1. Optimizaciones.

1.1. Inversión de ciclos.

El método consiste en invertir los ciclos exteriores del método básico. Al realizar la multiplicación A x B = C con el método básico, las matrices A y C se recorren por filas y la B por columnas. Al intercambiar el orden de los bucles, se cambia el sentido de recorrido de cada matriz, es decir, A y C por columnas y B por filas. Esto permite un mejor aprovechamiento de la localidad espacial que provee la dispoción de los elementos contiguos en la memoria bajo el criterio Column Major Order.

1.2. Básico con trasposición.

Como se explico en el m'etodo anterior, A y C se recorren originalmente por filas. Lo que hace este método es transponer A y luego efectuar la multiplicación recorriendo A y B por columnas simultáneamente. Como se utiliza un acumulador para cada elemento de C, se realizan menor cantidad de accesos a éstos últimos, con respecto a los elementos de las otras matrices. Al igual que en el caso anterior, el beneficio se obtiene debido al aprovechamiento de la localidad espacial.

Sin embargo, el desempeño de este método se ve impactado por la penalidad de calcular la matriz traspuesta, porque deben accederse simultáneamente los elementos simétricos respecto a la diagonal (que pertenecen a distintas filas y columnas de la matriz transponer).

1.3. Multiplicación por bloques

Es posible dividir la matriz en bloques de tamaño arbitrario y, utilizando el método básico, realizar multiplicaciones sucesivas de matrices mas pequeñas, mejorando la localidad temporal, ya que se mantienen en la caché los datos que se usarán a corto plazo.

1.4. Multiplicación por columnas.

Este método realiza la multiplicación de matrices en forma incremental, efectuando todas las operaciones posibles con los elementos disponibles de una de las matrices y sumando los resultados parciales en la matriz resultado.

Utilizando las propiedades de las matrices se consiguen las siguientes equivalencias que proporcionanan fundamento matemático al método. Por simplicidad, se utilizan matrices de 2x2.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

El producto del miembro izquierdo se puede escribir como

$$\left(\begin{bmatrix} a_{11} & 0 \\ a_{21} & 0 \end{bmatrix} + \begin{bmatrix} 0 & a_{12} \\ 0 & a_{22} \end{bmatrix} \right) \left(\begin{bmatrix} b_{11} & 0 \\ b_{21} & 0 \end{bmatrix} + \begin{bmatrix} 0 & b_{12} \\ 0 & b_{22} \end{bmatrix} \right)$$

Y expandiendo los paréntesis resulta

$$\begin{bmatrix} a_{11} & 0 \\ a_{21} & 0 \end{bmatrix}\begin{bmatrix} b_{11} & 0 \\ b_{21} & 0 \end{bmatrix}+\begin{bmatrix} 0 & a_{12} \\ 0 & a_{22} \end{bmatrix}\begin{bmatrix} b_{11} & 0 \\ b_{21} & 0 \end{bmatrix}+\begin{bmatrix} a_{11} & 0 \\ a_{21} & 0 \end{bmatrix}\begin{bmatrix} 0 & b_{12} \\ 0 & b_{22} \end{bmatrix}+\begin{bmatrix} 0 & a_{12} \\ 0 & a_{22} \end{bmatrix}\begin{bmatrix} 0 & b_{12} \\ 0 & b_{22} \end{bmatrix}$$

Así, la matriz producto se puede descomponer en la siguiente suma

$$\begin{bmatrix} a_{11}b_{11} & 0 \\ a_{21}b_{11} & 0 \end{bmatrix} + \begin{bmatrix} a_{12}b_{21} & 0 \\ a_{22}b_{21} & 0 \end{bmatrix} + \begin{bmatrix} 0 & a_{11}b_{12} \\ 0 & a_{21}b_{12} \end{bmatrix} + \begin{bmatrix} 0 & a_{12}b_{22} \\ 0 & a_{22}b_{22} \end{bmatrix}$$

Como se puede apreciar, las tres matrices se acceden por columnas. Esto resulta en una mejor utilización de la memoria cache.

Para que este método funcione correctamente, se requiere que la matriz donde se almacena el resultado haya sido inicializada en 0. Sin embargo, como esto no es tenido en cuenta en el calculo del tiempo (ya que se realiza para todos los metodos), no presenta una desventaja.

Otro factor importante es el grado de asociatividad de la cache, ya que para que no se produzca trashing tiene que tener como minimo 4 vías para evitar que se reemplacen entre si las columnas de las matrices A, B y C.

