Programación Multinúcleo y extensiones SIMD

Carlos Cao López, Pedro Vidal Villalba

Arquitectura de Computadores
Grupo 09
{carlos.cao,pedro.vidal.villalba}@rai.usc.es

26 de abril de 2024

Resumen

En este informe se recogen los resultados, así como las comparativas relativas a las distintas ejecuciones sobre un mismo código `C`, con distintos grados de optimización, que implementa un algoritmo con matrices en punto flotante. Para ello, se han utilizado estrategias para la reducción de fallos caché, extensiones vectoriales SIMD y parelización a nivel de hilos mediante la librería OpenMP. Además, se han añadido las ejecuciones del código base con los niveles de optimización automática del compilador `-O0`, `-O2` y `-O3`. Tras el estudio, se concluyó que el programa que mejor rendimiento ofrece es el que usa OpenMP con la optimización del compilador `-O2`.

Palabras clave: optimización, paralelización, AVX512, OpenMP, desenrollamiento, block-tiling, fallo caché, multinúcleo.

I. Introducción

El objetivo de este trabajo es realizar una comparativa entre distintas implementaciones de un mismo algoritmo que realiza operaciones sobre matrices, programado en C, con distintos grados de optimización. Para analizar el rendimiento de cada una de forma cuantitativa, se ha medido el número de ciclos de reloj que tarda en ejecutarse una determinada sección intensiva en cómputo en las distintas versiones.

Para ello, se ha partido de un código base que implementa el pseudocódigo 1. A partir del código base secuencial, se han escrito otros 3 programas que buscan obtener los mismos resultados, reduciendo el tiempo de ejecución. En el primero de ellos, se han reducido los fallos caché mediante diversas técnicas manuales; en el segundo, se ha partido del programa con las optimizaciones caché y se han usada instrucciones vectoriales `AVX512` para aprovechar el paralelismo a nivel de datos y, en el último, de nuevo partiendo del código con las optimizaciones caché, se ha usado programación paralela mediante OpenMP.

En la sección *II. Descripción de los experimentos*, se describen en detalle las diferentes versiones de los experimentos realizados, con los cambios más destacables entre cada una. En la sección *III. Resultados*, se recogen los resultados con las medidas de rendimiento realizadas en diferentes gráficas y se realiza un análisis detallado de los mismos. Finalmente, en la sección *IV. Conclusiones*, se realiza un resumen del trabajo realizado y se extraen conclusiones de los resultados obtenidos.

II. Descripción de los experimentos

Cada uno de los experimentos ha sido realizado en el supercomputador FinisTerrae III del Cesga. En el momento de la ejecución de los experimentos, este cuenta con las características técnicas expuestas en la tabla 1. Se ha solicitado un nodo con 64 cores (virtuales, en 32 cores físicas, que son las que tiene cada procesador) y 256 GiB de memoria para la ejecución de los programas.

En cada ejecución hemos obtenido el número de ciclos que tarda en ejecutarse una sección intensiva en cómputo del código, re-

Procesador	Intel Xeon Ice Lake 8352Y de 32 cores, 2.2GHz
Tamaño de línea caché	64 bytes
Compilador	gcc, *(Gentoo 10.1.0-r2 p3) 10.1.0*
Shell	GNU bash, *version 5.0.18(1)-release (x86_64-pc-linux-gnu)*
Sistema operativo	Rocky Linux, *release 8.4 (Green Obsidian)*
Kernel de Linux	4.18.0-305.3.1.el8_4.x86_64
Extensiones SIMD	AVX512
OpenMP	Versión 4.5

Tabla 1: *Características FinisTerrae III*

lacionada con operaciones sobre matrices en punto flotante de doble precisión. Las sucesivas versiones realizadas generan los mismos resultados que la versión base, esto es, obtienen la misma salida y dejan todas las variables iniciales con los mismos valores tras la ejecución. De esta forma, los cambios entre versiones vienen dados por cómo se ejecutan dichos cálculos, y no por calcular una menor cantidad de datos.

