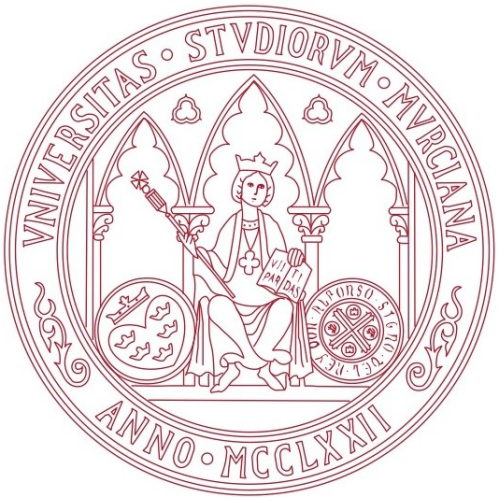

Arquitecturas Multimedia y de Propósito Específico

ACELERACIÓN DE UNA DNN CON AVX

Francisco Arnaldo Boix Martínez - 52041159W
Mariano Marín Ciller - 48837197D

Marzo 2023

Índice

1. Introducción

1.1. Como funciona AVX

Las instrucciones AVX se basan en el aprovechamiento de una arquitectura **SIMD**(Single Instruction Multiple Data) suele usarse para acelerar el procesamiento de grandes cantidades de datos. En lugar de procesar cada elemento de datos individualmente, SIMD permite que una única instrucción realizar la misma operación simultáneamente a múltiples elementos de datos en paralelo.

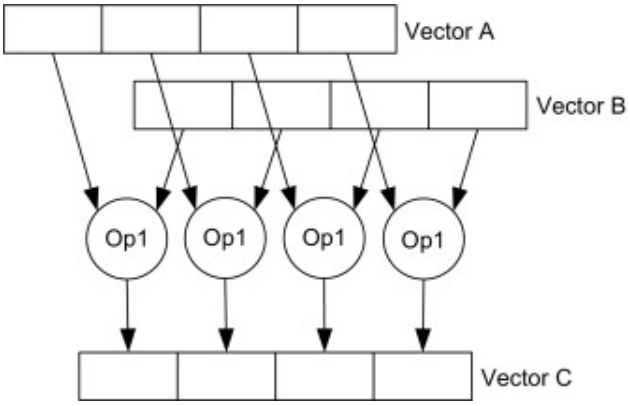

Figura 1: Como funciona una arquitectura SIMD (Referencia: [1])

1.2. Programación vectorial

Los intrínsecos y las etiquetas del compilador son dos formas diferentes de programar con SIMD. Ambas permiten aprovechar las extensiones SIMD del procesador para acelerar el procesamiento de datos.

Los intrínsecos son funciones de lenguaje C o C++ que permiten a los programadores utilizar las instrucciones SIMD del procesador en su código. Los intrínsecos proporcionan una interfaz de programación mas cercana al hardware y de esta forma nos asegura el máximo rendimiento, pero requieren de un buen nivel de conocimiento de los mismos.

Por otro lado, las etiquetas del compilador son directivas especiales que usa el compilador para tratar de transformar el código original en operaciones vectoriales. Con **GCC** se puede usar **-o3** para intentar vectorizar de manera automática. Suele ser una opción que mejora mucho el rendimiento pero no suele ser tan eficiente como los intrínsecos y puede generar resultados erróneos.

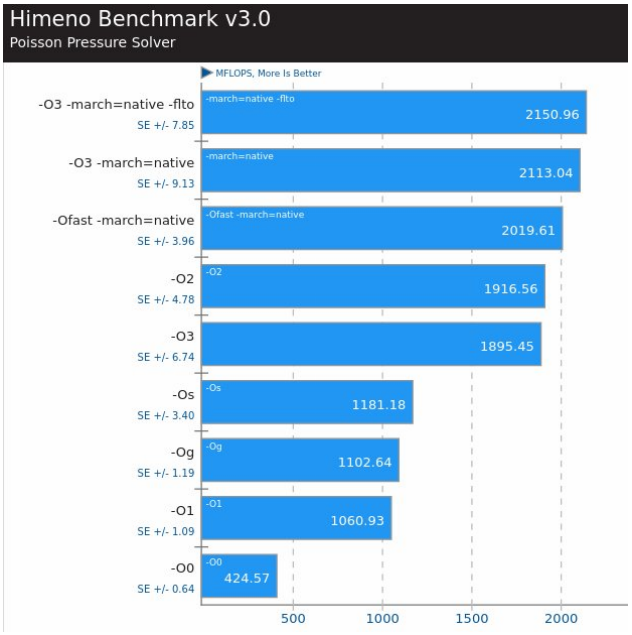

Figura 2: Comparativa de rendimiento usando distintos niveles de optimización con GCC 6.1 (Referencia: [2])

