Programación Multinúcleo y extensiones SIMD

Aldán Creo Mariño, Hugo Gómez Sabucedo

Laboratorio de Arquitectura de Computadores
Grupo 02
{aldan.creo,hugo.gomez.sabucedo}@rai.usc.es

7 de mayo de 2021

Resumen

En este informe se realizan operaciones aritméticas sobre matrices en punto flotante de diferentes tamaños, con el objetivo de estudiar el rendimiento del programa. Se proponen diferentes optimizaciones del código, para comprobar cuál de ellas es la más eficiente. Los resultados demuestran que la paralelización del código proporciona los mejores resultados, aunque también es importante unna buena optimización interna del código.

Palabras clave: Programación multinúcleo, arquitecturas vectoriales, extensiones SSE3, OpenMP

I. Introducción

El objetivo de este informe es elaborar un programa que, usando matrices con elementos en punto flotante, realice una serie de operaciones con ellas, para tomar medidas del rendimiento del programa variando tanto el tamaño del problema como el tipo de optimización que se utiliza. Cualquier operación en que se utilicen matrices de un tamaño relativamente grande se verá afectada por el tamaño de la memoria caché, pues es ahí en donde se almacenan los elementos para realizar dichas operaciones, y un tamaño grande de matriz implicará que ésta no quepa entera en caché.

Por ello, para obtener unos resultados más eficaces y rápidos, es vital intentar optimizar el máximo al código, bien sea mediante operaciones sencillas como el intercambio o desenrrollo de bucles, o usando otro tipo de soluciones, como pueden ser las extensiones vectoriales o la paralelización del código mediante APIs como OpenMP. Serán este tipo de soluciones las que analizaremos en este informe, para intentar analizar cuál de ellas es la más eficaz y proporciona unos mejores resultados.

En el desarrollo de este informe, explicaremos más en detalle el experimento. En primer lugar, analizaremos las características del sistema y el procesador, para posteriormente explicar qué decisiones se han adoptado para mejorar el rendimiento del código. Para ello, explicaremos distintas técnicas, como el *unrolling*, y el uso de extensiones como SSE o el API OpenMP. A continuación, se expondrán y se interpretarán los resultados obtenidos, mediante la ayuda de distintos gráficos, para los distintos experimento. Por último, se explicarán brevemente las conclusiones finales obtenidas.

II. Descripción del experimento

En esta sección se detallará el proceso que hemos seguido para llevar a cabo las pruebas efectuadas. En primer lugar, se averiguaron las características del procesador, tal y como requería el guión de la práctica. A continuación, se ha implementado en lenguaje C el pseudocódigo proporcionado, de 4 formas distintas: una versión sin ningún cambio, y tres optimizadas de diferentes formas: dividiendo la matriz en submatrices y realizando

operaciones por bloques, usando extensiones vectoriales SSE3 y utilizando OpenMP.

A. Características del sistema

Hemos ejecutado las pruebas en un Intel Core i5 1038NG7, de 4 núcleos, sobre Ubuntu 20.04. El tamaño de página virtual es de 4 KB (obtenido con `getconf PAGESIZE`). La información de la caché, importante para los apartados posteriores, se ha obtenido con `cd /sys/devices/system/cpu/cpu0; grep '' cache/*/*`:

- Memoria caché L1 de datos
  - Tamaño de línea: 64 B
  - Número de conjuntos: 64
  - Número de vías: 12
  - Tamaño total: 48 kB
- Memoria caché L1 de instrucciones
  - Tamaño de línea: 64 B
  - Número de conjuntos: 64
  - Número de vías: 8
  - Tamaño total: 32 kB
- Memoria caché L2
  - Tamaño de línea: 64 B
  - Número de conjuntos: 1024
  - Número de vías: 8
  - Tamaño total: 512 kB
- Memoria caché L3
  - Tamaño de línea: 64 B
  - Número de conjuntos: 8192
  - Número de vías: 12
  - Tamaño total: 6144 kB

B. Versión secuencial base

Para la realización del código de este apartado, simplemente se ha implementado en C el pseudocódigo del guion.

