PRÁCTICA 3

Entrenamiento con ARM: OpenMP y OpenCV

David Martínez García

Alejandro Gea Belda

Parte 1 - Paralelismo en sistemas de memoria centralizada: Introducción a OpenMP

Tarea 1.1: Estudio del API OpenMP

OpenMP es una interfaz de programación de aplicaciones (API) para la programación paralela en diferentes plataformas. La API define únicamente el estándar que hay que seguir para poder utilizar la interfaz. OpenMP está disponible en múltiples plataformas como Microsoft, Windows o Linux. En nuestro caso, su implementación está integrada en cualquier compilador de C y C + + de GNU con una versión 4.2 o superior.

Tarea 1.2: Estudio de OpenMP

1. Explique qué hace el código que se añade a continuación.

El código dado hace un par de operaciones sencillas. En primer lugar define tres vectores de tamaño 100 y asigna a los vectores a y b un valor a cada posición igual al índice del vector. Posteriormente, el programa suma en un bucle los vectores a y b, guardando el resultado en el vector c e imprimiendo por pantalla el resultado obtenido.

```
float a[N], b[N], c[N];

for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
```

```
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

En esencia, el programa hace estas operaciones básicas. Sin embargo, se utiliza OpenMP para que la ejecución del código sea simultánea en varios hilos de nuestro equipo.

2. Delimite las distintas regiones OpenMP en las que se está expresando paralelismo.

El uso de OpenMP modifica el código utilizando diferentes instrucciones que se identifican con el tipo #pragma que permiten modificar como se ejecuta nuestro programa. Una vez modificado el código, hay que asegurarse de que se compila añadiendo la opción -fopenmp para que el programa se ejecute correctamente.

La primera instrucción que se utiliza es `#pragma omp parallel shared (a ,b ,c, nthreads, chunk) private (i, tid)` y realiza varias instrucciones. La parte de `#pragma omp parallel` especifica que todo el código que hay dentro de la instrucción se puede ejecutar de forma paralela. La cláusula '**shared**' se utiliza para especificar qué variables van a ser compartidas por los diferentes hilos mientras que la cláusula '**private**' se utiliza para definir qué variables serán privadas en cada hilo. Esto implica que cada hilo

tendrá su propia copia de las variables definidas como private mientras que las variables definidas en shared serán comunes para todos los hilos..

```
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(i,tid)
{
    ...
    ...
    ...
}
```

La siguiente instrucción `omp_get_thread_num()` devuelve el identificador del hilo que invoca la función. Si ese hilo es el número 0, el programa devuelve el número de hilos activos mediante la función `omp_get_num_threads()`.

La segunda instrucción `#pragma omp for schedule(dynamic, chunk)` realiza varias instrucciones. La parte `#pragma omp for` permite que un bucle for se pueda ejecutar mediante varios hilos en paralelo. El fragmento `schedule(dynamic, chunk)` es una opción que se usa para controlar cómo se dividen las iteraciones del bucle entre los hilos. En este caso, la opción '**dynamic**' especifica que las iteraciones se asignan de manera dinámica a los hilos, es decir, que las iteraciones se irán asignando automáticamente a medida que estén disponibles para su uso. El parámetro '**chunk**' especifica cuántas iteraciones se asignan a la vez a cada hilo que, en nuestro caso, son 10 iteraciones por hilo.

```
#pragma omp for schedule(dynamic,chunk)
for (i=0; i<N; i++)
{
    c[i] = a[i] + b[i];
    printf("Thread %d: c[%d]= %f\n",tid,i,c[i]);
}
```

3. Explique con detalle qué hace el modificador de OpenMP schedule. ¿Cuáles pueden ser sus argumentos? ¿Qué función tiene la variable chunk en el código? ¿A qué afecta?

El modificador schedule en OpenMP se utiliza para controlar cómo se distribuyen las iteraciones de un bucle entre los hilos en un equipo paralelo. Esta permite controlar la asignación de trabajo a los hilos y, en consecuencia, la forma en que se aprovechan los recursos del equipo. El modificador schedule tiene varios argumentos que te permiten especificar cómo se asignan las iteraciones del bucle a los hilos:

- static: Con `schedule(static, chunk)`, se divide el bucle en bloques de un tamaño fijo llamado "chunk" y se asigna a los hilos en un patrón estático y equitativo. Esto significa que las iteraciones se distribuyen de manera uniforme entre los hilos al principio, y cada hilo obtiene un conjunto de iteraciones contiguas.

- dynamic: Con `schedule(dynamic, chunk)`, las iteraciones se asignan de manera dinámica a los hilos a medida que los hilos terminan su trabajo. Cada hilo obtiene un

conjunto de iteraciones, y cuando un hilo termina su conjunto, recibe otro conjunto de iteraciones disponibles.

- guided: Con `schedule(guided, chunk)`, se asignan inicialmente bloques grandes de iteraciones a los hilos, pero a medida que los hilos completan su trabajo, reciben bloques más pequeños.

- auto: Con `schedule(auto)`, OpenMP permite al compilador determinar la estrategia de planificación más adecuada.

- runtime: Con `schedule(runtime, chunk)`, implica que la política de planificación y el tamaño de "chunk" se determinarán en tiempo de ejecución, generalmente mediante la configuración de variables de entorno o directivas de control específicas proporcionadas por la implementación de OpenMP.

4. ¿Qué función tiene el modificador de OpenMP dynamic en el código?

Como se ha explicado anteriormente, al utilizar el modificador '**dynamic**' en nuestro código hace que el bucle for que se encarga de sumar los vectores a y b se ejecute en hilos diferentes. Estos hilos van recibiendo conjuntos de instrucciones conforme resuelven las instrucciones recibidas anteriormente sin esperar a que todos los hilos terminen de ejecutar sus órdenes.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
```

```
Thread 1 starting...
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
```

5. Investigue qué pasa si no declara como privadas las variables i y tid

Si no se declaran las variables i y tid como privadas en la región paralela, OpenMP tratará de manera predeterminada a estas variables como compartidas entre todos los hilos, lo que puede llevar a resultados incorrectos o comportamiento inesperado.

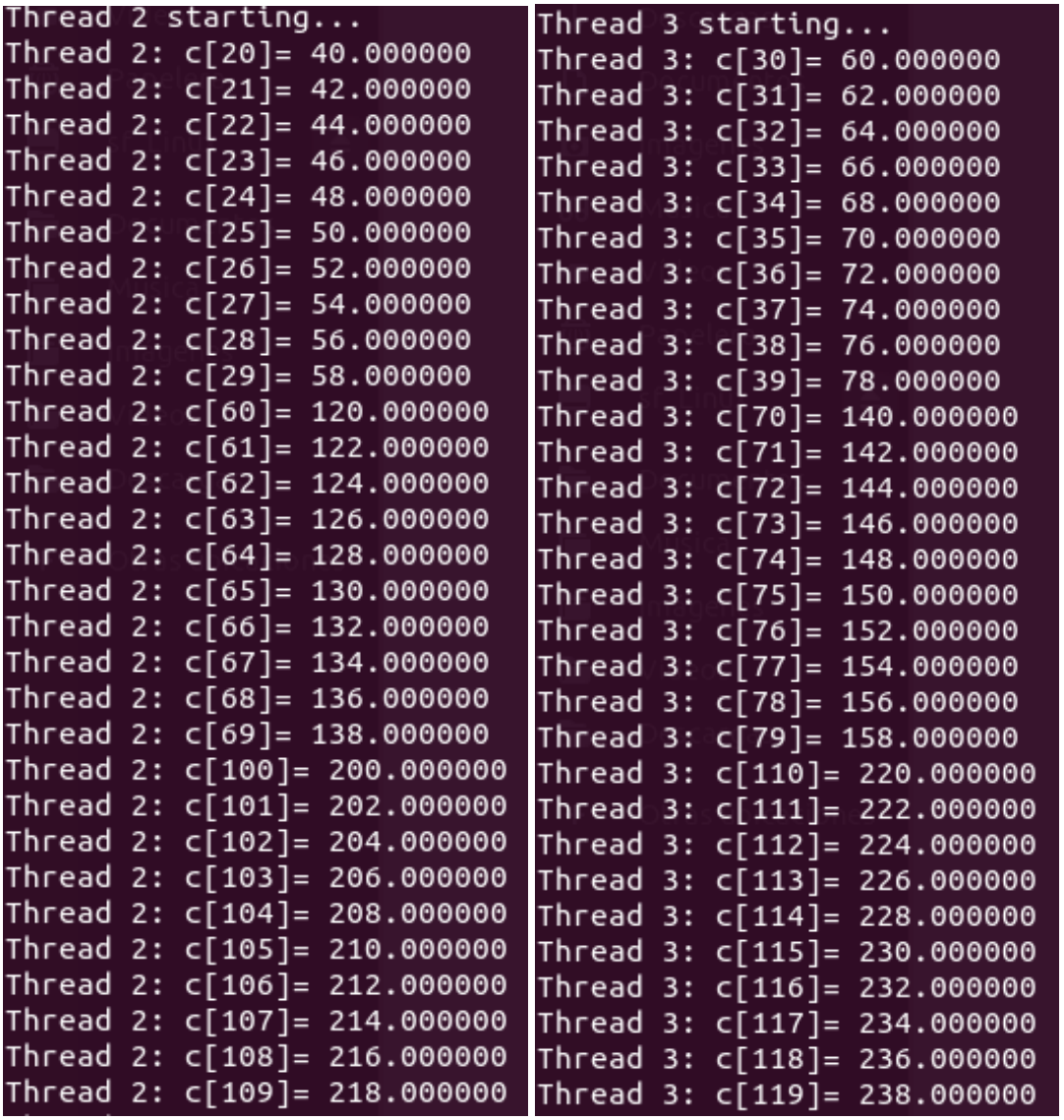
Dentro de una región paralela, OpenMP asume que las variables que no se declaran como privadas son compartidas, lo que significa que todos los hilos tendrán acceso y pueden modificar las mismas variables. Esto puede conducir a problemas de concurrencia, ya que múltiples hilos podrían estar intentando acceder o modificar las mismas variables i y tid al mismo tiempo.