Se repitieron 25 ejecuciones independientes de cada versión del programa, y se seleccionó la mediana del número de ciclos para cada versión como valor final de medida de tiempo de ejecución. Además, en las gráficas a lo largo del presente documento se representa este valor de la mediana junto a un intervalo de confianza delimitado por los cuantiles 0,25 y 0,75 (incluyendo, así, la mitad central de los datos) para mostrar el grado de variabilidad en las distintas ejecuciones.

A. Programa base

El programa inicial del que se parte —en adelante referido también como `versión 1`— implementa de forma llana el pseudocódigo 1 que se muestra a continuación, midiendo el número de ciclos de reloj que tarda en ejecutarse la parte marcada como *Computación*. Será este mismo bloque de código el que se mida y optimice en las versiones sucesivas.

Pseudocódigo 1: *Pseudocódigo base*

```
// Entradas
double a[N][8], b[8][N], c[8]; // Matrices y vector que almacenan valores aleatorios de tipo double
int ind[N]; // Vector desordenado aleatoriamente que contiene indices de fila/columna sin repetir

// Salida
double f; // Variable de salida tipo double

// Computacion
double d[N][N] = 0; // Inicializacion 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]);

f = 0;
for (i = 0; i < N; i++) {
  e[i] = d[ind[i]][ind[i]] / 2;
  f += e[i];
}
// Imprimir el valor de f
```

A.1. Optimizaciones del compilador

Cabe mencionar que el compilador `gcc` —ver versión en la tabla 1— ofrece distintas opciones de optimización automáticas.

La primera de ellas es `-O0`, con la cual la mayoría de las optimizaciones son desabilitadas con el fin de observar el funcionamiento de la traducción más literal del código.

La segunda usada, `-O2`, alinea la memoria para aprovechar mejor el prefetching y la carga de líneas caché, elimina algunas comprobaciones como la eliminación de punteros nulos, realiza algunas optimizaciones sobre los bucles, como la reducción, y expande las funciones en línea para evitar la sobrecarga en las llamadas.

La opción `-O3`, hace todo lo que la opción anterior y, además, aplica desenrollamiento e intercambio, fusión o separación de bucles (según convenga) así como uso de las instrucciones vectoriales que permita la máquina en que se compile. Para más información sobre las opciones, consúltese [1].

El código base se compiló con cada una de estas tres opciones, dando lugar a las versiones `1o0`, `1o2`, `1o3`, respectivamente, para

comparar los resultados de la optimización manual a realizar frente a la automática que provee el compilador.

B. Programa secuencial con optimizaciones manuales.

Para el segundo de los experimentos —en adeltante también, `versión 2`—, las optimizaciones se programaron manualmente, y han ido orientadas a la reducción de fallos caché, para lo cual se han seguido las siguientes estrategias:

1. Uso eficiente de registros para almacenamiento temporal de datos.
2. Reducción del número total de instrucciones a ejecutar.
3. Fusión y/o intercambio de lazos.
4. Desenrollamiento de lazos.
5. Realización de operaciones por bloques

Cada una de las optimizaciones particulares ha sido probada por separado para asegurar que no interfiriesen entre sí.

Concretamente, para la comparativa entre las distintas versiones, hemos ejecutado los códigos de cada una mediante un nodo interactivo mediante el comando `compute -c 64 --mem 16`.

Primero, hemos ejecutado de forma individual las distintas versiones de este ejercicio, obteniendo los resultados que se puede consultar en la tabla 2.

