Programación Multinúcleo e extensións SIMD

Juan Pablo García Amboage, Pablo Monteagudo Lago

Laboratorio de Arquitectura de Computadores
Grupo 2
{pablo.monteagudo.lago,juanpablo.garcia}@rai.usc.es

25 de septiembre de 2022

Resumen

Programouse un algoritmo simple para traballar con vectores de quaternions representados en punto flotante utilizando diferentes optimizacións: extensións vectoriaris SIMD, paralelización mediante OMP, optimización secuencial mediante técnicas coma loop unrolling e optimizacións do compilador. Tomáronse medidas do rendemento dos distintos programas para diversos tamaños do problema e concluiuse que facendo un uso apropiado das extensións vectoriais pódense acadar melloras considerables do rendemento, así como a través da paralelización pero únicamente para os tamaños máis grandes.

Palabras clave: optimización, quaternion, AVX, OpenMP...

I. Introducción

A computación de altas prestacións ten un papel fundamental na actualidade. Esta permítenos dispoñer de predicións meteorolóxicas a curto e medio plazo, descubrir novos fármacos, detectar novas partículas fundamentais... Así, ademais de dispoñer de máquinas moi potentes, requírese optimizar o máximo posible os programas para poder extraer resultados en intervalos de tempo razoables.

Agora ben, os compiladores actuais son programas dunha enorme complexidade que realizan numerosas optimizacións automáticas co obxectivo de que os programas resultantes en linguaxe ensamblador sexan o máis eficientes posibles. Por outra parte, os procesadores modernos dispoñen de mecanismos para reducir as latencias dos nosos programas: execución fóra de orde e paralela, predición de saltos...

Non obstante, o capacidade de optimización dos anteriores é limitada e, en numerosas ocasións, un bo coñecemento do problema en cuestión pode permitir a introdución de optimizacións que de ningún modo poderían ser realizadas de xeito automático. Deste xeito, coñecer as técnicas de optimización máis comúns resulta fundamental para a construción de programas eficientes.

Neste contexto, centrámonos en optimizar un algoritmo simple con vectores de quaternions en punto flotante. Así, aplicamos diferentes combinacións de optimizacións e estudamos os rendementos acadados para diferentes tamaños do problema.

O rendemento mediuse a través do número de ciclos requeridos para realizar o conxunto de computacións do algoritmo, excluíndo todas aquelas operacións de inicialización comúns ás diferentes implementacións. No cadro 1 detállanse as especificacións do procesador da computadora sobre a cal se realizou a experimentación.

II. Descripción da experimentación

As estruturas de datos sobre as que opera o algoritmo a optimizar son os quaternions. Estes trátanse de vectores de catro compoñentes en punto flotante que foron implementados do seguinte xeito:

```
struct Quaternion{
  double w;
  double x;
  double y;
  double z;
```

Nome de modelo	Intel Core i7-7500U
Arquitectura	x86_64
Litografía	14 nm
Núcleos	2
Fíos	4
Frecuencia básica	2,70 GHz
Frecuencia máxima	3,50 GHz
Caché L1d	32K
Caché L1i	32K
Caché L2	256K
Caché L3	4096K

Cuadro 1: *Especificacións procesador*

```
};
typedef struct Quaternion quaternion;
```

O algoritmo que debe realizar o programa móstrase na figura 1. Este involucra dúas operacións sobre os quaternions: a suma e a multiplicación.

```
Entradas:
vectores de quaternions a(N),b(N): N elementos de valor aleatorio coas
compoñentes dos quaternions de tipo double.
Saída:
dp: quaternions con componentes de tipo double.

Computación:
c(N): vector auxiliar de quaternions.
for i=1,N {
    c(i)=a(i)*b(i); // * indica multiplicación de quaternions.
}
dp=0; //inicialización do quaternion a cero (todas as compoñentes cero);
for i=1,N {
    dp=dp+c(i)*c(i) // * e +: multiplicación e suma sobre quaternions.
}
```

Figura 1: *Algoritmo a optimizar*

Realizáronse as catro versións seguintes do programa:

- Programa secuencial base.
- Programa secuencial optimizado.
- Programa secuencial optimizado utilizando extensións AVX para empregar procesamento vectorial SIMD.
- Programa utilizando OpenMP paralelizando a versión secuencial optimizada.

Todos estes programas compiláronse coa opción *-O0*, que indica ao compilador que non introduza optimizacións automáticas que, a priori, resultan impredecibles. Deste xeito, as diferenzas de rendemento foron explicadas polas optimizacións programadas, eliminando este factor de incertidumbre que podería dar lugar a comportamentos difíciles de explicar.

Non obstante, o rendemento do programa secuencial base avaliouse sobre dúas versións: unha compilada con *-O0*, como se indicou previamente, e outra con *-03* e autovectorización (*-mavx*). Con este último programa, comparamos as nosas optimizacións propias fronte as realizadas de xeito automático polo compilador.

A comparación dos rendementos fíxose en base aos **ciclos medios por quaternion**. A experimentación realizouse para tamaños de vectores $N = 10^q, q \in \{2, 4, 6, 7\}$. Para cada tamaño de vectores e versión realizáronse **20** medidas e tomouse como resultado final a **mediana** destes valores.

Para a medida dos ciclos empregamos as funcións da librería *rutinas_clock*. Así, mediante o uso da función *mhz* asegurámonos que a frecuencia de reloxo se manteña constante durante e execución. Isto é necesario xa que os procesadores actuais varian a súa frecuencia en función da carga de traballo, mais a nosa experimentación require que esta se manteña constante para obter unha medida fiable dos ciclos medios por quaternion. Coa función *start_counter* iniciamos o contador dos ciclos e, logo da execución do algoritmo, obtemos o valor deste mediante a función *get_counter*.

III. Secuencial base

O programa secuencial base constitúe unha traducción directa do pseudocódigo do algoritmo á linguaxe C. Non obstante, antes de comezar coa execución do algoritmo é preciso reservar memoria para os vectores de quaternions. Para isto empregamos a función *_mm_malloc* en lugar de *malloc* xa que queremos aliñar o vector de quaternions co tamaño dunha estrutura deste mesmo tipo, 32 bytes. Deste xeito, á hora da liberación de memoria empregamos *_mm_free*.

