Optimización de rendimiento en ejecución de programas multinúcleo y extensiones SIMD

Pablo González López, Fernando González Salas

Arquitectura de Computadores
Grupo II
pablo.gonzalez.lopez@rai.usc.es, fernando.gonzalez.salas@rai.usc.es

***Resumen*—La programación algorítmica se realiza frecuentemente a partir de un pseudocódigo que es transcrito al lenguaje de programación deseado. No obstante, diversos factores influyen en el rendimiento de los programas (más allá de la optimización del algoritmo), como la utilización de diferentes instrucciones, la optimización del código dependiendo del lenguaje o también la utilización de diversos hilos en una misma ejecución. Generalmente el uso de estas técnicas, sumadas a una correcta optimización del programa, permiten un mayor rendimiento a la hora de ser ejecutados. Esta diferencia que existe entre diferentes versiones será estudiada con precisión en este informe.**

***Palabras clave*— Optimización caché, paralelización, OpenMP, instrucciones vectoriales, SIMD.**

I. Introducción

A la hora de ejecutar un programa existen diferentes estrategias que, dependiendo de cuál sea la escogida, variarán el rendimiento del mismo. De esta forma, se ha decidido realizar una experimentación en la ejecución de un código. Este código será el dado por el enunciado de la práctica, y consistirá en un programa en C que compute una serie de matrices y vectores. El pseudocódigo es el siguiente:

```
Entradas:
a[N][8],b[8][N],c[8]: matrices y vector que almacenan valores aleatorios de
tipo double.
Salida:
f: variable de salida tipo double
Computación:
d[N][N]=0; // inicialización de todas las componentes de d a cero;
 for (i=0; i<N; i++) {
    for(j=0; j<N; j++) {
       for (k=0; k<8; k++) {
         d[i][j] += 2 * a[i][k] * ( b[k][j]- c[k]);
       }
     }
ind[N]: vector desordenado aletoriamente que contiene índices de fila/columna
f=0;
for (i=0;i<N;i++){
    e[i]= d[ind[i]][ind[i]]/2;
    f+=e[i];}
Imprimir el valor de f
```

Las pruebas realizadas serán las siguientes: un programa base (traducir a C el pseudocódigo, sin ningún tipo de optimización); un programa similar al anterior, pero optimizado de alguna forma (que se verá posteriormente); un programa optimizado que utilice extensiones SSE3; y un programa que utilice OpenMP (multiproceso).

En los siguientes apartados se detallarán las especificaciones del computador utilizado para realizar los experimentos (Sección II) y las diferentes pruebas que se realizarán (Sección III). Además, se expondrán los resultados (Sección IV) y se explicarán las conclusiones obtenidas de la comparación de resultados (Sección V).

II. ESPECIFICACIONES COMPUTADOR

Las características del computador utilizado para el experimento son importantes a la hora de extraer los resultados, ya que pueden dar explicación a algún fenómeno que ocurra. De esta forma, las especificaciones más importantes del ordenador empleado son las siguientes:

- Frecuencia procesador: 1990 MHz

Nivel de caché	Tamaño	N.º de Líneas
L1	32 KB	512
L2	256 KB	4096
L3	8192 KB	131072

Un punto a destacar a la hora de realizar la ejecución de los códigos es la importancia de evitar al máximo posible los procesos en segundo plano, ya que éstos pueden variar considerablemente los tiempos de ejecución del programa. De todas formas, en la realización de las pruebas los datos que se han obtenido son resultado de diversas pruebas, a las que se ha hallado una media. De esta forma evitamos al máximo posible cualquier error en la medición causado por estas variaciones.

III. VERSIONES DEL CÓDIGO

Como se ha mencionado anteriormente, se realizarán cuatro versiones del programa. Cada una de ellas tendrá una peculiaridad que será la causa que provoque la disparidad en los resultados de cada versión.