Versión	Ciclos.25 %	Ciclos.50 %	Ciclos.75 %
1o0	1073.4127	1074.2911	1075.7363
2_1	648.7931	649.1201	649.3966
2_2	1185.3165	1185.9633	1187.6433
2_3	1142.6020	1143.6600	1145.3816
2_4	537.4104	537.6302	538.0894
2_5	1193.2150	1193.3930	1193.7075
2	346.5976	346.8215	347.5839

Tabla 2: *Millones de ciclos de reloj para las diferentes versiones del ejercicio 2,* $N = 3500$

A continuación, hemos seleccionado las dos mejores versiones, correspondientes a juntar el uso de registros (`2_1`) con desenrollamiento (`2_4`) y, variando el programa con otras optimizaciones, hemos percibido que cuando añadimos operaciones por bloques obtenemos de forma consistente que esta versión del programa es más rápida que el programa que solo hace uso de registros y desenrollamiento, a pesar de que aplicando solamente operaciones por bloques los resultados eran incluso peores que en la versión base. La fusión de lazos ralentizaba muy ligeramente el programa, por lo que se prescindió de aplicarla en la versión final, resultando también en un código más simple.

Se explican a continuación, de manera pormenorizada, las distintas estrategias aplicadas para la optimización manual del programa secuencial basada en la reducción de fallos caché.

B.1. Registros

Para el uso eficiente de registros, se ha utilizado la palabra clave de C `register` como modificador de las variables usadas como índices en los bucles en que ocurre la computación intensiva del programa. Además, se han creado las variables `a_index` y `d_index` como registros precalculados en los bucles, para así poder usar el modificador sobre estos datos.

Gracias a estas dos nuevas variables, conseguimos que el programa guarde el contenido de los índices en un registro con mucha mayor probabilidad (aunque la decisión sobre si hacerlo o no sigue siendo del compilador), logrando así una de las mejoras claves en este apartado gracias al continuo acceso sobre estas variables.

B.2. Reducción del número de instrucciones

El pseudocódigo 1 contiene claramente un gran número de operaciones que no son relevantes para calcular el valor final de `f`, pues esta variable es simplemente la suma de los elementos diagonales de la matriz `d`, dividida por 2. Sin embargo, debemos realizar una tarea de optimización no solo para el cálculo de `f`, si no para todo el programa. Así, los programas optimizados tienen que obtener la misma salida desde el punto de vista de todos los datos, esto es, los valores de `d`, `e` y `f` deben calcularse en su totalidad y ser siempre los mismos. Así, no podemos ignorar el uso del vector de índices, pues obtendríamos

e desordenado, ni evitar la multiplicación y posterior división por 2, ya que cambiarían los valores almacenados en d.

Con todo, en esta versión el único cambio que se llevó a cabo fue la recodificación interna de las matrices a, b, c, d y e como arrays unidimensionales de C, en lugar de como arrays bidimensionales, como estaba en la versión básica. Este cambio, que reduce el número de accesos a memoria a realizar, junto con la asociatividad de las operaciones, se aplicó a todas las versiones en adelante.

B.3. Fusión y/o intercambio de lazos

Para la fusión de lazos, — versión 2— del apartado, nos hemos aprovechado de que a la salida del bucle j, habremos calculado ya toda la fila i, y en particular el elemento diagonal de d, luego podremos calcular el elemento correspondiente del vector e y, consecuentemente, la suma de ese elemento al vector f.

Sin embargo, debido al uso de un vector de índices sobre d, a la hora de calcular el elemento de e, a esta solución hemos tenido que añadir la computación asociada al cálculo de la permutación inversa del vector de índices ind[] para guardar el elemento del vector e que hemos calculado en la posición que le corresponde, pues partimos de la premisa de que el programa desde el punto de vista de los datos debe ser el mismo siempre, esto es, los valores de d, e y f deben calcularse en su totalidad y ser siempre los mismos. El coste asociado a calcular este índice inverso cancela el aumento en eficiencia por fusionar los bucles.

