# Práctica 2: Programación multinúcleo e extensións SIMD

Antonio Moscoso Sánchez, Adrián Vidal Lorenzo

Arquitectura de computadores
Grupo 04
{antonio.moscoso, adrian.vidal.lorenzo}@rai.usc.es

7 de maio do 2021

**Resumo**

*Nesta práctica comparamos o rendemento dun mesmo algoritmo empregando diferentes técnicas de programación, buscando acadar a menor latencia posible para a execución do pseudocódigco que se indica nas instruccións da práctica. As técnicas empregadas inclúen procesamento vectorial SIMD (mediante o uso de extensións SSE3) e paralelización do código con OpenMP. Aplicaranse tamén diferentes optimizacións que non serán dependentes de técnicas como o uso de fíos, etc.*

***Palabras clave: optimización, unrolling, extensións vectoriais, paralelizable...***

## I. Introducción

Por medio dun programa en C, calculouse o rendemento dunha operación matricial utilizando para isto diferentes grados de optimización, como poden ser as extensión vectoriais. Ademais, daránselle diferentes tamaños ao problema para así poder avaliar a escalabilidade de todas as solucións implementadas.

Trátase de obter con cada un dos diferentes apartados un código máis eficiente á hora de facer os cálculos sobre unha matriz, producíndose erros que se terán en conta na xerarquía de memoria.

O documento divídise en diferentes partes comezando polas especificacións do equipo utilizado para a proba dos códigos realizados:

- Explicación do código secuencial obtido.
- Aplicación de optimizacións e mellora na velocidade de execución respecto ao código secuencial
- Uso e aplicación de extesións vectoriais para a realización dos cálculos da matriz d.
- Aplicación da librería OpenMP para a obtención dun código paralelizable e zona de traballo compartida.

Finalmente, expóñense os diferentes resultados obtidos coas súas conclusións e comparacións dos distintos apartados, acompañado da interpretación sobre a gañanacia de velocidade entre os diferentes códigos realizados na práctica e coas diferentes opcións de optimización do compilador.

## II. Material para o experimento

O ordenador portátil utilizado para a práctica foi MSI GF63 thin 9SC que conta cun procesador Intel Core i7-9750H que ten 6 cores.

Este dispositivo conta con 3 niveis caché. O tamaño da liña en cada un dos niveis é de 64 bytes. Cada core contará con un tamaño caché:

| caché | L1(datos) | L1(Instrucións) | L2 | L3(Compartida) |
|---|---|---|---|---|
| Tamaño | 32 KB | 32 KB | 256 KB | 12 MB |

## III. Código secuencial

Para este apartado da práctica realizouse a programación secuencial do pseudocódigo proporcionado. Engásede un vector inicializado con valores aleatorios dende 0 a N, sendo N indiferente para os resultados.

A creación das matrices realízase de maneira dinámica na seguinte orde: a, b, c, e,ind

e finalmente d. As matrices a, b e o vector c son inicializados con números aleatorios comprendidos entre 0 e N.

Dentro da zona de computación atópase o cálculo da matriz d constituído por tres bucles anidados, ademais, realízase o cálculo do vector *e* que almacenará en cada posición un elemento almacenado previamente na matriz *d*, accedendo a eles coa axuda do vector *ind*. No mesmo bucle que se realiza dita acción calculase o valor de *f* que será a suma de cada compoñente do vector *e*.

### A. Cálculo da matriz

O cálculo das diferentes posicións da matriz *d* realízase a través de tres bucles anidados. O primeiro deles recorrerá as diferentes filas, mentres que o segundo recorrerá as columnas da matriz. O último dos bucles recorrerá 8 elementos da matriz, tnto por fila como por columna, e á súa vez recorrerá o vector *c*.

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

## IV. Código secuencial optimizado

Neste apartado optouse por unha optimización de código a través do uso do **unrolling**. Esta técnica consiste no desenvolvemento dun bucle, producindo unha maior aparición de instrucións. A finalidade desta optimización é a da mellora da velocidade do programa a custo dun código máis extenso a través da reescritura da mesma operación as veces necesarias evitando así un maior traballo de optimización para o ordenador.

Na programación deste apartado optouse pola utilización dunha matriz auxiliar (prima) que almacenará os diferentes valores da resta das matrices *b e c*. No cálculo desta matriz aparece por primeira o unrolling, xa que se pode evitar o uso do bucle que incrementa a variable k en cada execución, resultando así 8 instrucción por iteración do bucle que recorrerá as columnas da matriz *b*.