Tarea J1. [Reto JEDI borde exterior]

Para emplear un planificador estático con OpenMP hay que cambiar el parámetro de nuestro pragma vector `#pragma omp for schedule(dynamic, chunk)` por el parámetro '**static**'. Para mejor visualización del código, se ha cambiado el tamaño de los vectores a 120 para que el reparto de la ejecución de nuestro código sea equitativo para los 4 procesadores que se están usando.

```
Thread 0 starting...
Thread 0: c[0]= 0.000000
Thread 0: c[1]= 2.000000
Thread 0: c[2]= 4.000000
Thread 0: c[3]= 6.000000
Thread 0: c[4]= 8.000000
Thread 0: c[5]= 10.000000
Thread 0: c[6]= 12.000000
Thread 0: c[7]= 14.000000
Thread 0: c[8]= 16.000000
Thread 0: c[9]= 18.000000
Thread 0: c[40]= 80.000000
Thread 0: c[41]= 82.000000
Thread 0: c[42]= 84.000000
Thread 0: c[43]= 86.000000
Thread 0: c[44]= 88.000000
Thread 0: c[45]= 90.000000
Thread 0: c[46]= 92.000000
Thread 0: c[47]= 94.000000
Thread 0: c[48]= 96.000000
Thread 0: c[49]= 98.000000
Thread 0: c[80]= 160.000000
Thread 0: c[81]= 162.000000
Thread 0: c[82]= 164.000000
Thread 0: c[83]= 166.000000
Thread 0: c[84]= 168.000000
Thread 0: c[85]= 170.000000
Thread 0: c[86]= 172.000000
Thread 0: c[87]= 174.000000
Thread 0: c[88]= 176.000000
Thread 0: c[89]= 178.000000
```

```
Thread 1 starting...
Thread 1: c[10]= 20.000000
Thread 1: c[11]= 22.000000
Thread 1: c[12]= 24.000000
Thread 1: c[13]= 26.000000
Thread 1: c[14]= 28.000000
Thread 1: c[15]= 30.000000
Thread 1: c[16]= 32.000000
Thread 1: c[17]= 34.000000
Thread 1: c[18]= 36.000000
Thread 1: c[19]= 38.000000
Thread 1: c[50]= 100.000000
Thread 1: c[51]= 102.000000
Thread 1: c[52]= 104.000000
Thread 1: c[53]= 106.000000
Thread 1: c[54]= 108.000000
Thread 1: c[55]= 110.000000
Thread 1: c[56]= 112.000000
Thread 1: c[57]= 114.000000
Thread 1: c[58]= 116.000000
Thread 1: c[59]= 118.000000
Thread 1: c[90]= 180.000000
Thread 1: c[91]= 182.000000
Thread 1: c[92]= 184.000000
Thread 1: c[93]= 186.000000
Thread 1: c[94]= 188.000000
Thread 1: c[95]= 190.000000
Thread 1: c[96]= 192.000000
Thread 1: c[97]= 194.000000
Thread 1: c[98]= 196.000000
Thread 1: c[99]= 198.000000
```



En las imágenes se puede observar como, al utilizar el modificador static, los 4 hilos que están trabajando reciben 10 instrucciones cada uno y hasta que no terminan todos no vuelven a recibir instrucciones. Gracias a este modo de trabajar, podemos saber de antemano qué instrucciones exactas va a recibir cada hilo de nuestro procesador antes de ejecutar el código.

Tarea 1.3: Paralelización de una aplicación con OpenMP

Apartado 0:

Se ha modificado el programa pedido para que muestre por pantalla dos columnas de datos, en una se mostrará el tamaño de las matrices con las que se realizan las operaciones y en la otra se muestra el tiempo de ejecución del programa. Después se ha utilizado gnuplot para representar la gráfica resultante. Para medir el tiempo se utilizará la función de OpenMP `omp_get_wtime()`, que devuelve el tiempo de ejecución del programa.

```
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gcc -fopenmp -o matri
x matrix.c
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ matrix > matrix.steps
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ gnuplot matrix.gpi
```

Las gráficas obtenidas para las diferentes arquitecturas son las siguientes:

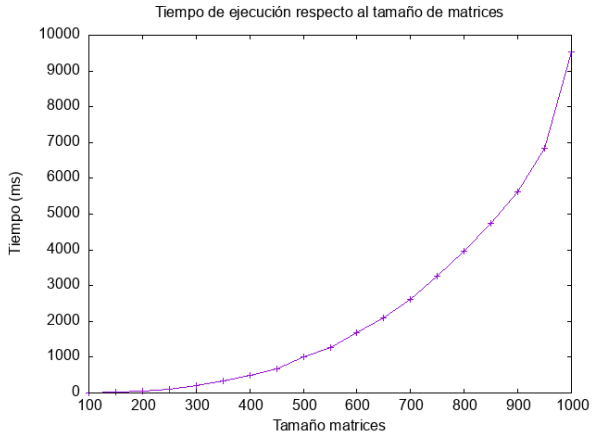


Figura 1: Arquitectura x86-64

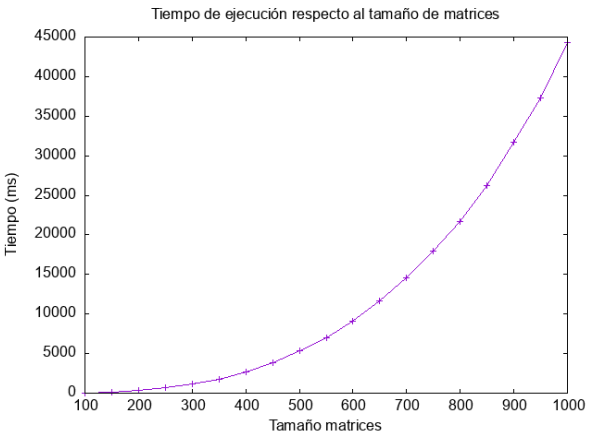


Figura 2: Arquitectura ARM

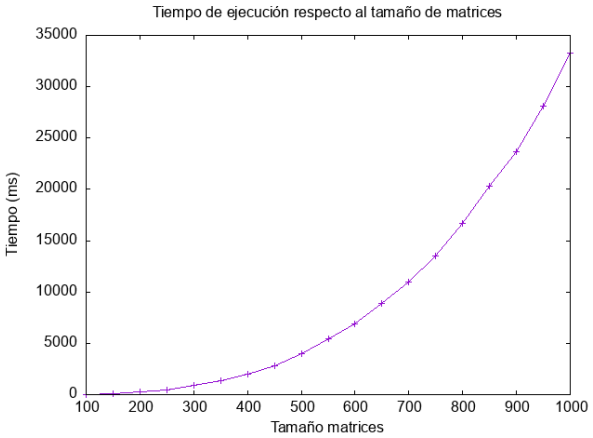


Figura 3: Arquitectura Risc V

Se puede observar que en ARM el programa tarda considerablemente más en ejecutarse para tamaños de matrices grandes. Por otro lado, en risc v el programa tarda menos en ejecutarse que para ARM pero el tiempo para tamaños grandes de matrices sigue siendo muy superior que en una arquitectura intel/AMD.

Ahora se va a hacer una comparación temporal en función del tipo de dato que contienen las matrices (int, double, long y char) compilados para arquitectura x86-64.