Pseudocódigo 2: *Fusion de lazos*

```
// Calculamos el indice inverso, sabiendo que
    ind_inv[ind[i]] = i
for (i = 0; i < N; i++) {
  inv_ind[ind[i]] = i;
}

// Realizar las operaciones especificadas
f = 0;
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    for (k = 0; k < M; k++) {
      d[i*N+j] += 2*a[i*M+k]*(b[k*N+j]-c[k]);
    }
  }
  e[inv_ind[i]] = d[i * (N + 1)] / 2;
  f += e[inv_ind[i]];
}
```

B.4. Desenrollamiento de lazos

Para la resolución de este apartado se ha desenrollado el bucle más interno en el que se relizaba el cómputo. Dado que este bucle tenía $M = 8$ iteraciones fijas, como se puede observar en el pseudocódigo 3, se han desenrollado 8 operaciones sobre d. Cabe mencionar que este valor resulta especialmente conveniente para la posterior implementación con instrucciones vectoriales AVX512 pues, como trabajamos con datos de tipo double, coincide que los 8 operandos resultado de desenrollar este bucle interno caben exactamente en este tipo de registros.

Pseudocódigo 3: *Desenrollamiento de lazos*

```
for (i = 0; i < N; i++) {
  for (j = 0; j < N; j++) {
    d_value = 0;

    d_value += a[i*M  ] * (b[    j] - c[0]);
    d_value += a[i*M+1] * (b[  N+j] - c[1]);
    d_value += a[i*M+2] * (b[2*N+j] - c[2]);
    d_value += a[i*M+3] * (b[3*N+j] - c[3]);
    d_value += a[i*M+4] * (b[4*N+j] - c[4]);
    d_value += a[i*M+5] * (b[5*N+j] - c[5]);
    d_value += a[i*M+6] * (b[6*N+j] - c[6]);
    d_value += a[i*M+7] * (b[7*N+j] - c[7]);

    d[i * N + j] = 2 * d_value;
{...}
```

B.5. Operaciones por bloques

Para la realización de este experimento, se han realizado las mismas operaciones que en el programa base secuencial, con la diferencia de que se ha adaptado el código para que opere por bloques [2], como refleja el pseudocódigo 4.

De esta forma, aprovechándonos de la localidad espacial, el tamaño de bloque se ha calculado de forma que quepa en una línea caché (de 64 bytes en la máquina utilizada). Para la correcta ejecución de este experimento, debemos asegurarnos de que la memoria esté alineada pues, si no lo estuviera, perdería todo el sentido por no aprovechar al máximo los datos cargados en la línea caché.

Pseudocódigo 4: *Operaciones por bloques*

```
// Realizar las operaciones por bloques
    especificadas
uint block_size = line_size / sizeof(double);

```

```
for (i = 0; i < N - N % block_size ; i += block_size) {
  for (j = 0; j < N - N % block_size; j += block_size) {
    for (ii = i; ii < i + block_size; ii++) {
      for (jj = j; jj < j + block_size; jj++){
        for (k = 0; k < M; k++){
          d[ii * N + jj] += a[ii * M + k] * (b[k * N + jj] - c[k]);
        }
        d[ii * N + jj] *= 2;
{...}
```

C. Programa paralelizado usando instrucciones vectoriales SIMD

Para la ejecución de este experimento —`versión 3`— se ha aprovechado la capacidad del ISA de Intel `x86_64` con extensiones `SIMD` [3] —cuya versión podemos encontrar en la tabla 1—, para poder aplicar operaciones vectoriales sobre los datos del programa. Se partió para esto del código final de la versión 2 que se detalla en la subsección B.

Como se puede ver en el pseudocódigo 5, y como hemos mencionado con anterioridad, hemos combinado el uso de `register` con los registros especiales `__m512d`, los cuales nos han permitido hacer todas las operaciones desenrolladas a la vez con una única instrucción. Debemos mencionar que, dado que los valores de `b` con los que se opera no se encuentran consecutivos en memoria (ya que se toman por columnas), hemos tenido que trasponer previamente la matriz `b`, por supuesto contabilizando este cómputo dentro del total.

