Introducción al HPC

Practica 1: SAXPY

*Merino Ortega Angel Nahum
**Parra Grimaldi Christopher Omar

***Abstract*— Arrays are stored in memory contiguously, meaning that the elements of the array are stored one after another in adjacent memory locations. This allows fast access to array elements using consecutive memory addresses. However, to get the most out of memory access speed, it is important to understand how the memory hierarchy works and apply cache optimization techniques. To improve performance when working with arrays, it is important to take full advantage of the memory hierarchy and minimize accesses to main memory, which is slower compared to the cache. Therefore, in this work we will analyze some applications by combining conditionals for filling matrices.**

Palabras Clave- Matrices,Combinacion,Multiplicacion,Memoria, Tiempo

INTRODUCCION

SAXPY : 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, yes 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].

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.

I. CODIGO

A. Crear un programa en C que multiplique 2 matrices cuadradas y el resultado lo almacene en un tercera matriz.

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>
#include <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, k, n=10000;
int ** matrizA, **matrizB, **matrizC;
double S1, E1 ;

// I n i c i a l i z a n d o m a t r i c e s

//MATRIZ A

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

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

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

//Llenando m a t ri c e s

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

matrizB[i][j] = rand( ) %6;
   }
}
get_walltime(&S1);


for (i = 0; i<n; ++i)
{
for (j=0; j<n; ++j)
{
for (k=0; k<n; ++k)
{
```

II. PRACTICA

A. *Posteriormente 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.*

B. *Cronometrar el tiempo que tarda cada una en resolverse*

C. *Ejecutar al menos 3 diferentes sistemas de matrices (100*100),(500*500),(1000*1000),(10000*10000)*

- **PARA IJK**

En (100*100)

```
valor de i: 96
valor de i: 97
valor de i: 98
valor de i: 99
Tiempo mÚtodo ijk: 0.006709 s

--------------------------------
Process exited after 0.1995 seconds with return value 0
Presione una tecla para continuar . . .
```

En (500*500)

```
valor de i: 497
valor de i: 498
valor de i: 499
Tiempo mÚtodo ijk: 0.396398 s

--------------------------------
Process exited after 0.4317 seconds with return value 0
Presione una tecla para continuar . . .
```

En (1000*1000)

```
valor de i: 997
valor de i: 998
valor de i: 999
Tiempo mÚtodo ijk: 3.795766 s
```

En (5000*5000)

```
valor de i: 4998
valor de i: 4999
Tiempo mÚtodo ijk: 561.209644 s
```

En (10,000*10,000)

```
valor de i: 9996
valor de i: 9997
valor de i: 9998
valor de i: 9999
Tiempo mÚtodo ijk: 4697.756263 s
```

PC: Usado Ryzen 5 5600g de 6 núcleos 12 hilos 3.90GHz 16gb de RAM DDR4 a 2999mHz

- **PARA IKJ**

En (100*100)

```
valor de i: 97
valor de i: 98
valor de i: 99
Tiempo mÚtodo ikj: 0.005999 s
```

En (500*500)

```
valor de i: 496
valor de i: 497
valor de i: 498
valor de i: 499
Tiempo mÚtodo ikj: 0.356378 s
```

En (1000*1000)

```
valor de i: 999
Tiempo mÚtodo ikj: 2.641843 s
```

En (5000*5000)

```
valor de i: 4996
valor de i: 4997
valor de i: 4998
valor de i: 4999
Tiempo mÚtodo ikj: 413.456204 s
```

En (10,000*10,000)

```
valor de i: 9998
valor de i: 9999
Tiempo mÚtodo ikj: 3166.492858 s
```

PC: Usado Ryzen 5 5600g de 6 núcleos 12 hilos 3.90GHz 16gb de RAM DDR4 a 2999mHz

- **PARA JIK**

En (100*100)

```
valor de j: 97
valor de j: 98
valor de j: 99
Tiempo mÚtodo jik: 0.006999 s
```

En (500*500)

