Programación Multinúcleo y extensiones SIMD

Adrián Martínez Balea, Pablo Liste Cancela

Arquitectura de Computadores
Grupo 02
{adrian.martinez.balea, pablo.liste}@rai.usc.es

***Resumen*-- En este informe se realiza un análisis de las principales técnicas de optimización de programas en C. Para eso, se emplearán diferentes grados de optimización (utilización de varios núcleos, utilización de extensiones vectoriales SIMD, estrategias para reducir los fallos caché, etc.) sobre un programa base. Tras realizar esto, se medirán tanto los tiempos de ejecución como los ciclos consumidos por cada uno de los programas con optimizaciones diferentes, para poder comparar sus técnicas empleadas y poder concluir cuál de estas técnicas consigue obtener el programa más eficiente.**

***Palabras clave*—optimización, paralelismo, caché, SIMD, AVX, OpenMP.**

I. Introducción

En el mundo de la tecnología existente a día de hoy, la optimización de programas se ha convertido en una necesidad crítica. En la actualidad, los usuarios esperan que los programas se ejecuten de manera casi instantánea, sin preocuparse por la complejidad del problema a resolver y el tiempo que conlleva ejecutar dicha tarea. Con esta demanda de aplicaciones de software más rápidas y eficientes, los desarrolladores de software y programadores se ven obligados a buscar formas de aumentar la eficiencia y rendimiento de sus códigos.

La optimización de programas consiste en modificar el código fuente de un programa con el fin de mejorar su velocidad de ejecución y su capacidad de respuesta pero, a su vez, reducir la utilización de ciertos recursos.

En este informe, se estudiarán diferentes técnicas de optimización de programas. Algunas de estas técnicas serán aplicadas a un programa concreto, en función de si mejoran la eficiencia del programa o no, con el fin de obtener el menor tiempo de ejecución posible.

Para empezar, se partirá de un programa base sin ningún tipo de optimización. A continuación, se aplicarán diversas técnicas de optimización basadas en la memoria caché al código, tales como el uso eficiente de registros para el almacenamiento temporal de datos, el desenrollamiento de bucles o la ejecución de operaciones por bloques, con el fin de lograr un tiempo de ejecución más corto para el mismo programa. Pero eso no es todo, ya que sobre este código ya optimizado, se aplicarán técnicas aún más avanzadas de optimización para programación paralela, tanto a

nivel de datos (usando procesamiento vectorial SIMD), como en memoria compartida (empleando OpenMP). Por último, todos los resultados de los tiempos de ejecución serán evaluados para hacer una comparación detallada de estas técnicas y llegar a una conclusión sólida.

No obstante, antes de comenzar con este estudio, se analizarán detalladamente las características del procesador utilizado y la metodología de trabajo que se ha llevado a cabo, lo que proporcionará un mayor contexto para el análisis de optimización de programas.

II. Entorno de trabajo

Para llevar a cabo este estudio, los programas empleados serán ejecutados en un nodo del FinisTerrae III.

El FinisTerrae III es un supercomputador con altas prestaciones que se encuentra en el CESGA [1]. Cuenta con 708 procesadores Intel Xeon Ice Lake 8352Y de 32 cores, 22.656 núcleos de proceso, 144 aceleradores matemáticos GPU Nvidia A100 y Nvidia T4, logrando una potencia de cómputo de 4 PETAFlops.

Es importante tener esto en cuenta ya que, con este supercomputador, los tiempos de ejecución obtenidos serán mucho menores que en el caso de ejecutar los mismos programas en un ordenador de propósito general.

Otro factor a tener en cuenta es la forma de compilar los códigos. Todos los códigos se compilan con GCC [2], pero GCC tiene diferentes opciones para compilar el programa en lo referido a las optimizaciones que realiza el compilador.

La opción -O0 es la opción por defecto del compilador G. Esta consiste en indicarle al compilador que no haga ningún tipo de optimización.

