Programación Multinúcleo y Extensiones SIMD

Cristian Novoa Gonzalez, Ivan Quintáns González

Arquitectura de Computadores
Grupo 04
{cristian.novoa,ivan.quintans}@rai.usc.es

7 de mayo de 2023

Resumen

Realización de un estudio sobre el efecto computacional de la utilización de programación multinúcleo, el uso de extensiones SIMD, la utilización de diferentes estrategias de reducción de fallos caché y de la realización de diversas optimizaciones en el código. Teniendo en cuenta el impacto en las prestaciones de programas en microprocesadores. Caracterización del coste computacional mediante la realización de estas optimizaciones en un código base en lenguaje C, determinando el impacto de cada una de ellas. Análisis del efecto de las optimizaciones mediante variaciones en el tamaño del conjunto de datos y en el numero de procesadores empleados.

Palabras clave: SIMD, Optimizaciones Caché , OpenMP, Ciclos, Paralelización, Vectorización, Compilador

I. Introducción

Este estudio se centra en el análisis de la influencia de los diferentes grados de optimización que se pueden aplicar a un código en lenguaje C implementando un algoritmo simple de matrices en punto flotante.

Éste se realizará mediante la medición y posteriormente comparación, de los ciclos que tarda el microprocesador en función de los diferentes grados de optimización aplicados.

Para la realización de este experimento se va a utilizar un nodo del CESGA con un procesador Intel(R) Xeon(R) Platinum 8352Y CPU @ 2.20G con 64 cores y 256GB de memoria ya que es un experimento muy costoso computacionalmente y en una máquina normal sería una ejecución muy llevadera y costosa.

En primer lugar se describirá la metodología empleada. En ella se describirá la implementación del algoritmo de matrices; el método empleado para obtener el número de ciclos de reloj y asegurarse de la validez de los resultados y, finalmente, las optimizaciones aplicadas a cada versión del experimento.

Cabe destacar la explicación de los métodos empleados en cada uno de las diferentes versiones del algoritmo presentes en la experimentación. En la primera de las versiones se implementará en lenguaje C el pseudocódigo proporcionado para el experimento de forma directa. En la segunda se realizará una optimización sobre la primera de las versiones mediante optimizaciones basadas en el uso de memoria caché en la que se buscara el análisis del impacto de estas sobre los ciclos de reloj. En la tercera se optimizará la segunda de las versiones y para ello se dará paso a las extensiones vectoriales SIMD obteniendo un código optimizado vectorizado utilizando paralelismo a nivel de datos. Finalmente se realizara en la cuarta y ultima de las versiones, en ella también se optimizará la segunda versión, y para ello se realizará un programa optimizado paralelizado utilizando programación paralela en memoria compartida mediante OpenMP.

En segundo lugar se tratarán los resultados obtenidos del experimento y se analizará su relación con los diferentes grados de optimización para los diferentes valores de N (número de filas y columnas) y otros parámetros como el numero de hilos empleados.

Finalmente, se describirán las conclusiones obtenidas de este estudio.

II. Metodología

En este apartado se realizará la implementación y explicación de las diferentes versiones de optimización realizadas sobre el pseudocódigo. Para ello se analizará en cuatro apartados diferentes la implementación de cada una de las versiones.

Para la experimentación de las optimizaciones fue proporcionado un pseudocódigo que implementa un algoritmo básico sobre matrices en punto flotante y devuelve la reducción de suma de la diagonal de una matriz D de tamaño *NxN* accedida mediante un vector de índices desordenado.

Entrada: *a* [N][8], *b*[8][*N*], *c*[8]
Salida: f:
Computación:
d[N][N]=0
for (i=0; i<N; i++) **do**
for j=0; j<N; j++) **do**
for k=0; k<8; k++) **do**
d[i][j] += 2 * a[i][k] * (b[k][j]- c[k]);
end for
end for
end for
f=0;
for (i=0; i<N; i++) **do**
e[i]= d[ind[i]][ind[i]] / 2;
f+=e[i];
end for
Imprimir: f

Con el fin de comparar la eficiencia de las diferentes versiones del programa y realizar este estudio de una manera objetiva, se debe asegurar que aunque se realicen las optimizaciones en el código este debe de tener el mismo valor de f. Se debe asegurar que las optimizaciones realizadas no tienen influencia en el correcto funcionamiento del código y se debe garantizar la inicialización de los valores de la matriz d a 0 y el cálculo de esta en su totalidad.