```
valor de j: 497
valor de j: 498
valor de j: 499
Tiempo mÚtodo jik: 0.406271 s
```

En (1000*1000)

```
valor de j: 996
valor de j: 997
valor de j: 998
valor de j: 999
Tiempo mÚtodo jik: 3.537133 s
```

En (5000*5000)

```
valor de j: 4996
valor de j: 4997
valor de j: 4998
valor de j: 4999
Tiempo mÚtodo jik: 484.423383 s
```

En (10,000*10,000)

```
valor de j: 9998
valor de j: 9999
Tiempo mÚtodo jik: 3683.747497 s
```

PC: Usado Ryzen 5 5600g de 6 núcleos 12 hilos 3.90GHz 16gb de RAM DDR4 a 2999mHz

- **PARA JKI**

En (100*100).

```
valor de j: 96
valor de j: 97
valor de j: 98
valor de j: 99
Tiempo mÚtodo jki: 0.007000 s
```

En (500*500)

```
valor de j: 499
Tiempo mÚtodo jki: 0.484001 s
```

En (1000*1000)

```
valor de j: 998
valor de j: 999
Tiempo mÚtodo jki: 7.534258 s
```

En (5000*5000)

```
valor de j: 4996
valor de j: 4997
valor de j: 4998
valor de j: 4999
Tiempo: 1385.994611 s

--------------------------------
```

En (10,000*10,000)

```
valor de j: 9992
valor de j: 9993
valor de j: 9994
valor de j: 9995
valor de j: 9996
valor de j: 9997
valor de j: 9998
valor de j: 9999
Tiempo: 29918.425312 s

--------------------------------
Process exited after 2.992e+004 seconds with retu
```

PC: Usado Ryzen 3 2200g de 6 núcleos 12 hilos 3.40GHz 16gb de RAM DDR4 a 3200mHz

- **PARA KIJ**

En (100*100)

```
valor de k: 96
valor de k: 97
valor de k: 98
valor de k: 99
Tiempo: 0.034676 s
```

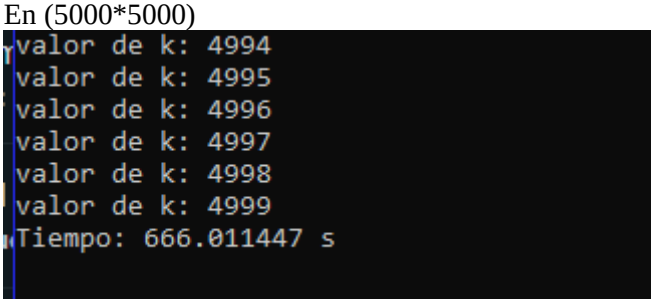
En (500*500)

```
valor de k: 498
valor de k: 499
Tiempo: 0.685057 s
```

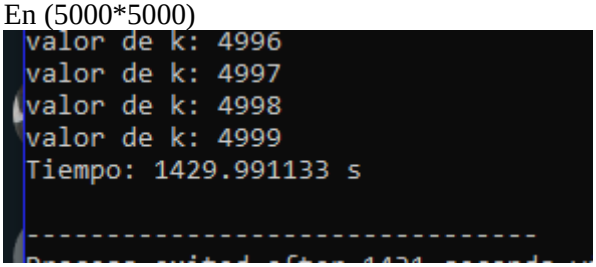
En (1000*1000)

