Generación y optimización de código

Fusión de bucles

Compiladores e Intérpretes - GREI USC

Índice

Resumen

En este informe se lleva a cabo un análisis de la técnica de fusión de bucles mediante la aplicación y testeo sobre un caso práctico. Primeramente se presenta la técnica y sus características. A continuación se lleva a cabo un análisis del código ensamblador, que permite reafirmar algunas de las características previamente atribuidas. Se describen las pruebas a realizar y su entorno y se presentan los resultados y sus conclusiones, los cuales determinan un ligero mejor rendimiento para la versión optimizada que se incrementa con el aumento de tamaño de los arrays de prueba.

Introducción

En este documento se llevará a cabo un análisis de la técnica de fusión de bucles para el código del Ejercicio 1 de la práctica. Para ello se realizarán una serie de pruebas, que tratarán de ser lo más fiables posibles y arrojar datos útiles sobre el rendimiento de la optimización aplicada.

Para dicho análisis se comenzará por un apartado de presentación de la técnica, en el que se describirá en qué consiste y sus posibles ventajas y desventajas. A continuación, se describirá el hardware del equipo, para el cual se generará código ensamblador, cuyo análisis formará el siguiente apartado. Seguido de dicho análisis se presentará el entorno y se describirán las pruebas a realizar. Los resultados de dichas pruebas se presentarán de forma gráfica y se analizarán en el siguiente apartado. Por último, se elaborarán conclusiones del trabajo realizado.

Técnica

En primer lugar se presentará la técnica utilizada para optimizar el código. En este caso se trata de una técnica basada en el tratamiento de bucles conocida como fusión de bucles o loop jamming. La técnica consiste en la unión de varios bucles con las mismas operaciones de control en un único bucle. Inicialmente se dispone de un bucle para las iteraciones relativas a reducir la discordancia en las medidas con 3 bucles en su interior visible en la Figura 1:

```
for (k = 0; k < ITER; k++)
{
    for (i = 0; i < N; i++)
        x[i] = sqrt((float)i);
    for (i = 0; i < N; i++)
        y[i] = (float)i + 2.0;
    for (i = 0; i < N; i++)
        x[i] += sqrt(y[i]);
}
```

Figura 1: Código sin optimizar.

La aplicación de la técnica da lugar a un único bucle interno, en el cual se realizan las 3 operaciones anteriormente separadas en diferentes bucles. El resultado final se puede visualizar en la Figura 2:

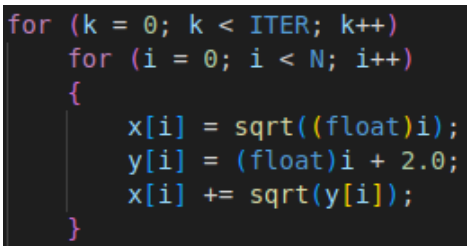

Figura 2: Código optimizado.

Ventajas e inconvenientes

La aplicación de la técnica de loop jamming presenta características que pueden tanto optimizar el rendimiento como empeorarlo dependiendo del caso. Una de las principales optimizaciones que ofrece es la reducción de las instrucciones de control de los bucles, esto es debido a que, una vez fusionados los bucles internos, las comprobaciones y operaciones necesarias para llevar a cabo las iteraciones de cada bucle (comprobación del valor de i e incremento de i) se reducirían un tercio, por pasar a ser el bucle compartido por las tres operaciones.

Otra de sus principales optimizaciones se basa en el uso de los registros. Como se puede observar en la Figura 2, la tercera operación depende del valor de y[i], el cual ha sido calculado en la instrucción anterior. Dicha dependencia en instrucciones secuenciales es fácilmente explotable a la hora de acceder al valor de y[i], ya que su reciente cálculo hará que con casi total seguridad este dato se encuentre todavía en el registro utilizado para el cálculo. Este hecho no ocurriría en el caso de no aplicarse la optimización, ya que se calcularían en primer lugar todo los valores de y, de forma que cuando se llegue a la última operación resulta improbable que el dato se halle todavía en un registro, por lo que la necesidad de accesos a memoria sería mayor en este caso, ralentizando el programa.