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

Unha vez obtida a matriz auxiliar, realízase o cálculo da matriz *d*.

### A. Cálculo da matriz

Neste cálculo tamén se realizou a optmización con unrolling, pois ao igual que no caso da matriz prima, non se precisa do bucle que itera coa variable k. O resultado será de 8 instrucións para o cálculo da matriz na posición *i* filas e *j* columnas. A matriz *d* xa non se inicializa co valor 0, no seu lugar iguálase o primeiro cálculo realizado para dita posición, mentres que o resto son sumados.

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

## V. Procesamento vectorial SIMD

Por terceiro lugar, levouse a cabo a programación dun programa secuencial no que se utilizará un procesamento vectorial SIMD para intentar conseguir un maior paralelismo a nivel de datos, xa que se aplicará a mesma instrucción sobre múltiples datos, sendo neste caso unha cantidade de dous doubles. Aparecen 4 rexistros do tipo *__mm128d*:

- *calA:* Rexistro onde se cargan os valores da matriz a.
- *calB:* Rexistro onde se cargan os valores da matriz b.
- *calC:* Rexistro na cal se cargan os valores do vector c.

- *reg2:* Rexistro onde se cargan os valores do vector creado que conterá o valor 2 polo que se multiplicará na operación a realizar para o cálculo da matriz d.

As diferentes matrices utilizadas o longo dos distintos programas realizados nesta práctica reservabanse dinámicamente co uso da función *malloc()*. Neste caso o compilador non realiza ningún tipo de comprobación para saber se esta reserva dinámica esta alineada, para lograr este aliñamento e non obter como resultado un erro na execución faise uso da función *_mm_malloc()*.

As instrucción vectoriais utilizadas farán uso de dous datos de tipo double, polo cal o aliñamento necesario para a función *_mm_malloc()* será de 16 bytes.

## A. Cálculo da matriz

Para o cálculo desta matriz reutilizarase o código do primeiro apartado sen optimizar. Como se cargarán de dous en dous doubles, o terceiro bucle (encargador de desonrolar a variable k) pasará a incrementarse de dous en dous. En primeiro lugar cárganse os valores da matriz a, correspondientes a esa fila. A dita matriz accedese por fila, polo tanto, só se necesita indicar a primeira posición de carga.

A continuación, cárganse os valores da matriz b. A esta matriz accederase por columnas, polo que se emprega un vector de doubles auxiliar que almacenará dous doubles en cada iteración do bucle, sendo estes os correspondentes a: *b[k][j]* y *b[k+1][j]*, con j a columna da matriz correspondente no momento. Para finalizar as cargas, cárganse os valores do vector c.

Reutilizando o rexistro no cal se cargan os valores de a, calcúlase a multiplicación por dous, utilizando para isto a función *_mm_mul_pd* na que se utilizará o rexistro *reg2*. Seguindo cos cálculos reutilízase o rexistro *calC*. Nel calcúlase a resta da posición en b e c, o que se consegue co uso da función *_mm_sub_pd*. Volvendo a utilizar *calA* e como último cálculo no que se utilizan instrucciones vectoriais multiplícanse os resultados obtidos anteriormente (*calA* e *calC*).

Para finalizar, almacénase o resultado obtido das operación anteriores no vector creado con anterioridade para a carga dos valores da matriz *b*. Como se trata dun vector que conterá dous doubles súmanse en ambas posicións o valor que xa se atopaba na fila *i* e na columna *j* da matriz *d*. O código resultante do cálculo da matriz será:

```
for(int64_t i = 0; i < N ; i++){
    for(int64_t j = 0; j < N; j++){
        for(int64_t k = 0; k < C; k+= 2){
              calA = _mm_load_pd(&a[i][k]);
              temporal[0] = b[k][j];
              temporal[1] = b[k+1][j];
              calB = _mm_load_pd(temporal);
              calC = _mm_load_pd(&c[k]);
              calA = _mm_mul_pd(reg2, calA);
              calC = _mm_sub_pd(calB, calC);
              calA = _mm_mul_pd(calA, calC);
              _mm_store_pd(temporal, calA);
         d[i][j] += temporal[0]+ temporal[1];
        }
    }
}
```

# VI. OpemMP

Por último lugar, faise uso da librería OpenMP. Esta librería proporciona a oportunidade de crear un programa paralelizable que utiliza memoria compartida. Esta baseada no modelo fork-join, onde o traballo se divide en distintos fíos para finalmente ser unificado nun mesmo resultado. O número de fíos a usar neste caso irá dende un 1 ata o número máximo de cores que posúe o portátil usado para o experimento, polo que o número máximo de fíos será 6.