Optimizar la compilación lleva algo más de tiempo y mucha más memoria para una función grande. Con -O1, el compilador intenta reducir el tamaño del código y el tiempo de ejecución, sin realizar optimizaciones que consumen mucho tiempo de compilación.

La opción -O2 optimiza aún más. GCC realiza casi todas las optimizaciones admitidas que no implican una compensación de velocidad espacial. El compilador no desenrolla bucles ni inserta funciones cuando especifica -O2. En comparación con -O1, esta opción aumenta tanto el tiempo de compilación como el rendimiento del código generado.

Por último, se encuentra la opción -O3, la cual realiza todas las optimizaciones que hace -O2 y otras a mayores, como desenrollamiento de bucles, renombrado de registros...

Una vez especificado esto, se pasará a analizar los códigos empleados.

III. Código base

El código base consiste en lo siguiente. En primer lugar, se inicializarán y se reservarán las variables: las matrices A, B, D; los vectores C, E, el vector de índices y la variable f. Las matrices A, B y el vector C tendrán valores aleatorios de tipo double. La matriz D se inicializará con valores de 0 y, el vector de índices tendrá valores de 0 hasta N-1, de forma desordenada.

Tras esto, se pasará a la parte de computación. La computación de la matriz D consistirá en realizar operaciones entre valores de las matrices A,B y el vector C (accediendo a los valores directamente), y almacenarlos en esta matriz D.

A continuación, se guardarán en el vector E la mitad de los valores de la diagonal de D, calculada previamente, accediendo a estos valores de forma indirecta a través del vector de índices desordenado. Finalmente, todos estos valores del vector E se sumarán en la variable f y se imprimirá este valor.

En la Figura I se muestra una gráfica en la que se representan los ciclos totales (contando la inicialización de las variables) y los ciclos de computación (sin contar la inicialización de las variables). Con esto, se pueden apreciar dos cosas, ambas bastante evidentes y esperables: el número de ciclos totales es ligeramente superior al número de ciclos de computación y que los ciclos, bien sean totales o de computación, aumentan a medida que el tamaño de las matrices y vectores aumenta, debido a que hay más números que computar.

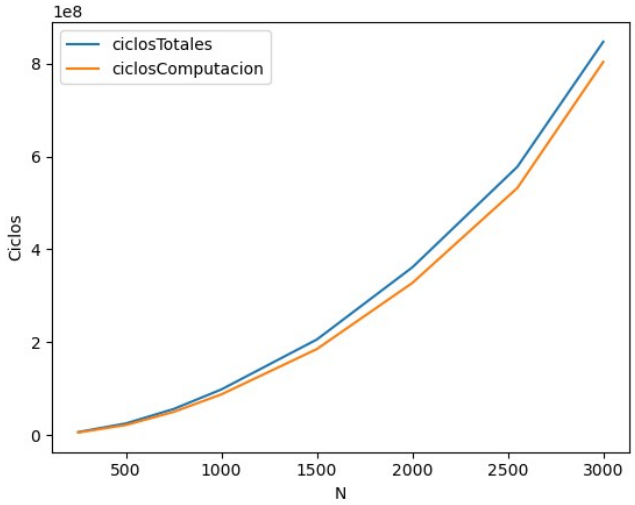

Figura I. Ciclos totales y de computación para diferentes valores de N para el ejercicio 1 (compilado con -O0).

Otra forma de medir esto es mediante el tiempo de ejecución, en lugar de mediante los ciclos. Para los tiempos de ejecución también ocurre algo muy similar, como se puede apreciar en la Figura II.

