Programación Multinúcleo y extensiones SIMD

Alex Baquero Domínguez, David Bernardez Fernández

Arquitectura de Computadores
Grupo I
{alex.baquero, david.bernardez.fernandez}@rai.usc.es

***Resumen* — Análisis del rendimiento de un algoritmo simple con matrices en punto flotante, combinando diferentes grados de optimizaciones y tamaños de problema. Entre estas optimizaciones encontramos el manejo de varios núcleos, uso de extensiones vectoriales SIMD, estrategias para reducir los fallos caché, empleo de diferentes opciones de compilación y utilización de la interfaz OpenMP.**

***Palabras clave* — Rendimiento, paralelismo, matrices, extensiones vectoriales, optimización, OMP….**

I. Introducción

Este trabajo se basa en los experimentos realizados en las secciones III,IV,V y VI, en las cuales se pretende comprobar si cuatro códigos con las mismas matrices en punto flotante consumen un menor número de ciclos al intercalar distintos tamaños del problema y optimizaciones.

A partir del manejo de unos programas en C que realizan la computación de un vector de salida, y utilizando un pseudocódigo de partida, estos estudios consisten en medir el número de ciclos (10 medidas) para cada caso y seleccionar la mediana de estos valores como valor final que represente el tiempo de ejecución.

IMAGEN I
PSEUDOCÓDIGO INICIAL

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 indices 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
```

En todos los casos se hace una reserva de memoria dinámica de las matrices y vectores, los cuales se inicializan con valores aleatorios en un intervalo prefijado : *[0, rand()/2]*.

El objetivo es variar el número N de filas y columnas de la matriz en punto flotante (N=250, 500, 750, 1000, 1500, 2000, 2550 y 3000) y utilizar diferentes grados de optimización del algoritmo simple para finalmente tomar medidas de rendimiento. En todas estas versiones, se ha asegurado de que el resultado final es el mismo, es decir, que todas las versiones realizan la computación correctamente.

La optimización de este problema inicial se interpretará mediante distintas gráficas obtenidas al introducir extensiones vectoriales SIMD, estrategias para reducir los fallos caché, la interfaz OpenMp así como distintas opciones de compilación.

Este informe queda estructurado de la siguiente manera:

II. Características del procesador utilizado.
III. Caso de estudio 1.
IV. Caso de estudio 2.
V. Caso de estudio 3.
VI. Caso de estudio 4.
VII. Resultados:
- A. Ganancia en velocidad de los códigos del caso de estudio I con respecto al II (compilado con -O0)
- B. Ganancia en velocidad de los códigos del caso de estudio III y IV con respecto al OO (compilado con -O0)
- C. Ganancia en velocidad de los códigos del caso de estudio I (compilado con -O3) con respecto al resto de casos.
- D. Ganancia en velocidad conseguida para los diferentes números de hilos variando el valor de N en el caso de estudio IV.
- E. Comportamiento para el valor más grande de N en el caso de estudio IV

VIII. Conclusiones.

En la sección II se dará a conocer la jerarquía de memoria caché del procesador utilizado en detalle. En los apartados III, IV, V y VI se presentará el código de programación C de los ejercicios 1,2,3 y 4 respectivamente, y para concluir, en la sección VI y VII se presentarán e interpretarán los resultados obtenidos.

II. CARACTERÍSTICAS DEL ORDENADOR

En este apartado se analizará el procesador que se utilizará en la experimentación, necesario para entender correctamente los resultados obtenidos.

Se puede averiguar esta información a través de diferentes comandos en la terminal de linux:

- *hardinfo* permitirá conocer el tamaño de las distintas cachés, además de su tipo (asociativa por conjuntos en este caso) y el número de vías.
- *getconf -a,* con el cual se podrá visualizar, entre otras características, el tamaño de las líneas de cada memoria caché.
- *lscpu* mostrará la arquitectura del procesador, dato con el que se podrá saber también el tamaño de palabra del sistema.

Después de ejecutar estos comandos, la información obtenida es la siguiente:

CUADRO 1

JERARQUÍA DE MEMORIA USADA

MODELO	INTEL CORE i5-8250u
NÚCLEOS	4
ARQUITECTURA	x86_64
PALABRAS	64 bits
NIVELES CACHÉ	3
L1	32KB
L2	256KB
L3	6144KB
TAMAÑO DE LÍNEA	64 bytes

Como se puede observar, en el **Cuadro 1**, el procesador de la máquina utilizada, es un Intel Core i5-8250u de 4 núcleos con una arquitectura x86_64 que utiliza palabras de 64 bits. Cuenta con 3 niveles de caché, el primero y segundo son propios de cada núcleo, mientras que el último es compartido por todos ellos. Los tamaños son de 32KB y 256KB para L1 y L2 (cada uno de los 4 núcleos) y 6144KB para la caché compartida L3.

III. CASO DE ESTUDIO 1

En esta sección se presentará en qué consiste el primer programa utilizado.

A través del pseudocódigo inicial, se ha programado un código C que consiste en un algoritmo simple con matrices en punto flotante que realizan la computación de un vector de salida.

En primer lugar, para declarar las matrices se utiliza un doble puntero **(Imagen 2).** Este doble puntero apuntará a un array de punteros, que contendrán las filas de la matriz. Será necesario realizar dos mallocs para cada matriz. En el caso de los vectores sólo será necesario una única asignación de memoria.

IMAGEN 2

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 1