## A. Paralelización da zona compartida

Pártese do código optimizado obtido no apartado 2. En primeiro lugar defínese a zona paralelizable, que comeza antes do cálculo da matriz auxiliar creada para o cálculo da resta das distintas posicións das matrices b e c.

**#pragma** parallel

A continuación, defínense as diferentes variables que serán compartidas polos diferentes fíos que traballarán na memoria compartida e establécese o número de fíos que traballarán no cálculo. Neste caso, todas as matrices

creadas no programa serán utilizadas polos fíos. Ademais a variable *f* será tamén compartida, e nela gardarase o resultado final.

```
shared(prima,a,b,c,d,f,e) num_threads(k)
```

Na zona paralelizable que se acaba de establecer realízanse os diferentes cálculos que lle tocan a cada fío participante.

## B. Zona compartida

Dentro da zona compartida, establécese en primeiro lugar unha variable privada para todos os fíos que fará o traballo de f, xa que esta pode producir carreiras críticas se seguía posicionada do mesmo xeito que anteriormente.

No interior desta zona realízanse diferentes bucles for, polo cal se debe paralelizar cada un dos diferente bucles para producir un resultado correcto xa que os fíos realizan diversas iteracións dunha maneira concorrente. A construción destes bucles paralelizables lévase a cabo coa seguinte sintaxis:

**#pragma** omp for

Dita sentencia atópase en tres lugares diferentes do programa. A súa primeira instancia aparece cando se calculan os valores que conterá a matriz auxiliar (prima). En segundo lugar, posiciónase antes do comezo dos bucles anidados que cumplen a responsabilidade de obter os diferentes valores da matriz d. Finalmente, sitúase no cálculo dos valores do vector *e* e cálculo da variable privada definida previamente.

Como se mencionou con anterioridade, a variable *f* podería traer como consecuencia a aparición de carreiras críticas, pois varios fíos poderían tentar escribir nela ao mesmo tempo, producindo un resultado non desexado. Para que esto non suceda, utilízase a variable privada para realizar o traballo que *f* debería levar a cabo. Unha vez finalizado o último bucle indícase creando unha zona crítica con *atomic*:

**#pragma** omp atomic
f += f_privado;

Con esta sentencia, aseguramos que só un fío escribirá na variable sen que ningún outro interfira na avaliación.

Finalmente, sáese da zona paralelizable, tomando así o fío principal o control de novo.

# VII. Opcións de optimización

Ao longo da práctica utilízanse diferentes opcións de compilación, como poden ser *-O0,-O1, etc*

- *-O0*: Opción por defecto, reduce o tiempo de compilación.
- *-O2*: Esta opción incrementa o tempo de compilación e o rendemento do código. Permite o uso de extensións vectoriais SSE ou AVX.
- *-fopenmp*: Habilitación de directivas OpenMP. Xérase código paralelo.
- *-O3*: Esta opción contén entre as súas optimizacións toda-las proporcionadas por *-O2*.

# VIII. Resultados

## A. Opción de compilacion *-O0*

A seguinte gráfica compara os distintos tempos de execución dos apartados da práctica, para os tamaños de N:

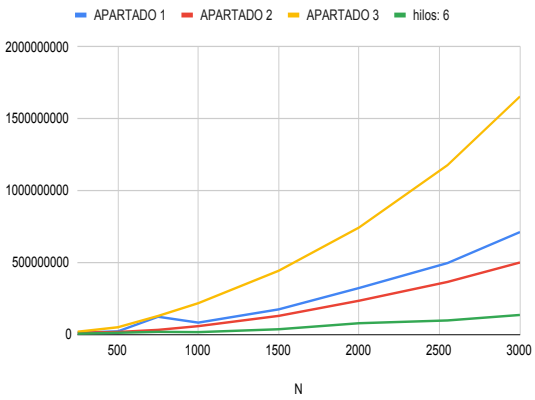


**Figura 1:** *Tempos de execución nos apartados*

Na gráfica aparecen representados os tempos para os distintos apartados e tamaños. O apartado 4 realizouse con 6 fíos, o máximos número de fíos respecto a cores que se podían empregar. As medidas dos catros códigos foron tomados coa opción de optimización *-O0*. Pódese apreciar como, ao contrario do que se podería esperar, as execucións do programa optimizado empregando extensión vectoriais é o que ten un tempo de execución máis alto. Para o resto de apartados, a tendencia é a

esperada: a latencia do apartado 1 (moi ineficiente) é maior que a do apartado 2, e esta á súa vez é maior que a do apartado 4. O caso especial do apartado 3 será explicado no apartado de conclusións. A tendencia do apartado 4 expón o nivel de optimización que se acada coa paralelización do código empregando fíos, que realiza a execución do código nun tempo moito menor que os outros apartados.