Una vez analizado el pseudocódigo base, el informe se centrará en analizar cada una de las mejoras realizadas en los distintos grados de optimización.

Finalmente, una vez implementado el pseudocódigo, este se compilará mediante un script de bash donde se le pasarán los argumentos necesarios para su correcta ejecución. Estos argumentos se corresponden con el valor del tamaño de la matriz *N* y, para la versión 4 empleando OpenMP, el número de hilos *C*.

A. Versión 1: Implementación de Pseudocódigo

El objetivo del primero de los apartados se centra en la implementación directa del pseudocódigo proporcionado. Esta implementación permite partir de un punto base que nos proporcione tener el código en lenguaje C para poder realizar las mediciones de tiempo y marcar un estándar sobre el que se realizarán las optimizaciones.

Para la implementación del pseudocódigo a lenguaje C se tuvieron en cuenta varios factores.

En primer lugar se realizó la reserva de memoria con la función *malloc* para cada una de las matrices y vectores que se fueran a utilizar en el código. Además se debía asegurar el paso por línea de comandos del tamaño de filas y/o columnas de las matrices a utilizar.

Para las matrices se pedía que algunas de ellas fueran inicializadas con valores aleatorios por lo que para poder comprobar que en cada una de las versiones se llegaba al mismo valor para f se definió una semilla para la función *rand* que permite que en todos los experimentos estas tomen los mismos valores aleatorios.

Además de esto en el script de compilación de los apartados asegura que la salida de cada una de las versiones sea el mismo valor de f para así poder asegurar una correcta realización de las optimizaciones sin modificar la funcionalidad del código.

A mayores se debe de realizar la medición de ciclos de reloj del script solamente en la parte de computación que es la parte a mejorar en cada una de las versiones del programa. Para ello se añadió una funcionalidad proporcionada que permite medir los ciclos de reloj de la parte de computación del pseudocódigo. Estos ciclos de reloj son calculados mediante las funciones *start_counter()* y *get_counter()*

colocadas al principio y final de la zona de computación.

Finalmente, esta implementación directa del pseudocódigo se compilará y ejecutará con 3 grados distintos de optimización realizados por el compilador. El primero de estos es con la flag *-O0* que se encarga de compilar el código con el mínimo de optimizaciones. Este ejecutable será el que se use como base para la comparativa con el resto de experimentos. Las otras dos compilaciones se realizarán con las flags *-O2* y *-O3*. Para observar una lista detallada de las optimizaciones habilitas por cada una de estas flags, consúltese [2].

B. Versión 2: Versión secuencial

El objetivo de la versión 2 del proyecto es realizar optimizaciones basadas en el uso de memoria caché con el fin de obtener el mismo resultado que en la anterior pero en menos ciclos de reloj. Para esto se implementaron las siguientes optimizaciones en el código.

La primera optimización que se implementó es emplear los registros de almacenamiento temporal de una forma eficiente y no redundante. Concretamente, para aplicar esta optimización, en esta versión se prescindió del vector *e* y los resultados de la reducción de suma se almacenaban directamente en la variable de retorno *f*.

De la misma manera, en el cálculo de la variable f, se buscó reducir el número de instrucciones empleadas realizando primero el sumatorio de todos los valores de la matriz *d* y una vez calculados, dividir la totalidad de f por la mitad, en vez de realizar esta división en cada iteración del bucle.

Para todos los bucles de la parte de computación se buscó realizar el desenrollamiento del lazo. Esta es una técnica de optimización que busca reducir el tiempo de acceso a memoria a coste de incrementar el tamaño del archivo compilado. Esta técnica consiste en reducir el número de iteraciones implícitas que realiza un bucle mediante la realización de estas iteraciones de forma explícita en el código. Esto reduce el número de veces en las que se tiene que realizar la predicción de salto en el bucle, comprobar su condición e incrementar sus iteradores. De esta manera, se está reduciendo el número de accesos a memoria en los iteradores, el número de ciclos totales y se está mejorando la localidad temporal de los datos permitiendo cargar más datos útiles en cache.

Para un correcto desenrollo de lazo, es fundamental asegurarse que el número de iteraciones *N* sea módulo del factor de desenrollo (8) para evitar accesos a posiciones no válidas del vector/matriz que acabarían provocando *SegFault*.

