Optimización de cálculo vectorial en emulador de CPU con arquitectura RISC DLX

Arquitectura de computadores

Grado en Ingeniería Informática

Universidad de Salamanca

Autores:

Luis Blázquez Miñambres
Samuel Gómez Sánchez

Optimización de cálculo vectorial en emulador de CPU con arquitectura RISC DLX

Presentación del problema

En esta práctica se propone realizar la siguiente operación mediante el procesador DLX utilizando el emulador WinDLX

$$M=(A\times B)\cdot\frac{1}{|A+B|}+C\cdot\alpha$$

con $A, B, C, M\in M_{3x3}(\mathbb{R})$ y $\alpha\in\mathbb{R}$.

Dado que el problema es muy concreto y lo que se busca es optimizar el cálculo, el problema se ha resuelto de modo mejorado únicamente para matrices de dimensión tres y coeficientes reales, como explicaremos a continuación. Cualquier otra combinación es subóptima con el algoritmo planteado.

Versión sin optimizar

La versión sin optimizar utiliza un algoritmo «natural». Señalamos lo siguiente:

- Se han reservado los registros F18 – F26 para almacenar el resultado temporalmente
- Se ha mantenido el registro F31 a 0 para borrar otros registros
- El producto de matrices se ha calculado a partir de la definición.
- Se ha utilizado el método de Laplace para el cálculo del determinante.

Bajo estas condiciones el algoritmo resultante es:

```
Cargar la matriz A en los registros F0 – F8
Cargar la matriz B en los registros F9 – F17
Almacenar el valor α en el registro F30
Calcular X = A x B
    Para cada fila Ai = (ai1, ai2, ai3)
        Para cada columna Bj = (b1j, b2j, b3j)
            Calcular y almacenar xij = ai1·b1j + ai2·b2j + ai3·b3j

Calcular Y = A + B
    Para cada (i, j)
        Calcular yij = aij + bij
Calcular |A+B| por el método de Laplace
```

Para k_i en (y_{11}, y_{12}, y_{13})

Calcular y almacenar $k_i \cdot Adj(k_i)$

Almacenar $|A+B| = k_1 - k_2 + k_3$

Calcular $\beta = 1/|A + B|$

Calcular $Z = 1/|A + B| \cdot (A \times B)$

Para cada (i, j)

Calcular $z_{ij} = \beta \cdot x_{ij}$

Calcular $W = C \cdot \alpha$

Para cada (i, j)

Calcular $w_{ij} = \alpha \cdot c_{ij}$

Calcular y almacenar $M = Z + W$

Versión optimizada

Para la versión optimizada se ha analizado la complejidad computacional de la operación como paso previo a la implementación del algoritmo. Para ello, se ha realizado un cálculo ponderado según el número de ciclos de ejecución necesarios por cada operación. En la configuración indicada en el enunciado, tenemos que

- Suma/resta de números en coma flotante: 2 ciclos de ejecución
- Multiplicación de números en coma flotante: 5 ciclos
- División de números en coma flotante: 19 ciclos

Pre-análisis

Una valoración del problema nos arroja lo siguiente:

- Dado que en general los valores de las matrices no se repetirán, la suma de matrices no es optimizable, y para cualesquiera matrices $X, Y \in M_{mxn}(\mathbb{R})$ tenemos que $X+Y \in O(m \cdot n)$. Si m = n, $X+Y \in O(n^2)$.
- De igual modo y por un razonamiento análogo, los productos de matriz por escalar tampoco son optimizables, y para cualesquiera $X \in M_{mxn}(\mathbb{R}), k \in \mathbb{R}$ tenemos que $k \cdot X \in O(m \cdot n)$. Si m = n, $k \cdot X \in O(n^2)$.
- La división es una operación disponible en el hardware, y por tanto tampoco optimizable.
- El cálculo del determinante es, en el peor caso (por las fórmulas de Laplace o Sarrus, que son equivalentes), O(n!), y en los casos habituales, $O(n^3)$ – mediante eliminación Gaussiana –. No obstante, dado que las matrices son de 3x3,

$O(n!)=O(n\cdot(n-1)\cdot(n-2))=O(n^3)$. El cálculo mediante descomposición LU arroja también un coste de $O(n^3)$. Por ello, optamos por utilizar la aproximación más simple y calculamos el determinante por la regla de Sarrus.

- El cálculo del producto de matrices es $O(n^3)$ cuando se utiliza la definición del producto. Existen algoritmos como el de Strassen que permiten reducir la complejidad por debajo de la cúbica, aunque al precio de realizar un mayor número de sumas. Dado que los ciclos empleados en las multiplicaciones adicionales pueden ser utilizados para el cálculo de las diversas sumas que hay que realizar para el cálculo de la operación completa, optamos por la opción natural y utilizamos el algoritmo habitual para el cálculo del producto.
- Por último, se valoró la posibilidad de que el cálculo del determinante y el del producto de matrices compartiera algún término, de modo que pudiera reciclarse. Un análisis simple de las expresiones de ambos cómputos hace patente que esto no es así, de modo que ambos casos se mantienen independientes.