No todo son ventajas en la optimización realizada, y es que las ganancias obtenidas descritas previamente se podrían ver minimizadas por una reducción de la localización de la localidad espacial. Entendida esta como la tendencia del procesador a acceder a conjuntos de posiciones de memoria próximas y de forma repetida. Dicha tendencia acostumbra ser explotada por la jerarquía de memoria de los ordenadores de forma que en los accesos a memoria se traen páginas (conjuntos de datos) en un intento por actuar de forma predictiva ante las próximas solicitudes del procesador.

En la versión sin optimizar, el hecho de recorrer cada array sin otro tipo de instrucciones intercaladas (como ocurre en la optimizada), hace más sencilla y eficiente la explotación de la localidad espacial en los accesos a los elementos del array de forma consecutiva. Dicha explotación de la localidad es probable que se vea reducida en la versión optimizada en la cual los bucles se han fusionado.

Hardware

El hardware utilizado para la obtención de datos consiste en un ordenador portátil común con unas prestaciones estándares suficientes para el usuario promedio, pero sin alcanzar cuotas de rendimiento de otros modelos de alta gama disponibles en el mercado. A continuación, se presentan de forma más pormenorizada sus características, en concreto aquellas más imprescindibles para la comprensión de los resultados obtenidos. Para ver cuáles son las especificaciones de la máquina se han utilizado los siguientes comandos, cuya descripción detallada es accesible a través del manual de Linux. Se utilizaron:

lscpu - Visualización de las cachés
lstopo – Topología del procesador

```
Arquitectura:                         x86_64
modo(s) de operación de las CPUs:     32-bit, 64-bit
Orden de los bytes:                   Little Endian
Address sizes:                        39 bits physical, 48 bits virtual
CPU(s):                               8
Lista de la(s) CPU(s) en línea:       0-7
Hilo(s) de procesamiento por núcleo:  2
Núcleo(s) por «socket»:               4
«Socket(s)»                           1
Modo(s) NUMA:                         1
ID de fabricante:                     GenuineIntel
Familia de CPU:                       6
Modelo:                               142
Nombre del modelo:                    Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Revisión:                             10
CPU MHz:                              1000.970
CPU MHz máx.:                         4000,0000
CPU MHz mín.:                         400,0000
BogoMIPS:                             3999.93
Virtualización:                       VT-x
Caché L1d:                            128 KiB
Caché L1i:                            128 KiB
Caché L2:                             1 MiB
Caché L3:                             8 MiB
CPU(s) del nodo NUMA 0:               0-7
```

Figura 3: Datos ordenador.

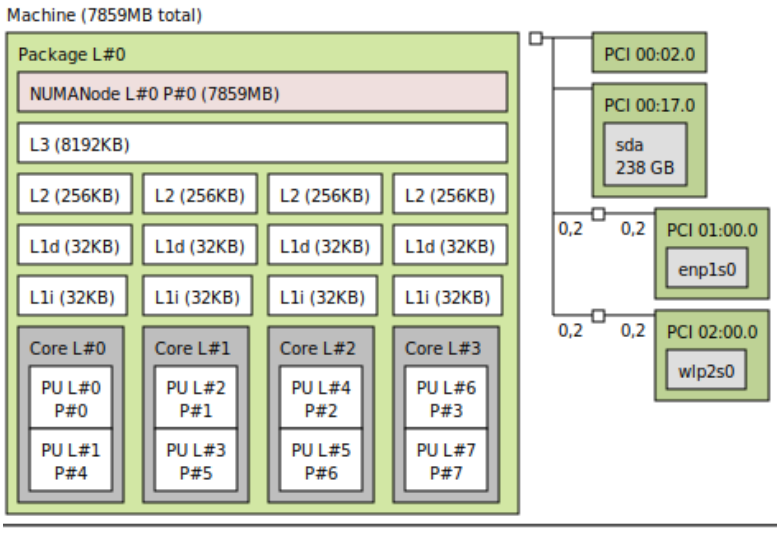

Figura 4: Distribución de memoria.

En la Figura 3 se pueden ver los datos del procesador, el cual es un Intel Core i7-8550U con 4 cores y 8 hilos. La capacidad total de cada uno de los niveles cachés es de 128 KiB en la L1 de datos, 1 MiB en la L2 y 8 MiB en la L3. Para ilustrar como se distribuye esta memoria entre los diferentes cores se utiliza la Figura 4, en ella se puede observar que cada core tiene su propia L1 de datos de 32 KB, su propia L2 de 256 KB y que la L3 es compartida para todos los cores y tiene 8192KB de capacidad.