Esta optimización aplicada a la versión 1 lleva a la supresión del bucle *k* y, junto con las 2 anteriores, a obtener el siguiente bucle de reducción de la diagonal:

```
end = N - (n%8)
for (i=0; i<end; i+=8) do
  f += d[ ind[i] ][ ind[i] ];
  f += d[ ind[i] + 1 ][ ind[i] + 1];
  ...
  f += d[ ind[i] + 7 ][ ind[i] + 7];
end for
for (i=end; i<N; i++) do
  f += d[ ind[i] ][ ind[i] ];
end for
f /= 2;
```

Las últimas optimizaciones implementadas solo afectan a la inicialización y cálculo de la matriz *d* y son la fusión de bucles y el acceso por bloques.

La fusión de bucles es la optimización con más impacto en la reducción del número de ciclos de reloj. La fusión consiste en combinar uno o más bucles en uno solo que contenga todas las funcionalidades de los anteriores. Para la implementación de la versión 1, la fusión se puede aplicar en la inicialización de la matriz *d* y en el cálculo de sus valores resultando en un único bucle.

Finalmente, la realización del acceso por bloques tiene como principal objetivo mejorar la localidad espacial de los datos aprovechándose del *prefetching* caché que consiste en traer a caché los datos adyacentes al accedido o que se prevé que se van a acceder en el futuro.

El acceso por bloques se implementa limitando el rango de valores que pueden tomar las variables *i* y *j* usadas para acceder a posiciones de memoria a un intervalo reducido del tamaño del bloque. De esta manera, se busca que para el intervalo reducido de *i* (supongamos de 0 a 2) se calculan todos los

valores de *j* bloque a bloque (0 a 4, 4 a 8...) de forma que primero se calcula el bloque 1 de *j* para todos los valores del intervalo de *i*, a continuación el bloque 2 de *j*... Una vez acabado ese intervalo de *i* se pasaría al siguiente y se repetiría el mismo método.

De esta forma, aunque se aumente el número de iteraciones, se consigue tener muchos más datos útiles en memoria caché que de realizar iteraciones normales. Para lograr esto es imprescindible calcular un buen valor para el tamaño del bloque teniendo en cuenta el tipo de datos utilizados, el número de datos accedidos en cada iteración y el tamaño de la caché del computador.

Así es que, empleando la caché de Nivel 1 (*L1*) de 48KiB y sin contar las 6 variables enteras de 4 bytes se obtienen 49136 bytes disponibles, lo que equivaldrían a espacio para 6142 doubles (1 double = 8 bytes = 64 bits) Considerando que el vector *c* (double) (accedido en su totalidad en cada iteración) se desea tener siempre en caché y tiene tamaño 8, y que en cada iteración del bucle se realizan 17 accesos a memoria (8 para el vector a, 8 para el vector b y 1 para d) se puede calcular que se requieren: $\lfloor \frac{(6142-8)}{17} \rfloor = 360$ iteraciones. Aplicándole la raíz cuadrada se obtienen un intervalo de 18 valores para la *i* y para la *j*. Con el fin de asegurar un tamaño de bloque adecuado y, para evitar fallos en el cálculo de la misma, se optó por un tamaño de bloque (*BSIZE*) de 15.

De esta manera, el bloque de computación de la matriz *d* una vez aplicadas estas optimizaciones es el siguiente:

```
for (bi = 0; bi <N; bi += BSIZE) do
  for (bj = 0; bj <N; bj += BSIZE) do
    for (i=bi; i<min(bi + BSIZE, N); i++)
    do
      for j=bj; j<min(bj + BSIZE, N); j++)
      do
        d[i][j] = 0;
        d[i][j]+=2*a[i][k]*(b[k][j]-c[0]);
        d[i][j]+=2*a[i][k]*(b[k][j]-c[1]);
        ...
        d[i][j]+=2*a[i][k]*(b[k][j]-c[8]);
      end for
    end for
  end for
end for
```

Una vez implementadas las optimizaciones, el script se compilará con la flag *-O0* de forma que se eviten optimizaciones no deseadas por parte del compilador y así poder observar correctamente el impacto en el número de ciclos de esta implementación con respecto a la anterior.

C. Versión 3: SIMD

Para esta versión se ha empleado la implementación de la versión 2 como base y tiene como objetivo vectorizar sus operaciones con el fin de tratar datos de forma paralela. Para lograr este objetivo se ha implementado la libreria AVX 512 aprovechando al máximo la capacidad del CESGA usando vectores de 8 doubles *__m512d* para las operaciones.