C. Versión secuencial optimizada

Para implementar el código de este apartado, se ha partido del realizado en el apartado B. El objetivo de este apartado es realizar una

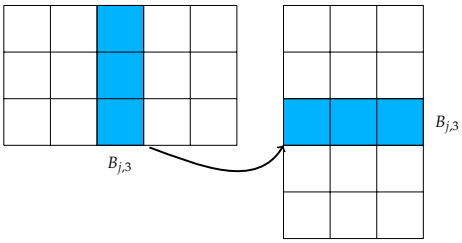

Figura 1: B^T

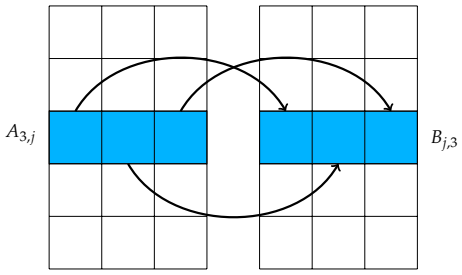

Figura 2: *Multiplicación de A por* B^T

modificación en la parte del código dedicada a la computación, de forma que, obteniendo el mismo resultado final de `f`, se reduzca el tiempo de ejecución. Para ello, hemos introducido varias mejoras, recogidas a continuación. Parte de las mismas están basadas en [4].

La opción por la que nos hemos decantado es una combinación del *unrolling* (tanto desenrrollando lazos como realizando operaciones por bloques) con la división de las matrices en submatrices más pequeñas, para realizar las operaciones sobre ellas.

Hemos decidido trasponer la matriz B para poder acceder a las columnas como si fueran filas, para realizar el producto matricial. Gráficamente, supone realizar el cambio de la figura 1.

Acceder a las columnas de la matriz como si fueran filas es mucho más eficiente porque permite al procesador hacer un mejor trabajo de precarga, dado que los datos accedidos tienen localidad secuencial. El proceso de cálculo del producto matricial se transforma, por tanto, al recogido en la figura 2.

Además, hemos cambiado el algoritmo, para tratar de mejorar su eficiencia. Uno de los cambios que hicimos fue restarle a B el vector

C antes de proceder al producto matricial. De esta forma, sólo tenemos que realizar una resta por cada elemento de B, para precomputar el valor adecuado de B. Por tanto, la complejidad de la resta del vector C es $O(N)$. De hacer el cálculo en el bucle de la multiplicación, tendría una complejidad $O(N^2)$, ya que realizaríamos el producto de cada elemento de B N veces (una por cada fila de A).

En el bloque de código que computa el producto, hemos aplicado el acceso por bloques, para incrementar la localidad de los accesos a memoria y que quepan en la caché. La idea es recorrer repetidamente una misma región de A y B y calcular los productos correspondientes a ellas, antes de seguir calculando otros bloques de la matriz resultado. Nuestra idea ha sido trabajar con conjuntos de X líneas de A y B, en los que haremos multiplicaciones de cada línea de A por cada línea de B, reduciendo las penalizaciones por fallos caché. El acceso por bloques podría implementarse de varias formas, y nosotros nos hemos decantado por esta, ya que no hay una claramente superior: cada posible implementación desaprovecha obligatoriamente parte de los datos, sean éstos de A o de B, ya que debemos realizar todas las posibles combinaciones fila-columna.

Concretamente, hemos calculado el tamaño de los bloques para que quepan aproximadamente en la caché L1. Teniendo en cuenta que un `double` y un puntero ocupan 64 b en nuestra máquina, y un `int` 32 b, calculamos que en un momento cualquiera el conjunto de trabajo del algoritmo será:

- Variables de control: `block_a`, `block_b`, `i`, `j`, `i_max`, `j_max`, `N`. ($32\text{b} * 7 = 224\text{b} = 28\text{B}$).
- Punteros: `a`, `b`, `d`, `lineaA`, `lineaB{0...9}`, X punteros a filas de A, X punteros a filas de B, X punteros a filas de D. ($64\text{b} * (14 + 3 * X)$).
- Valores: `elem{0...9}` , $8 * X$ elementos de A, $8 * X$ elementos de B, X^2 elementos de D. ($64\text{b} * (10 + 8 * 2 * X + X^2)$).

En base a las expresiones planteadas, resolviendo para que el tamaño total de los datos sea aproximadamente 48 kB, el tamaño de la caché de datos L1, obtenemos que X debe estar en torno a 20. Por tanto, ese es el tamaño que elegimos de bloque, definido como constante en el código.

Además, hacemos desenrollamiento del lazo de cálculo, de dos en dos. Otro cambio que hacemos con respecto al algoritmo original es hacer la multiplicación por 2 sólo una vez, al final. Esto es válido por la propiedad distributiva de la multiplicación. Adicionalmente, podría descartarse la multiplicación, porque el único uso que se le da a la matriz D es el cálculo de `f`, y al calcular su valor, dividimos los elementos de D por 2, con lo cual anulamos la primera operación. Pese a todo, sí hemos decidido mantener esta operación, ya que entendemos que se debe calcular el mismo valor para la matriz D que en el apartado anterior. Precisamente, si sólo importase el valor final de `f`, también se podría omitir el cálculo de cualquier elemento de D que no esté en su diagonal, ya que `f` se calcula en base a ella.