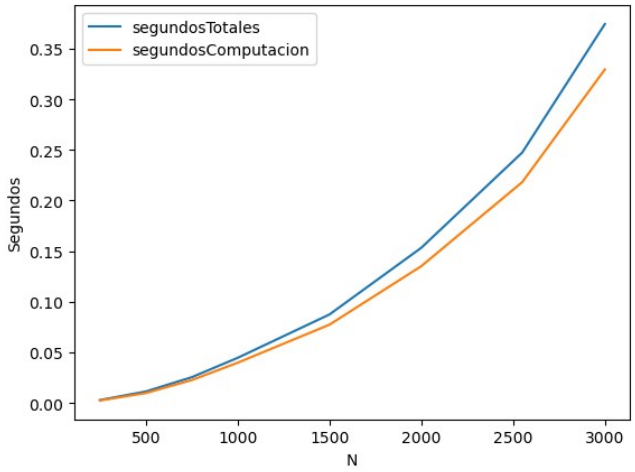

Figura II. Segundos totales y de computación para diferentes valores de N para el ejercicio 1 (compilado con -O0).

Debido a que el hecho de que los segundos y los ciclos siguen el mismo patrón en todos los ejercicios, no se comentará más con ambas formas de medición, con el fin de no hacerlo repetitivo, por lo que a partir de ahora se realizarán las comparaciones midiendo únicamente el número de ciclos.

Lo que sí es resaltable y merece la pena comentar es la diferencia de resultados modificando la opción de compilación en cuánto al nivel de optimización aplicado para este mismo ejercicio. Recordar que, como se comentó previamente, cuánto menor sea el valor que acompaña a la opción -O, menor será el nivel de optimización.

En la Figura III se ve representado esto y es que, las ejecuciones con un mayor nivel de optimización, se ejecutan en un menor número de ciclos. Esto recalca la importancia de estas optimizaciones ya que, por ejemplo, para el mismo valor de N = 500, el programa sin optimizaciones (-O0) requiere de 4510580156 ciclos, mientras que el que más optimizaciones tiene (-O3), tan sólo de 311217674 (14.5 veces menos, aproximadamente).

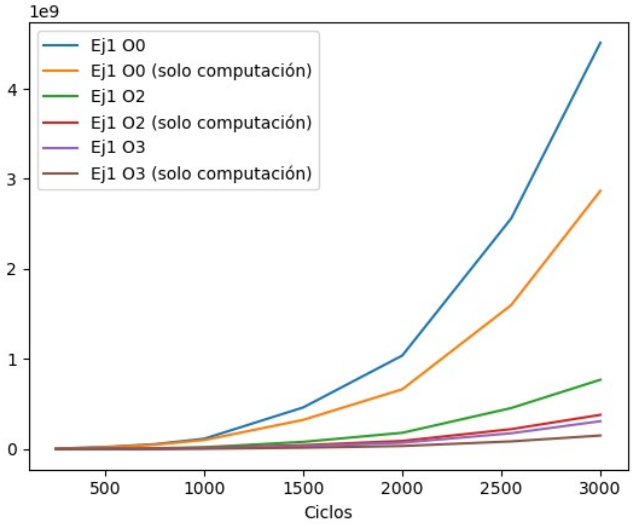

Figura III. Comparación del número de ciclos para diferentes niveles de optimización del ejercicio 1

IV. Optimizaciones basadas en el uso de caché

Una vez valorados los resultados del ejercicio 1, es decir, del código sin optimizaciones, se le aplicarán ciertas modificaciones y basadas en el uso de la memoria caché.

La primera de ellas es el uso eficiente de registros para almacenamiento temporal de datos. Esta es una técnica muy simple, y se ve implementada en el código mediante el uso de la variable “sumaTemporal”, la cual es empleada en el bucle de computación de la matriz D, y después se reutiliza también para calcular el resultado final de f. Además, se elimina el vector E, ya que no tiene ningún uso especial, pues una operación se almacena en un elemento de este vector para que justo después se le sume a otra variable, en lugar de sumar directamente la operación a la variable.