Análisis ensamblador

En este apartado se presentará el código ensamblador resultante, así como sus características. Para obtener dicho código ensamblador se utilizó la opción de compilación -S, presente en el compilador gcc. Destacar además que se compiló para una arquitectura x86 64 con la versión 9.4 de gcc.

En primer lugar se muestra en la figura 5 el código ensamblador de la versión sin optimizar. En este se puede observar como existen instrucciones de control para cada uno de los bucles, correspondiéndose las resaltadas en color azul con el bucle externo de iteraciones, siendo las demás las correspondientes a los bucles internos que, en la versión sin optimizar son 3. Además se pueden identificar 13 bloques básicos en la parte del código que se desea medir, los cuales se muestran agrupados con una barra verde a su izquierda. Por último, se puede destacar el número de instrucciones generadas en este caso, 65 excluyendo las llamadas a gettimeofday. De esas 65, 22 corresponden con instrucciones de los bucles, las cuales están resaltadas.

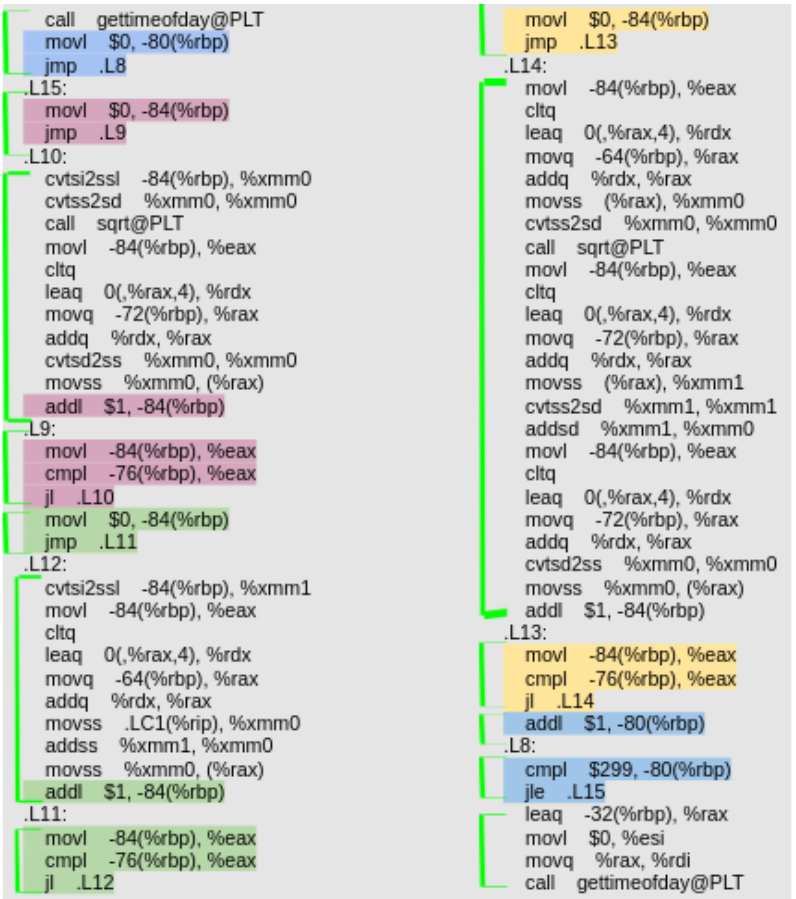

Figura 5: Código ensamblador sin optimizar

En este caso se muestra el código ensamblador resultante del programa optimizado. En este se destacan las instrucciones relativas al bucle de iteraciones (ITER) en color azul, y en color verde las del bucle interno. Además se pueden identificar 7 bloques básicos en el fragmento de código a medir, los cuales se identifican con una barra verde a su izquierda. Por último, destacar que en este caso el número de instrucciones (excluyendo las llamada a gettimeofday) son 56, correspondiéndose 11 de ellas con instrucciones relativas a los bucles for.