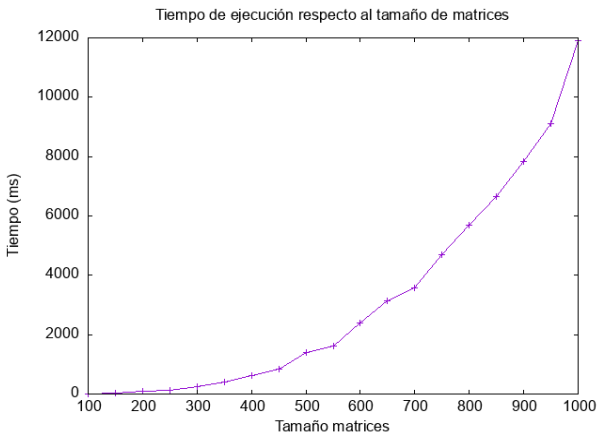


Figura 4: Tipo de dato: double

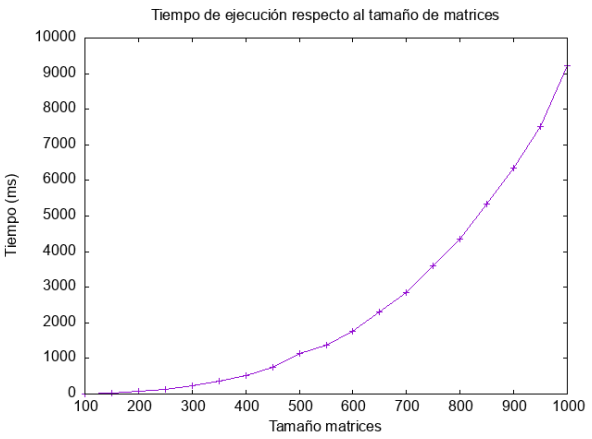


Figura 5: Tipo de dato: float

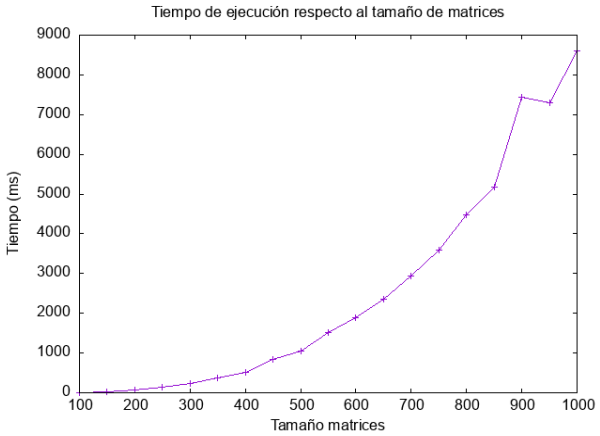


Figura 6: Tipo de dato: int

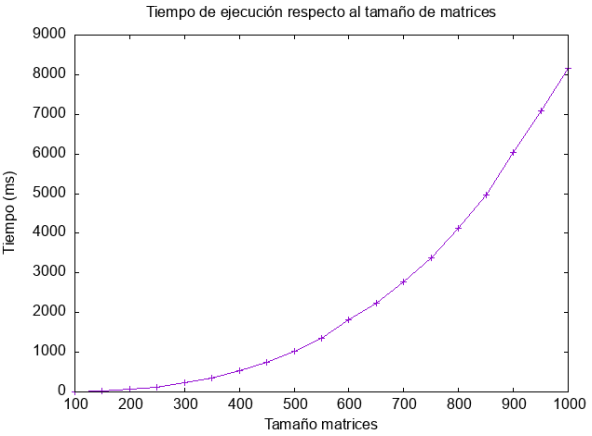


Figura 7: Tipo de dato: char

En las gráficas anteriores se puede observar que el tiempo de ejecución de nuestro programa varía ligeramente respecto al tipo de dato que utilicemos en las matrices. El tipo double es el que más tiempo tarda en ejecutarse, seguido del tipo long, int y, por último, el tipo char. Estas diferencias se aprecian claramente para tamaños altos, siendo de más de 3 segundos entre números definidos como double y enteros para un tamaño de matrices de 1000x1000.

Apartados 1, 2 y 3:

Se ha modificado el código del apartado anterior añadiendo las sentencias necesarias para paralelizar el código con OpenMP. Las sentencias son las siguientes:

- `omp_set_num_threads(numThreads)`: Para asignar la cantidad de hebras que trabajarán a la hora de ejecutar el código.
- `#pragma omp parallel shared(A,B,C,R,K) private(timeInit,timeFin)`: Para especificar que todo el código que hay dentro de esta instrucción se puede ejecutar de forma paralela.
- `#pragma omp for schedule(auto)`: Para permitir que el bucle for se pueda ejecutar mediante varios hilos en paralelo y con la opción auto para que el compilador determine la estrategia de planificación más adecuada.

El código quedaría de la siguiente manera:

```
timeInit=0, timeFin=0;
timeInit=omp_get_wtime();

omp_set_num_threads(numThreads); // Asignamos el numero de threads

#pragma omp parallel shared(A,B,C,R,K) private(timeInit,timeFin)
{
    // Asignación valores
    #pragma omp for schedule(auto)
    for(int i=0; i<d; i++){
        for(int j=0; j<d; j++){
            A[i][j]= (rand()%1000)/1000;
            B[i][j]= (rand()%1000)/1000;
            K[i][j]= rand()%255;
        }
    }

    // Producto vectorial
    #pragma omp for schedule(auto)
    for (int i = 0; i < d; i++) {
        for (int j = 0; j < d; j++) {
            C[i][j] = 0;
            for (int k = 0; k < d; k++) {
                C[i][j] += A[i][k] * A[k][j];
            }
        }
    }

    // Producto escalar
    #pragma omp for schedule(auto)
    for(int i=0; i<d; i++){
        for(int j=0; j<d; j++){
            R[i][j]=K[i][j]*C[i][j];
        }
    }
}

timeFin=omp_get_wtime();

printf("     %d \t        %f ms \n",d ,(timeFin-timeInit)*1000);
```

Al ejecutar el código asignando cada vez un número de threads distinto desde 1 a 4, podemos observar que el tiempo de ejecución es significativamente menor cuantos más threads utilizamos como se puede observar en las siguientes tablas:

Intel	Matrix.c	Parallelmatrix.c (Mono hebra)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	4,43	4,84	0,92	0,92
150	14,16	15,01	0,94	0,94
200	33,46	35,04	0,95	0,95
250	67,32	70,47	0,96	0,96
300	112,70	120,56	0,93	0,93
350	180,24	189,58	0,95	0,95
400	281,25	282,88	0,99	0,99
450	393,15	416,70	0,94	0,94
500	665,60	569,47	1,17	1,17
550	699,70	766,65	0,91	0,91
600	901,11	1022,53	0,88	0,88
650	1202,67	1326,46	0,91	0,91
700	1483,07	1657,24	0,89	0,89
750	1856,72	2046,33	0,91	0,91
800	2248,19	2486,19	0,90	0,90
850	2849,79	3083,61	0,92	0,92
900	3242,35	3671,68	0,88	0,88
950	3976,52	4432,48	0,90	0,90
1000	9371,03	9420,86	0,99	0,99

Tabla 1: Tiempo de ejecución, ganancia y eficiencia en mono-hebra (Intel)

Intel	Matrix.c	Parallelmatrix.c (2 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	4,43	3,55	1,25	0,62
150	14,16	12,45	1,14	0,57
200	33,46	24,12	1,39	0,69
250	67,32	41,77	1,61	0,81
300	112,70	74,95	1,50	0,75
350	180,24	110,33	1,63	0,82
400	281,25	162,24	1,73	0,87
450	393,15	232,01	1,69	0,85
500	665,60	385,41	1,73	0,86
550	699,70	408,57	1,71	0,86
600	901,11	815,47	1,11	0,55
650	1202,67	686,82	1,75	0,88
700	1483,07	852,45	1,74	0,87
750	1856,72	1060,51	1,75	0,88
800	2248,19	1283,15	1,75	0,88
850	2849,79	1609,02	1,77	0,89
900	3242,35	1920,53	1,69	0,84
950	3976,52	2176,17	1,83	0,91
1000	9371,03	4389,74	2,13	1,07

Tabla 2: Tiempo de ejecución, ganancia y eficiencia con 2 hebras (Intel)

Intel	Matrix.c	Parallelmatrix.c (3 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	4,43	2,74	1,62	0,54
150	14,16	6,44	2,20	0,73
200	33,46	15,94	2,10	0,70
250	67,32	31,76	2,12	0,71
300	112,70	54,16	2,08	0,69
350	180,24	86,99	2,07	0,69
400	281,25	144,98	1,94	0,65
450	393,15	195,81	2,01	0,67
500	665,60	610,23	1,09	0,36
550	699,70	277,42	2,52	0,84
600	901,11	357,55	2,52	0,84
650	1202,67	445,37	2,70	0,90
700	1483,07	1044,44	1,42	0,47
750	1856,72	720,97	2,58	0,86
800	2248,19	1065,45	2,11	0,70
850	2849,79	1061,81	2,68	0,89
900	3242,35	1330,74	2,44	0,81
950	3976,52	1459,71	2,72	0,91
1000	9371,03	2832,22	3,31	1,10

Tabla 3: Tiempo de ejecución, ganancia y eficiencia con 3 hebras (Intel)

Intel	Matrix.c	Parallelmatrix.c (4 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	4,43	3,30	1,34	0,34
150	14,16	6,89	2,06	0,51
200	33,46	16,57	2,02	0,50
250	67,32	34,58	1,95	0,49
300	112,70	55,05	2,05	0,51
350	180,24	78,62	2,29	0,57
400	281,25	115,13	2,44	0,61
450	393,15	127,64	3,08	0,77
500	665,60	233,97	2,84	0,71
550	699,70	284,81	2,46	0,61
600	901,11	351,48	2,56	0,64
650	1202,67	701,14	1,72	0,43
700	1483,07	524,70	2,83	0,71
750	1856,72	569,99	3,26	0,81
800	2248,19	780,70	2,88	0,72
850	2849,79	864,46	3,30	0,82
900	3242,35	1610,71	2,01	0,50
950	3976,52	1650,53	2,41	0,60
1000	9371,03	2142,27	4,37	1,09