Aplicamos la vectorización al código base de la versión 2 substituyendo las operaciones de desenrollo del lazo por operaciones vectoriales.

Es fundamental a la hora de implementar vectorización que los datos a vectorizar estén alineados. Para lograr esta condición, se reservó memoria para arrays y matrices empleando la función *aligned_alloc*.

De esta manera la operación $2 * a[i][k] * (b[k][j] - c[0])$ para todos los elementos del vector c se convierte en: $2 * a\_vec * (b\_vec - c\_vec)$ donde *a_vec* es un vector con los 8 elementos de la fila *i*, *b_vec* un vector con los 8 elementos en la posición *j* de cada una de las 8 filas de la matriz y *c_vec* el *array* de 8 elementos c convertido en vector.

De forma similar, la operación de reducción de suma de la diagonal de la matriz en el lazo $f+ = d[ind[i]][ind[i]]$; la realizamos calculando el *vec_filas*, que contiene 8 valores usados como índice de filas de la matriz d, y el *vec_cols*, que contiene los 8 valores usados como índice de las columnas.

Para obtener *a_vec* se emplea la función _mm512_load_pd(a + i * 8) para cargar 8 valores de la matriz a en la fila *i* en la que se encuentre. Obtener *b_vec* resulta de una complejidad algo superior ya que el algoritmo accede a la matriz b por columnas en vez de en filas y, al crear la matriz como un array de $N * N$ valores este acceso se realiza de la siguiente forma: $b[k * N + j]$ *for k = 0..8*. Para lograr una vectorización correcta de

esta operación se ha calculado con anterioridad al bucle un vector __m256i con único valor N y un vector __m256i que contiene las posiciones del inicio de cada fila de la matriz *b*, calculado como *n_vec* por un vector con valores del 0 al 7. Para realizar estas operaciones se usan $\_\_mm256\_set1\_epi32$ y $\_\_mm256\_mullo\_epi32$.

Finalmente, se obtiene *vec_filas* y *vec_cols* de la siguiente manera. *vec_cols* se obtiene cargando 8 valores del array *ind* a partir de la posición i mientras que *vec_filas* se obtiene mutliplicando *vec_cols* por *n_vec*. Unaa vez calculados, se realiza la suma de ambos vectores dando como resultado *ind_vec* y se cargan los valores de d de las posiciones indicadas por este vector usando: $\_mm512\_i32gather\_pd(ind\_vec, d, 8)$;

D. Versión 4: OpenMP

Para esta última de las versiones se buscaba un programa optimizado paralelizado utilizando programación paralela en memoria compartida, es decir utilizando la funcionalidad de OpenMP sobre la segunda de las versiones.

OpenMP permite la programación paralela en memoria compartida mediante la utilización de hilos. Esto permite que en el código en lugar de que solo exista un proceso realizando todas las iteraciones puedan haber varios hilos sobre la misma zona permitiendo así una reducción de ciclos de reloj y una ejecución más óptima.

Esto puede ser no tan beneficiable cuando estamos realizando pocas iteraciones ya que la paralelización puede ser más costosa que la realización con un solo proceso por el hecho de tener que paralelizar la región y repartir la CPU. Sin embargo, cuando realizamos esta paralelización sobre bucles que realizan un gran número de iteraciones, se observa notoriamente la reducción en ciclos de reloj como se apreciará en la sección de resultados III.

Para la implementación de este apartado, en primer lugar se debe de incluir la librería <*omp.h*> que nos permite realizar la paralelización. En segundo lugar debemos de declarar las regiones a paralelizar que en este caso son las regiones en las que se realizan los bucles *for*. En cada una de estas regiones debemos de utilizar *#pragma omp parallel for numthreads(C)* que hace que esa región se paralelice dado el número de hilos que participarán en la zona *C* y en nuestro caso con un manejador estático. Esta paralelización se realiza en dos de los bucles en el código, en el bucle que le proporciona los valores de la matriz D y el bucle que se encarga de realizar la reducción de suma de f.

Por otra parte cabe destacar el manejo de la zona paralelizada ya que esta puede tener diferentes modos de *Scheduling* que influyen a la hora de como los hilos se reparten el trabajo dentro de la región paralelizada. Para poder asignar estos modos a la región paralela se debe de utilizar *#pragma omp parallel for schedule() numthreads(C)* siendo schedule donde colocamos el tipo de manejador que se quiere aplicar a esta región paralelizada.