## B. Opción de compilación *-O2*

A continuación, móstrase o custo en ciclos do código secuencial aplicando a opción de optimización *-O2*. A compilación con esta opción incrementará o rendemento do código e o tempo de compilación.

Para ver unha comparación máis clara, a gráfica obtida representará o custo obtido a través do código no cal se utiliza as instrución da librería OpenMP.

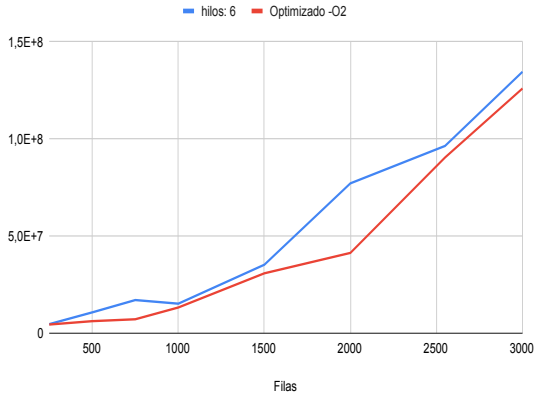


**Figura 2:** *Diferenza de ciclos da versión secuencial compilada con -O2 e a versión paralelizada*

Como se aprecia na gráfica, a versión inicial compilada coa opción -O2 consegue un nivel de optimización moi alto, cunha diferenza pequena coa versión do código empregando OpenMP.

## C. Diferenzas no apartado 4

A continuación, preséntanse os diferentes resultados obtidos para un número máximo de fíos coincidinte co número máximo de cores do portátil utilizado para as probas. En primeira instancia compilase o código coa opción de optimización *-O0*.

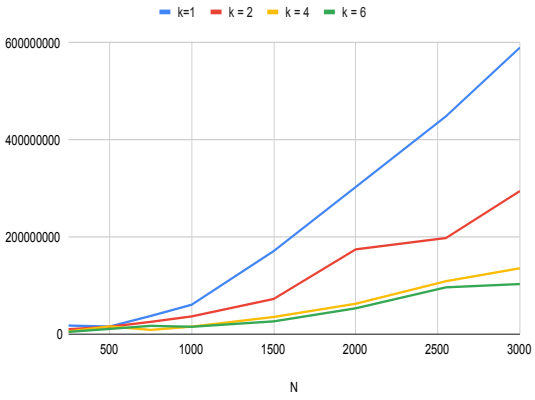


**Figura 3:** *Diferenza de ciclos entre k fíos*

Os resultados obtidos permiten observar, como según o tamaño da matriz é maior, o rendemento vai empeorando canto menor é o número de fíos. Isto débese a que cada fío terá asociado unha maior carga de traballo xa que non se pode repartir. Sendo o tamaño da matriz pequeno, obtense un número similar de ciclos, isto ven dado a que o reparto de traballo entre os distintos fíos se complica, non saíndo eficiente.

Móstrase a continuación o custo de ciclos obtidos con 6 fíos.

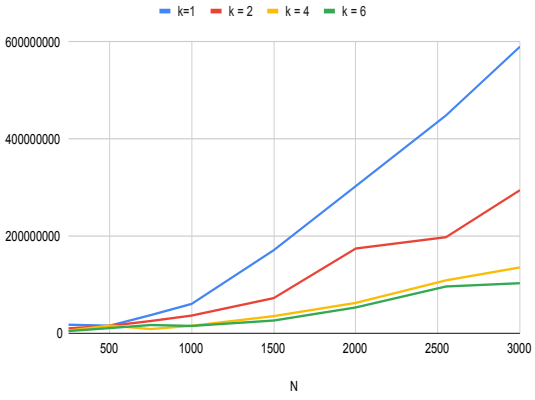


**Figura 4:** *Optimización -O0 e 6 fíos*

Como segunda instancia, realízase a compilación coa opción *-O2*.