Tabla 4: Tiempo de ejecución, ganancia y eficiencia con 4 hebras (Intel)

Como se puede observar, el tiempo de ejecución se reduce casi proporcionalmente al número de hebras utilizado para tamaños de matrices grandes pero no para tamaños de matrices pequeños, con lo que se puede concluir que la paralelización del código es un buen recurso para reducir el tiempo de ejecución cuando hay que resolver problemas de un gran tamaño. Si representamos gráficamente el tiempo secuencial y los tiempos paralelos para cada una de las hebras, obtenemos la siguiente figura:

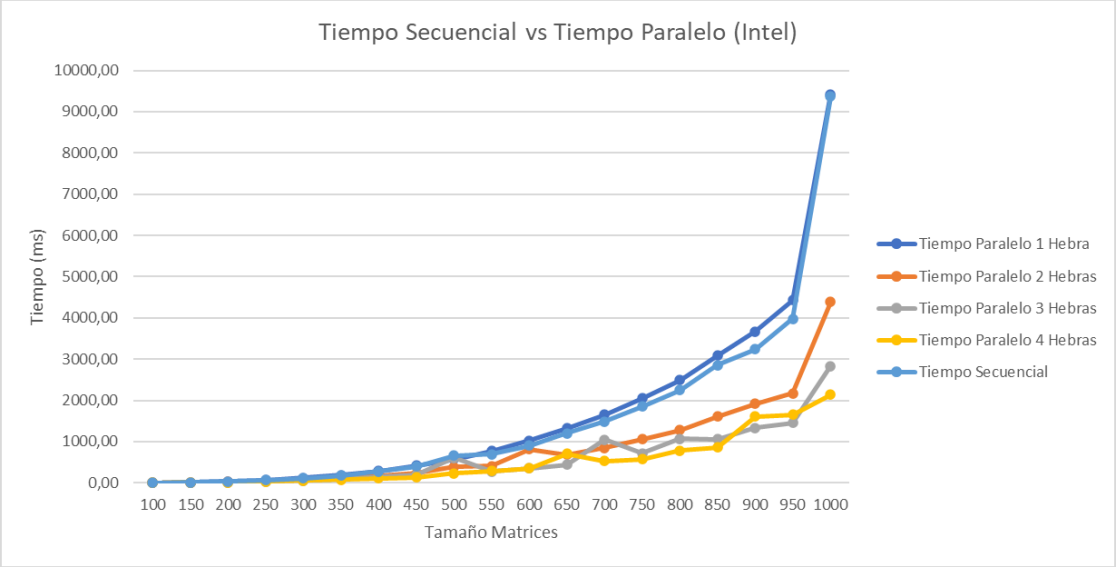


Figura 8: Tiempo secuencial vs tiempo paralelo (Intel)

También se ha calculado la ganancia en velocidad para cada uno de los tamaños de las matrices. Como se puede observar en respecto a la ganancia de los códigos paralelizados y al igual que en el tiempo de ejecución, para tamaños de matrices grandes se obtiene un valor de ganancia en velocidad muy próxima al número de hebras utilizado. De nuevo representado gráficamente obtendremos la siguiente figura:

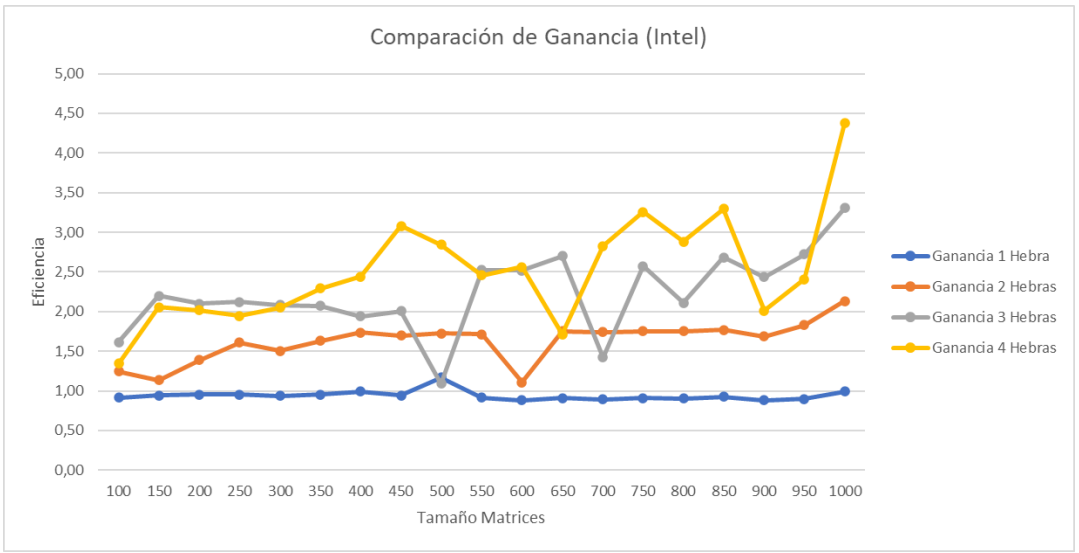


Figura 9: Comparación de ganancias (Intel)

En cuanto a la eficiencia, se puede ver como el rendimiento del código empeora conforme vamos aumentando el número de hebras en tamaños de matriz pequeños pero va aumentando conforme el tamaño de la matriz se va haciendo más grande. Sin embargo, si queremos saber con cuantas hebras es más eficiente el código, no solo debemos basarnos en la eficiencia ya que en este caso el mono-hebra sería el más eficiente. Para ello también hay que tener en cuenta la ganancia y el tiempo de ejecución, por tanto con 2 hebras reduciremos el tiempo de ejecución casi a la mitad y con valores de eficiencia próximos a 1. Si representamos la eficiencia gráficamente obtenemos la siguiente figura:

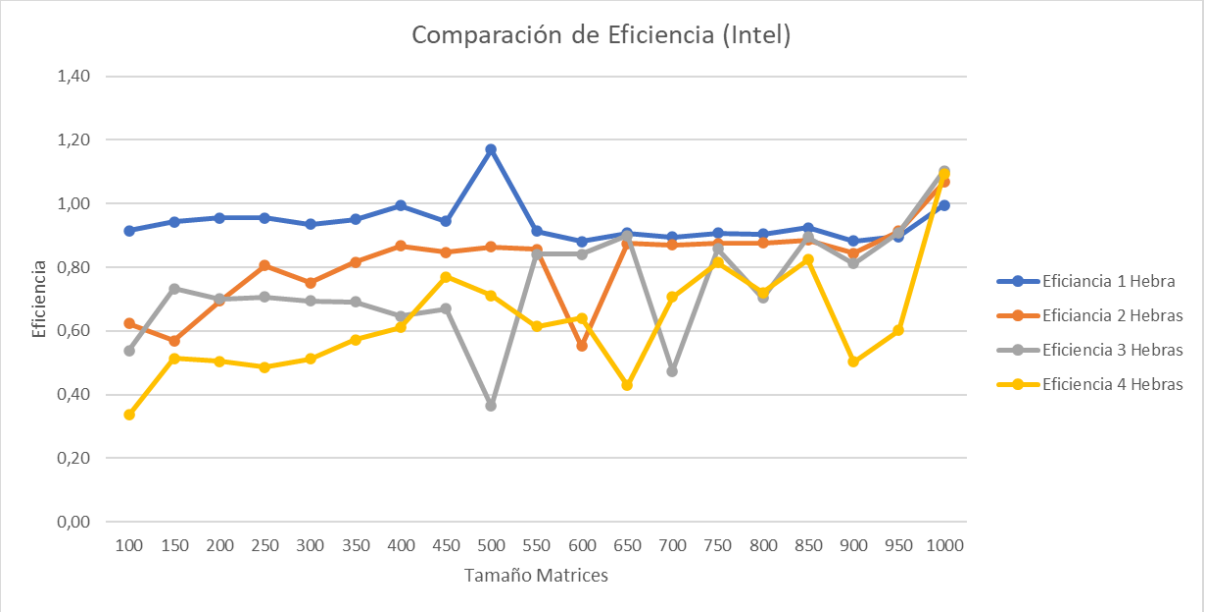


Figura 10: Comparación de eficiencia (Intel)

Finalmente vamos a comparar los resultados obtenidos anteriormente (AMD/Intel) con los que se obtendrían al compilar en ARM. Para ello compilamos el programa con arm-linux-gnueabihf-gcc -fopenmp -o y guardamos los valores obtenidos en las siguientes tablas:

ARM	**Matrix.c**	**Parallelmatrix.c (Mono hebra)**		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	25,97	26,36	0,99	0,99
150	82,80	82,35	1,01	1,01
200	191,25	191,30	1,00	1,00
250	378,13	374,95	1,01	1,01
300	650,95	643,85	1,01	1,01
350	1032,07	1028,03	1,00	1,00
400	1548,05	1528,97	1,01	1,01
450	2207,41	2183,47	1,01	1,01
500	3056,62	2997,64	1,02	1,02
550	4067,09	4031,65	1,01	1,01
600	5310,15	5227,34	1,02	1,02
650	6779,45	6673,10	1,02	1,02
700	8526,54	8321,09	1,02	1,02
750	10443,12	10297,39	1,01	1,01
800	12684,64	12562,11	1,01	1,01
850	15358,61	15200,90	1,01	1,01
900	18057,45	17958,78	1,01	1,01
950	25354,22	25767,04	0,98	0,98
1000	24887,48	24813,51	1,00	1,00