Schedule tiene 5 manejadores distintos: static (el por defecto), dynamic, guided, auto y runtime. Runtime y auto no son relevantes para este experimento ya que no afectan al comportamiento en sí, si no que solo determinan cuál de las otras tres opciones emplear. Además, schedule también toma un argumento numérico que define el tamaño de los bloques (*chunksize*) en los que se reparte el número de iteraciones de acuerdo con la estrategia del manejador.

Static reparte las iteraciones en bloques de tamaño *chunksize* de forma ordenada (0..N) a cada uno de los hilos. Dynamic realiza esta misma función pero distribuye los bloques a los hilos sin ningún orden fijado. Finalmente, guided tiene un funcionamiento análogo a dynamic con la diferencia de que el tamaño de los bloques se calcula como el número de iteraciones restantes hasta un mínimo de *chunksize*.

Por último, también es posible colapsar o no los bucles *for* usando la función *collapse* de OpenMP a la cuál se le especifica el número de bucles a colapsar y realiza una fusión de estos. De esta manera, se puede distribuir de forma más equitativa la carga de trabajo de cada hilo, especialmente para bucles en los que varía la carga computacional.

E. Interpretación y Presentación de los datos

Una vez comprobado el correcto funcionamiento de cada una de las versiones fue necesario el procesamiento de los datos. Se realizaron 10 mediciones para cada N y para cada una de las versiones de las cuales se trabajó con la mediana. Estos datos fueron redireccionados a un archivo para una más cómoda manipulación de estos.

Una vez obtenidos los resultados utilizamos matplotlib [1] para poder representarlos en gráficas y de esta manera poder conseguir una mejor visualización de los resultados obtenidos.

III. Resultados

Mediante el uso de las técnicas utilizadas en la sección E, los resultados obtenidos en la ejecución de las versiones del experimento serán representados en diferentes secciones.

A. Optimizaciones Compilador

La implementación del pseudocódigo base en lenguaje C permite marcar un punto de partida de comparación de velocidad con el resto de versiones optimizadas. Los resultados obtenidos en este apartado son visibles en la figura 1 en la que se muestra el *speed up* en función de los diferentes niveles de optimización ya realizados por el compilador. En este caso, se muestra el *speed up* de la versión 1 compilada con *-O2* y *-O3* frente a la versión compilada con *-O0*.

B. Optimizaciones Manuales

En la figura 2 se pueden observar los *speed up* con respecto al código base por parte de la implementación de las versiones 2, 3 y 4 (usando 4 hilos).

Esta comparativa nos permite analizar la eficacia de las optimizaciones realizadas en la versión 3 y 4 frente a la versión 2, donde es notoria la ganancia en velocidad de la implementación con OpenMP frente a la segunda de las versiones.

En el caso de la versión con extensiones SIMD se puede apreciar como no existe una

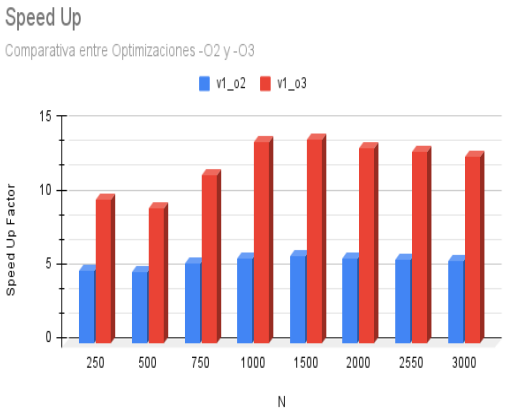


Figura 1: *Comparativa Optimizaciones del Compilador*

gran ganancia de velocidad con respecto a la versión 2. Una posible mejoría en la implementación de esta versión vectorizada sería realizar la traspuesta de la matriz b ya que se accede a esta por columnas. En cambio, no se ha querido realizar esta mejoría porque se consideró que sería un cambio substancial con respecto al código base y por consecuente su futura comparación no sería apropiada.

