Programación Vectorial

Rafael Valdés Valdazo, Angel María Vilaboa Pérez

UO196558@uniovi.es, UO197092@uniovi.es

Resumen. Este documento trata sobre los fundamentos básicos de la programación vectorial, sus características, las ventajas que representa frente a una programación escalar y como convertir un código escalar a uno vectorial. Para ello se desarrollan varios ejemplos sobre problemas conocidos donde se explican las numerosas ventajas. También se muestra un desarrollo práctico para el problema DAXPY ejecutado en una máquina de arquitectura DLX y otra DLXV, buscando mostrar una comparativa real entre la programación escalar y vectorial.

Palabras Clave: Paralelismo explícito, paralelismo implícito, tiempo de arranque, vectorización, velocidad de iniciación.

1 Introducción

La programación vectorial está unida al concepto de vector. Cabe destacar que dicho concepto se refiere a un conjunto de datos del mismo tipo almacenados en memoria normalmente en posiciones contiguas pero no tiene por qué ser así ya que imaginase una matriz de dos dimensiones que está almacenada en memoria por filas. Si consideramos que un vector es una fila sí que se cumple que los datos están almacenados de forma contigua en la memoria pero si se considera como vectores las columnas o incluso las diagonales se pierde esa contigüidad en memoria.

Decimos que la programación vectorial está unida al concepto de vector e hicimos hincapié en ese concepto debido a que la principal diferencia entre la programación vectorial y escalar estriba en que en la vectorial, utilizada en procesadores vectoriales, puede decodificar instrucciones cuyos operandos sean vectores por lo que pueden realizar operaciones tanto aritméticas como lógicas sobre las componentes de dichos vectores en una única instrucción. En este contexto surge a su vez el proceso de vectorización que se basa en la conversión de un programa correspondiente a un procesador escalar a otro vectorial.

A continuación se mostrará una parte del conjunto de operaciones vectoriales que se pueden realizar sobre operandos vectoriales (vectores) compuestos por un conjunto de n componentes (nos referiremos a dicho n como tamaño) donde cada componente es un escalar (por ejemplo entero, punto flotante…).

tipo	nemotécnico	descripción	
f1	VSQR	raíz cuadrada	$B(I) \leftarrow \sqrt{A(I)}$
	VSIN	seno	$B(I) \leftarrow sen(A(I))$
			$A(I) \leftarrow \overline{A(I)}$
f2	VSUM	suma de componentes	$S = \sum_{I=1}^{N} A(I)$
	VMAX	componente máxima	$S = max\ A(I)$
f3	VADD	suma de vectores	$C(I) = A(I) + B(I)$
	VMPY	multiplicación de vectores	$C(I) = A(I) * B(I)$
	VAND	AND	$C(I) = A(I)\ AND\ B(I)$
	VLAR	mayor vector	$C(I) = max(A(I).B(I))$
	VTGE	comparación	$C(I) = 0\ si\ A(I) < B(I)$
			$C(I) = 1\ si\ A(I) > B(I)$

Fig. 1. Instrucciones Vectoriales.

Cada una de estas instrucciones vectoriales (junto con el resto que no se reflejan por ser un número elevado de ellas y que no afectan a los conceptos que se tratan de desarrollar en este trabajo) es equivalente a la ejecución de un bucle completo de instrucciones ordinarias en el que en cada iteración del bucle trabajaría sobre cada una de las componentes del vector.

Todo esto nos permite establecer un conjunto de ventajas que nos ofrece la vectorización frente a la programación escalar como son:

1. En las operaciones vectoriales cada resultado es independiente de los anteriores lo que nos permite efectuar los cálculos sin que existan conflictos debido a dependencias de datos. Dichos riesgos por dependencias de datos los ha resuelto el compilador o el programador que ha decidido que sea una operación vectorial.
2. Una única instrucción vectorial es equivalente a ejecutar un bucle completo lo que es requerido un ancho de banda a la hora de la lectura de esa instrucción comparándose con el que producirían el conjunto de instrucciones escalares a las que equivaldría lo que nos reduce el efecto del cuello de botella considerablemente.
3. Cuando es necesario acceder a memoria en las instrucciones vectoriales se hace mediante un patrón fijo (normalmente serán adyacentes) lo que facilita su lectura tanto como si se dispone de memoria entrelazada como si no ya que las posiciones de memoria adyacentes se cargarán en caché lo que nos produce un considerable ahorro de tiempo.

4. Al sustituirse un bucle completo por una instrucción vectorial cuyo comportamiento ya está predeterminado los riesgos de control que podrían surgir del salto del bucle son inexistentes.