```
81
82      a = (double**) malloc(N * sizeof (double*));
83      b = (double**) malloc(8 * sizeof (double*));
84
85      d = (double**) malloc(N * sizeof (double*));
86
87      for (int i = 0; i < N; i++) {
88          d[i] = (double*) malloc(N * sizeof (double));
89          a[i] = (double*) malloc(8 * sizeof (double));
90      }
91
92      for (int i = 0; i < 8; i++) {
93          b[i] = (double*) malloc(N * sizeof (double));
94      }
95
96      ind = (int*) malloc(N * sizeof (int));
97      c = (double*) malloc(8 * sizeof (double));
98      e = (double*) malloc(N * sizeof (double));
99      usados = (int*) malloc(N * sizeof (int));
```

Una vez declaradas las distintas matrices y vectores, será preciso inicializar cada uno de ellos. Para esto, se utilizan bucles *for* que recorran todas sus posiciones.

En el caso de las matrices *a* y *b* y del vector *c*, se inicializan con valores aleatorios utilizando la función *rand()* y dividiendo su valor entre dos.

Para el vector ind se utilizan *rand() % n* para que tome solo valores entre 0 y N, y un vector auxiliar *usados* para no repetir posiciones.

El código principal se observa en la siguiente imagen. **(Imagen 3).**

IMAGEN 3

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 1

```
    ////////////////////
    start_counter();

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            d[i][j] = 0;
        }
    }

    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]);
            }
        }
    }

    for (int i = 0; i < N; i++) {
        e[i] = (d[ind[i]][ind[i]]) / 2;
        f += ((double) e[i]);
    }

    ck = get_counter();
    ////////////////////
```

En este Imagen se sigue el modelo del pseudocódigo:

1. Se inicializa la matriz *d* en todas sus filas y columnas.
2. Se realizan una operación de vectores y matrices, que se van almacenando en *d*.
3. Se guardan en el vector *e* los elementos de la diagonal de *d* (divididos entre dos) de forma desordenada.
4. Finalmente se va acumulando en f, que es el valor que se imprime por pantalla.

IV. CASO DE ESTUDIO 2

En este apartado se realizará un programa secuencial optimizado. Para ello, se modificará el código inicial (caso de estudio 1) de modo que se obtenga el mismo vector resultado pero que se reduzca el tiempo de ejecución.

Para mejorar el rendimiento del programa se han realizado una serie de optimizaciones de forma manual. Entre ellas, el desenrollamiento de lazos (aplicado en lo referente a la variable k) y el uso de la fusión de núcleos.

IMAGEN 4

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 2

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

            d[i][j] = 0;

            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]);

        }

    }
```

Como se puede apreciar en la **Imagen 4** el bucle interno ha desaparecido totalmente sustituyendo los valores de la variable k por los números del 0 al 7, evitando que el programa la ejecución de un tercer for.

La fusión simplemente consiste en introducir la inicialización de la matriz *d* dentro del mismo lazo, esquivando el uso de dos bucles for adicionales.