La primera versión es simplemente la reproducción del pseudocódigo del primer apartado a un programa en C. La segunda versión consiste en la optimización del código. Como resultado, tendremos una mejora en el tiempo de ejecución. Esta mejora se consigue realizando cambios en la estructura del código, como por ejemplo cambios en la inicialización de estructuras (en este programa no las hay, por lo que esta optimización no es válida), intercambio de bucles para mejorar la localidad espacial (es dependiente del lenguaje de programación), o fusión de bucles para mejorar la localidad temporal, entre otros tipos de mejora.

En este caso se ha decidido realizar un desenroscado de bucles (denominado en inglés *unrolling*), que consiste en el intento de mejorar la velocidad de ejecución reduciendo las instrucciones que controlan el bucle haciendo más extensa cada iteración dentro del mismo. De esta forma se consigue reducir la predicción de saltos y de instrucciones, a costa de aumentar el tamaño del programa. También es importante aplicar esta optimización con cautela, pues puede ocurrir que el aumento de memoria no compense a la mejora temporal.

En las siguientes fotos se muestra la diferencia entre un bucle sin optimizar (primera imagen) y un bucle optimizado con *unrolling* (segunda imagen).

```
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        for (int k = 0; k < 8; k++)
        {
            d[i][j] += 2 * a[i][k] * ( b[k][j] - c[k] );
        }

    }
}
```

Como podemos ver en la comparación, en la versión optimizada (arriba derecha) aumenta el número de instrucciones por bucle, pero se ahorra el tercer bucle k utilizado en la versión de arriba. Esto se debe a que este mismo bucle ha sido eludido utilizando 8 instrucciones independientes, en vez de un iterador k que realice la instrucción 8 veces. Así aumentará la memoria utilizada, pero el tiempo será menor al no haber un tercer bucle.

```
for(int i=0;i<N;i++){
    for(int j=0;j<N;j++){
        d[i][j] += 2 * a[i][0] * ( b[0][j] - c[0] );
        d[i][j] += 2 * a[i][1] * ( b[1][j] - c[1] );
        d[i][j] += 2 * a[i][2] * ( b[2][j] - c[2] );
        d[i][j] += 2 * a[i][3] * ( b[3][j] - c[3] );
        d[i][j] += 2 * a[i][4] * ( b[4][j] - c[4] );
        d[i][j] += 2 * a[i][5] * ( b[5][j] - c[5] );
        d[i][j] += 2 * a[i][6] * ( b[6][j] - c[6] );
        d[i][j] += 2 * a[i][7] * ( b[7][j] - c[7] );
    }
}
```

En el caso de la tercera versión se utilizarán instrucciones de procesamiento vectorial SIMD (Single Instruction Multiple Data). Concretamente, se usarán las instrucciones SSE3, que servirán para sumar, restar, multiplicar o dividir registros. Las instrucciones que componen a esta extensión se pueden ver en la página web de Intel [1]. Estas instrucciones sirven para registros de 128 bits. Esto significa que al usar variables tipo double en nuestro programa, cabrán 2 doubles por variable (1 double = 8*8 = 64 bits).

La cuarta y última versión se realiza utilizando OpenMP, que consiste en una serie de instrucciones que permiten paralelizar un código. Así, podremos computar diferentes cálculos a la vez, sin tener que realizarlos secuencialmente.

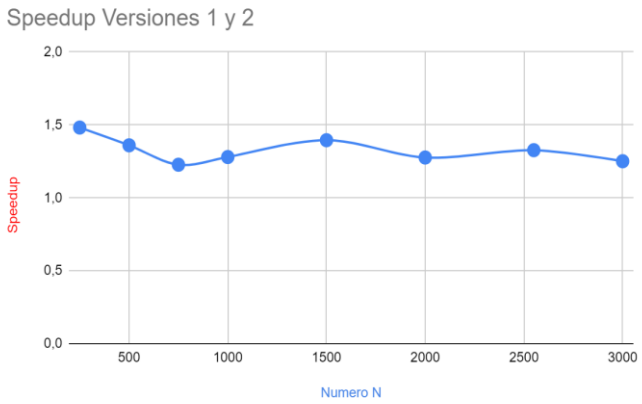
Todas las versiones se ejecutarán en función de diferentes tamaños de matrices: 250, 500, 750, 1000, 2000, 2500 y 3000. A la hora de compilar se utilizarán diferentes opciones de compilación: -o0, -o2 y -o3. Al emplear -o0 se desactivarán todas las optimizaciones de compilación. Las opciones -o2 y -o3 activan optimizaciones de compilación, por lo que cuanto más alto sea el número de la opción de compilación, más optimizaciones se usarán en ella.