En el bucle de cálculo del valor de `f`, hacemos desenrollamiento de 10 en 10. Vamos acumulando los elementos de la diagonal de D en `f`, según el patrón de acceso aleatorio generado en `ind`. Es relevante comentar que en ninguno de los casos en los que hacemos desenrollamiento se comprueba que estemos superando el límite del bucle. Esto es porque hemos seleccionado divisores enteros de los tamaños de N que se nos ha pedido probar, como factores para hacer el desenrollamiento. No realizamos la comprobación para obtener un mejor rendimiento, pero esto también implica que nuestro código falle con valores de N diferentes a los que se nos ha pedido probar. Entendemos que este aspecto es irrelevante, ya que lo que buscamos es el máximo rendimiento para las condiciones que se nos han pedido.

Hasta este punto, la optimización en el rendimiento de nuestro código era notable, pero no dramática. Por eso decidimos analizar el código compilado en este apartado usando Compiler Explorer[1], en comparación con el generado en la subsección B, con la opción de compilación `-O2`, que ofrecía unos muy

buenos resultados. Lo que observamos era que nuestro código repetía muchas veces ciertos cálculos. Por ejemplo, al hacer la multiplicación `a[i][0] * b[j][0]`, `a[i]` y `b[j]` son valores que no cambian en todo el lazo, pero que con la optimización desactivada (`-O0`), se recalculan cada vez. Lo que hicimos para resolver esta cuestión fue guardar `a[i]` en `b[j]` en variables separadas, de forma que evitemos recalcularlas constantemente. El incremento en rendimiento fue muy notable. Esta misma estrategia la aplicamos también en el primero de los bucles. Además, una optimización adicional fue acceder a los elementos de las líneas de la matriz A alternativamente, haciendo una especie de "precarga": cargamos el elemento siguiente al que vamos a acceder cuando aún estamos calculando la línea anterior. Esto produjo mejoras sensibles en el rendimiento.

D. Versión secuencial con procesamiento vectorial SIMD

Para este apartado, hemos tomado como referencia el código de la subsección C. Hemos vectorizado los bucles de cálculo del código original usando instrucciones SSE3. Para ello hemos usado la guía de Intel [2].

Las funciones principales que hemos usado en este apartado son `_mm_load_pd`, para cargar elementos de memoria a un registro vectorial; `_mm_store_pd`, para guardar en memoria los elementos de un registro; y `_mm_add_pd` , `_mm_sub_pd` y `_mm_mul_pd`, para realizar operaciones de suma, resta, multiplicación y división, respectivamente, entre dos registros vectoriales. Como trabajamos con `doubles`, podemos trabajar solamente con registros de dos valores (128 b en total).

El primer bucle de nuestro código, el que calcula la resta de las columnas de la matriz B con el vector C, lo hemos vectorizado recurriendo a dos bucles para evitar rehacer cálculos innecesariamente. En el lazo externo, cargamos primero (C_0, C_1), y en el lazo interno, se lo vamos restando a $(B_{0,0}, B_{1,0})$, $(B_{0,1}, B_{1,1})$ ... $(B_{0,N}, B_{1,N})$. Después, cargamos (C_2, C_3), y repetimos el proceso para las dos siguientes filas. El proceso se repite hasta finalizar la resta de toda la matriz B.

El siguiente bucle es el que se encarga de realizar los cálculos del producto matricial. Es una parte muy extensa del código, principalmente por el unrolling que hemos mantenido del apartado anterior (con factor 10). Probamos a cambiar el factor a 5, porque un factor 10 implica exceder el máximo de registros vectoriales de nuestra máquina (16), lo cual obliga a que el compilador almacene algunos de los registros en memoria. Pese a todo, el rendimiento experimental caía ligeramente con un factor 5, así que decidimos mantener un factor de 10. Al mantener el factor de desenrollamiento, no es necesario recalcular el tamaño de bloque, por lo que lo mantenemos en 20. Al igual que en el apartado anterior, recorremos en bloques las matrices A y B. Al realizar los cálculos dentro de un par concreto de bloques, los pasos que seguimos son:

1. Precalculamos las direcciones a las que vamos a acceder para ahorrarnos cálculos después (`lineaA`, `lineaB0` ... `lineaB9`).
2. Cargamos en un registro los dos primeros elementos de la línea de A, y en otros 10 registros, las líneas de B.
3. Guardamos en registros temporales los resultados de hacer la multiplicación de los dos elementos de la línea de A, y los dos correspondientes de las 10 líneas de B.
4. Sumamos a un registro de acumulación el resultado de la multiplicación.
5. Al acabar de recorrer las líneas por completo (4 iteraciones), recuperamos el contenido de cada registro de acumulación. Para ello, sumamos entre sí los dos valores del registro vectorial, usando un vector auxiliar. El resultado lo multiplicamos por 2, y lo almacenamos en la matriz D. Este es un "cuello de botella" inevitable, ya que debemos combinar obligatoriamente ambos valores.

Finalmente, realizamos el cómputo del valor de `f`. Los cálculos son más sencillos, y en

este caso, podemos mantener el factor de desenrollamiento de la subsección C, ya que usamos menos registros dentro del bucle. Pese a todo, la vectorización de este bucle es muy ineficiente, porque las operaciones se hacen con elementos de la diagonal de D, que están dispersos en memoria. Esto implica que no se puede realizar la carga en registros vectoriales de los datos de forma directa, sino que antes de ello debemos cargarlos contiguamnete en un mismo vector auxiliar del tamaño de dos doubles, para después cargarlos en el registro en sí. Dentro del bucle, lo único que hacemos es dividir ambos elementos por dos, y acumularlos en un registro vectorial. Al acabar el bucle, recuperamos los dos elementos del registro de acumulación, y los sumamos entre sí para obtener el valor de `f`.

E. Versión paralelizada usando OpenMP

El objetivo de esta subsección es mejorar el código realizado en el subsección C, paralelizándolo mediante el uso de directivas OpenMP.

Las dos directivas principales de OpenMP que usaremos serán `#parallel` y `#for`. La primera nos permite crear una región parelela; es decir, el código que se incluya dentro de esta directiva será repetido por todos los hilos. En estas regiones se pueden definir variables tanto privadas (diferentes para cada hilo) como compartidas (comunes a todos los hilos). La segunda de las directivas, `for`, nos permite repartir las iteraciones de los bucles entre los distintos hilos. Por último, otra directiva que se ha usado es `#pragma omp atomic`, que nos permite asegurarnos de que una dirección memoria se actualiza atómicamente, evitando que se produzcan escrituras o lecturas simultáneas por parte de varios hilos.

El primero de los bucles se traduce de forma bastante directa. Llega con especificar las variables que son privadas a cada hilo.

El bucle de computación principal podría paralelizarse de distintas formas, pero hemos decidido paralelizarlo al nivel del bucle `for` más externo, ya que nuestra máquina tiene pocos núcleos, y por tanto sólo se beneficiará del paralelismo de unos pocos hilos a la vez. En el bucle externo tenemos un punto de separación (*fork*) que consideramos adecuado, ya que cada hilo se encargará de realizar todos los cálculos anidados interiormente. No necesitamos usar directivas de sincronización dentro de las regiones paralelas, ya que todos los bucles modifican posiciones diferentes de la matriz D. Las variables usadas internamente en el bucle, eso sí, se declaran como privadas, ya que podrían ser fuente de carreras críticas.

Finalmente, en el bucle que computa `f`, lo que hacemos es mantener el desenrollamiento con factor 10, y vamos acumulando en una variable `acumulador` las sumas de los elementos de la diagonal de D.

En el guion de la práctica se indica que está prohibido usar variables tipo "reduction". Entendemos la prohibición se aplica al `reduction` de OpenMP, ya que para calcular `f` es imprescindible realizar una operación de acumulación de los elementos de la diagonal de D. debemos emplear la directiva de sincronización `atomic` para incrementar `f`, ya que es una variable compartida que modifican todos los hilos, y por tanto podría sufrir carreras críticas.

El número de hilos lo establecemos como variable de entorno en el momento de la compilación (`OMP_NUM_THREADS`).

III. Resultados e interpretación

En esta sección analizaremos los resultados obtenidos en base a los experimentos realizados. Hemos repetido cada experimento 20 veces, y hemos seleccionado la mediana del tiempo de ejecución, para evitar anomalías estadísticas.