Otra técnica de optimización basada en el uso de la memoria caché es la reducción del número total de instrucciones a ejecutar. Un ejemplo de esta técnica es la omisión de la inicialización de la matriz D con valores de 0, pues estos valores más adelante serán sobreescritos y no habrán tenido ningún tipo de influencia en el programa.

La siguiente técnica de optimización implementada en el código es la fusión de bucles. Para ello, lo que se ha hecho es aprovechar ciertos lazos que iteraban el mismo número de veces, y se han juntado en un único lazo. De esta forma, se mejora la localidad temporal. Hay que tener cuidado con este tipo de fusiones, ya que puede reducir la localidad espacial pero, a pesar de eso, debido a que mejora la latencia del programa, se ha decidido de aplicar al código, por ejemplo entre los bucles de inicialización de la matriz A y el vector de índices.

A continuación, se tratará la técnica de desenrollamiento de lazos. Esta técnica consiste en que el programador repita la instrucción que se repite en el bucle manualmente, para minimizar así la predicción de saltos. Esta técnica es aplicada a todos los lazos que aparecen a lo largo del código.

La última de las optimizaciones implementadas es la realización de operaciones por bloques. Esta es la técnica más compleja de las implementadas anteriormente, y consiste en subdividir las filas o columnas en bloques, para reusar datos antes de que el bloque sea reemplazado en la caché. Para esto, se ha decidido un tamaño de bloque de 8. Este valor no es un valor aleatorio cualquiera, sino que se ha decidido por el hecho de que es el número de elementos de tipo double que caben en una línea caché de la CPU que ejecuta el código.

Con todas estas optimizaciones, se obtiene un número de ciclos mucho menor con respecto al código base. Esta comparación se puede ver en la Figura IV. Además, destaca la casi inexistente diferencia entre los ciclos totales y los ciclos de computación de este segundo apartado. En la gráfica no se puede apreciar a simple vista, pero la línea verde de ciclos totales se encuentra muy ligeramente por encima de los ciclos de computación, tanto que se encuentran casi superpuestas.

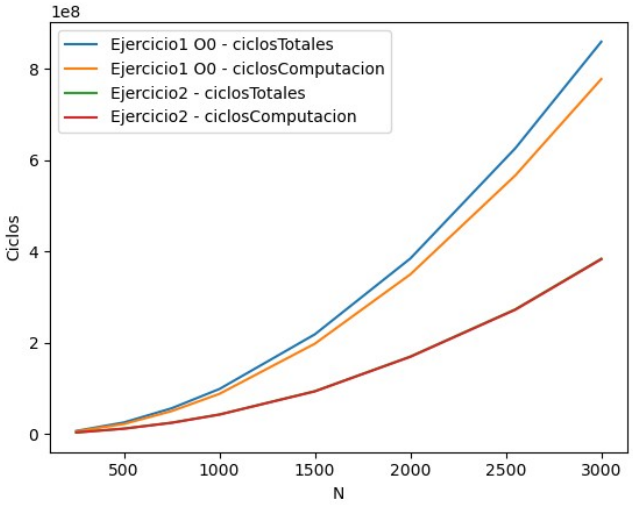

Figura IV. Comparación de los ciclos del ejercicio 2 con los ciclos del ejercicio 1.

V. Paralelismo a nivel de datos

Con el fin de obtener un paralelismo a nivel de datos para optimizar todavía más el programa, se han empleado instrucciones SIMD.

Para esto, se ha utilizado AVX512 (Advanced Vector eXtensions) [3]. AVX-512 es un conjunto de instrucciones vectoriales de 512 bits que se emplean en muchos sectores, como pueden ser la conversión y procesamiento de imágenes y videos, criptografía, computación, modelado o inteligencia artificial y aprendizaje automático. Estas instrucciones permiten aplicar la misma instrucción (aritméticas, lógicas...) sobre diferentes datos.