O bucle principal é o seguinte:

```
mhz(1,1);
start_counter();
//Realizamos as operacions indicadas
for(i = 0; i < N; i++){
    c[i]=multQuaternion(a[i],b[i]);
```

```
}

dp->w=0;
dp->x=0;
dp->y=0;
dp->z=0;

for(i = 0; i < N; i++) {
    *dp = sumQuaternion(*dp,
    multQuaternion(c[i],c[i]));
}
ck=get_counter();
```

As funcións *sumQuaternion* e *multQuaternion* realizan a suma e multiplicación de dous quaternions recibidos como parámetros e devolven o quaternion resultado:

```
quaternion sumQuaternion(quaternion a,
    quaternion b) {
  quaternion c;
  c.w = a.w+b.w;
  c.x = a.x+b.x;
  c.y = a.y+b.y;
  c.z = a.z+b.z;
  return c;
}

quaternion multQuaternion( quaternion a
    , quaternion b) {
  quaternion c;
  c.w = a.w*b.w - a.x*b.x - a.y*b.y - a
    .z*b.z;
  c.x = a.w*b.x + a.x*b.w + a.y*b.z - a
    .z*b.y;
  c.y = a.w*b.y - a.x*b.z + a.y*b.w + a
    .z*b.x;
  c.z = a.w*b.z + a.x*b.y - a.y*b.x + a
    .z*b.w;
  return c;
}
```

Neste programa non se inclúe ningún tipo de optimización, de maneira que se espera que as diferentes optimizacións a introducir permitan acadar rendementos superiores nos sucesivos programas.

IV. Secuencial optimizado

Tomando como punto de partida o programa secuencial base introducironse unha serie de optimizacións para tentar lograr melloras significativas no tempo de execución con respecto ao anterior.

As diferenzas con respecto ao secuencial base aprécianse, en primeiro lugar, na declaración dos punteiros aos vectores de quaternions:

```
quaternion*__restrict__ a = NULL;
quaternion*__restrict__ b = NULL;
quaternion*__restrict__ dp = NULL;
```

Así, engadiuse a palabra clave *__restrict__* que indicará que os punteiros declarados serán a única forma de acceder ás direccións ás que apuntan. Deste xeito, o compilador non introducirá comprobacións adicionais, as cales poderían diminuir o rendemento do noso programa.

A variable contador *i* que empregamos para iterar nos bucles declarouse empregando a palabra clave *register*. Deste xeito, indicamos ao compilador que esta debería ser gardada nun rexistro, na búsqueda de reducir as penalizacións por acceso a memoria, o que resulta esencial xa que se accederá continuamente a dita variable.

Á hora de realizar a inicialización do quaternion *dp* substituíuse a inicialización directa de cada unha das súas compoñentes a 0 por unha invocación a *memset*, función con este propósito específico.

No programa secuencial base o algoritmo descompoñíase en dous bucles. Nesta versión integráronse ambos bucles e desenroscouse este. Con isto, búscase maximizar o paralelismo a nivel de instrución, reducindo o número de saltos condicionais a ser evaluados.

Nun principio, contemplouse a posibilidade de realizar o bucle cara atrás (de N a 0), xa que a comparación con cero é máis eficiente que a comparación con calquer outro número (por supoñer esta última unha operación aritmética adicional):

```
//Realizamos as operacions indicadas
for(i = N; i ; i-=5){//LOOP UNROLLING
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i-1],b[i-1]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i-2],b[i-2]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i-3],b[i-3]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i-4],b[i-4]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i-5],b[i-5]),*dp);
}
```

Este fragmento de código dará lugar, en principio, a dependencias do tipo *Read After Write*, xa que o valor de *dp* é recibido como parámetro nas invocacións a *cuadradoYSumaQuaternion* logo de ser modificado na instrución anterior. Así, avaliamos unha alternativa em-

pregando resultados parciais para eliminar as posibles dependencias, comprobando que os rendementos de ambos programas eran totalmente parellos. Isto pode explicarse polo feito de que estamos anticipando as dependencias a través das instrucións nunha linguaxe de programación de alto nivel. Deste xeito, cando o compilador realice a tradución a linguaxe ensamblador, cada unha das liñas do bucle anterior, dará lugar a un elevado número de instrucións ensamblador, sendo así o impacto das posibles dependencias comentadas anteriormente mínimo.

Non obstante, recorrendoo deste xeito estariamos dificultando o aproveitamento da localidade secuencial a través do mecanismo de prefetching cuxa eficacia susténtase en que cando se emprega un dato nun instante de tempo cercano se empregue o dato seguinte en memoria. Así, ao recorrer o bucle cara adiante, é probable que se anticipen as cargas dos datos consecutivos, reducindo o número de fallos caché e, por tanto, a latencia do programa. En base a isto, o bucle quedou como segue:

```
//Realizamos as operacions indicadas
for(i = 0; i < N ; i+=5){//LOOP
    UNROLLING
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i],b[i]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i+1],b[i+1]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i+2],b[i+2]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i+3],b[i+3]),*dp);
    *dp = cuadradoYSumaQuaternion(
    multQuaternion(a[i+4],b[i+4]),*dp);
}
```

En esta versión, en lugar de empregar a función *multQuaternion* para realizar o cadrado do vector resultante de multiplicar *a* y *b* empregouse unha función específica que ademais sumará o resultado do cadrado co vector *dp*. As operaciones requeridas para realizar o cuadrado dun quaternion poden simplificarse con respecto de aquelas requeridas na multiplicación:

```
static inline quaternion
    cuadradoYSumaQuaternion(quaternion
    a, quaternion b) {
quaternion c;
    c.w = a.w*a.w - a.x*a.x - a.y*a.y -
     a.z*a.z + b.w;
    c.x = 2.*a.w*a.x + b.x;
    c.y = 2.*a.w*a.y + b.y;
    c.z = 2.*a.w*a.z + b.z;
    return c;
}
```

Finalmente, na declaración das funcións *multQuaternion* e *cuadradoYSumaQuaternion* engadiuse o modificador *inline*. As invocacións a funcións conlevan un sobrecoste en termos de rendemento xa que é preciso gardar o estado do programa no momento no que se invocou a función, saltar a posición de memoria na que comeza a función... Para evitar isto emprégase este modificador que indicará ao compilador que inclúa o código da función no punto no que esta se invocou, eliminando así a chamada a esta. Non obstante, segue recaendo no compilador a decisión de aplicar o *inline* ou non, de maneira que se engadiu o modificador *__attribute__((always_inline))* na declaración das funcións para forzalo a realizar este cambio. Unha consecuencia negativa de aplicar o *inline* é o incremento no tamaño do programa, o que non supón un problema na nosa situación xa que só se require optimizar o rendemento do programa en termos do tempo de execución.

V. Secuencial optimizado utilizando extensións AVX

A natureza dos quaternions (vectores de catro compoñentes) convérteos en candidatos excelentes á optimización mediante operacións vectoriais, o que se realizou empregando as extensións AVX.

O algoritmo presentado admite dúas posibilidades de vectorización: sobre as operacións con quaternions (suma, multiplicación e cadrado) ou sobre as iteracións do bucle. Implementamos ambas versións e realizamos un estudo cuantitativo de cal de elas proporciona un mellor rendemento.

A. Vectorización multiplicación

Na primeira das versións, comezamos vectorizando a multiplicación. Para iso, fixámonos na estrutura da multiplicación reagrupando os termos como se aprecia na figura 2.