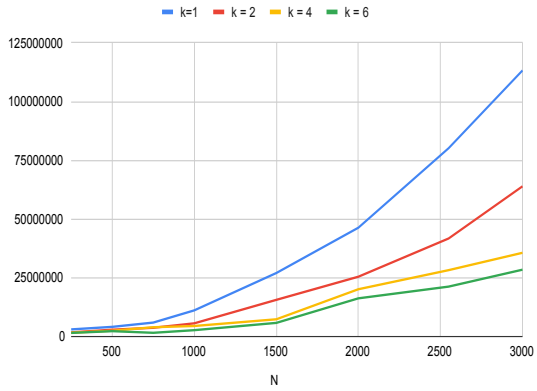


**Figura 5:** *k fíos con optimización -O2*

Como era esperado, o número de ciclos disminúe notablemente, xa que observando a figura 4, cando a matriz ten o tamaño máximo requerido o número de ciclos é similar os requeridos neste caso cando só se dispón dun fío.

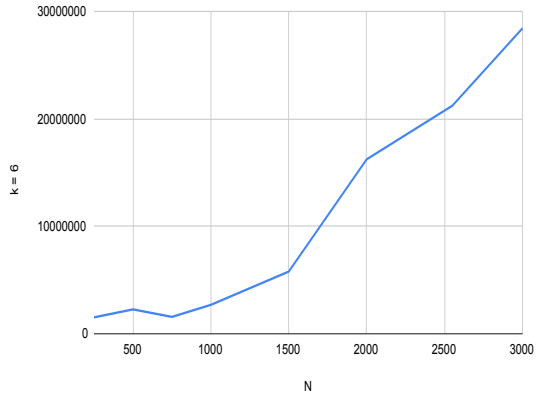


**Figura 6:** *Optimización -O2 e 6 fíos*

Pódese apreciar unha maior diferenza facendo unha lixeira comparación entre a figura 4 e 6, representando a mesma cantidade de fíos, pero, aplicando diferentes opcións de optimizacións. Neste caso, o uso da compilación *-O2* produce un decremento notable en número de ciclos, facendo que este código teña un alto rendemento.

## D. Speedup

A gráfica mostrada na figura 7 mostra a ganancia en velocidade do código secuencial optimizado fronte o código secuencial simple, incluíndo además as ganancias dos códigos que conteñen extensións vectoriais SIMD e uso da librería OpenMp fronte o código secuencial optimizado. Tomouse a decisión da representación a través dun gráfico formado por columnas, xa que así se pode apreciar con mellor claridade a evolución da ganancia en cada apartado respecto o tamaño da matriz. Tendo en conta que o speedup dun apartado (x) fronte outro apartado (y) se obteneñen a través da función:

$$S = \frac{TiempoExecucion_x}{TiempoExecucion_y}$$

Obtendo os seguintes resultados:

| | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2550 | 3000 |
|---|---|---|---|---|---|---|---|---|
| Apartado 2 | 1,10 | 1,28 | 3,95 | 1,43 | 1,35 | 1,38 | 1,36 | 1,42 |
| Apartado 3 | 0,59 | 0,33 | 0,24 | 0,26 | 0,29 | 0,31 | 0,31 | 0,30 |
| Apartado 4 | 2,28 | 1,50 | 1,81 | 3,74 | 4,88 | 4,39 | 3,78 | 4,84 |

**Cadro 1:** *Ganancia de velocidade por apartado*

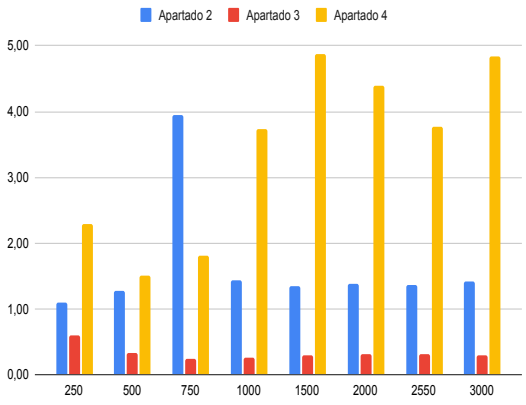


**Figura 7:** *SpeedUp con optimización -O0*

Observando a gráfica apréciase como o uso da librería OpenMP proporciona unha alta gañancia fronte un código secuencial optimizado a través de unrolling. Por outro lado, o código contedor de extensións vectoriais, sendo estas operacións de alto custo non se mellora o tempo do código secuencial optimizado, se non que incluso é peor. O resultado esperado en este caso é unha melloría por parte das extensións vectoriais. Esta gráfica presenta o explicado anteriormente onde a gañancia de velocidade cando se utilizan 6 fíos non é moi alta cando o tamaño da matriz