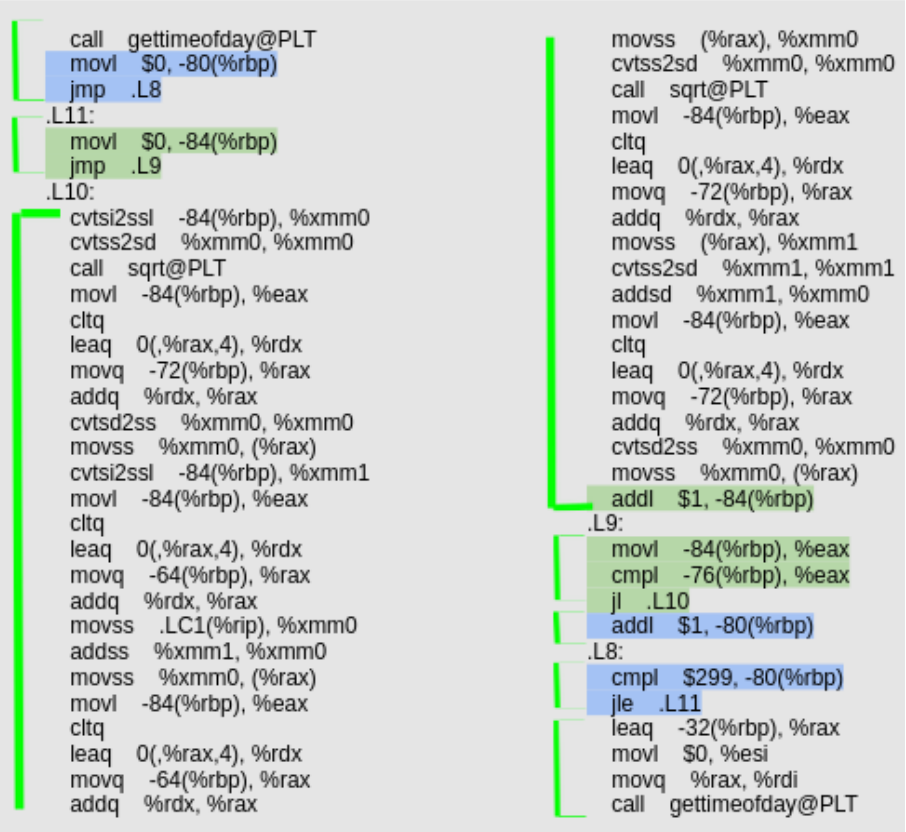

Figura 6: Código ensamblador optimizado

Dado el análisis del código ensamblador llevado a cabo, es posible afirmar que la optimización se ha llevado a cabo con éxito ya que, como se ha visto, los 3 bucles internos se encuentran fusionados en un único bucle, lo cual reduce el número de instrucciones de bucle de 22 en el no optimizado, a 11 en la versión mejorada. Esta no es la única diferencia observable, y es que la eliminación de bucles, y por ende de instrucciones de salto, ha reducido el número de bloques básicos (de 13 a 7), aumentando en consecuencia el tamaño de dichos bloques. Dicho aumento de tamaño se halla principalmente en el bloque relativo al código contenido por el bucle más interno, lo cual puede indicar que la fusión de bucles es una técnica adecuada para lograr optimizaciones, ya que a mayores tamaños de bloque básico más optimizaciones a nivel de bloque podrá realizar el compilador.

Entorno y pruebas

Para la ejecución de las pruebas se tratará de reducir el ruido en el proceso de toma de datos mediante la reducción al mínimo de los procesos activos en el dispositivo, así como de mantener la toma de corriente activa para que el equipo portátil no realice ningún tipo de reducción de rendimiento para ahorrar batería.

En lo referente al proceso de compilación, se utilizará la versión 9.4 de gcc con la opción -O0, lo cual permitirá reducir las optimizaciones realizadas por el compilador, con el objeto de que el código ensamblador resultante sea lo más fiel posible al código fuente sobre el que se aplica la optimización a objeto del análisis de este informe. Cabe destacara además que ambos ejecutables obtenidos tienen el mismo peso, 17.120 bytes.

Cabe puntualizar que el uso de dos ejecutables diferentes hace innecesario el calentamiento de caché, el cúal sí sería preciso de ejecutarse ambos en el mismo código, ya que el primero de ellos sería mucho más penalizado por los fallos caché.