V. CASO DE ESTUDIO 3

En este caso de estudio se realizará la optimización del código propuesto utilizando procesamiento vectorial SIMD (instrucciones vectoriales SSE). Para utilizarlas, añadiremos en la cabecera del código "#include <inmintrin.h> " para que el compilador reconozca las instrucciones vectoriales SSE3.

Para efectuar las operaciones se utilizarán registros de 128 bits, que almacenarán dos doubles cada uno.

Cuando se trabaja con la matriz b, se cargan elementos consecutivos de una fila de b, no de una columna, porque la matriz b está almacenada por filas. La solución que se ofrece es trasponer la matriz b antes de cargarla para que en cada fila de la matriz este una columna almacenada y así almacenar elementos consecutivos de la fila, (que realmente son elementos de la columna). De esta manera, cuando se carguen elementos consecutivos con la instrucción de carga vectorial de SSE, lo que se cargará será dos elementos seguidos almacenados en memoria.

Este cálculo de la matriz traspuesta supone un tiempo adicional que también deberá ser tenido en cuenta para las medidas de tiempo.

En la **Imagen 5** se declararán los registros necesarios y se utilizarán los mismos bucles *for* que antes.

IMAGEN 5

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 3

```
    __m128d x1, x2, x3, final, final2;

    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N-1; j+=2) {

            final = _mm_setzero_pd ();
            final2 = _mm_setzero_pd ();

            for (int k = 0; k < 8; k+=2){

                x1 = _mm_load_pd((bt[j] + k));

                x2 = _mm_load_pd((c + k));
                x3 = _mm_sub_pd(x1, x2); //resultado parentesis

                x1 = _mm_load_pd((a[i] + k));
                x2 = _mm_mul_pd(x1, x3); //resultado multiplicacion

                x1 = _mm_set_pd(2, 2);
                x3 = _mm_mul_pd(x2, x1);

                final = _mm_add_pd(final, x3);
```

Tanto los lazos de j, como de k, aumentarán de dos en dos unidades, ya que los registros trabajan con dos doubles en cada instrucción.

Dentro del bucle de j se comenzará inicializando a 0 *(_mm_setzero_pd)* los registros en los que se almacenará el resultado de las operaciones realizadas en el bucle más interno.

Para llevar a cabo las operaciones se utilizan distintas operaciones vectoriales, cuyos resultados se van almacenando en los registros temporales x1, x2 y x3.

- *_mm_load_pd*: Carga un valor de memoria dentro de un registro.
- *_mm_sub_pd/_mm_mul_pd/_mm_add_pd*: resta,multiplica y suma registros.

Al final de cada iteración, el valor obtenido se suma al valor del registro denominado final.

A continuación, en la **Imagen 6,** se realiza el mismo código para (j+1), ya que sino existiría un problema a la hora de cargar los resultados en la matriz.

IMAGEN 6

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 3

```
                x1 = _mm_load_pd((bt[j+1] + k));

                x2 = _mm_load_pd((c + k));
                x3 = _mm_sub_pd(x1, x2); //resultado parentesis

                x1 = _mm_load_pd((a[i] + k));
                x2 = _mm_mul_pd(x1, x3); //resultado multiplicacion

                x1 = _mm_set_pd(2, 2);
                x3 = _mm_mul_pd(x2, x1);

                final2 = _mm_add_pd(final2, x3);

            }


            _mm_storel_pd((d[i] + j), final);
            _mm_storeh_pd(&temporal, final);

            d[i][j] = d[i][j] + temporal;


            _mm_storel_pd((d[i] + j + 1), final2);
            _mm_storeh_pd(&temporal, final2);

            d[i][j+1] = d[i][j+1] + temporal;

        }

    }
```

Al realizar la instrucción de store en *d[i] + j*, es necesario realizar saltos de dos unidades entre las iteraciones del bucle j para que el programa funcione. Si no se repitiese el código para *j+1*, se estarían saltando la mitad de las filas.

A la hora de hacer los store, se obtienen los dos valores del registro y se suman en la misma posición de la matriz, utilizando una variable auxiliar *(temporal)*. Esta suma se debe a que en una posición del registro se encontrará los valores calculados con k pares y en otra con k impares.