1.3. Aumento de rendimiento

La introducción del procesamiento vectorial ha supuesto una increíble mejora de rendimiento, desde su uso para instrucciones multimedia hasta el procesamiento de grandes cantidades de datos en el ámbito científico.
Speedups conseguidos con el uso AVX2 en operaciones básicas:

- **Addition** 217 %
- **Subtraction** 209 %
- **Multiplication** 221 %
- **Division** 300 %

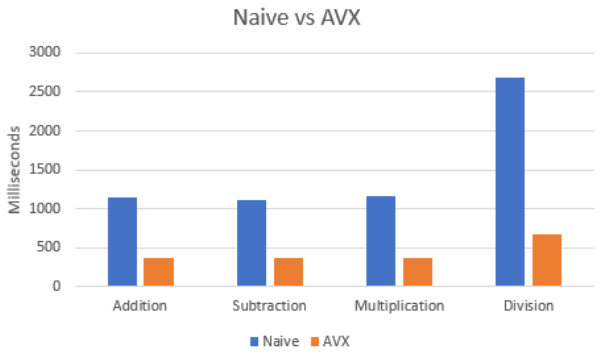

Figura 3: Mejora de rendimiento usando AVX (Referencia: [3])

El uso de instrucciones vectoriales ha ayudado a la emulación de la compleja arquitectura **Cell** de PS3. En esta imagen se puede ver el rendimiento en FPS con el uso en este orden de izquierda a derecha de las siguientes instrucciones: **SSE2**, **SSE4.1**, **AVX2/FMA**, y **Icelake tier AVX-512**.

Figura 4: Comparación de rendimiento emulando juegos de PS3(Referencia: [4])

Como se puede comprobar el rendimiento obtenido con AVX512, nos permitiría aprovechar los nuevos monitores de 240hz.

2. Objetivo de la practica

2.1. Multiplicación de matrices

En esta practica se optimizará el calculo de la multiplicación entre matrices. Donde tenemos dos matrices **MK** Y **KN**. Donde sus filas y columnas se definen de la siguiente manera:

- (M) numero de filas en la matriz **MK**
- (N) numero de columnas en la matriz **KN**
- (K) numero de columnas en la matriz **MK** y filas en la matriz **KN**

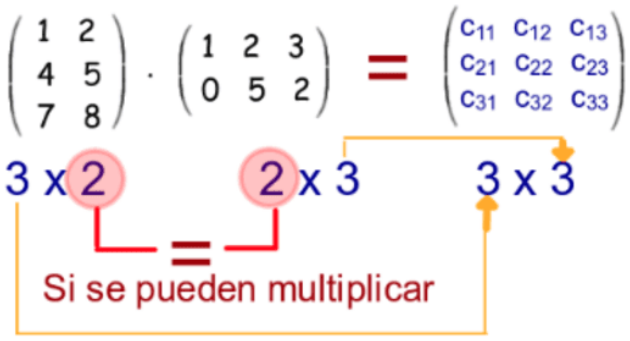

Figura 5: Ejemplo de multiplicación de matrices (Referencia: [5])

Como se puede ver en la figura, en este ejemplo **M=3, N=3 y K=3**, por lo cual obtenemos una matriz **3x3**

2.2. Equipo de pruebas

Para realizar las pruebas correspondientes a esta práctica hemos usado el portatil de **Mariano**, el cual cuenta con un procesador **AMD 5800H** junto a **16GB DDR4** y una **RTX 3050**, donde se listarán las características del procesador a continuación:

Componente	**Datos**
Cache Organization	-L1: 51KB, 256kB para L1I, 256KB L1D. Ambas de 8 vías asociativas, siendo la de datos de tipo "write-back" -L2: 4MB , 8 vías de 512 KB asociativas de tide tipo "write-back" -L3: 16MB, 16 vías de 16 MB asociativas deipode tipo "write-back".
Memory controller	-64GB on DDR4-3200, 32GB on LPDDR4-4266 -No soporta ECC
Instruciones Soportadas	MMX, EMMX,SSE,SSE2,SSE3,SSSE3,SSE4.1,SSE4.2,SSE4a,AVX,AVX2,ABM,BMI1,BMI2,FMA3,AES,RdRand,SHA ADX,CLMUL,F16C,X86-16,X86-32,X86-64,Real,Protected,SMM,FPU,NX,SMT,AMD-Vi,AMD-V,SenseMi
Specs	-Frecuencia: 3.2GHZ, Frecuencia turbo: 4.4GHZ -Nº núcleos: 8, Nº hilos: 16 -TDP: 45W
Diseño Zen3	-Etapas pipeline: 19 -Vias de busquedas de instruciones: 4 -Cada core tiene 4 Alus

Cuadro 1: Specs básicos del procesador (Referencia: [6])

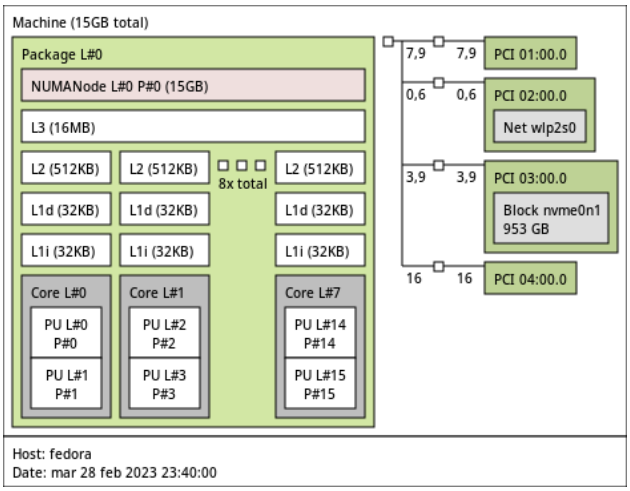

Figura 6: Procesador AMD 5800H, comando lstopo

3. Ejercicios

3.1. Ejercicio 1

Para calcular los GFLOPS se ha usado la siguiente formula:

$$GFLOPS = \frac{OPs \times 10^{-9}}{Tiempo \times 10^{-3}} = \frac{OPs}{Tiempo \times 10^{6}}$$

Donde el tiempo está representado en milisegundos, para obtener los GFLOPS, solamente hace falta dividir las operaciones totales en Punto Flotante entre el tiempo en milisegundos multiplicado por 1000000.

M	N	K	T Secuencial	Instruciones	OPS de PF	GFLOP/s
100	100	100	2 milliseconds	32.304.820	2.000.000	1
128	128	128	5 milliseconds	64.121.808	4.194.304	0.84
200	200	200	21 milliseconds	233.457.871	16.000.000	0.76
256	256	256	45 milliseconds	483.606.149	33.554.432	0.75
500	500	500	341 milliseconds	3.545.527.375	250.000.000	0.73
512	512	512	367 milliseconds	3.805.709.555	268.435.456	0.73
1000	1000	500	1367 milliseconds	14.099.469.668	1.000.000.000	0.73
1024	1024	512	1476 milliseconds	15.136.561.230	1.073.741.824	0.73
1024	1024	1024	2962 milliseconds	30.247.457.307	2.147.483.648	0.725

Cuadro 2: Tabla de tiempos secuenciales

Para complementar la información obtenida en la tabla anterior, se ha probado a ejecuatar la prueba con los siguientes parámetros: **M=1024**, **M=1024** y **K=32,64,128,256,512,1024**, generando matrices de 1024x1024, pero al variar el número de columnas de la primera matriz y el de filas de la segunda, tanto el tiempo de ejecución y el número de operaciones de punto flotante a realizar crece de manera lineal.