En cuanto a los tipos de dato, AVX tenía el tipo de dato __m256d para almacenar doubles pero AVX 512 añadió sobre éste, el tipo de dato __m512d. La diferencia entre estos dos tipos de datos, como se aprecia el nombre, es la longitud del vector: el primero es de 256 bits y, el segundo, 512. Un vector de 256 bits puede contener 4 elementos tipo double (8 bytes, 64 bits), mientras que un vector de 512 bits puede contener 8 elementos de tipo double. Por lo tanto, el tipo de datos __m512d puede procesar más elementos de datos en paralelo que el tipo de datos __m256d.

En lo referido a la implementación de estas instrucciones en el código, se utiliza la función reduce(), la cual recibe un vector SIMD de 512 bits y calcula la suma de todos los elementos del vector (de 8 doubles). En primer lugar, divide el vector en otros dos vectores de 256 bits, para posteriormente usar la instrucción _mm256_hadd_pd() para sumar los elementos en cada uno de los vectores. Finalmente, suma los cuatro elementos resultantes para obtener el resultado final de la suma de los 8 elementos dobles en el vector original.

Esta función es aplicada a la matriz D, que se calcula también utilizando funciones propias de AVX 512, como _mm512_mul_pd, _mm512_load_pd, _mm512_sub_pd y _mm512_set_pd, que multiplican elemento a elemento dos registros de 512 bits, cargan 8 doubles de memoria y los almacena en un registro de 512 bits, restan elemento a elemento dos registros de 512 bits e inicializan un registro de 512 bits con 8 doubles dados como argumentos, respectivamente.

Para el proceso de carga de los valores de B, se realizan de esa manera por ser la más rápida que hemos encontrado insertar los valores en un vector. También se probó a cargar como vectores las distintas partes y realizar las operaciones para obtener el valor de B mediante vectores, pero todas las soluciones encontradas eran más lentas.

Además, para las sumas finales de la variable "sumaTemporal" también se emplearon operaciones vectoriales. De la misma forma que antes, se probó a realizar las operaciones para obtener los valores de D mediante operaciones vectoriales, pero las mejoras en cuánto a tiempo eran nulas.

Una vez implementado todo esto, es momento de ver, analizar y comparar los resultados. En la Figura V se puede apreciar como claramente son peores resultados que los del ejercicio 2, ejercicio del que se ha partido para implementar estas operaciones vectoriales. Esto se debe a que el código del ejercicio 2 cuenta con desenrollamiento de bucles y reducciones del uso de variables, que permite al compilador reordenar y mejorar de mucha mejor manera. De todos modos, se obtienen mejores resultados que sobre el ejercicio 1 sin optimizar.

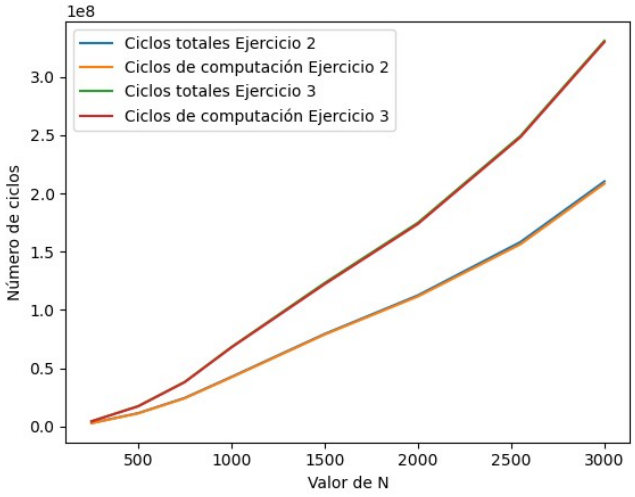

Figura V. Comparación de los resultados del ejercicio 2 y del ejercicio 3.

