Examen 1: SAXPY

Barrios López Francisco
317082555

Castillo Montes Pamela
317165935

I. Objetivos

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

II. Realización

A. *Código base*

Partiendo del código de apoyo proporcionado por el manual referente a la práctica 1 [1]"Fig. 1",

```c
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

void get_walltime(double* wcTime){
  struct timeval tp;
  gettimeofday(&tp,NULL);
  *wcTime=(tp.tv_sec + tp.tv_usec/1000000.0);
}

int main(int argc, char* argv[]){
  int i,j,n=3;
  int **matrizA;
  double S1,E2;

  //Inicializando matrices
  matrizA = (int **)malloc(n*sizeof(int *));
  for(i=0;i<n;i++)
      *(matrizA+i) = (int *)malloc(n*sizeof(int));

  //Llenando matrices
  for (i = 0; i < n; ++i)
  {
    for (j = 0; j < n; ++j)
    {
      matrizA[i][j] = rand() %6;
```

Fig. 1. *Fragmento de código base*

Tenemos que se desea crear un programa que cuente con las siguientes características:

- Programa en C
- Realice la multiplicación de 2 matrices cuadradas y almacene el resultado en una tercera matriz.

Realizando las modificaciones pertinentes, las cuales en su mayoría se trataron de modificaciones en las funciones usadas para el cronometraje, dado que la función del tiempo proporcionada tenia dificultades de ejecución en el sistema operativo Windows debido a la librería <sys/time.h>, se decidió hacer uso de las bibliotecas #include<time.h> en conjunto con #include<windows.h> para obtener la frecuencia del procesador y posteriormente realizar la transformación a segundos.

$$tiempo = \frac{E2.QuadPart - S1.QuadPart}{frecuency.QuadPart;} \qquad (1)$$

Dicho lo anterior, obtenemos el siguiente código en C.

```c
#include<stdio.h>
#include<stdlib.h>
#include<time.h>
#include<windows.h>

int main (int argc, char* argv []){
    int i, j, k, n = 10000;          //Definición tamaño matriz
    int **matrizA, **matrizB, **matrizC;
    LARGE_INTEGER S1, E2;

    //Frecuencia de reloj: Uso en cronometraje
    LARGE_INTEGER frecuency;
    QueryPerformanceFrequency(&frecuency);

    // Inicializando matrices
    matrizA = (int **)malloc(n*sizeof(int *));
    matrizB = (int **)malloc(n*sizeof(int *));
    matrizC = (int **)malloc(n*sizeof(int *));
    for (i=0; i<n; i++)
        *(matrizA + i) = (int *)malloc(n*sizeof(int *));

    for (i=0; i<n; i++)
        *(matrizB + i) = (int *)malloc(n*sizeof(int *));

    for (i=0; i<n; i++)
        *(matrizC + i) = (int *)malloc(n*sizeof(int *));

    //Llenando aleatorio de matrices (0 a 6)
    for(i=0; i<n; i++){
        for(j=0; j<n; j++){
            matrizA[i][j] = rand() % 6;
            matrizB[i][j] = rand() % 6;
            matrizC[i][j] = 0;
        }
    }

    //get_walltime(&S1);
    QueryPerformanceCounter(&S1);   //Inicio de cronometro


    //Combinación i-j-k
    for (i=0; i<n; i++){
        for (j=0; j<n; j++){
            for (k=0; k<n; k++)
                matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
        }
    }

    QueryPerformanceCounter(&E2);       //Fin de cronometro
    //get_walltime(&E2);

    //Tiempo de ejecución
    double elapsed_time = (double)(E2.QuadPart - S1.QuadPart)/frecuency.QuadPart;
    printf("Tiempo método ijk: %f s\n", elapsed_time);
    return 0;
}
```

Fig. 2. *Código modificado*

III. Resultados

Nota: Los resultados presentados a continuación se realizaron en un equipo de computo con un procesador core i3-1115G4 de 11° generación, a una velocidad base de 3GHz, con sistema operativo Windows 10:

Siguiendo con las instrucciones dadas, se realizó la variación en el orden de los bucles "for" para el acceso a la información de la estructura de los datos en sus 6 combinaciones, así como el cronometro de tiempo de ejecución para cada uno de estos, obteniendo así, los siguientes resultados para los diferentes tamaños de matrices.