Las pruebas consistirán en la ejecución reiterada del ejecutable en C, el cúal devuelve el tiempo de ejecución medio. De esta forma el código será ejecutado un número de iteraciones igual al producto por el número de iteraciones establecido en el código (ITER), que en este caso será 300, por el número de ejecuciones del ejecutable. El objetivo de realizar diversas ejecuciones a mayores de las ya establecidas en el código es tanto reducir las posibles discordancias como la extracción de otros datos útiles mediante la automatización del proceso con un script de bash. En este caso se extraerán máximos y mínimos.

De esta forma las pruebas se realizarán mediante dicho script, el cual recibe como parámetros los nombres de los ejecutable y el número de veces a ejecutar cada uno para cada valor de N (tamaño de los array). De esta forma el script recorrerá un conjunto de valores de N (1.000, 50.000, 100.000, 150.000, 250.000, 500.000, 750.000 y 1.000.000) ejecutando el código optimizado y sin optimizar por separado almacenando los resultados para el cálculo de media, máximo y mínimo. En este caso se optó por ejecutar un total de 10 veces cada ejecutable, por lo que la invocación del script si se desea repetir las pruebas sería:

```
./script.sh nombre_ejecutable1 nombre_ejecutable2 10
```

Una vez ejecutadas las pruebas se extraen los obtenidos para su procesamiento y posterior presentación en el apartado de resultados.

Resultados

En este apartado se presentarán los resultados obtenidos en un formato visual que facilite su interpretación. En primer lugar, se presentará una gráfica en la que se recogen los tiempos medios de cada código en función del valor de N. El eje vertical se corresponde con

el tiempo en segundos, mientras que en el horizontal se representan los valores de N. La línea azul se corresponde con los resultados de los tiempos medios para el código sin optimizar, mientras que la naranja se corresponderá con los tiempos del código optimizado. En la Figura 7 se puede hallar dicha gráfica.