IV. RESULTADOS

Para la toma de resultados se ha decidido ejecutar cada programa de forma individual, es decir, ejecutando el programa una sola vez. Como ejecutar cada código una sola vez puede dar lugar a datos erróneos, cada uno será ejecutado numerosas veces para evitar que se tomen valores atípicos. De esta forma y como se ha mencionado en anteriores apartados, se hallará la media de las ejecuciones.

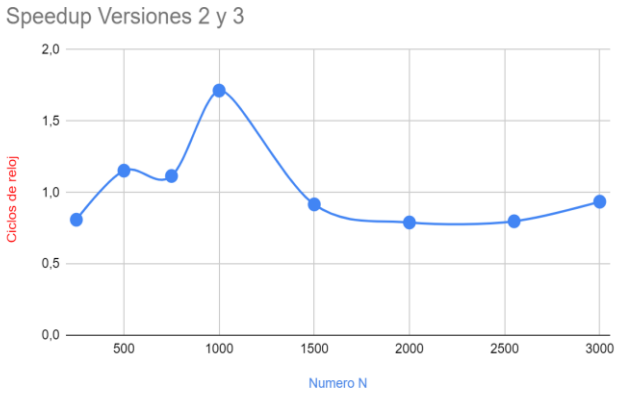
Una gran parte de las gráficas que van a ser presentadas reflejan el *speedup* de un programa respecto a otro. Este término consiste en la relación entre el tiempo de ambos programas. En otros términos, es la aceleración (o ganancia de velocidad) que se consigue con un programa respecto al otro. Al ser una relación entre dos tiempos, la fórmula para su cálculo es hallar el cociente de los dos tiempos que se comparan.

La primera gráfica que obtenemos es el *speedup* entre la versión secuencial optimizada del código y la versión inicial compilada con -o0, es decir, sin optimizaciones de compilación.



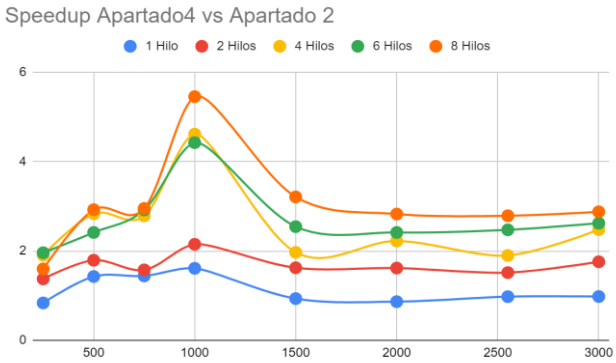
Como se pude ver, el speedup se mantiene más o menos constante en función del tamaño de la matriz. Al ser mayor que 1, se puede confirmar que la versión secuencial optimizada es verdaderamente rentable.

La segunda gráfica representa la ganancia de velocidad entre la versión realizada con instrucciones vectorial SSE3 y la versión secuencial optimizada.



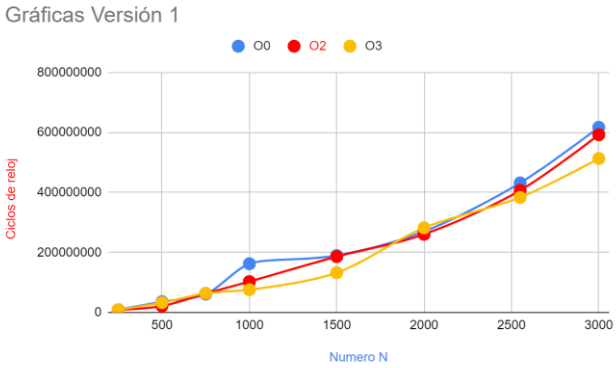
En este caso ocurre un pico cuando el tamaño es igual a 1000, aunque en el resto se mantiene estable. Este pico es debido al aumento de ciclos que ha habido en la ejecución de la versión secuencial optimizada del código cuando el tamaño es 1000. Este pico se verá también en la siguiente gráfica ya que en ella también se representa el *speedup* respecto al ejercicio 2. También se puede concluir que la versión secuencial optimizada es generalmente más rápida que la versión con implementación en SSE3, pues el *speedup* es menor que 1.