En la figura 3 podemos observar el número de ciclos de reloj que ha necesitado cada uno de los programas, en sus diferentes versiones (por ejemplo, el programa secuencial base compilado con `-O0` y `-O2` o el programa

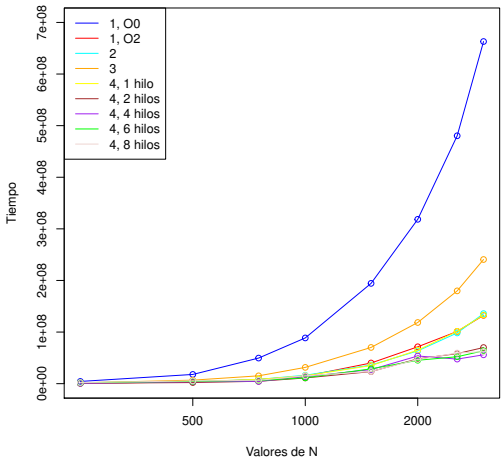

Figura 3: *Número de ciclos para las distintas versiones*

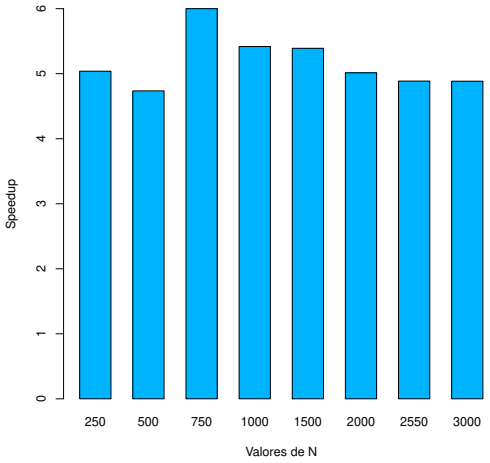

Figura 4: *Speedup de la versión optimizada respecto a la secuencial con* `-O0`

con OpenMP para los 5 valores diferentes del número de hilos), para los distintos tamaños de las matrices. Como cabría esperar, cuanto mayor es el tamaño de la matriz, mayor es el número de ciclos de reloj, independientemente de la optimización que se use.

En este caso, el experimento que ha obtenido unos peores resultados ha sido el código secuencial base, compilado con `-O0`. Esto es esperable, ya que el código no es optimizado ni por el compilador ni por el programador. Por otra parte, el experimento más eficiente, con un menor número de ciclos, es el experimento con OpenMP, usando múltiples hilos (analizaremos las diferencias entre el número de hilos más adelante).

Por otra parte, se nos pedía representar la ganancia en velocidad (*speedup*) de distintos programas en varias gráficas distintas. Comencemos por el análisis del *speedup* de la versión secuencial optimizada con respecto a la versión inicial compilada con `-O0`. De esta representación, que podemos ver en la figura 4, se puede concluir que la optimización que se ha realizado ha sido excelente, pues nuestro código optimizado realiza los cálculos unas 5 veces más rápido que el código secuencial sin optimizaciones.

Además, el código optimizado por nosotros se equipara al optimizado por el compilador