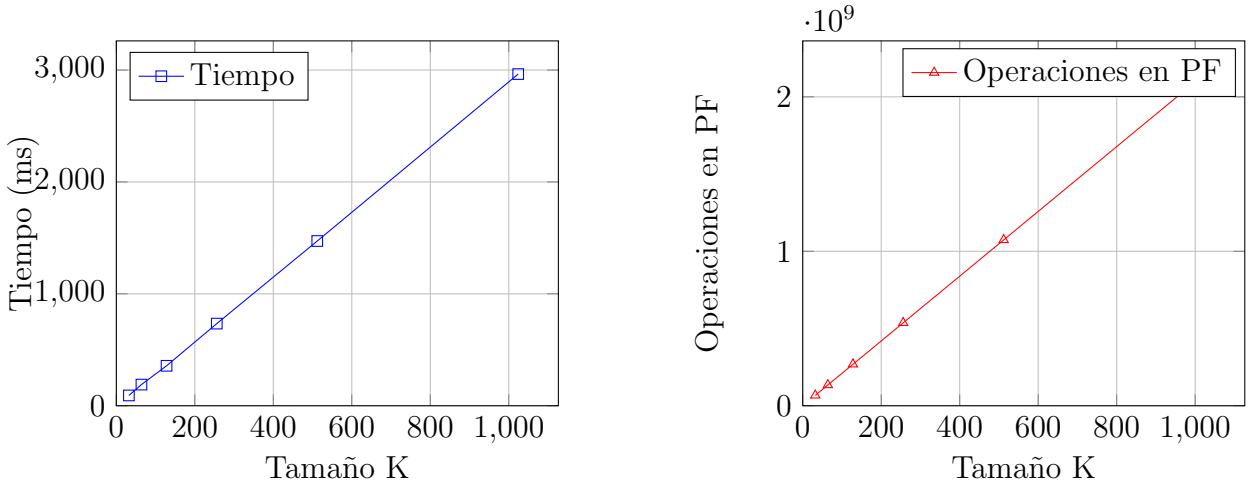

Figura 7: Gráficas de crecimiento del tiempo y operaciones en punto flotante

3.2. Ejercicio 2

El algoritmo de multiplicación de matrices que hemos desarrollado, toma la versión secuencial con orden de complejidad $O(n^3)$, y se efectúa una vectorización a la hora de multiplicar **filas** por **columnas**. Esta vectorización se lleva a cabo con extensiones SIMD de ISA, concretamente **AVX2**, proporcionadas por el compilador a través de `intrinsics`. Los `intrinsics` son funciones que te ofrece el compilador que te dan la funcionalidad una funcionalidad parecida a ensamblador, sin llegar a serlo realmente. **GCC** ofrece `intrinsics` para este tipo de instrucciones las cuales pueden ser consultadas en [7].

```
void practica2Linear(float *MK_matrix, float *KN_matrix, float *
    output_matrix, int M, int K, int N)
{
    const int vector_size = 8;
    const int vector_rest = K % vector_size;
    const int last_k = K - vector_size;

    for (int i = 0; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {

            __m256 acumulador = _mm256_setzero_ps();
            int k = 0;

            for (k = 0; k <= last_k; k += vector_size)
            {

                __m256 a = _mm256_loadu_ps(MK_matrix + i * K + k);
                __m256 b = _mm256_loadu_ps(KN_matrix + j * K + k);

                acumulador = _mm256_fmadd_ps(a, b, acumulador);
            }


            float temp[vector_size];
            _mm256_storeu_ps(temp, acumulador);

            output_matrix[i * N + j] = temp[0] + temp[1] + temp[2] + temp
    [3] + temp[4] + temp[5] + temp[6] + temp[7];
            float suma = 0.0;
            k = K - vector_rest;
            for (; k < K; k++)
            {

                suma = MK_matrix[i * K + k] * KN_matrix[j * K + k];
                output_matrix[i * N + j] += suma;
            }
        }
    }
}
```

La idea principal detrás del algoritmo es tratar el cálculo de cada una de las casillas de la matriz resultante como una multiplicación de vectores. Si observamos la Figura 8, la multiplicación de una fila por una columna de **K elementos**, puede traducirse por una operación `mac` (multiply and accumulate) sobre dichos vectores. En un caso ideal, podríamos multiplicar y acumular vectores de una longitud **arbitaria**, lo que permitiría que el calculo de una celda fuese realizado en una única instrución. Sin embargo, el soporte hardware necesario para esta hazaña sería **imposible**. Por tanto, tendremos que computar varios vectores de x tamaño.