Así, o cálculo de cada unha das columnas sinaladas involucra o produto de dúas permutacións dos quaternions *a* e *b*. Para construir

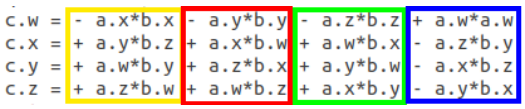

Figura 2: *Multiplicación de quaternions*

estas permutacións empregamos a función *_mm256_permute4x64_pd*. Por tanto, realizando o produto das permutacións apropiadas obtéñense as catro columnas sinaladas. A continuación, móstrase a implementación realizada:

```
static inline void multQuaternion(
    quaternion *a, quaternion *b,
    quaternion *c) {
__m256d vA = _mm256_load_pd((double *)a
    );
__m256d vB = _mm256_load_pd((double *)b
    );
__m256d vC;
__m256d mask = _mm256_set_pd
    (-1.,-1.,-1.,1.);

__m256d vAp = _mm256_permute4x64_pd(vA,
     0xC9); //xywz 11001001
__m256d vBp = _mm256_permute4x64_pd(vB,
     0x2D); //xzyw 00101101

vC = _mm256_mul_pd(vAp, vBp);

vAp = _mm256_permute4x64_pd(vA, 0x36);
    //yxzw 00110110
vBp = _mm256_permute4x64_pd(vB, 0xD2);
    //ywxz 11010010

vC = _mm256_fmadd_pd(vAp, vBp, vC);

vAp = _mm256_permute4x64_pd(vA, 0x63);
    //zwyx 01100011
vBp = _mm256_permute4x64_pd(vB, 0x87);
    //zxwy 10000111

vC = _mm256_fmadd_pd(vAp, vBp, vC);

vAp = _mm256_permute4x64_pd(vA,0x9C);//
    wzxy 10011100
vBp = _mm256_permute4x64_pd(vB,0x78);//
    wyzx 01111000

vC = _mm256_fmsub_pd(vAp, vBp, vC);
vC = _mm256_mul_pd(vC, mask);

_mm256_store_pd((double *)c, vC);
```

Para o cálculo do cadrado do quaternion poderiamos empregar esta mesma función. Porén, a simplificación realizada no caso do programa secuencial optimizado suxire que esta tamén podería ser aplicada na versión vectorizada. Así, na figura 3 identificáronse as compoñentes involucradas no cadrado e suma de quaternions:

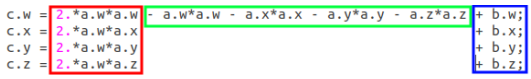

Figura 3: *Cadrado e suma de quaternions*

Apréciase en 3 que o cadrado dun quaternion require o produto da primeira compoñente do quaternion polas compoñentes do mesmo (columna vermella) e o produto escalar deste quaternion por si mesmo. Esta é unha operación de redución sobre o vector xa que transforma este nun escalar. Este tipo de operacións non se adaptan tan ben ao cálculo vectorial, sendo a nosa primeira aproximación para realizar este cálculo a seguinte:

```
__m256d vA = _mm256_load_pd((double *)a
    );
__m256d mask = _mm256_set_pd(0., 0.,
    0., 1.);
__m256d neg = _mm256_mul_pd(vA,vA);
neg = _mm256_hadd_pd(neg, neg);
neg = _mm256_permute4x64_pd(neg, 0x9C);
     //xywz 11001001
neg = _mm256_hadd_pd(neg, neg);
neg = _mm256_mul_pd(neg, mask);
```

A función clave é *_mm256_hadd_pd* que realiza unha suma alterna dos dous pares de compoñentes consecutivas de dous vectores, como se indica na figura 4. Deste xeito, realizamos a multiplicación do quaternion *a* por si mesmo e posteriormente realizamos unha suma alterna (*hadd*), permutando o resultado de maneira que os pares de compoñentes consecutivas conteñan as sumas das compoñentes *0 e 1* e *2 e 3*. Mediante outra suma alterna obtemos un vector que en cada compoñente contén a produto escalar de *a* por si mesmo.

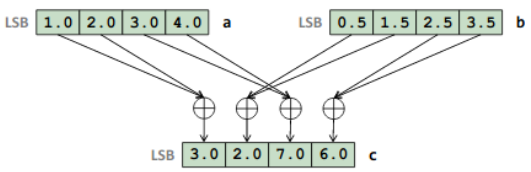

Figura 4: *Funcionamento hadd*

Coa implementación indicada realizamos unha serie de medidas para avaliar o rendemento do programa vectorizado fixándonos así en que este era significativamente inferior

ao do programa secuencial base, sendo o principal cuello de botella a función para realizar o cadrado e, dentro desta, a parte adicada a realizar o produto escalar. Así, puxemos o foco en optimizar este cálculo e, sendo este un problema recorrente, adaptamos unha versión proposta en *StackOverflow* [1]:

```
//Extraemos os 128 bits menos
    significativos
__m128d vlow  = _mm256_castpd256_pd128(
    neg);
//Extraemos os 128 bits mais
    significativos
__m128d vhigh = _mm256_extractf128_pd(
    neg, 1);
//Realizamos a suma dos dous pares de
    doubles
vlow  = _mm_add_pd(vlow, vhigh);
//Extraemos os 64 bits mais
    significativos do resultado
//Que se gardaran nas duas componentes
    dun vector
__m128d high64 = _mm_unpackhi_pd(vlow,
    vlow);
//Realizamos a suma e extraemos os 64
    bits menos significativos  do
    resultado
neg = _mm256_set_pd(0.,0.,0.,
    _mm_cvtsd_f64(_mm_add_sd(vlow,
    high64)));
```

Logo de realizar a multiplicación de *a* por si mesmo, extraemos os 128-bits menos significativos do resultado empregando *_mm256_castpd256_pd128* e gardamos estes na variable *vlow*. Seguidamente, extraemos os 128-bits máis significativos con *_mm256_extractf128_pd* almacenando o resultado en *vhigh*. Sumando estas dúas variables obtemos un vector de dúas compoñentes, contendo a suma dos pares de compoñentes consecutivas do vector inicial (*neg*). Logo disto, mediante *_mm_unpackhi_pd* extraemos os 64-bits máis significativos dos dous vectores pasados como parámetros (neste caso é o mesmo) e gardamos estes no vector *high64*. Así, as dúas compoñentes deste último conterán os 64-bits máis significativos de *vlow*. Deste xeito, ao sumar *vlow* e *high64* o vector resultado conterá nos seus 64-bits menos significativos o resultado do produto escalar. Para extraer este escalar empregamos a función *_mm_ctvsd_f64*.

Nas táboas 2 e 3 realizamos unha comparación das dúas implementacións. Os valores das latencias das instrucións extraéronse da *Intel Intrinsics Guide* [2] para a arquitectura

APARICIÓNS	INSTRUCIÓN	LATENCIA	TOTAL
1	_mm256_load_pd	0	0
1	_mm256_set_pd	X	X
2	_mm256_mul_pd	4	8
2	_mm256_hadd_pd	6	12
1	_mm256_permute4x64_pd	3	3
			23

*Indicamos con **X** os datos que non se proporcionaban na páxina de Intel

Cuadro 2: *Latencia primeira implementación produto escalar*

Skylake. Aínda que a arquitectura do noso procesador é **Kaby Lake**, está é sucesora de Skylake, sendo ambas moi similares, de maneira que os valores para as latencias empregados resultan significativos. Apréciase así que a diferenza, en termos de rendemento, é importante: a latencia da segunda implementación é aproximadamente a metade da primeira.