Por último, si el tamaño de la matriz a multiplicar es muy grande, las columnas de cada matriz no se pueden mapear completmente en la caché, y si el grado de asociatividad no es el suficiente como para que no se produzcan conflictos no solo entre las matrices, sino también entre las columnas que cada una posee, se puede generar trashing.

1.5. Multiplicación por columnas, usando operación en bloques

Este método es la unión de los 2 anteriores, combinando las ventajas de cada uno. Básicamente, con la operacion por bloques se logra reducir el problema que se produce en el método de multiplicación por columnas, al operar con matrices de gran tamaño. Si se escoje el valor adecuado para el tamaño de bloque, se puede lograr operar con matrices pequeñas cuyas columnas entran completamente en cada linea de la caché. De esta manera, se hace un gran aprovechamiento de la localidad espacial y temporal.

2. Hardware utilizado

El programa de multiplicación de matrices, con cada optimización implementada, se ejecutó en dos configuraciones de hardware que se describen a continuación.

2.1. Intel® Core™2 Duo Desktop Processor E6400

-

Spec Number: SL9T9
CPU Speed: 2.13 GHz
PCG: 06
Bus Speed: 1066 MHz

Cache Info:
1st-level data cache: 32 KB, 8-way set associative, 64-byte line size
Data TLB: 4 KB Pages, 4-way set associative, 256 entries
1st-level instruction cache: 32 KB, 8-way set associative, 64-byte line size
2nd-level cache: 2 MB, 8-way set associative, 64-byte line size
L0 Data TLB: 4 MB pages, 4-way set associative, 16 entries
L0 Data TLB: 4 MB pages, 4-way set associative, 16 entries
64-byte Prefetching
Data TLB: 4 MB Pages, 4-way set associative, 32 entries
Instruction TLB: 4 KB Pages, 4-way set associative, 128 entries
Instruction TLB: 4 MB Pages, 4-way set associative, 4 entries Instruction TLB: 2 MB Pages, 4-way set associative, 8 entries

3. Corridas

3.1. Core 2 Duo E6400

Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion basica de tres bucles.

Tam: 31 mflop/s: 956.216
Tam: 32 mflop/s: 944.568
Tam: 96 mflop/s: 1266.29
Tam: 97 mflop/s: 1244.16
Tam: 127 mflop/s: 1283.46
Tam: 128 mflop/s: 651.338
Tam: 129 mflop/s: 1277.71
Tam: 191 mflop/s: 1271.9
Tam: 192 mflop/s: 645.99
Tam: 229 mflop/s: 1275.01
Tam: 255 mflop/s: 1282.98
Tam: 256 mflop/s: 644.159
Tam: 257 mflop/s: 1279.6
Tam: 319 mflop/s: 1278.86
Tam: 320 mflop/s: 647.424
Tam: 321 mflop/s: 1266.74
Tam: 417 mflop/s: 444.455
Tam: 479 mflop/s: 370.454
Tam: 480 mflop/s: 372.397
Tam: 511 mflop/s: 353.249
Tam: 512 mflop/s: 143.71
Tam: 639 mflop/s: 339.37
Tam: 640 mflop/s: 356.774
Tam: 767 mflop/s: 337.243
Tam: 768 mflop/s: 166.853
Tam: 769 mflop/s: 331.102

Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion por 3 bucles invertido.

Tam: 31 mflop/s: 1321.45
Tam: 32 mflop/s: 1282.66
Tam: 96 mflop/s: 1140.83
Tam: 97 mflop/s: 1205.84
Tam: 127 mflop/s: 1270.47
Tam: 128 mflop/s: 745.792
Tam: 129 mflop/s: 1227.73
Tam: 191 mflop/s: 1310.07
Tam: 192 mflop/s: 742.715
Tam: 229 mflop/s: 1295.13
Tam: 255 mflop/s: 1273.96
Tam: 256 mflop/s: 753.4
Tam: 257 mflop/s: 1330.58
Tam: 319 mflop/s: 1202.46
Tam: 320 mflop/s: 713.453
Tam: 321 mflop/s: 1005.99
Tam: 417 mflop/s: 309.13
Tam: 479 mflop/s: 289.258
Tam: 480 mflop/s: 333.348
Tam: 511 mflop/s: 244.422
Tam: 512 mflop/s: 149.318
Tam: 639 mflop/s: 200.872
Tam: 640 mflop/s: 272.988
Tam: 767 mflop/s: 195.256
Tam: 768 mflop/s: 180.878
Tam: 769 mflop/s: 192.686

Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion de matrices simple por bloques.