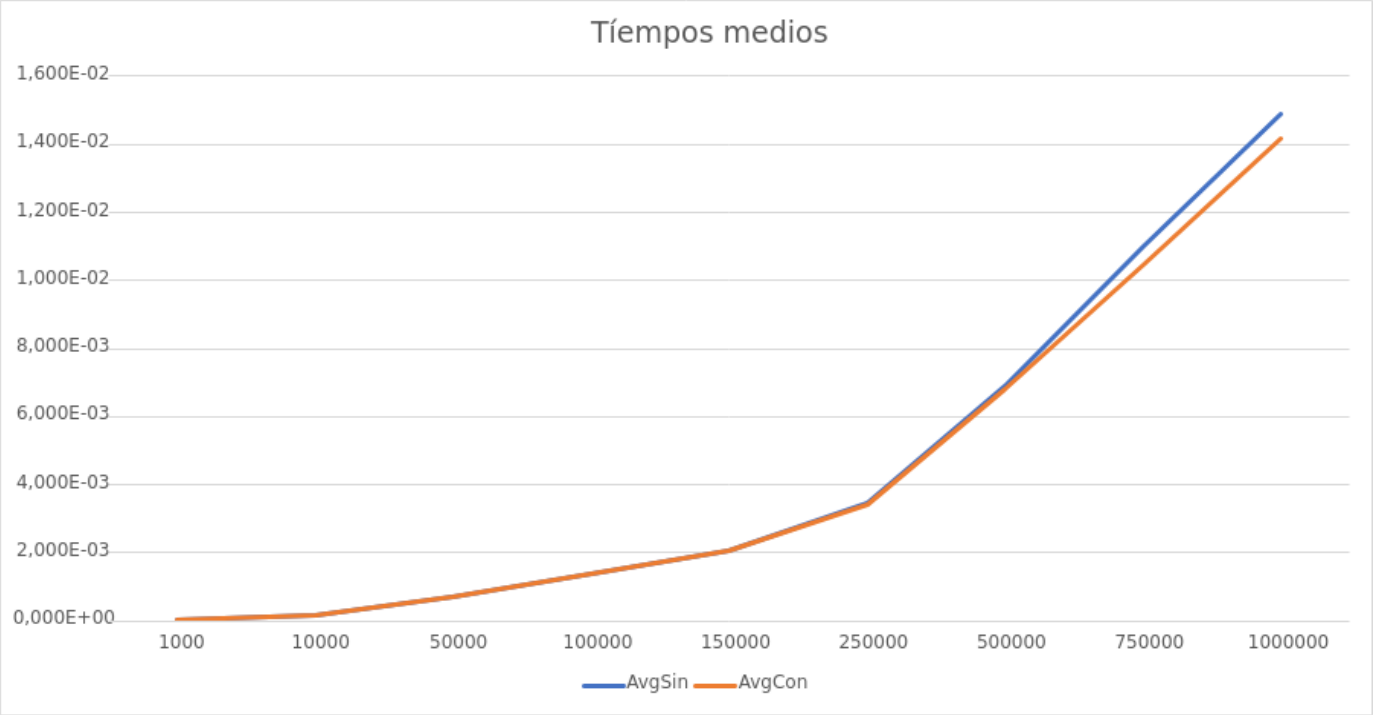

Figura 7: Tiempos medios.

Como se puede observar en la anterior gráfica, los tiempos obtenidos resultan bastante parejos, sin embargo es posible comenzar a apreciar como, a partir de un N mayor que 500.000, las diferencias empiezan a hacerse visibles y el rendimiento del código optimizado se pone de manifiesto, siendo la diferencia entre ambos códigos cada vez mayor.

En la siguiente gráfica se presentan unos datos similares, solo que en esta ocasión se añaden los valores máximos y mínimos obtenidos para las 10 ejecuciones efectuadas desde el script (descripción detallada en el apartado de Entorno y Pruebas). Los colores fríos (azules y verde) se corresponden con las medidas del código sin optimizar mientras que los cálidos (rojo, amarillo y naranja) se corresponden con las del código optimizado. En la Figura 8 se muestra dicha gráfica.