VI. Programación paralela en memoria compartida

Para este apartado se ha empleado programación paralela en memoria compartida mediante OpenMP [4]. Con OpenMP, se pueden hacer programas que se dividan en múltiples hilos de ejecución, cada uno de los cuales puede ser ejecutado simultáneamente en un núcleo de procesador diferente. Esto permite una mejor utilización del procesador y una aceleración del tiempo de ejecución para programas que se benefician de la ejecución en paralelo.

Su implementación es mucho más simple que AVX 512, ya que basta con escribir un par de sentencias antes de las instrucciones que se quieran paralelizar. Para paralelizar un bucle for, que es el caso en donde se utiliza esto en el código, se emplea #pragma omp for.

Esto se utiliza en el cálculo de los valores de D, mediante las líneas "#pragma omp parallel private(sumaTemporal, temporalGeneral1, temporalGeneral2) num_threads(k)" y "#pragma omp for collapse(1) schedule(static)". Con la primera de estas líneas, se crea un bloque paralelo en el que se ejecutará un conjunto de hilos. Este número de

hilos viene determinado por k, que se pasa como parámetro al programa. Además, también se especifica que las variables “sumaTemporal”, “temporalGeneral1” y “temporalGeneral2” son privadas para cada hilo. Esto es importante ya que por defecto ocurriría el caso contrario, es decir, que esas variables fuesen compartidas por todos los hilos, lo que daría lugar a carreras críticas y errores en los resultados.

En cuánto a la segunda línea, las cláusulas de collapse(1) y schedule(static) no fueron elegidas aleatoriamente, sino que fueron elegidas tras hacer un pequeño análisis de cuál era más eficiente para este programa en concreto. Este pequeño análisis se puede leer a continuación.

Collapse(1) indica que no se debe anidar el bucle. El parámetro que recibe es el número de bucles que se quieren fusionar. Como se comprobará más adelante, lo más eficiente es emplear collapse(1), pues así se consigue reducir la sobrecarga del sistema asociada con la gestión de múltiples bucles y, como el bucle y la cantidad de datos no son excesivamente grandes, se obtienen tiempos más óptimos si no se anidan los bucles.

La cláusula chedule(static) especifica el modo de asignación de trabajo a los hilos. Esta asignación divide el trabajo en bloques del mismo tamaño y se asigna cada bloque a un hilo diferente.

Por otro lado, el planificador dynamic, divide el número total de iteraciones en pequeños grupos. Cada hilo coge uno de este pequeño conjunto de iteraciones y, cuando termina su trabajo, se le asigna otro conjunto de iteraciones; hasta que se completan todas las iteraciones. Esto permite una distribución más equilibrada del trabajo entre los hilos, ya que las iteraciones más largas se asignan a hilos más rápidos y las iteraciones más cortas a hilos más lentos.

Como el coste adicional de asignar iteraciones dinámicamente puede disminuir la eficiencia en comparación con la asignación estática, y en ese caso las iteraciones son muy similares y no va a variar demasiado, se ha optado por utilizar el planificador static.

Una vez explicado esto, se revisará cómo afecta el número de hilos (que se pasa como parámetro al programa), el scheduling y el collapse al número de ciclos del programa. Para ello, se compararán 4 casos diferentes: scheduling static con collapse(1), scheduling dynamic con collapse(1), scheduling static con collapse(2) y scheduling dynamic con collapse(2); todo esto para diferente número de hilos: 1, 2, 4, 8, 16, 32, 64. Además, con el fin de no añadir más complejidad a la gráfica y no hacerla más difícil de interpretar, se ha ejecutado con el valor de N más grande, 3000, para que, en caso de que haya diferencias, estas se resaltasen más. Esta comparativa se puede observar en las figuras de a continuación.

