PROGRAMACIÓN PARALELA Y COMPUTACIÓN DE ALTAS PRESTACIONES

TRABAJO AUTÓNOMO: ENTORNOS DE PROGRAMACIÓN PARALELA

PRÁCTICAS DE MPP 17/18

José Luis Cánovas Sánchez

9 OCTUBRE

Algoritmos Matriciales Básicos

CUESTIÓN 1

Comparar prestaciones diferentes multiplicaciones matriciales.

Tamaños de valores entre 10 y 20.

Tabla de tiempos (seg):

Algoritmo	N=500	N=1000	N=2000	N=3000	N=4000	R=1000 C=1500	R=2000 C=1500
matriz_matriz	0.112411	3.016618	54.451037	228.945907	597.308980	8.847286	38.525340
matriz_matriz_ld	0.090204	0.957873	15.497439	47.372445	156.147463	2.665864	10.526499
matriz_matriz_mv	0.089872	0.712091	5.840717	19.574129	46.351312	1.065627	4.231485
matriz_matriz_mv*	0.091063	0.765406	14.383261	29.547487	146.667864	1.283370	7.023438
matriz_matriz_pe	0.090355	0.968561	15.152740	47.041302	157.112019	2.834806	10.142969
matriz_matriz_tras	0.084790	0.687765	5.644784	19.127753	44.927649	1.070011	4.211631

La primera diferencia notable es que en matriz_matriz_ld, a pesar de tener que realizar una suma y multiplicación en cada acceso de un elemento de una matriz, a[i*lda+k], es más rápido que la versión matriz_matriz. Mirando el código, vemos que matriz_matriz realiza una reserva de memoria primero de un vector de punteros a las filas, y por cada fila, otra reserva de memoria para los elementos. Al acceder a un elemento de matriz_matriz, primero se accede a dos direcciones de memoria señaladas por los punteros, y por último el elemento, a[i][j], mientras que en matriz_matriz_ld, como es un único vector de tamaño RxC (N^2), sólo se accede a la dirección de un puntero y se obtiene el dato.

En el caso de matriz_matriz_mv había un error en el código, donde el leading dimension de las columnas de b estaba puesto a 1 cuando debería ser ldb. Los tiempos con ld a 1 son más propios de la versión traspuesta, como se puede comprobar, pues el código accede a la dirección de memoria siguiente inmediata, en vez de a la siguiente fila a unos ldb bytes de distancia.

La versión corregida, matriz_matriz_mv* tiene tiempos más cercanos a lo esperado, comparándose con matriz_matriz_ld y matriz_matriz_pe. La ventaja de la versión _mv* es que al separar la multiplicación en tres sub-funciones, minimiza los cálculos necesarios para obtener la dirección del siguiente elemento, es decir, la sub-función matriz por vector se ahorra calcular el offset de la columna de b, y la sub-función producto escalar además ahorra calcular el offset de la fila de a. El ahorro en tiempo es pequeño, no tan notable como el cambio de matriz_matriz, con arrays bidimensionales, a la versión con un solo array de tamaño n*m.

El código de matriz_matriz_tras obtiene los mejores resultados por optimizar el acceso a memoria al máximo. Primero guarda en bt la traspuesta de b, y luego realiza la multiplicación recorriendo las filas en a y b, guardando parte del cálculo de la dirección de memoria en da y db, equivalente al caso anterior al llamar a subfunciones.

CUESTIÓN 2

Programar una rutina de multiplicación para matrices dispersas.

Tiempos en segundos:

Dispersión	N=500	N=1000	N=2000	N=3000	N=4000
25% de ceros	0.543462	4.475287	36.083237	121.242076	286.920930
50% de ceros	0.479229	3.880770	31.269870	105.968002	250.231690
75% de ceros	0.244973	1.959482	16.035858	54.177447	127.935224

Teniendo en cuenta que para acceder al elemento de una celda debemos leer al menos 3 punteros (array de datos, fila y columna), es equiparable a la primera versión de matriz_matriz de matrices densas, donde había un puntero a las filas y otro para la columna, pero de tamaños N.