Coa segunda versión do produto escalar, o cadrado e suma de quaternions quedaría como segue:

```
static inline void
    cuadradoYSumaQuaternion(quaternion*
     a, quaternion* b, quaternion* c) {
  __m256d vA = _mm256_load_pd((double
    *)a);
  __m256d vB = _mm256_load_pd((double
    *)b);
  __m256d vAw = _mm256_set1_pd(a->w);

  __m256d neg = _mm256_mul_pd(vA,vA);

  //Extraemos os 128 bits menos
    significativos
  __m128d vlow  =
    _mm256_castpd256_pd128(neg);
  //Extraemos os 128 bits mais
    significativos
  __m128d vhigh = _mm256_extractf128_pd
    (neg, 1);
  //Realizamos a suma dos dous pares de
     doubles
  vlow  = _mm_add_pd(vlow, vhigh);
  //Extraemos os 64 bits mais
    significativos do resultado
  //Que se gardaran nas duas
    componentes dun vector
  __m128d high64 = _mm_unpackhi_pd(vlow
```

APARICIÓNS	INSTRUCIÓN	LATENCIA	TOTAL
1	_mm256_load_pd	0	0
1	_mm256_set_pd	X	X
1	_mm256_castpd256_pd128	0	0
1	_mm256_extractf128_pd	3	3
2	_mm_add_pd	4	8
1	_mm_unpackhi_pd	1	1
1	_mm_cvtsd_f64	0	0
			12

*Indicamos con **X** os datos que non se proporcionaban na páxina de Intel

Cuadro 3: *Latencia segunda implementación produto escalar*

```
    , vlow);
  //Realizamos a suma e extraemos os 64
     bits menos significativos  do
    resultado
  neg = _mm256_set_pd(0.,0.,0.,
    _mm_cvtsd_f64(_mm_add_sd(vlow,
    high64)));

  __m256d pos = _mm256_mul_pd(vAw,vA);
  pos = _mm256_add_pd(pos,pos);

  __m256d vC = _mm256_sub_pd(pos,neg);
  vC = _mm256_add_pd(vB, vC);

  _mm256_store_pd((double *)c, vC);

}
```

B. Vectorización por iteracións

A vectorización por iteracións é similar ao desenroscado de bucles: en cada pasada do bucle realizaremos as operacións correspondentes a catro iteracións. O bucle principal é o seguinte:

```
for(i = 0; i < N;i+=4){//LOOP UNROLLING
//carga de datos
__m256d aw = _mm256_set_pd(a[i+3].w, a[
    i+2].w, a[i+1].w, a[i].w);
__m256d ax = _mm256_set_pd(a[i+3].x, a[
    i+2].x, a[i+1].x, a[i].x);
__m256d ay = _mm256_set_pd(a[i+3].y, a[
    i+2].y, a[i+1].y, a[i].y);
__m256d az = _mm256_set_pd(a[i+3].z, a[
    i+2].z, a[i+1].z, a[i].z);

__m256d bw = _mm256_set_pd(b[i+3].w, b[
    i+2].w, b[i+1].w, b[i].w);
__m256d bx = _mm256_set_pd(b[i+3].x, b[
    i+2].x, b[i+1].x, b[i].x);
__m256d by = _mm256_set_pd(b[i+3].y, b[
    i+2].y, b[i+1].y, b[i].y);
__m256d bz = _mm256_set_pd(b[i+3].z, b[
    i+2].z, b[i+1].z, b[i].z);

__m256d cw;
__m256d cx;
__m256d cy;
__m256d cz;

//calculo de c
cw = _mm256_mul_pd(ax, bx);
cw=_mm256_fmadd_pd(ay, by, cw);
cw=_mm256_fmadd_pd(az, bz, cw);
cw=_mm256_fmsub_pd(aw, bw, cw);

cx=_mm256_mul_pd(az,by);
cx=_mm256_fmsub_pd(aw, bx, cx);
cx=_mm256_fmadd_pd(ax, bw, cx);
cx=_mm256_fmadd_pd(ay, bz, cx);

cy=_mm256_mul_pd(ax,bz);
cy=_mm256_fmsub_pd(aw, by, cy);
cy=_mm256_fmadd_pd(ay, bw, cy);
cy=_mm256_fmadd_pd(az, bx, cy);

cz=_mm256_mul_pd(ay,bx);
cz=_mm256_fmsub_pd(aw, bz, cz);
cz=_mm256_fmadd_pd(ax, by, cz);
cz=_mm256_fmadd_pd(az, bw, cz);

__m256d auxDw;
__m256d auxDx;
__m256d auxDy;
__m256d auxDz;

auxDw = _mm256_mul_pd(cx,cx);
auxDw = _mm256_fmadd_pd(cy, cy, auxDw);
auxDw = _mm256_fmadd_pd(cz, cz, auxDw);
auxDw= _mm256_fmsub_pd(cw, cw, auxDw);
dw = _mm256_add_pd(auxDw, dw);

auxDx=_mm256_mul_pd(cw,cx);
auxDx=_mm256_add_pd(auxDx,auxDx);
dx = _mm256_add_pd(auxDx, dx);

auxDy = _mm256_mul_pd(cw,cy);
auxDy = _mm256_add_pd(auxDy, auxDy);
dy= _mm256_add_pd(auxDy, dy);

auxDz = _mm256_mul_pd(cw,cz);
auxDz = _mm256_add_pd(auxDz, auxDz);
dz= _mm256_add_pd(auxDz, dz);
}
```

En cada iteración, cárganse catro quaternions para cada vector (*a* e *b*). Para o cálculo do vector de quaternions *c*, realizamos os produtos dos catro pares de quaternions cargados e almacenamos os resultados corres-

podentes a cada compoñente nunha variable vectorial (*cw, cx, cy, cz*).

Posteriormente, calculamos o cadrado para os catro quaternions resultantes (para os cales cada compoñente se atopa nunha variable vectorial distinta) e acumulamos o resultado en *dw, dx, dy, dz*.