La siguiente gráfica muestra igualmente un *speedup*, pero esta vez de la versión en OpenMP del código respecto a la versión secuencial optimizada.



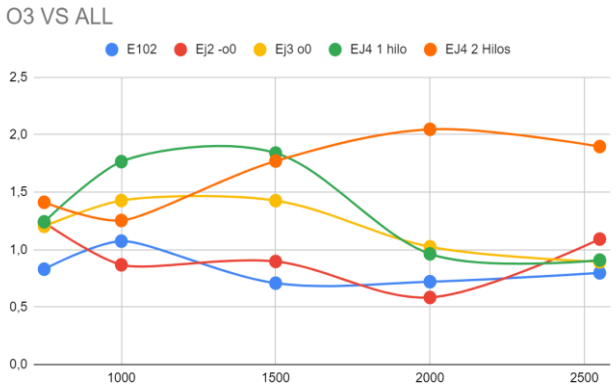
En esta gráfica también se mantiene constante la aceleración, excepto en el pico ya explicado anteriormente.

La siguiente gráfica muestra una comparación de los ciclos de reloj entre las diferentes opciones de compilación de la versión inicial sin optimizar del código. Estas opciones que aparecen son: -o0, -o2 y -o3, ordenadas en este orden de menor a mayor optimización.



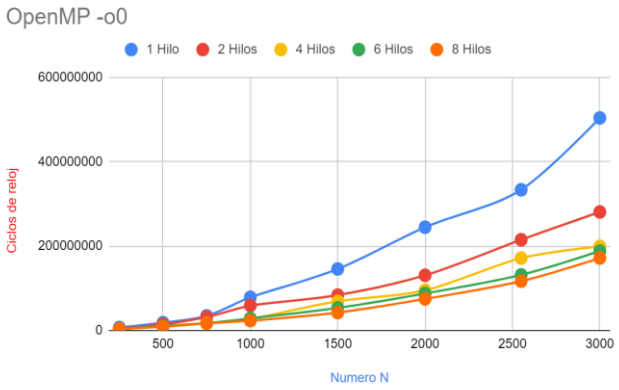
Como es normal, los ciclos aumentan cuando el tamaño aumenta. Además, se puede ver que las opciones de compilación reducen ligeramente el número de ciclos en la ejecución.

Otra gráfica importante es la comparación de la versión inicial del código, compilada con -o3, respecto al resto de versiones. De cada versión de código se ha elegido sólo una opción de compilación para simplificar la gráfica. Asimismo, de la versión 4 sólo se han escogido las versiones de 1 y dos hilos.

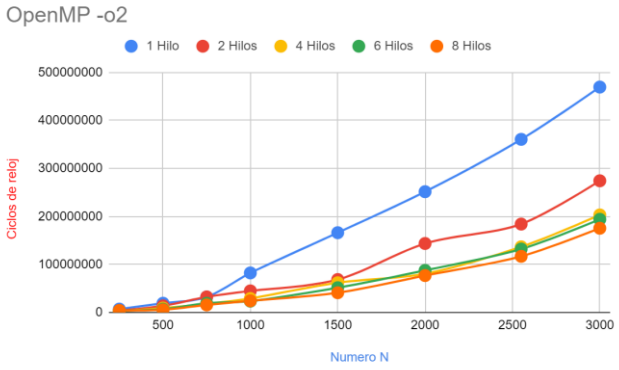


Como se puede ver en esta gráfica, las versiones realizadas en OpenMP son mucho más rentables. A partir del tamaño 1500, la versión con dos hilos comienza a ser más rápida que la de un simple hilo. Además, se puede ver como en algunos casos las versiones 1 y 2 llegan a ser más lentas (aceleración < 1) que la versión 1 compilada con -o3.

