utilidad para medir el rendimiento máximo, los problemas reales no manejan vectores ilimitados, y la sobrecarga existente en los problemas reales puede ser mayor.
- $N_{1/2}$. La longitud de vector necesaria para alcanzar la mitad de R_∞. Esta es una buena medida del impacto de la sobrecarga.
- N_v. La longitud de vector necesaria para hacer que el modo vectorial sea más rápido que el modo escalar. Esta medida tiene en cuenta la sobrecarga y la velocidad relativa del modo escalar respecto al vectorial.

Veamos como se pueden determinar estas medidas en el problema DAXPY ejecutado en el DLXV. Cuando existe el encadenamiento de instrucciones, el bucle interior del código DAXPY en convoys es el que se muestra en la figura 3.12 (suponiendo que Rx y Ry contienen la dirección de inicio).

`LV V1,Rx`	`MULTSV V2,F0,V1`	`Convoy 1: chained load and multiply`
`LV V3,Ry`	`ADDV V4,V2,V3`	`Convoy 2: second load and ADD, chained`
`SV Ry,V4`		`Convoy 3: store the result`

Figura 3.12: Formación de convoys en el bucle interior del código DAXPY.

El tiempo de ejecución de un bucle vectorial con n elementos, T_n, es:

$$T_n = \left\lceil \frac{n}{MVL} \right\rceil \times (T_{loop} + T_{start}) + n \times T_{chime}$$

El encadenamiento permite que el bucle se ejecute en tres *chimes* (y no menos, dado que existe un cauce de memoria); así $T_{chime} = 3$. Si T_{chime} fuera una indicación completa del rendimiento, el bucle podría ejecutarse a un ratio de $2/3 \times$ratio del reloj MFLOPS (ya que existen dos FLOPs por iteración).

Así, utilizando únicamente T_{chime}, un DLXV a 200-MHz ejecutaría este bucle a 133 MFLOPS suponiendo la no existencia de *strip-mining* y el coste de inicio. Existen varias maneras de aumentar el rendimiento: añadir unidades de carga-almacenamiento adicionales, permitir el solapamiento de convoys para reducir el impacto de los costes de inicio, y decrementar el número de cargas necesarias mediante la utilización de registros vectoriales.

3.6.2 Rendimiento máximo del DLXV en el DAXPY

En primer lugar debemos determinar el significado real del rendimiento máximo, R_∞. Por ahora, continuaremos suponiendo que un convoy no puede comenzar hasta que todas las instrucciones del convoy anterior hayan finalizado; posteriormente eliminaremos esta restricción. Teniendo en cuenta esta restricción, la sobrecarga de inicio para la secuencia vectorial es simplemente la suma de los tiempos de inicio de las instrucciones:

$$T_{start} = 12 + 7 + 12 + 6 + 12 = 49$$

Usando $MVL = 64$, $T_{loop} = 15$, $T_{start} = 49$, y $T_{chime} = 3$ en la ecuación del rendimiento, y suponiendo que *n* no es un múltiplo exacto de 64, el tiempo para una operación de *n* elementos es

$$T_n = \left\lceil \frac{n}{64} \right\rceil \times (15 + 49) + 3n = (n + 64) + 3n = 4n + 64$$

El ratio sostenido está por encima de 4 ciclos de reloj por iteración, más que el ratio teórico de 3 chimes, que ignora los costes adicionales. La mayor parte de esta diferencia es el coste de inicio para cada bloque de 64 elementos (49 ciclos frente a 15 de la sobrecarga del bucle).

Podemos calcular R_∞ para una frecuencia de reloj de 200 MHz como

$$R_\infty = lim_{n\to\infty} \left(\frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{Ciclos\ de\ reloj\ por\ iteración} \right)$$

El numerador es independiente de *n*, por lo que

$$R_\infty = \frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{lim_{n\to\infty}\,(Ciclos\ de\ reloj\ por\ iteración)}$$

$$lim_{n\to\infty}\,(Ciclos\ de\ reloj\ por\ iteración) = lim_{n\to\infty} \left(\frac{T_n}{n} \right) = lim_{n\to\infty} \left(\frac{4n + 64}{n} \right) = 4$$

$$R_\infty = \frac{2 \times 200\ MHz}{4} = 100\ MFLOPS$$

El rendimiento sin el coste de inicio, que es el rendimiento máximo dada la estructura de la unidad funcional vectorial, es 1.33 veces superior. En realidad, la distancia entre el rendimiento pico y el sostenido puede ser incluso mayor.

