Arquitectura de Computadores

Práctica 2:
Implementación y optimización de un algoritmo en ensamblador DLX

Autores:
Fiz Rey Armesto
Oliver Suárez Velázquez

Primero hemos empezado realizando la versión sin optimizar en la cual hemos seguido los siguiente pasos:

- Hemos cargado las variables a1 y a 2 en los registros de punto flotante simple f0 y f1 respectivamente, a continuación hemos hecho la división, comprobando previamente que el divisor no sea 0, y la multiplicación para obtener los otros 2 valores de la matriz A.
- Para la matriz b hemos cargado las variables a3 y a 4 en los registros f4 y f5 respectivamente y hemos seguido los mismos pasos que en la matriz anterior.
- Una vez obtenidas las matrices A y B hemos empezado a hacer las multiplicaciones por orden para obtener el producto de Kronecker de ambas matrices.
- Después hacemos la suma de a1+a4, el cálculo de la matriz MF(a2,a3) y su respectivo determinante, todo en ese mismo orden.
- Una vez obtenidos estos últimos datos hacemos la división de la suma y el determinante comprobando que el divisor sea distinto de 0.
- Lo siguiente que hemos hecho ha sido multiplicar la matriz resultante del producto de Kronecker por lo obtenido en la última división. Vamos calculando cada componente de la matriz y guardandolo en memoria.
- Para hacer el guardado en memoria guardamos en el registro r1 la suma inmediata de r0 + 60(16 posiciones *4 bytes= 64, contando que como se guardan al revés porque vamos restando cada posición y la última no se avanza solo habría que contar 60 bytes, 4 por cada dato). Después de cada multiplicación guardamos el dato del registro donde se haya guardado en memoria y restamos 4 posiciones a r1.

```
multf   f0,f8,f30
sf      M(r1), f0
subi    r1,r1,#4
```

- Una vez tenemos toda esa matriz ya en memoria empezamos con los vectores VM y HM que para ello hacemos el mismo procedimiento de antes para cada vector, pero contando 12 bytes ya que serian 4 posiciones * 4 bytes=16 bytes y restamos 4 bytes por lo mencionado anteriormente y se quedarían en 12 bytes.
- Con los dos vectores ya calculados y guardados en memoria procedemos a hacer la suma de todos los valores de los dos vectores para obtener el valor check, y lo guardamos en memoria también.

Para optimizar el programa hemos realizado los siguientes cambios:

- Como detalle principal tenemos que ver que las operaciones que más tiempo tardan y por las que más ciclos se pierden son las divisiones, por lo que las desplazamos por el código de manera que siempre esté una en ejecución sin que esté una esperando a que acabe la anterior.
- Después de las divisiones las que más numerosas y por las que más tiempo se pierde en stalls que no son recuperables son las multiplicaciones.
- Al principio del programa hemos adelantado una de las cargas de memoria para posponer la multiplicación y nada más esta acabe empiece ya la siguiente a calcular.
- Los cálculos que se pueden hacer sin dependencias de las primeras divisiones son:
    - a1*a2
    - a3*a4
    - a2*a3
    - a1+a4

Por lo que estos cálculos los hemos adelantado para que se hagan de manera que no consuman más ciclos y usen partes del procesador que no se estén utilizando en ese justo momento.

- Las multiplicaciones de Kronecker que no tengan un valor α12 (la división de la matriz) también se pueden adelantar.
- A la hora de hacer el cálculo de la matriz M como lo siguiente a cada multiplicación sería guardar el resultado en memoria, lo que hemos hecho es hacer la primera multiplicación antes del cálculo para guardar en memoria el vector y así mover todas las multiplicaciones un paso antes y que cuando se vaya a guardar el dato ya no tenga que esperar porque este ya ha sido calculado previamente.
- Por último a la hora de calcular los vectores VM y HM como lo que pide el valor check es una suma de sus componentes lo que vamos haciendo es según se calculan los componentes, como estos son multiplicaciones se va sumando entre medias en el registro donde tenemos el valor check.

Sin optimizar

ESTADÍSTICAS	
Total	
Nº de ciclos:	353
Nº de instrucciones ejecutadas (IDs):	121
Stalls	
Raw stalls:	155
LD stalls:	2
Branch/Jump stalls:	0
Floating point stalls:	153
WAW stalls:	0
Structural stalls:	71
Control stalls:	0
Trap stalls:	4
Total	230
Conditional Branches	
Total:	4
Tomados:	0
No tomados:	4
Instrucciones Load/Store	
Total:	30
Loads:	5
Stores:	25
Instrucciones de punto flotante	
Total:	58
Sumas:	9
Multiplicaciones:	45
Divisiones:	4
Traps	
Traps:	1

Optimizado

ESTADÍSTICAS	
Total	
Nº de ciclos:	239
Nº de instrucciones ejecutadas (IDs):	121
Stalls	
Raw stalls:	6
LD stalls:	1
Branch/Jump stalls:	0
Floating point stalls:	5
WAW stalls:	0
Structural stalls:	101
Control stalls:	0
Trap stalls:	4
Total	111
Conditional Branches	
Total:	4
Tomados:	0
No tomados:	4
Instrucciones Load/Store	
Total:	30
Loads:	5
Stores:	25
Instrucciones de punto flotante	
Total:	58
Sumas:	9
Multiplicaciones:	45
Divisiones:	4
Traps	
Traps:	1