

UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO FACULTAD DE INGENIERÍA



Sistemas Distribuidos Examen 1: SAXPY

NOMBRE DEL ALUMNO:

Salas Diaz Miguel Angel



FECHA DE ENTREGA: 11/Septiembre/2023

NOMBRE DEL PROFESOR: Ing. José Antonio Ayala Barbosa

GRUPO: 01

SEMESTRE: 2024-1

Objetivos:

- Entender la forma en que se almacenan los arreglos en memoria.
- Comprender técnica de optimización de caché.

Conceptos teóricos

- SAXPY “Single-Precision A X Plus Y”

Es una función en la biblioteca estándar de Basic Linear Algebra Subroutines (BLAS).

Dicha operación es realizada en un tiempo de reloj. SAXPY es una combinación de una multiplicación escalar y una suma de vectores, y es muy simple:

Toma como entrada dos vectores, comúnmente float (32 bits), $x[]$ y $y[]$ con n elementos cada uno, y un valor escalar A . Multiplica cada elemento $x[i]$ por a y agrega el resultado en $y[i]$. Una implementación simple en C se muestra en la figura 1.

```
1
2 void saxpy(int n, float a, float * x, float * y)
3 {
4     for (int i = 0; i < n; ++i)
5         y[i] = a * x[i] + y[i];
6 }
```

Figura 1

El ejemplo más común de la utilización de esta función se encuentra en la multiplicación estándar de matrices, donde es necesario la multiplicación del renglón de la primera por la columna de la segunda, más el resultado almacenado en la matriz resultado.

```
1 for (i = 0; i < n; ++i)
2 {
3     for (j = 0; j < n; ++j)
4     {
5         for (k = 0; k < n; ++k)
6         {
7             matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
8         }
9     }
10 }
```

Donde i es la variable que itera los renglones de la matriz A . j es la variable que itera las columnas de la matriz B . k la variable temporal que itera elemento a elemento, de ya sean, los renglones o las columnas.

- **Orden de almacenamiento**

Los arreglos multidimensionales (matrices) son omnipresentes en el cómputo científico. El acceso a datos es un tema crucial en este campo, ya que el mapeo entre elementos debe coincidir con el orden en que el código carga y almacena datos en la computadora para que la localidad en tiempo y espacio pueda ser empleada.

El acceso estricto (Strided access) a una matriz unidimensional reduce el espacio en memoria, lo que lleva a una baja utilización del ancho de banda disponible y así, finalmente, a la optimización de recursos. Dependiendo del lenguaje de programación se utiliza un orden de recorrido de los vectores en el caché. Esquema de almacenamiento de matriz de orden-fila (Row major order), utilizado por el lenguaje de programación C. Las filas de la matriz se almacenan consecutivamente en la memoria.

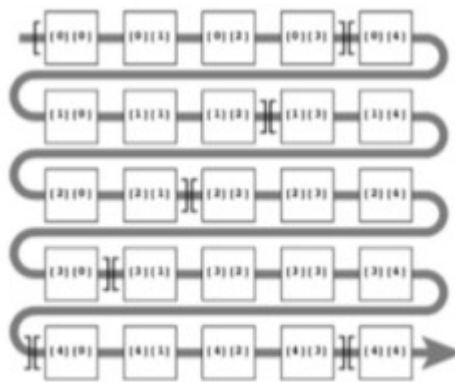


Figura 2

En cambio, en el lenguaje de programación Fortran (figura 3), se rige por el esquema de almacenamiento de matriz de orden-columna (Column major order), donde las líneas de caché almacenan las columnas consecutivamente.

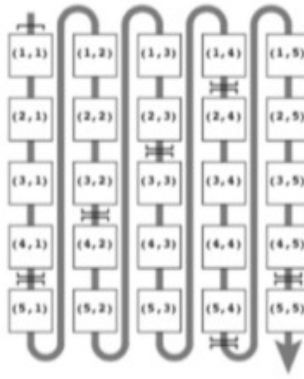


Figura 3

Desarrollo:

- 1- Crear un programa en C que multiplique 2 matrices cuadradas y el resultado lo almacene en una tercera matriz.
- 2- Cronometrar el tiempo que tarda cada una de las combinaciones en resolverse.
- 3- Variar el orden de los bucles for en sus 6 combinaciones para cambiar la forma de acceso a la información de la estructura de datos
- 4- Ejecutar al menos 3 diferentes sistemas de matrices (100*100) (500*500) (1000*1000) (5000*5000) (10 000*10 000)

Se analizaron las formas de llenado: ijk, ikj, jki, jik, kij, kji
Y se obtuvieron los siguientes resultados:

Primer caso: matriz de (100*100)

```

D:\Descargas\practica01.exe
===== { Practica01 } =====
Llenado de Matrices: 0.000000 s
Tiempo metodo ijk: 0.005001 s
Tiempo metodo ikj: 0.004000 s
Tiempo metodo kij: 0.006000 s
Tiempo metodo jki: 0.004999 s
Tiempo metodo kji: 0.005001 s
Tiempo metodo jik: 0.006001 s

-----
Process exited after 0.743 seconds with return value 0
Presione una tecla para continuar . . .
  