* * *

Aplicando a estas decisiones las ponderaciones arriba indicadas tenemos:

- Cualquier suma de matrices de 3x3 requiere de 9 sumas, i.e., 9 sumas · 2 ciclos/suma = 18 ciclos.
- Cualquier producto de matriz de 3x3 por escalar requiere de 9 productos, i.e., 9 productos · 5 ciclos/producto = 45 ciclos.
- La división requiere de 19 ciclos.
- El cálculo del determinante de una matriz M de 3x3 resulta $|M| = m_{11}\cdot m_{22}\cdot m_{33} + m_{12}\cdot m_{23}\cdot m_{31} + m_{13}\cdot m_{21}\cdot m_{32} - m_{31}\cdot m_{22}\cdot m_{13} - m_{32}\cdot m_{23}\cdot m_{11} - m_{33}\cdot m_{21}\cdot m_{12}$. Es decir: 12 productos · 5 ciclos/producto + 5 sumas · 2 ciclos/suma = 70 ciclos.
- El cálculo de cada componente de la matriz resultado del producto de matrices requiere de tres productos y dos sumas. Así, la matriz completa requerirá: 9 componentes · (3 productos · 5 ciclos/producto + 2 sumas · 2 ciclos/suma)/componente = 171 ciclos.

Con lo que resulta que el coste de las operaciones es:

$$Producto\,vectorial(171\,ciclos) > Determinante(70\,ciclos) > Producto\,escalar(45\,ciclos) > División(19\,ciclos) > Suma(18\,ciclos)$$

Algoritmo

Con estos datos, tomamos las siguientes decisiones para el cómputo de la operación:

- Se comienza cargando los datos necesarios para comenzar el producto vectorial lo antes posible. Almacenaremos AxB en F0 – F8.
- Comenzamos por calcular los productos necesarios para el cómputo del producto vectorial completo, y en los tiempo de ocupación de la ALU de multiplicación, vamos calculando las sumas necesarias para el cálculo del determinante, esto es, las componentes de A + B.
- Además, dado que aún se pueden aprovechar ciclos, vamos cargando dos elementos entre cada par de multiplicaciones, y además almacenamos el resultado de las sumas para liberar registros. Esto nos permite utilizar únicamente el registro F31 para el cálculo de las componentes de A + B.
- Cuando el cálculo de A + B está terminado, comenzamos a calcular algunas sumas del producto de matrices en los espacios que antes utilizábamos para el cálculo de las componentes de esa matriz.
- Una vez se termina el cálculo de AxB, todas las operaciones posibles son productos de números reales – salvo por la división –, de modo que tenemos dependencias estructurales inevitables. Se aprovechan los ciclos necesarios para cargar el valor α y la matriz C para comenzar el producto α·C tan rápido como se termine de calcular |A + B|.
- Al tiempo, una vez se termina de calcular el determinante, se comienza a calcular la división 1/|A+B|.
- Por último, se aprovecha para ir calculando el producto 1/|A+B|·AxB al mismo tiempo que el resultado del mismo se va sumando con la matriz ya calculada α·C. Durante los ciclos de productos y sumas se va almacenando al mismo tiempo el resultado definitivo en la matriz M.

Comparativa

El resultado es una mejora sensible de la eficiencia del algoritmo (cf. *Imagen 1* e *Imagen 2*):

Ciclos

- Se reduce el número de ciclos de 384 a 301.