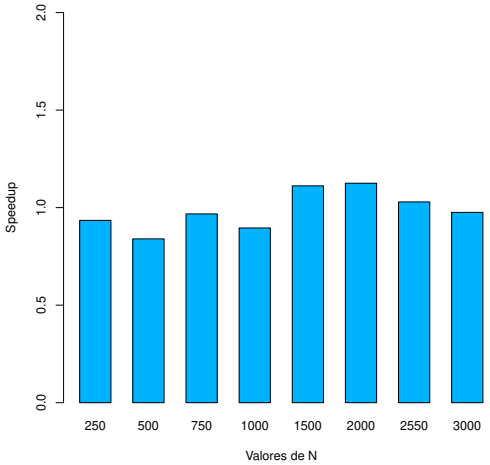

Figura 5: *Speedup de la versión optimizada respecto a la secuencial con* `-O2`

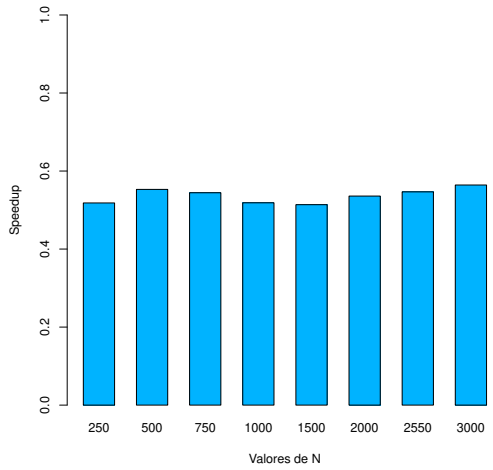

Figura 6: *Speedup de la versión vectorial respecto a la optimizada*

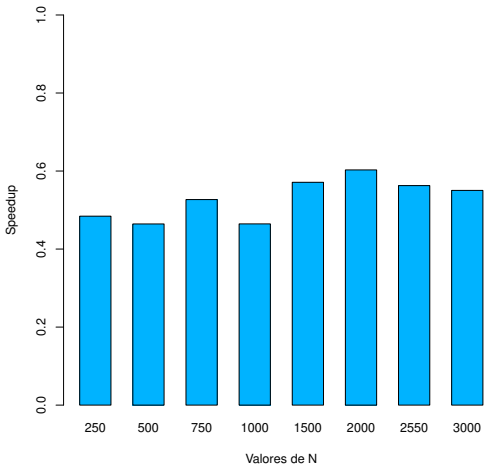

Figura 7: *Speedup de la versión vectorial respecto a la secuencial con* `-O2`

(`-O2`), como vemos en la figura 5. Los valores del *speedup* están en torno a 1, indicando que el consumo de tiempo es más o menos el mismo. Consideramos que este es un gran logro, ya que `gcc` está diseñado para aplicar estrategias muy complejas y avanzadas de optimización del código, y nuestras optimizaciones han conseguido los mismos resultados.

La siguiente gráfica que se pedía realizar era la ganancia en velocidad del código de la subsección D, de la sección anterior, respecto a la versión secuencial optimizada. Esta representación se puede observar en la figura 6. Es destacable que el *speedup* es menor que 1. Lo mismo ocurre con la figura 7, que lo compara con el código de la subsección B compilado con optimización automática (`-O2`). En esa gráfica, parece haber cierta variabilidad, pero sólo es porque estamos usando una escala reducida. Si tenemos en cuenta la escala, podemos ver que el *speedup* se mantiene más o menos constante.

Ya que la optimización con `-O2` suponía un tiempo de ejecución aproximadamente igual al que proporcionaba nuestra optimización, el *speedup* es también menor que 1. Esto indica que la versión vectorial requiere más ciclos de ejecución que las versiones optimizadas, lo cual es previsible por los motivos que comentaremos a continuación. Pese a que los resultados son mejores con respecto a la versión secuencial sin optimización, la relativa mejora en velocidad se debe únicamente a que partíamos del código optimizado por nosotros. Realmente, aplicar vectorización en este caso es contraproducente, ya que existe un *overhead* demasiado grande por el uso de las extensiones vectoriales.

Principalmente, la pérdida de velocidad se debe a que en nuestro código realizamos operaciones relativamente simples, con muchos valores. SSE3 sólo nos permite trabajar con dos `doubles` a la vez, y por tanto, el máximo beneficio teórico que podríamos conseguir sería un speedup de 2, teniendo sólo en cuenta las operaciones vectorizadas. El problema es que, por cada operación que realizamos, en general, tenemos que empezar llamando a una función que cargue los valores adecuados en un registro vectorial, luego llamar a otra que compute un resultado, y finalmente llamar a otra que almacene el resultado. Realizar todas estas llamadas añade un sobrecoste que es superior a la ganancia en velocidad de vectorizar las operaciones. Además, por la propia naturaleza del algoritmo implementado, existen "cuellos de botella", que comentamos en el análisis del código. Aumentar el paralelismo sería posible usando otro tipo de extensiones vectoriales, que permiten registros de hasta 512 b, o usando otro tipo

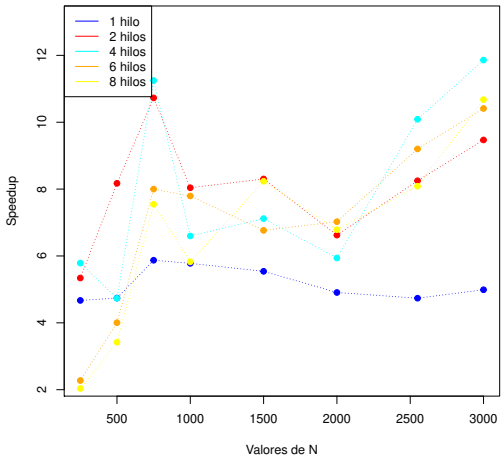

Figura 8: *Speedup de la versión con OpenMP respecto de la versión secuencial compilada con -O0, variando el número de hilos*