Las siguientes gráficas representan los ciclos obtenidos en la versión en OpenMP y en función de N, siendo N el número de hilos utilizados, que puede ser 1, 2, 4, 6 y 8. La primera refleja la compilación con -o0:

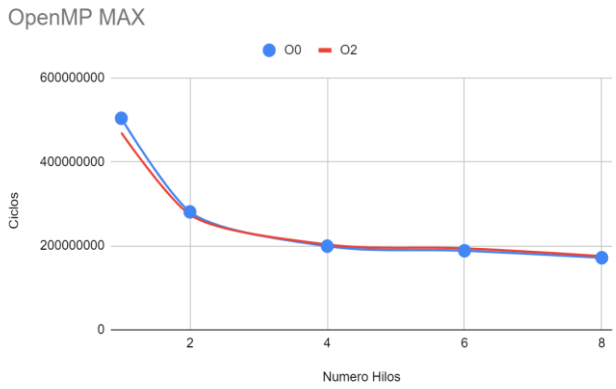


La segunda refleja el mismo escenario pero con compilación en -o2:



De ambas gráficas deducimos que el uso de más hilos ayuda a mejorar el tiempo de ejecución. Sin embargo, esa mejoría es cada vez menor una vez se va aumentando el número de hilos. También se puede observar que el número de ciclos es ligeramente inferior en la compilación en -o2.

La última gráfica a representar muestra una comparación de las versiones en OpenMP en función del número de hilos utilizados mostrando el número de ciclos de reloj para el tamaño de matriz más grande (3000). Se muestran dos líneas, que representan la compilación con -o0 (sin optimizaciones) y -o2 (más optimizaciones).



Gracias a esta gráfica se puede volver a comprobar que un mayor número de hilos optimiza la ejecución. También se puede volver a observar, y esta vez de forma más clara, la ligera mejoría al emplear mejoras en la compilación, sobre todo en el caso de un hilo.

V. Conclusiones

A lo largo de este informe se ha presentado un problema de optimización de un código con diferentes opciones de optimización para ver cuáles eran los resultados. De estos resultados obtenidos, se pueden obtener diferentes conclusiones.

La primera y más evidente conclusión es la mejoría a la hora de paralelizar el código. Gracias a las instrucciones OpenMP, se pueden usar varios hilos a la vez que permiten ejecutar simultáneamente diferentes partes del programa. Como resultado, el tiempo de ejecución disminuye.

Otro efecto bastante notorio es la diferencia en la programación de un código. Simplemente con optimizar la manera de programar se puede conseguir una mejora sustanciable en el tiempo de ejecución. Este fenómeno de mejoría es claro si se comparan los tiempos entre la versión secuencial simple y la versión secuencial optimizada.

En cuanto a la programación con instrucciones vectoriales, el intento de optimización no ha dado sus frutos. Generalmente, este tipo de instrucciones provoca un mayor número de ciclos que la versión secuencial optimizada.

También ayuda en la optimización del código el uso de opciones de compilación, ya que como se puedo ver en el apartado anterior, todos los códigos ejecutados con opciones de optimización daban mejores resultados.

Como corolario, la mejor forma de optimizar un programa es empleando varios hilos, teniendo en cuenta que a partir de un número elevado de hilos el factor de mejor va a ser menor, por lo que llegado a un punto no seré rentable utilizar un mayor número de hilos. Además, las optimizaciones de compilación suponen una ayuda para mejorar el tiempo de ejecución de un programa. Por otra parte, la mejora en la algoritmia del código también juega un papel importante, demostrando así el especial hincapié que se hace generalmente en optimizar el código a la hora de desarrollarlo.

Los cuatro códigos utilizados para realizar los resultados serán adjuntados en el .zip en el que se presenta este informe.

REFERENCIAS

[1] Intel Intrinsics Guide, https://software.intel.com/sites/landingpage/IntrinsicsGuide/ (última revisión 8 Mayo de 2021).