Pseudocódigo 5: *Instrucciones AVX*

```
register __m512d vec_a, vec_b, vec_c, vec_d, vec_aux;
{...}
vec_c = _mm512_load_pd(c);  // Cargamos el vector c, que ya es de tamano 8
for (i = 0; i < N - N % block_size ; i += block_size) {
  for (j = 0; j < N - N % block_size; j += block_size) {
    for (ii = i; ii < i + block_size; ii++) {
      vec_a = _mm512_load_pd(a + ii * M);
      vec_aux = _mm512_mul_pd(vec_a, vec_c);
          // Calculamos a * (b - c) como a * b - a * c; guardamos a * c en vec_aux
      for (jj = j; jj < j + block_size; jj++)
          {
        vec_b = _mm512_load_pd(b + jj * M);
        vec_d = _mm512_fmsub_pd(vec_a, vec_b, vec_aux);
        d[ii * N + jj] = 2 * _mm512_reduce_add_pd(vec_d);
{...}
```

D. Programa paralelizado con OpenMP

Para la última versión del programa —`versión 4`, que se compiló con opciones de optimización `-O0` y `-O2`, dando lugar a las respectivas versiones `4o0` y `4o2`—, hemos probado interactivamente cada uno de los tipos de planificaciones en el reparto de iteraciones de los hilos, a saber: `static`, `dynamic`, `auto`, `guided` y `runtime` (véase [4], p. 57) obteniendo el mejor resultado usando la planificación `auto`.

Posteriormente, y dado que en la sección donde realizamos el cómputo intensivo tenemos 4 bucles anidados, se han probado la opción de `collapse` con los valores 2 y 4. La mejora con el primer valor resultó notable respecto a no usar esta cláusula; sin embargo, el segundo, valor no presentó resultados concluyentes de mejora frente al primero.

Finalmente, para la prueba del número de hilos hemos ejecutado los programas con `sbatch`. Esta elección se debe a la necesidad de asegurarnos que no existan otros procesos que puedan interferir con los hilos, pues podrían ocasionar un comportamiento anómalo conforme nos acercamos al número máximo de cores del nodo.

III. Resultados

Se presentan a continuación los resultados tras la ejecución de los experimentos. Recordemos que para la versión 2 hemos usado el programa que mejores resultados ha dado, siendo este el obtenido usando operaciones por bloques, con uso de registros y desenrollamiento. Además, en las siguientes gráficas, cada vez que se representan los resultados de alguna de la versión 4 junto a otras versiones, se muestran los resultados de ejecución con 64 hilos, que es la configuración que mejores resultados obtiene, como se puede ver en las gráficas 7 o 5.

Lo primero que se puede obsevar en la gráfica 1, es que la versión secuencial del programa sin ningún tipo de optimizaciones presenta un rendimiento mucho peor que cualquier otra versión del programa, como cabría esperar. Por lo demás, todas las demás versiones

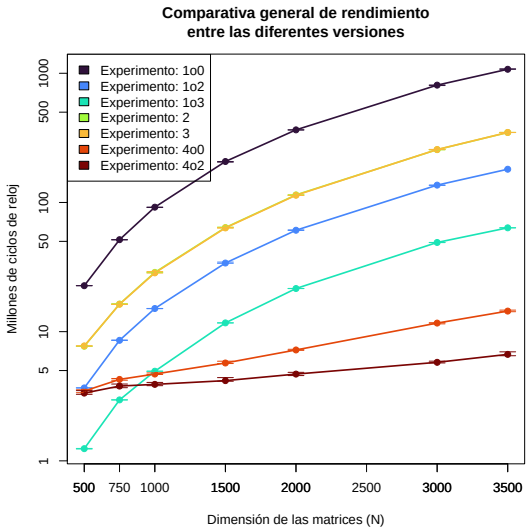

Figura 1: *Comparativa general de los programas*

parecen tener un comportamiento similar, a excepción de las versiones que utilizan múltiples hilos, en las que se observa una diferencia cualitativa en el escalado en función del tamaño de las matrices, siendo que resultarán todavía más eficientes que el resto conforme aumente la cantidad de trabajo a realizar.