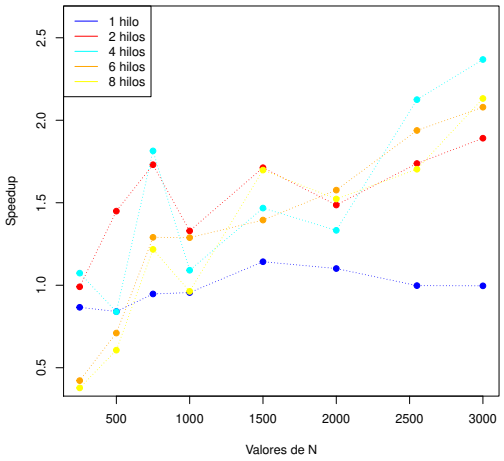

Figura 9: *Speedup de la versión con OpenMP respecto de la versión secuencial compilada con -O2, variando el número de hilos*

de datos más pequeños, para poder trabajar con más a la vez. Por ejemplo, si el tipo de datos de la matriz fuesen `shorts`, podríamos realizar los cálculos con líneas completas, en vez de tener que realizar 4 iteraciones por cada línea, ya que sólo podemos trabajar con 2 `doubles` a la vez. También conviene recordar la dispersión en memoria de algunos datos, que impide el uso eficiente de extensiones vectoriales, que comentamos al hablar del bucle de cálculo de `f`.

La gráfica que se presenta en la figura 8 nos muestra la aceleración obtenida por el programa optimizado usando OpenMP, para los diferentes números de hilos y valores de N, respecto a la versión secuencial base compilada con `-O0`. Se ha optado por una representación de puntos, pero unidos con una línea, para poder observar mejor los valores para cada número de hilos.

El caso en que se usa únicamente 1 hilo es el que nos proporciona una peor aceleración, situándose en torno al 0'4 o 0'5. Al usar un único hilo, no se está paralelizando de ninguna manera el trabajo (que, como se explicó en el apartado E, es el objetivo de OpenMP), y por eso obtenemos un resultado equivalente al de la subsección C.

Los valores presentan una alta variabilidad en tamaños pequeños, pero no debe ser tenida en cuenta, ya que se debe a que el tiempo de ejecución para esos tamaños es reducido, y por tanto, variaciones insignificantes en las mediciones producen grandes cambios en el *speedup*. De forma complementaria a esta gráfica, conviene recordar la figura 3, donde se aprecia claramente lo bajos que son los tiempos de ejecución en esos tamaños (y que ayuda a entender por qué la gráfica varía tanto). Por esto, vamos a realizar nuestro análisis en base a los valores de tamaños grandes, que sí evolucionan de forma coherente.

El caso que ha proporcionado, en general, los mejores resultados, es aquel en el que usamos 4 hilos, obteniendo un *speedup* de hasta 12 en comparación con el código sin optimizar.

Para completar la explicación es conveniente fijarnos en la figura 9, que compara el *speedup* con la versión compilada con `-O2` (que es aproximadamente el mismo que con el código de la subsección C). En este caso, lo que vemos es que la paralelización del código con OpenMP es la que devuelve los mejores resultados, siendo dramáticamente superiores a los del código sin optimizar, y notablemente mejores que los del código optimizado. Esto es razonable, ya que se ejecutan las mismas

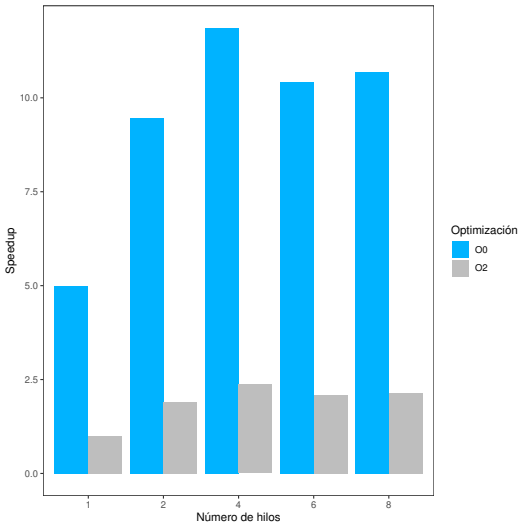

Figura 10: *Speedup de la versión con OpenMP respecto de la versión secuencial con `-O0` y `-O2`, variando el número de hilos, para el mayor tamaño de N*

operaciones que en el código optimizado, añadiendo la mejora de la paralelización.