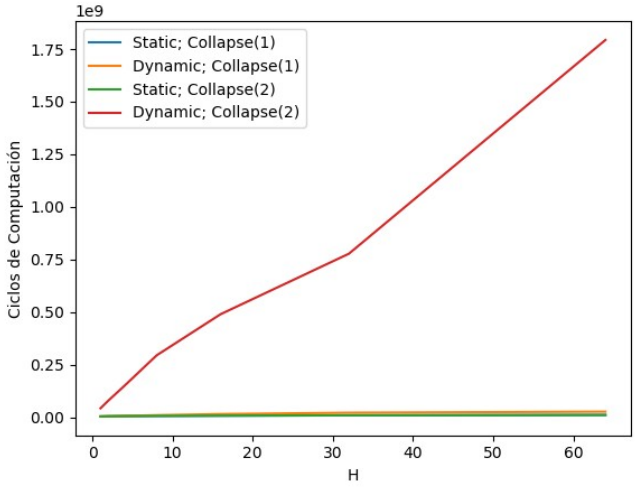

Figura VI. Comparación del scheduling y el collapse para N=3000 y diferente número de hilos.

En la Figura VI, se puede apreciar cómo, por una parte la opción de sheduling dynamic junto con collapse(2) tiene unos resultados pésimos. Por otra parte, las otras tres opciones están muy próximas, por lo que para determinar cuál es la mejor, se muestra la Figura VII, que no es más que la simple ampliación de la Figura VI, centrándose en las tres mejores opciones.

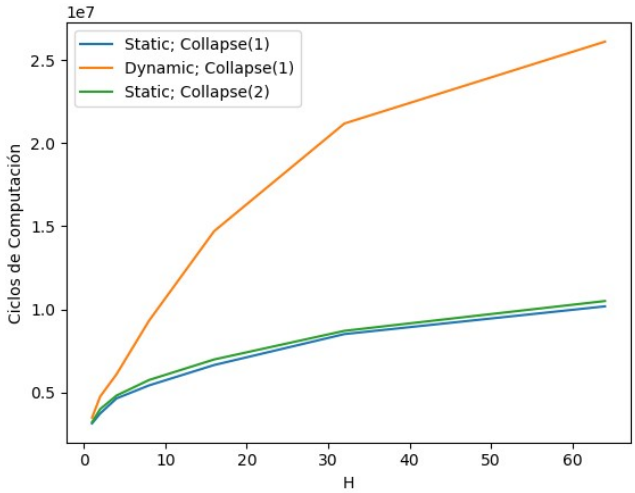

Figura VII. Ampliación de la comparación del scheduling y el collapse para N=3000 y diferente número de hilos.

Esta figura confirma lo comentado previamente: la opción más óptima para implementar esta programación paralela es con la sentencia “#pragma omp for collapse(1) schedule(static)”.

Una vez visto esto, se pasará a analizar este programa con el scheduling y el collapse previamente seleccionados, en función de los valores de N, y del nivel de optimización que se le indica al compilador. Además, se compararán con el programa del ejercicio 2, del que se ha partido a la hora de implementar OpenMP. Los resultados de esto aparecen en la Figura VIII. En ella, se puede ver que

ocurre lo mismo que en el caso del ejercicio 1: cuánto más valor le demos al nivel de optimización en la opción de GCC, menor número de ciclos se van a obtener, lo que implica programas más eficientes. Además, se puede ver como la programación paralela en memoria compartida es mucho más eficiente y consume un menor número de ciclos que el ejercicio 2.

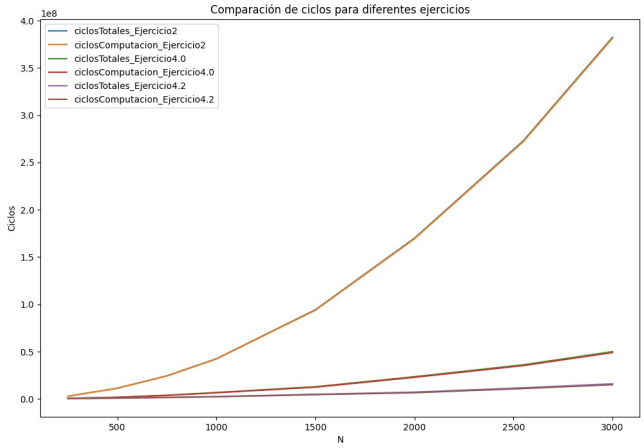

Figura VIII. Ciclos del ejercicio 2 y del ejercicio 4 con diferentes tipos de optimizaciones; y en función de N.