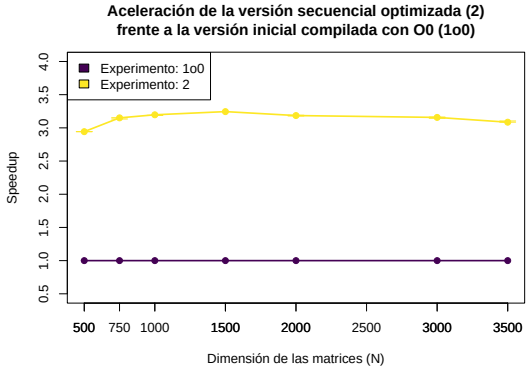

Figura 2: *Aceleración del programa con optimizaciones caché frente al programa base sin optimizar*

Observaciones similares en cuanto al rendimiento se pueden realizar analizando otras gráficas. Así, mientras que en la gráfica 2 se puede ver cómo la versión 2 es unas tres veces más rápida que la versión base sin optimizar, pero de forma consistente en función del tamaño de las matrices, en las gráficas 3 y 4 se puede ver cómo la aceleración obtenida al paralelizar aumenta con la cantidad de trabajo a realizar de forma aproximadamente lineal (nótese que en la gráfica 4 se utiliza escala logarítmica para visualizar mejor las diferencias entre las versiones `1o3` frente a `2` y `3`).

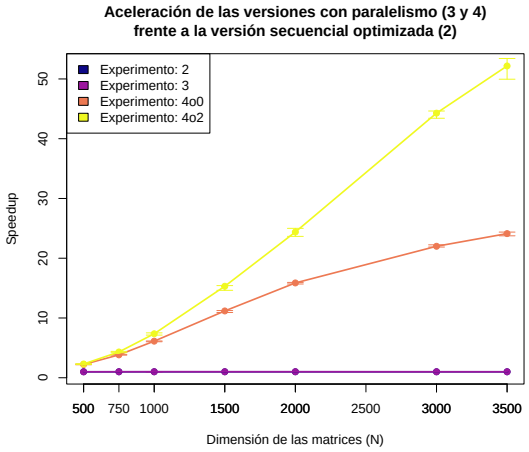

Figura 3: *Aceleraciones de los programas con paralelismo frente al secuencial optimizado*

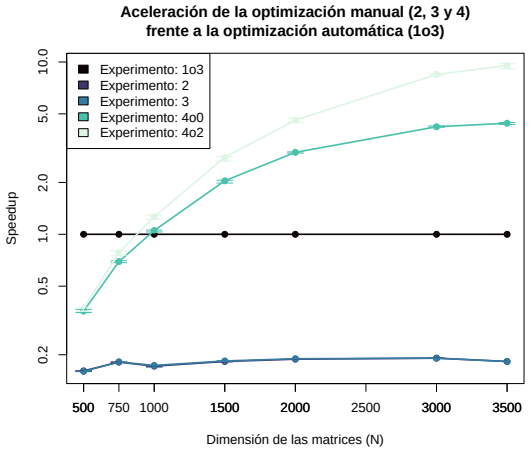

Figura 4: *Aceleraciones de las optimizaciones manuales frente a las automáticas*

Estas diferencias cualitativas se explican por la dificultad asociada a repartir el trabajo a realizar entre los distintos hilos de ejecución. Así, si se cuentan con demasiado hilos pero el trabajo a realizar es poco, se gastará mucho tiempo (proporcionalmente) en repartir el trabajo, y no se notará una aceleración significativa por el hecho de paralelizarlo. Es cuando el trabajo a realizar aumenta que se puede explotar realmente la capacidad de paralelizarlo,

y por eso es que para valores grandes de `N` se aprecia más esta diferencia en rendimiento.