Tam: 31 mflop/s: 1212.9
Tam: 32 mflop/s: 1208.78
Tam: 96 mflop/s: 1231.96
Tam: 97 mflop/s: 1176.52
Tam: 127 mflop/s: 1276.96
Tam: 128 mflop/s: 716.413
Tam: 129 mflop/s: 1288.7
Tam: 191 mflop/s: 1270.03
Tam: 192 mflop/s: 1277.75
Tam: 229 mflop/s: 1258.1
Tam: 255 mflop/s: 1224.33
Tam: 256 mflop/s: 625.331
Tam: 257 mflop/s: 1190.77
Tam: 319 mflop/s: 1189.44
Tam: 320 mflop/s: 1200.76
Tam: 321 mflop/s: 1255.07
Tam: 417 mflop/s: 1154.13
Tam: 479 mflop/s: 1251.7
Tam: 480 mflop/s: 1245.69
Tam: 511 mflop/s: 1238.65
Tam: 512 mflop/s: 632.629
Tam: 639 mflop/s: 1197.49
Tam: 640 mflop/s: 654.848
Tam: 767 mflop/s: 1241.31
Tam: 768 mflop/s: 621.655
Tam: 769 mflop/s: 1201.25

Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion por columnas de A.

Tam: 31 mflop/s: 1627.4
Tam: 32 mflop/s: 1580
Tam: 96 mflop/s: 1420.21
Tam: 97 mflop/s: 1416.65
Tam: 127 mflop/s: 1474.29
Tam: 128 mflop/s: 1386.12
Tam: 129 mflop/s: 1491.18
Tam: 191 mflop/s: 1457.47
Tam: 192 mflop/s: 1381.82
Tam: 229 mflop/s: 1547.82
Tam: 255 mflop/s: 1527.84
Tam: 256 mflop/s: 1388.78
Tam: 257 mflop/s: 1536.32
Tam: 319 mflop/s: 1525.52
Tam: 320 mflop/s: 1542.17
Tam: 321 mflop/s: 1488.85
Tam: 417 mflop/s: 1355.6
Tam: 479 mflop/s: 1195.36
Tam: 480 mflop/s: 1200.6
Tam: 511 mflop/s: 1122.46
Tam: 512 mflop/s: 1033.02
Tam: 639 mflop/s: 971.96
Tam: 640 mflop/s: 873.935
Tam: 767 mflop/s: 891.468
Tam: 768 mflop/s: 930.549
Tam: 769 mflop/s: 910.804

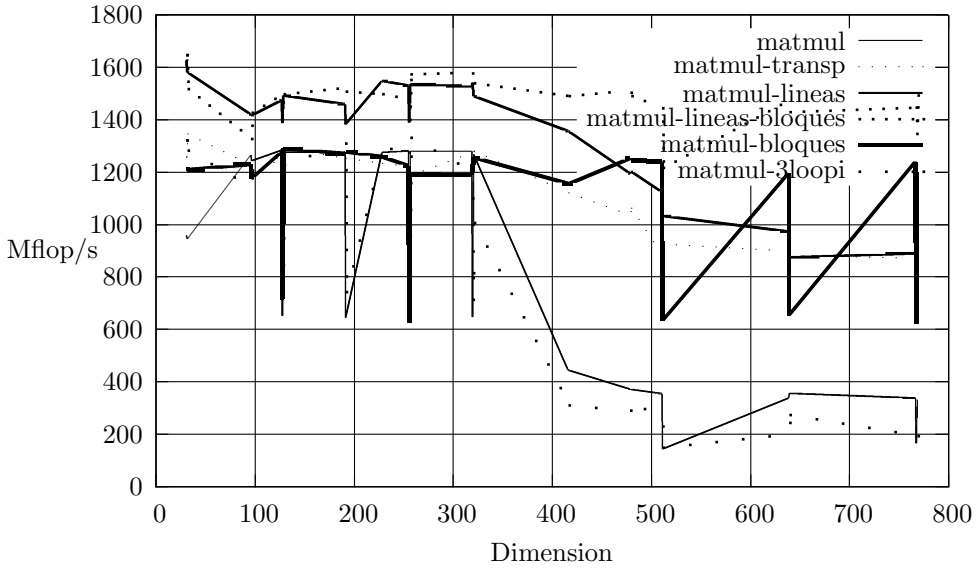
Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion basica de tres bucles luego de trasponer.