Tabla 5: Tiempo de ejecución, ganancia y eficiencia con mono-hebra (ARM)

ARM	Matrix.c	Parallelmatrix.c (2 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	25,97	26,04	1,00	0,50
150	82,80	75,79	1,09	0,55
200	191,25	121,54	1,57	0,79
250	378,13	229,92	1,64	0,82
300	650,95	384,02	1,70	0,85
350	1032,07	602,72	1,71	0,86
400	1548,05	878,80	1,76	0,88
450	2207,41	1236,19	1,79	0,89
500	3056,62	1702,89	1,79	0,90
550	4067,09	2229,88	1,82	0,91
600	5310,15	2906,94	1,83	0,91
650	6779,45	4661,74	1,45	0,73
700	8526,54	4555,49	1,87	0,94
750	10443,12	5746,58	1,82	0,91
800	12684,64	6688,89	1,90	0,95
850	15358,61	8433,80	1,82	0,91
900	18057,45	10391,41	1,74	0,87
950	25354,22	13572,11	1,87	0,93
1000	24887,48	14049,83	1,77	0,89

Tabla 6: Tiempo de ejecución, ganancia y eficiencia con 2 hebras (ARM)

ARM	Matrix.c	Parallelmatrix.c (3 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	25,97	24,35	1,07	0,36
150	82,80	40,40	2,05	0,68
200	191,25	84,43	2,27	0,76
250	378,13	164,24	2,30	0,77
300	650,95	269,21	2,42	0,81
350	1032,07	421,83	2,45	0,82
400	1548,05	873,25	1,77	0,59
450	2207,41	847,56	2,60	0,87
500	3056,62	1177,40	2,60	0,87
550	4067,09	1694,51	2,40	0,80
600	5310,15	1970,62	2,69	0,90
650	6779,45	2492,36	2,72	0,91
700	8526,54	3239,09	2,63	0,88
750	10443,12	4174,10	2,50	0,83
800	12684,64	5487,79	2,31	0,77
850	15358,61	5931,39	2,59	0,86
900	18057,45	7251,75	2,49	0,83
950	25354,22	9698,76	2,61	0,87
1000	24887,48	11310,53	2,20	0,73

Tabla 7: Tiempo de ejecución, ganancia y eficiencia con 3 hebras (ARM)

ARM	Matrix.c	Parallelmatrix.c (4 Hebras)		
Tamaño M	Tiempo (ms)	Tiempo (ms)	Ganancia	Eficiencia
100	25,97	26,67	0,97	0,24
150	82,80	36,81	2,25	0,56
200	191,25	70,67	2,71	0,68
250	378,13	201,75	1,87	0,47
300	650,95	288,21	2,26	0,56
350	1032,07	358,81	2,88	0,72
400	1548,05	514,92	3,01	0,75
450	2207,41	926,13	2,38	0,60
500	3056,62	1059,11	2,89	0,72
550	4067,09	1392,76	2,92	0,73
600	5310,15	1846,69	2,88	0,72
650	6779,45	2353,77	2,88	0,72
700	8526,54	2822,65	3,02	0,76
750	10443,12	3602,97	2,90	0,72
800	12684,64	4062,20	3,12	0,78
850	15358,61	5673,98	2,71	0,68
900	18057,45	6096,17	2,96	0,74
950	25354,22	7680,95	3,30	0,83
1000	24887,48	8972,08	2,77	0,69

Tabla 8: Tiempo de ejecución, ganancia y eficiencia con 4 hebras (ARM)

Y si representamos los valores en distintas gráficas, obtenemos las siguientes figuras:

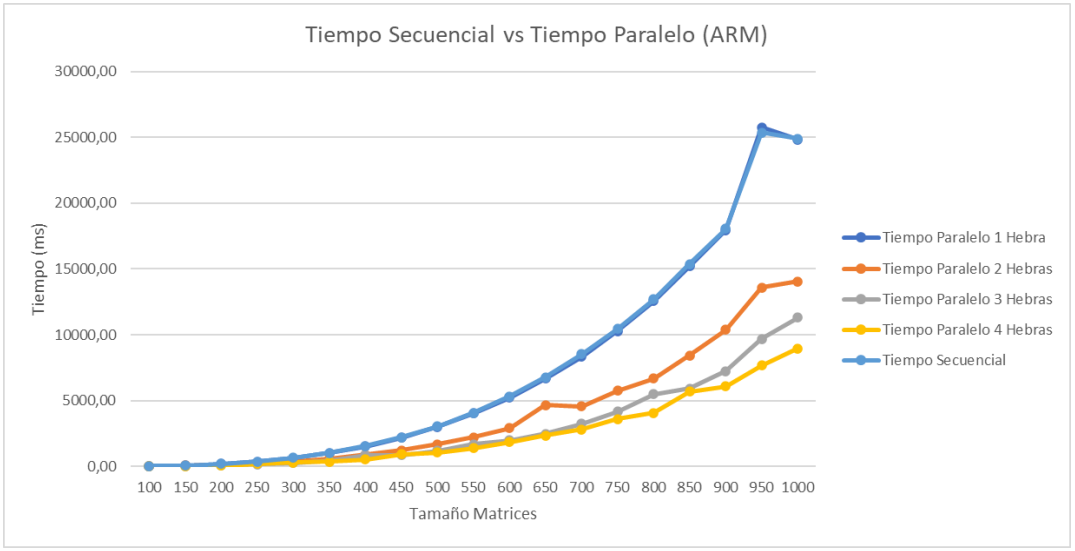


Figura 11: Tiempo secuencial vs tiempo paralelo (ARM)

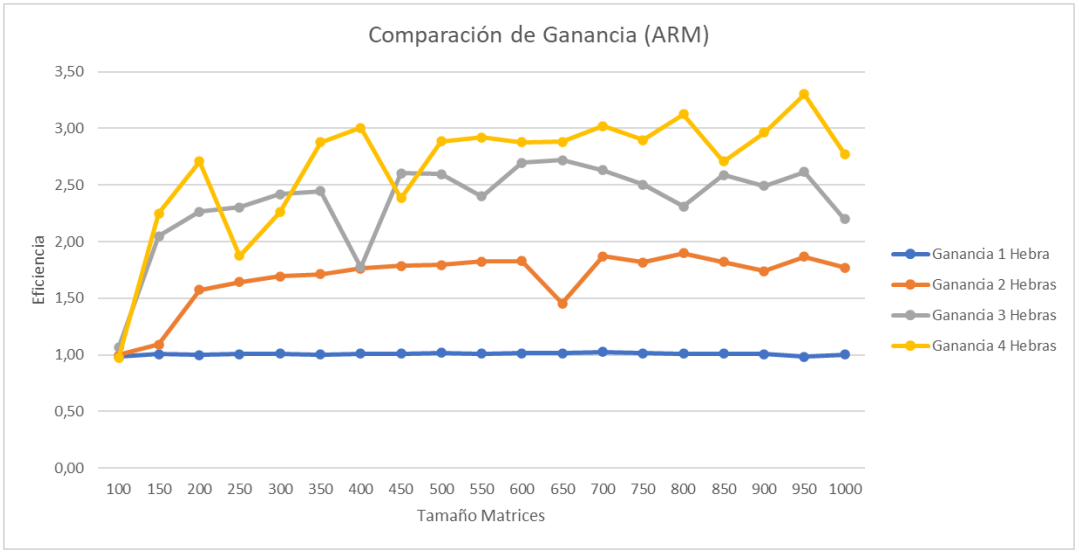


Figura 12: Comparación de ganancia (ARM)

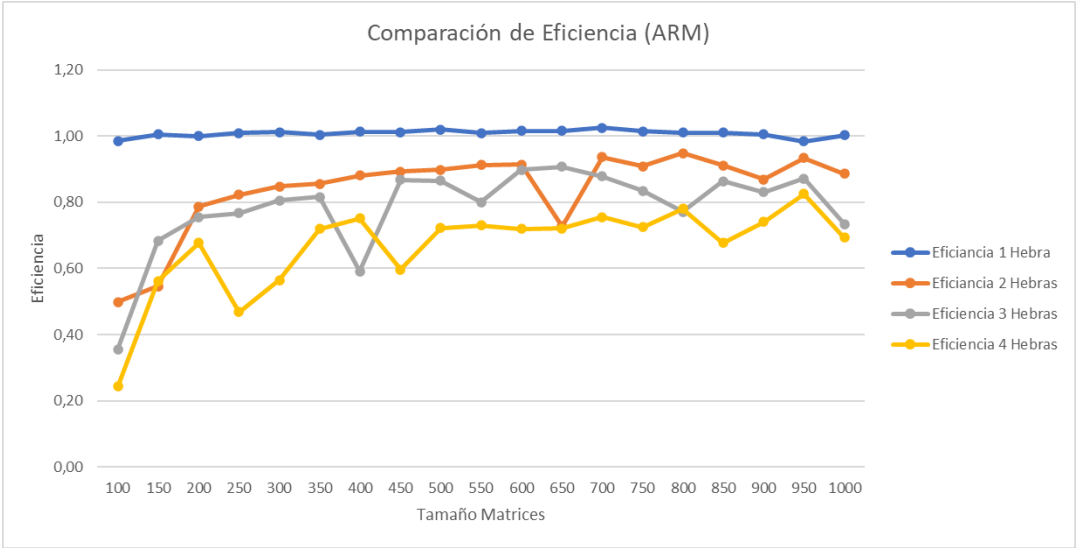


Figura 12: Comparación de eficiencia (ARM)

Como podemos observar, la principal diferencia entre los resultados obtenidos en ARM e Intel es que los tiempos de ejecución de ARM son mucho mayores que los de Intel, el código es menos eficiente para cualquier tamaño de matriz y la ganancia también es menor.