Debido a estas razones las operaciones vectoriales pueden ejecutarse de forma mucho más rápida que lo que tardaría la secuencia de instrucciones equivalentes sobre ese mismo conjunto de datos lo que hace que se tienda a diseñar y fabricar máquinas vectoriales que soporten estas instrucciones en aplicaciones donde con cierta frecuencia se ejecuten este tipo de operaciones.

2 Características de los lenguajes vectoriales

Para evitar la pérdida del paralelismo de un algoritmo vectorizable es necesaria la existencia de lenguajes de alto nivel que estén adecuados a los componentes vectoriales. Estos lenguajes deben tener una serie de propiedades para expresar el paralelismode los algoritmos y así explotarlo con más eficiencia. Dichas propiedades son las siguientes:

1. Deben poseer un alto grado de flexibilidad para declarar diferentes clases de objetos con distintas estructuras y formas de almacenamiento, es decir, el lenguaje debe poder expresar las diferentes formas de almacenar las componentes de un mismo objeto, por ejemplo por filas, columnas o diagonales.
2. El lenguaje debe ser eficaz para la manipulación de matrices y vectores dispersos, es decir, con componentes nulas, ya que son muy habituales en problemas reales, y por ellos el lenguaje de programación debe suministrar los medios suficientes para almacenarlos sin ocupar excesiva memoria.
3. Deben disponer de operaciones vectoriales nativas que trabajen directamente sobre las estructuras de datos comentadas durante este documento sin necesidad de bucles. Generalmente será el compilador del lenguaje el que transforme esas operaciones de alto nivel en las instrucciones vectoriales de la máquina y deberá ser capaz de aplicar técnicas de seccionamiento para descomponer algunas instrucciones de alto nivel en otras más sencillas cuando el número de componentes del vector sea superior a la capacidad de los registros vectoriales.

Todo lo expuesto anteriormente no significa que los lenguajes no permitan programación mediante bucles ya que puede haber muchas operaciones combinadas que solo puedan detallarse mediante bucles específicos ya que resulta evidente que los lenguajes vectoriales no pueden tener como operaciones nativas todas las posibles combinaciones de operaciones vectoriales ya que podrían ser infinitas aunque si poseen las más frecuentes.

2.1 Conversión de código escalar en código vectorial

Existen dos formas de generar código vectorial: La programación manual o también llamada paralelismo explícito o la vectorización automática o también llamada Paralelismo implícito.

En el paralelismo explícito se consigue el código vectorial usando lenguajes vectoriales. Actualmente existen pocos lenguajes vectoriales y no existe una normalización aceptada para estos lenguajes.

En la otra opción, paralelismo implícito, se utilizan compiladores capaces de vectorizar un código escalar, es decir la programación se realiza en un lenguaje de programación secuencial y es el compilador el que se encarga de generar código paralelo para aquellas partes del programa que pueden ser vectorizadas. En medida que el compilador sea capaz de vectorizar más el código, el rendimiento mejorará en la misma medida.

Ejemplos de lenguaje que soportaría el paralelismo implícito serían el HPF (High Performance Fortran) que es una extensión de Fortran 90 con constructores que soportan computación paralela, Id un lenguaje de programación paralelo de propósito general o MATLAB M-Code.

Existen una serie de dificultades en la vectorización del código debidas a las instrucciones de control y especialmente en las condicionales, dependencia de datos y las indexiones indirectas, es decir componentes de vectores o matrices que son índice de otros vectores o matrices, que sólo se resuelven en ejecución.

Por ello para los lenguajes de programación imperativo la complejidad del problema es casi prohibitivo y permite que los resultados sean positivos sólo para un selecto conjuntos de aplicaciones como por ejemplo, las aplicaciones que realizan operaciones intensivas en matrices.

Las ventajas de la vectorización automática o parelelismo implícito serían que el programador que escribe el código no tiene por qué preocuparse por la división de tareas y solo se centra en el problema de que su programa está destinado a resolver por lo que facilita el diseño de programas paralelos, aumentando sustancialmente la productividad del programador.

En cambio las desventajas que posee frente al paralelismo explícito serían que se reduce el control que el programador tiene sobre la ejecución paralela y a veces resultan menos óptimos (si un vector tuviese un número reducido de componentes como por ejemplo 3 sería ineficiente realizar esta práctica pero el compilador la realizaría transparentemente al programador).

Como inconveniente del paralelismo explícito, según los creadores del lenguaje de programación Oz descubrieron investigando que con esta forma de vectorización la depuración de los programas es mucho más dificultosa.

A continuación se va a mostrar los resultados obtenidos en unos experimentos que tratan de reflejar una comparativa de rendimiento entre 3 modelos de vectorización distintos[1].