Deste xeito, a suma horizontal dos elementos en *dw* proporcionanos a primeira compoñente do vector *dp* resultado. Así, para calcular o resultado final precisamos realizar catro sumas horizontais. A nosa primeira aproximación foi a seguinte:

```
//Sumamos os pares de componentes
    consecutivas de dw e dx
__m256d tempWx = _mm256_hadd_pd( dw, dx
    );
//Sumamos os pares de componentes
    consecutivas de dy e dz
__m256d tempYz = _mm256_hadd_pd( dy, dz
    );

//Extraemos os 128 bits menos
    significativos
__m128d vlow1  = _mm256_castpd256_pd128
    (tempWx);
// Extraemos as sumas parciais
    correspondentes a cada componente
__m256d parcial1 = _mm256_insertf128_pd
    (tempYz,vlow1,0);

// Extraemos as sumas parciais
    correspondentes a cada componente
__m256d parcial2 =
    _mm256_permute2f128_pd(tempWx,
    tempYz, 0x21);

//Sumamos as sumas parcias obtendo o
    resultado final
__m256d res = _mm256_add_pd(parcial1,
    parcial2);

_mm256_store_pd((double *)dp, res);
```

En primeiro lugar, aplicamos dúas reducións mediante *_mm256_hadd_pd* para calcular a suma de cada par de compoñentes consecutivas. De cada un destes vectores será necesario extraer as compoñentes *0 e 2* e *1 e 3* xa que conteñen o resultado das sumas para unha mesma compoñente. Para isto, empregamos *_mm256_insertf128_pd* e *_mm256_permute2f128_pd* de maneira que *parcial1* e *parcial2* conterán as sumas parciais correspodentes a cada compoñente de *dp* na mesma posición. Deste xeito, realizando unha última suma obtemos un vector co resultado das catro sumas horizontais.

Non obstante, unha vez validada esta implementación, documentámonos para buscar un xeito máis eficiente de realizar a redución anterior, que podería formar parte dunha implementación que permitira realizar catro produtos escalares simultaneamente. Deste xeito, adaptamos unha solución proposta en *StackOverflow* [3] para un problema similar:

```
//Sumamos os psares de componentes
    consecutivas de dw e dx
__m256d tempWx = _mm256_hadd_pd( dw, dx
    );
//Sumamos os pares de componentes
    consecutivas de dy e dz
__m256d tempYz = _mm256_hadd_pd( dy, dz
    );
// Extraemos as sumas parciais
    correspondentes a cada componente
__m256d parcial1 =
    _mm256_permute2f128_pd( tempWx,
    tempYz, 0x21 );
// Extraemos as restantes sumas
    parciais correspondentes a cada
    componente
__m256d parcial2 = _mm256_blend_pd(
    tempWx, tempYz, 0x0C);
//Sumamos as sumas parcias obtendo o
    resultado final
__m256d res = _mm256_add_pd(parcial1,
    parcial2);

_mm256_store_pd((double *)dp, res);
```

Esta implementación é moi similar á que ideamos, diferenciándose unicamente en substituir o papel de *_mm256_castpd256_pd128* e *_mm256_insertf128_pd* por *_mm256_blend_pd*, lixeiramente máis eficiente, en termos de latencia, de acordo coa *Intel Intrinsics Guide* [2]. Deste xeito, optamos en incluir esta última solución na nosa implementación, aínda que en termos de rendemento a diferenza fose insignificante.

VI. Paralización do programa secuencial optimizado utilizando OpenMP

Tomando como punto de partida o programa secuencial optimizado paralelizáronse os cálculos mediante o uso de OpenMP. Como se pode observar en apartados anteriores, no programa secuencial optimizado todos os cálculos realízanse nun único bucle. Polo tanto, fíxose uso da instrucción *pragma omp for*

que permite paralelizar dito bucle facendo un reparto das iteracións entre os diferentes fíos. O código resultante é o seguinte:

```
#pragma omp parallel private(i,tid)
   num_threads(numFios)
 {
   tid = omp_get_thread_num();
   #pragma omp for
     for(i = 0; i < N; i+=5){//LOOP
   UNROLLING
       dp[tid] =
   cuadradoYSumaQuaternion(
   multQuaternion(a[i],b[i]),dp[tid]);
       dp[tid] =
   cuadradoYSumaQuaternion(
   multQuaternion(a[i+1],b[i+1]),dp[
   tid]);
       dp[tid] =
   cuadradoYSumaQuaternion(
   multQuaternion(a[i+2],b[i+2]),dp[
   tid]);
       dp[tid] =
   cuadradoYSumaQuaternion(
   multQuaternion(a[i+3],b[i+3]),dp[
   tid]);
       dp[tid] =
   cuadradoYSumaQuaternion(
   multQuaternion(a[i+4],b[i+4]),dp[
   tid]);
     }
 }//termina a execucion paralela
```

Coa directiva *pragma omp parallel* indicamos que vai comezar a sección paralela. Ademais, empregamos *private()* para indicar as variables que deben ser privadas para cada fío. No noso caso, estas serán *i*, posto que se utiliza para recorrer o bucle, e *tid* que é un enteiro que identifica a cada fío, o cal variará entre *0* e *numFios*. Esta última debe ser privada porque, lóxicamente, cada fío terá un identificador distinto. Utilizamos *num_threads()* para indicar o número de fíos entre os que se vai repartir a tarefa da sección paralela. Así, *numFios* non é máis que unha variable de tipo *int* que o programa recibe como parámetro en tempo de execución.

Unha vez dentro da sección paralela cada fío obtén o seu identificador no programa mediante unha chamada a *omp_get_thread_num* e almacenao na variable privada *tid*, comezando seguidamente a execución paralela do bucle. Cómpre mencionar que os fíos non escriben todos no quaternion *dp*, se non que, nesta versión do porgrama, *dp* é un vector dinámico de *numFios* quaternions. Deste xeito, cada fío escribe únicamente na posición de *dp* que lle corresponde, evitando así a aparición de carreiras críticas. Debido a isto, unha vez rematada a sección paralela, é necesario que o fío principal sume os resultados parciais. Estes atoparanse almacenados nas distintas compoñentes de *dp*, obtendo así o valor final de *dp*, o cal se almacena na posición *0* do mesmo. Para acumular os resultados parciais, engadiuse o seguinte código:

```
//cando todos os fios calcularon a sua
    parte o master as suma
for(i = 1; i<numFios; i++) {
    dp[0] = sumQuaternion(dp[i],dp[0]);
}
```

VII. Análise de resultados

Nesta sección, analizaremos os rendementos dos programas comentados anteriormente para diferentes tamaños do problema, avaliando así se as optimizacións introducidas deron lugar a incrementos significativos no rendemento.

Ademais, realizaranse comparacións individuais co obxectivo de avaliar a influencia dunha optimización particular nos tempos de execución obtidos.

A. Comparativa programa secuencial optimizado

Nesta subsección analizaremos diferentes gráficas nas que se compararán os ciclos medios por execución empregados polo programa secuencial optimizado fronte ao resto de programas, variando as lonxitudes dos vectores de quaternions *a* e *b*.