Cabe mencionar que con la instrucción *_mm_storel_pd* cargamos el valor inferior del registro en memoria, mientras que con *_mm_storeh_pd* cargamos el valor superior.

VI. CASO DE ESTUDIO 4

En este cuarto apartado, se ha hecho un programa utilizando OpenMP para paralelizar la versión secuencial optimizada, sin utilizar variables tipo "reduction" y sin extensiones vectoriales. A través del uso de hilos con ejecución simultánea, se realizará una mayor optimización con respecto a los anteriores casos de estudio.

OpenMP es una interfaz de programación de aplicaciones multiproceso de memoria compartida, actualmente, estándar en la mayoría de compiladores. Está basado en una serie de directivas que, en tiempo de compilación, son usadas por el compilador para introducir el código necesario para lanzar al mismo tiempo múltiples hilos; los cuales, se dividen el trabajo de la región paralela según lo haya definido el programador y luego unifica los resultados.

Para poder trabajar con OpenMP, en primer lugar, es necesario incluir la librería <*omp.h*> y definir el número de hilos con el que se querrá trabajar. Este último valor se almacenará dentro de la constante *K*.

Para especificar el código que se quiere ejecutar con hilos será necesario introducirlo dentro de un bloque delimitado por corchetes e iniciado con la sentencia *#pragma omp*. Seguidamente, irán los distintos parámetros que se quieran utilizar.

Como se puede apreciar en la **Imagen 7**, la sentencia pragma delimitará un bloque con ejecución paralela, con las variables *a, b, c,* y *d* compartidas, las variables *j* e *i* privadas para cada hilo y especificará por último el número de hilos.

Dentro de ese bloque se instrucciones *#pragma omp* for para indicar que se paralicen los bucles for. De este modo cada iteración del bucle será llevada a cabo por un hilo.

IMAGEN 7

FRAGMENTO DEL PROGRAMA EN C UTILIZADO PARA ESTUDIAR EL CASO DE ESTUDIO 4

```
#pragma omp parallel shared(a,b,c,d) private(i,j) num_threads(K)
{
    #pragma omp for
    for (i = 0; i < N; i++) {

        for (j = 0; j < N; j++) {

            d[i][j] = 0;

            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]);
        }
    }

    #pragma omp for
    for (int i = 0; i < N; i++) {
        e[i] = (d[ind[i]][ind[i]]) / 2.0;
        f += ((double) e[i]);
    }
}
```

VII. Resultados

Se mostrará cómo afecta el valor del parámetro N al rendimiento del procesador para diferentes casos de optimización. Mencionar para el caso de estudio relacionado con OpenMP también se ha variado el número de hilos hasta el máximo número de cores del PC (1, 2, 4, 6 y 8)

Para ello, se seguirá el siguiente procedimiento en ambos casos de estudio:

1. Medir el número de ciclos de la parte del programa en que se hace la computación indicada en el pseudocódigo.
2. Repetir el proceso para cada número N de filas y columnas de la matriz .
3. Realizar un total de 10 medidas para diferentes valores.
4. Para cada caso, anotar en un excel y seleccionarar mediana de estos valores como valor final de medida de tiempo de ejecución para elaborar unas gráficas (necesarias para interpretar los resultados).

En todos los experimentos realizados, se ha utilizado el sistema operativo Linux y en ningún caso se ha abierto algun programa, aparte del terminal, durante la realización del experimento.

Por otro lado, cada apartado se ha compilado del siguiente modo:

A) El código secuencial se ha compilado sin optimizaciones del compilador (gcc -O0), con optimización incluyendo la autovectorización (gcc -O2) y por último con -O3.

B) El código secuencial mejorado se ha compilado sin optimizaciones del compilador (gcc -O0)

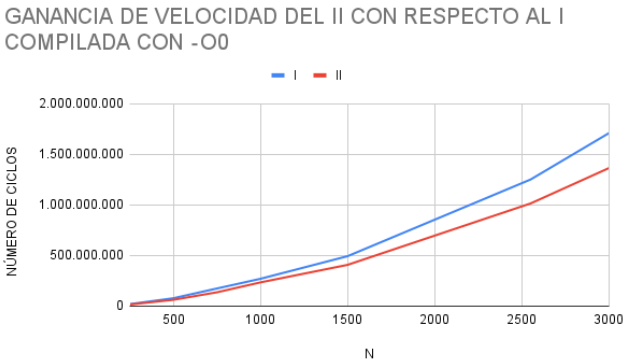
C) El código secuencial optimizado utilizando procesamiento vectorial SIMD se ha compilado sin optimizaciones del compilador (gcc -O0).