---

[1] Sisal: Un lenguaje paralelo implícito.
Sr y C: Ambos lenguajes son explícitos (Sr imperativo).

Para ello se utilizará un procesador Silicon Graphics Iris 4D/340 con memoria compartida, cuatro procesadores de 33Mhz, 64Mbytes de memoria principal, 64Kbytes de caché de datos, 64Kbytes de caché de instrucciones y 256Kbytes de cache de datos secundaria.

Los resultados obtenidos en un programa de multiplicación de matrices son los siguientes (las medidas están en segundos):

	400 × 400			500 × 500			600 × 600		
CPUs	SISAL	SR	C	SISAL	SR	C	SISAL	SR	C
1	34.5	60.5	28.2	66.9	118.	56.2	115.0	215.	95.6
2	17.7	30.7	14.6	34.5	60.0	29.3	59.6	114.	52.9
3	12.3	20.8	10.2	23.7	40.8	21.2	40.9	77.8	38.0
4	9.12	16.0	8.73	19.2	31.1	16.7	31.9	60.4	29.7

Cabe destacar que el lenguaje de programación C guarda las matrices en el orden de las filas, por lo tanto el acceso a la columna de la matriz de los resultados requiere menos fallos en la caché.

Los resultados obtenidos en un programa que realiza el método de iteración de Jacobi son los siguientes (las medidas también están en segundos):

	75 × 75			100 × 100			150 × 150		
CPUs	SISAL	SR	C	SISAL	SR	C	SISAL	SR	C
1	24.6	117.	22.9	58.6	209.	51.3	178.	772.	162.
2	12.5	59.8	11.4	36.7	105.	25.0	90.9	381.	76.0
3	12.1	39.2	7.47	26.3	70.8	16.8	70.2	254.	47.4
4	8.54	30.5	5.82	21.5	53.4	12.5	52.8	196.	35.5

Fig. 2. Resultados para método de iteración Jacobi

De esto se puede observar que SR, método explícito es la que peores resultados demuestra y sin embargo C, el otro método explícito, el que los presenta mejores. Esto quiere decir que el paralelismo implícito es mejor si el compilador crea un programa eficiente.

Cabe destacar que tanto el programa en C como el Programa en Sr el programador utiliza sólo dos matrices, mientras que los programas de SISAL asignan y borran una matriz en cada iteración ya que el programador sabe que la matriz puede ser re-utilizado en el siguiente iteración, sin embargo, esto no puede ser expresado en SISAL.

Sin embargo vamos a mostrar los resultados obtenidos con el problema de Cuadratura Adaptativa con dos programas diferentes para SR, Bag y Prune.

Prune es un programa que limita el número de número de tareas que se insertan en la bolsa para mejorar el resultado[2].

	SISAL		SR		
CPUs	osc	fsc	*bag*	*prune*	C
1	26.7	23.7	25.8	18.2	7.79
2		12.1	15.3	9.10	3.87
3		8.17	13.3	6.07	2.61
4		6.65	22.5	5.64	2.18

Fig. 3. Resultados para Prume

Aquí se ve que Sr "gana" a SISAL cuando ejecuta el programa prune y sin embargo más lento cuando ejecuta el bag. Esto se debe a que el programador de prune fue más hábil a la hora de realizar su trabajo ya que tuvo en cuenta cuestiones de pivotage y otras cuestiones relacionadas con el problema, por lo que la eficiencia está relacionada con las capacidades del programador.

3 Comparativa Arquitectura Escalar y Arquitectura Vectorial

A continuación se mostrará una comparativa en la ejecución del siguiente programa, llamado DAXPY:

$$\mathbf{Y=A*X+Y} \quad (1)$$

Donde A es un número flotante, y tanto X como Y son vectores de flotantes. Para ello se utilizará la arquitectura escalar vista en clase DLX y DLXV para la vectorial, junto con las herramientas correspondientes WinDLX y WinDLXV.

Se debe tener en cuenta el cálculo de los tiempos vectoriales, ya que los escalares ya están vistos en clase y no se considera mencionarlos explícitamente.

En un computador vectorial podemos distinguir dos tiempos en la ejecución de instrucciones en las unidades funcionales:

[2] Fsc: compilador modificado de Sisal.

- *Tiempo de arranque*: El que transcurre desde el comienzo hasta obtener el resultado de la primera componente; depende de la latencia de la unidad funcional.
- *Velocidad de iniciación*: El que transcurre entre la obtención de cada una del resto de componentes después del tiempo de arranque. En computadores totalmente segmentados, como es el caso del DLXV, su valor es 1 ciclo.