Esperas

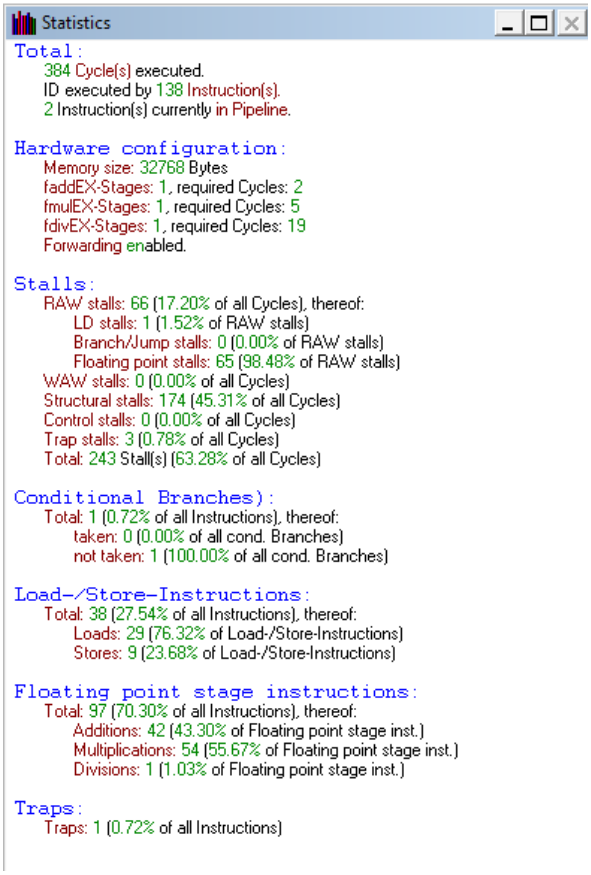

Imagen 1: estadísticas del algoritmo sin optimizar

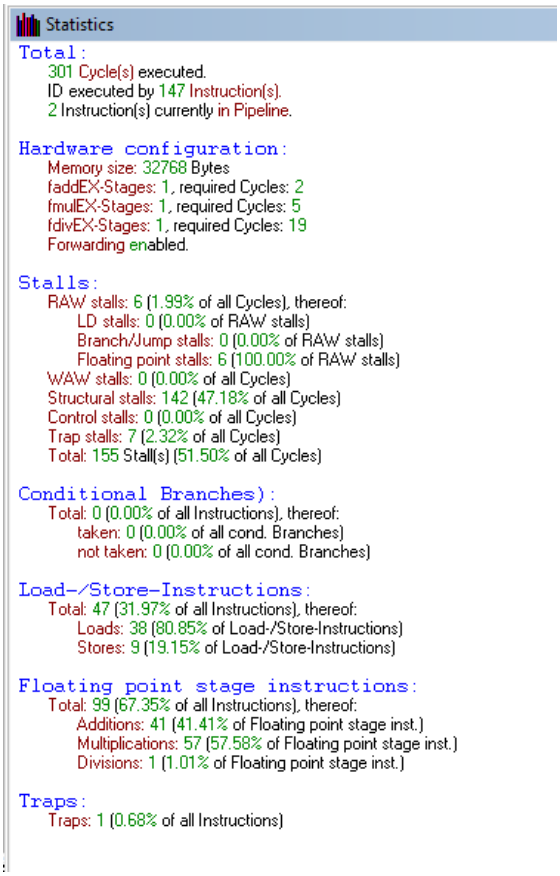

Imagen 2: estadísticas del algoritmo optimizado

- El número de dependencias de datos (RAW) se reduce de 66 – lo que suponía casi el 20% de los ciclos – a 6. Además, todas son dependencias en las ALU de coma flotante, y no perdemos tiempo en operaciones poco costosas como las de carga y almacenamiento (a diferencia de en la primera versión que pierde un ciclo esperando para cargar un dato). Por último cabe añadir que las dependencias RAW se dan en la versión optimizada en los últimos momentos del algoritmo, cuando ya no se puede sino esperar a que terminen las últimas operaciones para almacenar los datos. Por ello, son omisibles, y podemos afirmar que *eliminamos las dependencias de datos*.
- El número de dependencias estructurales se reduce de 174 a 142.
- El número total de ciclos de espera se reduce de 243, lo que supone un 63.28% de los ciclos, a 155, que constituyen un 51.50% del total.

Saltos condicionales

- Eliminamos el salto condicional presente en la primera versión del algoritmo.

Cargas y almacenamientos

- Aumenta el total de operaciones de carga y almacenamiento. Esto se debe fundamentalmente a la necesidad de, primero, descargar el resultado de A+B para permitir reutilizar registros, para después volver a cargarla para el cálculo del determinante. Es posible que una ordenamiento diferente de las operaciones permitiera mejorar este comportamiento, pero no se ha encontrado en el análisis realizado para la práctica.

Operaciones en coma flotante

- El número de productos aumenta. Esto se debe probablemente a que el algoritmo sigue siendo subóptimo. Un análisis a posteriori revela que el cálculo del determinante mediante la fórmula de Sarrus desaprovecha factores comunes, aumentando respecto al uso de la expansión de Laplace el número de productos de 9 a 12 (esto se comprueba fácilmente observando las expresiones en ambos casos). Precisamente son tres los productos adicionales en el algoritmo optimizado. Es posible, por tanto, que el algoritmo propuesto sea mejorable modificando el cálculo del determinante para emplear la expansión por menores de Laplace; de este modo reduciríamos la operación de cálculo del determinante a 9 productos · 5 ciclos/producto + 5 sumas · 2 ciclos/suma = 55 ciclos (vs. 70 ciclos necesarios en el caso anterior).

$$|M| = \begin{vmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{vmatrix} = m_{11} \cdot \begin{vmatrix} m_{22} & m_{23} \\ m_{32} & m_{33} \end{vmatrix} - m_{12} \cdot \begin{vmatrix} m_{21} & m_{23} \\ m_{31} & m_{33} \end{vmatrix} + m_{13} \cdot \begin{vmatrix} m_{21} & m_{22} \\ m_{31} & m_{32} \end{vmatrix} =$$

$$= m_{11} \cdot (m_{22} \cdot m_{33} - m_{32} \cdot m_{23}) + m_{12} \cdot (m_{21} \cdot m_{33} - m_{31} \cdot m_{23}) + m_{13} \cdot (m_{21} \cdot m_{32} - m_{31} \cdot m_{22})$$