A partir de 4 hilos, el *speedup* no parece mejorar. Entendemos que esto se debe a las características de nuestra máquina, ya que pese a que nuestro procesador soporta *hyperthreading* [3], lo cual permite hasta un máximo de 8 hilos en ejecución simultánea, estamos ejecutando las pruebas en una máquina virtual que probablemente no permita una visión del procesador en la cual se permita aprovechar este factor. Es decir, la máquina virtual observa un procesador simple, sin *hyperthreading*, que sólo permite 4 hilos simultáneos. Por eso se degrada el rendimiento más allá de los 4 hilos.

La última gráfica requería que se representase el *speedup* conseguido por el código optimizado mediante OpenMP respecto a la versión secuencial base, en una gráfica separada, para el valor más grande de N (3000), variando el número de hilos. En nuestro caso, en la figura 10, hemos representado el *speedup* obtenido compilando con `-O0` y con `-O2`.

En ambos casos se han obtenido *speedup*s mayores que 1, a excepción del programa compilado con `-O2` para 1 hilo. Esto es totalmente lógico, ya que no se está paralelizando nada, y por tanto el rendimiento es comparable al del código optimizado. A medida que aumentamos el número de hilos, obtenemos resultados mejores, hasta llegar a 4 hilos. También cabe destacar que, compilando con `-O2`, obtenemos un *speedup* menor que compilando con `-O0`. Esta situación es totalmente lógica, por los motivos explicados antes (tenemos un *speedup* menor, al comparar con un código más rápido).

IV. Conclusiones finales

En el desarrollo de este informe hemos estudiado cómo mejorar el tiempo de ejecución y el número de ciclos de CPU necesarios para ejecutar un programa que computa una serie de operaciones sobre matrices de diversos tamaños en punto flotante.

Para ello, partiendo de una versión secuencial simple, se han aplicado diversas técnicas, como el desenrollamiento de lazos, el acceso por bloques, la precomputación de resultados, o la redistribución de datos en memoria para incrementar la localidad. A continuación, se han empleado extensiones vectoriales (SSE3), para vectorizar la computación. Por último, se ha paralelizado el código usando OpenMP, con distintos valores para el número de hilos.

Una vez se han realizado todos los experimentos y tras obtener los resultados correspondientes, estos se han analizado e interpretado. Se ha representado gráficamente el tiempo de computación para cada uno de los tamaños de matrices y tipos de optimización. Además, se ha representado el *speedup*, o ganancia en velocidad, de los diferentes códigos entre sí, para poder analizar más fácilmente cuáles son más eficientes, y cuáles lo son menos.

En base a los resultados, se ha observado que la mejor manera de ejecutar eficientemente este programa es mediante el uso de directivas OpenMP, que permitan paralelizar el código. Dentro de las directivas OpenMP, aunque todos los resultados obtenidos han sido generalmente mejores que los obtenidos en los otros apartados, se puede concluir que el

mejor se tiene cuando usamos un número de hilos igual al número de núcleos del computador, aunque puede variar dependiendo de las características de la máquina.

Los códigos optimizados automáticamente (`-O2`) y manualmente (subsección C), empatan en rendimiento. Esto demuestra que la optimización que hemos realizado es muy exhaustiva.

El código vectorizado usando SSE3 es peor que el de la versión optimizada manualmente, pero aún así sigue siendo mejor que el código sin optimizaciones. Los motivos principales son el *overhead* y la incompatibilidad de partes del algoritmo con la vectorización de los cálculos.

La peor versión de todas es la versión secuencial básica sin optimizaciones (`-O0`), como cabe esperar.

En conclusión, podemos afirmar que las técnicas de paralelización pueden ofrecer resultados muy positivos, pero no siempre (el caso de SSE3). Es recomendable aplicar siempre estrategias para optimizar el código, como las ya descritas, especialmente en cálculos como los que hemos desarrollado. También es recomendable recurrir a las mejoras que ofrecen las optimizaciones del compilador. Pero, sin duda, la mejor manera de implementar este código sería realizar una operación distinta sobre los elementos de la matriz, de forma que se accediese a ellos con más localidad y más eficientemente, pudiendo incluso no ser necesario realizar tantas optimizaciones como ha sido el caso de este informe.

Referencias

[1] Compiler explorer. `https://godbolt.org`. En línea.

[2] Intel intrinsics guide. `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`. En línea.

[3] What is hyper-threading? `https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html`. En línea.

[4] James Demmel. Lecture 2: Memory hierarchies and optimizing matrix multiplication. `https://people.eecs.berkeley.edu/~demmel/cs267_Spr99/Lectures/Lect_02_1999b.pdf`. En línea.