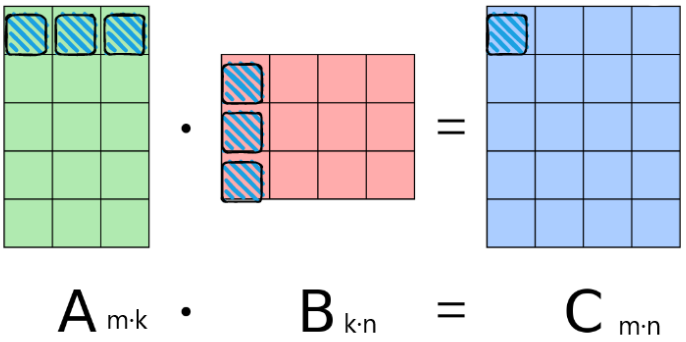

Figura 8: Multiplicación de matrices usando vectores

Por este motivo, los ISAs modernos ofrecen instrucciones con un tamaño de vector fijo. En el caso de las instrucciones **AVX2** este tamaño es de **256 bits**. Dependiendo del tipo de datos que se almacenen dentro de los vectores, cabrá un número mayor de elementos o no. Concretamente si usamos `floats` y sabiendo que cada float ocupa 32 bits en memoria, tenemos espacio para 8 elementos, puesto que `256 / 32 = 8`.

Si nos fijamos en el código hasta la **línea 29**, tenemos el cómputo de vectores de longitud 8 para el cálculo de una celda. Evidentemente, si el tamaño de los vectores (siendo k mayor que 8) no es múltiplo de 8, habrá una parte que nos habremos dejado de computar. Por este mismo motivo, existe ese último bucle **for**, que recorre desde el primer hasta el último elemento no computado para hacer el cálculo de forma secuencial.

En un momento dado, pensamos que sería una buena idea usar **máscaras** para computar esta última parte y no tener que hacer este pequeño cálculo en secuencial. Sin embargo, aplicando la **Ley de Amdahl**, no merecía la pena el esfuerzo teniendo el cuenta que como máximo van a ser 7 operaciones en secuencial por celda.

3.3. Ejercicio 3

M	N	K	T AVX	Instruciones	OPS de PF	GFLOP/s	Speedup
100	100	100	0	11.255.263	11.255.263	TimeE	TimeE
128	128	128	1	16.657.794	4.308.992	43.1	5
200	200	200	6	34.013.184	16.280.000	27.1	3.5
256	256	256	12	101.859.352	34.013.184	28.34	3.75
500	500	500	97	731.777.526	251.750.000	26	3.5
512	512	512	98	743.605.687	270.270.464	27.6	3.75
1000	1000	500	390	2.844.469.279	1.007.000.000	25.8	3.5
1024	1024	512	395	2.888.144.520	1.081.081.856	27.4	3.74
1024	1024	1024	781	5.718.117.180	2.154.823.680	27.6	3.8

Cuadro 3: Tabla de tiempos con AVX

Tras la realización de las pruebas anteriores se han creado las siguientes gráficas: donde se han tomado los datos de las tablas procedentes del ejercicio 1 y 3. Se han tomado 4 casos: donde **N,M,K** tienen los siguientes valores:**128,256,512,1024**.