| | 250 | 500 | 750 | 1000 | 1500 | 2000 | 2550 | 3000 |
|---|---|---|---|---|---|---|---|---|
| Apartado 2 | 0,14 | 0,23 | 0,38 | 0,32 | 0,19 | 0,16 | 0,11 | 0,09 |
| Apartado 3 | 0,08 | 0,08 | 0,09 | 0,08 | 0,06 | 0,05 | 0,03 | 0,03 |
| Apartado 4 (k = 6) | 0,33 | 0,35 | 0,69 | 1,20 | 0,93 | 0,69 | 0,42 | 0,46 |

**Cadro 2:** *Ganancia de velocidade por apartado*

é pequeno, permitindo deducir que se debe a custosidade do reparto do traballo.

A continuación, móstrase o SpeedUp realizado coa compilación do código secuencial con optimización *-O3* e o resto de códigos que foron compilados coa opción *-O0*.

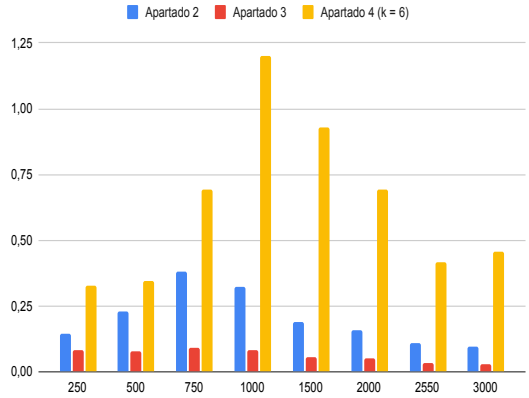


**Figura 8:** *Optimización -O0 frente a -O3*

A anterior gráfica mostra como unha compilación do código secuencial con optimización *-O3* é moito máis eficiente fronte os apartados secuencial optimizado e o ante o uso de diferentes extensións vectoriais. En cambio, o apartado realizado coa librería OpenMP pode ser máis eficiente a según o tamaño da matriz.

## IX. Conclusións

Ao longo do documento mostráronse diferentes resultados de rendemento sobre un programa que realiza unha operación matricial. A este programa aplicáranselle diferentes métodos de optimización como pode ser o desenvolvemento de bucles (unrolling), o uso de rexistros e extensións vectorias (instrución SIMD) e por último o uso da librería OpenMP para a paralelización dunha zona de traballo e poder así aplicar o modelo fork-join. Ademáis destes tipos de optimización, tivéronse en conta as opcións de optimización do compilador, concretamente utilízanse: *-O0*, *-O2*, *-O3*.

Cabe destacar que o resultado proporcionado polas extensión vectoriais non é o esperado, pois debería producir un menor número de ciclos. Unha posible razón para esto pode ser o alto custo que teñen estas instrucións e sendo o número de doubles a cargar dous, o procesador non consegue facer un bo rendemento do programa, pois desperdicia moito tempo para conseguir un número de datos pequeno en cada iteración dos bucles. Tamén hai que recalcar a importancia das opcións de optimización do compilador e da súa utilidade cando se fala de rendemento. Por último, a librería OpenMP proporciona boas funcionalidades para conseguir un bo resultado, sen obter un programa de gran tamaño como pode ocurrir co uso de unrolling e sen necesidade dun aliñamento de memoria como no caso das extensións vectorias, xa que a través da paralelización da zona interesada se pode conseguir que o código gañe unha elevada velocidade respecto os outros tipos de optimizacións vistos nos demáis códigos.

Como futuro experimento plantéxase a idea da realización dun estudo do comportamento das extensións vectoriais para que poidan producir un resultado non desexado como resulta ser neste caso.

## Referencias

[1] Blanco Heras,Dora, Piñeiro Pomar, César Alfredo, Quesada Barriuso, Pablo. *Tutorial sobre programación usando extensiones vectoriales*

[2] Blanco Heras,Dora. *Práctica 2: Mejoras de caché*

[3] Antelo Suárez, Elisardo. *Introducción a OpenMP*

[4] *OpenMP Application Program Interface*

[5] GCC, the GNU Compiler Collection, Option Summary https://gcc.gnu.org/onlinedocs/gcc/Option-Summary.html

[6] Intel Corporation, https://www.intel.es/content/www/es/es/homepage.html