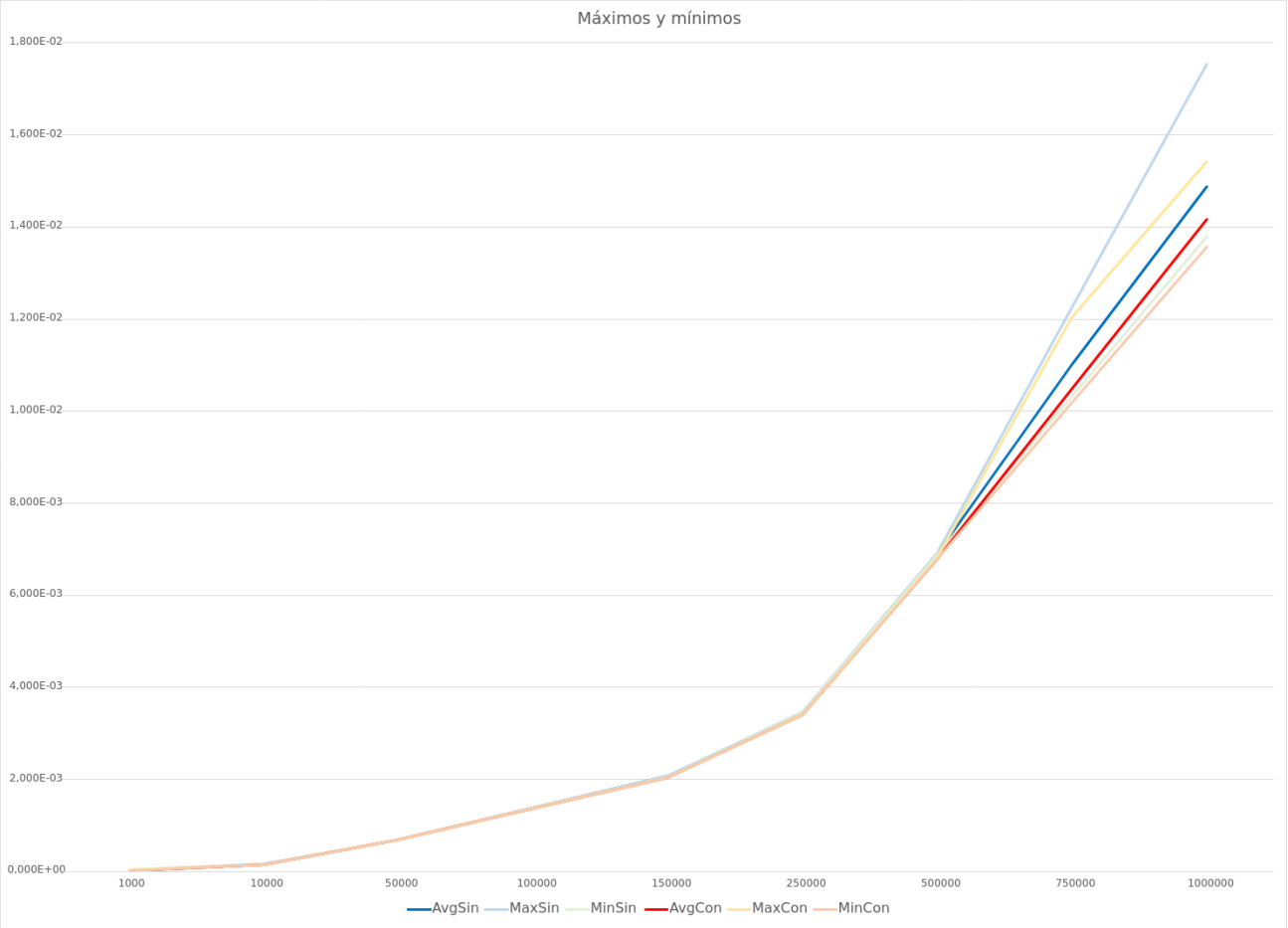

Figura 8: Máximos y mínimos.

En esta gráfica se puede vislumbrar, además de la tendencia ya observada en la anterior, una menor dispersión en los tiempos del código optimizado. Se observa que, mientras que los máximos del código sin optimizar se encuentran cada vez más lejanos a la media, los del código optimizado permanecen cercanos al valor medio. En los mínimos tiene lugar un comportamiento similar pero en menor medida. La presencia de dicho distanciamiento en los máximos del código sin optimizar, deja entrever una menor fiabilidad en cuanto al rendimiento temporal del código, pudiendo darse ejecuciones considerablemente más lentas de lo común.

Una vez presentadas estas gráficas, se puede evaluar también en qué medida el código optimizado mejora al original. Para ello se utilizará el speed-up, que se calculará como el cociente entre el tiempo empleado por el código original y el optimizado. De esta forma un valor mayor que uno supondría que el código optimizado mejora al original, igual a uno que mantiene el rendimiento y menor que 1, lo empeora. El resultado de dicho cálculo se muestra en la Figura 9.