VII. Resultados generales

Con el fin de recopilar todos los datos de todos los tipos de optimizaciones, se han resumido los principales en la Figura IX. En ella, se muestran los ciclos totales de diferentes programas con diferentes formas de ser compilados: el programa base (ejercicio 1) sin ningún tipo de optimización, el programa base con el máximo de optimizaciones (-O3), el programa con optimizaciones basadas en el uso de memoria caché (ejercicio 2), el programa con paralelismo a nivel de datos (ejercicio 3) y, por último, el programa con programación paralela en memoria compartida, con dos niveles de optimización diferentes, -O0 y -O2.

Como se puede apreciar en esta gráfica, el programa base es el que más ciclos consume, como era de esperar. A este le siguen las operaciones vectoriales mediante AVX 512 y las optimizaciones basadas en uso de caché. Tras estas tres, se produce un gran salto en cuánto al número de ciclos, ya que se reduce en gran cantidad, comenzando por el programa base indicando al compilador que haga un gran nivel de optimizaciones. Por último, las dos optimizaciones más eficientes son las de programación paralela en memoria compartida con OpenMP, por supuesto la de un nivel de optimización superior con menos número de ciclos.

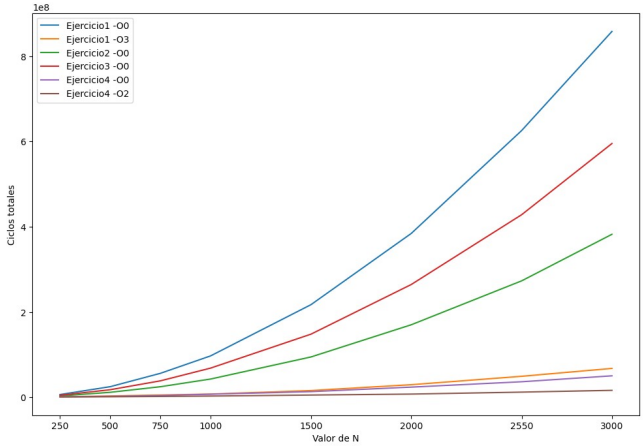

Figura IX. Resumen del costo de ciclos de los tipos de optimizaciones principales tratados en este informe.

VIII. Conclusiones

En resumen, se han realizado una serie de pruebas y experimentos en un nodo del FinisTerrae III del CESGA con el fin de valorar y analizar en profundidad qué tipo de optimizaciones son las más óptimas.

Para ello, se ha partido de un programa base sin ningún tipo de optimización y se le han ido aplicando diferentes técnicas de optimización, además de analizar también las opciones con las que cuenta GCC a la hora de optimizar.

Finalmente, estos experimentos subrayan la importancia de eficiencia en un programa, a la que se puede llegar aplicando las técnicas aplicadas en este informe. Sobre todas, destaca la programación paralela en memoria compartida con OpenMP.

Referencias

[1] CESGA, https://es.wikipedia.org/wiki/Centro_de_Supercomputaci%C3%B3n_de_Galicia, (última visita 4 de mayo de 2023).

[2] GCC, the GNU Compiler Collection, https://gcc.gnu.org/, (última visita 4 de mayo de 2023).

[3] Intel® Advanced Vector Extensions 512, https://www.intel.es/content/www/es/es/architecture-and-technology/avx-512-overview.html, (última visita 4 de mayo de 2023).

[4] OpenMP, https://www.openmp.org/, (última visita 4 de mayo de 2023).