3.6.3 Rendimiento sostenido del DLXV en el Benchmark Linpack

El benchmark Linpack es una eliminación de Gauss sobre una matriz de 100×100. Así, la longitud de los elementos va desde 99 hasta 1. Un vector de longitud k se usa k veces. Así, la longitud media del vector viene dada por

$$\frac{\sum_{i=1}^{99} i^2}{\sum_{i=1}^{99} i} = 66.3$$

Ahora podemos determinar de forma más precisa el rendimiento del DAXPY usando una longitud de vector de 66.

$$T_n = 2 \times (15 + 49) + 3 \times 66 = 128 + 198 = 326$$

$$R_{66} = \frac{2 \times 66 \times 200\, MHz}{326} = 81\, MFLOPS$$

El rendimiento máximo, ignorando los costes de inicio, es 1.64 veces superior que el rendimiento sostenido que hemos calculado. En realidad, el benchmark Linpack contiene una fracción no trivial de código que no puede vectorizarse. Aunque este código supone menos del 20% del tiempo antes de la vectorización, se ejecuta a menos de una décima parte del rendimiento cuando se mide en FLOPs. Así, la ley de Amdahl nos dice que el rendimiento total será significativamente menor que el rendimiento estimado al analizar el bucle interno.

Dado que la longitud del vector tiene un impacto significativo en el rendimiento, las medidas $N_{1/2}$ y N_v se usan a menudo para comparar máquinas vectoriales.

Ejemplo ¿Cuál es $N_{1/2}$ para el bucle interno de DAXPY para el DLXV con un reloj de 200 MHz?

Respuesta Usando R_∞ como ratio máximo, queremos saber para qué longitud del vector obtendremos 50 MFLOPS. Empezaremos con la fórmula para MFLOPS suponiendo que las medidas se realizan para $N_{1/2}$ elementos:

$$MFLOPS = \frac{FLOPs\ ejecutados\ en\ N_{1/2}\ iteraciones}{Ciclos\ de\ reloj\ para\ ejecutar\ N_{1/2}\ iteraciones} \times \frac{Ciclos\ de\ reloj}{Segundos} \times 10^{-6}$$

$$50 = \frac{2 \times N_{1/2}}{T_{N_{1/2}}} \times 200$$

Simplificando esta expresión y suponiendo que $N_{1/2} \leq 64$, tenemos que $T_{n\leq 64} = 1 \times 64 + 3 \times n$, lo que da lugar a

$$T_{N_{1/2}} = 8 \times N_{1/2}$$

$$1 \times 64 + 3 \times N_{1/2} = 8 \times N_{1/2}$$

$$5 \times N_{1/2} = 64$$

$$N_{1/2} = 12.8$$

Por lo tanto, $N_{1/2} = 13$; es decir, un vector de longitud 13 proporciona aproximadamente la mitad del rendimiento máximo del DLXV en el bucle DAXPY.

Ejemplo ¿Cuál es la longitud del vector, N_v, para que la operación vectorial se ejecute más rápidamente que la escalar?

Respuesta De nuevo, sabemos que $R_v < 64$. El tiempo de una iteración en modo escalar se puede estimar como $10 + 12 + 12 + 7 + 6 + 12 = 59$ ciclos de reloj, donde 10 es el tiempo estimado de la sobrecarga del bucle. En el ejemplo anterior se vio que $T_{n\leq 64} = 64 + 3 \times n$ ciclos de reloj. Por lo tanto,

$$64 + 3 \times N_v = 59N_v$$

$$N_v = \left\lceil \frac{64}{56} \right\rceil$$

$$N_v = 2$$

3.6.4 Rendimiento del DAXPY en un DLXV mejorado

El rendimiento del DAXPY, como en muchos problemas vectoriales, viene limitado por la memoria. Consecuentemente, éste se puede mejorar añadiendo más unidades de acceso a memoria. Esta es la principal diferencia arquitectónica entre el CRAY X-MP (y los procesadores posteriores) y el CRAY-1. El CRAY X-MP tiene tres cauces de acceso a memoria, en comparación con el único cauce a memoria del CRAY-1, permitiendo además un encadenamiento más flexible. ¿Cómo afectan estos factores al rendimiento?.

Ejemplo ¿Cuál sería el valor de T_{66} para el bucle DAXPY en el DLXV si añadimos dos cauces más de acceso a memoria?.

Respuesta Con tres canales de acceso a memoria, todas las operaciones caben en un único convoy. Los tiempos de inicio son los mismos, por lo que

$$T_{66} = \left\lceil \frac{66}{64} \right\rceil \times (T_{loop} + T_{start}) + 66 \times T_{chime}$$

$$T_{66} = 2 \times (15 + 49) + 66 \times 1 = 194$$

Con tres cauces de acceso a memoria, hemos reducido el tiempo para el rendimiento sostenido de 326 a 194, un factor de 1.7. Observar el efecto de la ley de Amdahl: Hemos mejorado el ratio máximo teórico, medido en el número de *chimes*, en un factor de 3, pero la mejora total es de 1.7 en el rendimiento sostenido.