Tarea J2. [Reto JEDI borde exterior]

El motivo por el cual multiplicar 2 matrices A y B, siendo B transpuesta, produce menos fallos de caché es porque se accede a los elementos de la matriz B en un orden más amigable para la caché, especialmente en comparación con el orden en que se accederían los elementos de la matriz B si no fuera transpuesta. Esto se debe a cómo se almacenan las matrices en la memoria y cómo se realizan los accesos a la memoria durante la multiplicación de matrices.

Cuando se accede a un elemento de la matriz, también se cargan en la caché los elementos adyacentes. Por tanto, si esos elementos adyacentes se utilizan pronto se aprovecha la localidad espacial, ya que ya están en la caché.

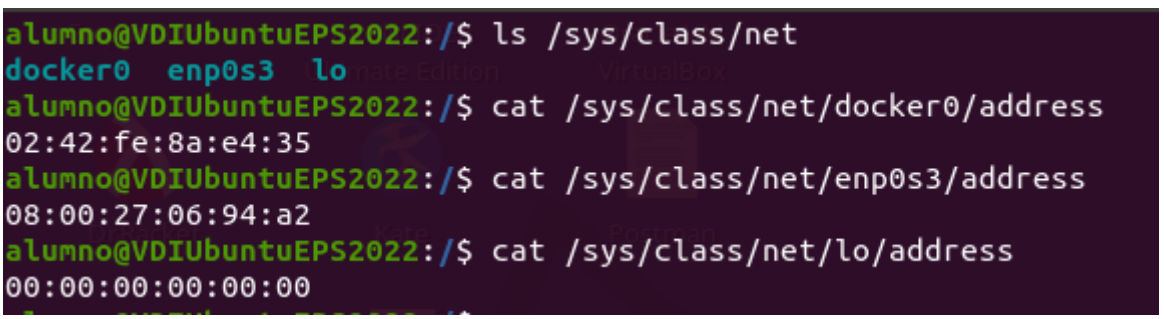
Cuando multiplicamos dos matrices A y B, los elementos de A son accedidos en filas, y los elementos de B son accedidos en columnas. Pero si la matriz B es transpuesta , entonces los elementos de B se acceden en filas en lugar de columnas y de esta manera las operaciones de multiplicación se aprovecharán mejor para la caché, ya que los elementos de B, al ser accedidos por filas, serán contiguos en memoria.

Para comprobarlo, vamos a ejecutar un código en el cual realizaremos la multiplicación de A*A, luego calcularemos la transpuesta de A y volveremos a realizar la multiplicación pero por su transpuesta (A*At). Ejecutaremos el código y compararemos el tiempo de ejecución. Obtenemos que la multiplicación de matrices de un tamaño 10000x10000 sin transpuesta ha tardado 708.852334 ms y con la transpuesta ha tardado 388.683423 ms, verificando así este hecho.

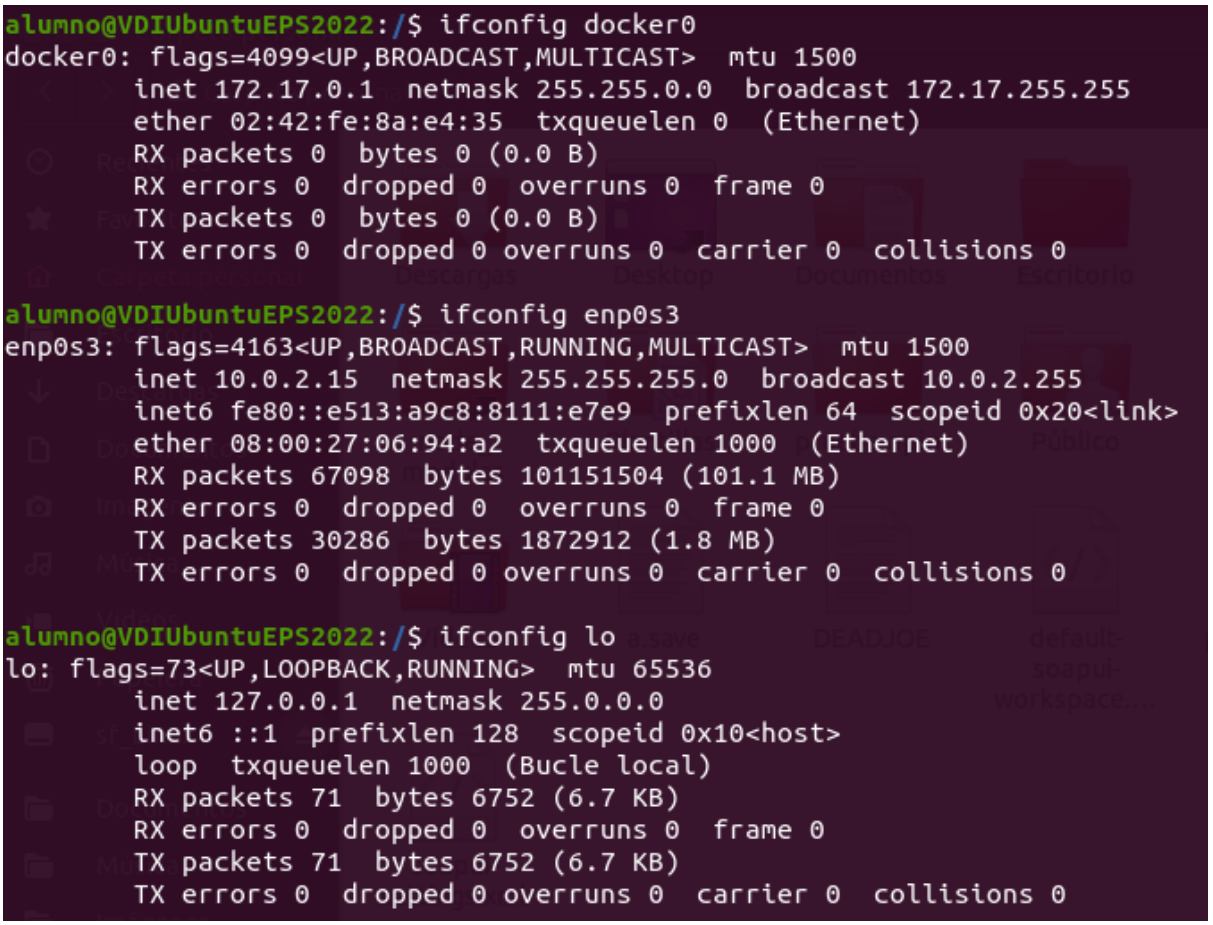
Parte 2 - Preparación y estudio del entorno de trabajo

Tarea 2.1: Acceso a Internet

Al utilizar el comando `ls /sys/class/net` en la terminal obtenemos tres interfaces llamadas docker0, enp0s3 y lo. Para obtener su dirección IP utilizamos el comando `cat /sys/class/net/<interfaz>/address`.



Al usar `ifconfig <interfaz>` podemos ver más detalles sobre la interfaz de red seleccionada.



La dirección MAC es un identificador único que cada fabricante le asigna a la tarjeta de red de sus dispositivos. Estas tarjetas de red se utilizan principalmente en dispositivos conectados a Internet. Un mismo dispositivo puede tener varias tarjetas de red (WiFi, Ethernet, etc) donde cada tarjeta tiene una dirección MAC diferente.

Las direcciones MAC están formadas por 48 bits binarios, representados generalmente por 12 dígitos hexadecimales agrupados en seis parejas separadas generalmente por dos puntos.

Las direcciones MAC de nuestras direcciones web se pueden ver con el comando `cat /sys/class/net/<interfaz>/address` y son:

- **docker0** → 02:42:fe:8a:e4:35
- **enp0s3** → 08:00:27:06:94:a2
- **lo** → 00:00:00:00:00:00

Por otro lado, el parámetro MTU (unidad de transmisión máxima) es el tamaño del mayor paquete permitido que se puede transferir a través de la conexión. Cuanto mayor sea la MTU, mayor cantidad de datos se pueden transferir en un solo paquete.

Para cambiar el MTU podemos utilizar el comando `sudo ifconfig <interfaz> mtu <valor>`. En este caso cambiaremos el valor de la interfaz docker0 a 2000 bytes/paquete.