En la gráfica 6 se recoge la eficiencia por hilo (esto es, la aceleración obtenida dividida entre el número de hilos) como medida de este hecho. Se puede ver cómo, para un número grande de hilos, la eficiencia comienza siendo baja en tamaños de matrices pequeños, pero que aumenta conforme lo hace la dimensión del problema. Se ve también que la eficiencia parece tender a un valor estable para cada número de hilos; este valor es significativamente menor que 1, pues, debido a los costes de crear, destruir y sincronizar los hilos, jamás se alcanzará la eficiencia por hilo que se tiene en el programa secuencial. Se puede constatar este comportamiento también en la gráfica 5, donde se ve que la aceleración no crece con el valor de `N` cuando se utilizan pocos hilos, pues ya se ha alcanzado la eficiencia pico.

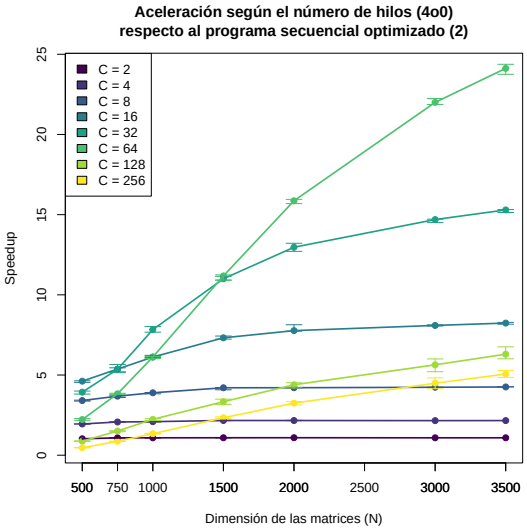

Figura 5: *Aceleración según el número de hilos respecto al programa secuencial optimizado*

En lo relativo al número de hilos en la versión 4, destacar por último el drástico descenso en rendimiento que ocurre cuando superamos el número de hilos que pueden estar en ejecución al mismo tiempo en la máquina —64, en este caso—. A partir de ese punto, añadir más hilos solo aportará el sobrecoste de crearlos y conmutar entre ellos, y provocará un drástico descenso en el rendimiento y en la eficiciencia, como se ve en las gráficas 7 y 6, respectivamente.

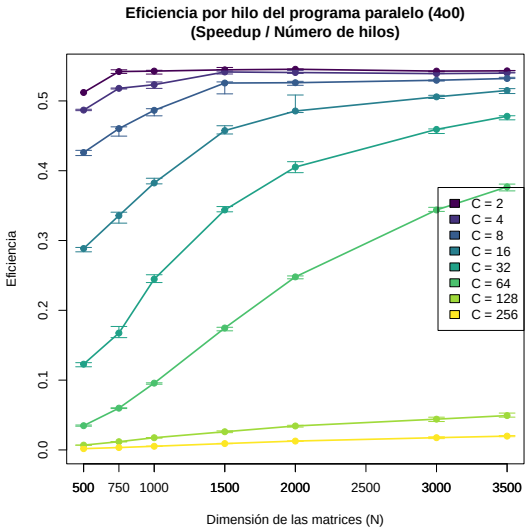

Figura 6: *Eficiencia por hilo*

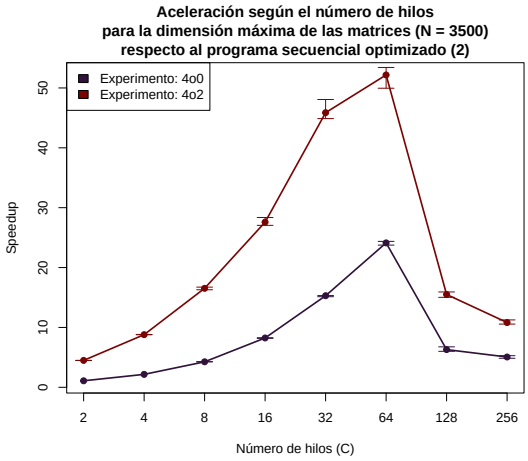

Figura 7: *Aceleración según el número de hilos*

Por otro lado, la versión 3 no parece suponer una mejora respecto a la versión 2 (de hecho, las líneas se encuentran casi totalmente solapadas); como se puede ver en la gráfica 8 para varios valores de `N`, este programa supera en rendimiento al anterior, pero las diferencias son poco consistentes y muy poco significativas. La escasa diferencia entre los resultados obtenidos se debe a que la ganancia que supone operar sobre varios datos a la vez queda opacada por la sobrecarga adicional que supone la carga de los datos en los registros especiales `__m512d`, además de que una de las operaciones realizadas (`reduce_add`) es en realidad secuencial.