D) El código utilizando OpenMP se ha compilado sin optimizaciones del compilador y con el flag de OpenMP (gcc -O0 -fopenmp) y también con la autovectorización (gcc -O2 -fopenmp).

Se procederá a la interpretación de los resultados obtenidos en las distintas gráficas.

A. Ganancia en velocidad de los códigos del caso de estudio I con respecto al II (compilado con -O0)

Gráfica 1



En este apartado se interpretará el efecto de la productividad del PC cuando el programador realiza una optimización.

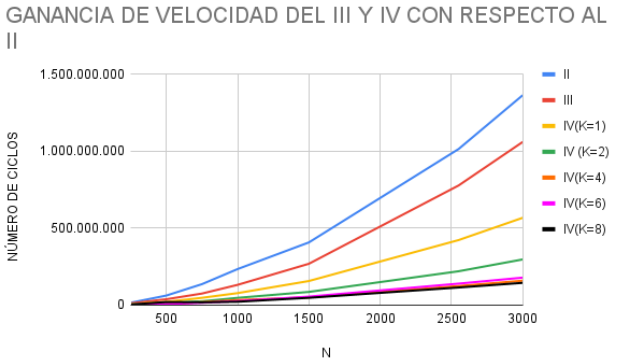
Sobre el caso de estudio I, se ha tratado de mejorar el código secuencial de forma manual, teniendo como resultado el caso de estudio II. En primer lugar, se han fusionado dos bucles para mejorar la localidad temporal y en segundo lugar, se ha realizado un desenrollamiento de lazos provocando la eliminación de la estructura del bucle. Esto evita que se produzcan comprobaciones de final de bucle y con ello una reducción del tiempo de ejecución.

Todo esta optimización se aprecia en la **Gráfica 1**. Se observa una mejora del segundo apartado, en el que los ciclos pasan de alrededor de 1 700 000 000 ciclos en el programa secuencial base (caso de estudio I), a estar por debajo de los 1 500 000 000 en el Programa secuencial optimizado (caso de estudio II).

B. Ganancia en velocidad de los códigos del caso de estudio III y IV con respecto al II (compilado con -O0)

En esta **Gráfica 2**, se verá como el código secuencial optimizado no es la mejor opción a la hora de hablar de ganancia de velocidad. Este código, que supuso una mejora, queda ahora como el peor de los resultados.

GRÁFICA 2



Así, existen otras opciones para mejorar aún más esta aceleración, cómo utilizar un procesamiento vectorial SIMD (extensiones SSE3) o una interfaz del estilo OpenMP para paralelizar la versión secuencial optimizada anterior.

El resultado mejora usando extensiones vectoriales, pero mejora todavía más con el uso de OMP. Además, se puede apreciar como los resultados mejoran a medida que se aumenta el uso de hilos, aunque se estabilizan a la hora de trabajar con hilos cercanos a los máximos.

Las extensiones vectoriales mejoran el rendimiento en las nuevas aplicaciones, y algunas existentes, mediante el manejo de paquetes de datos vectoriales más grandes, y el uso de más hilos y núcleos del procesador. Además también en el procesado de imagen ,tratamiento de vídeo, procesamiento de audio, modelado 3D y permite realizar operaciones en paralelo.