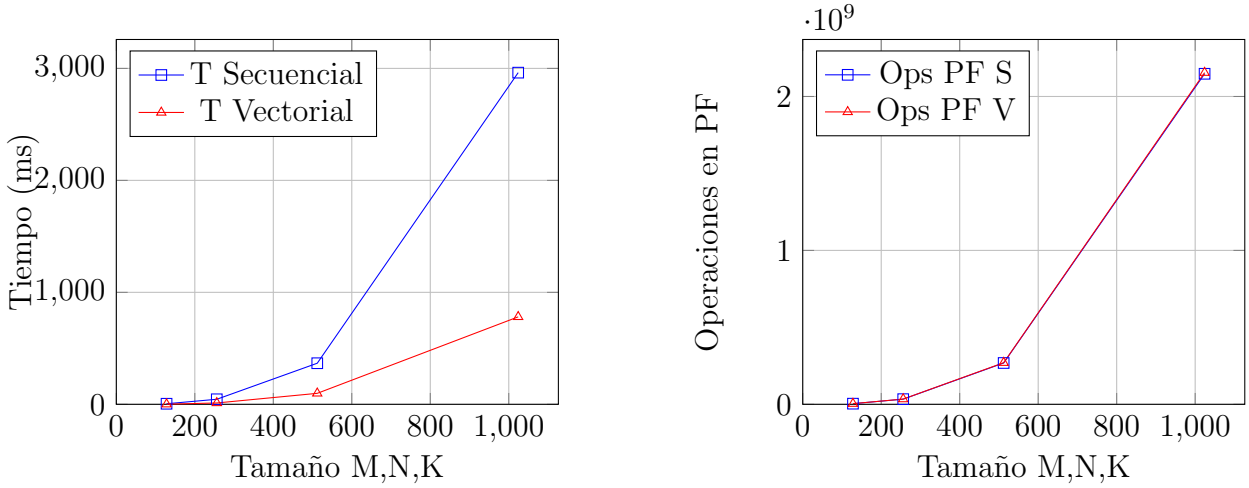

Figura 9: Gráficas de crecimiento del tiempo y operaciones en punto flotante

Tras analizar estas gráficas se puede concluir que el uso de **AVX2** reduce considerablemente el tiempo de ejecución del algoritmo de multiplicación de matrices, mientras que el uso de instruciones de punto flotante es prácticamente el mismo.

3.4. Ejercicio 4

Dentro de del directorio `p1` se pueden encontrar los archivos referentes a la realización de este ejercicio. Estos son `AMPE_dnn_p2.py` e `inferencia_p2.py`. En el primero de ellos, encontramos dos cambios respecto a la versión de la práctica 1:

- **practica2Linear:** En vez de usar los algoritmos de multiplicación de matrices que nos proporciona `PyTorch`, usamos el algoritmo vectorizado, desarrollado a lo largo de esta práctica.
- **cálculo de tiempo para la inferencia.**

La única diferencia en el archivo `inferencia_p2.py` frente a la versión de la práctica 1, es que usamos la clase `My_Dnn` del archivo `AMPE_dnn_p2.py` en vez de la versión de la primera práctica.

Sabiendo esto, hemos obtenido los siguientes resultados:

```
(ampe) [ampe@fedora p1]$ python3  inferencia_p2.py -i images/0.png -f
   weights/pesos-80.pt
Running the code for matrix multiplication
Running the code for matrix multiplication
Running the code for matrix multiplication
Predicted Digit = 0
Elapsed time =  0.0022907257080078125
(ampe) [ampe@fedora p1]$ python3  inferencia.py -i images/0.png -f weights
   /pesos-80.pt
Predicted Digit = 0
Elapsed time =  0.0013427734375
```

PyTorch Linear	Practica2Linear
0.0013427734375 s	0.0022907257080078125 s

Como podemos ver, obtenemos resultados **peores** que los que nos ofrece el framework. ¿Significa esto que la vectorización no funciona, o que no merece la pena el esfuerzo?. La respuesta es rotundamente **negativa**. La única conclusión que podemos sacar de estos resultados es que **PyTorch** hace optimizaciones mas allá de la vectorización, las cuales quedan fuera del alcance de la asignatura.

4. Pruebas adicionales

Para complementar hemos usado **OMP** para usar la vectorización en todos los núcleos del procesador, obteniendo los siguientes resultados:

```
Maci@fedora ~/P/v/d/p/practica2 (main)> ./practica2_test -LINEAR -N=1024 -M=1024 -K=1014
Changing N to 1024
Changing M to 1024
Changing K to 1014
Running the code for optimized matrix multiplication
Running the code for matrix multiplication
Test passed correctly
Maci@fedora ~/P/v/d/p/practica2 (main)> ./practica2_test -LINEAR -N=1390 -M=1874 -K=1123
Changing N to 1390
Changing M to 1874
Changing K to 1123
Running the code for optimized matrix multiplication
Running the code for matrix multiplication
Test passed correctly
Maci@fedora ~/P/v/d/p/practica2 (main)> ./practica2_eval_seq -LINEAR -N=1390 -M=1874 -K=1123
Changing N to 1390
Changing M to 1874
Changing K to 1123
Running the code for optimized matrix multiplication
Time to compute sequential (original) version: 8616 milliseconds
Maci@fedora ~/P/v/d/p/practica2 (main)> ./practica2_eval_opt -LINEAR -N=1390 -M=1874 -K=1123
Changing N to 1390
Changing M to 1874
Changing K to 1123
Running the code for matrix multiplication
Time to compute optimized version: 232 milliseconds
```

Listing 1: Test con OMP

Se puede observar como se pasan los test de manera satisfactoria y el incremento de rendimiento es muy notorio. Para obtener este resultado solamente se ha añadido este pragma: **pragma omp parallel for collapse(2)** antes del primer bucle for, además de añadir su correspondiente tag en el compilador.