A.1. Programa base fronte a secuencial optimizado

Na figura 5 pódese ver que, como cabía esperar, o programa secuencial optimizado supera ó programa secuencial base para todos os tamaños probados, chegando a consumir únicamente arredor dun terzo dos ciclos medios que o programa base para os tamaños máis grandes. Isto confirma que as optimizacións realizadas tiveron éxito. Á vista das gráficas, pode semellar que cada vez a diferenza de rendemento vai diminuíndo a medida que aumenta o tamaño dos vectores, o que resulta certamente falso. A razón disto é que, a pesar

de que a diferenza absoluta nos ciclos medios entre ambos programas é cada vez menor, ao aumentar o tamaño dos vectores a proporción de mellora proporcionada pola versión optimizada vai aumentando, xa que é preciso relativizar a diferenza absoluta nos ciclos medios fronte ós valores dos mesmos. Deste xeito, para un tamaño de *1e+04* a versión optimizada precisa algo menos da metade dos ciclos que emprega a versión base, mentres para un tamaño de *1e+06* esta proporción é dun terzo. Pode chamar a atención que os ciclos medios de ambos programas son cada vez menores segundo aumentan os tamaños, tendencia que se repetira nos resultados das execucións de todos os programas do estudo. Isto xa foi observado e analizado con detalle na práctica anterior a esta, sendo numerosos factores os que dan lugar a mesma, entre eles a perda de importancia dos costes constantes fronte o resto das computacións. Porén, isto non debe distraernos do obxecto de estudo deste informe que é comparar a diferenza en ciclos medios entre os distintos programas para varios tamaños, non comparar a diferenza entre os ciclos medios para difrentes tamaños dentro dun mesmo programa.

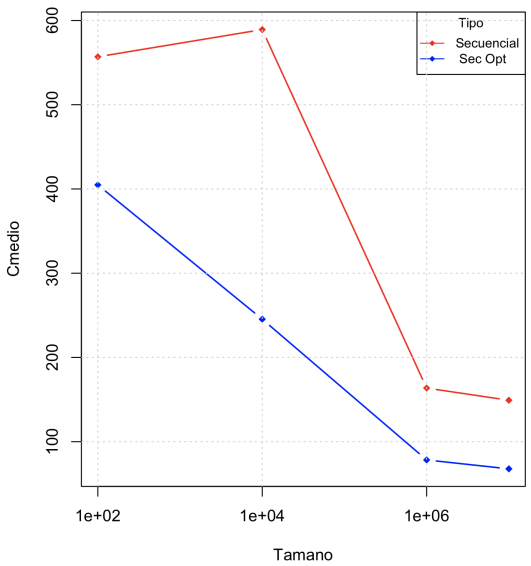

Figura 5: *Programa base (-O0) fronte a secuencial optimizado*

A.2. Programas con AVX fronte a secuencial optimizado

Na figura 6 móstrase unha comparativa entre os ciclos medios das dúas versións do código vectorizado con AVX: vectorizado por quaternions e vectorizado por iteracións do bucle. Como se pode observar, o programa secuencial optimizado é máis rápido que o vectorizado por quaternions para todos os tamaños. Pola contra, o programa vectorizado por iteracións supera ao secuencial optimizado para todos os tamaños. Para explicar o mal rendemento do programa vectorizado por quaternions debemos fixarnos no seu código. Este executa unha gran cantidade de instruccións por cada iteración do bucle: cada vez que se recorre o mesmo invócanse 5 veces ás funcións de multiplicación e cadrado e suma de quaternions. Estes métodos fan uso dun elevado número de instruccións vectoriais, tanto para cargar en rexistros vectoriais os quaternions cos que se vai operar, coma para facer os cálculos necesarios para obter o quaternion que proporcionan coma resultado. O elevado número de operacións vectoriais débese, en gran medida, a forma en que está definida a multiplicación de quaternions, a cal non se realiza compoñente a compoñente, o que a fai menos susceptible a ser vectorizada. Este elevado número de instruccions supón un gran lastre en ciclos de forma que, realizando a vectorización por quaternions, empeórase considerablemente o rendemento do programa fronte ao secuencial optimizado, sobre o cal se introduciu a vectorización. Pola contra, a vectorización por iteracións do bucle resulta moito mais natural e, en efecto, con dito programa conséguense reducir os ciclos medios respecto do secuencial optimizado para todos os tamaños do problema, sacando verdadeiramente proveito da vectorización.

A.3. Programas con OMP fronte secuencial optimizado

Nas figuras 7, 8 e 9 pódense ver diferentes comparativas entre os ciclos medios da versión do código paralelizado con OMP utilizando *1, 2 e 4* fíos fronte ao programa secuencial optimizado. O que máis chama a atención da figura 7 é o mal rendemento de

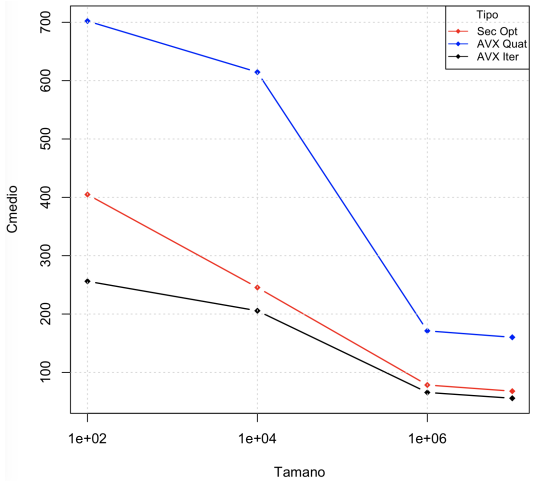

Figura 6: *Programas vectorizados fronte a secuencial optimizado*

todos os programas que usan OMP para tamaño *100*. Ademais, obsérvase que o número de ciclos medios é maior cando se incrementa o número de fíos colaboradores, o que pode resultar contraituitivo. Porén, isto ten unha sinxela explicación xa que a creación dos fíos ten un coste moi elevado en ciclos o que fai que a paralelización non sexa rentable para tamaños pequenos. Neste sentido, resulta lóxico que o programa paralelizado teña peor rendemento cantos máis fíos se utilizan para os tamaños do problema máis pequenos. Non obstante, a situación invírtese cando traballamos con tamaños de vectores máis grandes xa que o custo fixo da creación dos fíos é cada vez menos significativo comparado co custo dos cálculos. Isto fai que a reducción de ciclos obtida ao paralelizar ditos cálculos compense ó sobrecoste asociado creación dos fíos. Deste xeito, pódese ver que para os tamaños máis grandes (*1e+06 e 1e+07*) o programa paralelizado con catro fíos acada os mellores rendementos, xa que se compensa o custo fixo por creación de cada fío coa significativa disminución en ciclos que supón repartir os cálculos entre catro fíos. Por outra banda, o programa paralelizado con dous fios, para os tamaños probados, non consegue compensar o custo da creacion dos fíos coa ventaxa de repartir o traballo entre dous, mais aprecíase que, a medida que se incrementan os tamaños dos vectores, o seu rendemento achégase ao do programa secuencial optimizado. Finalmente tamén resulta chamativo que o programa paralelizado cun so fío non teña un rendemento igual ao programa secuencial optimizado. Isto pode deberse a que, como o numero de fíos do programa non se decide en tempo de compilación, durante a execución débense realizar unha serie de cálculos para repartir as iteracións do bucle entre os disintos fíos aínda que solo se empregue un, facendo así que o rendemento do programa paralelizado con OMP para único fío sexa peor que o do secuencial optimizado en todos os tamaños.