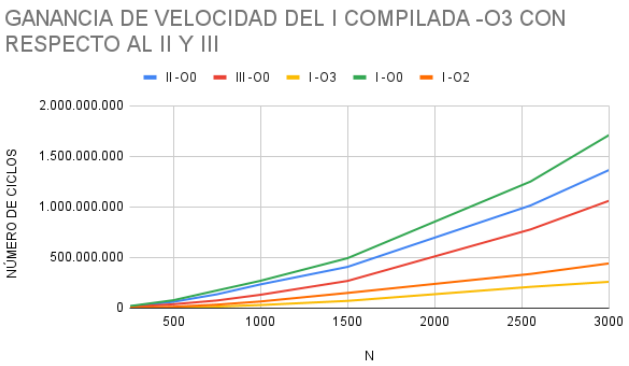
El mayor poder de procesamiento provoca que las aplicaciones que usen intensivamente estas instrucciones pueden realizar el trabajo más eficientemente, dando un rendimiento por watt más alto que con otro conjunto de instrucciones. De esta manera, este código provoca que tenga un menor número de ciclos.

OpenMP también se trata de un modelo de paralelismo pero está basado en hilos. Este paralelismo es explícito pues el programador indica las partes del programa que son paralelizables. En este caso, se ha especificado paralelizar sólo la parte del código en la que se mide el rendimiento.

De esta manera, a través de esta interfaz simple y flexible que es OpenMP, se ha acelerado el número de ciclo por medio del lanzamiento de múltiples hilos para que trabajen en paralelo. A mayor número de K, menor tiempo de ejecución, siendo la diferencia más notable para los primeros valores de k.

C. Ganancia en velocidad de los códigos del caso de estudio I (compilado con -O3) con respecto al resto de casos

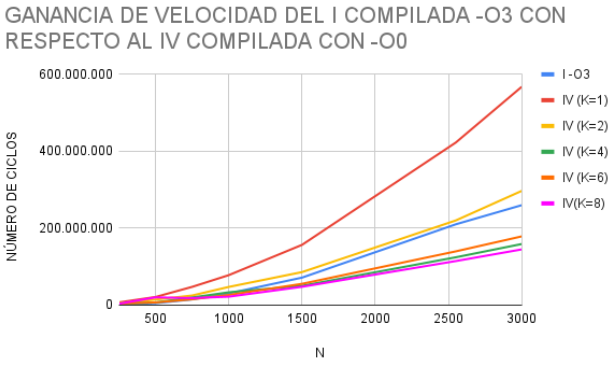
GRÁFICA 3



Un programa puede ser parelizado por el hardware, el programador o el compilador. De esta última forma, el programador puede aplicar manualmente algunas de las optimizaciones que aplica el compilador pero no puede actuar sobre otras. Entre ellas están las optimizaciones al compilar, como la compilación -O3 como muchas otras:

- -O0 Sin optimización. Se debe usar en las etapas de desarrollo y debugging del programa
- -O2 Maximiza la velocidad. Es la optimización que se aplica por defecto.
- -O3 Optimización a nivel 2 más optimizaciones de bucles y memoria mucho más agresivas. Este nivel es el recomendado para aplicaciones con bucles que usan operaciones de coma flotante intensivamente o procesos grandes conjuntos de datos.

GRÁFICA 4



En las **Gráficas 3 y 4** se mostrará cómo afectan las optimizaciones del compilador en el speedup del procesador utilizado.

El primer programa (caso de estudio I) compilado con -O3 ofrece el rendimiento óptimo respecto al resto de programas (menos con las ultimas K de OpenMP). Este código pasa de ejecutar alrededor de 2 000 000 000 ciclos con -O0 a alrededor de 250 000 000 con -O3. En esta compilación se realizan unas optimizaciones de bucles y memoria mucho más agresivas (optimiza los bucles con extensiones vectoriales), por lo que los tiempos de ejecución disminuyen.

El segundo puesto lo ocupan los programas compilados con -O2, que vectoriza con instrucciones SSE de manera automática, quitando las mejoras extra que realiza -O3.

Tal y como se ha dicho, la compilación en -O3 gana a todos los programas menos algunos códigos de OpenMP. Cuando el número de hilos con el que se trabaja en OMP es superior a 4, se obtienen mejores resultados.

D. LA GANANCIA EN VELOCIDAD CONSEGUIDA PARA LOS DIFERENTES NÚMEROS DE HILOS VARIANDO EL VALOR DE N EN EL CASO DE ESTUDIO IV

En este apartado, se mostrarán las diferencias de tiempo de ejecución a la hora de realizar el programa utilizando un número diferente de hilos a través de la **Gráfica 5 y 6.**

La mayor diferencia en este caso ocurre de 1 a 2 hilos (*K*), y esta va decreciendo a medida que aumentamos este valor, siendo apenas apreciable la diferencia entre la ejecución con 6 hilos y con 8 hilos.