- 100 * 100
- 500 * 500

- 1,000 * 1,000
- 5,000 * 5,000
- 10,000 * 10,000

```
//Combinación i-j-k
for (i=0; i<n; i++){
    for (j=0; j<n; j++){
        for (k=0; k<n; k++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}

//Combinación j-i-k
for (j=0; j<n; j++){
    for (i=0; i<n; i++){
        for (k=0; k<n; k++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}

//Combinación i-k-j
for (i=0; i<n; i++){
    for (k=0; k<n; k++){
        for (j=0; j<n; j++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}

//Combinación k-j-i
for (k=0; k<n; k++){
    for (j=0; j<n; j++){
        for (i=0; i<n; i++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}

//Combinación j-k-i
for (j=0; j<n; j++){
    for (k=0; k<n; k++){
        for (i=0; i<n; i++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}

//Combinación k-i-j
for (k=0; k<n; k++){
    for (i=0; i<n; i++){
        for (j=0; j<n; j++)
            matrizC[i][j] += matrizA[i][k] * matrizB[k][j];
    }
}
```

Fig. 3. Combinaciones de ciclos ”for”

Se presenta el resultado a pantalla de la ejecución de todas las combinaciones en una matriz de tamaño 1,000 x 1,000 , cabe mencionar que no se presentan las ejecuciones de cada uno de los tamaños de matrices debido a la repetición y similitudes que existen entre estos. Por lo que, se opta por presentar una tabla de tiempos de ejecución “Tab. I”,

```
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método ijk: 14.479923 s
PS C:\Users\MiPC\Desktop>  & 'c:\Users\MiP
ngine-In-d3llu4yr.tfh' '--stdout=Microsoft
-dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método jik: 9.887924 s
PS C:\Users\MiPC\Desktop>  & 'c:\Users\MiP
ngine-In-qsop41ha.dsd' '--stdout=Microsoft
-dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método ikj: 5.471094 s
PS C:\Users\MiPC\Desktop>  & 'c:\Users\MiP
ngine-In-kzlxopik.syj' '--stdout=Microsoft
-dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método kji: 19.380131 s
PS C:\Users\MiPC\Desktop>  & 'c:\Users\MiP
ngine-In-bro4g4ug.ubr' '--stdout=Microsoft
-dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método jki: 19.812378 s
PS C:\Users\MiPC\Desktop>  & 'c:\Users\MiP
ngine-In-cn1zifle.wqr' '--stdout=Microsoft
-dbgExe=C:\MinGW\bin\gdb.exe' '--interpret
PS C:\Users\MiPC\Desktop> ./practica1.exe
Tiempo método kij: 5.760001 s
```

Fig. 4. Ejecución matriz 1,000 x 1,000

A. *Comparación*

Comparando los resultados de ejecución para cada combinación de los ciclos for, obtenemos la siguiente tabla de tiempos y su gráfica asociada.

TABLE I
TIEMPOS DE EJECUCIÓN

Tiempos de Ejecución [s]						
Matriz	**i-j-k**	**j-i-k**	**i-k-j**	**k-j-i**	**j-k-i**	**k-i-j**
100	0.00565	0.008302	0.007014	0.004418	0.00587	0.003913
500	0.512539	0.686956	0.582709	0.70837	0.820248	0.53937
1,000	5.1583	4.381764	4.709614	5.912671	6.338853	4.214326
5,000	903.820526	582.785436	426.981873	1770.15634	1885.43891	631.128793
10,000	8497.927346	6907.539109	3810.44075	18903.435096	15370.478452	4196.035042

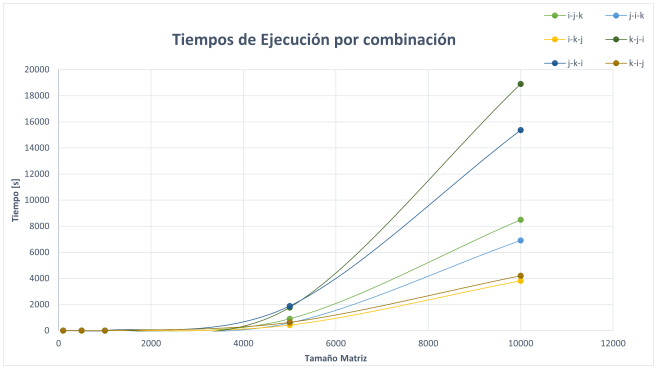

Fig. 5. Tendencias de tiempos

Como se puede observar en la “Tab. I”, para matrices de tamaños menores a 1,000, las diferencias entre tiempos de ejecución ronda entre los 0.5s y 2s aumentando la dificultad del análisis de la búsqueda de la combinación óptima.