Los tiempos de arranque de las unidades funcionales del DLXV se muestran en la siguiente tabla. Si dos operaciones consecutivas son dependientes se debe añadir al tiempo de arranque 4 ciclos.

Operación	**Arranque**
Suma vectorial	6
Multiplicación vectorial	7
División vectorial	20
Carga vectorial	12

Table 1. Tiempos de arranque

A continuación se muestra el código ensamblador para DLX:

```
.data
.align 2
x:        .double 1.2,1.4,2.2,2.4,3.2,3.4
y:        .double 2.1,4.1,2.2,4.2,2.3,4.3
a:        .double 2.7


.text
          .global main


main:
add r1,r0,x ;Guarda la dirección de comienzo de x en R1
add r2,r0,y ;Guarda la dirección de comienzo de y en R2
ld f0,a     ; Carga la variable a en el registro F0
addi r4,r1,40 ;Almacena en R4 la última posición del
              ; vector


inicio:
```

```
ld f2, 0(r1)  ; Carga el contenido de x[i] en F2
multd f4, f2, f0 ; Multiplica a*x[i] y lo guarda en F4
ld f6, 0(r2)  ; Carga y[j] en el registro F6
addd F6,F4,F6  ; Suma (a*x[i]) + y[j] y lo guarda en F6
sd 0(r2), F6  ; Almacena el contenido de F6 en y[j]
addi r1, r1, 8   ; Incrementa en 8 el índice i
addi r2,r2, 8  ; Incrementa en 8 el índice j
sgt r3, r1, r4   ; R1>R4? -> Guarda el resultado en R3
beqz r3, inicio ; Si no se cumple que R1>R4 salta a
                ; inicio
trap 0      ; Fin
```

El código ensamblador para DLXV es:

```
.data
rx:       .double 1.2,1.4,2.2,2.4,3.2,3.4
ry:       .double 2.1,4.1,2.2,4.2,2.3,4.3
a:        .double 2.7
.text
main:
ld f0,a
lv v1,rx
multsv v2,f0,v1
lv v3,ry
addv v4,v2,v3
sv ry,v4
```

Antes de pasar a la ejecución de los programas, se puede observar que para el código vectorial se necesita menos de la mitad de instrucciones que para la escalar 6 frente a 14), además en la escalar tendrá que afrontar la ejecución del programa a través de un bucle.

3.1 Resultados obtenidos en la ejecución del programa escalar

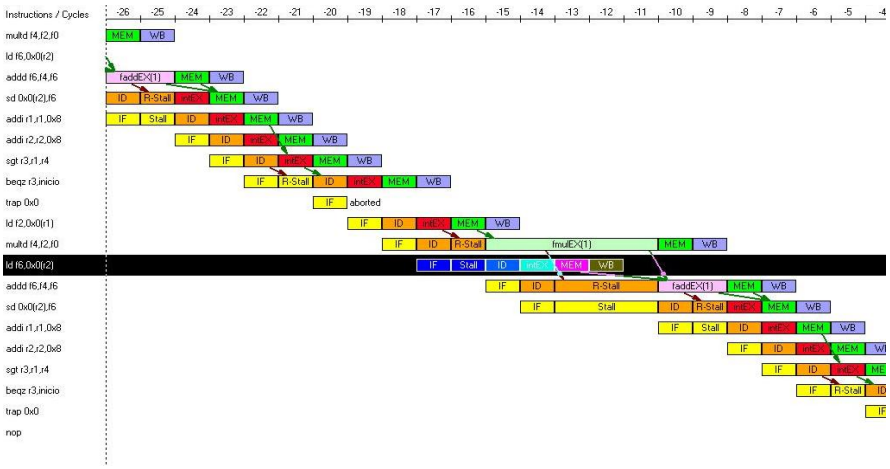

Fig. 4. Traza de ejecución para el programa escalar

Ciclos: 104
Stalls: 44

Tipo de Instrucciones	**Nº de instrucciones**	**%**
Carga/Almacenamiento	19 (13/6)	32,20
Aritmético/lógicas	7 por iteración	66,11
Otras	1	1,69
Total Instrucciones:	62	100

Table 2. Instrucciones ejecutadas para DLX

Tipo de Salto	**Nº de saltos**	**%**
Saltos Condicionales	6	10,17
Saltos Incondicionales	0	0,00
Total Saltos:	6	10,17

Table 3. Saltos para DLX

Tipo de detenciones	Nº de detenciones	%
RAW/WAR	36	34,62
WAW	0	0,00
Riesgos estructurales	0	0,00
Riesgos de control	5	4,81
Trap stalls	3	2,88