GRÁFICA 5

Este incremento del rendimiento al ampliar el número de hilos es debido al aumento del paralelismo a nivel de tareas. Mientras que un código secuencial se ejecuta en un solo core del procesador, con la creación de hilos se permite que varios cores trabajen a la vez ejecutando diferentes secciones del código.

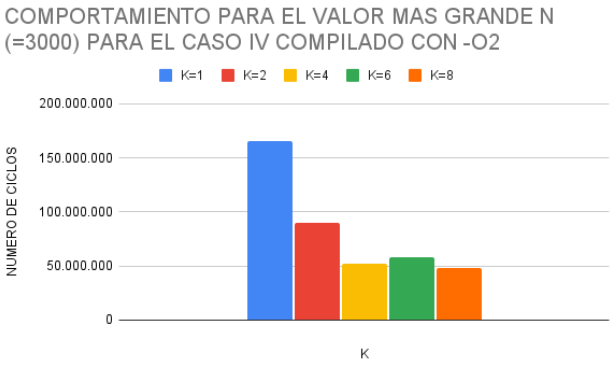
GRÁFICA 6

En la **Gráfica 6** se aprecia principalmente como los valores tienen una estructura similar a la **Gráfica 5**, pero con la opción -o2 la optimización que realiza el compilador es mayor. Así, los tiempos que se obtienen son mejores para todos los casos, reduciéndose hasta en torno a los 50 000 000.

E. COMPORTAMIENTO PARA EL VALOR MÁS GRANDE DE N EN EL CASO DE ESTUDIO IV

En este caso, se observará la diferencia únicamente de tiempos para los valores más grandes de N.

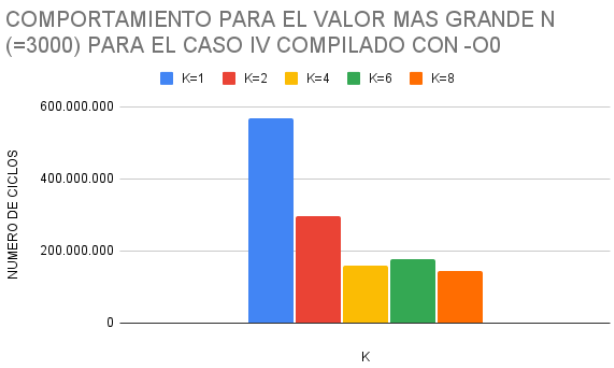
GRÁFICA 7



Se observa en la **Gráfica 7** que los tiempos de ejecución mejoran a medida que se aumentan los hilos, especialmente en los primeros tres valores. Sin embargo, el valor de k = 6 es peor que los valores de k = 4 y k = 8, y la diferencia entre estos tres valores es muy pequeña.

Esta disminución del aumento de velocidad podría deberse a la cantidad de recursos consumidos por una cantidad de hilos demasiado grande.

GRÁFICA 8



En esta última representación (**Gráfica 8**) se puede ver un gráfico muy similar al anterior, pero con valores más bajos para todos los valores de k. De nuevo, se puede apreciar que los valores se estancan a partir de k = 4, y que incluso incrementan para k = 6.

VIII. Conclusiones

Se ha analizado el speedup al variar diferentes rangos de optimizaciones y tamaños de matrices en punto flotante en un algoritmo simple.

Con este fin, se han realizado unos programas en C que impriman el tiempo de ejecución en número de ciclos de procesador. Tras tomar 10 medidas y anotar los resultados, se han elaborado una serie de gráficas mostrando su comportamiento y se ha comprobado que el rendimiento es diferente estructurando el código con distintas optimizaciones.

Estas optimizaciones se han llevado a cabo a través de la compilación de estos códigos con diferentes opciones y manualmente por el programador.