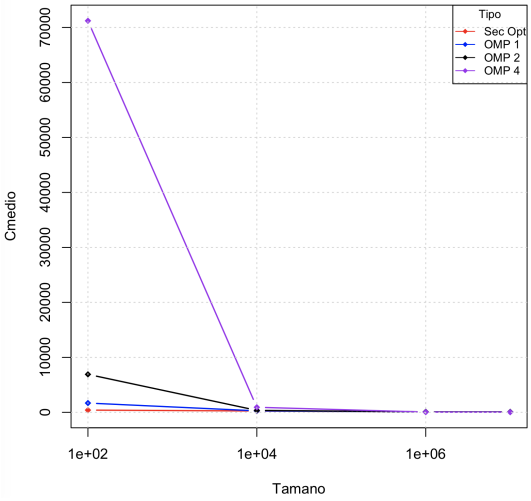

Figura 7: *Programas con OMP fronte a secuencial optimizado*

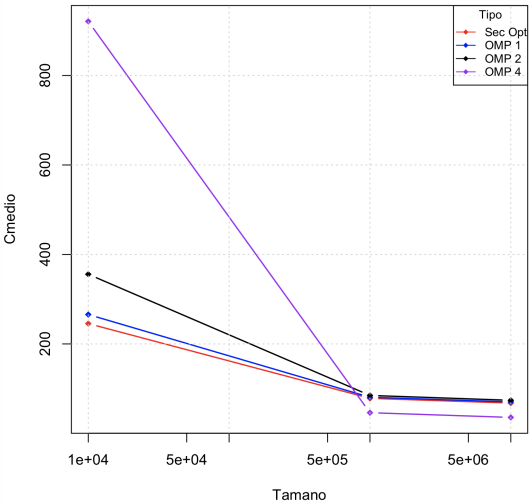

Figura 8: *Programas con OMP fronte a secuencial optimizado sin tamaño 100*

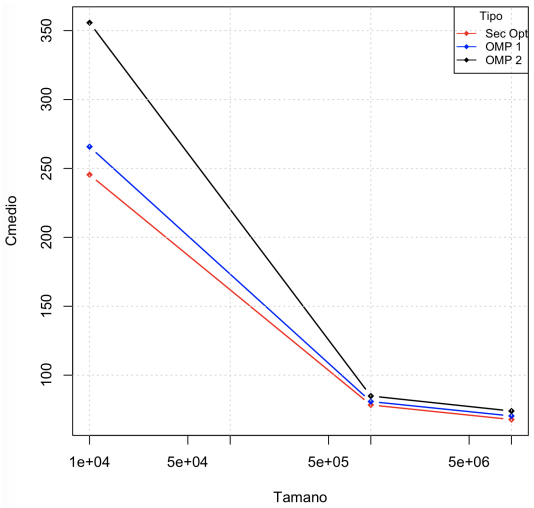

Figura 9: *Programas vectorizados fronte a secuencial optimizado sin 4 fíos e tamaño 100*

B. Comparativa programa compilado con -O3

Como se comentou previamente, completaremos a comparativa dos diferentes programas co programa base compilado coa opción *-O3* e autovectorización.

B.1. Programa base con optimizacións do compilador fronte a programa base sen optimizacións

Ao realizar a experimentación fixámonos en que a diferenza de rendementos proporcionada pola programa base introducindo ou non as optimizacións do compilador resultaba case insignificante, como se aprecia na figura 10 (liñas *Sec* e *Sec -O3*). Para explicar este feito, barallamos a posibilidade de que o emprego da estrutura de datos *Quaternion*, enmascarara a súa estrutura interna (vector de catro *doubles*) limitando así as posibilidades de optimización a introducir polo compilador. Así, realizamos un programa adicional eliminando a estrutura *Quaternion* e empregando no seu lugar vectores de catro *doubles*. Introducindo no programa base as modificacións estritamente necesarias para adaptar este ao cambio da estrutura de datos, incluimos na nosa comparativa o novo programa (*Sec -03 Double*), como se ve na figura 10. Comprobamos así que a diferenza en rendemento é moi significativa con respecto ao programa base coa estrutura *Quaternion*, quedando patente que "desenmascar" a estrutura de datos permite ao compilador detectar e levar a cabo numerosas optimizacións adicionais. Finalmente, para asegurar a veracidade da nosa hipótese, realizamos medidas co programa sen estrutura de datos *Quaternion*, compilándoo sen optimizacións (*-O0*). Así. comprobamos que o seu rendemento era parello ao do programa secuencial base compilado coas mesmos opcións. Deste xeito, verificouse que as melloras obtidas coa opción *-03* se debían exclusivamente a maior posibilidade de beneficiarse das optimizacións do compilador e non as modificacións introducidas para non empregar a estrutura *Quaternion*.

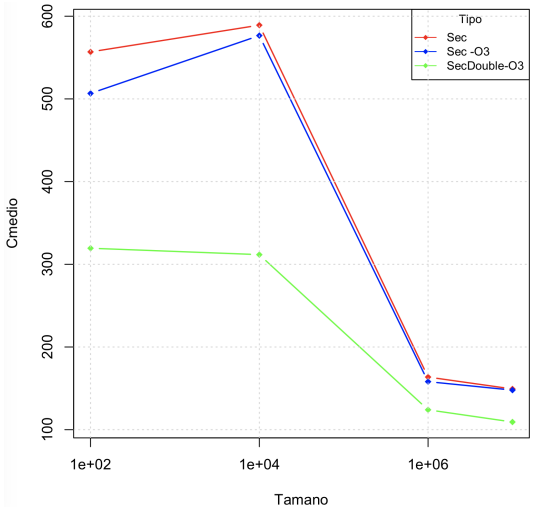

Figura 10: *Programa base (-O0) fronte a secuencial optimizado*

B.2. Programa base con optimizacións do compilador fronte a programa base optimizado

Como se viu previamente, a diferenza de rendementos coas optimizacións do compilador é significativa se se emprega a estrutura *Quaternion* ou non. A pesar disto, na figura 11 apréciase que o rendemento do programa base optimizado segue a ser superior, o que é coherente co feito de que as optimizacións particularizadas ao problema concreto a tratar permiten obter rendementos superiores ás optimizacións xerais que pode chegar a realizar automaticamente o compilador.