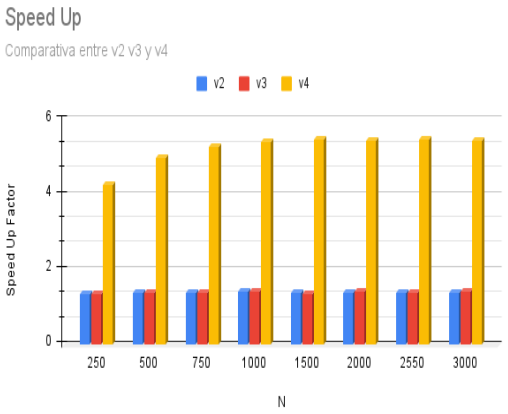
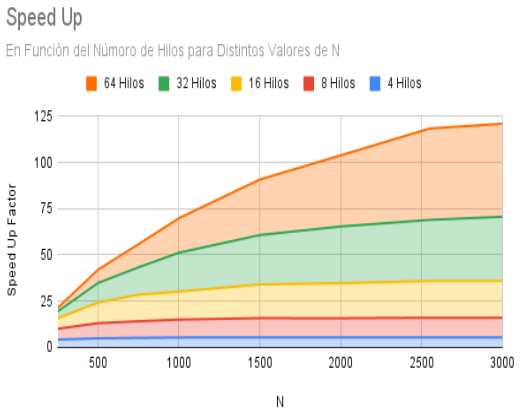


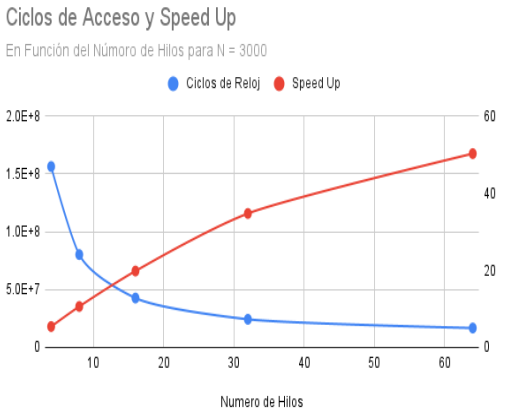
Figura 2: *Comparativa Optimizaciones Manuales*

C. Influencia del Número de Hilos en la Implementación con OpenMP

Una de las ventajas de la paralelización empleando OpenMP es la posibilidad de emplear los recursos del CESGA en su máxima capacidad ya que se puede variar el número de hilos en los que se divide la computación.



(a) *Para los distintos valores de N*



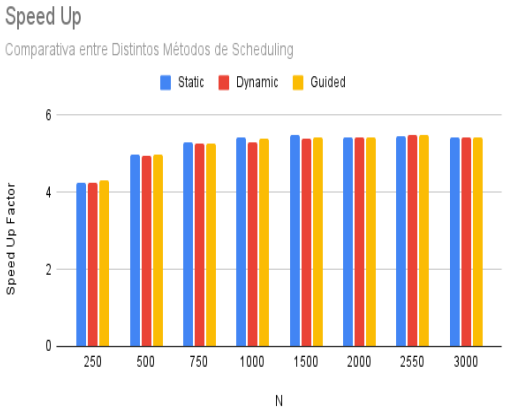
(b) *Para el valor máximo de N*

Figura 3: *Comparativa para los distintos números de hilos*

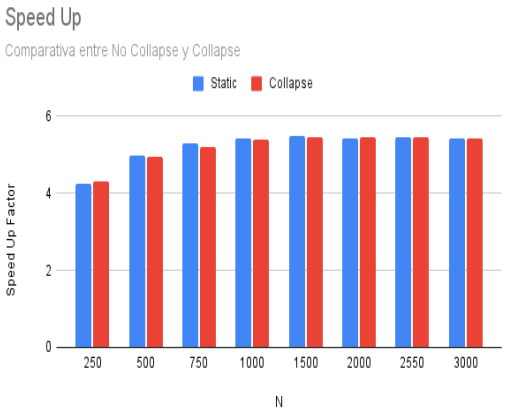
En la figura 3 se puede apreciar los distintos *speed ups* que ofrece el uso de 4, 8, 16, 32 o 64 hilos para los distintos valores de N en el caso de la figura 3a y concretamente para el máximo valor de N en la figura 3b. Como es apreciable, a mayor número de hilos, mejores resultados se obtienen de la paralelización y, por consecuente, mayor *speed up*.