A diferencia de las matrices de tamaños 5,000 y 10,000, [Fig. 5] donde se puede observar con facilidad la tendencia de alta duración para dos de las combinaciones, siendo estas:

- k-j-i
- j-k-i

Gracias a las líneas de tendencia presentadas, podemos inferir que para la combinación **k-j-i** se trata de la combinación con la ejecución más lenta mientras que **i-k-j** es la combinación más rápida en tiempo de ejecución.

El razonamiento del porque esto sucede radica en dos argumentos centrales, siendo el primero el acceso a la memoria de forma no contigua lo que nos puede generar un llenado y vaciado repetidamente de la caché del CPU, usualmente conocido como *”cache thrashing”* [2], provocando así que la memoria tarde más tiempo en responder.

Por otro lado, tenemos la localidad de referencia la cual hace referencia a la tendencia de acceder repetidamente a una ubicación o conjunto de ubicaciones de la memoria. [3].

B. *Combinación K-J-I*

- En este caso, i es la variable que cambia con mayor frecuencia. Esto hace que se itere sobre cada fila de A y C con mayor frecuencia, es decir, se realizan saltos a localidades de memoria no contiguas.

- La variable j es la siguiente a incrementar, esto cambia a la siguiente columna de B y C, pero para que esto pase, se debe de iterar sobre todas las filas de A y C, y pese a que esto significa saltar a valores no contiguos en la memoria, se incrementa el tiempo de ejecución.
- K es la última variable a incrementar. Por cada valor de K, se recorre una columna de A y una fila de B.

Esto significa que se está recorriendo A y C de arriba a abajo (filas), y una vez terminado esto, se recorre C y B por columnas. El hecho de recorrer primero filas y luego columnas, hace que el tiempo de ejecución sea muy grande en comparación con el resto de las combinaciones.

C. Combinación I-K-J

- Sin embargo, ahora se recorren primero las columnas de B[k][j] y de C[i][j]. Al recorrer localidades de memoria contiguas, el tiempo de ejecución se vuelve más rápido.
- Luego, para cada fila de B, se recorre una columna de A, y se vuelve a recorrer sobre localidades contiguas debido a j.
- Finalmente, i es el valor que más tiempo toma en incrementarse. Esto repercute en que se cambia de fila con menor frecuencia sobre A y C.

En comparación con las otras combinaciones, se recorre C y B por columnas, seguido de A por columnas. Esto resulta óptimo debido a que se recorren localidades de memoria contiguas, y se recorre cada fila de C y A una sola vez, evitando una mayor cantidad de saltos a localidades no contiguas.

IV. Conclusión

Se concluye que la diferencia clave entre las combinaciones es el orden en que se recorren las filas y columnas de las matrices en general, lo cual, por los argumentos previamente presentados puede representar un impacto en la localidad de memoria y el acceso a caché, afectando así su rendimiento.

Sin embargo, cabe destacar que el rendimiento también depende de la arquitectura y especificaciones de la computadora, así como el tamaño de las matrices usadas.

Se observó que recorrer filas implica un salto a una localidad de memoria *lejana*, mientras que recorrer columnas significa ir a la localidad de memoria inmediata. Las combinaciones que iteran mayormente sobre filas son las que mayor tiempo consumen, en contraste con las combinaciones que iteran primero sobre columnas, esto debido a que hacen saltos a localidades de memoria *lejanas* con mucha menor frecuencia.

V. Código

El código fuente se encuentra disponible en un repositorio de *GitHub*, disponible en el siguiente link.

References

[1] Ing. J. A. Ayala Barbosa, "Practica 1: SAXPY. [PDF]," [Online]. Available: https://www.notion.so/antonio-ayala/Sistemas-Distribuidos-71c7307438cb4218958fa57f2f26eddf?pvs=4

[2] C. McKenzie, "cache thrash," Software Quality, Aug. 2014, [Online]. Available:https://www.techtarget.com/searchsoftwarequality/definition/cache-thrash

[3] UCM, "Tema 6: Memoria Caché," Estructura de Computadores, Facultad de Informática. [Online]. Available: https://www.fdi.ucm.es/profesor/jjruz/WEB2/Temas/EC6.pdf