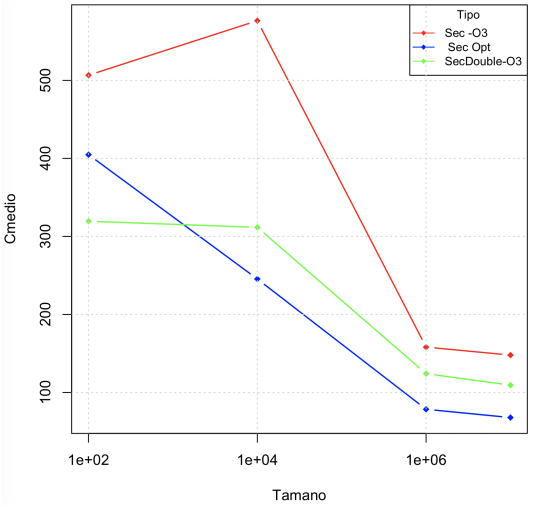

Figura 11: *Programa base (-O0) fronte a secuencial optimizado*

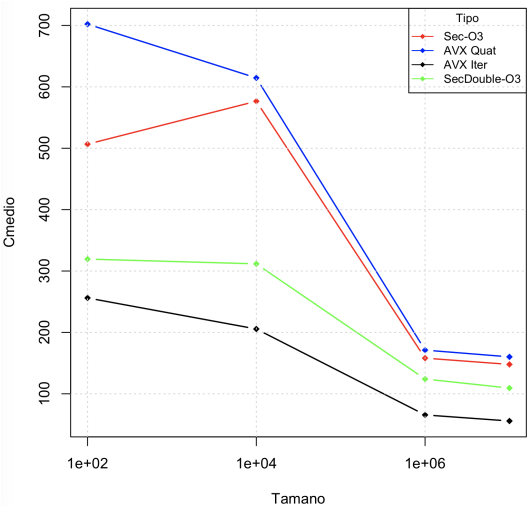

Figura 12: *Programa base (-O0) fronte a secuencial optimizado*

B.3. Programas con AVX fronte a programa base con optimizacións do compilador

Na figura 12 mostramos o rendemento dos programas con AVX fronte ao programa base con optimizacións do compilador. Como se comentou previamente, a vectorización da multiplicación non proporciona unha mellora significativa do rendemento, de maneira que era esperable que, ao introducir as optimizacións do compilador no programa base, este último o superara.

Non obstante, a autovectorización do compilador non se chega a aproximar ao rendemento obtido a través da vectorización por iteracións, a cal se adaptaba perfectamente o algoritmo en cuestión.

B.4. Programas con OMP fronte a programa base con optimizacións do compilador

Na figura 9 quedaron patentes que os rendementos acadados pola paralización para *1* e *2* fíos eran moi similares aos do programa secuencial optimizado. Deste xeito, as diferenzas con respecto ao programa base con optimizacións do compilador serán similares ás observadas en 12, como se aprecia na figura 13. De novo, para vectores grandes, o rendemento proporcionado polo paralelización con *4* fíos é moi superior ao do resto das versións.

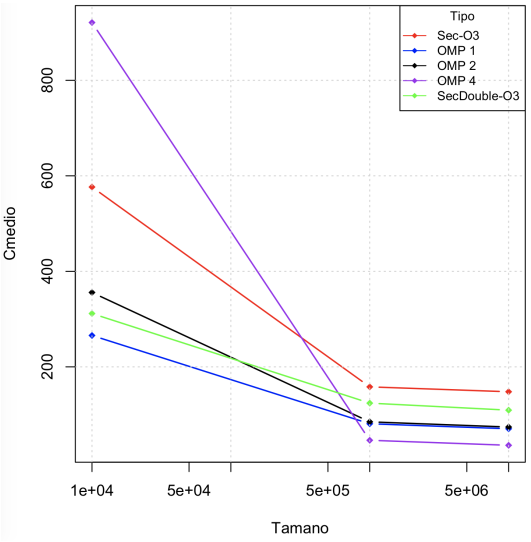

Figura 13: *Programa base (-O0) fronte a secuencial optimizado*

C. Outras comparacións de interese

Nas seccións anteriores quedou patente que os mellores rendementos se acadan mediante a vectorización por iteracións e coa paralelización con OMP, mais neste último caso para tamaños grandes dos vectores e empregando *4* fíos. Así, na figura 14 pódese ver que, para tamaños grandes, a versión paralelizada con *4* fíos acada a cota superior dos rendementos

para os programas estudados.

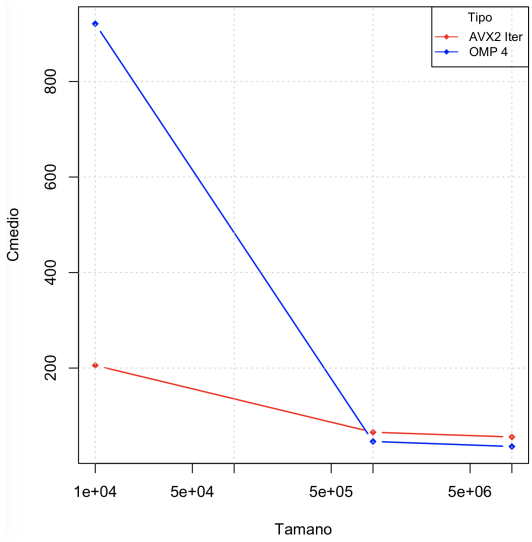

Figura 14: *Programa base (-O0) fronte a secuencial optimizado*

VIII. Conclusións

Na análise realizada quedou patente a importancia da optimización dos nosos programas para obter os mellores rendementos posibles e así reducir en ordes de magnitude os tempos de execución, o que resulta especialmente crítico en programas que deban executarse en estrictos marxes temporais.

Además, demostrouse que as optimizacións realizadas cun bo coñecemento do problema en cuestión permiten obter rendementos claramente superiores aos obtidos mediante as optimizacións automáticas realizadas polo compilador. Neste senso, convén sinalar que a eficacia destas ven determinada en gran medida pola estrutura dos nosos programas xa que, como se viu, o emprego dunha estrutura *Quaternion* limitou considerablemente as posibles optimizacións que este foi capaz de realizar.

En canto ao emprego da vectorización, as súas posibilidades de aplicación dependen en gran medida da natureza do problema. Así, unha vectorización por operacións (multiplicación e cadrado) só permitiu acadar rendementos comparables ao programa secuencial base. Pola contra, a vectorización por iteracións permitiu mellorar considerablemente o rendemento acadado polo programa secuencial optimizado.

Por outra parte, é convinte destacar que a paralelización mediante fíos solamente é apropiada para tamaños do problema moi grandes, de maneira que o sobrecusto asociado a creación, xestión e destrución dos fíos non se impoña fronte ao custo dos propios cálculos.

Finalmente, resultaría de interese realizar máis probas aumentando o espectro de tamaños probados, o que sería especialmente relevante para estudar o comportamento do programa paralelizado mediante fíos, para o cal non se obtiveron as melloras esperadas no caso de 2 fíos. A maiores, poderíase paralelizar mediante fíos o programa vectorizado por iteracións e estudar se isto permite rebaixar as cotas inferiores de ciclos medios acadadas.

Referencias

[1] Peter Cordes. Get sum of values stored in m256d with sse avx. `https://stackoverflow.com/questions/49941645/get-sum-of-values-stored-in-m256d\-with-sse-avx/`, Apr 2018.

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

[3] Gunther Piez. Intel avx: 256-bits version of dot product for double precision floating point variables. `https://stackoverflow.com/questions/10454150/intel-avx-256-bits-version-of-dot\-product-for-double\-precision-floating-point-v/`, May 2012.