Tam: 31 mflop/s: 1258.3
Tam: 32 mflop/s: 1346.76
Tam: 96 mflop/s: 1226.76
Tam: 97 mflop/s: 1180.43
Tam: 127 mflop/s: 1258.67
Tam: 128 mflop/s: 1250.07
Tam: 129 mflop/s: 1291.28
Tam: 191 mflop/s: 1252.98
Tam: 192 mflop/s: 1253.05
Tam: 229 mflop/s: 1232.31
Tam: 255 mflop/s: 1221.02
Tam: 256 mflop/s: 1199.36
Tam: 257 mflop/s: 1185.93
Tam: 319 mflop/s: 1273.79
Tam: 320 mflop/s: 1278.29
Tam: 321 mflop/s: 1279.12
Tam: 417 mflop/s: 1122.18
Tam: 479 mflop/s: 1036.39
Tam: 480 mflop/s: 1063.2
Tam: 511 mflop/s: 882.452
Tam: 512 mflop/s: 926.115
Tam: 639 mflop/s: 899.544
Tam: 640 mflop/s: 882.849
Tam: 767 mflop/s: 873.074
Tam: 768 mflop/s: 893.43
Tam: 769 mflop/s: 869.49

Compilador: gcc
Opciones: -O3 -mcpu=pentium3
Plataforma: i686-pc-linux-gnu
Descripcion: Multiplicacion de matrices por linea por bloques.

Tam: 31 mflop/s: 1646.9
Tam: 32 mflop/s: 1517.89
Tam: 96 mflop/s: 1326.21
Tam: 97 mflop/s: 1425.29
Tam: 127 mflop/s: 1495.96
Tam: 128 mflop/s: 1463.36
Tam: 129 mflop/s: 1496.36
Tam: 191 mflop/s: 1522.35
Tam: 192 mflop/s: 1506.85
Tam: 229 mflop/s: 1496.02
Tam: 255 mflop/s: 1477.82
Tam: 256 mflop/s: 1398.82
Tam: 257 mflop/s: 1572.1
Tam: 319 mflop/s: 1579.07
Tam: 320 mflop/s: 1532.77
Tam: 321 mflop/s: 1538.13
Tam: 417 mflop/s: 1489.86
Tam: 479 mflop/s: 1512.6
Tam: 480 mflop/s: 1490.32
Tam: 511 mflop/s: 1443.57
Tam: 512 mflop/s: 1215.07
Tam: 639 mflop/s: 1474.68
Tam: 640 mflop/s: 1428.28
Tam: 767 mflop/s: 1445.49
Tam: 768 mflop/s: 1345.2
Tam: 769 mflop/s: 1491.86

3.2. Grafico Comparativo:



4. Analisis de resultados y conclusiones:

4.1. Inversión de ciclos.

Como se observa en los gráficos, el multiplicar básico es muy inestable respecto del tamaño de matriz utilizado y el más ineficiente. Asimismo, lo único que produce la mejora propuesta es un **pequeño** aumento de la estabilidad y del rendimiento. La curva mantiene la misma tendencia que la del método básico.

4.2. Básico con trasposición.

En este caso podemos notar que el método es bastante estable (en contraposición con los anteriores).

Cuando la matriz es pequeña, el costo de transposición es pequeño, y la ventaja de recorrer por columnas no se ve tan perjudicada, obteniendo un buen desempeño.

Cuando la matrices a multiplicar son muy grandes, este método presenta 2 problemas: Por un lado el costo de transposición aumenta, porque los elementos simétricos están más separados en memoria y se pierde localidad espacial, y por el otro lado, cada columna columna sobrepasa el tamaño de la caché y se genera trashing. Este fenómeno se puede apreciar en el gráfico (con la brusca caída de mflops alrededor de un tamaño identificable).

4.3. Multiplicación por columnas.

Se puede ver que este método se comporta de manera similar al anterior (porque se basan en el mismo principio) sin el tiempo consumido por el proceso de transposición, por lo que la curva mflops(tamaño) se encuentra desplazada hacia arriba.

4.4. Multiplicación por bloques.

Este método evidencia la ventaja de trabajar con matrices de un tamaño deseado (pudiendo elegir el mas conveniente para cada situación).

El desempeño es muy variable y depende de la compatibilidad entre las dimensiones de las matrices y el tamaño del bloque (sin dejar de lado el tamaño de la cache y el grado de asociatividad de la misma). Esto significa que se puede lograr un gran rendimiento al procesar matrices muy grandes, eligiendo correctamente el bloque, cosa que no se puede realizar con los otros métodos.

Todo esto esto se puede ver claramente en el gráfico.

4.5. Multiplicación por columnas y bloques.

Como supusimos, se logran aprovechar las ventajas de los dos métodos anteriores.

Se pudo elegir el tamaño de bloque de manera tal de operar la mayor parte del tiempo con matrices que povocan un rendimiento óptimo para el método de multiplicación por columnas. De esta manera se logra mantener este nivel sin importar el tamaño de la matriz original.

5. Código fuente