```
valor de k: 997
valor de k: 998
valor de k: 999
Tiempo: 5.323927 s
```

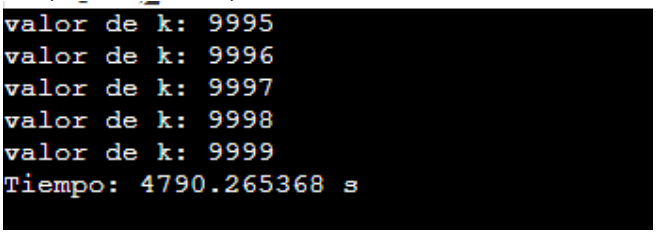
En (5000*5000)



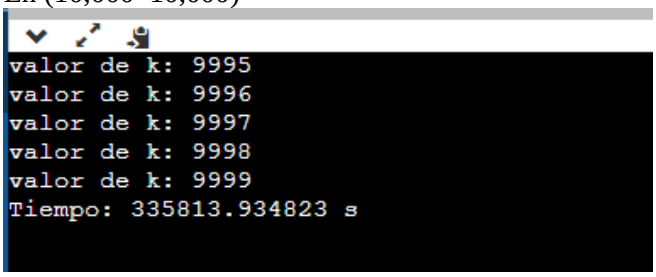
En (5000*5000)



En (10,000*10,000)



En (10,000*10,000)

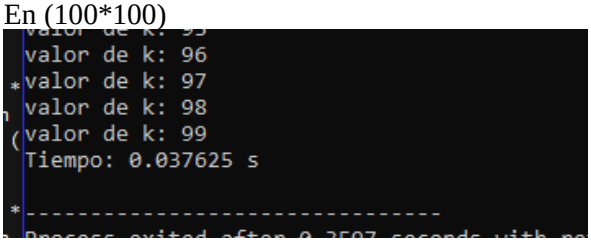


PC: Usado Ryzen 3 2200g de 6 núcleos 12 hilos 3.40GHz 16gb de RAM DDR4 a 3200mHz

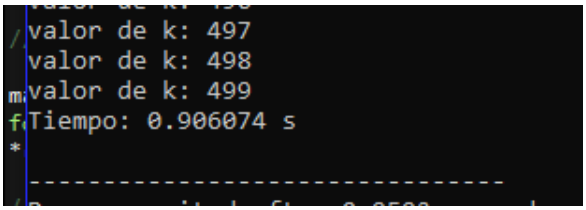
PC: Usado Ryzen 3 2200g de 6 núcleos 12 hilos 3.40GHz 16gb de RAM DDR4 a 3200mHz

- **PARA KJI**

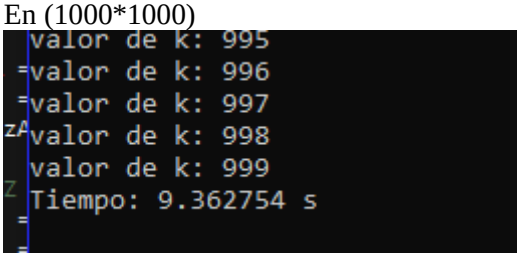
En (100*100)



En (500*500)



En (1000*1000)



D. *Compara los resultados y explica cuál disposición de ciclos for se ejecuta más rápido y cuál más lento. ¿Por qué ocurre esto?*

- *Con el punto anterior podemos notar que la combinación más rápida fue* para ikj **y** la más lenta fue kji. Esto debido a que los arreglos o en este caso las matrices, son conjuntos de datos que se encuentran cerca entre si en el conjunto de datos. Por lo que el metodo mas rapido es aquel que recorre las matrices A y B de tal forma que no tarda nada en encontrar el valor siguiente en el segmento de datos. En este caso el mejor orden fue el metodo ikj esto puede ser debido a que en la matrizB este valor se actualiza rápidamente ya que este se recorre de forma continua todo el tiempo, en la matriz A este valor queda estático durante unos ciclos y cuando se actualiza igual recorre la matriz de forma continua. Por último el recorrido en la matrizC para guardar los datos igual que en los otros caso es de forma continua.En resumen, el tiempo de ejecución al llenar y combinar matrices puede variar según diversos factores, incluyendo el algoritmo utilizado, el tamaño de las matrices y el hardware en el que se ejecuta el código. La optimización adecuada del código y la elección de algoritmos y bibliotecas eficientes pueden ayudar a mejorar el rendimiento en estas operaciones.

III. CODIGO

https://github.com/angelnahum0/SisDisGP01

IV. BIBLIOGRAFIA