Otra mejora se puede conseguir del solapamiento de diferentes convoys y del coste del bucle escalar con las instrucciones vectoriales. Esta mejora requiere que una operación vectorial pueda usar una unidad funcional antes de que otra operación haya finalizado, complicando la lógica de emisión de instrucciones.

Para conseguir una máxima ocultación de la sobrecarga del *strip-mining*, es necesario poder solapar diferentes instancias del bucle, permitiendo la ejecución simultánea de dos instancias de un convoy y del código escalar. Esta técnica, denominada *tailgating*, se usó en el Cray-2. Alternativamente, podemos desenrollar el bucle exterior para crear varias instancias de la secuencia vectorial utilizando diferentes conjuntos de registros (suponiendo la existencia de suficientes registros). Permitiendo el máximo solapamiento entre los convoys y la sobrecarga del bucle escalar, el tiempo de inicio y de la ejecución del bucle sólo sería *observable* una única vez en cada convoy. De esta manera, un procesador con registros vectoriales puede conseguir unos costes de inicio bajos para vectores cortos y un alto rendimiento máximo para vectores muy grandes.

Ejemplo ¿Cuales serían los valores de R_∞ y T_{66} para el bucle DAXPY en el DLXV si añadimos dos cauces más de acceso a memoria y permitimos que los costes del *strip-mining* y de inicio se solapen totalmente?.

Respuesta

$$R_\infty = lim_{n\to\infty}\left(\frac{Operaciones\ por\ iteración \times frecuencia\ de\ reloj}{Ciclos\ de\ reloj\ por\ iteración}\right)$$

$$lim_{n\to\infty}\left(Ciclos\ de\ reloj\ por\ iteración\right) = lim_{n\to\infty}\left(\frac{T_n}{n}\right)$$

Dado que la sobrecarga sólo se observa una vez, $T_n = n + 49 + 15 = n + 64$. Así,

$$lim_{n\to\infty}\left(\frac{T_n}{n}\right) = lim_{n\to\infty}\left(\frac{n+64}{n}\right) = 1$$

$$R_\infty = \frac{2 \times 200\ MHz}{1} = 400\ MFLOPS$$

Añadir unidades adicionales de acceso a memoria y una lógica de emisión más flexible da lugar a una mejora en el rendimiento máximo de un factor de 4. Sin embargo, $T_{66} = 130$, por lo que para vectores cortos, la mejora en el rendimiento sostenido es de $\frac{326}{100} = 2.5$ veces.

Para finalizar, la figura 3.13 muestra una comparación de la diferencia de rendimiento entre los procesadores vectoriales y los procesadores superescalares de última generación. En esta figura podemos comprobar como en los últimos años se ha ido reduciendo la diferencia en rendimiento de ambas arquitecturas.

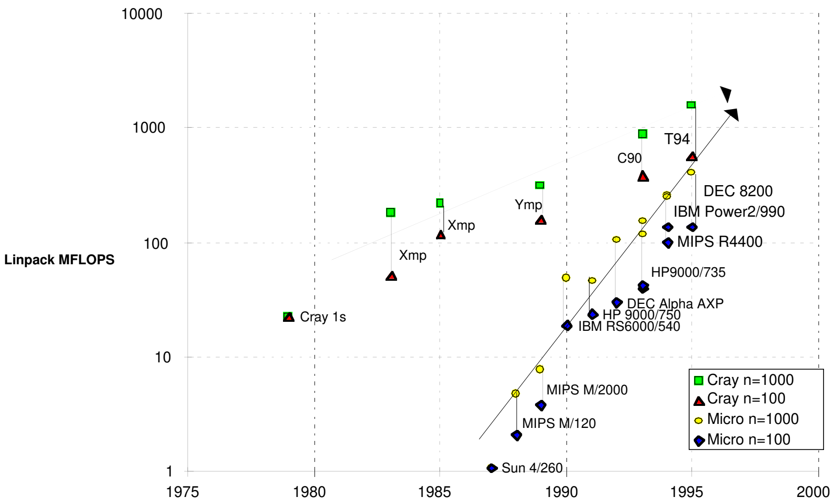

Figura 3.13: Comparación del rendimiento de los procesadores vectoriales y los microprocesadores para la resolución de un sistema de ecuaciones lineales denso (tamaño de la matriz=$n \times n$).