```

Segundo caso: matriz de (500*500)

```
D:\Descargas\practica01.exe

===== { Practica01 } =====
Llenado de Matrices: 0.010000 s
Tiempo metodo ijk: 0.644073 s
Tiempo metodo ikj: 0.631937 s
Tiempo metodo kij: 0.950386 s
Tiempo metodo jki: 0.848041 s
Tiempo metodo kji: 0.789057 s
Tiempo metodo jik: 0.634046 s

-----
Process exited after 5.451 seconds with return value 0
Presione una tecla para continuar . . .
```

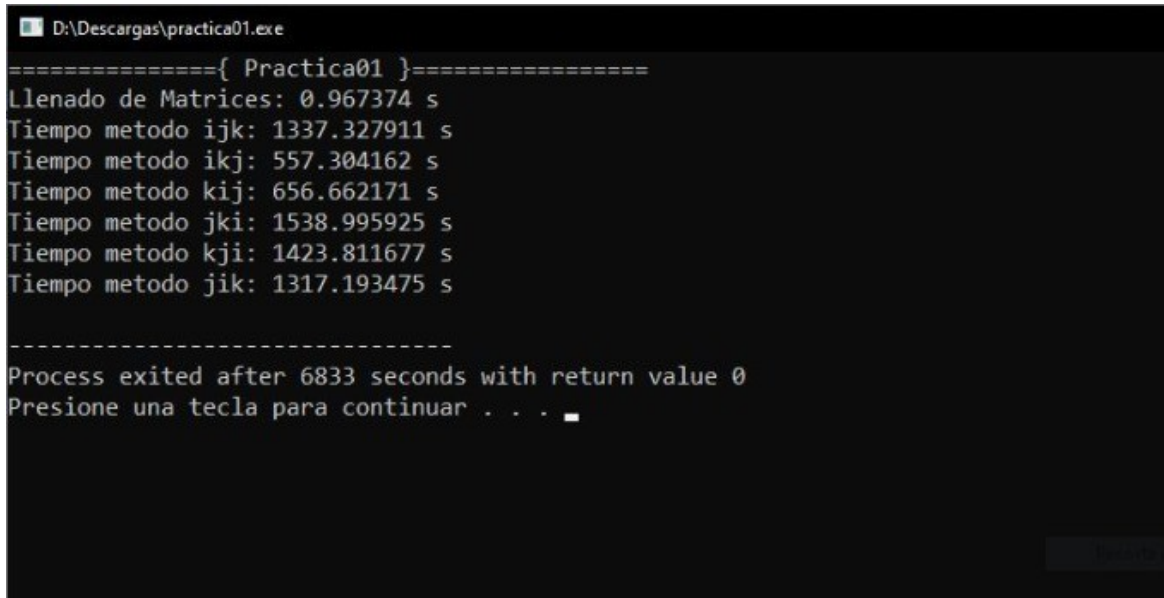
Tercer caso: matriz de (1000*1000)

```
D:\Descargas\practica01.exe

===== { Practica01 } =====
Llenado de Matrices: 0.035003 s
Tiempo metodo ijk: 6.926132 s
Tiempo metodo ikj: 4.690599 s
Tiempo metodo kij: 5.316670 s
Tiempo metodo jki: 10.144612 s
Tiempo metodo kji: 9.993767 s
Tiempo metodo jik: 5.400563 s

-----
Process exited after 46.12 seconds with return value 0
Presione una tecla para continuar . . .
```

Cuarto caso: matriz de (5000*5000)



```
D:\Descargas\practica01.exe
===== { Practica01 } =====
Llenado de Matrices: 0.967374 s
Tiempo metodo ijk: 1337.327911 s
Tiempo metodo ikj: 557.304162 s
Tiempo metodo kij: 656.662171 s
Tiempo metodo jki: 1538.995925 s
Tiempo metodo kji: 1423.811677 s
Tiempo metodo jik: 1317.193475 s

-----
Process exited after 6833 seconds with return value 0
Presione una tecla para continuar . . .
```

Quinto caso: matriz de (10000*10000)

No se obtuvieron resultados, pues el compilador tardo demasiado en realizar una accion y la ventana se cerro despues de una espera de mas de 3 horas.

Compara los resultados y explica ¿cuál disposición de ciclos for se ejecuta más rápido y cual más lento?, además de ¿por qué ocurre esto?

Comparando los valores obtenidos anteriormente podemos apreciar que la combinación de los ciclos más rápidos son los que se encuentran en ikj, kij, jik, mientras que los más lentos son los que se encuentran en jki, ijk y kji, lo anterior debido a que el acceso a la información en los ciclos for anidados resulta ser más inmediata para aquel ciclo que se encuentra más interno en la anidación, lo que permite que la diferencia entre los tiempos anteriormente obtenidos se deba a este acceso que se tiene a la información, ya que las dos disposiciones que arrojaron tiempos más cortos resultan ser aquellas que tienen el acceso más rápido a las columnas, lo que permite realizar las operaciones necesarias en un tiempo menor, mientras que para las dos disposiciones que arrojaron tiempos más largos se tiene el

acceso más rápido a los renglones, dificultando considerablemente la obtención de un resultado a menor tiempo.

Conclusión

En conclusion, al realizar operaciones de Single-Precision $A \times B + C$ en matrices que van desde 100×100 hasta $10,000 \times 10,000$, podemos apreciar cómo las dimensiones crecientes de las matrices tienen un impacto significativo en la complejidad computacional y el tiempo requerido para completar los cálculos. Este proceso, aunque puede resultar más largo a medida que aumenta el tamaño de las matrices, es esencial para tareas que requieren una gran cantidad de datos y cálculos numéricos, como la simulación, la investigación científica y el análisis de datos a gran escala. La eficiencia en la implementación de estos cálculos es crucial para optimizar el rendimiento de las aplicaciones que trabajan con conjuntos de datos extensos.