Comparando según el nivel de dispersión, cuando tenemos un 25% o 50% de ceros en la matriz, los tiempos obtenidos son relativamente próximos. Al comparar con un 75% de ceros, dividimos el tiempo del caso de 50% de dispersión a la mitad.

En el uso de memoria, para cualquier N, una matriz densa ocuparía, sin contar los punteros de la matriz, $N^2 \cdot 64$ bits (valores double), mientras que una matriz con un 75% de ceros, $¼ \cdot N^2 \cdot 64$ bits para los datos y $2 \cdot ¼ \cdot N^2 \cdot 32$ bits para los arrays de enteros con las filas y columnas, lo que nos deja $½ \cdot N^2 \cdot 64$ bits, la mitad de memoria que la matriz densa. Sin embargo, con un 50% de elementos, $½ \cdot N^2 \cdot 64$ bits para los datos y $2 \cdot ½ \cdot N^2 \cdot 32$ bits para los índices, en total $N^2 \cdot 64$ bits, la misma memoria utilizada que en el caso de una matriz densa. Y para una matriz con sólo un 25% de ceros, $¾ \cdot N^2 \cdot 64$ bits para los datos y $2 \cdot ¾ \cdot N^2 \cdot 32$ bits para los índices, en total $1.5 \cdot N^2 \cdot 64$ bits, un 50% más de memoria utilizada que con la versión densa.

Es de esperar que para los casos del 50% y 25% de ceros, al utilizar la misma o más memoria, con un acceso menos eficiente, obtengamos peores tiempos que en las versiones densas, tras ver el efecto de transponer la matriz o usar un array de NxN en vez de una matriz bidimensional. Sin contar con la primera versión de matriz_matriz, esto es exactamente lo que ocurre. Con un 50% o 25% de ceros, los tiempos son cerca del doble que las versiones densas (ld, mv, pe). Si comparamos con la primera

versión matriz_matriz, a partir de N=2000, las matrices dispersas aprovechan el realizar menos operaciones de coma flotante y que al transponer la matriz dispersa b, aprovechan mejor la localidad espacial que la versión densa bidimensional.

En ningún caso conseguimos superar en tiempo a la versión de multiplicar matrices con transpuesta. Aunque hemos visto que el uso en memoria con un 75% de ceros es más eficiente y puede ser un factor decisivo. A partir de N=2000, con un 75% de ceros los tiempos son muy próximos a los casos ld, mv y pe. Es notable que para N=4000 tarda ~30 segundos menos que esas versiones, pero sigue sin superar a la versión transpuesta.

Podemos concluir que a pesar de realizar menos operaciones en coma flotante, la representación en memoria de las matrices dispersas afecta a su rendimiento. Sólo a partir de dimensiones suficientemente grandes y matrices suficientemente dispersas, la reducción en operaciones de coma flotante es mayor que la desventaja del acceso a memoria.

Generación de matrices dispersas

A la hora de realizar las pruebas con dimensiones grandes, el mayor problema aparecía con la generación de matrices dispersas, donde el tiempo para multiplicarlas era despreciable frente al tiempo para generarlas, haciendo la sesión de pruebas muy lenta. En AlgMatBas/io.c incluyo el mismo fichero añadiendo la función generar_matriz_dispersa_fast() que, aunque pueda tener aun muchas posibles mejoras, se basa en el algoritmo de transposición rápida y genera las matrices dispersas más rápido que la versión previa. Abajo una tabla con la comparativa (en segundos) de generar la matriz a con la función original y la matriz b con la nueva versión. Las matrices generadas tienen un 50% de ceros:

	N=500	N=1000	N=2000	N=3000	N=4000
gen_m_sparse	6.681610	106.622945	-1909.460666*	¬	¬
gen_m_sp_fast	0.012239	0.069600	0.464713	1.447692	3.315785

* Overflow de (tf1-ti1)/1000000. Donde tf1 y ti1 son enteros de 32 bits multiplicados por 1000000, lo que deja de margen menos de una hora de medición, i.e, la generación de 2000000 elementos para una matriz de 2000x2000 llevó más de una hora.