El caso de estudio I confirma que un programa base, sin ningún tipo de optimización realizada, supone un tiempo de ejecución demasiado elevado. Sin embargo, este se puede reducir a través de opciones de compilación como -O2 (vectoriza con instrucciones sse de manera automática), pero sobre todo -O3 (además de vectorizar como -O2, optimiza los bucles con extensiones vectoriales). Este último puede incluso ganar en tiempo de ejecución a mejoras manuales en el propio código (como las vistas en el caso de estudio II y III).

El caso de estudio II aprueba que se pueden llevar a cabo optimizaciones de manera manual: cambiar el código de manera que se realicen las operaciones en otro orden, cambiar el orden de los lazos del producto de matrices para acceder con más localidad a ellas o realizar unrolling (desenrrollamiento de lazos) , realizar operaciones por bloque, etc. Estas optimizaciones pueden suponer un aumento del tamaño del código, disminuyendo la legibilidad del mismo y haciendo más difícil su mantenimiento, aunque suponen un aumento del rendimiento que no es despreciable.

En el caso de estudio III, se ha comprobado la mejora que suponen las extensiones vectoriales respecto a las optimizaciones anteriores del caso de estudio II. De nuevo, el uso de estas, dificulta la programación, obligando a trabajar con registros y operaciones que se alejan de un lenguaje de programación de alto nivel. Estas tienen un poder de procesamiento que provoca que las aplicaciones que usen intensivamente estas instrucciones pueden realizar el trabajo más eficientemente.

En el caso de estudio IV, se ha apreciado tanto la mejora como la facilidad que supone el uso de OMP, aumentando el paralelismo de la aplicación y por tanto el rendimiento, con muy pocas líneas extra de código. De esta manera, se concluye, que el paralelismo a nivel de hilo es muy eficiente, tanto que se puede afirmar que esta mejora del paralelismo de tareas tiene mayor peso sobre el rendimiento que una optimización agresiva sobre los bucles y el acceso a memoria como -O3 (único caso que el tiempo de ejecución es menor a esta compilación).

Después de observar las diferencias de productividad entre los distintos valores de k, se ha llegado a la conclusión de que un valor de k = 4 podría ser un valor óptimo que ofrecería buenos resultados en cuanto a tiempos de ejecución. Pues no utiliza muchos recursos en el sistema y además con otros valores mayores, la diferencia en número de ciclos no sería muy grande.

Por tanto, OpenMP (caso de estudio IV) es el mejor caso presentado para optimizar nuestro programa, quedando en segundo lugar las extensiones vectoriales (caso de estudio III) y en el tercer puesto, el sistema secuencial optimizado (caso de estudio II).

En cuanto a las opciones de compilación, ha quedado claro, que el uso tanto de -O2 como -O3 suponen una mejoría en cualquiera de los programas, y unidas a los métodos mencionados en el párrafo anterior, se puede llegar a tener resultados increíblemente óptimos.

Finalmente, exponer que este estudio sería posible completarlo en un futuro trabajo a través del uso de otro tipos de extensiones vectoriales como las AVX y AVX2 y con otro tipo de paralización de OMP (por ejemplo, en secciones), comprobando si tienen un efecto parecido a las SSE3 y a la paralelización *#pragma omp for*

Referencias

[1] Patterson, David. A. y Hennessy, John .L., Computer Organization and Design ARM Edition: The Hardware Software Interface, 4 Edición, USA, Morgan Kaufmann, 2017, 978- 0128017333.

[2] Arquitectura de Computadores Tema 4, https://www.fdi.ucm.es/profesor/rhermida/AC-Grado/AC_tema4.pdf, (última visita 12 mayo de 2021).

[3] OPTIMIZACIÓN Y PARALELIZACIÓN DE UN ALGORITMO DE SINCRONIZACIÓN MEDIANTE EL USO DE GPUS Y LA TECNOLOGÍA CUDA,http://oa.upm.es/22711/1/3b5726d751a783bb30d1112f8e308238_MEM_memoriatfg.pdf (última visita 10 mayo de 2021).
[4] Extensiones vectoriales avanzadas,https://es.wikipedia.org/wiki/Extensiones_Vectoriales_Avanzadas (última visita 11 mayo de 2021).
[5] Compilación en el Superordenador,https://supercomputacion.uca.es/wp-content/uploads/2017/10/Compilacion-en-el-Superodenador.pdf?u(última visita 9 mayo de 2021).