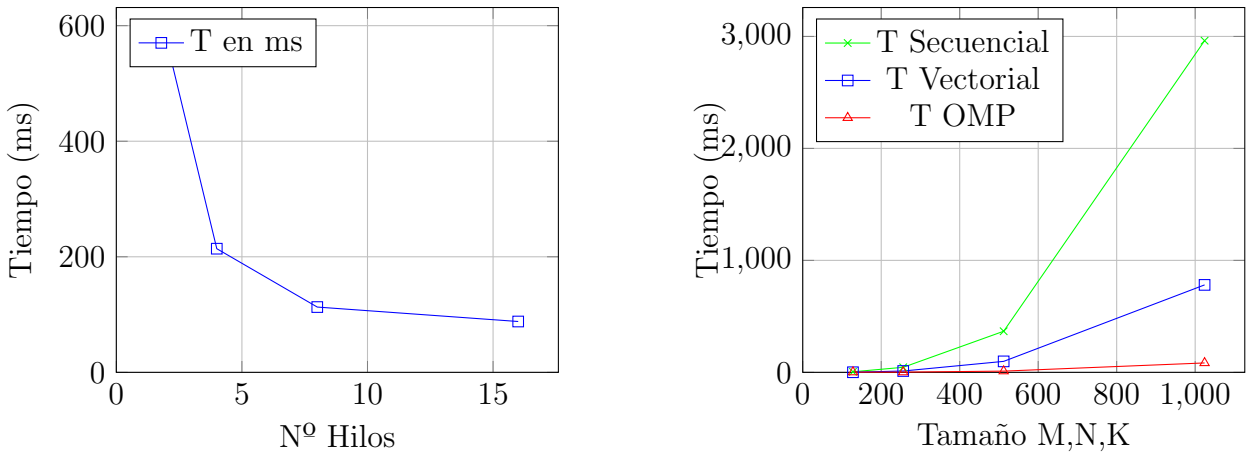

Figura 10: Gráficas de crecimiento del tiempo y operaciones en punto flotante

En la primera gráfica se puede ver como el tiempo ser reduce de manera muy notaria al ir aumentado el número de hilos(matrices cuadradas de 1024x1024), mientras en la segunda gráfica el rendimiento obtenido en un equipo con 16 hilos junto a la vectorización es simplemente abismal obteniendo un speed up de: **35,26x** respecto al secuencial en la multiplicación de dos matrices cuadradas de 1024x1024.

5. Repositorio

El enlace al código se puede encontrar pinchando aquí. Por el momento es privado, por tanto se debe de solicitar acceso como colaborador.

6. Bibliografía

Referencias

[1] J. M. Cardoso, *Single Instruction Multiple Data*, 2017. dirección: https://www.sciencedirect.com/topics/computer-science/single-instruction-multiple-data (visitado 26-02-2023).

[2] E. Klausmeier, *GCC 6.1 Compiler Optimization Level Benchmarks*, 2016. dirección: https://eklausmeier.goip.de/blog/2016/05-30-gcc-6-1-compiler-optimization-level-benchmarks/ (visitado 26-02-2023).

[3] A. Herd, *Performance differences when using AVX instructions*, 2018. dirección: https://albertherd.com/2018/01/11/performance-differences-when-using-avx-instructions/ (visitado 26-02-2023).

[4] M. Campbell, *RPCS3 Developer explains why AVX-512 is important for PS3 emulation and more*, 2022. dirección: https://overclock3d.net/news/software/rpcs3_developer_explains_why_avx-512_is_important_for_ps3_emulation_and_more/1 (visitado 26-02-2023).

[5] A. Avendaño, *Matriz: Multiplicación*, 2010. dirección: https://www.geogebra.org/m/S6R8A2xD (visitado 25-02-2023).

[6] wikichip, *wikichip*, 2023. dirección: https://en.wikichip.org/wiki/WikiChip (visitado 02-03-2023).

[7] Intel, *Intrinsics*, 2023. dirección: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html (visitado 04-03-2023).