D. Influencia del Scheduler y de Collapse en la Implementación con OpenMP

Las estrategias de Scheduler y de Collapse tienen especial relevancia en casos donde la carga computacional varía de una iteración a otra, pero en este caso, en todas las iteraciones se realizan el mismo número de cálculos. Como se refleja en al figura 4, donde se puede comparar los *speed ups* empleando static, dynamic y guided en la subfigura 4a y el uso o ausencia de collapse en la subfigura 4b, apenas existen variaciones entre las distintas implementaciones por lo que, para este experimento, su uso es indiferente.



(a) *Diferentes modos de Scheduling*



(b) *Collapse vs No Collapse*

Figura 4: *Ciclos de Acceso a Memoria para Doubles*

E. Comparativa con O3

Finalmente, se ha comparado el código base de la versión 1 aplicándole todas las optimizaciones posibles del compilador empleando la *flag* -O3 con respecto a las versiones 2, 3 y 4 en las que se implementaron muchas de las optimizaciones que realiza el compilador con esa *flag* de forma manual.

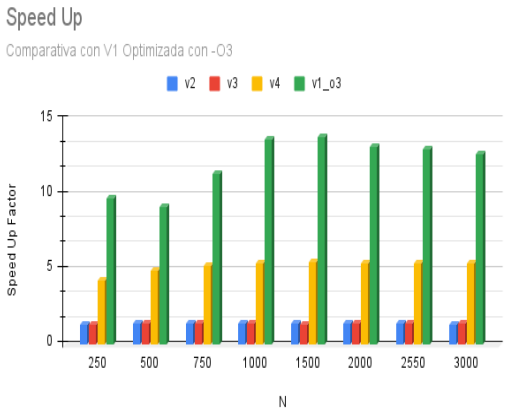


Figura 5: *Comparativa Optimizaciones Manuales vs. -O3*

Como se puede apreciar en la figura 5, las optimizaciones del compilador son muy superiores a las implementadas de forma manual, siendo unas 4 veces más rápida que la versión con OpenMP y unas 10 veces superior a las versiones 2 y 3. Esta gran diferencia en parte se debe a que el compilador aplica las optimizaciones de la versión 2 y la vectorización de forma conjunta, junto con muchas otras optimizaciones que no se han considerado realizar de forma manual en este experimento.

IV. Conclusiones

En este estudio se ha analizado la influencia de las diferentes optimizaciones realizadas sobre un código base. Esta influencia se ve reflejada en el número de ciclos de reloj que tarda un programa con un algoritmo básico de matrices en punto flotante y que devuelve la reducción de suma de la diagonal de una matriz D de tamaño *NxN*, accedida mediante un vector de índices desordenado. Para lograr dicho estudio, se han realizado varios experimentos en los que se han aplicado diferentes grados de optimización empleando diferentes técnicas. Estas técnicas son las optimizaciones basadas en el uso de memoria caché, optimizaciones empleando paralelismo a nivel de datos(procesamiento vectorial SIMD) y finalmente optimizaciones empleando programación paralela en memoria compartida (OpenMP).

En cada una de estas versiones se ha calculado el impacto en ciclos de reloj en función del tamaño de la matriz. De este modo se observa la influencia de las optimizaciones que puede realizar el compilador y el impacto de estas en cuanto a términos de velocidad de ejecución.

En base a los resultados obtenidos se puede observar la importancia de las distintas técnicas de optimización empleadas (mejoras de uso caché, paralelismo a nivel de datos y paralelismo en memoria compartida) que se han estudiado y analizado. Cabe destacar que la implementación manual de estas optimizaciones no es comparable a su implementación por el compilador. Sin embargo, el uso de estas técnicas cuando los requerimientos del código las permitan obtienen un cambio notorio en cuanto a velocidad de ejecución.

Como posibles estudios futuros a realizar serían interesantes la implementación de la versión 3 realizando la traspuesta de la matriz b, la comparación en tiempo de ejecución en realizar operaciones en punto flotante o no y las ventajas e inconvenientes de la paralelización empleando OpenMP.

Referencias

[1] Matplotlib 3.7.1 documentation., https://matplotlib.org/stable/index.html, [online] última visita 20 de marzo de 2023.

[2] Optimize Options (Using The GNU Compiler Collection (GCC)), https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, [online] última visita 07 de Mayo de 2023.

[3] OpenMP - Scheduling(static, dynamic, guided, runtime, auto) - Yiling's Tech Zone, https://610yilingliu.github.io/2020/07/15/ScheduleinOpenMP/, [online] última visita 07 de Mayo 2023.