Table 4. Detenciones para DLX

3.2 Resultados obtenidos en la ejecución del programa vectorial[3]

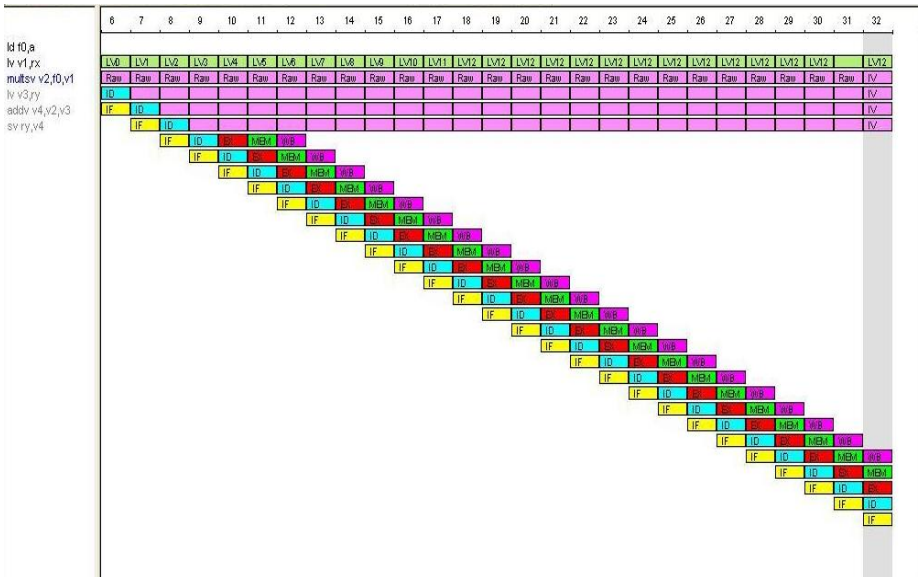

Fig. 5. Traza de ejecución de programa vectorial

[3] **Nota:** Para la realización en DLXV se ha activado el adelanto de resultados y la predicción de no tomar el tratamiento de los saltos.

Ciclos: 31
Instrucciones: 21
CPI: 1,476

Unidades Funcionales Escalares		
Unidad funcional	Segmentación	Latencia
Suma/Resta FP	Si	4
Multiplicación FP	Si	7
División FP	No	25

Table 5. Unidades funcionales escalares

Unidades Funcionales Vectoriales		
Unidad functional	**Segmentación**	**Latencia**
Suma/Resta FP	Si	7
Multiplicación FP	Si	8
División FP	No	21
Carga/Almacenamiento Vectorial	Si	13

Table 6. Unidades funcionales vectoriales

3.3 Conclusiones

1. La máquina vectorial (DLXV) necesita 31 ciclos para realizar la misma operación que DLX, que necesita 104 ciclos, por lo que el rendimiento de DLXV es claramente superior.
2. La frecuencia de interbloqueos de las etapas: En DLX cada ADDD debe esperar por un MULTD, y cada SD debe esperar por un ADDD. En DLXV, cada instrucción vectorial opera sobre todo los elementos del vector independientemente; sólo se requiere una detención por operación vectorial. Hablando en cifras 44 detenciones en total para DLX, algo más de un 40% del número total de instrucciones ejecutadas.
3. Además, se observa que para este problema concreto DLX necesita realizar unas instrucciones de salto, en concreto 6, las cuales afectan al rendimiento pero no son imprescindibles, es decir, se podría repetir el código las veces que fueran necesarias sin tener que ejecutar dichas instrucciones. Estas 6 instrucciones consumen un 10% de la ejecución total, lo cual es significativo.

4-Blibliografía

http://www.infor.uva.es/~bastida/Arquitecturas%20Avanzadas/Vectoriales.pdf
-Introducción general, definiciones básicas, características, tiempo de arranque...
-http://atc2.aut.uah.es/~acebron/cap3vect.pdf
-Nociones generales, juego de instrucciones, arquitectura DLXV
-http://www.angelfire.com/ca6/angie/vectoriales.htm
-Información muy general
-Artículo "A Comparison of Implicit and Explicit Parallel Programming" de Vincent W. Freeh (Universidad de Arizona).
-Ejemplos de comparación de programación paralela explicita e implicita.
-Práctica 4 "Procesadores Vectoriales" Universidad de Alcalá.
-Familiarización con arquitectura DLXV
-Tema 4 Computadores Vectoriales DIESA.Arquitectura y Tecnología de computadores.
-Multi-core Programming: Implicit Parallelism. Transparencias de Tuukka Haapasalo
-Paralelismo explícito e implicito; definiciones, ventajas y desventajas...