Podemos comprobar que se ha cambiado correctamente utilizando los comandos anteriores.

```
alumno@VDIUbuntuEPS2022:/$ sudo ifconfig docker0 mtu 2000
[sudo] contraseña para alumno:
alumno@VDIUbuntuEPS2022:/$ ifconfig docker0
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 2000
        inet 172.17.0.1  netmask 255.255.0.0  broadcast 172.17.255.255
        ether 02:42:bd:a2:05:f5  txqueuelen 0  (Ethernet)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Tarea 2.2: Exploración del sistema Linux

Apartado a)

El directorio /proc es una especie de “ventana” que nos proporciona información sobre qué sucede en el kernel en un momento dado. Si ejecutamos el comando ls /proc obtenemos por pantalla bastantes datos. La gran mayoría de estos datos son números, esto se debe a que son la ID de ciertos procesos y sus subdirectorios contienen información específica de cada proceso. A continuación se muestra una parte de los contenidos del directorio /proc.

```
alumno@VDIUbuntuEPS2022:/$ ls /proc
1     1152   1428    24      3918  4468  5411  6684        crypto        net
10    1154   1451    245166  3919  4480  5415  7           devices       pagetypeinfo
100   116    1454    248076  398   4482  5425  8           diskstats     partitions
1002  1166   1460    248090  3992  4484  5441  805         dma           pressure
1007  1167   1478    248091  4     4492  5471  8444        driver        schedstat
1015  117    147831  25      40    4497  548   91          dynamic_debug scsi
102   1171   15      251371  4001  4584  5523  92          execdomains   self
```

Para ver el nombre que tienen los procesos asociados a esas ID podemos utilizar el comando `ps -pid=<id_proceso>`.

```
alumno@VDIUbuntuEPS2022:/$ ps --pid=1152
    PID TTY          TIME CMD
   1152 ?        00:00:01 run-cupsd
```

Apartado b)

Para obtener las características completas de nuestro procesador podemos utilizar varios comandos. Si queremos usar la información que tenemos en el directorio /proc existe el comando `cat /proc/cpuinfo` o podemos utilizar el comando `lscpu` que nos proporciona una salida similar. El comando anterior nos mostrará la misma información para cada uno de los cores de nuestro procesador. La información para uno de ellos es:

```
alumno@VDIUbuntuEPS2022:/$ cat proc/cpuinfo
processor       : 0
vendor_id       : AuthenticAMD
cpu family      : 25
model           : 80
model name      : AMD Ryzen 7 5800H with Radeon Graphics
stepping        : 0
cpu MHz         : 3193.962
cache size      : 512 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 4
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 16
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ht
 syscall nx mmxext fxsr_opt rdtscp lm constant_tsc rep_good nopl nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3 cx1
6 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignss
e 3dnowprefetch vmmcall fsgsbase bmi1 avx2 bmi2 invpcid rdseed clflushopt arat
bugs            : fxsave_leak sysret_ss_attrs null_seg spectre_v1 spectre_v2
bogomips        : 6387.92
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

En el apartado “flags” podemos ver todas las extensiones al repertorio ISA de nuestro procesador. En nuestro caso, la CPU sí que contiene extensiones vectoriales. Algunas de ellas son mmx, sse o avx y sus derivadas. Estas extensiones permiten utilizar registros de 64, 128 y 256 bits (con mmx, sse y avx respectivamente) para realizar varias operaciones de forma simultánea.

Para ver el número de procesadores que tiene nuestra CPU podemos utilizar el comando `cat /proc/cpuinfo | grep processor`. Podemos observar que tenemos 4 procesadores, que son los que hemos asignado a nuestra máquina virtual.

```
alumno@VDIUbuntuEPS2022:/$ cat /proc/cpuinfo | grep processor
processor       : 0
processor       : 1
processor       : 2
processor       : 3
```

Apartado c)

Con el comando `grep flags /proc/cpuinfo` obtenemos la información de las extensiones de instrucción específicas que admiten diferentes tipos de operaciones que posee nuestro procesador. En este caso, para que nuestro procesador pueda ejecutar el cálculo de un CRC de 32 bits tiene que tener las extensiones ‘crc32’ o ‘sse’.

```
alumno@VDIUbuntuEPS2022:/$ grep flags /proc/cpuinfo
flags           : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
 mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt rdtscp lm constant_tsc rep_good nopl nonstop_tsc
 cpuid extd_apicid pni pclmulqdq ssse3 cx16 sse4_1 sse4_2 x2apic movbe popcnt aes xsave avx rdran
d hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a misalignsse 3dnowprefetch vmmcall fsgsbase b
mi1 avx2 bmi2 invpcid rdseed clflushopt arat
```

En Linux, se pueden utilizar comandos como `mksum` o `crc32` seguidos de un archivo nuestro para calcular el valor CRC32 del archivo.

```
alumno@VDIUbuntuEPS2022:~/Escritorio/Empotrados/Practica3$ cksum matrix
2707334629 17176 matrix
```

Apartado d)

Utilizando el directorio /proc podemos poner el comando `cat proc/uptime` que nos devolverá el tiempo en segundos que lleva el ordenador encendido y el tiempo donde la CPU no ha estado realizando tareas de procesamiento.

```
alumno@VDIUbuntuEPS2022:/$ cat proc/uptime
2937.20 9409.90
```

También podemos utilizar el comando `uptime` directamente, lo que nos da el tiempo actual, el tiempo de ejecución del ordenador, el número de usuarios que hay conectados y la carga media de la CPU en los últimos 1, 5 y 15 minutos.

```
alumno@VDIUbuntuEPS2022:/$ uptime
 14:28:53 up 49 min,  1 user,  load average: 1,25, 1,35, 1,39
```

Apartado e)

Para obtener la versión de Linux y del compilador de GNU a través del directorio /proc podemos utilizar el comando `cat /proc/version`.

```
alumno@VDIUbuntuEPS2022:/$ cat  /proc/version
Linux version 5.14.0-1042-oem (buildd@lcy02-amd64-044) (gcc (Ubuntu 9.4.0-1ubunt
u1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2.34) #47-Ubuntu SMP Fri Jun
 3 18:17:11 UTC 2022
```

Otra manera de obtener esta información sin utilizar el directorio /proc es escribiendo los comandos `uname -r` y `gcc --version`. Estos comandos nos proporcionan la versión de Linux y de GNU.

```
alumno@VDIUbuntuEPS2022:/$ uname -r
5.14.0-1042-oem
alumno@VDIUbuntuEPS2022:/$ gcc --version
gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0
Copyright (C) 2019 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.  There is NO
warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
```

Apartado f)

Para ver todos los sistemas de archivos que soporta nuestro sistema podemos utilizar el comando `cat /proc/filesystems`. Si nos fijamos en la respuesta de este comando podemos ver que hay sistemas de archivos que tienen el prefijo “nodev” y otros que no. Este prefijo indica que esos sistemas no se pueden montar directamente en dispositivos

físicos, es decir, no se pueden utilizar directamente en un disco duro u otro tipo de almacenamiento.

El sistema de archivos bdef permite controlar los archivos de dispositivos que están almacenados en el directorio /dev.

El sistema de archivos configfs se encarga de configurar y administrar los módulos y dispositivos del sistema. Los archivos de configfs existen de forma independiente en el dispositivo, es decir, no son necesarios que estén cargados.

Los sistemas de archivos ext2, ext3 y ext4 permiten al sistema trabajar con archivos de gran tamaño. Su versión más moderna, ext4, permite trabajar con archivos de hasta 2^{50} bytes.

Los sistemas de archivos fuse y derivados permiten al usuario crear sus propios sistemas de archivos sin tener que modificar el código del kernel. El sistema fuse proporciona un "puente" desde el código creado hasta la interfaz del núcleo real.
El sistema de archivos vboxsf permite ejecutar máquinas virtuales independientes a partir del propio sistema operativo.

Apartado g)

Si ponemos el comando `cat /proc/stat` obtenemos lo siguiente:

Ese comando muestra muchos números por pantalla que dan información sobre la actividad del sistema. Las primeras 4 filas aportan información sobre los diferentes procesos que está realizando cada núcleo de nuestro procesador. El significado de las columnas, de izquierda a derecha es:

- **user** = procesos normales ejecutándose en modo usuario.
- **nice** = procesos “nice” ejecutándose en modo usuario.
- **system** = procesos ejecutándose en modo kernel.
- **idle** = espacio sin usar del procesador.
- **iowait** = procesos esperando a que se complete la E/S.
- **irq** = servicio de interrupciones.
- **softirq** = servicio de “softirqs”.

Las siguientes filas del comando muestran otro tipos de datos que son:

- **intr**: número de interrupciones atendidas desde el arranque del programa. La primera columna es el total de interrupciones y cada columna es el total de interrupciones de cada tipo en particular que nuestro sistema ha atendido.
- **ctxt**: procesos de todas las CPUs que se han pospuesto y reanudado posteriormente.
- **bitime**: hora a la que arrancó el sistema en segundos, contados desde el 1 de enero de 1970 (época Unix).
- **processes**: número de procesos e hilos creados por las llamadas de las funciones al sistema fork() y clone().
- **procs_running**: número de procesos que hay actualmente en ejecución en todos los cores.
- **procs_bloqued**: procesos bloqueados, esperando a que se complete la E/S.