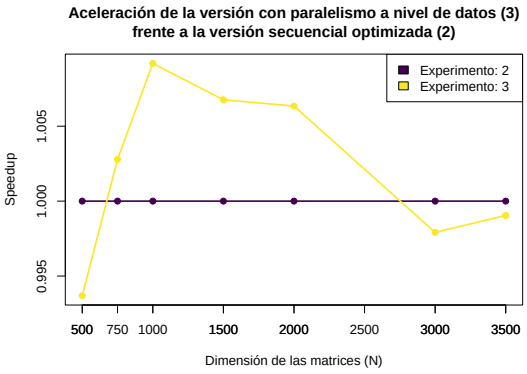

Figura 8: *Aceleración del programa con instrucciones vectoriales frente a la versión secuencial optimizada*

Finalmente, destacar también que, como reflejan las gráficas 4 y 1, el compilador realiza un mejor trabajo optimizando el código 1 de lo que un programador puede alcanzar de forma manual, pues utiliza técnicas ampliamente probadas, muy sofisticadas y adaptadas a la máquina en la que se va a ejecutar.

IV. Conclusiones

Para analizar diferentes estrategias de optimización se ha partido de un código base con cálculos sobre matrices y, a partir de este, se han creado otros 3 programas que hacen uso de optimizaciones del uso de la caché, instrucciones vectoriales y programación paralela multihilo.

Cada una de las optimizaciones caché ha sido probada por separado, con los mejores resultados obtenidos para las técnicas de uso de registros y desenrollamiento. El programa final ha resultado alcanzar el mejor rendimiento con la adicion de operaciones por bloques sobre las otras 2 técnicas.

Partiendo de este código, se han cambiado las operaciones desenrolladas por instrucciones `AVX512`. No se aprecia una mejoría significativa al aplicar estos cambios. Esto se debe al tiempo extra que se necesita para la carga de datos en los registros especiales, lo que contrarresta la ganancia de ejecutar una instrucción sobre varios datos.

Finalmente, se ha paralelizado la ejecución de los bucles mediante el uso de la librería `OpenMP`, obteniendo el mejor resultado con la ayuda adicional de la optimización automática del compilador `-O2`.

Como resultados de este estudio concluimos que el compilador realiza un excelente trabajo de forma automática en la optimización del código y que, por norma general, debería evitarse gastar tiempo en hacer complicadas optimizaciones manuales, pues los resultados obtenidos son peores que con la compilación automática. Además, el compilador trabaja mejor con códigos sencillos y fáciles de entender, por lo que ofuscar el programa resultará en un código menos eficiente cuando se aplique la compilación optimizada. Si el programa soporta un alto grado de paralelismo, como es el caso del código con el que se trabajó aquí, resulta también muy beneficioso aplicar paralelismo a nivel de hilos (por ejemplo, con `OpenMP`), además de usar las optimizaciones del compilador.

Como trabajo futuro, se podría considerar la casuística del programa base paralelizado con la optmizazción `-O3` con el fin de entender mejor el proceso de optimización del compilador.

Referencias

[1] GCC, Compiler Optimizations. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html [online] (última visita 24 de abril de 2024)

[2] Intel, Loop Optimizations. https:// www.intel.com/content/www/us/en/developer/articles/technical/loop-optimizations-where-blocks- are-required.html [online] (última visita 24 de abril de 2024)

[3] Intel, Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html# [online] (última visita 24 de abril de 2024)

[4] OpenMp, Manual. https://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf [online] (última visita 24 de abril de 2024)