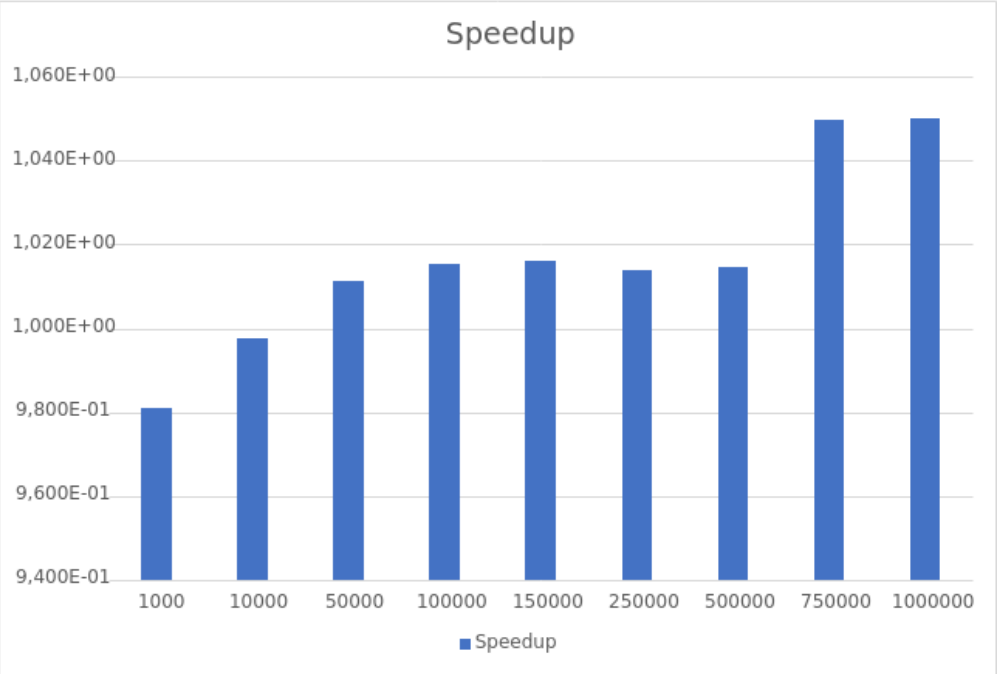

Figura 9: Speed-up.

La aplicación del speed-up permite vislumbrar un resultado que había sido pasado por alto hasta el momento debido a las escasas diferencias entre ambos tiempos que lo hacían pasar inadvertido en las anteriores gráficas. Este es el hecho de que para valores pequeños de N el código sin optimizar presenta un rendimiento ligeramente mejor (por ser el speed-up menor que cero), mientras que a partir de N=500.000 comienzan a ser visibles los resultados de la optimización, haciéndose más considerables para valores grandes de N como 750.000 0 1.000.000.

Conclusión

El análisis llevado a cabo hasta el momento comenzó con una presentación de la técnica de fusión de bucles aplicada y sus posibles ventajas y desventajas. Algunas de sus ventajas son el ahorro de instrucciones de bucle y la reutilización del valor calculado en la segunda instrucción del bucle en la tercera, ventajas que permiten dar explicación al comportamiento observado, dado que a mayor número de iteraciones mayor será el número de operaciones asociadas a los bucles internos evitadas, así como el beneficio obtenido por la localidad temporal entre instrucciones.

Así mismo, la desventaja apuntada podría dar explicación a la no mejora para los primeros valores de N, ya que la reducción de la localidad espacial en los accesos a y en el código optimizado podría pesar más las ventajas que se obtienen, menores para valores bajos de N.

El análisis continuaba con la inspección del código ensamblador resultante, el cúal permitía comprobar afirmaciones como la de la reducción de instrucciones para bucles y que, a su vez, abre nuevas vías de estudio hacia un análisis más exhaustivo del código obtenido, en concreto de los accesos a memoria. Dado que el alcance determinado para

este informe delimita en cierta medida el análisis de la interacción con la memoria y el uso de las cachés, podría ser objeto de estudio también el comportamiento de las cachés y los fallos de página de estas para futuras investigaciones.

Por último, se llevaron a cabo las pruebas en un entorno controlado y descrito que pretendió reducir el ruido en los resultados lo máximo posible. Estas mostraron resultados ligeramente mejores para el código optimizado (en concreto para valores grandes de N), pero que distan bastante de ser una optimización de gran calibre que nunca se deba pasar por alto. Otro de los resultados observados era el hecho del menor rendimiento para los valores pequeños de N, los cuales como se mencionaba anteriormente podrían ser derivados de la menor localidad espacial en el acceso ahí, pero este constituye otro posible ámbito de estudio, también muy relacionado con el funcionamiento de las cachés.

En definitiva, la fusión de bloques en casos como este constituye una optimización válida, si bien no con mejoras de gran calibre, pero que puede ser una optimización útil, especialmente para arrays de gran tamaño. Cabría valorar a la hora de realizar optimizaciones a dicho código otras alternativas, como el unrolling de bucles, que tal vez podrían optimizar más el rendimiento.

Bibliografía

[1] Wikipedia contributors. (2022, March 30). Loop fission and fusion. In *Wikipedia, The Free Encyclopedia*. Retrieved 20:36, June 5, 2022, from https://en.wikipedia.org/w/index.php?title=Loop_fission_and_fusion&oldid=1080118181

[2]Hmong (2022, March 30). Fisión y fusión de bucle, from https://hmong.es/wiki/Loop_fusion

[3] Programación en ensamblador (x86-64), Miguel Albert Orenga y Gerard Enrique Manonellas PID_00178132 https://www.exabyteinformatica.com/uoc/Informatica/Estructura_de_computadores/Estructura_de_computadores_(Modulo_6).pdf