Toda la información comentada anteriormente se puede ver de forma más clara utilizando el comando `mpstat -A`. Este comando muestra los datos anteriores en porcentajes que, en su conjunto, dan el 100% del procesamiento de cada core.

```
alumno@VDIUbuntuEPS2022:~$ mpstat -A
Linux 5.14.0-1042-oem (VDIUbuntuEPS2022)        15/11/23        _x86_64_        (4 CPU)

09:50:20     CPU    %usr   %nice    %sys %iowait    %irq   %soft  %steal  %guest  %gnice   %idle
09:50:20     all    9,97    0,31    5,25    0,06    0,00    0,06    0,00    0,00    0,00   84,34
09:50:20       0   10,76    0,30    5,69    0,05    0,00    0,01    0,00    0,00    0,00   83,19
09:50:20       1   10,97    0,34    5,81    0,04    0,00    0,04    0,00    0,00    0,00   82,81
09:50:20       2    8,71    0,33    4,92    0,09    0,00    0,12    0,00    0,00    0,00   85,83
09:50:20       3    9,46    0,29    4,61    0,07    0,00    0,05    0,00    0,00    0,00   85,53
```

Apartado h)

El comando `cat /proc/cmdline` muestra una línea de información sobre el kernel así como algunas opciones adicionales para el arranque del sistema.

```
alumno@VDIUbuntuEPS2022:~$ cat /proc/cmdline
BOOT_IMAGE=/boot/vmlinuz-5.14.0-1042-oem root=/dev/sda2 ro quiet splash
```

La información que nos devuelve es:

- **BOOT_IMAGE**: indica la ubicación del archivo del kernel que se está utilizando para el arranque. En este caso, el archivo del kernel está en el directorio /boot y se llama vmlinuz-5.14.0-1042-oem.
- **root**: especifica la partición raíz del sistema de archivos, es decir, el lugar en el sistema de archivos desde el cual se inicia el sistema. En este caso, la partición raíz está en el dispositivo /dev/sda2.
- **ro**: opción que indica que partición raíz se monta en modo de solo lectura (read-only).
- **quiet**: opción que indica que se deben reducir al mínimo las salidas del kernel durante el arranque.
- **splash**: opción que proporciona una interfaz gráfica agradable durante el arranque del sistema.

En resumen, la línea de comandos del kernel que se muestra indica la ubicación del kernel, la partición raíz, el modo de montaje de la partición raíz, y algunas opciones adicionales para el arranque del sistema.

Apartado i)

Las interrupciones suaves “softirqs” son una forma de manejar tareas de forma asíncrona en el kernel del sistema operativo. Estas tareas están asociadas a operaciones de red, temporizadores y otros eventos que pueden ocurrir en segundo plano.

El comando `cat /proc/softirqs` muestra información sobre las interrupciones suaves y la cantidad de tiempo que cada procesador le ha dedicado a cada tipo. Estos tipos son:

- **HI**: interrupciones suaves de alta prioridad.
- **TIMER**: interrupciones de temporizador.
- **NET_TX**: interrupciones relacionadas con la transmisión de datos a través de la red.
- **NET_RX**: interrupciones relacionadas con la recepción de datos a través de la red.
- **BLOCK**: interrupciones relacionadas con operaciones de bloqueo.

- **IRQ_POLL**: interrupciones relacionadas con la consulta de interrupciones (IRQ).
- **TASKLET**: interrupciones relacionadas con el manejo asíncrono del kernel.
- **SCHED**: interrupciones relacionadas con el planificador del kernel, que administra la ejecución de procesos.
- **HRTIMER**: interrupciones relacionadas con temporizadores de alta resolución.
- **RCU**: interrupciones relacionadas con operaciones de lectura/copia en el kernel.

```
alumno@VDIUbuntuEPS2022:~$ cat /proc/softirqs
                    CPU0       CPU1       CPU2       CPU3
          HI:          10          6          2          0
       TIMER:      484957     427975     529496     517448
      NET_TX:        1219          3          4       1169
      NET_RX:          88         36         61      40188
       BLOCK:        1095      27647     470121       6891
    IRQ_POLL:           0          0          0          0
     TASKLET:          75         33         18      13271
       SCHED:     1529356    1536950    1495065    1584039
     HRTIMER:           0          0          0          0
         RCU:     2813749    2839095    2715457    2726133
```

Apartado j)

Los números que aparecen en el directorio /proc son la ID de ciertos procesos y sus subdirectorios contienen información específica de cada proceso. Como se comentó anteriormente, puedes consultar el nombre que tienen los procesos asociados a esas ID utilizando el comando `ps -pid=<id_proceso>`.

Por ejemplo, el proceso con la ID 100 es el watchdog timer de nuestro sistema:

```
alumno@VDIUbuntuEPS2022:~$ ps --pid=100
    PID TTY          TIME CMD
    100 ?        00:00:00 watchdogd
```

Apartado k)

Para ver los sistemas de archivos ext3 o ext4 que hay montados en nuestro sistema podemos utilizar el comando `cat /proc/mounts | grep -E 'ext[34]'`. Este comando filtra todas las líneas que contengan alguno de estos dos nombres.

```
alumno@VDIUbuntuEPS2022:~$ cat /proc/mounts | grep -E 'ext[34]'
/dev/sda2 / ext4 rw,relatime,errors=remount-ro 0 0
```

En nuestro caso, únicamente está montado el sistema ext4. La salida del comando nos muestra el lugar de montaje (/dev/sda2), el tipo de sistema (ext4) y las opciones adicionales con las que está montado.

Apartado l)

Para ver la memoria libre que tenemos en nuestro sistema podemos utilizar el comando `cat /proc/meminfo`.

```
alumno@VDIUbuntuEPS2022:~$ cat /proc/meminfo
MemTotal:        4024720 kB
MemFree:          191628 kB
```

Si comparamos con el comando free, podemos observar que nos da los mismos datos de memoria disponible.

```
alumno@VDIUbuntuEPS2022:~$ free
               total        usado       libre  compartido búfer/caché  disponible
Memoria:     4024720     1451564      191780      100964     2381376     2196496
Swap:        4192252       69212     4123040
```

Apartado m)

Para ver los módulos que tiene instalado el kernel podemos utilizar el comando `cat /proc/modules`. Esto nos devolverá por pantalla una lista con todos los módulos con el formato "nombre_del_modulo tamaño uso_por referencia_usuarios".

```
alumno@VDIUbuntuEPS2022:~$ cat /proc/modules
xt_conntrack 16384 1 - Live 0x0000000000000000
xt_MASQUERADE 20480 1 - Live 0x0000000000000000
nf_conntrack_netlink 49152 0 - Live 0x0000000000000000
nfnetlink 20480 2 nf_conntrack_netlink, Live 0x0000000000000000
xfrm_user 36864 1 - Live 0x0000000000000000
xfrm_algo 16384 1 xfrm_user, Live 0x0000000000000000
```

Para ver esta información de forma más legible podemos utilizar el comando `lsmod`. Este comando nos da la misma información que el comando anterior pero ordenado en una tabla.

```
alumno@VDIUbuntuEPS2022:~$ lsmod
Module                  Size  Used by
xt_conntrack           16384  1
xt_MASQUERADE          20480  1
nf_conntrack_netlink   49152  0
nfnetlink              20480  2 nf_conntrack_netlink
xfrm_user              36864  1
xfrm_algo              16384  1 xfrm_user
xt_addrtype            16384  2
iptable_filter         16384  1
iptable_nat            16384  1
nf_nat                 49152  2 iptable_nat,xt_MASQUERADE
```

Apartado n)

Para ver los módulos que usan el módulo de bluetooth podemos utilizar los comandos anteriores filtrando con la opción `grep`. Sin embargo si escribimos los comandos no obtenemos ningún resultado. Esto se debe a que estos comandos devuelven los módulos cargados en el kernel.

```
alumno@VDIUbuntuEPS2022:/$ cat /proc/modules | grep bluetooth
alumno@VDIUbuntuEPS2022:/$ lsmod | grep bluetooth
```

Esto no significa que nuestro sistema no soporte bluetooth. Para ver que módulos pueden llegar a utilizar bluetooth podemos buscar en el directorio /lib/modules escribiendo el comando `ls /lib/modules/$(uname -r)/kernel/drivers/bluetooth`. Todos los archivos con extensión .ko son los módulos del kernel que se pueden cargar y que pueden hacer uso del módulo de *bluetooth*.

```
alumno@VDIUbuntuEPS2022:/$ ls /lib/modules/$(uname -r)/kernel/drivers/bluetooth/
ath3k.ko    bfusb.ko        bpa10x.ko   btbcm.ko    btmrvl.ko       btmtksdio.ko  btqca.ko  btrtl.ko   bt
usb.ko    hci_nokia.ko  hci_vhci.ko
bcm203x.ko  bluecard_cs.ko  bt3c_cs.ko  btintel.ko  btmrvl_sdio.ko  btmtkuart.ko  btrsi.ko  btsdio.ko  dt
l1_cs.ko  hci_uart.ko   virtio_bt.ko
```